An Analysis on the Wikipedia Data of Famous Individuals

Grace Jiang

Ivy Ho

Thank you to our professors Ada Lerner, Panagiotis Metaxas, and Jean Herbst for their patience and diligence in teaching and guiding us to our accomplishments this semester!

I. Introduction

In this study, we analyzed the given Wikipedia data which contained two different formats of data to extract from, one for edges and one for nodes. All the data was given in TSV (Tab Separated Values) files. The purpose of this project is to read in the Wikipedia profile data of the given people, create a graph from that data, and analyze the given information.

Each person represents a node in the graph, and a link from one person's page to another person's page is considered an arc, also known as a directed connection from one node to another. If there is an arc both ways between two people, that is considered an edge. The number of arcs a person has pointing towards other people is called their "out-degree". The number of arcs a person has pointing towards them is called their "in-degree". The "in-degree" value in this circumstance is a good indicator of a person's importance, as it shows the number of people who have referred to the said person as a link in their Wikipedia page. The node data also included several categories of information such as the individual's nationality, gender, and domain of work. To analyze the information and draw conclusions, we looked at the intersection between different subsections of the data to find evidence for the variety of different patterns and trends that might be occurring.

II. Methods

First, we created an implementation of the Graph interface which uses a Vector to hold the "nodes" of the graph and a Vector of LinkedLists where each LinkedList holds the successors of a specified node. This implementation includes basic methods that would pertain to creating a Graph object, such as methods to add and remove arcs and nodes. These basic methods would allow us to create and modify the graph as the Wikipedia data is read. It also includes three different graph traversal methods: breadth-first search and depth-first search, which both return a Vector of the specified type T values in the order of the traversal, as well as another breadth-first search method

that returns a Vector of LinkedLists which holds the paths from the starting node to every other node in the graph.

When we began planning our program, we were only looking at the file containing edges data. Considering that the file only contained Strings representing the people and the edges between them, we wanted to create a Vector of Triples, where each Triple would hold a node's name as a String and its in- and out-degrees as ints. We would have created a Triple class to hold information of multiple types. When we presented this idea to Professor Metaxas, he suggested that we create a Person class instead. This is because when we analyze the question about comparing properties by gender, we would need to access fields (e.g. gender, occupation) given in the node TSV files. Given this information, we decided to design our graph as a graph of Person objects, instead of a graph of Strings. The Person class contains instance variables for a person's en_curid (Wikipedia page ID number), name, birth city, birth state, country, gender, occupation, industry, domain, in-degree, and out-degree. All of these instance variables are mutable so that we can create a Person object that stores all the fields of each node and increments its in- and out-degrees while reading the TSV file of edges.

In the InvestigateWiki class, we are answering the following questions:

1. What are the minimum, maximum, and average in-degree and out-degree for these pages?
2. Which person in this graph seems to have the highest importance? How might you define "importance" in such a graph?
3. Which nodes are farthest away from the highest importance people in the graph?
4. What algorithm will you use to find minimum distances between nodes?
5. Compare the properties of the pages of men and women in the graph. How do they differ?

Considering that the size of the data is quite large, we wanted to minimize the number of times we need to iterate through the graph. To do this, we created the method iterateGraph(), which will iterate through the graph only once and invokes other methods to collect data that helps answer questions one, two, and five.

To easily access the minimums, maximums, and averages of the in- and out-degrees, we created two int arrays as instance variables of the class. One array for the minimum, maximum, and average number of in-degrees, respectively, and one for the out-degrees in the same order. The method iterateGraph() calls on the method setMinMaxAvg() which continuously updates the two arrays as it iterates through each Person.

Question two also requires iterating through the graph to determine the person with the highest in-degree. We solved this by creating an instance variable called mostImportantPerson that holds the current "most important" Person. The variable is then updated with the method setMinMaxAvg() when a Person with a greater in-degree than the previous most important Person. By the end of the graph iteration, the person with the highest in-degree should be stored in mostImportantPerson.

The third analysis question is to determine which people are the farthest away from the most important person in the graph. A BFS traversal would be helpful in this case because once a complete graph traversal list starting with the most important person is found, the people farthest away would be at the end of the list. Using this logic, we simply iterated the graph backwards to see where the first change in distance from the starting person occurred and then returned all values from that point to the end.

For the fourth analysis question, we looked to our other implementation of BFS, called BFSPath(), to find the minimum distances between nodes. If we consider the nature of the breadth-first search algorithm itself, it is a very effective option because it automatically searches for the shortest path to each node. Since BFSPath() is part of our graph implementation AdjListGraph, we invoked it through our getMinDist() method. This method takes a start and end node, uses BFSPath() to produce the Vector of paths from the start node to every other node, and loops through it in search of a path that ends with the specified end node. It then returns the length of that path, which is the minimum distance between the two given nodes.

In order to analyze the differences in properties between males and females, we needed access to data separated by gender. We specifically want to compare the

difference in fields of work, which can be characterized by the categories domain, industry, and occupation. We stored this data by creating two Vectors for male and female. In each vector, there are three Hashtables for domains, industries, and occupations, respectively. In each Hashtable, the keys are the different kinds of domains, industries, or occupations, and the values are the number of occurrences of each key. For example, there are 1,205 males in the domain of humanities. Domains is the Hashtable in the maleInfo Vector's index of 0. In that Hashtable, the key "HUMANITIES" has a value of 1,205. In iterateGraph(), we increment the values of each key according to the traits of the Person.

III. Conclusion

General Information about the In-Degrees and Out-Degrees of the Dataset			
	Minimum #	Maximum #	Average #
In-Degree	0	705	59
Out-Degree	0	472	59

In the table above, we have presented the information concerning the overall minimum, maximum, and average values concerning the in-degrees and out-degrees of the graph. Considering that the minimum value for both in-degrees and out-degrees is 0, it is reasonable to suspect that there may be a disconnected node in the graph if those two values are referring to the same Person object. Further looking into this question, we discovered that there is a disconnected node which represents Melvil Dewey, an inventor who also happens to be the least important person in this graph (as his in-degree is 0). On that note, the person with the maximum in-degree value can also be considered the most important person. In the case of this dataset, that person is Barack Obama with 705 links pointing to his page. More information on him and his place in the graph can be found in the table below.

Info about the Most Important Person	
Most Important Person	Barack Obama
# of In-Degrees	705
Farthest Away People (Path Length: 6 arcs)	Sunny Leone Jamal Baku Utamaro Tokyo Daniel Alves Robert Koren Rodrigo Taddei

Some Testing for Finding Minimum Distances Between Nodes		
Person From	Person To	Minimum Distance
Barack Obama	Abraham Lincoln	1
Barack Obama	Albert Einstein	2
Barack Obama	Barack Obama	0
Abraham Lincoln	J. K. Rowling	3
Abraham Lincoln	Albert Einstein	2
Albert Einstein	Abraham Lincoln	2

As for the data about people farthest away from a single person, our results are highly relevant to the Six Degrees of Separation Theory which hypothesizes that every living person in the world is only six or fewer “degrees”, or people, away from each other. In the table describing the most important person of the graph, Barack Obama, we can see that the farthest people from him are *only* ***six*** people, or arcs, away. This is surprising because, in a dataset of **11,314** people/nodes , we expected the distance to be greater. Our data aligns with and provides some small evidence for this famous theory. The minimum distance data also reflects this pattern as each of the sampled minimum distance values are also under six.

General Male Vs. Female Comparisons					
	Total	Male	% Male of Total	Female	% Female of Total
# of People	11,314	9,825	86.8	1,489	13.2
# of Domain Types	8	8	100	8	100
# of Industry Types	27	27	100	25	92.6
# of Occupation Types	88	86	97.7	56	63.6

Looking at some general data values in terms of gender, we can see that the overall Wikipedia data has an extreme overrepresentation of males. They make up 86.8% of the total person data, while females only make up a scant 13.2%. To make sense of the data, we must look at what domain, industry, and occupation represent. They are all related to the fields in which a person works, but vary in their specificity. Domain is the least specific, with types like Institutions, Arts, and Science & Technology. Industry is more specific, with types such as Government, Medicine, and Philosophy. Occupation is the most specific category with types like Magician, Social Activist, and Chemist. It is also important to note that as the categories get more specific, they also broaden out with more types. We can see a slight downward trend with this data in that as the information gets more specific, the more visible the disparity between gender. As can be seen above, women have much less variety in terms of types of occupation, only 63.6% of the total types, compared to 97.7% for men. This close analysis of the data reveals the importance of disaggregation in data science. Had we only analyzed the domains, we would not have been able to see this pattern. Despite possible issues of efficiency, an effective analysis must look closely at both what the data is and is not showing to uncover meaningful conclusions. Otherwise, important issues of access and opportunity, such as those shown by occupation data for females, are at risk.

Popularity of Fields of Work for Males				
	Most Popular Kind(s)	# of People in Each Kind	Least Popular Kind(s)	# of People in Each Kind
Domains	Institutions	3,237	Exploration	94
Industries	Government	2,523	Dance	5
Occupations	Politician	2,393	American Football Player Chef	1

Popularity of Fields of Work for Females				
	Most Popular Kind(s)	# of People in Each Kind	Least Popular Kind(s)	# of People in Each Kind
Domains	Arts	791	Business & Law	5
Industries	Film & Theatre	503	Law Computer Science Engineering	1
Occupations	Actor	500	Military Personnel Computer Scientist Architect Wrestler Chessmaster Lawyer Archaeologist Photographer Basketball Player Judge Chef Engineer Racecar Driver	1

Percentages of Males and Females in Each Domain Type		
Domains	% of Males Out of Total Males	% of Females Out of Total Females
Institutions	3,237/9,825 = 32.9	212/1,489 = 14.2

Humanities	12.3	8.1
Arts	21.0	53.1
Public Figure	1.7	12.8
Science & Technology	13.5	2.9
Exploration	1.0	0.5
Sports	16.6	8.1
Business & Law	1.0	0.3

Looking further into the data, we can see some patterns among what fields of work are popular and not popular for males and females. The data seems to align with common gender norms in American society. For example, the most popular kind of occupation for males in institutions and government is being a politician. In this dataset, 32.9% of the total number of males are in the domain “Institutions,” and of that percentage, 2,393 male politicians are listed. This coincides with reality, as most politicians are male, with only a small, but growing number of female politicians. The least popular industry for men is Dance, with only five people, and one of the least popular occupations is being Chef, with only one of each. Working in Dance and being a Chef are comparatively artistic professions.

Meanwhile, the most popular categories for women are the Arts, Film and Theatre, and professional Acting. If we look at the table above, the percentages of males and females that are in specific domains show the same trend: there are only 21.0% of males in the arts as compared to 53.1% of females. This data reflects how the arts are seen as more feminine fields of work in modern society and therefore, are more populated by women than men. One more point to notice is that in the domain of Science and Technology as well as Business and Law, very few women seem to participate in such fields, with the percentage of women at 2.9% and 0.3% respectively. The graphs above also indicate that Law, Computer Science, and Engineering are the least popular industries for females. This aligns with the past knowledge that females are underrepresented in the STEM field, but because of how rapidly the world has changed in the last 10-20 years, we would not be surprised that this dataset is not representative of the

popularity of these industries for women today. However, this data could very well be representative of women in these fields who have their own Wikipedia page -- very few.

If the given dataset is representative of Wikipedia's complete data in terms of gender, then we can conclude that there are just simply less females who have their own Wikipedia pages as compared to males. This could be because most Wikipedia contributors are male, and people might be more likely to write about someone of their own gender because it is more relevant to them. This could also be because much of Wikipedia's data is historically-based. Because of how limited women were in terms of occupation, they had much less opportunity to expand in most fields of work. In addition, many noteworthy women of the past were not given credit for their work. As we see the world continue to change, females will hopefully become more represented on Wikipedia.

IV. Code

AdjListsGraph.java

```
/**
 * Creates a AdjListsGraph object, a.k.a. a graph object, as well as contains
 * related methods to extract information from a graph.
 *
 * @author Ivy Ho and Grace Jiang
 * @version Saturday, May 9, 2020
 */

import javafoundations.*;
import java.io.*;
import java.util.Vector;
import java.util.LinkedList;
```

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

   /**
    * Constructor for objects of class AdjListsGraph
```

```
 */
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 vertices, and of course, no edges.
 *
 *  @return true if this graph is empty, false otherwise.
 */
public boolean isEmpty() {
   return vertices.isEmpty();
}

/**
 * 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 numArcs = 0;
   for (int i = 0; i < vertices.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;
   }
   int index = vertices.indexOf(vertex1);
   return arcs.get(index).contains(vertex2);
}

/**
 * Returns true if an edge exists between two given vertices, i.e,
 * an arc 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 < vertices.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);
      vertices.remove(index);
      arcs.remove(index);
      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 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 index = vertices.indexOf(vertex1);
      arcs.get(index).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)) {
      int index = vertices.indexOf(vertex1);
      arcs.get(index).remove(vertex2);
   }
}

/**
 * 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) {
   if (vertices.contains(vertex1) && vertices.contains(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 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) {
   if (vertices.contains(vertex1) && vertices.contains(vertex2)) {
      removeArc(vertex1, vertex2);
      removeArc(vertex2, vertex1);
   }
}
```

```
/**
 * 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)) {
      int index = vertices.indexOf(vertex);
      return arcs.get(index);
   }

   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> predecessors = new LinkedList<T>();

   for (int i = 0; i < arcs.size(); i++) {
      if (arcs.get(i).contains(vertex))
         predecessors.add(vertices.get(i));
   }

   return predecessors;
}

/**
 * Returns a string representation of this graph.
 *
 * @return a string representation of this graph, containing its vertices
 * and its arcs/edges
 */
public String toString() {
   String info = "";
```

```
    info += "Vertices: " + "\n[";

    for (int i = 0; i < vertices.size(); i++) {
        info += vertices.get(i);
        if (i < vertices.size() - 1) {
            info += ", ";
        }
    }

    info += "]\n" + "Edges: " + "\n";

    for (int i = 0; i < vertices.size(); i++) {
        info += "from " + vertices.get(i) + ": [";
        for (int j = 0; j < arcs.get(i).size(); j++) {
            info += arcs.get(i).get(j);
            if (j < arcs.get(i).size() - 1) {
                info += ", ";
            }
        }
        info += "]\n";
    }

    return info;
}

/**
 * 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 filename) {
    try {
        PrintWriter writer = new PrintWriter(new File(filename));

        for (int i = 0; i < vertices.size(); i++) {
            writer.println(i + " " + vertices.get(i) + "\n");
        }

        writer.println("#\n");

        for (int i = 0; i < arcs.size(); i++) {
            for (int j = 0; j < arcs.get(i).size(); j++) {
                writer.println(i + " " + vertices.indexOf(arcs.get(i).get(j)) + "\n");
```

```
            }
        }

        writer.close();
    }
    catch (IOException e) {
        System.out.println (e);
    }
}

/**
 * Returns all the vertices x, in this graph, in the order of the
 * breadth-first search traversal from the starting vertex
 *
 * @param a vertex from which the traversal will begin
 * @return a LinkedQueue containing all the vertices x in the graph in
 * the order of the breadth-first search traversal
 */
public Vector<T> BFSTraversal(T vertex) {
    if (!vertices.contains(vertex)) {
        return null;
    }
    LinkedQueue<T> traverser = new LinkedQueue<T>();
    Vector<T> visited = new Vector<T>();
    T currentVertex;

    visited.add(vertex);
    traverser.enqueue(vertex);

    while (!traverser.isEmpty()) {
        currentVertex = traverser.dequeue();
        for (T nextVertex: getSuccessors(currentVertex)) {
            if (!visited.contains(nextVertex)) {
                visited.add(nextVertex);
                traverser.enqueue(nextVertex);
            }
        }
    }
    return visited;
}

/**
 * Returns all the paths in this graph in the order of the
 * breadth-first search traversal from the starting vertex
```

```
 *
 * @param a vertex from which the traversal will begin
 * @return a Vector of LinkedLists containing all the paths
 */
public Vector<LinkedList<T>> BFSPaths(T start) {
   if (!vertices.contains(start)) {
      return null;
   }

   Vector<LinkedList<T>> result = new Vector<LinkedList<T>>();
   int[] visited = new int[vertices.size()];
   LinkedQueue<LinkedList<T>> q = new LinkedQueue<LinkedList<T>>();

   visited[vertices.indexOf(start)] = 1;
   LinkedList<T> startResult = new LinkedList<T>();
   startResult.add(start);
   result.add(startResult);

   LinkedList<T> startPath = new LinkedList<T>();
   startPath.add(start);
   q.enqueue(startPath);

   while (!q.isEmpty()) {
      LinkedList<T> currentPath = q.dequeue();
      for (T nextVertex: getSuccessors(currentPath.get(currentPath.size() - 1))) {
         if (visited[vertices.indexOf(nextVertex)] != 1) {
            visited[vertices.indexOf(nextVertex)] = 1;

            LinkedList<T> nextPath = new LinkedList<T>();
            nextPath.addAll(currentPath);
            nextPath.add(nextVertex);
            q.enqueue(nextPath);
            result.add(nextPath);
         }
      }
   }

   return result;
}

/**
 * Returns all the vertices x, in this graph, in the order of the
 * breadth-first search traversal from the starting vertex
 *
```

```
 * @param a vertex from which the traversal will begin
 * @return a ArrayStack containing all the vertices x in the graph in
 * the order of the breadth-first search traversal
 */
public Vector<T> DFSTraversal(T start) {
   if (!vertices.contains(start)) {
      return null;
   }
   Vector<T> result = new Vector<T>();
   int[] visited = new int[vertices.size()];
   ArrayStack<T> stack = new ArrayStack<T>();

   T top;
   LinkedList<T> next;
   T unvisited;

   stack.push(start);
   result.add(start);
   visited[vertices.indexOf(start)] = 1;

   while (!stack.isEmpty()) {
      top = stack.peek();
      next = arcs.get(vertices.indexOf(top));
      unvisited = null;
      for (int i = 0; i < next.size(); i++) { // determines if there are any more unvisited nodes
         T current = next.get(i);
         if (visited[vertices.indexOf(current)] != 1) {
            stack.push(current);
            result.add(current);
            visited[vertices.indexOf(current)] = 1;
            unvisited = current;
         }
         if (unvisited != null) {
            break;
         }
      }

      if (unvisited != null) {

      } else {
         stack.pop();
      }
   }
```

```
        return result;
    }
}
```

Person.java

```
/**
 * Creates a Person object
 *
 * @author Grace Jiang and Ivy Ho
 * @version 5/11/2020
 */
public class Person {
    private int en_curid;
    private String name;
    private String birthcity;
    private String birthstate;
    private String countryName;
    private String gender;
    private String occupation;
    private String industry;
    private String domain;
    private int numInDegrees;
    private int numOutDegrees;

    /**
     * Constructor for objects of class Person
     */
    public Person(String newName) {
        name = newName;
        birthcity = "";
        birthstate = "";
        countryName = "";
        gender = "";
        occupation = "";
        industry = "";
        domain = "";
        numInDegrees = 0;
        numOutDegrees = 0;
    }
```

```
/**
 * Sets the en_curid of this Person.
 *
 * @param new String value of en_curid
 */
public void setEnCurid(int newID) {
   en_curid = newID;
}

/**
 * Gets the en_curid of this Person.
 *
 * @return String value of en_curid
 */
public int getEnCurid() {
   return en_curid;
}

/**
 * Sets the name of this Person.
 *
 * @param new String value of name
 */
public void setName(String newName) {
   name = newName;
}

/**
 * Gets the name of this Person.
 *
 * @return String value of name
 */
public String getName() {
   return name;
}

/**
 * Sets the birthcity of this Person.
 *
 * @param new String value of birthcity
 */
public void setBirthcity(String newCity) {
   birthcity = newCity;
```

```
}

/**
 * Gets the birthcity of this Person.
 *
 * @return String value of birthcity
 */
public String getBirthcity() {
   return birthcity;
}

/**
 * Sets the birthstate of this Person.
 *
 * @param new String value of birthstate
 */
public void setBirthstate(String newBirthstate) {
   birthstate = newBirthstate;
}

/**
 * Gets the birthstate of this Person.
 *
 * @return String value of birthstate
 */
public String getBirthstate() {
   return birthstate;
}

/**
 * Sets the countryName of this Person.
 *
 * @param new String value of countryName
 */
public void setCountryName(String newCountry) {
   countryName = newCountry;
}

/**
 * Gets the countryName of this Person.
 *
 * @return String value of countryName
 */
public String getCountryName() {
```

```
    return countryName;
}

/**
 * Sets the gender of this Person.
 *
 * @param new String value of gender
 */
public void setGender(String newGender) {
    gender = newGender;
}

/**
 * Gets the gender of this Person.
 *
 * @return String value of gender
 */
public String getGender() {
    return gender;
}

/**
 * Getter for occupation
 *
 * @return String value of occupation
 */
public String getOccupation() {
    return occupation;
}

/**
 * Setter for occupation
 *
 * @param new String value of occupation
 */
public void setOccupation(String o) {
    occupation = o;
}

/**
 * Getter for industry
 *
 * @return String value of industry
 */
```

```
public String getIndustry() {
   return industry;
}

/**
 * Setter for industry
 *
 * @param new String value of industry
 */
public void setIndustry(String i) {
   industry = i;
}

/**
 * Getter for domain
 *
 * @return String value of domain
 */
public String getDomain() {
   return domain;
}

/**
 * Setter for domain
 *
 * @param new String value of domain
 */
public void setDomain(String d) {
   domain = d;
}

/**
 * Getter for number of in-degrees
 *
 * @return int value of in-degrees
 */
public int getInDegs() {
   return numInDegrees;
}

/**
 * Setter for number of in-degrees
 *
 * @param new int value of in-degrees
```

```
 */
public void setInDegs(int degs) {
   numInDegrees = degs;
}

/**
 * Getter for number of out-degrees
 *
 * @return int value of out-degrees
 */
public int getOutDegs() {
   return numOutDegrees;
}

/**
 * Setter for number of out-degrees
 *
 * @param new int value of out-degrees
 */
public void setOutDegs(int degs) {
   numOutDegrees = degs;
}

/**
 * Increments number of out-degrees by 1
 */
public void incrementOutDegs() {
   numOutDegrees++;
}

/**
 * Increments number of in-degrees by 1
 */
public void incrementInDegs() {
   numInDegrees++;
}

/**
 * Returns a neat String to describe this Person.
 *
 *@return a String
 */
public String toString() {
   String info = en_curid + "\t" + name + "\t" + birthcity + "\t"
```

```
            + birthstate + "\t" + countryName + "\t" + gender + "\t"
            + occupation + "\t" + industry + "\t" + domain + "\t"
            + numInDegrees + "\t" + numOutDegrees;
        return info;
    }
}
```

InvestigateWiki.java

```
/**
 * The class InvestigateWiki takes and analyzes real wikipedia data.
 * Methods will:
 *  - Read TSV (tab-separated values) files and create an AdjListsGraph Object
 *  - Determine minimum, maximum, and average in-degree and out-degree for Wiki pages
 *  - Determine the person of highest importance in the graph
 *  - Determine which nodes are farthest away from the highest importance persson
 *  - Determine the minimum distance from one node to another
 *  - Analyze the differences in domains, industries, and occupations by gender
 *
 * @author Grace Jiang and Ivy Ho
 * @version Sunday, May 10, 2020
 */

import java.util.Vector;
import java.util.LinkedList;
import java.util.Scanner;
import java.util.Hashtable;
import java.util.Set;
import java.util.ArrayList;
import java.util.Collections;
import java.io.File;
import java.io.PrintWriter;
import java.io.FileNotFoundException;
import java.io.IOException;
```

```
public class InvestigateWiki
{
    private AdjListsGraph<Person> graph;
    private Hashtable<String, Person> hash;
```

```
private int[] inDegCalcs;
private int[] outDegCalcs;
private Person mostImportantPerson;
private Person leastImportantPerson;

private int numMale;
private int numFemale;
private Vector<Hashtable<String, Integer>> maleInfo;
private Vector<Hashtable<String, Integer>> femaleInfo;

int longest;
Vector<LinkedList<Person>> paths;
/**
 * Constructor for objects of class InvestigateWiki
 */
public InvestigateWiki() {
    graph = new AdjListsGraph<Person>();
    hash = new Hashtable<String, Person>(12000);
    inDegCalcs = new int[] {-1,-1,-1}; // min, max, and avg values
    outDegCalcs = new int[] {-1,-1,-1};

    Hashtable mDomain = new Hashtable<String, Integer>(12);
    Hashtable mIndustry = new Hashtable<String, Integer>(35);
    Hashtable mOccupation = new Hashtable<String, Integer>(95);
    maleInfo = new Vector(3);
    maleInfo.add(mDomain);
    maleInfo.add(mIndustry);
    maleInfo.add(mOccupation);

    Hashtable fDomain = new Hashtable<String, Integer>(12);
    Hashtable fIndustry = new Hashtable<String, Integer>(35);
    Hashtable fOccupation = new Hashtable<String, Integer>(95);
    femaleInfo = new Vector(3);
    femaleInfo.add(fDomain);
    femaleInfo.add(fIndustry);
    femaleInfo.add(fOccupation);
}

/**
 * Reads a CSV file of nodes and creates a new Person object for each node.
 * Then adds each Person object to the graph and to the hash table of people.
 *
 * @param CSV file
 */
```

```
private void readNodes(String nodeFile) {
   try {
      Scanner scan = new Scanner(new File(nodeFile));
      scan.nextLine(); //read past the header

      while (scan.hasNextLine()) {
         String entry = scan.nextLine();
         String[] data = entry.split("\t");

         Person newPerson = new Person(data[1]);

         newPerson.setEnCurid(new Integer(data[0]));
         newPerson.setBirthcity(data[2]);
         newPerson.setBirthstate(data[3]);
         newPerson.setCountryName(data[4]);
         newPerson.setGender(data[5]);
         newPerson.setOccupation(data[6]);
         newPerson.setIndustry(data[7]);
         newPerson.setDomain(data[8]);

         hash.put(data[1], newPerson);
         graph.addVertex(newPerson);
      }
   }
   catch (FileNotFoundException e) {//something went wrong reading from the file
      System.out.println (e);
      System.out.println ("The program terminates.");
      System.exit(0); //exit the whole program.

   }
}

/**
 * Reads the CSV file of edges and adds arcs to the graph. Method should
 * only be executed after readNodes has been completed.
 *
 * @param CSV file
 */
private void readArcs(String edgeFile) {
   try {
      Scanner scan = new Scanner(new File(edgeFile));
      scan.nextLine(); //read past the header

      while (scan.hasNextLine()) {
```

```
            String entry = scan.nextLine();
            String[] data = entry.split("\t");

            Person from = hash.get(data[0]);
            Person to = hash.get(data[1]);

            graph.addArc(from, to);
            from.incrementOutDegs();
            to.incrementInDegs();
        }
    }
    catch (FileNotFoundException e) {//something went wrong reading from the file
        System.out.println (e);
        System.out.println ("The program terminates.");
        System.exit(0); //exit the whole program.

    }
}

/**
 * Loops through the data and performs various methods to analyze the
 * data by gender.
 */
private void iterateGraph() {
    for (Person currentPerson : graph.vertices)
    {
        setMinMaxAvg(currentPerson);
        if (currentPerson.getGender().equals("Male")) {
            numMale++;
            incrementMDomain(currentPerson);
            incrementMIndustry(currentPerson);
            incrementMOccupation(currentPerson);
        } else {
            numFemale++;
            incrementFDomain(currentPerson);
            incrementFIndustry(currentPerson);
            incrementFOccupation(currentPerson);
        }

    }
    inDegCalcs[2] = inDegCalcs[2]/graph.vertices.size();
    outDegCalcs[2] = outDegCalcs[2]/graph.vertices.size();
}
```

```
/**
 * Updates the hash table of male information for a specific domain.
 * If the person's domain isn't already in the hash table, the
 * domain will be added as a key. If the domain is already a key,
 * the value will be incremented by 1. Function is called in iterateGraph().
 *
 * @param Person object
 */
private void incrementMDomain(Person current) {
   if (!maleInfo.get(0).containsKey(current.getDomain())) {
      maleInfo.get(0).put(current.getDomain(), 1);
   } else {
      Integer count = maleInfo.get(0).get(current.getDomain()).intValue() + 1;
      maleInfo.get(0).put(current.getDomain(), count);
   }
}

/**
 * Updates the hash table of male information for a specific industry.
 * If the person's industry isn't already in the hash table, the
 * industry will be added as a key. If the industry is already a key,
 * the value will be incremented by 1. Function is called in iterateGraph().
 *
 * @param Person object
 */
private void incrementMIndustry(Person current) {
   if (!maleInfo.get(0).containsKey(current.getIndustry())) {
      maleInfo.get(0).put(current.getIndustry(), 1);
   } else {
      Integer count = maleInfo.get(0).get(current.getIndustry()).intValue() + 1;
      maleInfo.get(0).put(current.getIndustry(), count);
   }
}

/**
 * Updates the hash table of male information for a specific occupation.
 * If the person's occupation isn't already in the hash table, the
 * occupation will be added as a key. If the occupation is already a key,
 * the value will be incremented by 1. Function is called in iterateGraph().
 *
 * @param Person object
 */
private void incrementMOccupation(Person current) {
   if (!maleInfo.get(0).containsKey(current.getOccupation())) {
```

```
            maleInfo.get(0).put(current.getOccupation(), 1);
        } else {
            Integer count = maleInfo.get(0).get(current.getOccupation()).intValue() + 1;
            maleInfo.get(0).put(current.getOccupation(), count);
        }
    }

    /**
     * Updates the hash table of female information for a specific domain.
     * If the person's domain isn't already in the hash table, the
     * domain will be added as a key. If the domain is already a key,
     * the value will be incremented by 1. Function is called in iterateGraph().
     *
     * @param Person object
     */
    private void incrementFDomain(Person current) {
        if (!femaleInfo.get(0).containsKey(current.getDomain())) {
            femaleInfo.get(0).put(current.getDomain(), 1);
        } else {
            Integer count = femaleInfo.get(0).get(current.getDomain()).intValue() + 1;
            femaleInfo.get(0).put(current.getDomain(), count);
        }
    }

    /**
     * Updates the hash table of female information for a specific industry.
     * If the person's industry isn't already in the hash table, the
     * industry will be added as a key. If the industry is already a key,
     * the value will be incremented by 1. Function is called in iterateGraph().
     *
     * @param Person object
     */
    private void incrementFIndustry(Person current) {
        if (!femaleInfo.get(1).containsKey(current.getIndustry())) {
            femaleInfo.get(1).put(current.getIndustry(), 1);
        } else {
            Integer count = femaleInfo.get(1).get(current.getIndustry()).intValue() + 1;
            femaleInfo.get(1).put(current.getIndustry(), count);
        }
    }

    /**
     * Updates the hash table of female information for a specific occupation.
     * If the person's occupation isn't already in the hash table, the
```

```
 * occupation will be added as a key. If the occupation is already a key,
 * the value will be incremented by 1. Function is called in iterateGraph().
 *
 * @param Person object
 */
private void incrementFOccupation(Person current) {
   if (!femaleInfo.get(2).containsKey(current.getOccupation())) {
      femaleInfo.get(2).put(current.getOccupation(), 1);
   } else {
      Integer count = femaleInfo.get(2).get(current.getOccupation()).intValue() + 1;
      femaleInfo.get(2).put(current.getOccupation(), count);
   }
}

/**
 * Sets the minimum, maximum, and average values for both the in-degree
 * and out-degree values of the current person
 *
 * @param Person object
 */
private void setMinMaxAvg(Person current) {
   int inDegs = current.getInDegs();
   int outDegs = current.getOutDegs();
   if (inDegCalcs[0] == -1) //fix
   {
      inDegCalcs[0] = inDegs;
      inDegCalcs[1] = inDegs;
      inDegCalcs[2] = inDegs;
   }
   else
   {
      if (inDegs < inDegCalcs[0])
      {
         inDegCalcs[0] = inDegs;
         leastImportantPerson = current;
      }
      if (inDegs > inDegCalcs[1])
      {
         inDegCalcs[1] = inDegs;
         mostImportantPerson = current;
      }
      inDegCalcs[2] += inDegs;
   }
```

```
    if (outDegCalcs[0] == -1)
    {
       outDegCalcs[0] = current.getOutDegs();
       outDegCalcs[1] = current.getOutDegs();
       outDegCalcs[2] = current.getOutDegs();
    }
    else
    {
       if (outDegs < outDegCalcs[0])
       {
          outDegCalcs[0] = outDegs;
       }
       if (outDegs > outDegCalcs[1])
       {
          outDegCalcs[1] = outDegs;
       }
       outDegCalcs[2] += outDegs;
    }
 }

 /**
  * Finds the people farthest from the given person in this graph.
  *
  * @param a Person object
  * @return a Vector of Person objects
  */
 private Vector<Person> getFarthest(Person start) {
    Vector<LinkedList<Person>> paths = new Vector<LinkedList<Person>>();
    paths = graph.BFSPaths(start);
    int longest = 0;

    Vector<Person> farthestPeople = new Vector<Person>();
    for (int i = paths.size() - 1; i > 0; i--) {
       LinkedList<Person> currentPath = paths.get(i);
       if (farthestPeople.size() == 0) {
          farthestPeople.add(currentPath.getLast());
          longest = currentPath.size();
       } else {
          if (currentPath.size() == longest) {
             farthestPeople.add(currentPath.getLast());
          } else {
             break;
          }
       }
```

```
        }

        return farthestPeople;
    }

    /**
     * Gets the minimum distance (number of edges) from a given Person to
     * the other given Person.
     *
     * @param Person object to start from
     * @param Person object to end at
     * @return int value of path size - 1 to get number of edges between nodes
     */
    private int getMinDist(Person start, Person end) {
        Vector<LinkedList<Person>> paths = new Vector<LinkedList<Person>>();
        paths = graph.BFSPaths(start);

        LinkedList<Person> path = new LinkedList<Person>();
        for (int i = 0; i < paths.size(); i++) {
            if (paths.get(i).getLast().equals(end)) {
                path = paths.get(i);
            }
        }
        int size = path.size() - 1;
        return size;
    }
/**
     * Presents Person Object data found in the maleInfo and femaleInfo
     * Vectors that are related to the given infoCategory and neatly
     * sorted by male and female.
     *
     * @param a String for the name of the category to find data in
     * @return a String with all the relevant data by gender
     */
    public String presentDataByGender(String infoCategory) {
        String info = "";
        int index;
        if (infoCategory.equalsIgnoreCase("domain")) {
            index = 0;
        } else if (infoCategory.equalsIgnoreCase("industry")) {
            index = 1;
        } else if (infoCategory.equalsIgnoreCase("occupation")) {
            index = 2;
        } else {
```

```
    return "Given data category not found. Try again.";
}

Hashtable categoryM = maleInfo.get(index);
info += "There are " + categoryM.size() + " different " + infoCategory
      + " types among the males in this graph.\n";
Set<String> categoryKeys = categoryM.keySet();
for (String key : categoryKeys){
   info += "Number of males in " + key + ": " + categoryM.get(key);
}
info += "The most popular are:\n";
ArrayList<String> popularM = mostPopular(categoryM);
for (String key: popularM) {
   info += key + "\t";
}
info += "\n";
info += "The least popular are:\n";
ArrayList<String> notPopularM = leastPopular(categoryM);
for (String key: notPopularM) {
   info += key + "\t";
}
info += "\n";

Hashtable categoryF = femaleInfo.get(index);
info += "There are " + categoryF.size() + " different " + infoCategory
       + " types among the females in this graph.\n";
categoryKeys = categoryF.keySet();
for (String key : categoryKeys){
   info += "Number of females in " + key + ": " + categoryF.get(key);
}
info += "The most popular are:\n";
ArrayList<String> popularF = mostPopular(categoryF);
for (String key: popularF) {
   info += key + "\t";
}
info += "\n";
info += "The least popular are:\n";
ArrayList<String> notPopularF = leastPopular(categoryF);
for (String key: notPopularF) {
   info += key + "\t";
}
info += "\n";

return info;
```

```
    }
/**
     * Creates a neat String to describe some important information
     * involving this Wikipedia Graph object.
     *
     * @return String of information
     */
    public String toString() {
        String info = "";
        info += "Number of people (vertices): " + graph.getNumVertices() + "\n";
        info += "Number of connections (arcs): " + graph.getNumArcs() + "\n";
        info += "\n";
        info += "Number of males: " + numMale + "\n";
        info += "Number of females: " + numFemale + "\n";
        info += "\n";
        info += "Minimum in-degree of the graph: " + inDegCalcs[0] + "\n";
        info += "Maximum in-degree of the graph: " + inDegCalcs[1] + "\n";
        info += "Average in-degree of the graph: " + inDegCalcs[2] + "\n";
        info += "Minimum out-degree of the graph: " + outDegCalcs[0] + "\n";
        info += "Maximum out-degree of the graph: " + outDegCalcs[1] + "\n";
        info += "Average out-degree of the graph: " + outDegCalcs[2] + "\n";
        info += "\n";
        info += "The most important person in this graph (as in who has the most in-degrees) is: "
                + mostImportantPerson.getName() + "\n";
        info += "\t" + mostImportantPerson.getName() + " has " +
        mostImportantPerson.getInDegs() + " in-degrees and " +
        mostImportantPerson.getOutDegs() + " out-degrees.\n";
        info += "\t" + mostImportantPerson.getName() + "'s Info: " + mostImportantPerson
                + "\n";
        info += "\tThe people farthest from " + mostImportantPerson.getName()
                + " in this graph are:\n";
        for (Person currentPerson : getFarthest(mostImportantPerson)) {
            info += "\t\t" + currentPerson + "\n";
        }
        info += "\tThe minimum distance from " + mostImportantPerson.getName()
                + " to these farthest people is: "
                + getMinDist(hash.get("Barack Obama"), hash.get("Sunny Leone"))  + " people\n";
        info += "\n";
        info += "The least important person in this graph (as in who has the least in-degrees) is: "
                + leastImportantPerson.getName() + "\n";
        info += "\t" + leastImportantPerson.getName() + " has "
                + leastImportantPerson.getInDegs() + " in-degrees and "
                + leastImportantPerson.getOutDegs() + " out-degrees.\n";
        info += "\t" + leastImportantPerson.getName() + "'s Info: " + leastImportantPerson + "\n";
```

```
        info += "\tThe number of people farthest from " + leastImportantPerson.getName()
                + " is: " + getFarthest(leastImportantPerson).size() + "\n";

        return info;
    }
}
```

V. Collaboration

For this project, we completed the majority of the work together by sharing our screen through Zoom and taking turns typing. Before writing the code, we discussed and deliberated over different methods on how to efficiently and elegantly execute our code. Discussion was often longer than actual coding because we tried to take into consideration the optimization of speed and memory usage, too.

When we began the project, we both had almost complete implementations of AdjListsGraph.java from lab. Together, we wrote the BFSPaths() and DFSTraversal() methods. We also wrote the Person class together. As for InvestigateWiki.java, everything was done together except that Ivy solved the incrementing issue with the gender analysis methods that begin with “increment,” and Grace wrote the methods getFarthest() and getMinDist(). All the final and neatly printed main method executions were written by Ivy.