Group 8 : CCDSALG Slayers
MCO2 Documentation

Members :
Cansino, Joehanna Llarenas
Dy, Harmony Claire Cheng
Uy, Justine Nicole Chya

I. Introduction

In this project, we will use a real-world dataset gathered from Facebook, specifically the one that Traud, Mucha, and Porter presented in 2005. This dataset includes the complete Facebook networks from a chosen group of colleges and universities. The networks were captured as a snapshot on a single day in September 2005. The focus of this project will be the analysis of the structure of the social networks within the chosen colleges and universities.

Outline:

1. **Dataset Overview**
    a. This dataset includes Facebook networks of a few chosen American colleges and universities.
    b. Each network represents each college or university, and it consists of anonymous accounts.
    c. Relationships between users are represented as friendships, which forms a social graph.
    d. The social graph can be viewed as a graph. Where nodes represent the accounts and the edges represent friendships.
    e. Accounts are defined by unique integer identifiers instead of actual names.
2. **Project Scope**
    a. Analysis of social Facebook networks from the chosen American colleges and universities.
    b. This will be able to handle networks with up to 5000 accounts.
3. **Data Format**
    a. Two integers, n and e, which stand for the quantity of accounts and friendships, respectively, are present at the beginning of each network data file.
    b. Next, there are e lines in the file, each of which represents a friendship between two accounts.
    c. Each line has two integers, a and b, which signify that account a and account b are friends.
4. **Project Tasks**
    a. Implement a function to read network data from the supplied files. This is known as data loading. Create code to show the social network for a chosen institution or university using network visualization.

b. Fundamental Network Analysis. Determine the network's nodes (accounts) and edges (friendships). Find out how many friendships are typical for each account. The maximum and lowest degrees (number of friendships) of nodes should be noted.
c. Analysis of Connectivity. Verify the social graph's connectivity and identify any isolated parts. Determine the greatest linked component's size.
d. Results visualization: Construct visualizations to effectively communicate the analysis' findings.

5. Conclusion

a. Write a summary of the conclusions and revelations drawn from the data structures and algorithms used.
b. Consider the results' importance and potential uses for the analysis.

II. Description of the data structure used in representing the social graph data, include rationale behind the use of such data structure/s. Show code if necessary.

```
public class Main {

    2 usages
    private static Map<Integer, List<Integer>> socialGraph = new HashMap<>();
    1 usage
    private static void loadData(String file) throws IOException {
        System.out.println("\nLoading data...");
        try (BufferedReader bufferedReader = new BufferedReader (new FileReader(file))) {
            String[] data = bufferedReader.readLine().trim().split( regex: "\\s+");
            int n = Integer.valueOf(data[0]); //number of accounts or the edges
            int e = Integer.valueOf(data[1]); //number of friendships or the vertices

            //initialize adjacency list
            for(int i = 0; i < n; i++) {
                socialGraph.put(i, new ArrayList<>());
            }
```

The main data structure that we used to represent the social graph data is the adjacency list which is implemented with the help of the following data structures:

1. **Map<Integer, List<Integer>> socialGraph:** This code is the adjacency list implementation of the social graph. It uses a HashMap with List<Integer> values that represent each user's friends and integer keys that represent user IDs. By utilizing their ID, users may quickly find their friends on the Map, and each user can store several friend connections on the List.
2. **Map:** allows us to look-up for the user's friends using their ID.
3. **List:** allows us to store the multiple friend connections for each user.
4. **Hashmap:** allows us to store and retrieve the adjacency list of the social graph.
5. **Arraylist:** allows us to store the list of friends for each user in the social graph.

An adjacency list consists of associating each vertex (node) with a list of its neighboring vertices to represent a graph. In this instance, the graph is one from a social network, where each vertex denotes a user and the edges signify user friendships.

Rationale:

1. **Space efficiency:** The number of friendships (edges) in social networks is often significantly smaller than the number of users (vertices). We simply keep the friendship information in an adjacency list, which is memory-efficient for sparse graphs.
2. **Fast access to friends:** Given a user ID, we can quickly retrieve their list of friends using the adjacency list by navigating directly to the matching list on the map. When working with big networks, where effective data retrieval is essential for performance, this is especially significant.
3. **Flexible in adding and removing edges:** By adding the friend's ID to the list of an existing user, the adjacency list makes it simple to add new friendships (edges). By removing the corresponding friend ID from the list, it is equally simple to end a friendship.
4. **Bi-directional relationships:** By adding the friendship connection in each direction to the adjacency list, the implementation in this code is designed to provide bi-directional relationships (undirected graph), meaning that if User A is friends with User B, User B will also be friends with User A.

III. Description of algorithms used to display friend list and connection between two people in the network

A. Display Friend List Algorithm

The ‘displayFriendList’ function is the algorithm used to display a friend list of a given user in a social graph.

```
private static void displayFriendList(String file) throws IOException {

    Scanner sc = new Scanner(System.in);

    System.out.print(s:"Enter ID Number: ");
    int ID = sc.nextInt();
```

‘displayFriendList’ Function Part 1

As shown in the image above, the social graph data is loaded using the file path, which is a parameter that is taken as an input. The user is then asked to enter an ID number of a person, after which the function searches for the friend list for that user.

```
        try (BufferedReader bufferedReader = new BufferedReader(new FileReader(file))) {
            ArrayList<Integer> friends = new ArrayList<>();
            String line;

            //populate the list of friends given the ID
            while((line = bufferedReader.readLine()) != null) {
                String[] friendship = line.trim().split(regex:"\\s+");
                int id1 = Integer.valueOf(friendship[0]);
                int id2 = Integer.valueOf(friendship[1]);

                if(id1 == ID) {
                    friends.add(id2);
                } else if(id2 == ID) {
                    friends.add(id1);
                }
            }
```

'displayFriendList' Function Part 2

The 'bufferedReaer.readLine()' reads the file line by line, and as long as 'readLine() returns a non-null value, the loop will continue. The line that was then read from the file will be saved in the line variable, and will be divided into an array of strings using the 'split("\\s+") method. It then checks if the user's ID matches id1 or id2 by parsing the IDs of the two individuals participating in the friendship into integers (id1 and id2). If the ID matches, then the user with the provided ID is friends with the other person (id1 or id2). The ID of the other person is then added to the friend list.

```
        if(!socialGraph.containsKey(ID)) {
            System.out.println(x:"\n=========================================");
            System.out.println(x:"\nID does not exist.. Returning to Main Menu..");
            System.out.println(x:"\n=========================================");

        } else {
            System.out.println(x:"\n=========================================");
            System.out.println(x:"             Number of Friends           ");
            System.out.println("\nUser " + ID + " has " + friends.size() + " friend/s");
            System.out.println(x:"\n=========================================");
            System.out.println(x:"               Friend List               ");
            System.out.println("\nUser " + ID + "'s Friend List:");
            for(int i = 0; i< friends.size(); i ++) {
                int friend = friends.get(i);
                System.out.println(friend);
            }
            System.out.println(x:"\n=========================================");
            System.out.println(x:"   Friend list successfully displayed!");
        }
    }
}
```

'displayFriendList' Function Part 3

This last code snippet checks if the inputted ID is included in the socialGraph data structure (represents the social network). If the socialGraph doesn't contain the ID, then the person associated with it does not exist in the graph and an error message will be displayed, returning the user back to the main menu. However if the socialGraph contains the ID, then the function will display the user's total number of friends as well as each of those friend's ID. Lastly, it will print out "Friend List successfully displayed" confirming the list of friends of the ID entered.

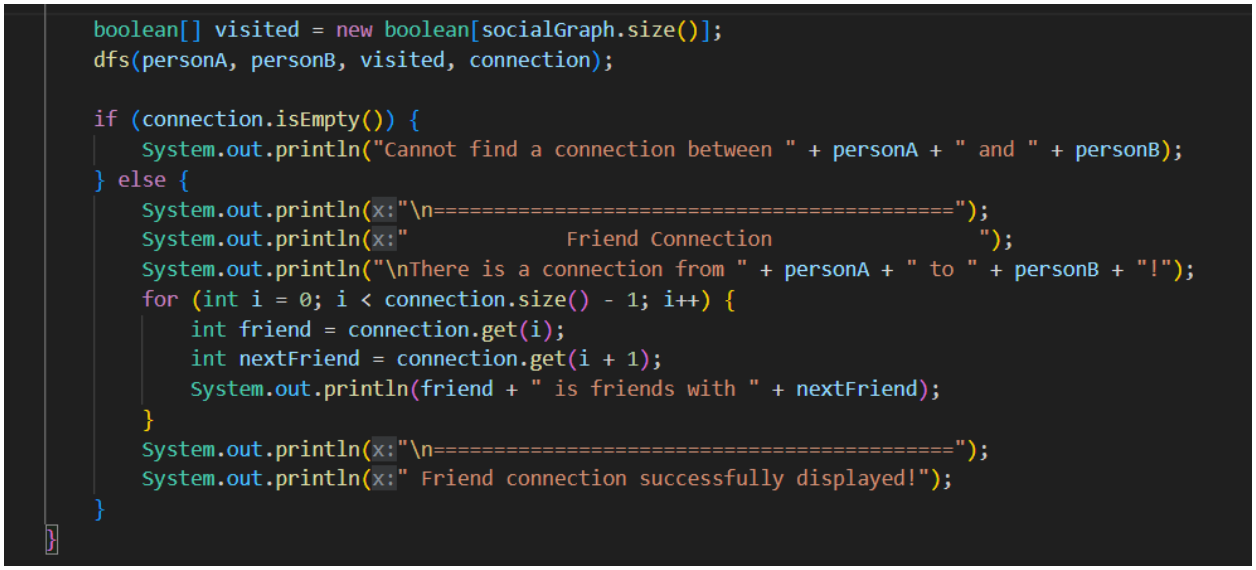
B. Display Connection Algorithm

The 'displayConnection' function is the algorithm used to find a connection between two people in the network in a social graph.

```
private static void displayConnection(String file) throws IOException {
    Scanner sc = new Scanner(System.in);

    System.out.println(x:"Enter ID of first person: ");
    int personA = sc.nextInt();

    System.out.println(x:"Enter ID of second person: ");
    int personB = sc.nextInt();

    List<Integer> connection = new ArrayList<>();
    if (!socialGraph.containsKey(personA) || !socialGraph.containsKey(personB)) {
        System.out.println(x:"\n============================================================");
        System.out.println(x:"\nOne or both of the inputted IDs do not exist in the dataset.");
        System.out.println(x:"\n============================================================");
        return;
    }

    if (personA == personB) {
        System.out.println(x:"\n==========================================");
        System.out.println(x:"\n     The two IDs inputted are the same.");
        System.out.println(x:"\n==========================================");
        return;
    }
```

'displayConnection' Function Part 1

The algorithm requests the user to enter the first person's ID (person A), then uses sc.nextInt() to get the integer input from the user. It then requests the same thing for the second person's ID (person B) which the user's integer input is read using the next sc.nextInt(). A new ArrayList is then generated with the name connection, which stores the list of IDs representing the connection between personA and personB. A printing error message is displayed if one or both of the inputted IDs (personA or person B) do not exist in the dataset and terminates the connection-finding process with the method return. However if Person A and B are the same in the graph (personA=personB), it indicates that the user entered the same ID and a message stating the two IDs inputted are the same and then returns.



```
    boolean[] visited = new boolean[socialGraph.size()];
    dfs(personA, personB, visited, connection);

    if (connection.isEmpty()) {
        System.out.println("Cannot find a connection between " + personA + " and " + personB);
    } else {
        System.out.println(x:"\n========================================");
        System.out.println(x:"            Friend Connection                ");
        System.out.println("\nThere is a connection from " + personA + " to " + personB + "!");
        for (int i = 0; i < connection.size() - 1; i++) {
            int friend = connection.get(i);
            int nextFriend = connection.get(i + 1);
            System.out.println(friend + " is friends with " + nextFriend);
        }
        System.out.println(x:"\n========================================");
        System.out.println(x:" Friend connection successfully displayed!");
    }
}
```

'displayConnection' Function Part 2

The visited array with a size equal to the number of nodes (IDs) tracks visited nodes in the socialGraph during Depth-First Search (DFS). The DFS method calls personA, personB, visited array, and connection list to initiate the traversal to find the connection between personA and personB. Once the DFS traversal is completed, it checks if the connection list is empty, indicating no connection between personA and personB. Otherwise, a message is displayed printing the connection path by iterating through the list and displaying the friendships in a sequence. From the first element (i=0) through the next-to-last element (i < connection.size()-1), the for loop iterates over the connection list. It only proceeds up to the second-to-last element to make sure it can access the following friends in the sequence since the goal is to display the friendships in pairs. Using connection.get(i) and connection.get(i + 1), respectively, it obtains the current friend and the next friend in the sequence for each iteration. The information that the current friend is friends with the next friend is then printed. The output is prepared to provide the details for every friendship in the order of connections. A success message is then printed by the procedure to show that the friend connection has been successfully shown.

```java
private static boolean dfs(int current, int target, boolean[] visited, List<Integer> connection) {
    visited[current] = true;
    connection.add(current);

    if (current == target) {
        return true;
    }

    for (int friend : socialGraph.get(current)) {
        if (!visited[friend]) {
            if (dfs(friend, target, visited, connection)) {
                return true;
            }
        }
    }

    connection.remove(connection.size() - 1);
    return false;
}
```

'displayConnection' Function Part 3

The image shown above is the DFS method, which is a recursive function that performs the Depth-First Search(DFS) traversal of the social graph to identify the link between the current and target. It first starts by marking the current node as visited in the visited array and adds it to the connection list. Then it determines whether the current node and the target node are identical. If the current node and target node are the same then a connection has been found and it returns true, but if it is not the same, then a for loop is used to investigate the nodes that are nearby (friends). The method checks if the nearby friend of the current node has been visited or not with (!visited[friend]). The DFS algorithm searches for connections between current and target nodes by recursively calling DFS with the nearby friends as the new current node. If a connection is found, the method returns true. If no connection is found, the method backtracks and explores other nearby friends. If all nearby friends have been visited or explored, the method returns false. The connection list tracks the current path being explored and stores the sequence of IDs representing the connection path if found. If no connection is found, the connection list will be empty when the DFS method returns.

IV. Algorithmic analysis of the algorithms in terms of time complexity

a. Main Algorithm

Time Complexity: O(n+k)

The time complexity of the main method has a Big O notation of n+k as the frequency count of the lines of code within the while loop depends on how many times the loop runs (n = amount of times the menu is used) with an additional 1 for when the condition for the while loop gets checked. With that, the (k) which comes from the lines of code running just once when calling the main method (try, catch, initializations).

b. Load Graph/Data (Part 1)

Time Complexity: O(n+k)

The time complexity of the loadData method also has a Big O notation of n+k as the frequency of the lines inside the for loop will run (n = the amount of accounts + friendships in the social graph) amount of times plus 2 since (i) in both loops starts at zero and also for the checking of the condition one last time. The rest of the (k) comes from the lines of code that runs just once (try, initializations, print statements).

c. **Display Friend List (Part 2)**

Time Complexity: O(n+k)

The time complexity of the displayFriendList method also has a Big O notation of n+k. The (n = amount of connections in the social graph + amount of variables in the friends arraylist) represents the amount of times the codes inside the while and for loops run with an addition of 1 for the while loop and 2 for the for loop due to the checking of the conditions. The lines inside the for loop also have an additional 1 each due to the (i) starting at 0 instead of 1. The rest (k) comes from the lines of code that run just once when calling the method (print statements, initializations, if-else statements, try).

d. Display Connections (Part 3)

Time Complexity: O(n+k)

The time complexity of the methods displayConnection and dfs has a Big O notation of n+k. The (n = amount of variables in the connection arraylist + amount of variables in the social graph) represents the amount of times the codes inside the for loops of both methods. The (k) comes from the other lines of codes that only run once when the method is called (initializations, if-else statements, print statements).

e. Overall Time Complexity Analysis

The overall frequency count of the whole program has a Big O notation of O(n+k), as all methods observe a frequency count with a Big O notation of O(n+k).

V. Summary of Findings with respect to data structures and graphs

When writing the program, the group has learned and implemented the hash map as the data structure due to the fact that it is one of the most efficient algorithms when representing a social network or graph – along with adjacency lists. This is due to the fact that hashmaps use a key to directly access the data to add and retrieve it with little runtime which makes searching for the frequency of an ID and its connections from the data set more efficient and less complex. Moreover, hash maps allow for duplicates which are present in the data sets as they are meant to represent the bidirectionality of the friend function of facebook. Additionally, according to the research the group has made, hash maps have a time complexity with a Big O notation of O(1) which supports the fact that it is one of the most efficient algorithms for these types of prompts which may prove to be useful for the group's members future coding projects.

With respect to the time complexity of the group's program, the group believes most of the time spent running are due to the searching algorithms that are, at best, n+k as the data files were believed not to be sorted which prompted the group to opt for linear search algorithms. On the bright side, the group did not use any algorithms which requires recursion nor nested loops which will make the time complexity of the program grow exponentially faster (compare O(n*n) to O(n+k)).

VI. References (APA Format)

AlgoDaily. (n.d.). Implementing Graphs: Edge List, Adjacency List, Adjacency Matrix. Retrieved from https://algodaily.com/lessons/implementing-graphs-edge-list-adjacency-list-adjacency-matrix

GeeksforGeeks. (n.d.). Adjacency List - Meaning & Definition in DSA. Retrieved from https://www.geeksforgeeks.org/adjacency-list-meaning-definition-in-dsa/

GeeksforGeeks. (n.d.). Graph and its representations. Retrieved from https://www.geeksforgeeks.org/graph-and-its-representations/

Programiz. (n.d.). Graph - Adjacency List. Retrieved from https://www.programiz.com/dsa/graph-adjacency-list#:~:text=An%20adjacency%20list%20is%20efficient,adjacent%20to%20a%20vertex%20easily.

ProgressiveCoder. (n.d.). Graph Implementation in Java using HashMap. Retrieved from https://progressivecoder.com/graph-implementation-in-java-using-hashmap/

Software Testing Help. (n.d.). Java Graph Tutorial: Graphs in Java with Program Examples. Retrieved from https://www.softwaretestinghelp.com/java-graph-tutorial/#:~:text=Usually%2C%20we%20implement%20graphs%20in,adjacency%20matrix%20or%20adjacency%20list.

Stack Overflow. (n.d.). Have Java find/read a file from user input. Retrieved from https://stackoverflow.com/questions/19626468/have-java-find-read-a-file-from-user-input

Tutorial Horizon. (n.d.). Graph Implementation – Adjacency List - Better | Set 2. Retrieved from https://tutorialhorizon.com/algorithms/graph-implementation-adjacency-list-better-set-2/

W3Schools. (N/A). The Try Statement in Java. Retrieved [Accessed Date], from https://www.w3schools.com/java/ref_keyword_try.asp.

VII. Group Contribution Table

Members	Contributions
Joehanna	- Code for loadData - Introduction - Description of the data structure used in representing the social graph data, include rationale behind the use of such data structure/s.
Harmony	- Code for displayConnection - Description of algorithms used to display friend list and connection between two people in the network
Justine	- Code for displayFriendList - Algorithmic analysis of the algorithms in terms of time complexity - Summary of Findings with respect to data structures and graphs