Presented to the College of Computer Studies
De La Salle University - Manila

Term 3, A.Y. 2022-2023
In partial fulfillment of the course
In CCDSALG

Major Course Output 2 Report for Data Structures and Algorithms

Submitted by:

Gilo, Joshua Armaine G.	Parker, Peter B.
Group No. 23 S-13	Group No. 19 S-12

Submitted to:
Chu, Shirley

31 July, 2023

Contents

Abstract

This paper presents a Java-based implementation of a pseudo-social network. It utilizes a graph-based data structure and employs real-world data derived from Facebook in 2005. We utilized HashMaps and HashSets for efficient data management. This data structure represents various relationships within the social network, where each user and edge represents a user and relationship, respectively. A user's friends are stored as a HashSet, which allows for constant-time complexity during lookups and insertions, making it ideal for large datasets. The Breadth-First Search (**BFS**) algorithm is used to determine the path between two users within the network. We also provide a detailed analysis of the algorithm's implementation using its time complexity. This paper underscores the importance of choosing the appropriate data structures for efficient data representation and analysis.

Keywords: Java, pseudo-social network, graph-based data structure, HashMaps, HashSets, real-world data, Facebook, Breadth-First Search, algorithm, time complexity, data representation, data analysis.

Introduction

Graph-based data structures, together with a wide range of traversal algorithms, serve as the foundational building blocks for numerous real-world applications. These applications span across **social networks**, route planning, as well as interactions among various biological entities. In this project, we utilized these fundamental principles to construct a program capable of representing a *pseudo-social network* using **Java**.

In the digital age, **social networks** have become an integral part of our daily lives. These networks have reshaped the ways in which we communicate, share, and interact with one another. As these networks grow in size, the need for tools to analyze and understand them becomes increasingly important. This project aims to leverage the principles of graph theory to model a social network.[3]

The objective of this project is to develop a program that could efficiently model a **social network** using graph-based data structures, as well as implement two (2) functionalities which allow users to analyze the network. To achieve these objectives, we used a real-world dataset from **Facebook** in 2005, which encapsulates the networks of numerous American universities. By analyzing this particular dataset, we aim to gain insights into the nature of **social networks**.

This paper presents an in-depth discussion of the design and implementation of our programs as well as the various operations it offers, and finally some results obtained from analyzing the dataset. We also provide the time complexity of the various algorithms within the program. The conclusions as well as the contributions of each member are also discussed within this paper.

Background

Graph Theory and Social Networks

Social networks have emerged as prominent platforms in human interaction and communication. As these networks grow in size, the need for tools to analyze them also escalates. This is where graph theory becomes invaluable. The use of graphs provides tools to represent real life individuals and entities within a social network, and their interactions in a more structured and visual manner.

The fundamental components of a graph are nodes and edges. Nodes represent individual entities within a network, while edges signify the relationships or connections between these entities. For example, in a social media platform, a node could represent an individual while an edge could indicate a friendship or connection between two individuals.

Adjacency List Representation

In a graph, nodes are considered adjacent if they share a common edge. In a social network, two individuals are adjacent if they have a relationship (edge) with one another. The set of all vertices adjacent to a node v is often referred to as its adjacency list.

An adjacency list is a way to represent a graph where each node in the graph is associated with a list of neighboring vertices. This list is commonly referred to as the **neighbors** of v. The choice of using an adjacency list is a direct consequence of the nature of social media platforms. These networks are typically sparse. A sparse network means that each node is (on average) connected to only a small fraction of all other users within the network.

The use of adjacency lists are also preferred whenever most of the analysis is done using graph traversal algorithms[3]. In such scenarios, an adjacency list is a more space-efficient alternative to a matrix representation which requires more space that is proportional to the square number of nodes, regardless the number of edges. In our project, each user is represented as a single node in the graph. For each user within the network, we maintain a list of all other users that they are connected to. This is the user's adjacency list.

HashMap and HashSet Data Structures

In Java, HashMaps and HashSets are part of the Collections Framework which provides pre-packaged data structures that we can use to manipulate data more efficiently.

A HashMap stores key-value pairs, also known as entries. This is similar to how a dictionary stores the word as well as it's definition.[1]. Without going into much detail, a HashMap uses a technique known as hashing to determine where in the array the value for a key is stored. The key's hashcode is then used to retrieve the value. On average, this makes the retrieval time constant, as it does not depedend on the number of entries within the HashMap [2][1]

In our code, we chose to use a **HashSet** to represent the adjacency list of a single user. A **HashSet** is a collection of items where each items within the set is unique. Due to the nature of a HashSet, it also provides (on average) constant time performance for basic operations. This ensures that our program can handle large datasets efficiently.

When a neighbor-lookup is performed, we can simply look at the adjacency list of the user, which gives us an efficient way to access the required information. Similarly, when we want to find the path between two users, we can perform a Breadth-First Search (BFS) starting from the source user until we iteratively explore the adjacency list of each visited user until eventually, the program reaches the target user.

Breadth-First Search Algorithm

Breadth-First Search (BFS) is a graph traversal algorithm. It starts at a chosen source node, then explores its neighbors iteratively before moving on to their neighbors. This algorithm explores the graph in "layers" from the source node, then visiting the nodes that are a single edge away, then two edges away, and so on.

One of the features of BFS is that it finds the shortes path between the source node and all other nodes. This assumes that each edge is unweighted. In the context of a social network, BFS can be used to determine the shortest path between two individuals. The BFS algorithm is implemented to find the shortest number of connections between two people within the network.

Dataset

For this project, we used with a **real-world dataset**, derived from *Facebook in 2005.* This data encapsulates the Facebook networks of numerous American

[1]HashMaps in Java are equivalent to dictionaries in other programming languages, such as Python.

[2]Assuming there are no Hash Collisions within the HashMap. This also assumes a good hashing function was used. See Section: Time Complexity Analysis

colleges and universities and provides a snapshot of a single day in September 2005. In the original dataset, the user accounts were anonymized and assigned unique integer identifiers, preserving their suitability for analysis. This dataset enables the construction of a social graph that represents various accounts and the relationships formed between them. This relationship is depicted using an *undirected graph*, mirroring the bidirectional nature of friendships on Facebook. [5] However, it is important to note that the dataset has its limitations. It only represents a snapshot of the Facebook network at a single point in time. Since 2005, Facebook has grown and evolved, and so this dataset wouldn't fully reflect the current state of the network. Also, this dataset only includes data on friendships and not other types of user interactions or attributes of the user.

Data Processing

To facilitate processing, the original data, which was in MATLAB (.mat) format, was converted to a text (.txt) file. The first line of each file consists of two integers, n and e, representing the **number of accounts** and **friendships** within the network, respectively. This line is followed by e lines that describe the friendships between two accounts.

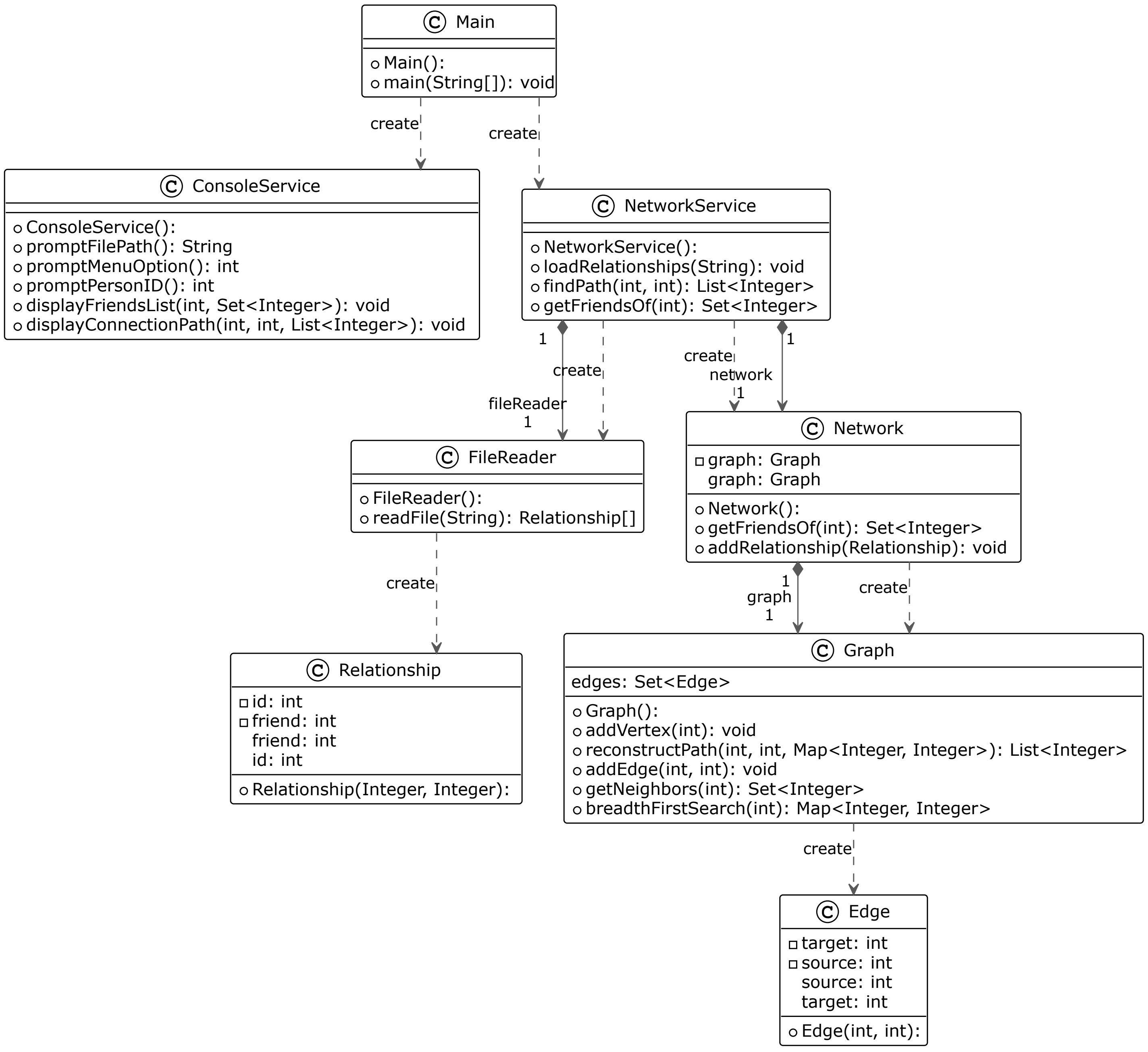

Figure 1: UML Diagram of the Program

Program Design and Implementation

FileReader Class

The FileReader Class is responsible for reading and parsing data from a file. It reads a social network dataset stored in a file, each line within this file represents a friendship between two users. The class processes each line, extracting the user IDs and creating the relationships. The FileReader class transforms raw data into a format ready for the construction of a graph.

Graph Class

The Graph class represents the social network as a graph structure. This class contains the adjacency list where keys are user IDs and values are a HashSet of IDs representing friendships. The HashSet ensures that duplicate edges are not added and allows for efficient look-up. The Graph class also provides methods for adding edges and performing BFS, as well as reconstructing its path.

Edge Class

The Edge class represents the connections in the social network graph. Each instance of this object represents a direct connection between two users. It contains two Node objects, indicating the two users involved in a relationship. The Edge class provides an object-oriented way to represent friendships within the social network.

Network Class

The Network class is a high-level abstraction of the entire network. It contains an instance of the Graph class, which encapsulates the entire network. This class contains methods for loading data from a file, displaying a user's friend list, and finding the shortest path between two users.

NetworkService and ConsoleService Classes

The NetworkService class is the bridge between the underlying Network model and the ConsoleService view. It contains an instance of the Network class and provides methods to execute user commands on the network. The ConsoleService class is responsible for handling user interactions. It provides a command-line interface (CLI) which allows the users to interact with the social network by displaying a menu, accepting input and presenting the results of the various operations.

Relationship Class

The Relationship class encapsulates the details of a relationship in the social network. Each instance of this class represents a relationship between two users, identified by their respective IDs. The class comprises two main attributes: id, representing the user, and friend, representing the user's friend.

Main Class

The Main class serves as the entry point of the program. It is responsible for starting the application, creating an instance of the ConsoleService and NetworkService classes. It calls the method to start the interface.

Program Operations

Listing 1: A Sample Run of the Program

```
Input file path: ./data/Caltech36.txt
[1] Get friend list
[2] Get connection
[3] Exit
Enter your choice: 1
Enter ID of Person: 10
Person 10 has 2 friends!
List of friends:
341 663
[1] Get friend list
[2] Get connection
[3] Exit
Enter your choice: 2
Enter ID of Person: 10
Enter ID of Person: 20
There is a connection from 10 to 20!
10 is friends with 341
341 is friends with 686
686 is friends with 20
[1] Get friend list
[2] Get connection
[3] Exit
Enter your choice: 3
```

Loading a Dataset

When the user asks the program to load a dataset, it begins a multi-step process that involves reading the file, parsing its contents, and creating the graph using the parsed data.

1. **File Selection and Opening** The process begins when the user inputs the path of the file. This integers represent two people who are friends with one another. The FileReader creates a Relationship object for each pair of integers and adds it to a list of relationships.

2. **Building the Network Graph** After all relationships have been read from the file and added to the list, the FileReader returns this list to

the loadRelationships method of the NetworkService class. This method then iterates over the list then adds each relationship to the Network object. This is accomplieshed by invoking the addRelationship method of the Network class. The add Relationship method adds each person from the relationship as a node in the graph, if they're not already in the graph. Then, it adds an edge (relationship) between them.

3. **Completion of Network Loading** After all relationships from the file have been added to the network, the graph is fully constructed. The application is now prepared to perform operations based on the user's choices, such as finding a person's list of friends or finding a path of connections between two people.

List Friends of a Person

If the user selects the first option on the menu, the application proceeds to list the friends of a person. The user is prompted to enter an ID. This ID is passed to the getFriendsOf method of the NetworkService class which uses the getNeighbors method of the Graph class to fetch all the friends of the person with the given ID. These friends are represented as a set of IDs. The displayFriendsList method of the ConsoleService class then prints the list of friends to the console.

Path of Connection Between Two People

When a user selects the second option, the application is tasked with finding a path of connections between two individuals within the network.
The user is asked to input two IDs, which represent the two individuals for whom a path is constructed. The `promptPersonID` method of the `ConsoleService` class is invoked twice to get two IDs. The IDs `id1` and `id2` are then passed to the `findPath` method of the `NetworkService` class.
The `findPath` method uses BFS to find the shortest path between the two individuals. BFS is a traversal algorithm that explores the graph in layers starting from the source node.
The BFS algorithm starts by initializing the following data structures:

- A queue, which will hold the nodes to be explored.
- A visited set, which keeps track of nodes that have been explored to avoid re-exploration.

- A parentMap, which records the parent-child relationships between nodes.

The process begins by adding the starting node to both the queue and the visited set. It also adds the starting node to the parentMap, with its parent set as `null`.
Next, the BFS enters a loop that continues until the queue is empty. If the queue is already empty, then it signifies that all reachable nodes have been explored. Each iteration of the loop dequeues a node from the front of the queue. For each neighbor of this current node that has not been visited yet, it does the following:

- Add the neighbor to the queue, marking it for future exploration.
- Add the neighbor to the visited set, which marks it as discovered.
- Set the current node as the neighbor's parent in the parentMap. This step records the path the algorithm took to discover the neighbor.

Path Reconstruction

Once the BFS has been completed, the `findPath` method now has a complete parent map which it can use to reconstruct the path from the starting node to the target node. The `reconstructPath` method of the `Graph` class is then invoked.
The `reconstructPath` method starts at the goal node and follows the parent pointers in the parent map all the way back to the starting node. It does this by initializing a `currentNode` variable as the goal node. It then enters a loop that continues until the `currentNode` is `null`. In each iteration, it adds the `currentNode` to a path list and then sets the current node to its parent in the parentMap.
After this, the path list represents the path from the goal node to the start node, so it is then reversed so that it starts from the starting node.
Finally, this method also checks if the start of the path is the start node. If it's not, this means that the start and target nodes aren't connected, and the method returns an empty list. Otherwise, it returns the path list.

Time Complexity Analysis

Optimizing Adjacency Lists: HashMap's *get()* and *put()*

A ***Map*** is a data structure which could store key-value pairs, akin to how a physical dictionary stores a "word" and a "definition". However unlike a physical dictionary where a single word could have multiple definitions, a ***Map*** could only have a single value for each key. A ***HashMap*** extends this concept by using a ***Hash Function*** which "links" the key-value pair together.
Every time data (a key-value pair) is added to a ***HashMap***, the data is hashed and the key is converted into a hash-code which determines the location (bucket) in which the data is stored. We make the assumption for analysis that these keys have a uniform distribution after the hash function has placed them. This would imply each bucket has a single entry only and no hash collision would occur.
A ***Hash Collision*** occurs when two or more keys are mapped to the same bucket, which could overwrite data and lead to inefficiencies within the algorithm. Given these assumptions, the *get()* method with a particular key as its parameter would have a time complexity of $\mathbf{O(1)}$ due to the nature of a ***HashMap*** itself. A *put()* method would also have a similar time complexity because it takes a similar amount of time to "hash" any particular key-value pair, and store them into the bucket generated from the key of the data. [1]

Breadth-First Search (BFS) Algorithm Analysis

Initialization

The BFS algorithm begins by initializing a parent map for backtracking, a queue, and a set for visited nodes. It adds the source node to both the queue and the visited set, and assigns its parent as null. Given that each of these steps has a constant time complexity, the initialization is $\mathbf{O(1)}$.

Queue Operations

Each node is added into the queue upon its first discovery. Once all its neighbors have been explored, the node is removed from the queue. This process is similar to the "coloring" of nodes in the CLRS implementation where 'White' signifies an **unvisited node**, 'Gray' signifies a node with **unvisited neighbors**, and 'Black' signifies a node with **all neighbors explored**. Since add and remove operations from a LinkedList have constant time complexity, the total time complexity of these operations is $\mathbf{O(V)}$, where $\mathbf{V}$ represents the number of nodes in the graph.[2]

Adjacency List Check

For each node, BFS checks its neighbors in the adjacency list to identify the unvisited nodes. These unvisited nodes are then added to the queue and the visited set, and their parents are set in the parent map. The overall time complexity for this process is equivalent to the summation of all adjacency list sizes, represented by **E**, since each edge appears twice, for each endpoint.[**?**]

Final Runtime for Breadth-First Search

In conclusion, the sum of the time complexities of these processes results in a final time complexity of $\mathbf{O}(\mathbf{V} + \mathbf{E})$, which is linear with respect to the size of the graph. This demonstrates that BFS is a notably efficient algorithm for graph traversal, as its performance scales linearly with the size of the graph.[**?**]

Results and Discussion

In this section, we present and discuss the results of running our application. We examine the program's performance under normal conditions and its handling of various edge cases.

Normal Conditions

Under normal conditions, our program performs exceptionally well. When given valid input, the program quickly loads the dataset and allows the user to perform operations on the network.
For instance, consider the following interaction:

```
Input file path: ./data/Caltech36.txt
[1] Get friend list
[2] Get connection
[3] Exit
Enter your choice: 1
Enter ID of Person: 10
Person 10 has 2 friends!
List of friends:
341 663
```

In this example, the program correctly identified the friends of person 10 from the Caltech36 dataset. This was verified by using the application **Cytoscape** and the **PathExplorer** extension.[4]

Edge Cases

The program also handles various edge cases gracefully. Here are some examples:

Invalid File Path

When given an invalid file path, the program displays an error message and prompts the user for a valid file path. For example:

```
Input file path: 1
Error: File not found. Please enter a valid file path.
Input file path:
```

This shows that the program is robust to errors in file input, and can recover gracefully from such situations.

Invalid User ID

When asked to list the friends of a user who does not exist in the network, the program informs the user that the ID is invalid. For example:

```
Enter your choice: 1
Enter ID of Person: 99999
Error: Person 99999 does not exist in the network.
Enter ID of Person:
```

This demonstrates the program's error handling capabilities and its ability to provide feedback to the user.

Multiple Shortest Paths

In cases where multiple shortest paths exist between two users, the program arbitrarily selects one to display. This is a result of the BFS algorithm, which does not guarantee a particular shortest path in cases where multiple paths exist.

Listing 2: Example of a Dataset with Multiple Paths

```
6 6
1 2
2 3
3 4
1 5
5 6
6 4
```

Listing 3: Actual Output

```
Input file path: data/small_multiple.txt
[1] Get friend list
[2] Get connection
[3] Exit
Enter your choice: 2
Enter ID of first person: 1
Enter ID of second person: 4
There is a connection from 1 to 4!
1 is friends with 2
2 is friends with 3
3 is friends with 4
```

Contributions

Member	**Contributions**
Peter Parker	Created figures, slides
Joshua Gilo	Analyzed the time complexity, wrote the report in $\LaTeX$
Both Members	Developed the program, wrote the report, analyzed the algorithms

Conclusion

Our study on data structures and algorithms focused on Adjacency Lists, Graphs, HashMaps, and the Breadth-First Search (BFS) Algorithm. We went deep into the intricacies of finding neighboring nodes, utilizing various methods and classes within the Collections library as well as reconstructing the path of BFS. We found out that BFS is efficient enough for traversing graphs because of its linear time complexity O(V+E), it also scales well with the size of any given graph. These insights underscore the importance of properly applying the appropriate data structures to represent real world data. These learnings will personally serve as a foundation for further study by the authors of this paper as they further explore and innovate within the field.

Bibliography

[1] Dinesh Bajracharya. A review on Java HashMap and TreeMap. *International Journal of Engineering Applied Sciences and Technology*, 5(1):134–138, 2020. Published Online May 2020 in IJEAST.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[3] Norman P. Hummon and Patrick Doreian. Computational methods for social network analysis. *Social Networks*, 12(4):273 – 288, 1990.

[4] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S Baliga, Jonathan T Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research*, 13(11):2498–2504, 2003.

[5] Amanda L Traud, Peter J Mucha, and Mason A Porter. Social structure of facebook networks. *Physica A: Statistical Mechanics and its Applications*, 391(16):4165–4180, 2012.