PROGRAMMING PROJECT

DOCUMENT

SOCIAL NETWORK

Group Members
Oihan Irastorza Carrasco
oirastorza001@ikasle.ehu.eus
Eneko Pizarro Liberal
epizarro001@ikasle.ehu.eus

Mon, 12 Jan 2022
Subject: Data Structures and Algorithms
Degree: Informatics Engineering
Academic Year: 2021/2022
School: Facultad de Informática UPV/EHU

Table of Contents

Abstract

GENERAL VIEW

The aim of this document is to summarize the project developed in the Data Structures and Algorithms subject. This project consists of coding the structure of a basic social network, where people can join, leave or make relationships between them.

Throughout the document, the reader will find the different versions of our project as well as the hierarchy of the defined classes and packages, the implementation of the code and the decisions taken in order to select the appropriate data structures.

OBJECTIVES

1. Build a social network where people can establish relationships.
2. Learn and implement different proper data structures for networks and people.

Introduction

0.1. GENERAL VISION

The programming project consists in the creation of a **social network** that connects different people that will be loaded into the network and contain the information of the users as the relationships they maintain between them.

The intention of the programming project is to implement in Java the **data structure that represents the entire network and the people that form it** so advancements in the study of the design and implementation of data structures and algorithms throughout the course can be made.

It is useful to study the basic data structures and algorithms that allow the efficient solution of typical computational problems. Such problems will be encountered throughout the project as there will have to be implemented efficient searching and sorting algorithms for the people of the social network and the relationships between them.

Furthermore, there will have to be studied the different data structures that can shape the model of the social network so the most appropriate one can be found and use it for creating the social network without changing the client's perception and use of the aforementioned.

It will have to be taken into account that as the network's structure develops, changes in the design will be made, and alternatives will have to be rethinked. That will be the result of getting to know the best data structure and algorithm types that adjust to the requirements of the programming project. As a conclusion, for the same reason the different versions of the programming project; created throughout the development and conclusion of the project, will be explained and detailed in this same document. Note that some methods and algorithms will be entirely changed or not implemented as changes may occur in the middle of an implementation of a specific method.

0.2 OBJECTIVES

1. Create a menu via console for the user to interact with and choose between actions for interacting with the social network.
2. Read and recognize the inputs of the user in the menu and respond correctly to them.
3. Create a social network using the most appropriate data structure, and administer the information that contains: sorting it out appropriately, applying specific methods to them...
4. Load information from external .txt files into the network.

5. Retrieve the information into an external file or print it out onto the screen.

0.3 TASKS

1. Create a clique of people for introducing it to the network.
    a. Information included in each person's social network profile:
        i. Identifier (unique): For singling out each person
        ii. Name
        iii. Surname(s)
        iv. Birth date
        v. Gender
        vi. Birthplace
        vii. Home: Location address
        viii. studiedat: Multiple places where the user has studied at
        ix. workedat: Multiple places where the user has worked at
        x. movies: Films that the user most likes, can be multiple ones.
        xi. groupcode: Code that designs a specific group of users and the relations between them.
2. Implement a menu so the user can select between different choices for interacting with the social network.
    a. There will have to be loaded the people and their respective information into the network from an external .txt file as well as the relations between them from another archive.
    b. The information gathered in the social network must be able to be retrieved into a file or printed out on the screen for reading.
3. Create a **JUnit** for **testing** the implementation of the project, methods, algorithms, data structures... without taking into account the actual implementation of them. There will only be tested the functioning and efficiency of the project from a user or client point of view.

1st Project Version

Creation of the menu, the initial network structure and the class that represents the user.

1.1. CLASSES DESIGN

Package: networkinterface

This package contains the user interface and the classes related to it. The main use of this package is to organize the classes of the menu of the social network and the subclasses that contain the methods that execute the options of the menu.

Menu

This class is the main class that will be executed in the console. Contains the menu prints into the console, the options and reads the user input.

This is implemented into a loop that ends only when the user introduces the number -1, so it quits the menu and terminates the network.

JFileChooserWindow

Contains the methods so a pop up window can be displayed in the menu for the user to choose a file from the directory.

Package: data

Network

This class represents the network where the people and the relations between them will be stored, as the methods for administering it.

People

This class contains the information of each people (user) loaded into the network.

1.2. DESCRIPTION OF THE DATA STRUCTURES USED IN THE PROJECT

Package: data

Network and People

Both Network and People structures are implemented using ArrayList because at the beginning was the most suitable data structure for accessing information using an index and searching through it using indexes as well.

ArrayList

The ArrayList[1] class is a resizable array, which can be found in the java.util package.

The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a bigger new one and copy all its contents to the new one). While elements can be added and removed from an ArrayList whenever you want -internally Java does something similar-. The syntax is also slightly different.

Method	Description
boolean add(Type name)	Appends the element to the end of this list.
void add(int index, E element)	Inserts the specified element at the specified position in this list.
void clear()	Removes all of the elements from the list.
boolean contains(Object o)	Returns true if this contains the specified element.
E get(int index)	Returns the element at the specified position in this list.
boolean isEmpty()	Returns true if this list contains no elements.
E remove(int index)	Removes the element at the specified position in this list.
Boolean remove(Object o)	Removes the first occurrence of the specified element from this list, if it is present.
int size()	Returns the number of elements in this list.
indexOf(Object o)	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

[1] "ArrayList (Java Platform SE 7) - Oracle Help Center." https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html. Se consultó el 25 oct. 2021.

1.3. DESIGN AND IMPLEMENTATION OF THE METHODS

Package: data

Network

- **loadPeopleFromTxt.** Loads people from a txt file.

```
public void loadPeopleFromTxt(String txtPath)
{
    try
    {
        Scanner sc = new Scanner(new FileReader(txtPath));
        while(sc.hasNextLine())
        {
            String peopleInfo = sc.nextLine();
            String[] parts = peopleInfo.split(",");
            people.add(new People(parts[0], parts[1], parts[2], parts[3],
                                  parts[4], parts[5], parts[6], parts[7],
                                  parts[8], parts[9], parts[10]));
        }
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
}
```

People

- **toString**

```
@Override
public String toString()
{
    return idperson + " " + name + " " + lastname + " " +
            gender + " " + birthplace + " " + home + " " +
            studiedat + " " + workplaces + " " + films + " " + groupcode;
}
```

Package: networkinterface

InitialMenu

- **main**

```java
package networkinterface;

import java.util.Scanner;

public class InitialMenu {
    public static void main(String[] args) {
        System.out.println("    \"Menu\"\n");
        System.out.println("1. Load \"people\" into the network...");
        System.out.println("2. Load \"relationships\"...");
        System.out.println("3. Print out people...");
        System.out.println("4. Search...");
        System.out.println("...");
        System.out.println("...");
        System.out.println("5. Log Out");

        int userchoice = 0;
        while(userchoice != logOutKey)
        {
            System.out.print("\nInsert option: ");
            try
            {
                switch(sc.nextInt())
                {
                    case 1:
                        jFW.openJFileChooserWindow();
                        myNetwork.loadPeopleFromTxt(jFW.getPath());
                        System.out.print("People successfully loaded.");
                        break;
                    case 3:
                        System.out.println("People in the network:\n");
                        myNetwork.printAllPeople();
                        break;
                    case 5:
                        System.out.println("Quit");
                        userchoice = logOutKey;
                        break;
                    default:
                        System.out.println("The option selected is not"
                                        + "available at this time.\n");
                }
            }
            catch(Exception e)
            {
                System.out.println("An unexpected error has occurred!");
            }
        }
        sc.close();
    }
}
```

JFileChooserWindow

- **openJFileChooserWindow**
    - Opens the JFileChooserWindow and gets the file, path and filename (if you want to get them use the getters).

```
20•     public void openJFileChooserWindow()
21      {
22          int returnValue = jfc.showOpenDialog(null);
23          //int returnValue = jfc.showSaveDialog(null);
24
25          if(returnValue == JFileChooser.APPROVE_OPTION)
26          {
27              File selectedFile= jfc.getSelectedFile();
28              path = selectedFile.getAbsolutePath();
29              filename = selectedFile.getName();
30              System.out.println("absolute path: " + path);
31              System.out.println("filename: " + filename);
32          }
33      }
```

- **geters**

```
35•     /**
36       * @return The file that the user has entered.
37       */
38•     public File getFile()
39      {
40          return selectedFile;
41      }
42
43•     /**
44       * @return Returns the absolute path of the file entered by the user.
45       */
46•     public String getPath()
47      {
48          return path;
49      }
50
51•     /**
52       * @return The name of the file entered by the user.
53       */
54•     public String getFileName()
55      {
56          return filename;
57      }
58  }
```

1.3.1 DESCRIPTION OF TASKS FOR 1ST MILESTONE

1.3.1.1 Creation of Menu

In this task the objective is to present the user; this is, the backend of the project, with all the options available for the user to access the network and work with it.

The user is presented with a square of 10 options which to choose from. Here are all the options listed, those options will be presented to the user at the beginning of the program via the console inside a square for clarity sake (note that each number corresponds to the number for selecting their option):

1. Load people into the network.
2. Load relationships.
3. Print out people.
4. Print relationships

The implementation is done using a while loop, which can be exited once the 11 is entered in the terminal. This allows us to once the first action is executed continue with another one just by selecting the corresponding option. This is suggested by showing the user the initial option table again once the previous action is finished.

Each case inside the switch has a unique function and a break, so no other option is executed without the user's permission. Every option's function is implemented right under the switch statement, this allows us to read crearly what every option executes. Each function either prints in the screen the results, saves them to a .txt file selected using the *JFileChooser* or both if the user desires so.

1.3.1.2 Load people into the network

This is the first option of the menu and you can execute it by inserting the number 1 into the console when asked.

This option lets you choose a specific *.txt* file where to load the people from, thanks to the JFileChooser user interface, which pops a window for getting to the path of the specific file or multiple files.

Once the file is selected, it is begun to be read. If the chosen file is not the correct one or the internal structure of itself is not the requested one the user will be told to select another option again until a correct file is found.

While the file is being read, avatars of the user are being created (objects representing a single person) and added to the network, without any relationship between them. Each user has no friends initially, they are disconnected between them (a single connected component; there will be as many components as people in the network).

****The persons are represented using the class named *Person* included in the *packnetwork* package.**

****Once the person is instanced it will be included in a HashMap data type whose keys are the unique ID (identification code) of each person and content (mapped value) will be all their *Person* instance. So when a person's ID is given either by the user or another function, the entire information of the person is obtained.**

Furthermore, each person to be included into the network must have specific attributes, some unique to them. Those attributes will also be included into the person's information.

These are the attributes that are stored into each of the person's instance while being included into the network:

1. ***ID*** (string): Unique identification code for recognizing and discerning every single person included in the network.
2. ***firstName*** (string): First name for the person. If the person has a compound name, all of it will be stored in this field.
3. **lastName** (string): The first surname of the person. If the person has a compound surname, all of it will be stored in this same field.
4. ***birthdate*** (string): The birthday of the person. The format should be as follows: day-month-year; all the fields of the date must be separated by a line, the day and the month bust have 2 digits not as the year, that must have 4. All of the components must be in the same string.
5. ***gender*** (string): Gender of the person, either "male", "female" or "non-binary"; no other formats will be allowed, all must be in lower case without spaces.
6. ***birthplace*** (string): Town of city where the person was born in.
7. ***home*** (string): Current residence town or city of the person.
8. ***studiedat*** (list of strings): Different institutions where the person has studied at. They must be separated by a ';', there is no maximum of institutions and space characters are allowed.
9. ***workedat*** (list of strings): Cities or towns where a person has worked in. They must be separated by a ';', there is no maximum of work places and space characters are allowed.
10. ***movies*** (list of strings): List of the favourite movies of the person. They must be included between the given options, they must be separated by a ';', there is no maximum of films and space characters are allowed.
11. ***group code*** (string): A code identifying the work group to which the person is part of, it can include digits and letters.

- **How each person is read from the .txt file and included into the network:**

****This operation is performed by the function loadPeople of class Menu (packmenu) which calls the function loadPeople from class Network (packnetwork).**

- *packnetwork.Network - > loadPeople()*

This function is implemented using the *BufferedReader*. Each line is readed every time until there are no more lines from the .txt file selected by the user using the JFileChooser.

Everytime a line is read, the content is separated by the split function, following the format agreed upon, and the line's elements are converted to their corresponding data type and created a new People object. Finally each person is inserted into the network.

1.3.1.3 Print the people in the network onto a file

If the user requires it, all the people and their information can be printed out to a text file selecting the 3rd option on the menu.

This is implemented using a loop and calling to each person's .toString function, which returns each person's information in the same format as they were inserted.

The out file's directory can be chosen by the user with the JFileChooser.

1.3.1.4 Load the relationships of each People

Given that each person's relationships are reciprocal, we will go storing them using the function ***addRelationship* in the *Network* class.**

We have to pass two person's IDs as arguments for the addRelationships. If either one of the person is not registered on the network an exception is raised (InvalidRelationshipException) and the user will have to either load another file or continue executing another menu option.

Also, if the relationship is already loaded then, the same exception is raised.

Relationships are between two different users, so, if by mistake the same person is related with itself also an exception is raised.

For the actual implementation, the relationships are stored in the map by inserting the friend's person object to their hashmap corresponding key as a value, so each person has all their relationships stored at the same place and access by selecting a certain user's ID (key string) and retrieving their value iterable set; different from the people set, this one is called relations and is implemented using a HashMap (note that we are referring to the final implementation of the function, as in the beginning this was coded using ArrayList).

Finally, as far as the user is concerned, they will have to choose the 2nd option on the menu to execute this function and select the corresponding .txt file directory from the computer (using the JFileChooser) from where the relations can be retrieved.

To take into consideration:

1. Format: **friend1,friend2. Each line on the friends.txt file must be a single non**

repeated relationship.

2. Each pair of friends must be already included in the network by themselves.
3. The relationship can’t be repeated, this is, we only have to declare them for one of the friends. And we have to take into account whether the relationship is already on the network or not; note that the relationships are reciprocal.
4. There cannot be relationships between the same person.

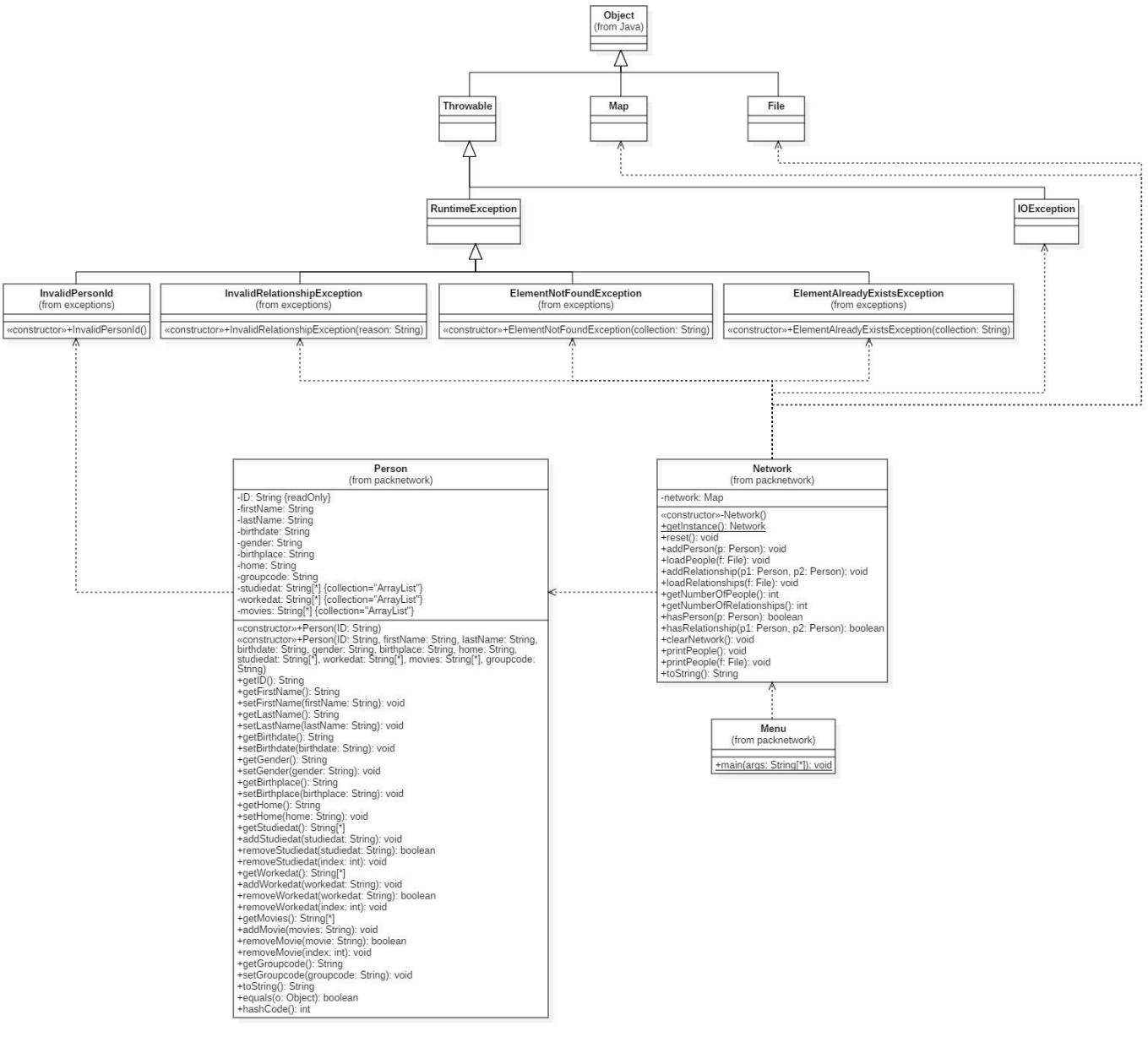
2nd Project Version

In this second approach, **we decided to change our data structure**. Researching on the internet, we have been looking for the best data structures to represent a network, the one with which we can define vertices and edges that represent users and relationships respectively.

We also started to work with **Github**, in order to have version control over our project and enhance the management of it.

Before starting to change code, we have ensured to have every class perfectly defined in a **UML diagram**.

2.1. CLASSES DESIGN



Since our package structure was a bit unclear, we decided to split the packages in two main parts: the **client part** and the **developer part**.

Package: packclient

Holds the **classes with which the client interacts with the social network**.

Menu

The unification of the previously named “InitialMenu” and “JFileChooserWindow”. It now contains the following options:

1. **Load people into the network**. Add people from several files to the network.
2. **Load relationships**. Add relationships between people from several files.
3. **Print out people**. Print out to an external file all the data about the users.
4. **Search users.** Added the option but not implemented in this 1st milestone.
5. **Print relationships**. Display relationships between people to the standard output.
6. **Quit**. Shut down the application.

Package: packnetwork

In this package we have defined the **data structures that the network implements**.

Network

Stays unchanged, but the data structure used to represent the object has been modified and will be explained in the following sections.

Person

Stays unchanged, but the data structure used to represent the object has been modified and will be explained in the following sections.

Package: packexceptions

ElementAlreadyExistsException

Represents the situation in which the target element is already in the target data structure.

ElementNotFoundException

Represents the situation in which a target element is not present in a collection.

InvalidPersonIdException

Represents the situation in which the person id is not valid, for one or more reasons.

InvalidRelationshipException

Represents the situation in which the relation inside the network is invalid for one or more reasons.

Package: packcheck

We have implemented this package in order to execute some **fast tests** while developing the main classes and methods. Note that there are not formal JUnit tests, but more like a testing ground to the recent implementations.

NetworkCheck

Used for testing the Network class:

- Create people
- Add people to the network
- Add relationships between 2 people
- Print network
- Print number of people inside the network

PersonCheck

Used for testing the Person class:

- Create people
- Print people
- Test every getter and setter
- Test equals override

Package: packnetworktests

The package where the **JUnit tests are stored**. Here we verify that every method in the *packnetwork* class is working as expected.

NetworkTests

JUnit tests to check Network class methods.

PersonTests

Junit tests to check Person class methods.

2.2. DESCRIPTION OF THE DATA STRUCTURES USED IN THE PROJECT

HashMap with Linked Lists. Adjacency list.

After a deep research about data structures related to social networks, we have found that a structure that simulates a **graph**[2] could be the most suitable option.

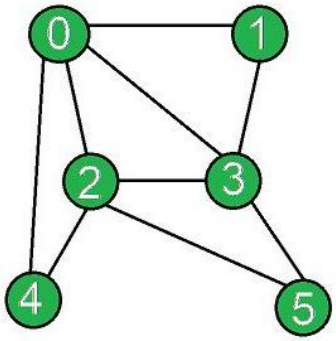
In our renewed structure, the **vertices**[3] of the graph **corresponds to users** inside the network, while the **edges are equivalent to relationships**.

In order to implement a graph, we thought about using some sort of structure that handles users and relates them to other users. In other words, some structure that defines keys and one or more values related to these keys. Fortunately, Java already has several interfaces and classes that can help us with these **key-value** implementations. We also have this implementation in egela. We could create our own definition of this structure, but it is not necessary to reinvent the wheel.

We have used the **HashMap class**, which implements **Map interface**. The **keys are formed by Person objects**, while the **values for each key are composed by lists of Strings, that correspond to other user id's**. For this part we have tried 2 different approaches.

Approach 1: adjacency matrix.

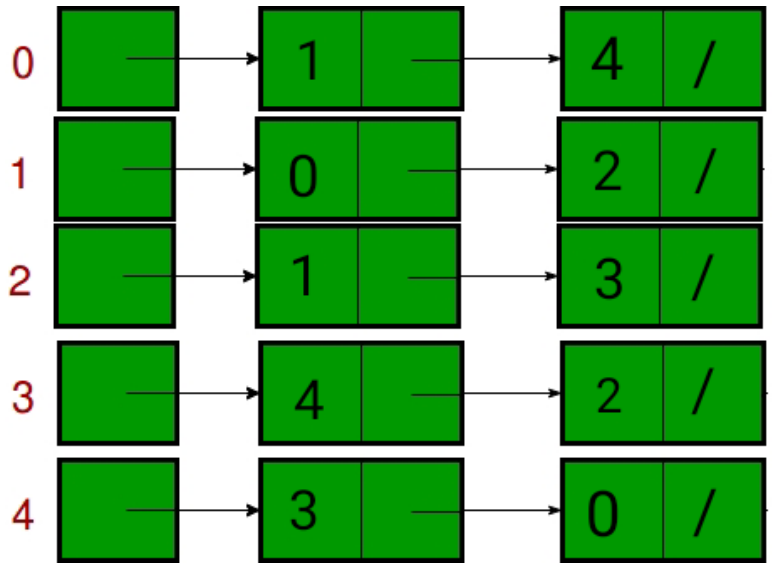
[2] "Implementing Generic Graph in Java - GeeksforGeeks." 7 aug. 2019, https://www.geeksforgeeks.org/implementing-generic-graph-in-java/. 25 oct. 2021.
[3] "Graphs in Java | Baeldung." 22 sept. 2020, https://www.baeldung.com/java-graphs. 25 oct. 2021.



	0	1	2	3	4	5
0	0	1	1	1	1	0
1	1	0	0	1	0	0
2	1	0	0	1	1	1
3	1	1	1	0	0	1
4	1	0	1	0	0	0
5	0	0	1	1	0	0

In this approach, the relationships are implemented in a matrix (M) of numbers. The dimension of M is equal to the number of people in the network. When a relationship is registered between x and y people, the matrix will be updated in the position M[i][j] and M[j][i] with the value 1, meaning that there is a relationship between them.

Approach 2: adjacency list.



The relationships in this case are stored in a linked list. Each key in the HashMap will be associated with a single linked list. Each element in the list means a relationship (edge or value) that a single user (vertex or key) has.

Before choosing one of the two, we have analyzed the time that most common operations could take in each case.

Operations	Adjacency Matrix	Adjacency List
Storage space	This representation makes use of VxV matrix, so space required in worst case is **O(|V|^2)**.	In this representation, for every vertex we store its neighbours. In the worst case, if a graph is connected O(V) is required for a vertex and O(E) is required for storing neighbours corresponding to every vertex .Thus, overall space complexity is **O(|V|+|E|)**.
Adding a person (vertex)	In order to add a new vertex to VxV matrix the storage must be increases to **(|V|+1)^2**. To achieve this we need to copy the whole matrix. Therefore the complexity is **O(|V|^2)**.	There are two pointers in the adjacency list: first points to the front node and the other one points to the rear node.Thus insertion of a vertex can be done directly in **O(1)** time.
Adding a relationship (edge)	To add an edge, say from i to j, matrix[i][j] = 1 which requires **O(1)** time.	Similar to insertion of vertex here also two pointers are used pointing to the rear and front of the list. Thus, an edge can be inserted in **O(1)**time.
Removing a person (vertex)	In order to remove a vertex from V*V matrix the storage must be decreased to |V|^2 from (|V|+1)2. To achieve this we need to copy the whole matrix. Therefore the complexity is **O(|V|^2)**.	In order to remove a vertex, we need to search for the vertex which will require O(|V|) time in worst case, after this we need to traverse the edges and in worst case it will require O(|E|) time.Hence, total time complexity is **O(|V|+|E|)**.
Removing a relationship (edge)	To remove an edge say from i to j, matrix[i][j] = 0 which requires **O(1)** time.	To remove an edge traversing through the edges is required and in worst case we need to traverse through all the edges.Thus, the time complexity is **O(|E|)**.
Searching a person	In order to find an existing edge the content of the matrix needs to be checked. Given two vertices, say i and j matrix[i][j] can be checked in **O(1)** time.	In an adjacency list every vertex is associated with a list of adjacent vertices. For a given graph, in order to check for an edge we need to check for vertices adjacent to a given vertex. A vertex can have at most O(|V|) neighbours and in the worst case we would have to check for every adjacent vertex. Therefore, time complexity is **O(|V|)**.

Since we are constantly adding and querying users, even if the querying time is favourable for the adjacency list, if we think of a scalable application, a data structure that needs to be resized every time we add a new user could be a critical slowdown. Also, the space needed for the adjacency matrix will grow faster than the linked list.

For the mentioned reasons, we decided to implement the values of the HasMap as linked lists of Strings, where each String corresponds to the id of one user.

Why HashMap over ArrayList?

Most of our queries will be done with user ids. So, instead of accessing our data structure by indexes, **we will probably be accessing it by some sort of key**, for example an user id or any other attribute of a person. Also, the **time complexities are in general more favourable for the Map**. We took in consideration the following comparison[4]:

Data structure	Add/put	Remove	Get	Contains
ArrayList	O(n) in given index	O(n)	O(1)	O(n)
LinkedList	O(1)	O(n)	O(n)	O(n)
HashMap	O(1)	O(1)	O(1)	Amortized O(1)

2.3. DESIGN AND IMPLEMENTATION OF THE METHODS

Every method has been correspondingly documented using Javadoc. Doc folder can be found in the project hierarchy. Consequently, every method is already described and commented step by step in the code. As mentioned above, the most important methods of Person and Network class have been verified using JUnit test units, which can also be found in the project folder.

[4] "Runtime Complexity of Java Collections - gists · GitHub." https://gist.github.com/psayre23/c30a821239f4818b0709. Se consultó el 25 oct. 2021.

3rd Project Version

The 3rd version of the project was focused on completing the tasks for the 2nd milestone. In order to achieve these objectives, **we had to adapt our previous data structure**. We have concluded that **the selected data structure is appropriate but it needs some modifications** in order to obtain a better performance.

3.1. CLASSES DESIGN

The milestone where we are working is mainly focused in the comparison and searching of people. So we will see that the changes made are related to this type of operation. The UML diagram has received the corresponding update, but we are going to show it in the final version of the project, which is the 4th one.

Package: packclient

Menu

It is possible to test the new implementations from the client menu, which includes the following new options:

1. **Search**. Since there are several search options, we have created a submenu in order to choose the type of search and filtering.
2. **Group by movies**. Returns a map with all people inside the network split in groups of people that share the same profile. Two users have the same profile if they match the same collection of favorite movies.

The client now has the option to choose between several search options:

1. **Friends by last name**. Search friends of people with a given surname.
2. **By city**. Search people by their birthplace.
3. **Between two dates**. Search people that were born between two dates.
4. **By residence file**. Given a file with a set of identifiers, returns people whose birthplace matches the hometown of the people who are described in the input file.

For every option that has an output, the client has now the option to **print the result to stdout or/and to an external file**.

Apart from all the new implementations, the code has been refactored so now each function is separated instead of being inside the whole *switch case* block.

Package: packnetwork

Network

We have modified the data structure used to represent the network. More information in *DESCRIPTION OF THE DATA STRUCTURE*.

Also, all the new methods implemented can be found in the *DESIGN AND IMPLEMENTATION OF THE METHODS* section and in the Javadoc, as always.

Person

This class remains the same in essence, but we have added some improvements for the performance of Person comparison and creation. Changes are described in *DESIGN AND IMPLEMENTATION OF THE METHODS* section.

PersonComparator

In order to perform custom comparison, we have created the PersonComparator class.

If we take a look at the assignments, it is quite clear that people are going to be **compared using different criteria**. In order to perform this comparison, we've created the following comparators:

- ***bySurname***. By ascending last name.
- ***byBirthplace***. By ascending order of birthplace.
- ***byBSN***. By ascending order of birthplace, last name and the first name.
- ***byBirthplaceAndHome***. This is a particular comparator that compares the birthplace of Person 1 to the home of Person 2.

3.2. DESCRIPTION OF THE DATA STRUCTURES USED IN THE PROJECT

2 HashMaps with Linked Lists (Adjacency List)

After the 2nd milestone, we still think that a Hashmap with a linked list is an appropriate data structure in order to simulate the graph that represents the social network - this is, an adjacency list, as explained in the 2nd version of the project-.

But there was a problem in finding a particular person by different criteria. The thing is that the keys of the HashMap were storing the whole Person object, while the relations between them were covered by the linked list in the values of the HashMap. But what if we want to search for a person by id?

The expected behaviour for a HasMap is to use a key to find a value, not to send an attribute of an object to find a key. There are no methods for that. So we decided to split our HashMap into 2 different maps.

The first map is called **people**, and has a String → Person structure. Keys are used to store people ID's, and values for the rest of the information of the Person object. In this way, we can find people by their ID in a time complexity of O(1).

The second map is **relations**, and as its name points, holds the relations between people. The structure is String → LinkedList<String>, where the keys conform to all the people in the social network and the LinkedList represents all the friends that a single user has.

There is no reason for creating our own version of a Map, so we decided that the most suitable option was to keep using Java HashMaps, but combined in this way so network searching is more efficient. As an alternative, Java also provides LinkedHashMap[5]. The main difference between both implementations is the ordering of elements. The LinkedHashMap provides a way to order and trace elements. Comparatively, the HashMap does not support the ordering of the elements. In LinkedHashMap, if we iterate an element, we will get a key in the order in which the elements were inserted. In our case, since the network does not follow any ordering patterns, we have decided to use the HashMap implementation, and also because it requires low memory, since the LinkedHashMap uses the same implementation process as HashMap but with an additional doubly LinkedList to maintain the order of the elements.

[5] "LinkedHashMap vs HashMap - Javatpoint." https://www.javatpoint.com/linkedhashmap-vs-hashmap-in-java. Se consultó el 12 ene. 2022.

3.3. DESIGN AND IMPLEMENTATION OF THE METHODS

There are several new methods that we have implemented in order to carry out the assignments of the 2nd milestone.

Network

- ***findPeople(Person personToCompare, Comparator<Person> c)***. Used to develop the assignments 6 and 7. Given a comparator and a Person object, returns all people that are the same for that comparator. This method is useful in order to retrieve all the people that have the same name, last name, home town... it will depend on what the comparator compares.
- ***findSortedPeopleBetweenDates(String date1, String date2)***. Used to develop the assignment 8 of the project. Given two dates, returns people IDs that were born between both dates.
- ***findPeopleBetweenDates(String date1, String date2)***. auxiliary function that performs the actual search for the *findSortedPeopleBetweenDates* method.
- ***findByResidenceFile(File f)***: used to develop the assignment 9 of the project. Given a file with a set of identifiers, returns people whose birthplace matches the hometown of the people who are described in the input file.
- ***FindFriends(String p)***. Used to develop the assignment 6. Given a person id, returns a list of its friend ID's.
- ***printPeople(LinkedList<String> peopleIDs, int[] attributes)***. Prints to the stdout the specified attributes of each person in the list passed by parameter.
- ***printPeople(LinkedList<String> peopleIDs, int[] attributes, File f)***. Prints to an output file the specified attributes of each person in the list passed by parameter.
- ***groupByMovies()***. Used to implement the assignment 10 of the project. Returns a map with people classified by profiles. Two users have the same profile if they match the same collection of favorite movies.
- ***toString(LinkedList<String> peopleIDs, int[] attributes)***. Returns a string representation of all the people in the given list. Only attributes passed in the second argument are taken into account.

Person

Person attributes that were defined as an array -this is, that have many elements, such as movies- **are now automatically sorted at Person instance creation moment and whenever we add a new value to the attribute**.

When the constructor is called, it will automatically sort all the values, as well as transform them to lower case.

```
this.studiedat = new ArrayList<String>();
if (studiedat != null) {
    for (String item: studiedat) this.studiedat.add(item.toLowerCase());
    Collections.sort(studiedat);
}

this.workedat = new ArrayList<String>();
if (workedat != null) {
    for (String item: workedat) this.workedat.add(item.toLowerCase());
    Collections.sort(workedat);
}

this.movies = new ArrayList<String>();
if (movies != null) {
    for (String item: movies) this.movies.add(item.toLowerCase());
    Collections.sort(movies);
}
```

If we try to add a new element to these attributes, we first check whether the element is already inside the array or not by using binary search. If the element is not present, the binary search method itself will give us the correct position of where the new element has to be inserted.

```
public void addWorkedat(String workedat) {
    int pos = Collections.binarySearch(this.workedat, workedat);
    if (pos < 0) this.workedat.add(-pos-1, workedat.toLowerCase());
}
```

Apart from this, we developed a new method in order to retrieve an attribute by its “numerical value” from the Person. Numerical values are specified in the Javadoc.

```
/**
 * Given an integer, returns the attribute
 * 0  => idperson
 * 1  => name
 * 2  => lastname
 * 4  => birthdate
 * 5  => gender
 * 6  => birthplace
 * 7  => home
 * 8  => studiedat
 * 9  => workplaces
 * 10 => films
 * 11 => groupcode
 * @param i The attribute number equivalent
 * @return The attribute value
 */
public String getAttribute(int i) {
    String result = "";
    switch(i) {
        case 0:
            result = getID();
            break;
        case 1:
            result = getFirstName();
            break;
        case 2:
```

Finally, we have added the default comparison for Person that will sort these objects lexicographically based on the Person ID.

```
public int compareTo(Person o) {
    return getID().compareTo(o.getID());
}
```

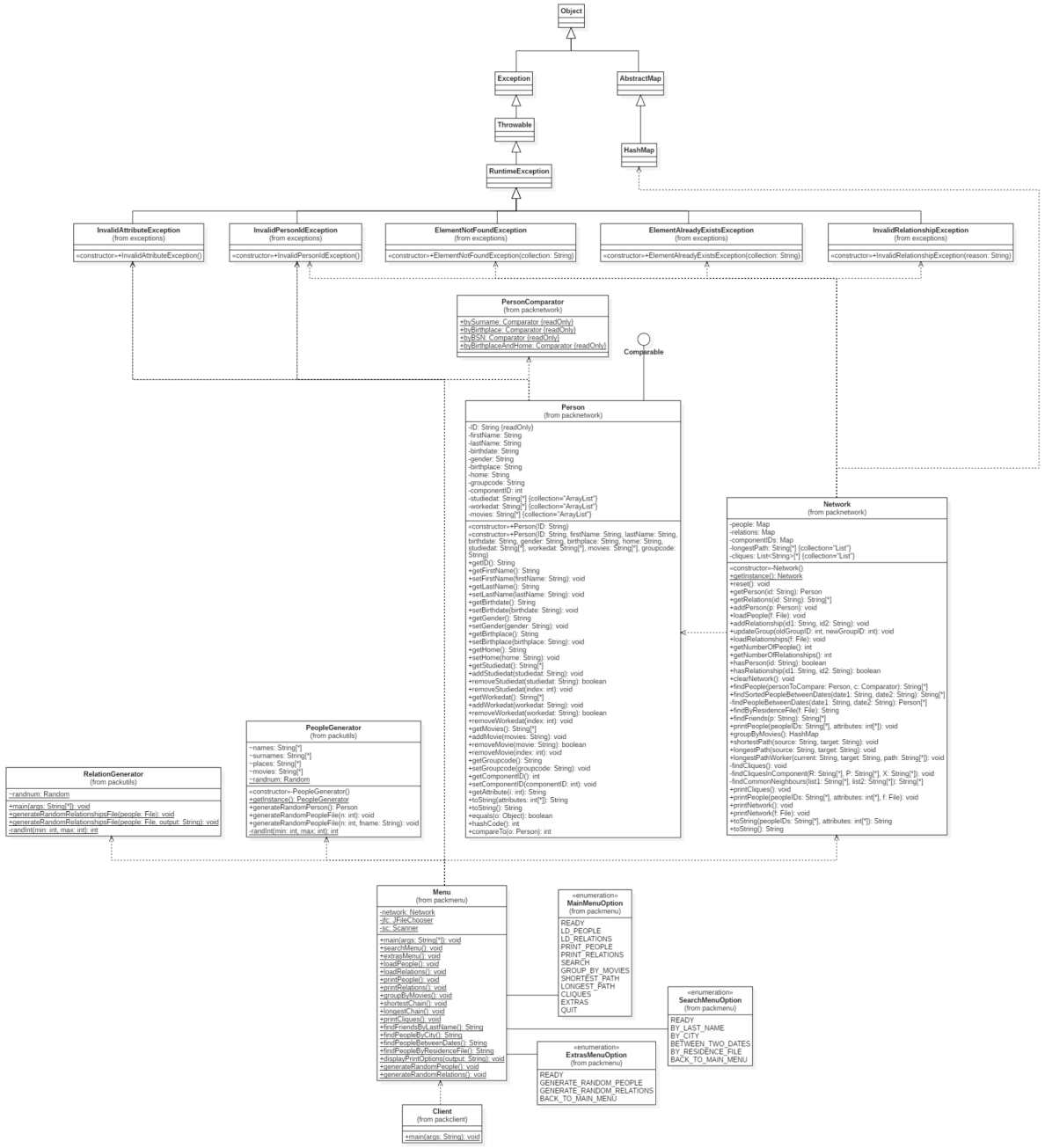
As always, all the methods are explained in detail in Javadoc format. Full documentation available in the */doc* directory.

4th Project Version

This last milestone is where the **structure of the graph takes more importance**. The assignments in which we have worked while developing this final version are the following:

1. Six degrees of separation is the theory that everyone on Earth is six or fewer steps away, by way of introduction, from any other person in the world, so that a chain of “a friend of a friend" statements can be made to connect any two people in a maximum of six steps (or five intermediaries). **Given two people in the network**, **retrieve the shortest chain that relates them**.
2. **Given two people**, **recover the largest chain of different people linking them** (duplicate intermediaries are not allowed). Use backtracking.
3. **Retrieve all the cliques of friends** (crews) **with more than 4 friends**. A clique is a group of friends in which each person has friendship with each other. Use backtracking.

4.1. CLASSES DESIGN



Taking into account that this was going to be the last version of the project, we have decided to isolate crearly the client side of the project from the backend or the developer side. Thus, our final package scaffolding looks as follows:

Package: packclient

Client

The viewpoint of the user. This is the class that contains the main method where the user can interact with the networks. It only contains the main method which **initializes the menu for interacting with the social network**.

The aim is to use the application without knowing the implementation itself.

Package: packmenu

Menu

This class holds the menu for the social network. Apart from displaying it in the standard output, it also contains all the methods used to interact with the network and some extra features (explained later).

The final look the menu contains three different sections:

1. Main menu: here we can observe this options:
    1.1. Load people into the network
    1.2. Load relationships
    1.3. Print out the people
    1.4. Print relationships
    1.5. Search
    1.6. Group by movies
    1.7. Shortest path
    1.8. Longest path
    1.9. Cliques
    1.10. Extras
2. Search menu (option 1.5)
    2.1. Friends by last name
    2.2. By city
    2.3. Between two dates
    2.4. By residence file
    2.5. Back to the main menu
3. Extras menu (option 1.10)
    3.1. Generate random people
    3.2. Generate random relationships

MenuEnums

Using numbers as menu options might be a bit confusing, so we decided to create 3 different **Enums** -one per submenu- in order to **improve the readability of the code**. The point of this is that we can use Enum in switch case statements in Java like int primitives, so each number of the menu corresponds to a constant inside the Enum.

```
try {
    option = MainMenuOption.values()[sc.nextInt()];

    switch(option) {
        case LD_PEOPLE:
            loadPeople();
            break;

        case LD_RELATIONS:
            loadRelations();
            break;

        case PRINT_PEOPLE:
            printPeople();
            break;

        case PRINT_RELATIONS:
            printRelations();
            break;
```

Reading the code using meaningful constants rather than apparent nonsense integers makes the readability much better.

Package: packnetwork

Remains the same as the last version, but we have implemented the needed methods for the assignments in the Network class. We will see them in the DESIGN *AND IMPLEMENTATION OF THE METHODS* section.

Package: packutils

The aim of this package is to give to the extra features we have created a place where to be listed. We will also talk about these new features below, in the ADDITIONAL *FEATURES* section.

4.2. DESCRIPTION OF THE DATA STRUCTURES USED IN THE PROJECT

Thanks to the research made in the second version of the project, **the implemented data structure seems to work very well** for our objective of creating a social network.

We have been discussing how we can improve our data structure, particularly for some searching operations that do not use the person id as input parameter. Our alternative was to **implement several Maps where the keys of the maps were values of a certain attribute, and the values were those people who had the map key as the attribute**. The structure would look like String → String[], where the key is the value of a particular attribute, and the value of the group of people that matches that value in that attribute.

For instance, a Map for the surname attribute would look like this:

Map<String, LinkedList<String> surnameMap = new HashMap<String, LinkedList<String>();

And adding an entry:

List<Person> people = new LinkedList();

List.add("person_id1"); List.add("person_id2");

surnameMap.put("surnameX" , people);

So each time we want to find a set of people by a specific attribute, the time complexity of these operations would be most of the time of O(1).

The biggest disadvantage of this structure is the space complexity. We would be defining a Map for each attribute and each of them would take a maximum of N entries. So, even if in a social network the time complexity has more priority than the space complexity, we have concluded that it was excessive to use that much memory. So we have adapted the code to make as few O(n) searches as possible, but maintaining the structure that we had until the moment.

For the implementation of the last point of the project, we have realized that 2 people might be totally isolated in the network. In other words, the graph might have disconnected components. So we have been thinking in an **efficient way to identify the component corresponding to each person in the network**.

Firstly, we have a Map to relate each component with the set of people that are inside that component. The structure follows the Integer → String[] pattern, where the key is the component identifier and the values are the Person identifiers that make up that component.

```
// Map to relate component id to each member of the component
private Map<Integer, List<String>> componentIDs;
```

Now, Person objects have a new attribute which identifies the component where he/she is. This attribute is the *componentID*. When a new Person is added to the network, it has no relationships, so it creates a new component.

```
// Assign a component to the person
p.setComponentID(componentIDs.size()-1);

people.put(p.getID(), p);
relations.put(p.getID(), new LinkedList<String>());
```

When a relationship between two people is added, we have to compute the new component if two different components are joined.

```
    if (p1.getComponentID() != p2.getComponentID()) {
        int newGroup = Math.min(p1.getComponentID(), p2.getComponentID());
        updateGroup(Math.max(p1.getComponentID(), p2.getComponentID()), newGroup);
    }
}


/**
 * Joins two components when a new relation is added and both people where
 * from different components
 * from different component
 * @param oldGroupID the component identifier to be removed
 * @param newGroupID the component identifier that will identify the
 * previously isolated components
 */
public void updateGroup(int oldGroupID, int newGroupID) {
    // change the component id property of each person in the old group
    for (String pId: componentIDs.get(oldGroupID))
        people.get(pId).setComponentID(newGroupID);

    // move the old group to the new one in the map
    componentIDs.get(newGroupID).addAll(componentIDs.get(oldGroupID));

    // remove the old group (duplicated) from the map
    componentIDs.remove(oldGroupID);
}
```

Our method is based on taking the component with the biggest integer as an identifier and passing it to the entry corresponding to the component with the smallest id between both components. In that way, the previously isolated components are now successfully joined with the same component id.

4.3. DESIGN AND IMPLEMENTATION OF THE METHODS

As previously mentioned, every method is accordingly documented in the Javadoc. But in this case, since the implementation is a bit more complex, we will explain the methods used for the different assignments of the last milestone.

1. Given two people in the network, retrieve the shortest chain that relates them.

We have followed the common pattern to find the shortest path in a graph, that is using Breadth First Search, iterative in our case. Basically what we do is to run BFS in the adjacent nodes until we find the target. While we are adding the adjacent elements to the queue, we also save an entry in a Map in order to keep track of the path we are building, so the key of the map is the child node and the parent node is the value.

```
// add friends to the queue if not added yet
for (String adj: relations.get(parent))
    if (!edgeTo.containsKey(adj)) {
        // add friend to the queue
        q.add(adj);
        edgeTo.put(adj, parent);
    }
```

In that way, after having found the target element, we can traverse the map starting from the target to obtain the reversed path. This can be trivially reordered using a Stack.

It is remarkable to say that we do not need an auxiliary structure to track the visited nodes. It is enough to check if the mentioned Map contains the adjacent node as a key.

2. Given two people, recover the largest chain of different people linking them

This second assignment is a bit more complicated than the previous one. Taking into account that our graph is undirected, this problem is NP-Hard[6] and the decision version of the problem, which asks whether a path exists of at least some given length, is NP-complete. Of course, this problem has a linear solution if we are talking about an undirected acyclic graph, which is not our case.

So our algorithm is based on calculating all the possible paths, and retrieving the longest path. Of course, having in mind the time complexity of performing DFS recursively -and space complexity of storing all paths-, we have tried to optimize the algorithm as far as we could.

[6] "Longest path problem - Wikipedia." https://en.wikipedia.org/wiki/Longest_path_problem. Se consultó el 12 ene. 2022.

What we do is create a path and pass it by parameter while using recursion. For each node, we try every possible adjacent, and if there are no more possibilities to develop the path, then we delete that node from the current path, this is, using backtracking. If the target node is found, then we compare the current path with the longest path, and we save only the longest path between both. In that way, we save a noticeable amount of memory. Apart from that, if we find a path that is equal to the number of vertices, then we exit the function because there is not going to be a longer path.

3. Retrieve all the cliques of friends (crews) **with more than 4 friends**.

For the last point of the project we have created our own version of the Bron-Kerbosch[7] algorithm in order to find maximal cliques so it only finds cliques of more than 4 people. Found cliques are stored in an instance variable called *cliques*.

The used algorithm can be optimized using pivoting. This involves forgoing pivoting the outermost level of recursion, and instead choosing the ordering of the recursive calls carefully in order to minimize the sizes of the sets *P* of candidate vertices within each recursive call.

[7] "Bron–Kerbosch algorithm - Wikipedia." https://en.wikipedia.org/wiki/Bron%E2%80%93Kerbosch_algorithm. Se consultó el 12 ene. 2022.

Additional features

RANDOM DATA GENERATOR

In order to test network functions with more realistic samples, we have created a random data generator. This can be found, as mentioned before, in the *packutils* package. The generator is separated into two different classes.

PeopleGenerator

Using several files as databases, performs a random generation of people data. The number of people can be requested by parameter. The generated people are firstly stored in a Set in order to ensure that no duplicate users are created. Then, a *people.txt* is automatically created with the file name passed by parameter (or *randomPeople.txt* if no file name is given).

The used databases can be found in the root */db* folder.

RelationGenerator

Given a filename with people data, generates a random number of relationships. Once again, the information is first processed inside a Set to ensure that no duplicate relations are created. Then, a *friends.txt* file is created with the file name passed by parameter (or *randomFriends.txt* if no file name is given).

This random data tends to generate a dense network rather than sparse network. In our implementations, that means that the longer path is going to take probably less time to be computed than usual, and finding cliques will take more time.

ANALYSIS OF NETWORK PERFORMANCE

Having all the features implemented, it is the moment to check the performance of our network, so we have tested different network situations, one with less people and sparse, and another one with more people and dense distribution.

In order to get the times, we have added some timers to the code, then can be found commented. The reader can freely prove the results obtained in the following table by uncommenting the lines of code that contain the timers.

17 people - 32 relationships			**150 people - 5658 relationships**		
Load people		**0.002 s**	Load people		**0.005 s**
Load relationships		**0.001 s**	Load relationships		**0.015 s**
Print out people		**0.002 s**	Print out people		**0.03 s**
Print relationships		**0.002 s**	Print relationships		**0.08 s**
Group by movies		**0.003 s**	Group by movies		**0.017 s**
Shortest path		**0.001 s**	Shortest path		**0.001 s**
Longest path		**0.002 s**	Longest path		**0.013 s**
Cliques		**0.001 s**	Cliques		**18.777 s**
Searches	Friends by last name	**0.0001 s**	Searches	Friends by last name	**0.002 s**
	By city	**0.0001 s**		By city	**0.001 s**
	Between two dates	**0.001 s**		Between two dates	**0.001 s**
	By residence file	**0.001 s**		By residence file	**0.002 s**

Conclusions

We have relied mainly on several internet resources in order to select a proper data structure. Some sources worth mentioning are GeeksForGeeks[8], Lewis and Chase's "Designing and Using Data Structures"[9] book, Javatpoint[10] and Medium[11].

We also have acquired a solid knowledge in terms of data structures thanks to the presential classes and laboratories.

The workload has been splitted equally for each member. We have focused the time spent in laboratories mainly for meetings and project decisions and implementing the code. The hours spent outside university were mainly aimed at research, testing and documentation.

On the technical side, we have understood how a social network works at the level of data structures and algorithms. We have learned how to compare social network parameters thanks to the definicion of several comparators, we have researched different graph traversing methods, data organization, algorithms and their respective optimization and so on.

And the most important thing, we have realized the importance of having a solid base on which to structure a project on such a large scale, and the consequences it has not only on the memory space occupied, but what is more important on a social network, which is the execution time.

The project has been developed with a meticulous version control thanks to Git. You can find the repository of the project in https://github.com/Botxan/Social-Network.git.

[8] "GeeksforGeeks | A computer science portal for geeks." https://www.geeksforgeeks.org/. Se consultó el 25 oct. 2021.
[9] "Lewis & Chase, Java Software Structures: Designing and Using ...." https://www.pearson.com/us/higher-education/program/Lewis-Java-Software-Structures-Designing-and-Using-Data-Structures-4th-Edition/PGM321497.html. Se consultó el 25 oct. 2021.
[10] "Javatpoint: Tutorials List." https://www.javatpoint.com/. Se consultó el 12 ene. 2022.
[11] "Medium – Where good ideas find you.." https://medium.com/. Se consultó el 25 oct. 2021.

Annexes

Annex A. UML Diagram.

Annex B. Input files used for testing.