Isabel Martínez Gómez

Algorithm and Complexity

2019-2020

PROBLEM 1

Report with the implementation of Problem 1 explained, justification of why the implementation has been chosen and finally some examples with the correct operation of the implementation.

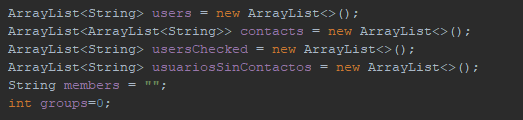
SOLUTION EXPLAINED

**VORACIOUS ALGORITHM**

This is the implementation of the voracious algorithm.

Firstly, I will explain what the main data structures are, and what they are used for:

* **users** is an ArrayList that contains all the users that are part of the social network ThinksThat ([www.thinksthat.com](http://www.thinksthat.com)).
* **contacts** is an ArrayList that contains as many ArrayList as number of users inside and each one contains the contacts of an user.
* The algorithm starts with an empty ArrayList, **usersChecked.** In each iteration, all the users that are in the same group will be added to the list.
* **contactlessUsers.** It is an arraylist that saves users who do not have contacts. This is done so that for each user who does not have contacts, the group counter is not increased if it has not been previously added to the usersChecked list.
* **members** is a string that we will use to show the members of each group in the output file.
* **groups** is a field of type int that counts the total groups there are in the social network ThinksThat.



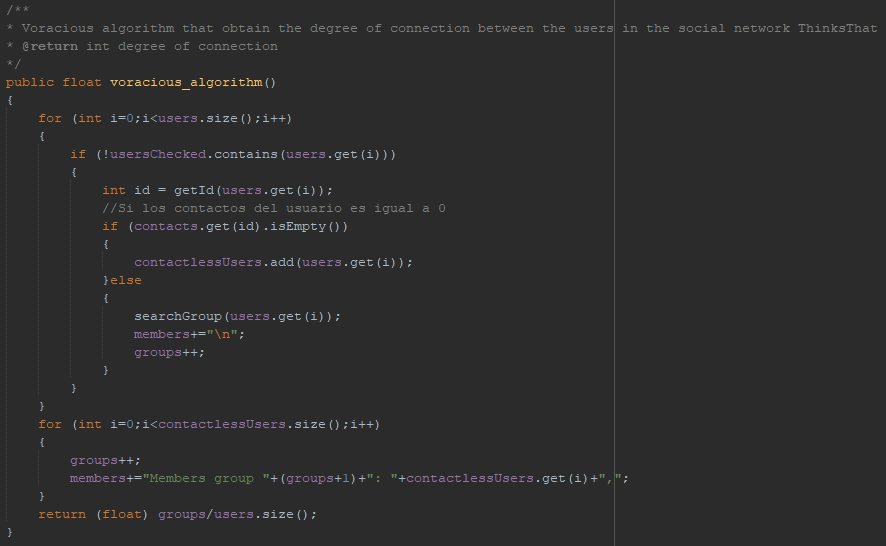
Once commented the data structures and attributes, I will explain how the algorithm works. The implementation of the voracious algorithm is supported by another function called searchGroupsthat I will explain later.

The algorithm goes through users looking for a group of connected users. If the ArrayList userChecked contains the user that we are looking at, we will look at the next one because we have already checked it and added to the ArrayList userChecked so it’s already in some group previously looked at.

On the other hand, if userChecked doesn’t contain the user then it is checked if that user has contacts or not:

* If the user doesn’t have contacts, we add it to the contactlessUsers arraylist and we look at the following user. This is done so that the number of groups doesn’t increase by one since it may be some user who has this user as a contact and therefore, they would be in the same group, but not in different groups.
* On the other case, we call the searchGroup function that will end when it finds a group of connected users. Then, the groups counter will increment by one since a group has been found.

The algorithm ends when all users inside the ArrayList users have been checked. It is at this moment when we go through the contactlessUsers arraylist in which we will look at the contacts it contains, since they are not connected with any other user in the social network. That is, the user doesn’t have any contact and no user has him as a contact, so we increase the groups counter for each user within this array.



**SEARCHGROUPS**

The function searchGroups searches for a group of connected users.

First, the function gets the name of the user as a parameter and adds it to usersChecked and to the string members so that we can write it later in the output text file.

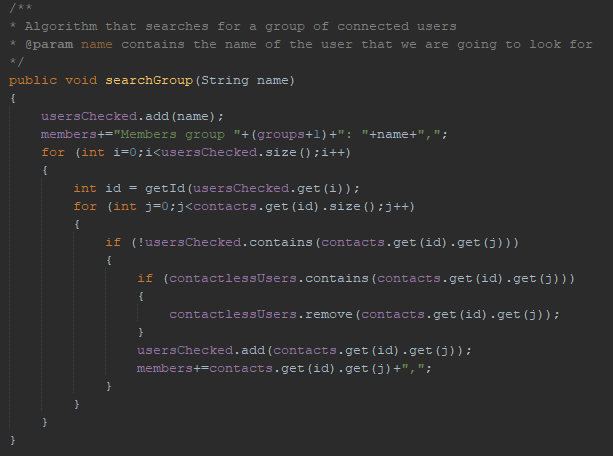
Then, we go through usersChecked to look one by one the contacts that the user checked has since they will be part of the group we are looking for.

Inside the nested for loop, we check if the user's contact is not already in the ArrayList usersChecked. If it is not, we will add it and if it is, we will not add it because we have already looked at it before and we will look at the next contact of the user. Also, if it is not in usersChecked, we will add the contact in the string members so that we can then write the information to the output file.

In case that we haven't looked at the contact before:

* We check if the contact of the user we are looking at, is inside the contactlessUsers arraylist. If it is inside, we remove it from the arraylist since it will belong to the group we are looking at and we add it to the usersChecked arraylist.
* If the contact we are looking is not inside the contactlessUsers we will just add it to the usersChecked arraylist.

The function searchGroup ends when all the users of a same group are added into usersChecked.



**GETID**

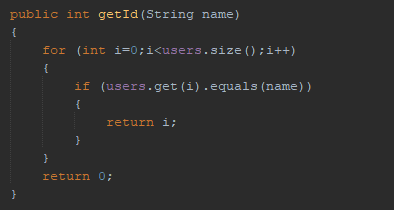
To understand how to get the contacts of a user since they are in different arraylists I present an example. For example, if we have:

users 🡪 [1,2,3,4,5,6]

contacts 🡪 [[2,3],[3,6,7],[1,4],[9,10],[1,2],[10]]

The contacts of user 1 are [2,3], of user 2 [3,6,7], of user 3 [1,4] and so on, so to get the contacts of a user we use the function getId() which I explain below.

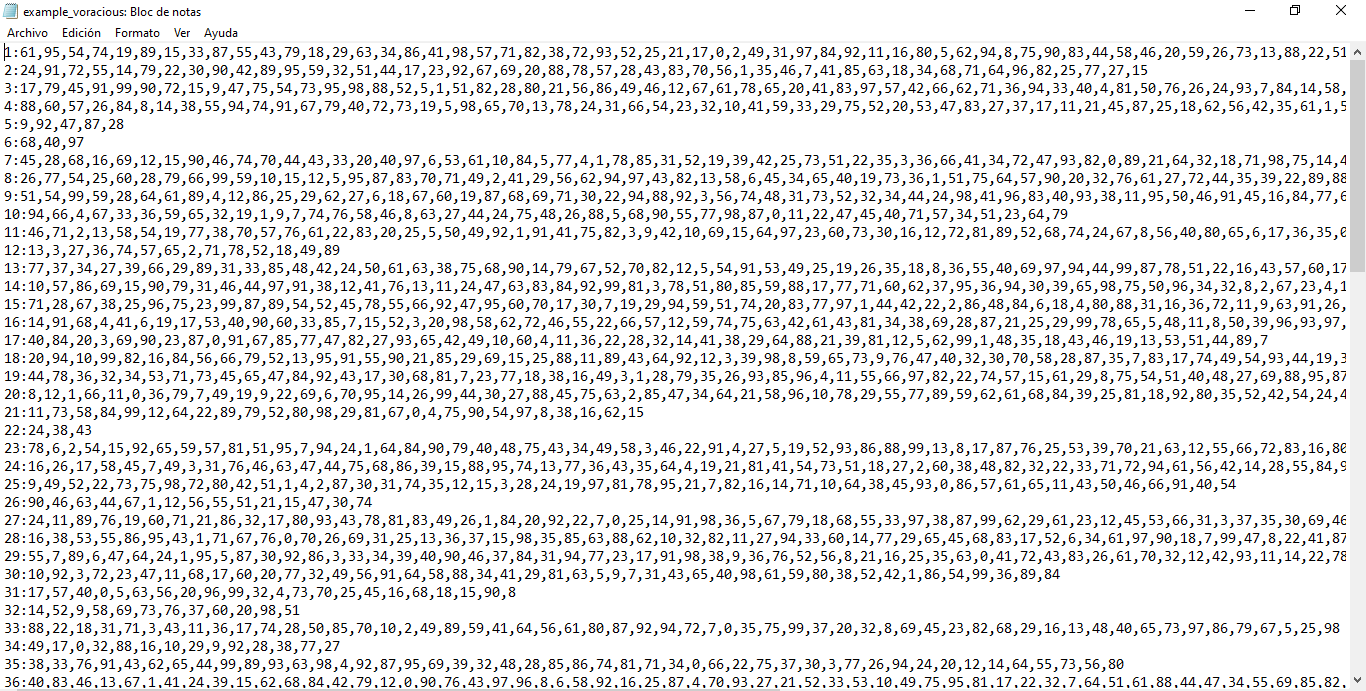
The purpose of this function is only to return the user's position in the arraylist users since it will be the same position in the arraylist contacts.



**INPUT FILE**

Once explained the implementation of the voracious algorithm, I will explain how I collect the data from the input file and how I write the results in the output file.

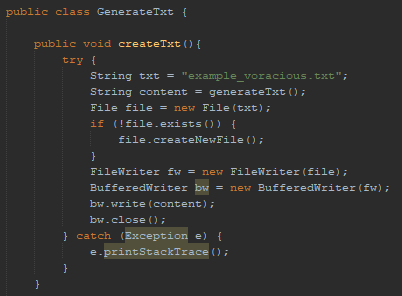
The style of the input file “example\_voracious.txt” is the following:



To generate the input txt file, I have used the GenerateTxt class that generates 100 users and assigns each one a random number of contacts. I have saved the generated txt in the folder of the exercise code in the generateTxtExample.txt file in case you want to test it, you will have to copy and paste it into example\_voracious.txt

**CREATETXT**

This function creates the file if it doesn’t exist and if it exists, writes the content that we generate. To generate the content, we use the generateTxt and generateLine functions that I will explain below. Once the content is written, the file is closed.

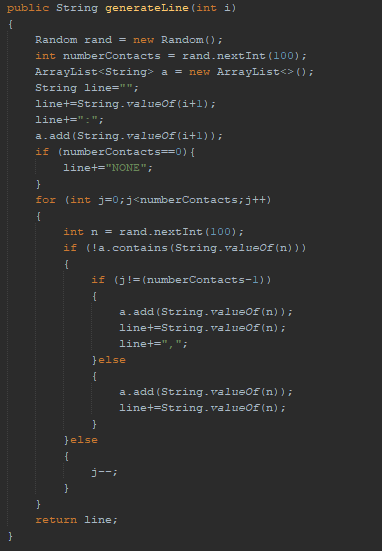


**GENERATELINE**

To generate each line of the file I have created this function, what I do is put a format of:

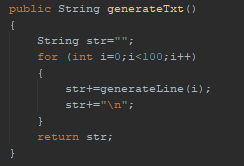
*Name of the user:contact1,contact2,contact3...*

Therefore, I generate random contacts for the user as well as a random number of contacts.



**GENERATETXT**

This function creates 100 lines so, by default I will have 100 users in the generated input file.



**INITILIZEFROMTXT**

This function reads from the input file “example\_voracious.txt” and initialize program data structures with the data in the txt. The file is read by FileReader and BufferedReader.

As we have seen, the format of a line of the input file is:

*Name of the user:contact1,contact2,contact3…*

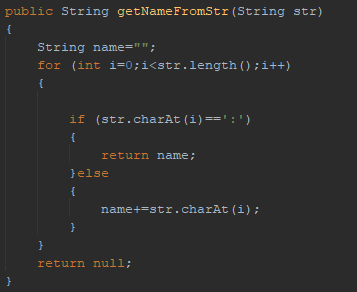
In this way, I create a function getNameFromStr to obtain the name of the string str and another getContactsFromStr to obtain the user's contacts. Once we have the name of the user and the user’s contacts, the users arraylist and the contacts arraylist are initialized.

The function gets the name of the file by parameter.

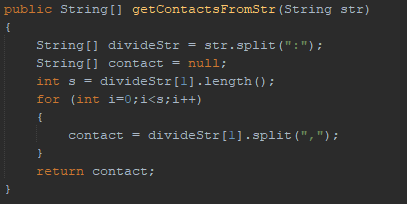


These are the previously mentioned functions that are used to initialize the program's data structures.

getNameFromStr receives a line from the file as parameter and returns the name of the user.



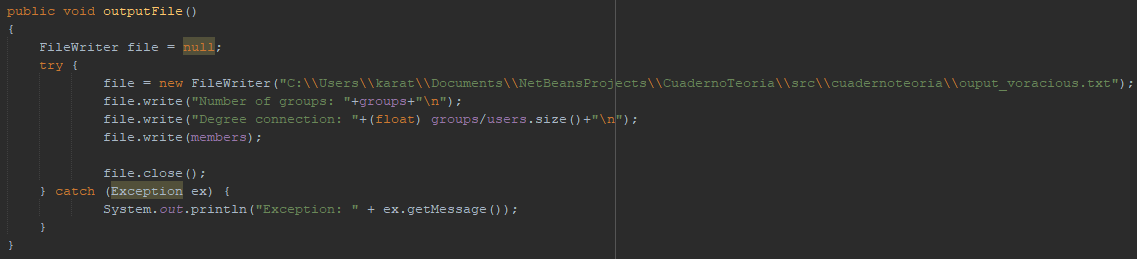
getContactsFromStr receives the same line as getNameFromStr as parameter and returns an array with all the contacts of the user so that later this data is added to an arraylist in the function initializefromtxt and then the arraylist, to the arraylist contacts.



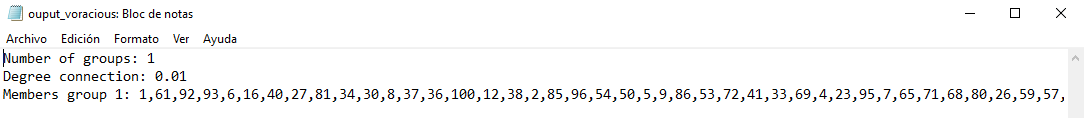
**OUTPUT FILE**

**OUTPUTFILE**

This function writes the result of the problem to it using FileWriter. It writes the number of groups, the members of each group and the degree of connection.

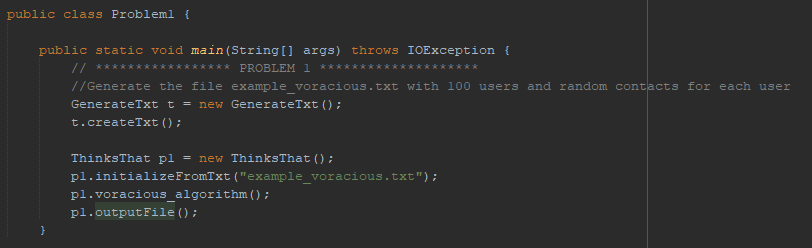


This is an example of the output file “output\_voracious.txt”:



**MAIN FUNCTION**

First, we create the input file “example\_voracious.txt” with 100 users and random contacts for each user and then we solve the problem with the functions initializeFromTxt, voracious\_algorithm and finally, outputFile to write the results in “output\_voracious.txt”.

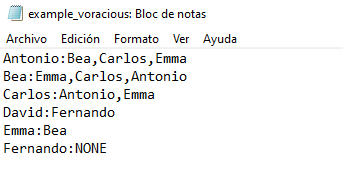


EXAMPLES

**FIRST EXAMPLE**

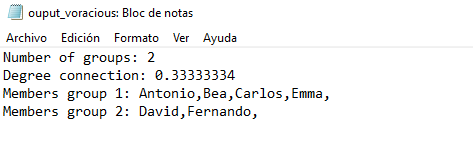
First, I will try the example of the statement

**INPUT FILE**



**OUTPUT FILE**

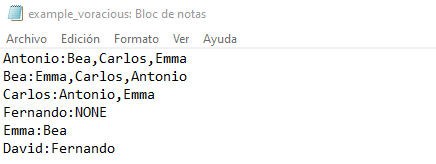
As we can see there are two groups, the first formed by Antonio, Bea, Carlos and Emma and the second formed by David and Fernando.



**SECOND EXAMPLE**

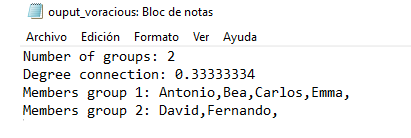
If we now reverse the order of the lines David and Fernando:

**INPUT FILE**



**OUTPUT FILE**

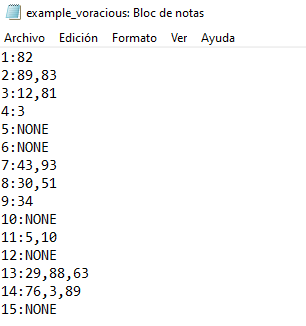
As we can see, although Fernando is in the input file before any user has him as a contact, after looking at the user David and David having Fernando as a contact, they are added to the same group.



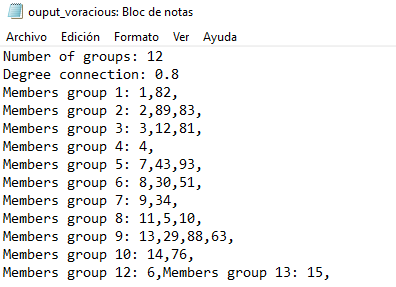
**THIRD EXAMPLE**

**INPUT FILE**

In this example I generate 15 contacts with a random number between 0 and 3 contacts.



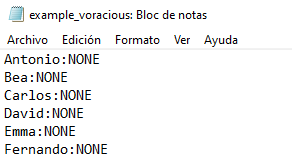
**OUTPUT FILE**



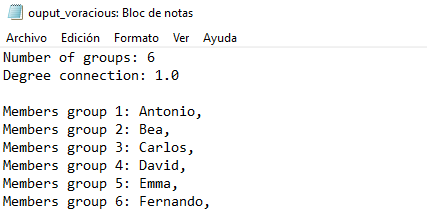
**FOURTH EXAMPLE**

**INPUT FILE**

If we now make the users have no contact, the degree of connection should be 1, since there are 6 groups and 6 users.



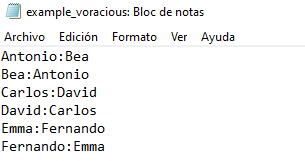
**OUTPUT FILE**



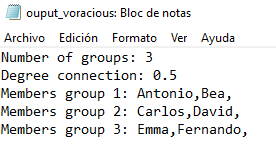
**FIFTH EXAMPLE**

**INPUT FILE**

Now we are going to make the users connected in pairs. Being in pairs of Antonio-Bea, Carlos-David and Emma-Fernando, there should be 3 groups and therefore 3/6 degree of connection



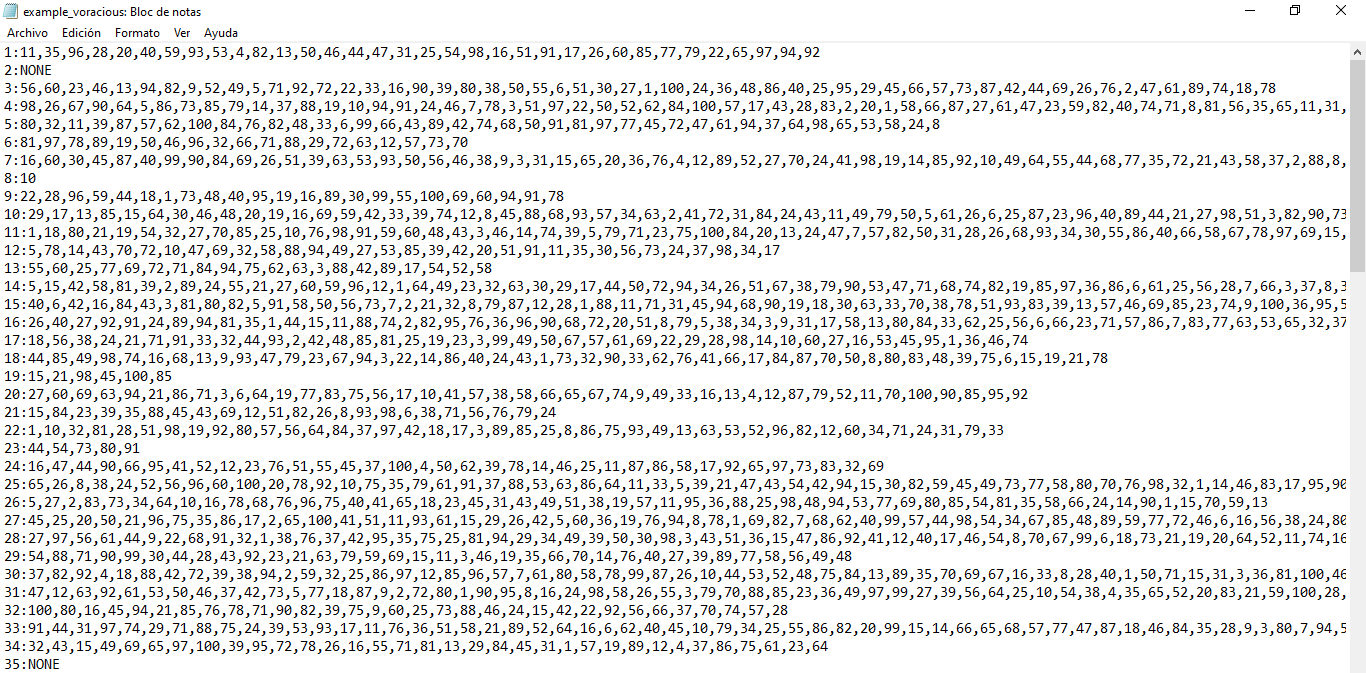
**OUTPUT FILE**



**SIXTH EXAMPLE**

**INPUT FILE**

Now we will generate an input file in which there are 100 users and each user has a random number between 0 and 99 contacts and those random and non-repeating contacts. Let's see how many groups come out.



**OUTPUT FILE**

As we can see, only a group of connected users come out, this is logical since there are users who have many contacts and then there is very little probability that any user is isolated.

