a97540 Ana Rita Moreira Vaz
a95114 Emanuel Lopes Monteiro da Silva
a95536 João Ferreira Longo

Guide Three Report

Laboratórios de Informática III

Introduction

The program has as inputs tree files that have the information of users, repositories and commits.
In the first guide, it was asked to examine the lines of all files, counting the number of wrong ones and creating new files with only the right ones.
In the second, it was asked to execute some queries and create output files with the specific result.
Now, in the third guide, it was asked to make the interface, tests of the queries, and data management.
There are four objectives in this guide. First, it's to consolidate the modularity and encapsulation concepts applied in the second guide. Second, it's to include new mechanisms between program and user. Third, to manipulate and consult a large volume of data. Lastly, it is to perform functional tests and performance evaluation.

Motivation

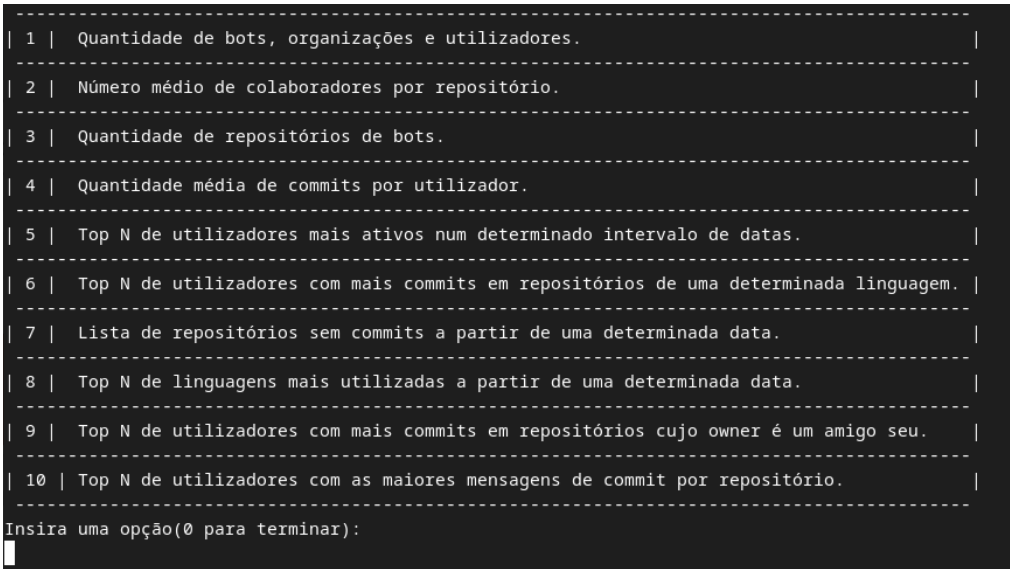
This project was planned to deal with data from a program that we use every day such as GitHub. We had to verify all data contained in the 3 files and manage that in order to create some useful features. It also prepared us for a future job where we must think about what the customer will see and how to deal with what he may require.
These kinds of projects gave us more motivation compared to previous projects because we also had to deal with a lot of information such as 10 million users which is more appropriate to the real world. To do it we had to apply different concepts of programming such as modularity and encapsulation in order to allow other programmers to manipulate the code and avoid some rookie mistakes and also to permit the best way to implement new features without modifying what was made.

Interface

In this part of the project, we had to make an interface to display what the users would be able to do. Although this mechanism of interaction should only appear when file commands were not given.

Once the user doesn't insert the commands' file, the program starts to verify the input files with the information about the users, commits, and repositories, as it does when receiving the input commands' file. In this process, it shows the faulty users, commits and repositories that is finding in the input files.
After this common process instead of calculating all the output files according to the commands' file, it will only process the file of queries 1, 2, 3, 4, 9 and 10 because they always have the same information to display, and we can gain some time only reading this files instead of calculating them every time that they are needed. As opposed to these, the information from queries 5, 6, 7 and 8 will be calculated at the moment of displaying it accordingly with users input.

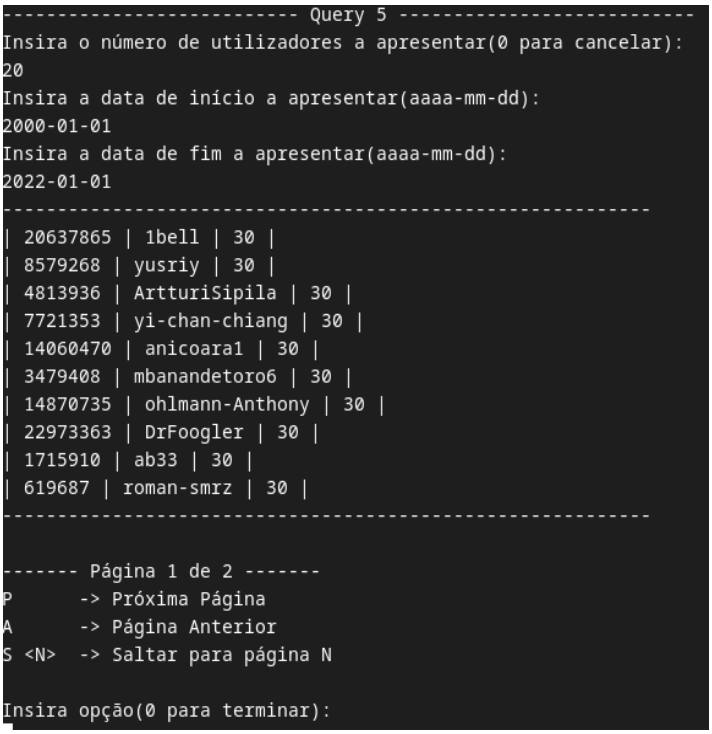
Right after these calculations, the output files, without any errors at this stage, are going to be cataloged so they can be used in the process of queries that the user may choose. The terminal will be cleaned and will also display a menu with all the available options for example the number of queries and a brief explanation of what the user can expect from each query. To collect the option from the user, we are using a scanf function that will work with a switch case function and run the interface for the query chosen.



Once the choice is made by the user, on queries 1, 2, 3 and 4 there is going to be displayed all the information that was written in each file without any more interference from the user.

However, on query 5 will be asked how many users the user wants to be displayed in the terminal within a certain space-time. To accomplish this requirement there will be a space for the user to be able to write the date for the beginning of the presentation and another space for the end of the presentation.

With these specifications, the function query 5 will run with data written by the user as it would run if was given a commands' file. When the processing of query is completed it is going to be stored in the array of structs in order to present a sub-menu which will display the number of users that the user witted, although if these number is bigger than 10, the remaining users will only be displayed if the user press P to print the next page or S <number> to print one page that the user may want to be presented instead of the next one. If the user wants to return to that main menu, it's only required to press 0 instead of P, S <number> or A which prints the previous page.



In the rest of the queries, a similar menu will be displayed and also will be requested the arguments needed as it was in query 5.

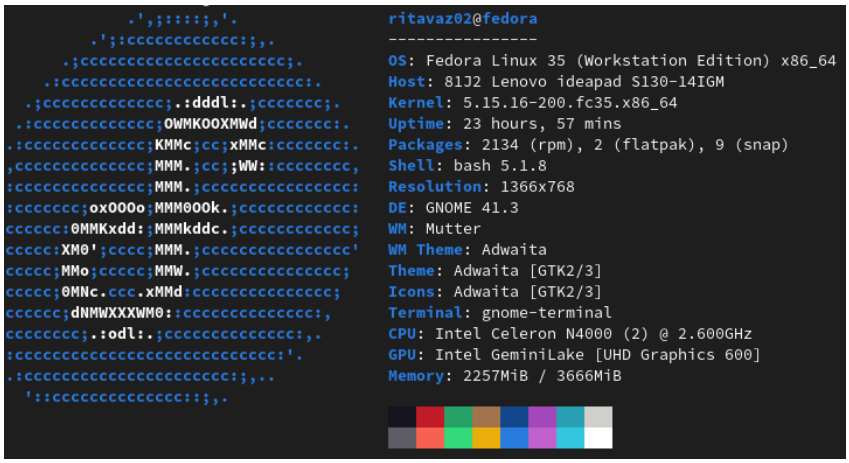
Unlike the other queries, in query 10 we decided to store the information a little bit differently. Instead of storing the information of each query in an array of structs we made a catalog of linked lists where each list would have the information about a specific repository. These types of storage were chosen to allow us to print a specific repository and let the user quickly know about the top N of users of each one.

In order to support these functions were created some auxiliary functions to create and to free the memory used by the struct of queries.

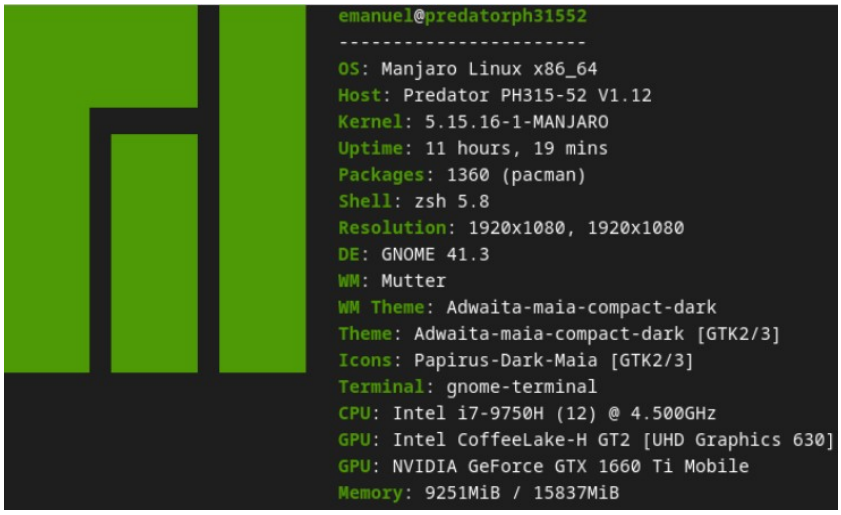
Tests

It was considered three different machines:

(1) Characteristics of Rita's machine and results:



(2) Characteristics of Emanuel's machine:

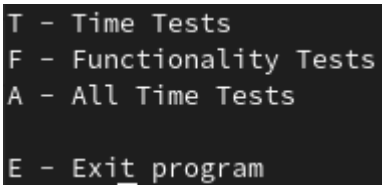


(3) Characteristics of João machine:

Tests Procedure:

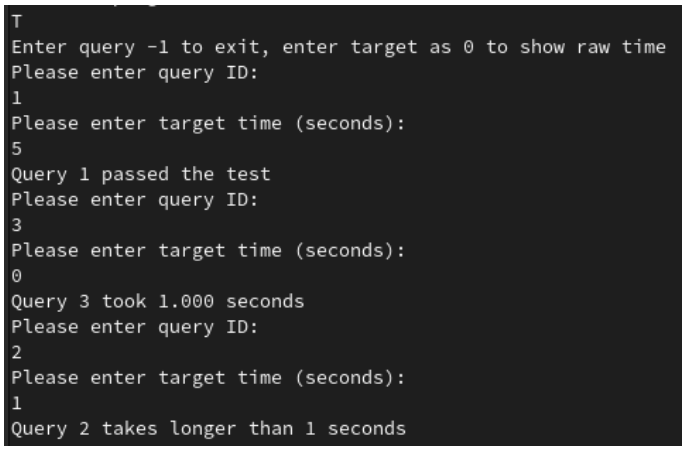
To make the executable tests independent of the executable guiao-3, guide 1 is executed in the initial of the program. This way, the wrong lines are eliminated, and the cats can be created.

There are four options present when the executable start:



If the user choices to run the time tests, it appears one message asking which one the user wants to verify and the target time. There are three types of answers: “Query x passed the test”, “Query x took y seconds” or “Query x takes longer than x seconds”.

For example,



Another option available is testing the result of a query. To do that, the output of the query (file test.txt) is compared with the expected output. Note that the files with the expected output of each query are in the folder “entrada”, and they are the result of running the file commands.txt in guide 2.

So, the tests that are realized are exactly the ones that are in the commands.txt, the difference being that they are the result of guide 3.

The results can be seen in the next picture:

```
F
Write the query you want to test, -1 to go back
1
Query has failed
Write the query you want to test, -1 to go back
2
Query is working as intended
Write the query you want to test, -1 to go back
3
Query is working as intended
Write the query you want to test, -1 to go back
4
Query is working as intended
Write the query you want to test, -1 to go back
5
Query is working as intended
Write the query you want to test, -1 to go back
6
Query is working as intended
Write the query you want to test, -1 to go back
7
Query has failed
Write the query you want to test, -1 to go back
8
Query is working as intended
Write the query you want to test, -1 to go back
9
Query is working as intended
Write the query you want to test, -1 to go back
10
Query has failed
```

The query 1 failed because guide 2 considered one more user with the type User, probably the result of reading the last white line of the intermediate file.

Query 7 failed because the comparison is done line by line. As the result of the query 7 is printed in the file using a hash table, each time is printed in a different order.

Query 10 failed because we use the file with the result of query10 as an intermediate file to do the interface. So, there are white lines between each different repository.

Finally, the other option besides the “exit” one, is to run all-time tests. It is asked how many times the user wants to run the tests. Then, for each query, it is removed the longer and shorter time that a specific query takes. The result is the average with the remaining times.

The results are for each machine are:

Machine (1)

```
Query 1: 0.625
Query 2: 2.750
Query 3: 0.250
Query 4: 0.000
Query 5: 1.000
Query 6: 0.750
Query 7: 2.000
Query 8: 0.500
Query 9: 1.750
Query 10: 5.625
```

Machine (2)

```
Query 1: 0.000
Query 2: 1.000
Query 3: 0.000
Query 4: 0.000
Query 5: 0.625
Query 6: 0.125
Query 7: 0.625
Query 8: 0.125
Query 9: 0.375
Query 10: 1.875
```

Machine (3)

```
Query 1: 1.000
Query 2: 3.333
Query 3: 1.000
Query 4: 0.000
Query 5: 3.000
Query 6: 1.667
Query 7: 4.333
Query 8: 2.000
Query 9: 3.333
Query 10: 8.000
```

Modularity and Encapsulation

The only thing we changed in terms of modularity and encapsulation concerning guide 2, was when returning strings. For example, in the model of commits, when returning the string of the function getCommitsCommit_at, it was returning the string saved in memory. To prevent changing the saved data by mistake, instead of returning the pointer to the string in memory, is returned one pointer to a copy of that string.

Observation: It was also created the header file of the file Tree.c, used in the resolution of the guide one.

Performance before

→ 5,4 Gb of RAM in 21 seconds (09-01-2022)
It was the junction of the functions of guides one and two.

Performance after optimization

→ 1,1 Gb of RAM in 20 seconds

Firstly, we began removing the unnecessary data. In guide 2, the program saved in memory one struct with all the information of the line read. However, solving the queries only required half of the information in memory. So, in each struct (user, repositories and commits), it was evaluated what was the information needed to solve the queries and removed the fields of the structs that weren’t used.

It was created intermediate files of queries 1,2,3,4 and 10. Each of that queries is executed at the beginning of the program, the result going to the intermediate files. Every time the queries are asked to execute, it is copied the result in the intermediate file to the file output in question. Note that if the query in processing has a restricted number of lines to the output, it is only copied that number. Only in these queries is used this method because the input in the specific query is always the same, only the number of lines differ. In the others queries, the result file is different according to the input, for example, a date range.

Another optimization made was creating the output of the query4 at the same time the catQueries were created, once it only needs the number of users and commits.

One of the biggest problems before was the lost memory because in the previous guides it wasn’t done free of a lot of data. So, using Valgrind as an auxiliary tool, we found all the memory that was lost and create a lot of functions to free them, for example, to free the catQueries, queries or pointers to strings.

The catUser was composted by one hash table that had as keys the id of the user and as values the struct user, one array that in each position had one struct user and the length of catUser.
The program only used the array of the catUser one time, to execute the query1, so it was replaced by the hash that was already part of the catUsers. The difference is that the time to pass through all positions of an array is significantly lesser than the time to pass through a hash table.

The same method was used in the array of the catRepos, however, we concluded that the reduction of the memory (0,1Gb of RAM) wasn't beneficial concerning the increase of time (3 seconds). It was chosen to maintain the array of the catRepos.

After doing these alterations, the program was running in 20 seconds and using 1,2Gb of RAM.

One solution to reduce the memory occupied by the program suggested by the professors was to put the catQueries in files. To do that, it was created a lot of functions in the models of the catQueries as saveCatCommitsInFile and respective auxiliary functions.

Firstly, it was saved in a file the catCommits, as the functions only needed to pass through all commits, not mattering what commit was or the order in which the commit was accessed. The program became four seconds slower, only reducing 0,1Gb of RAM.

The difference wasn't satisfactory, however, we continued and saved the catRepos in a file. The problem was that the queries sometimes required to go through the array of the catRepos, other times, it required to search one specific repository. So some queries needed to know the position of the specific repository in the file. To know that, firstly, it was sorted the array of catRepos by the id of the repository and then, each repository was saved in the intermediate file. Beyond this file, it was created, at the same time, one file that would be used as an index. In the index file were the specific id and the position of the repository in the intermediate file. To search the repository in the index file was used binary search. However, after running the program, it became apparent that this wasn’t one good solution because the time of execution had a huge increase. So, the program was modified to instead of having one index file, have a hash table with ids as keys and the position in the intermediate file as values. Another change was in the function saveCatReposInFile. Instead of creating the catRepos and only then creating the intermediate file, the intermediate file was created in the beginning, without the catRepos. The same was done to the function that saved catCommits in a file. The time reduced, yet it was still high. The program was running in 30 seconds, using 0,9 Gb of RAM. As the catUser was used in more queries than the catRepos, and it would need another hash table with the ids of the users, it was estimated that the program would only reduce another one point two or one point three Gb of RAM, however, it would become slower to at least forty seconds.

Finally, applying parallelism also was considered, yet after analyzing the results of the time each query took to complete, we came to the conclusion that it was not necessary.

Conclusion

In conclusion, the increase of time wasn't compensated by the decrease of the memory, so we preferred the version in which the program was running in 20 seconds, using 1,2 Gb of RAM. That was the chosen version because we consider the program should be the fastest running the queries, mainly if the program would be executed with the interaction between user and program. Besides that, most computers can run the program even if they have little RAM.