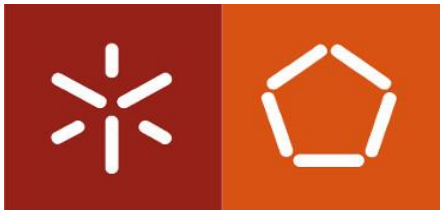
Relatório Guião 2

Grupo 37 | Laboratórios de Informática III

Hugo dos Santos Martins
a95125

João Bernardo Teixeira Escudeiro
a96075

José Pedro Batista Fonte
a91775



Departamento de Informática
Licenciatura em Engenharia Informática
2º ano | 1º Semestre
Laboratórios de Informática III

ÍNDICE

1. Introdução

O presente relatório visa descrever e explicar em detalhe o trabalho desenvolvido pelo grupo 37 no Guião 2 âmbito da cadeira de Laboratórios de Informática III, lecionada no curso de Licenciatura em Engenharia Informática na Universidade do Minho, no 1º semestre do ano letivo 2021/2022.

O guião 2 apresenta-se como a continuação do guião 1, já entregue pelo grupo. O objetivo do guião 1 é ler três ficheiros, nomeadamente os ficheiros user.csv, repos.csv e commits.csv e filtrar os dados corretos de acordo com os parâmetros apresentados. Como output o guião-1 deve apresentar a informação correta nos ficheiros users-ok.csv, repos-ok.csv e commits-ok.csv.

O objetivo do guião 2 é utilizar o output do guião 1, nomeadamente os ficheiros users-ok.csv, repos-ok.csv, commits-ok.csv e executar várias *queries* sobre os seus dados. A arquitetura da aplicação está ilustrada na figura 1.1.

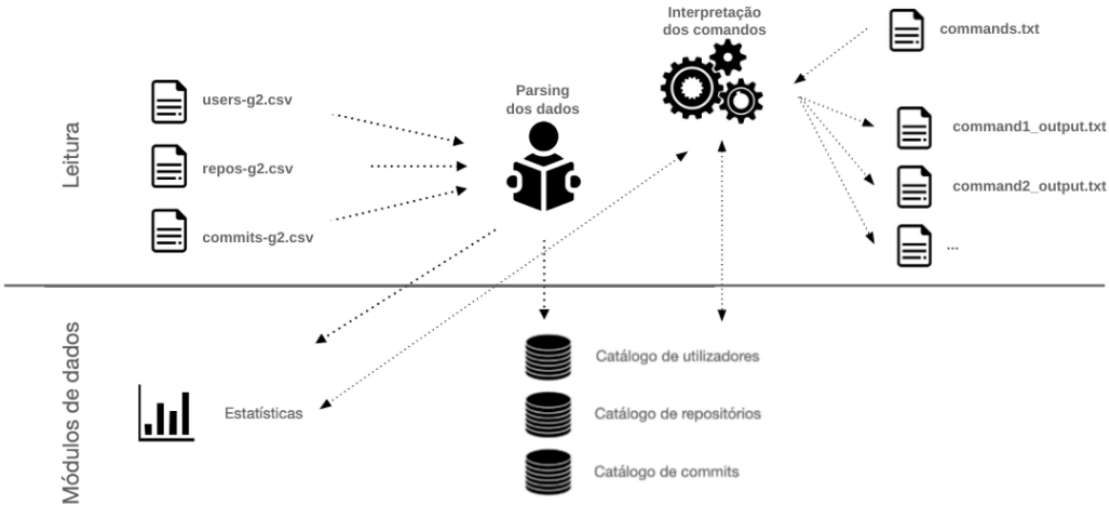


Figura 1. Arquitetura de referência para a aplicação a desenvolver

Resumidamente o guião 2 funciona do seguinte modo, as *queries* a ser executadas são escritas no ficheiro commands.txt, o ficheiro é lido pelo interpretador que depois executa-as sobre o conjunto de dados guardados pelo grupo. Como output, o guião 2 deve produzir um ficheiro .txt com o output de cada *querie*.

1. Organização dos Ficheiros

A pasta ***guiao-2/*** submetida no Github apresenta a seguinte estrutura:

- guiao-2/
  - docs/
  - entrada/
    - users-g2.csv
    - repos-g2.csv
    - commits-g2.csv
  - libs/
  - obj/
  - src/
    - guiao2.c
    - struct.c
    - struct.h
  - Makefile

Na pasta ***entrada/*** encontram-se os ficheiros .csv que contém todos os dados sobre os quais as *queries* vão ser executadas.

Na pasta ***src/*** encontram-se os ficheiros ***guiao2.c***, ***struct.c*** e ***struct.h***.

- **guiao2.c** – contém o código principal constituído pela função main(), *readline*() e a de todas as *queries*, *query*_().
- **struct.c** – contém o código auxiliar com todas as funções auxiliares das funções presentes em guiao2.c.
- **struct.h** – contém as estruturas de dados e a declaração das funções struct.c. Funciona também como *header*.

2. Explicação do Código

3.1 Estruturas de Dados

O grupo decidiu criar as seguintes *structs* com o objetivo de responder a cada uma das *queries* da forma mais eficiente possível.

Struct STATS

Para as *queries* estatísticas, o grupo decidiu criar uma estrutura que contemplasse todos os campos que as estas necessitavam para a sua execução. Estes campos são contabilizados à medida que o *parsing* dos dados é efetuado. Todos os campos são do tipo *int*.

```
typedef struct stats
{
    int user;
    int bot;
    int organization;
    int repos ;
    int colaboradores ;
    int bots_colab;
    int utilizadores;
    int commits;
}*  STATS;
```

***Figura 2**-Struct STATS*

Struct DATA

Esta *struct* serve de auxílio à realização das *queries* que tratam datas. Possui os campos ano, mês e dia, que são todos do tipo inteiro.

```
typedef struct data
{
    int ano;
    int mes;
    int dia;

}* DATA;
```

***Figura 3**- Struct DATA*

Struct AUX10

Esta *struct* serve de auxílio à realização das *queries* que tratam datas. Possui os campos ano, mês e dia, que são todos do tipo inteiro.

```
typedef struct aux10
{
    KEY author_id;
    int size;

} * AUX10;
```

***Figura 4**- Struct AUX10*

Struct LANG_ARR

Esta *struct* que guarda a *language*, o *number*, que é o número de ocorrências da linguagem, a *pos* é a posição no array, o *size* é o tamanho do array ocupado.

```
typedef struct lang_arr
{
    char* language;
    int number;
    int pos;
    int size;

} * LANG_ARR;
```

***Figura 5** - Struct LANG_ARR*

Struct IDS_ARR

Esta *struct* guarda o id de um utilizador, o *number* que é o número de commits daquele ID, a *pos* é a posição no array, o *size* é o tamanho do array ocupado.

```
typedef struct aux10
{
    KEY author_id;
    int size;

} * AUX10;
```

***Figura 6**- Struct IDS_ARR*

KEY: Todas as principais estruturas possuem uma *key* que diz respeito ao id do respetivo parâmetro (ou id do user ou do repositório).

Struct USERS

Para os users, separamos os seguintes campos, que nos são uteis à realização de cada uma das *queries*. A *Key* da tabela de *hash* que é uma string contempla o id de cada user. Os campos: login, *type*, *following_list*, *repos* são tratados como uma string. Os outros parâmetros *followers*, *following*, são inteiros enquanto campo *state* é um *char* e indica se a posição na tabela de *hash* está ou não ocupada.

```
typedef struct users
{
    KEY id;
    char state;
    int followers;
    int following;
    char* login;
    char* type;
    char* follower_list;
    char* following_list;
    char* repos;
} * USERS;
```

Figura 7 *- Struct USERS*

Struct REPOS

Para os repos, separou-se os seguintes campos, que são uteis à realização de cada uma das *queries*. Assim sendo, também temos a KEY que é o *id* de repositório, um *state*, que indica se a posição está ou não ocupada. O *owner_id* do repositório, a linguagem do mesmo que é uma string, um DATA *updated_at* que indica a data em que o repositório foi atualizado, e a *description* que também é uma string.

```
typedef struct repos
{
    KEY id;
    char state;

    int owner_id;
    char* language;
    DATA updated_at;
    char* description;

} * REPOS;
```

Figura 8 *- Struct REPOS*

Struct COMMITS

Para os commits, separou-se os seguintes campos, que são úteis à realização de cada uma das *queries*. Assim sendo, também temos a KEY que é o *id* de repositório, um *state*, que indica se a posição está ou não ocupada. Os três inteiros (*size*, *p*, *bot*), indicam, respetivamente, o tamanho alocado do array de *strings* (commits), tamanho utilizado desse array e o campo *bot* que nos diz se o repositório tem ou não *bots* como colaboradores.

```
typedef struct commits
{
    KEY id_repo;
    char state;
    int size;
    int p;
    int bot;
    char ** commit_line;


} * COMMITS;
```

Figura 9 *- Struct COMMITS*

Struct COMMITAUX

Esta *struct* apenas serve no auxílio da query2, para verificar quantos colaboradores possuímos

```
typedef struct commits_aux
{
    KEY author_id;
    char state;

} * COMMITS_AUX;
```

Figura 10 *- Struct COMMITS_AUX*

3.2 Hash Tables

Motivo para o seu uso

Na análise do guião o grupo entendeu que teria de guardar todos os dados dos ficheiros .csv em memória. Estruturas de dados como listas ou listas ligadas, em que a procura é linear, causariam imenso problemas no tempo de execução. Por isso a utilização de *hash tables* é um passo natural na organização dos dados em memória, visto que, tendo em conta a quantidade de linhas que cada ficheiro pode atingir, a utilização de *hash tables* reduz consideravelmente o tempo de execução das *queries*.

Construção das Hash Tables

Procedimentos no catálogo de USERS

A tabela de *hash* que possui os utilizadores, não é nada mais do que um array em que cada posição é uma *struct* do tipo USERS. A sua posição na tabela é dada pela função de *hash*, que para um user id (*key*) retorna uma posição no array que deve ser ocupada por aquele user. Assim quando se pretende procurar por algum user, a função de *hash* retorna de imediato qual a possível posição que o user se encontra. Caso a função dê valores iguais para dois users diferentes, tem que se verificar que o campo *state* está ou não ocupado. Caso já esteja, um destes tem de prosseguir para a posição seguinte mais próxima que não esteja ocupada. Para os utilizadores são guardados os campos que estão contemplados na *struct* USERS. É feita também uma contagem para o número de *Bots*, *Organization* e *Users* que existam, bem como o total de utilizadores, campos estes que dizem respeito à *struct* STATS.

Procedimentos no catálogo de REPOS

O catálogo dos repos, o método é semelhante ao catálogo dos USERS. A única diferença é que no array utiliza-se *structs* REPOS e a *key* é o id do repositório.

É também adicionado ao campo *repos* do user cujo id coincida com o id do dono do repositório. Por fim é feita uma contagem do número de repositórios adicionados, para adicionar ao campo SATS que contempla o número de repositórios.

Procedimentos no catálogo de COMMITS

Aqui os procedimentos tomados são ligeiramente diferentes. A *key* da tabela de *hash* é na mesma o id do repositório para o qual o commit é feito. Assim, e de forma natural, podem existir mais do que um commit para o mesmo repositório. Quando um commit está a ser adicionado à tabela, primeiramente é verificado se a posição dada pela tabela de *hash* está ocupada. Caso esteja e o ID seja o mesmo então adicionamos ao array, que possui as commits, mais uma linha. Caso ainda não esteja ocupada essa posição, então é criada e adicionado o commit ao array de commits. Assim temos uma espécie de relação entre repositórios e commits. À medida que se vão catalogando os commits é contabilizado o número total de commits bem como o número de commits que possuam um *bot* como colaborador, para os campos da STRUCT STATS. A figura 11 explica de forma visual a estrutura construída pelo grupo.

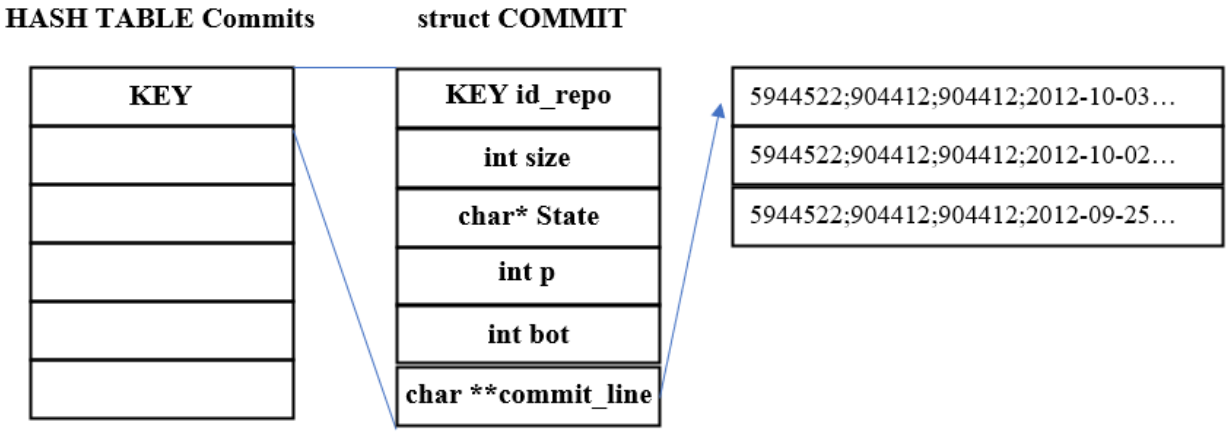


Figura 11 *- Esquematização da Hash Table de Commit*

3.3 Funções Principais

Função main()

A função main() é responsável por abrir os ficheiros presentes na pasta entrada/ e criar as Hash Tables dos dados dos users.csv, repos.csv e commits.csv seguindo os procedimentos indicados em cima. De seguida, invoca a função *readfile*() que vai ler as linhas do commands.txt e vai direcionar para a função das *queries* em questão.

Função readfile()

Para o leitor de linha o grupo usou a função *readfile*(), que recebe todos as *structs* inicializadas na main, bem como o *filepath* para o ficheiro de entrada. Assim e mediante o ficheiro de comandos, esta função separa cada uma das linhas mediante os "*tokens*" da instrução e direciona para cada uma das *queries* com os respetivos parâmetros que são necessários para a sua execução.

Query 1

Para a *query* 1 e face à estrutura STATS que já contempla tudo o que é necessário, nomeadamente o número de *Bots*, *Organizations* e User. Assim apenas é necessário retirar os valores dos respetivos domínios da *struct* STATS.

Query 2

Para a *query* 2, e de forma semelhante à *query* 1, a estrutura STATS já possui os campos colaboradores e repositórios. Assim, é apenas necessário fazer uma divisão entre estes campos e apresentar o resultado com duas casas decimais.

Query 3

Na *query* 3 o objetivo é retornar o número de repositórios contendo *bots* como colaboradores. Assim e já contendo na *struct* STATS esse valor, apenas temos de o colocar no ficheiro de output.

Query 4

Na *query* 4 o objetivo é retornar a média entre o número de commits e o número de utilizadores. Assim e já contendo na *struct* STATS esses valores, apenas temos de colocar o valor dessa divisão no ficheiro de output com duas casas decimais.

Query 5

Na *query* 5 o objetivo é, mediante os parâmetros, apresentar o Top N de utilizadores com mais commits num determinado intervalo de datas. A função cria uma estrutura de dados do tipo IDS_ARR para guarda os ids dos utilizadores e o nº total de commits.

Primeiramente, inicializa-se o IDS_ARR e verifica-se que a *data_inicio* é antes da *data_fim*. De seguida, o método adotado é percorrer todas as *commit_lines* de cada commit. Em cada *commit_line* verifica-se se a data está dentro do intervalo de tempo definido, caso não esteja passa à próxima *commit_line*. Caso esteja, verifica-se se o author_id desse commit está na lista IDS_ARR com a função *checkID*(), que retorna -1 se não tiver ou a sua posição no array se ele estiver. Na situação em que retorna -1 adiciona-o à lista, na situação de já pertencer vai aumentar o seu *number* em +1.

Assim que a lista está concluída, esta é organizada de forma decrescente e escreve no output.txt os primeiros N ids, login e a quantidade de commits.

Query 6

Na *query* 6 o objetivo é, mediante os parâmetros, apresentar o Top N de utilizadores presentes em commits de uma determinada linguagem. Há semelhança da *query* 6, a função cria um array para guardar os ids dos repositórios da linguagem de input, outro array do tipo IDS_ARR para guardar os ids dos users e nº de presenças e um *commitaux* do tipo COMMITS.

Primeiramente, cria-se uma lista de ids dos repositórios da linguagem input com a função *searchIDREPOSlang*(). De seguida inicializa-se o array *author_ids_array*, e percorre-se a lista array_id_repos. Com o *id_repo*, guarda-se no *commitaux struct* commits com todas as *commit_lines* correspondentes a este repositório. Em cada linha de commit, verifica-se se o author_id desse commit está na lista IDS_ARR com a função *checkID*(), que retorna -1 se não tiver ou a sua posição no array se ele estiver. Na situação em que retorna -1 adiciona-o à lista, na situação de já pertencer vai aumentar o seu *number* em +1.

Assim que a lista está concluída, esta é organizada de forma decrescente e escreve no output.txt os primeiros N ids, login e a quantidade de commits.

Query 7

Na *query* 7 o objetivo é, mediante um input (data), devolver todos os repositórios sem commits a partir de essa data. O método que o grupo optou foi para cada repositório guardar numa *struct* DATA a data do commit mais recente. Depois de já ter percorrido todas as datas de commit de um repositório, é feita a comparação com a data de input e se a data guardada na *struct* for antes da data do input, então adiciona o id do repositório e a respetiva descrição ao ficheiro output.txt.

Query 8

Na *query* 8 o objetivo é, mediante os parâmetros, apresentar Top N de linguagens presentes com mais frequência em repositórios a partir de uma determinada data. Como estruturas auxiliares, cria-se um array do tipo LANG_ARR que guarda a linguagem e todas as suas ocorrências.

Primeiramente, inicializa-se o *lang_array* e de seguida percorre-se todos os COMMITS. Dentro do commit, procura qual a linguagem do repositório em questão, e com o *checkLANG* verifica se esta linguagem já pertence ou não à lista. Caso não pertença adiciona a linguagem à lista e com a função *ncommitvalid* verifica quantos commits são validos de acordo com data de input, de seguida adiciona esse número à linguagem no *lang_array*. Caso a linguagem já pertença, faz os dois últimos passos descritos. Assim que a lista está concluída, esta é organizada de forma decrescente e escreve no output.txt as primeiras N linguagens.

Query 9

Na *query* 9 o objetivo é, mediante um input (N) devolver os N utilizadores com mais commits em repositórios cujo *owner* é um amigo seu. A estratégia utilizada foi criar um array do tipo AUX10 que em cada posição tivesse o nº de commits feitos pelo utilizador para repositórios de amigos e o id do user em questão. Depois de o array já contemplar todos os users é feita a ordenação do array (*MergeSort*) pelo parâmetro size, ou seja, ordenar o array pelo nº de commits. No fim apenas se imprime o número desejado de author_id mediante o parâmetro (N);

Query 10

Na *query* 10 o objetivo é, mediante um input (N), devolver as N maiores mensagens de cada repositório. A estratégia utilizada foi criar um array do tipo AUX10 que em cada posição tivesse o tamanho da mensagem e o id do user que fez commit daquela mensagem. Depois de o array já contemplar todas as mensagens de um repositório é feita a ordenação do array (utilizando a função

mergeSort) pelo parâmetro *size*, ou seja, ordenar o array pelo tamanho de mensagem. De seguida é feita a inversão do array para que fique ordenada de forma decrescente. No fim apenas se escreve no ficheiro output.txt o número desejado de author_id mediante o parâmetro (N).

4 Testes de desempenho

Para os testes de desempenho foi utilizado o computador portátil Acer Predator Triton 500, com as seguintes especificações:

- CPU: Intel i7 9750H @ 2.60 GHz/core (6 cores/12 Threads)
- GPU: NVIDIA GeForce RTX 2060
- RAM: 16 GB DDR4

	Teste1 (s)	**Teste2 (s)**	**Teste3 (s)**	**Teste4 (s)**	**Teste5 (s)**	**Média (s)**
Query 1	0.000072	0.000089	0.000072	0.000073	0.000070	0.0000752
Query 2	0.000112	0.000102	0.000083	0.000069	0.000070	0.0000872
Query 3	0.000096	0.000067	0.000072	0.000099	0.000070	0.0000808
Query 4	0.000141	0.000145	0.000187	0.000111	0.000252	0.0000167
Query 5	17.30235	17.37891	17.33788	17.01996	16.97594	17.203008
Query 6	2.827633	2.892126	2.900205	2.756457	2.847861	2.8448564
Query 7	0.359119	0.354847	0.377199	0.355236	0.371871	0.3636544
Query 8	0.443413	0.444781	0.410073	0.409091	0.414178	0.4243072
Query 9	1.233782	1.219822	1.236210	1.281097	1.293110	1.2528042
Query 10	0.700256	0.681528	0.681522	0.707390	0.693707	0.6928806

5 Conclusão

No geral o grupo autoavalia-se positivamente visto que demos por concluído este guião. Fazer *hash tables* foi uma das tarefas mais difíceis dada a inexistência de conhecimento prévio, no entanto a performance melhorou imenso. O grupo também teve dificuldade a lidar com a gestão de memória e o tipo de mensagens de erro em C. Visto que maior parte do tempo foi gasto a resolver *segmentation faults*.