RELATÓRIO

GUIÃO 2

Laboratórios de Informática III

2021/2022

Licenciatura em Engenharia Informática | 2ºAno

Diana Ribeiro Mateus | A95985

Pedro Marcelo Bogas Oliveira | A95076

Rodrigo José Teixeira Freitas | A96547

Índice

1. Introdução

No âmbito da Unidade Curricular de Laboratórios de Informática III, foi proposta a realização de um trabalho que inclui vários guiões, sendo que este relatório incide sobre o Guião 2.

O Guião 2 consiste na utilização da linguagem C para o tratamento de ficheiros, nomeadamente ficheiros do tipo CSV (*comma-separated values*), incluindo a geração de catálogos de dados incluídos nesses ficheiros, a leitura de ficheiros TXT (*text file*) com comandos a ler e executar, e ainda a geração de ficheiros *txt* finais com os resultados pretendidos nos comandos. A geração de catálogos de dados (utilizadores, repositórios e *commits*) inclui a leitura de ficheiros *csv* através de uma função que permite o *parsing* dos dados. Para além disso, a leitura de ficheiros *txt* permite ler e executar os comandos neles presentes. Finalmente, ocorre a geração de ficheiros finais do tipo *txt*, cada um correspondente a um comando com o output pretendido da respetiva *query* e obtido a partir dos catálogos produzidos inicialmente.

Foi necessário recorrer a princípios mais avançados de programação, como o uso de estruturas de dados eficientes para armazenar e consultar grandes quantidades de informação, garantindo sempre o encapsulamento dos dados e a estruturação do projeto em módulos.

Os objetivos principais deste trabalho passam pela utilização da linguagem C e de todas as suas funcionalidades, com destaque na modularidade e encapsulamento, estruturas dinâmicas de dados e medição de desempenho, para além da consolidação de todos os conhecimentos essenciais ao desenvolvimento de um projeto, nomeadamente a compilação, linkagem, depuração de erros e gestão dos respetivos repositórios.

2. Estrutura do Projeto

O desenvolvimento do guião 2 passou pela implementação de uma arquitetura de aplicação baseada em quatro passos importantes, nomeadamente, o *parsing* dos dados, a construção de catálogos, a interpretação de comandos e a execução de *queries*.

2.1. *Parsing* dos Dados

O *parsing* dos dados baseia-se no desenvolvimento de uma função ou parte do código no qual é realizada a leitura e tratamento dos dados dos ficheiros. Nesta fase do projeto, foram utilizadas funções que introduziam os elementos de cada linha dos ficheiros num nodo de uma lista ligada, permitindo assim o armazenamento dos dados numa estrutura dinâmica e com memória por si só, também, dinâmica. A implementação de listas ligadas pareceu uma boa opção de entre as estruturas de dados conhecidas nesta fase, considerando que posteriormente seria necessário percorrer todos os dados presentes nos ficheiros. Contudo, numa fase mais avançada do projeto a nossa abordagem foi alterada, tendo em conta as dificuldades encontradas.

2.2. Construção de Catálogos

Nesta fase, a construção de catálogos, que advém essencialmente do *parsing* dos dados, é realizada. Os dados encontram-se em listas ligadas, de relativamente fácil acesso.

2.3. Interpretação dos Comandos

A interpretação dos comandos é feita com recurso a uma função que lê linha a linha o ficheiro dos comandos e executa a *query* indicada, através de uma função com um *switch* que encaminha a linha e o ID da *query*, bem como outros argumentos necessários para a respetiva função de execução da *query*.

2.4. Execução das *Queries*

O módulo das *queries* é o responsável pelo fluxo destas, ou seja, trata de receber o input do utilizador, chamar a função correspondente da *query*, fornecendo-lhe este mesmo input, e enviar o output para os ficheiros correspondentes, de forma a ser apresentado ao utilizador.

2.4.1. *Query* 1

A *query* 1 implica a descrição da quantidade de *bots*, organizações e utilizadores, no catálogo dos utilizadores. Esta *query* apenas necessita do seu ID como input. A lista ligada relativa ao catálogo dos utilizadores é percorrida, fazendo a contagem destes três tipos e retornando, no final, os valores obtidos.

2.4.2. *Query* 2

A *query* 2 solicita o número médio de colaboradores por repositório. O input necessário é apenas o ID da *query*. Numa fase inicial é percorrido o catálogo dos repositórios, de forma a ser contado o número de repositórios existentes. De seguida, é contado o número de colaboradores, isto é, o número de utilizadores que são *author_id* ou *commiter_id*, campos existentes no catálogo dos *commits*. No final, retorna o valor resultante da divisão do número de colaboradores pelo número de repositórios.

2.4.3. *Query* 3

A *query* 3 tem como input o seu próprio ID e pretende que o output seja a quantidade de repositórios com colaboradores listados com o tipo *bot*. O catálogo dos utilizadores é percorrido, obtendo-se os *ID's* dos utilizadores com o tipo *bot*, guardando estes numa estrutura de dados auxiliar. Posteriormente, percorre-se o catálogo dos *commits* para a obtenção dos *ID's* dos colaboradores. Nesta fase, o ID do colaborador é procurado na estrutura de dados que guarda os utilizadores *bots*. Se for encontrado, o ID do respetivo repositório é gravado numa outra estrutura de dados auxiliar. No final, são contados o número de nodos da estrutura auxiliar com os *ID's* dos repositórios, sendo esse o valor retornado pela *query*.

2.4.4. *Query* 4

A *query* 4 pretende a obtenção da quantidade média de *commits* por utilizador, ou seja, o quociente entre o número total de *commits* e o número total de utilizadores. Foi percorrido o catálogo dos utilizadores, de forma a saber quantos existem e posteriormente o catálogo dos *commits* para saber qual o número total de *commits*. O input desta *query* é o seu ID e o output é o valor obtido pelo quociente mencionado anteriormente.

2.4.5. *Query* 5

A *query* 5 permite obter o top N utilizadores mais ativos num determinado intervalo de datas. O input pretendido é o número desejado de utilizadores, a data de início e a data de fim. É percorrido o catálogo de *commits* e inserido numa estrutura de dados auxiliar os *ID's* dos utilizadores que se inserem no critério estabelecido, bem como as vezes em que cada um desses utilizadores realiza um *commit*. Posteriormente, é identificado o utilizador com mais *commits* através do valor associado na estrutura auxiliar, sendo imediatamente impresso no ficheiro o ID e o *login*, obtidos de um catálogo específico dos utilizadores, e número de *commits* deste mesmo utilizador. Este processo é repetido para os N utilizadores mais ativos.

2.4.6. *Query* 6

A *query* 6 permite obter o top N utilizadores com mais *commits* em repositórios de uma determinada linguagem. O input pretendido é um número desejado de utilizadores e as respetivas linguagens. Numa primeira instância, são filtrados os repositórios que apresentam a linguagem pretendida. O ID destes repositórios é guardado numa estrutura de dados auxiliar. Posteriormente, é percorrido o catálogo dos *commits* e procurado na estrutura auxiliar gerada anteriormente o ID do repositório que se encontra neste mesmo catálogo. Se o ID for encontrado, guarda numa outra estrutura de dados auxiliar o *author_id* do *commit* e as vezes em que este realiza um *commit*. No final, é identificado o utilizador com mais *commits* através do valor associado na segunda estrutura auxiliar, sendo imediatamente impresso o ID e o *login*, obtidos de um catálogo específico dos utilizadores, e número de *commits* do utilizador. Este processo é repetido para os N utilizadores mais ativos.

2.4.7. *Query* 7

A *query* 7 permite obter a lista dos repositórios inativos a partir de uma determinada data. O input é uma data de início e o output deve incluir o ID do repositório e a descrição do mesmo. O catálogo dos *commits* é percorrido com o intuito de se obter os *ID's* dos repositórios que se encontram inativos a partir da data facultada. Para isso, foi utilizada uma função auxiliar que compara os valores das datas presentes no campo *commit_at* e retorna 1 caso, a data presente no catálogo seja menor do que a data do input. Estas informações são guardadas numa estrutura de dados auxiliar. Posteriormente, é percorrida esta estrutura auxiliar, de forma a ser possível imprimir o ID do repositório, bem como a descrição associada, com recurso a um catálogo específico dos repositórios.

2.4.8. *Query* 8

A *query* 8 permite obter a lista de N linguagens mais utilizadas a partir de uma determinada data. O input deve incluir o número desejado de linguagens e a data de início. Para obter o output desejado, foi percorrido o catálogo dos repositórios de forma a ser possível, com recurso a uma estrutura de dados auxiliar, listar todas as linguagens presentes (exceto quando o repositório apresenta no campo linguagem "*None*") e também quantas vezes cada uma aparece. No final, é identificada a linguagem mais utilizada, isto é, que contem o maior valor, associado ao número de vezes que a linguagem aparece, imprimindo, neste instante, o nome da linguagem. Este processo é repetido para as N linguagens mais utilizadas.

2.4.9. *Query* 9

A *query* 9 permite obter o top N de utilizadores com mais *commits* em repositórios cujo *owner* é um amigo, ou seja, cujo *owner* está contido na sua lista de *followers* e *following*. O input é o número desejado de utilizadores. Primeiramente, é percorrido o catálogo dos *commits* de forma a se obter o *repo_id* e o *commiter_id*. Posteriormente, é procurado numa estrutura de dados auxiliar (que contém o ID do repositório e o ID do *owner* deste), o ID do *owner* relativo ao repositório em que se encontra, no catálogo dos *commits*. Numa fase seguinte, pesquisa-se numa outra estrutura auxiliar (que contém o ID do utilizador e um apontador para o nodo da lista ligada, onde se encontra toda a informação sobre este, relativa ao catálogo dos utilizadores), o ID do *commiter*. De seguida, é criada uma outra estrutura de dados auxiliar com os amigos desse mesmo utilizador. Caso sejam encontrados amigos, o ID do utilizador é guardado e ao mesmo tempo é contado o número de vezes em que este faz *commit*. No final, é identificado o utilizador com mais *commits*, imprimindo neste instante, o seu ID e o *login*. Este processo é repetido para os N utilizadores com mais *commits*.

2.4.10. *Query* 10

A *query* 10 permite obter o top N de utilizadores com as maiores mensagens de *commit* por repositório. O input é o número desejado de utilizadores. Inicialmente, é percorrido o catálogo dos *commits* e verificado o tamanho da mensagem de cada um. Posteriormente, e com recurso a uma estrutura auxiliar, são guardados os *ID's* dos utilizadores que realizaram o *commit*, bem como o tamanho da mensagem associada ao seu *commit*. Por último, são identificados os N utilizadores com as maiores mensagens e é impresso o seu ID, *login* e o tamanho da mensagem associada.

3. Módulos e Estruturas de Dados

3.1. Main

O módulo principal permite a abertura e o fecho dos ficheiros, bem como a chamada das principais funções associadas à criação dos catálogos e gestão das *queries*.

3.2. Utilizadores

A estrutura de dados utilizada para o armazenamento do catálogo dos utilizadores foi uma lista ligada, em que cada nodo apresenta toda a informação de uma linha do ficheiro. O módulo associado aos utilizadores (*data_users*) permite realizar operações de inserção, pesquisa e obtenção dos dados associados aos utilizadores, bem como a realização de operações de *free* de estruturas deste tipo.

3.3. Commits

A estrutura de dados utilizada para o armazenamento do catálogo dos *commits* foi uma lista ligada, em que cada nodo apresenta toda a informação de uma linha do ficheiro. O módulo associado aos *commits* (*data_commits*) permite realizar operações de inserção, pesquisa e obtenção dos dados associados aos utilizadores, bem como a realização de operações de *free* de estruturas deste tipo.

3.4. Repositórios

A estrutura de dados utilizada para o armazenamento do catálogo dos repositórios foi uma lista ligada, em que cada nodo apresenta toda a informação de uma linha do ficheiro. O módulo associado aos repositórios (*data_repos*) permite realizar operações de inserção, pesquisa e obtenção dos dados associados aos utilizadores, bem como a realização de operações de *free* de estruturas deste tipo.

3.5. Estruturas Auxiliares

Existem dois módulos associados às estruturas auxiliares, *data_tpll* e *data_tree*.

O primeiro módulo trata estruturas auxiliares do tipo lista ligada com tuplos.

```
struct top
{
    int info;
    int counter;
    struct top * next;
};
```

Figura 1 - Estrutura de Dados Auxiliar "TOP"

```
struct string
{
    char *info;
    int counter;
    struct string * next;
};
```

Figura 2 - Estrutura de Dados Auxiliar "STRING"

Este módulo permite a realização de operações de inserção, pesquisa e obtenção de dados nela presentes. Para além disso, apresenta algumas funções auxiliares utilizadas em várias *queries*, tal como *mostFreq*, que permite verificar o valor com o *counter* maior, bem como a realização de operações de libertação de memória destas estruturas auxiliares.

O segundo módulo trata de árvores binárias e todas as funções associadas a este tipo de estrutura. Além das estruturas auxiliares das *queries*, apresenta ainda várias formas diferentes de catálogo (tal como a *treeUsers* e a *treeUsers9* na imagem), mais fáceis de

aceder e introduzidas para colmatar o longo tempo que determinadas *queries* demoravam a executar.

```
struct tree
{
    int value;
    struct tree *l, *r;
};
```

Figura 3 - Estrutura de Dados Auxiliar "Árvore Binária"

```
struct treeUsers
{
    char * id;
    char * login;
    struct treeUsers *l, *r;
};
```

Figura 4 - Estrutura de Dados Catálogo "*treeUsers*"

```
struct treeUsers9
{
    char * id;
    struct users *node;
    struct treeUsers9 *l, *r;
};
```

Figura 5 - Estrutura de Dados Catálogo "*treeUsers9*"

Este módulo permite a realização de operações de inserção, pesquisa, contagem de nodos e libertação de memória. Bem como, funções de filtragem de informações pedidas nas *queries*, bem como a geração de estruturas específicas para estas, tal como a *friendsTreeF* (para a *query* 9).

3.6. Queries

O módulo das *queries* é o responsável pelo fluxo destas, tratando de receber o input do utilizador, chamar a função da *query* correspondente, fornecendo-lhe o input necessário e enviando o output para os ficheiros correspondentes, para poder ser apresentado ao utilizador.

4. Complexidade das Estruturas

Para a criação dos catálogos dos utilizadores, *commits* e repositórios utilizou-se listas ligadas e árvores binárias. A primeira estrutura foi utilizada inicialmente, contudo, em algumas *queries* o tempo de execução era demasiado lento, daí a utilização de árvores binárias para permitir o aceleramento de algumas destas *queries*, considerando que o tempo de acesso das árvores binárias é menor do que o tempo de acesso a listas ligadas. As listas ligadas foram inicialmente usadas considerando que não seria necessário aceder de forma ordenada aos dados nelas presentes, que teriam de ser acedidos todos os dados e considerando que a complexidade de tempo de inserção destas estruturas é pouco custosa (constante). Contudo, posteriormente, aquando da execução das *queries*, principalmente, as que requeriam a interseção de dados entre os vários catálogos, foi necessário a introdução de uma estrutura que permitisse uma complexidade do tempo de acesso menor, daí a necessidade de recorrer a árvores binárias.

Além disso, foram utilizados três tipos de estruturas auxiliares (*top*, *string* e *tree*). Estas duas primeiras permitiam essencialmente a contagem de dados e a segunda foi utilizada principalmente para o armazenamento temporário de dados únicos que serviria como uma lista de valores a utilizar posteriormente, principalmente em funções de procura.

A estrutura *top* contém um inteiro com o valor a guardar e outro inteiro, *counter*, com o propósito de contar quantas vezes aparece o primeiro valor. Foi utilizada, por exemplo, na *query* 5, guardando o ID do utilizador e o número de vezes em que este realiza um *commit*. A estrutura *string* é essencialmente igual à *top*, contudo apresenta uma *string* como variável a guardar e um inteiro, *counter*, que conta o número de vezes em que esta aparece. Foi utilizada, por exemplo, na *query* 8, de forma a guardar as linguagens mais utilizadas.

A estrutura *tree* contém apenas um inteiro com o valor a guardar. Esta estrutura foi utilizada, por exemplo, na *query* 2, com o intuito de guardar o ID dos utilizadores que seriam colaboradores.

As estruturas do tipo *treeUsers* e *treeRepos* apresentam duas *strings* que permitem guardar o valor do ID e do login. Foram criadas para permitir um acesso mais rápido desta informação específica, na execução de algumas *queries*. Para além destas, foi criada a estruturas treeUsers9, especificamente para a *query* 9, visto que esta, sem a utilização destas estruturas, não apresentava uma execução em tempo útil.

5. Testes de Desempenho

Para medir o tempo de execução do nosso programa foi utilizado o comando *time* que associado à compilação/execução permitia a obtenção dos valores de execução. Os resultados obtidos estão apresentados na figura 5.

Queries	Ficheiro 55MB	Ficheiro 132MB
catalogos	0.962 s	1.969 s
1	0.01 s	0.02 s
2	0.07 s	0.19 s
3	0.1 s	0.18 s
4	0.02 s	0.05 s
5	3.3 s	29.70 s
6	0.39 s	3.22 s
7	1.82 s	65.67 s
8	0.04 s	0.03 s
9	0.14 s	1.60 s
10	17.31 s	129.15 s

Figura 6 - Resultados dos testes de tempo de execução

Os resultados obtidos são a média de três medições realizadas às diferentes *queries* e criação de catálogos. De uma forma geral:

- As *queries* 1, 4 e 8 têm um tempo de execução consideravelmente baixo comparando com as restantes. A *query* mais lenta é a 10, visto que tem de percorrer o catálogo dos *commits* e determinar o tamanho de todas as mensagens presentes neste ficheiro, através da utilização de algumas estruturas auxiliares no percurso. Estes resultados são, de forma geral, consistentes relativamente ao tamanho dos ficheiros.
- Um aumento do tamanho dos ficheiros aumenta consideravelmente o tempo de execução de algumas *queries*, mais especificamente nas *queries* 2, 3, 5, 6, 7 e 10. Este aumento pode ter sido causado pelo facto de que, na generalidade, têm de ser percorridos vários catálogos e ser feitas várias operações auxiliares em cada iteração, sendo que como o seu tamanho aumenta, é natural que também aumente significativamente o tempo de execução.
- Em *queries* que apenas percorrem os catálogos (*query* 1 e 4), bem como as *queries* 3 e 8, o tamanho dos ficheiros não parece afetar o tempo de execução.

É necessário atribuir uma pequena variação dos resultados a erros estatísticos.

Para além dos testes de tempo de execução, foram também realizados testes de memória, usando a ferramenta *Valgrind*.

```
LEAK SUMMARY:
==159==    definitely lost: 47,183,908 bytes in 3,183,789 blocks
==159==    indirectly lost: 2,555,745 bytes in 145,864 blocks
==159==      possibly lost: 0 bytes in 0 blocks
==159==    still reachable: 309,621,493 bytes in 6,762,739 blocks
==159==         suppressed: 0 bytes in 0 blocks

P SUMMARY:
==159==     in use at exit: 359,361,146 bytes in 10,092,392 blocks
==159==   total heap usage: 15,362,702 allocs, 5,270,310 frees, 427,134,275 bytes allocated
```

Figura 7 - Resultados da Gestão de Memória com recurso ao *Valgrind*

No que concerne às *memory leaks*, podemos verificar que existe uma má gestão da alocação da memória no programa, considerando os altos valores dos campos *definitely lost* e *indirectly lost*. Estes são relativos às situações em que a memória é alocada, mas não pode ser libertada pois o programa já não apresenta *pointers* para o bloco de memória. Uma *memory leak* do tipo *still reachable* é quando a memória foi alocada, mas não libertada antes do final do programa, sendo este também um dos aspetos a melhorar.

De uma forma geral, a gestão da memória ficou aquém do desejado, considerado que o *heap usage* apresenta 15,362,702 *allocs* e apenas 5,270,310 *frees*.

6. Conclusão

A realização deste trabalho foi importante na consolidação dos conhecimentos da linguagem C, através da aplicação dos conceitos de memória dinâmica, modelamento e encapsulamento, bem como interpretação de comandos e extração de dados. Sendo assim, podemos concluir que este objetivo foi cumprido.

Para além disso, a utilização do *Makefile* possibilitou a utilização de ferramentas de compilação, nomeadamente a utilização de comandos de otimizações, bem como linkagem e a criação de ficheiros que permitiam a depuração de erros. A gestão dos repositórios ficou um pouco aquém do esperado. Na generalidade, estes objetivos foram cumpridos.

Sentimos que poderíamos ter ido mais longe no que toca ao tipo de estruturas utilizadas. Poderiam ter sido utilizadas estruturas de dados mais eficientes, considerando o tempo de execução das *queries* serem gravemente influenciadas pelo tipo de estruturas utilizadas (listas ligadas). Uma alternativa seria a utilização de *HashTables*, sendo que um dos impedimentos relativamente à utilização destas estruturas foi o tempo para a realização do projeto.

No que concerne aos testes de tempo e memória, concluímos que estes ficaram muito aquém do esperado, principalmente devido ao que foi mencionado anteriormente relativamente às estruturas de dados utilizadas e à pobre gestão de memória.

Desta forma, as principais dificuldades que sentimos foram, então, a gestão do tipo de estruturas a utilizar e a gestão de memória, sendo dois aspetos a melhorar na próxima fase do projeto.

Em conclusão, este trabalho foi realizado cumprindo maioritariamente os objetivos, advindo um processo intenso de aprendizagem relativamente à gestão de memória e tempo de execução e ao desenvolvimento de *Software* em equipa.