Universidade do Minho
Escola de Engenharia

Universidade do Minho
Departamento de Informática
Laboratórios de Informática III

João Miguel da Silva Pinto Ferreira (a89497)
Luís Alberto Barreiro Araújo (a96351)
Marisa Ferreira Soares (a92926)
14 de dezembro de 2021

Conteúdo

Introdução

Este relatório foi realizado no âmbito da segunda parte do projeto de grupo da unidade curricular de *Laboratórios de Informática III* e tem como principais objetivos:

- A leitura e cruzamento de dados entre ficheiros *.csv*.

- A implementação de métodos essenciais de C, tal como, modularidade, encapsulamento, estruturas dinâmicas de dados, e medição de desempenho.

- Utilização de ferramentas de desenvolvimento de projetos em grande escala em C.

Estrutura de Dados

2.1 Módulo de dados utilizados

2.1.1 User

```
struct user{
    int id;
    char* username;
    char* type;
    int friends[100000];
    int n_amigos;
    int n_commits;
};
```

Em primeiro lugar esta é a estrutura que armazena os dados necessários dos utilizadores lidos no ficheiro "users-g2.csv", composta por um *id, username, type*, que correspondem aos campos do ficheiro e em relação ao *array* de amigos e nº de amigos é deduzido pela lista de *followers* e *following* provenientes do ficheiro. Por fim, o *n_commits* servirá para guardar o número de commits de um utilizador que um conjunto de queries vai requerer, nomeadamente nas queries 5,6 e 9.

2.1.2 Commit

```
struct commit{
    int repo_id;
    int author_id;
    int commiter_id;
    char* date;
    char* message;
};
```

Em segundo lugar esta é a estrutura responsável por armazenar o tipo de dados commit, lidos no ficheiro "commits-g2.csv", composta pelos campos: *repo_id, author_id, committer_id, date* (que corresponde ao campo commited_at no ficheiro) e *message*.

2.1.3 Repos

```
struct repos{
    int id;
    int owner_id;
    char* language;
    char* data;
    char* description;
};
```

Por fim esta é a estrutura encarregue por armazenar o tipo de dados repos, lidos no ficheiro "repos-g2.csv", composto pelo: *id, owner_id, language, date* (corresponde ao created_at) e *description.*

2.2 Catálogos

```
typedef struct lUsers{
    GHashTable *user;
} *LISTAUSERS;

typedef struct lCommits{
    GHashTable *commit;
} *LISTACOMMITS;

typedef struct lRepos{
    GHashTable *repos;
}*LISTAREPOS;
```

Após a leitura dos dados iremos colocá-los todos num catálogo que é uma *GHashTable* da biblioteca do *Glib*, ou seja, todos os dados lidos nos ficheiros estão armazenados em hash tables e isto aplica-se a todos os ficheiros.
Adicionamos assim as respetivas structs dos diferentes dados às suas listas. Na Hash Table user a key associada a cada struct é o ID de utilizador. Na Hash Table repos a key associada é o ID do repositório. Em relação à estrutura de dados dos commits, uma vez acabado o trabalho, conseguimos perceber que Hash Tables poderá não ter sido a melhor opção, uma vez que nunca procuramos nada na Lista de Commits, mas sim só percorremos (levando assim sempre ao uso de GLists) e também porque não há uma key definida para os commits. Para este problema, a key utilizada para a Hash Table dos commits foi uma variável que estaria a ser incrementada sempre que fosse adicionado um novo commit.

Estratégia

3.1 Estratégias utilizadas

3.1.1 Queries estatísticas

Começamos por percorrer cada ficheiro e ler uma linha de cada vez, recorrendo à função strsep(). Uma vez processados todos os dados, construímos a respetiva struct gerada por estes, de cada linha, e adicionamos à respetiva lista (*Hash Table*), anteriormente já inicializadas. Por exemplo, a função *adicionarUsers()* trata a leitura do ficheiro *users-g2.csv*, linha a linha e a função *adicionarU()* corresponde ao tratamento de dados, inicialização de uma *struct* com os campos dessa linha e finalmente à inserção na *Hash Table* dos utilizadores. O procedimento é o mesmo para os repositórios e commits, chamando-se as funções, respetivamente, *adicionarRepos()* e *adicionarR()* e *adicionarCommits()* e *adicionarC()*;

Depois disto, procede-se, então, à execução das queries. Segue-se a lista destas e respetiva estratégia utilizada na implementação.

Query 1: Primeiramente criamos uma lista ligada do tipo *GList* através da Hash Table. De seguida percorremos a GList e usamos a função *strcmp()* juntamente com 3 contadores (bot, organization e user) para verificar o número de types: "Bot", "Organization"e "User"presentes na lista de users, incrementando assim cada contador respetivamente. Finalmente, escrevemos no ficheiro criado para esse fim, o valor final correspondente a cada contador anteriormente referido.

Query 2: Começamos por:

- Inicializar uma nova Hash Table de users ao qual chamamos "lista";
- Criar uma GList llc (lista ligada);
- Gerar 3 variáveis inteiras: "colaboradores", "repositorios"e"avg"

Para dar o respetivo valor as variáveis referidas, percorremos a lista de commits no caso dos colaboradores. Sempre que na Hash Table não fosse encontrado ou o author_id ou o commiter_id, estes seriam inseridos na nova Hash Table (chamada "lista") e a respetiva variável "colaboradores"seria incrementada, pois desta forma todos os users que não deram qualquer commit não serão inseridos e todos os commits repetidos também não serão inseridos. No caso da variável "repositorios"apenas, usamos uma função do Glib que dá o tamanho da lista de repositórios. Finalmente foi apenas fazer a divisão de ambas as variáveis e atribuir esse valor ao "avg"que será dado como output.

Query 3: Iniciamos esta query criando uma lista ligada ("GList* llc") e uma Hash Table vazia. De seguida fazemos um ciclo para percorrer a lista de commits, cujo objetivo é verificar se o numero de repositórios contem como colaboradores o type "Bot". Começamos o ciclo por criar 2 USER 's, em que são atribuídos a cada um deles um "author_id"e um "commiter_id"respetivamente. Assim, através de duas condições vamos verificar quando é que em cada um dos USER 's, o

type "Bot"está presente, adicionando o "repo_id"(apenas se não este estiver presente na "llr") correspondente a cada USER à nova Hash Table ("llr"). Por sua vez damos o tamanho da Hash Table "llr"e escreve-mo lo no novo ficheiro de "commandsX.txt".

Query 4: Nesta query simplesmente inicializamos duas variáveis, "commits"e "users", com o tamanho da Hash Table de commits e o tamanho da Hash Table de users respetivamente. Desta forma apenas dividimos uma variável pela outra e damos esse valor a outra variável "avg", que vai ser dada para output de forma a obtermos a média de commits por utilizador, como é pedido no enunciado.

3.1.2 Queries parametrizáveis

Query 5: Na query5 temos como o objetivo obter o nº de utilizadores mais ativos num intervalo de datas. Assim começamos por criar uma função auxiliar chamada "acrescentaN_commits", que irá receber como argumentos a Hash Table de users e a Hash Table de commits, juntamente com a data inicial e a data final dadas como input. Nesta função iremos percorrer uma lista ligada de commits "llc". Dentro desse ciclo criado vamos buscar a data em que foi criado o commit, ou seja, "created_at"e verificamos que esteja entre as duas datas dadas, a inicial e a final. Após esse passo comparamos o "author_id"com o "commiter_id"de forma a se forem iguais apenas incrementamos uma vez o contador "n_commits", ou se forem id's diferentes incrementamos duas vezes o contador uma vez que foram feitos dois commits. Por fim criamos uma lista ligada nova "lista"e ordena-mo la(atraves da "g_list_sort") por ordem crescente, com base no "n_commits".

Query 6: Começamos por percorrer a lista de commits, para verificar a linguagem do repositório associado a cada commit. Se a linguagem for a desejada, incrementamos a variável n_commits do utilizador que realizou esse commit. Mais tarde, convertemos os valores da Hash Table dos utilizadores numa lista ligada, de modo a ordená-la pelo número de commits e percorrê-la para escrever o output no ficheiro destinado N vezes.

Query 7: Para verificar quais os repositórios não ativos a partir de determinada data, começamos por verificar quais os ativos. Para esse efeito, percorremos uma lista ligada dos commits e adicionamos a uma Hash Table nova os repositórios que contêm commits a partir dessa data. Depois percorremos a lista de todos os repositórios para verificar quais pertencem à lista feita anteriormente (repositórios ativos). Os que não pertencem, escrevemos no ficheiro de output o respetivo ID e descrição do repositório.

Query 8: Para esta query, criamos uma struct nova, de modo a contabilizar o número de ocorrências das linguagens presentes no repositório.

```
typedef struct query8{
    char* language;
    int n_ocorr;
} *NLang;
```

Percorremos a lista de repositórios para verificar a data do repositório e para identificar cada linguagem e vamos adicionando numa *Hash Table* nova criada anteriormente, a struct identificada pela linguagem, se esta ainda não existir e caso o repositório tenha sido criado a partir de uma determinada data (fornecida no input). Caso exista é incrementado apenas o campo *n_occor*.

Posteriormente os valores dessa *Hash Table* são convertidos numa lista ligada, para que possamos ordenar e percorrer. Ordenamos por ordem decrescente de número de ocorrências de uma determinada linguagem e escrevemos no ficheiro respetivo, N linguagens, enquanto percorremos a lista para esse efeito.

Query 9: Percorremos a lista de commits e verificamos se o *owner* do repositório é amigo do utilizador que realizou o commit. Se sim, incrementamos o campo *n_commits* da *struct user* em questão, acedida na Hash Table de utilizadores. Posteriormente guardamos os valores dessa Hash Table numa lista ligada, para ordená-la por ordem decrescente em relação ao número de commits. São percorridos N nodos dessa lista, para que sejam escritos no novo ficheiro os dados necessários.

Query 10: Para a resolução desta query, bastou-nos percorrer a lista de commits, para isso convertemos a Hash Table numa GList. Depois ordenamos essa lista numa ordem decrescente pelo tamanho da mensagem e finalmente escrevemos no ficheiro apenas as primeiras N linhas necessárias para o top N introduzido no input. É escrito o ID do utilizador, o username, o tamanho da mensagem e o id do repositório em questão, tudo isto pode ser acedido pelas structs armazenadas na lista anteriormente referida. Sendo assim são percorridos N nodos dessa lista para a escrita no ficheiro.

Outras funções que achamos importantes mencionar e explicar:

getFriendsList(): Esta função é referente aos utilizadores. É usada para determinar a lista de amigos de um utilizador (utilizadores que ele segue e o seguem de volta mutuamente), dado uma lista de seguidores e uma lista de seguindo, adicionamos ao array de amigos do utilizador, ao mesmo tempo que incrementamos a variável n_amigos, se certo utilizador existir em ambas as listas.

initUsers(): Esta função está presente no ficheiro *queries.c* e serve para inicializar os commits dos utilizadores a 0, uma vez que há três *queries* parametrizáveis onde se conta o número de commits de um utilizador em certas condições diferentes. Sendo assim, para que não interferisse com o número de commits dessa segunda *query*, decidimos implementar esta função. É usada após as *queries* 5 e 6.

Resultados

4.1 Input e output

Input	**Output**
1	Bot : 72 Organization: 22903 User: 404410
2	7.10
3	0
4	1.875854
5 100 2010-01-01 2015-01-01	6958510;netfun;30 1095509;jalsk;30 2811088;naezin;30
6 5 Python	2856670;slevirus;30 1426783;macfire;30 3711439;iellswo;30 3186531;antcer1213;30 2609682;dataoko;30
7 2014-04-25	49520431;This is a basic Ember app using routes and bootstrap. 4766163;Step two in playing around with the web api 49544270;This project created a very basic CRUD application with firebase. 10533221;? 92654219;OpenSSL Project ...
8 3 2016-10-05	C++ None HTML
9 4	2856670;slevirus 1426783;macfire 3711439;iellswo 3186531;antcer1213
10 3	9153048;zhangbolily;643;59936802 9153048;zhangbolily;625;59936802 34973682;lvnkae;604;115835197

Tabela 4.1: Exemplos de *input* e *output* de execução das *queries*

4.2 Tempos de execução

O tempo de execução de ler os três ficheiros *.csv* e armazenar os dados nos respetivos módulos de dados foi de 2.67967 segundos. Na tabela seguinte estão apresentados os tempos de execução de cada query.

O hardware utilizado para os testes foi: processador *i5-8300H*, 16 *GiB* de memória *RAM* e *SSD*.

Query	**Tempo** (s)
Query1	0.164284
Query 2	0.061801
Query 3	0.132436
Query 4	0.000006
Query 5	0.203460
Query 6	0.219348
Query 7	0.219634
Query 8	0.096965
Query 9	0.217670
Query 10	0.161197

Tabela 4.2: Tempos de execução das *queries*

Conclusão

Concluindo, fizemos tudo o que era requerido no enunciado com sucesso. Conseguimos implementar todas as *queries*, que era o principal objetivo desta fase.

Apesar do tempo de execução destas ter sido menor do que um segundo, achamos que o desempenho e eficiência do nosso programa poderia ser melhor. Para isso, poderíamos ter usado diferentes estruturas de dados. É de salientar também que temos a noção que a implementação das *queries* estatísticas poderia ter sido feita de uma forma mais simples e eficiente. No entanto, fomos sempre alterando e refazendo nosso código de maneira a melhorar sempre, principalmente na resolução das *queries*. É também pertinente referir que a escrita do relatório permitiu-nos identificar vários erros nas nossas estratégias e código, levando assim a que alterassemos o mesmo.

A nossa maior dificuldade passou por pensar nas estratégias de resolução das *queries* de modo a que não baixasse o desempenho e eficiência do nosso código, no entanto sentimos que fomos bem sucedidos nesse quesito.