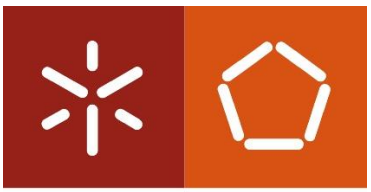


Universidade do Minho

Departamento de Informática

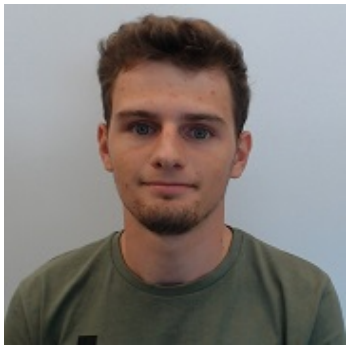
LI3 - Trabalho em C

Sistema de Gestão de Recomendações

Grupo Nº 89

18 de maio de 2021

1 Autores



Bohdan Malanka
a93300

Diogo Rebelo
a93278

Henrique Alvelos
a93316

Conteúdo

2 Introdução

O trabalho tem como objetivo processar grandes volumes de dados de uma forma rápida e eficiente. No fundo, pega em três ficheiros essenciais (Ficheiros .csv, em que contém os utilizadores, os negócios e as avaliações dadas pelos utilizadores aos negócios) e coloca em determinadas estruturas de modo a que a interação do Utilizador do programa seja mais fácil. Além disso, este projeto proporcionou o conhecimento de uma biblioteca muito pouco conhecida, que é o **GLIB.h**

3 Descrição da API

3.1 AuxStructs.h

Adaptamos a estrutura de dados do *GLIB* **GArray*** para guardar Strings. Decidimos utilizar esta estrutura porque é uma alternativa ao **char****, o que iria necessitar sempre o uso de *malloc* com um tamanho mais preciso e posteriormente *realloc*, se fosse necessário adicionar mais do que o tamanho inicialmente alocado. Assim, com o **StrArray** (**GArray*** de Strings) não precisávamos de nos preocupar, pois bastava utilizar os métodos disponíveis na biblioteca *glib.h*. Além disso, este ficheiro tem várias funções auxiliares para gerir **GPtrArray** e tratar de Strings.

3.2 User.h

```
struct user {
    char* id;
    char* name;
    char* friends;
    GSList *businessReviewed;
};
```

Figura 1: Struct User

Estrutura de dados responsável por guardar a informação de um **USER**. Para além de ter o ID, Name e Friends (ainda sem o parcing) também contém um campo, responsável por guardar os *ID's de BUSINESS* aos quais o **USER** avaliou. Para isso, utilizamos um **GSList***. Assim, a Query 4 é resolvida de forma mais fácil.

3.3 Business.h

```
struct business{
    char* categories;
    char* business_id;
    char* name;
    char* city;
    char* state;
    int totalReviews;
    float totalStars;
};
```

Figura 2: Struct Business

Estrutura de dados responsável por guardar a informação de um **BUSINESS**. Para além de ter o ID, Name, City, State e Categories (ainda sem o *parcing*) adicionamos dois campos: um **int** totalReviews, que guarda o número total de **REVIEW** deste *BUSINESS*, e um **float** totalStars para guardar o número total de estrelas que o *BUSINESS* recebeu. Assim, para calcular as estrelas médias do *BUSINESS* basta dividir totalStars por totalReviews. Logo, as Queries 5, 6 e 8 são resolvidas de forma mais rápida.

3.4 Review.h

```
struct review{
    GString *review_id;
    GString *user_id;
    GString *business_id;
    gfloat stars;
    gint useful;
    gint funny;
    gint cool;
    GString *date;
    GString *text;
};
```

Figura 3: Struct Review

Estrutura de dados responsável por guardar a informação de um **REVIEW**. Não adicionamos nada a mais do que o enunciado tinha nesta *struct*, mas em vez de **int** e **char*** convencionais do C utilizamos **gint** e **GString** do **GLIB**, não só por curiosidade e na tentativa de aprender mais sobre o mesmo, mas também por causa do tamanho da **GString**, que é direto e não necessita da chamada da função *strlen*.

3.5 Users.h

```
struct users{
    GHashTable* userHashT;
};
```

Figura 4: Struct Users

Estrutura de dados que contém uma **GHashTable*** e é responsável por guardar os **USER**. A key é o *ID do USER* e a value é uma **struct USER***. Visto que estamos a processar uma grande quantidade de dados, a *Hash Table* é uma ótima estrutura, dado que, se tivermos a key, a procura é feita em tempo constante (O(1)).

3.6 Businesses.h

```
struct businesses{
    GHashTable* hashT;
};
```

Figura 5: Struct Businesses

Estrutura de dados que contém uma **GHashTable*** e é responsável por guardar os **BUSINESS**. A key é o *ID do BUSINESS* e a value é uma **struct BUSINESS***. Escolhemos esta estrutura pelo mesmo motivo de escolha em **USERS**.

3.7 Reviews.h

Figura 6: Struct Reviews

Estrutura de dados que contém uma **GHashTable*** e é responsável por guardar as **REVIEW**. Em que a key é o *ID do REVIEW* e a value é uma **struct REVIEW***. Escolhemos esta estrutura pelo mesmo motivo de escolha em **USERS**.

3.8 TopBusiness.h

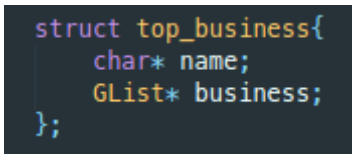


Figura 7: Struct TopBusiness

Estrutura de dados auxiliar que resolvemos criar para a resolução das queries 6 e 8. **TOPBUSINESS** é um *pointer* para uma *struct* que contém os atributos Name (*String*) e businessList (**GList***). A ideia é o Name pertencer a uma City ou Category, e a businessList ter uma lista de **BUSINESS** que pertencem à City ou Category.

3.9 TopBusinesses.h

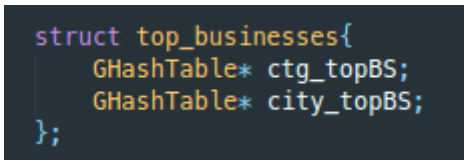


Figura 8: Struct TopBusinesses

Estrutura de dados que contém duas **GHashTable***, qua são responsáveis por guardar os **TOPBUSINESS**, uma para as cidades e outra para as categorias. A key é o *NAME do TopBusiness* e a value é uma **struct TOPBUSINESS***. Inicialmente pensamos em fazer uma só *Hash Table* e ter um indicador se é cidade ou categoria, mas, por mais difícil que seja uma cidade ter o mesmo nome que uma categoria, decidimos separar e organizar em duas *Hash Table*.

3.10 TABLE.h

Estrutura de dados responsável por guardar a informação sob a forma de uma tabela, utilizada para o output das queries. A **TABLE** é conposta por dois **GPtrArray** em que o primeiro corresponde ao *Header* e o segundo ao *Content*. Assim, no *Header* (que podia ser um **StrArray**), cada índice corresponde a uma String que é o cabeçalho da respetiva coluna. Já no *Content*, cada índice é uma linha que contém um **StrArray** com as respetivas colunas do *Header*, assemelhando-se a uma matriz. Assim sendo, definimos algumas macros para um tamanho fixo da largura de cada coluna:

Tipo de Tabela	Tamanho
ID's	22 **char**
Name	57 **char**
City	35 **char**
Stars	5 **char**
State	5 **char**
Total Reviews	13 **char**

Tabela 1: Tamanho de cada coluna

```
struct table {
    GPtrArray* header;
    GPtrArray* content;
};
```

Figura 9: Struct Table

Ao imprimir a **TABLE** nota-se que pode ocorrer alguma leve deformatação, achamos que é por causa de caracteres especiais como acentos, tils, marca registada, entre outros. Por fim, possui funções não usadas como *PrintTable* que, por lapso, não foram eliminadas. Além disso possui algumas funções responsáveis pelos comandos do interpretador, como *fromCSV* e *toCSV*.

3.11 Pagination.h

```
struct window{
    TABLE table;
    int page;
    int last_page;
};
```

Figura 10: Struct Windows

API responsável pela paginação de uma **TABLE**, possui uma **struct Window** que contém a dita **TABLE** e um inteiro para a pagina atual e outro inteiro para a última página. Decidimos que a nossa paginação irá imprimir 12 linhas por pagina da **TABLE** e implementamos comandos como: avançar/retroçeder de página, ir para a primeira/última página, um indicador para a pagina atual e Quit. Neste mesmo módulo também implementamos funções de *printf* do display principal do programa, que ponderamos fazer num módulo à parte (*view.c*) mas como este já tinha funções de *printf* então ficou neste módulo.

3.12 Interpreter.h

```
struct variables {
    GHashTable* varHash;
};
```

Figura 11: Struct Variables

Módulo que tem o interpretador do programa. Faz *parcing* do comando digitado e tenta perceber se é um comando válido ou não. Os comandos disponiveis são:

1. Queries;
2. Comandos do enunciado;
3. Count (devolve o número de linhas da **TABLE** que no nosso caso corresponde ao tamanho do *Content*).

3.13 SGR.h

```
struct sgr{
    Users us_s;
    Businesses bs_s;
    Reviews rw_s;
    TopBusinesses top_bs;
};
```

Figura 12: Struct SGR

Módulo responsável pela base de dados do programa. Faz a leitura dos ficheiros de input e guarda em *Hash Table* os respetivos **USERS**, **BUSINESSES**, e **REVIEWS**. Os **TOPBUSINESS** são carregados quando é executada uma Query que necessita dessa informação. A API também contém a implementação de cada Query do programa.

4 Complexidade das estruturas e Otimizações das Queries

4.1 Query 1

São três ciclos *while* por cada ficheiro logo a complexidade é O(N). A leitura é feita por linhas (função *getline*), ou seja, não é um buffer com comprimento estático o que é melhor para ler linhas que qualquer tamanho. Uma otimização que foi feita no load dos ficheiros é descartar os Friends do ficheiro *user.csv*, embora tenhamos a opção de os carregar.

4.2 Query 2

Para a sua realização, percoremos a *Hash Table* dos **BUSINESSES** e verificamos se o Name faz match com a letra pedida. Portanto, a complexidade é O(N). Não fizemos otimização para esta, deixamos tal como foi feita de primeira vez. No entanto, uma possível otimização podia ser criar um módulo auxiliar que continha uma *struct* com as tabelas de nomes dos **BUSINESS** de cada letra do alfabeto e que carregava-se na primeira utilização da query, assim, nas proximas utilizações, seria feito de um modo mais rápido.

4.3 Query 3

Para esta query, vamos buscar o respetivo **BUSINESS** à *Hash Table* e retiramos a sua informação que, de seguida, inserimos no **array**. Como o *lookup* na *Hash Table* é em tempo constante e só se faz cinco inserções no **GPtrArray**, a complexidade desta query é O(1). Quanto à otimização, achamos que não há grandes mudanças possíveis.

4.4 Query 4

Aqui pesquisamos pelo **USER** pretendido (O(1)) e retiramos a **GSList*** que contém já os *ID do BUSINESS* que avaliou, depois é só percorer essa lista (O(N)) e buscar o respetivo *Name do BUSINESS*, logo a complexidade é O(N) em que N é o numero de business_id que a lista contém. A otimização que fizemos foi adicionar à **struct USER*** essa tal **GSList*** com os *ID* já prontos.

4.5 Query 5

Para esta query, nós percorremos a *Hash Table* dos **BUSINESS** e veificamos se o **BUSINESS** pertence a dada City e se tem tantas ou mais Stars pretendidas. Assim sendo, a complexidade deste algoritmo é O(N). Uma otimização possível seria, por exemplo, ter no nosso módulo **TOPBUSINESSES**, onde temos a **Hash Table** de City e Categories, neste caso a primeira ter cada **GList*** de cada cidade ordenada por ordem decrescente de Stars e era só percorrer essa lista até a o número de estrelas ser maior ou igual.

4.6 Query 6

Nesta query, para a sua realização, percorre-se a *Hash Table* das City e, por cada cidade, a sua **GList*** dos **BUSINESS** e depois retiramos a informação necessária dos N primeiros elementos. Esta **GList*** tem os **BUSINESS** ordenados por ordem decrescente pelo número de estrelas dadas. Como o nome da City não é pedida no enunciado, para sabermos depois na impressão da TABLE quando ocorre a mudança da City nós colocamos a String SWITCHER no final da informação, assim, se esse codigo aparece, o separador da linha vai ser diferente. Quanto à complexidade, achamos que não chega a ser O(N^2), porque o ciclo exterior percorre o número de cidades existentes e o ciclo interior faz N (top pedido) iterações, portanto é O(N). Para otimizar isto podia ser como suprareferimos: fazer o sort de cada **GList*** depois de ter carregado tudo, visto que o **GLIB** aconcelha não fazer *insert_sorted* a cada elemento inserido.

4.7 Query 7

Aqui nós percorremos os **USER** e em cada verificamos se este esteve em mais do que um State, ou seja, percorremos a lista dos **BUSINESS** que avaliou e verificamos se o State anterior é diferente do atual a cada iteração, mas em média o ciclo interior executa 2 ou 3 vezes. Como esta query só tem o **SGR** como input e o resultado é sempre o mesmo se não houver novas inserções de **USER**, uma otimização seria no interpretador onde são guardados os resultados das queries, verificar se isto já foi calculado, mas para isso tínhamos que modificar a estrutura de armazenamento das **TABLE**, porque senão teriamos que verificar o *Header* de cada **TABLE** guardada.

4.8 Query 8

Esta ficou bem simples com o nosso módulo **TOPBUSINESSES**, é só ir à respetiva *Category* que está na *Hash Table* e ordenar os **BUSINESS** por ordem decrescente e retirar os top N. A complexidade é O(N), em que N é o número do top pedido. A otimização para isto é a que já referimos: em vez de estar sempre a ordenar a cada inserção, podia já estar ordenado, depois de carregar tudo.

4.9 Query 9

O algoritmo para esta query ficou um pouco pesado, visto que nós percorremos os **REVIEW** da *Hash Table* e depois em cada **REVIEW** comparamos se a palavra se encontra do texto do **REVIEW**, para isso fazemos um ciclo para fazer o parcing mas antes fazemos mais um ciclo para remover a pontuação e substituir por espaços. Por isso, são três ciclos aninhados, mas não chega a ser O(N^3), porque na primeira ocorrência da palavra sai logo do ciclo. Portanto, para otimizar isto, podiamos ter feito a remoção da pontuação dos textos ao carregar e não sempre que se pede a query.

4.10 Tempo de execução das várias queries

Query	Função	Tempo (em segundos)
2	*businesses_started_by_letter(sgr,'a');*	0.128
3	*business_info(sgr,"bvN78flM8NLprQ1a1y5dRg");*	0.000
4	*businesses_reviewed(sgr,"ak0TdVmGKo4pwqdJSTLwWw");*	0.000
5	*businesses_with_stars_and_city(sgr,3,"Austin");*	0.120
6	*top_businesses_by_city(sgr,5);*	0.175
7	*international_users(sgr);*	2.271
8	*top_businesses_with_category(sgr,4,"Restaurants");*	0.293
9	*reviews_with_word(sgr,"Jesus");*	102.929

5 Apreciação crítica

Em geral estamos contentes com o resultados, pois achamos que superamos o desafio. No entanto, existem algumas falhas no projeto: Uma delas foi no commit final, o Bohdan fez o push para adicionar a documentação mas também foram para o repositório funções que já foram apagadas e sendo assim, temos funções que não são usadas. Outro problema é com o **Valgrind** e as *memory leaks*, inicialmente tinhamos preocupação com isso, para que à medida que se fosse acrescentado código não houvessem problemas, mas depois deparamo-nos com um problema que não conseguámos resolver e decidimos avançar e prioretizar a finalização do trabalho. O mesmo caso para o encapsulamento, que foi violado em alguns *getters* (como por exemplo nas **Hash Table**) Quanto ao código em si, achamos que em algumas partes foi um pouco naive, mas tentamos seguir o exemplo que os professores disponibilizaram sobre o **GLIB** e também as boas práticas de C. Por fim, quanto á modularidade, achamos que se possa ser melhorado e que dê para dividir melhor. Por exemplo, no nosso trabalho a paginação já inclui a *view* do projeto, o interpretador e controlador.

6 Conclusão

Em suma, aprendemos bastante com este projeto, pois conhecemos uma API nova (glib). Além disso, também ficamos familiarizados com gestão e processamento de grande quantidade de dados que se aproxima muito da realidade atual. Para finalizar, se refizéssemos de novo ou se houvesse mais tempo, focaríamos mais no **Valgrind**, tratariamos a situação da modularidade, de ter uma melhor organização, isto para além das otimizações supracitadas em cima.