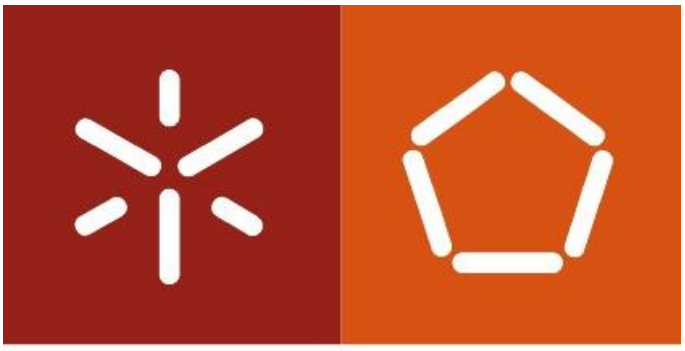

Universidade do Minho[8]

Departamento de Informática[1]

Projeto de Laboratórios de Informática III
Grupo 15

João Barbosa (a93270) Simão Cunha (a93262)
Tiago Silva (a93277)

05 de maio de 2021

Conteúdo

1 Introdução

O presente relatório descreve o projeto realizado no âmbito da disciplina de Laboratórios de Informática III (LI3), ao longo do segundo semestre, do segundo ano, do Mestrado Integrado em Engenharia Informática da Universidade do Minho. Tem como objetivo criar um sistema capaz de processar dados contidos em ficheiros CSV para responder a um conjunto de questões de forma eficiente, utilizando a linguagem de programação C.

1.1 Descrição do Problema

Os ficheiros em formato CSV que deveriam ser processados continham informação referente ao website *Yelp* [9], onde utilizadores de todo o mundo podem avaliar vários estabelecimentos, escrevendo as suas críticas.

Em concreto, o trabalho prendia-se em extrair a informação necessária dos vários ficheiros, de forma a conseguirmos responder a 9 querys obrigatórias e outras funcionalidades relacionadas com o conteúdo das mesmas da forma mais eficiente possível, isto é, tendo especial atenção ao tempo de execução do programa, bem como ao encapsulamento dos dados.

1.2 Ficheiros CSV

Os 3 ficheiros necessários para a boa execução do programa são *business.csv*, *reviews.csv* e *users.csv* (nomes meramente ilustrativos), que continham as informações dos negócios, das críticas e dos utilizadores, respetivamente.

No ficheiro *business.csv*, extraímos o identificador(`business_id`), o nome, a cidade, o estado e as categorias em que cada estabelecimento se insere.

No ficheiro *reviews.csv*, recolhemos a informação relativa ao identificador da crítica (`review_id`), do identificador do utilizador que fez a classificação(`user_id`), do identificador do negócio que obteve a crítica (`business_id`), do valor das estrelas (0 a 5) atribuídas, do número de pessoas que acharam a review útil, do número de pessoas que acharam-na engraçada, do número de pessoas que a acharam *cool*, da data da publicação da review e do comentário feito pelo utilizador.

Por último, retiramos do ficheiros dos utilizadores, o identificador de um utilizador (`user_id`), do seu nome e da sua lista de amigos.

NB: Os vários parâmetros dos ficheiros vêm separados por ';'.

1.3 Conceção da Solução

Até obtermos a solução final, esta sofreu várias alterações. Numa primeira fase, extraímos a informação dos ficheiros, dando-lhes um buffer de tamanho fixo, mas observámos que não conseguíamos ler o ficheiro de exemplo dos utilizadores por ser demasiado pesado. Decidimos então mudar para um tamanho do buffer individual para cada tipo de ficheiro. No entanto, apesar de conseguirmos ler os ficheiros exemplos, queriamos uma resolução mais genérica, assim, acabando por utilizar a função `getline`[2] com um buffer inicialmente a NULL e deixar esta ao encargo da alocação do buffer. No ínicio guardávamos toda a informação em arrays, mas devido à ineficiência de procura, optamos pela utilização da API GLIB [7], mais precisamente *hash tables*, utilizando como chaves os identificadores, melhorando, assim, o nosso tempo de procura da informação. De seguida, procedemos para a implementação das queries, da criação do tipo de dados TABLE e do interpretador de comandos.

O relatório está organizado da seguinte forma: a Secção 2 descreve as estruturas de dados adotadas, a Secção 3 apresenta e discute a solução proposta para a resolução do problema. O relatório termina com conclusões na Secção 4, onde é também apresentada uma análise crítica aos resultados obtidos em testes de performance.

Link do nosso repositório : `https://github.com/dium-li3/Grupo15` [3].

2 Organização dos Dados

Para desenvolvermos este trabalho adotamos as seguintes estruturas de dados:

2.1 Tipo de Dados Concretos

```
typedef GHashTable* USERS;
typedef GHashTable* REVIEWS;
typedef GHashTable* BUSINESSES;

typedef struct sgr {
    USERS users;
    BUSINESSES businesses;
    REVIEWS reviews;
}*SGR;
```

Os dados necessários para responder às queries foram armazenados na estrutura `SGR`, a estrutura de dados principal do nosso trabalho. De modo a respondermos às queries da forma mais eficiente possível, utilizamos estruturas de dados já existentes e, para isso, recorremos ao GLIB, uma biblioteca que fornece os blocos de construção do núcleo da aplicação para bibliotecas e aplicações escritas em C [7].

Em primeiro lugar, para as estruturas **users**, **reviews** e **businesses** utilizamos uma GHashTable [5] para armazenar cada tipo de dados, isto é, guarda numa *tabela de hash* as respetivas structs, onde as keys de cada tabela são os identificadores de cada tipo de dados (i.e. user_id, review_id ou business_id).

A razão para esta escolha deve-se ao facto de os algoritmos de procura, de inserção e de remoção de elementos ser constante ($\mathcal{O}(1)$) em tempo médio e, por isso, assintoticamente melhor do que os algoritmos em arrays (dinâmicos ou estáticos) ou em listas ligadas (por exemplo). Utilizamos como os valores de *hash* os id's das *reviews*, dos *businesses* ou dos *users*, por estes serem únicos dentro dos seus respetivos ficheiros. Assim, com estes *hash values*, conseguimos trabalhar nos problemas de forma mais rápida e simples pois conseguimos aceder aos elementos da estrutura de dados muito eficientemente com o recurso a funções já existentes no glib (por exemplo, g_hash_table_lookup [6], g_hash_table_foreach [4], ...).

2.2 Estruturas de Dados Complementares

```
typedef struct user {
    char* user_id;
    char* name;
    char* friends;
}*USER;
```

A estrutura de dados `users` contém o ID do utilizador, o seu nome e a sua lista de amigos.

```
typedef struct business {
    char* business_id;
    char* name;
    char* city;
    char* state;
    char* categories;
}*BUSINESS;
```

A estrutura de dados `business` contém o ID do negócio, o seu nome, a cidade onde se localiza, o estado onde se situa e as categorias em que se insere.

```
typedef struct review {
    char* review_id;
    char* user_id;
    char* business_id;
    float stars;
    int useful;
    int funny;
    int cool;
    char* date;
    char* text;
}*REVIEW;
```

A estrutura de dados `review` contém o ID da review (*review_id*), o id do utilizador que fez a criica (*user_id*) , o id do negócio que recebeu a review (*business_id*), o valor das estrelas (0 a 5) atribuídas, o número de pessoas que acharam a review "útil", o número de pessoas que acharam-na "engraçada", o número de pessoas que a acharam "cool", a data da publicação da review e o comentário feito pelo utilizador (*text*).

```
typedef char* string;

typedef struct table {
    string *info;
    string *titles;
    string title_lines;
    int lines;
    int columns;
    int size;
    int number_of_pages;
    int lines_for_page;
    int *biggest_length;
    int buffer_size;
}*TABLE;
```

A estrutura de dados `table` serve representar a informação calculada pelas queries de uma forma apelativa para o utilizador. Neste tipo de dados, podemos inserir *strings* nos títulos e nas diversas linhas da TABLE, definir o número de colunas e de linhas, o número de páginas para tornar mais apelativa a leitura da informação, o número de linhas a aparecer por página (que neste caso vão ser 10) e o tamanho do buffer para carregar uma TABLE. Além disso, podemos saber o comprimento de uma linha para desenhar a TABLE de forma a que cada informação fique centrada na respetiva linha.

```
typedef struct hashdata {
    void* table;
    void* datas[5];
}*HASHDATA;
```

Tendo em conta a função que o GLIB nos oferece (g_hash_table_foreach [4]), onde dada uma hash, uma função e um valor que essa função possa receber, aplica a todos os elementos da *hash* essa função. Aqui surgiu a necessidade de passar mais que um valor e com isso surgiu a `struct hashdata` que não só guarda espaço para 5 valores, como ainda guarda espaço para a TABLE. Apesar de esta ser um `void*` (tal como as *datas*) e poder ser guardada dentro desse array, decidimos destacar dentro da struct como um "*data*"especial para facilitar o seu manuseamento.

3 Implementação

3.1 Modularização Funcional

O trabalho é composto por vários módulos listados abaixo, que foram surgindo consoante as necessidades que fomos detetando.

- model:
  - `business.c`: contém todas as funções para aceder à struct business;
  - `user.c`: contém todas as funções para aceder à struct user;
  - `review.c`: contém todas as funções para aceder à struct review;
  - `reading.c`: contém todas as funções para carregar e processar dados oriundos dos ficheiros CSV;
  - `hashes.c`: contém todas as funções para aceder às diferentes *tabelas de hash*;
- controller:
  - `hashcontrol.c`: contém todas as funções auxiliares de acesso a *tabelas de hash* que intervêm nas querys pedidas;
  - `interpreter.c`: contém todas as funções necessárias para o interpretador de comandos, assim como as outras funcionalidades para o menu;
  - `sgr.c`: contém todas as funções responsáveis pelo acesso às diferentes *tabelas de hash* e pelas querys pedidas pela equipa docente;
- view:
  - `table.c`: contém todas as funções necessárias para representar a informação oriunda das queries;
  - `show.c`: contém todas as funções necessárias para a componente gráfica do programa;

Os vários módulos criados podem ser vistos como unidades interdependentes que se complementam, cada uma com objetivos específicos, que interagem e que estão ligadas entre si apenas por funções (única interface), garantindo o encapsulamento dos dados.

Cada módulo tem uma única funcionalidade, um objetivo específico, como, por exemplo, responder a uma query ou realizar a leitura de dados. De facto, com a divisão física dos ficheiros tentamos garantir a abstração dos dados, bem como facilitar a reutilização do código.

3.2 Abstração de Dados

Conforme reparou na secção anterior, uma das estratégias adotadas para garantir a abstração dos dados foi através da modularização funcional do código. Aliado a isso, o encapsulamento de todos os dados foi garantido com "*getters*"e "*setters*". De facto, os ficheiros *header* criados impossibilitam que os atributos das diversas estruturas de dados criadas sejam acedidos diretamente. Por exemplo, a estrutura de dados que contém todos as informações relevantes e referentes ao usuário foi definida no ficheiro `review.c` da seguinte forma:

```
typedef struct review {
    char* review_id;
    char* user_id;
    char* business_id;
    float stars;
    int useful;
    int funny;
    int cool;
```

```
    char* date;
    char* text;
}*REVIEW;
```

sendo que , por exemplo, o acesso ao campo `review_id` da `struct review` só é possível através das funções `char* get_review_id(REVIEW r)` que retorna uma cópia desse campo, ou `void set_review_id(REVIEW r, char* id)` que atribui um valor a esse campo, já que o ficheiro header review.h possui apenas a declaração `typedef struct review *REVIEW`.

3.3 Queries

Query 1

"Dado o caminho para os 3 ficheiros (Users, Businesses, e Reviews), ler o seu conteúdo e carregar as estruturas de dados correspondentes."

Nesta query começamos por inicializar a variável `SGR` através da `init_sgr()`, que cria uma *hash table* para cada um dos três tipos pretendidos (`USERS`, `BUSINESSES` e `REVIEWS`), e, em seguida, dividimos o processo de leitura em 3 funções que carregam os respetivos tipos de dados separadamente. Note que, ao para carregar a *tabela de hash* REVIEWS, este necessita de validar a pré-existência dos ID's da *review*, do *business* e do *user* noutras*tabelas de hash*, sendo, por isso, o último a ser lido. As funções de leitura são muito semelhantes nos três casos, exceto nos detalhes por exclusividade/diferença das estruturas. Estas utilizam da função `getline()`[2], que lê linha a linha do ficheiro, cria a respetiva estrutura (durante a criação efetua-se a validação dos dados) e, por fim, coloca na respetiva hash.

Query 2

"Determinar a lista de nomes de negócios e o número total de negócios cujo nome inicia por uma dada letra. A procura não deverá ser case sensitive."

A estratégia para esta query passa por inicializar a estrutura TABLE, através da `init_table()`, colocar os titulos necessários ("business_id, name"), e, em seguida, percorrer a *hash table* BUSINESSES e colocar dentro da TABLE os id's e os nomes dos negócios que começam pela letra pedida.

Query 3

"Dado um id de negócio, determinar a sua informação: nome, cidade, estado, stars, e número total reviews."

Nesta query (e nas seguintes), o processo da criação da TABLE é semelhante à da query anterior. Passamos por encontrar o BUSINESS através do seu ID e da *hash table* BUSINESSES (com recurso da função g_hash_table_lookup do GLIB [6]), e, em seguida, percorremos a *hash table* REVIEWS para calcular as estrelas médias atribuídas a esse negócio. De seguida, adicionamos toda a informação calculada numa TABLE.

Query 4

"Dado um id de utilizador, determinar a lista de negócios aos quais fez review. A informação associada a cada negócio deve ser o id e o nome."

Para esta query, percorremos a *hash table* REVIEWS, e, para cada REVIEW, verificamos se o ID do utilizador pedido existe. Em caso afirmativo, através do ID de negócio, vamos procurar na *hash table* BUSINESSES o negócio que obteve uma *review*, e colocando numa TABLE o ID do negócio e o seu nome.

Query 5

"Dado um número n de stars e uma cidade, determinar a lista de negócios com n ou mais stars na dada cidade. A informação associada a cada negócio deve ser o seu id e nome."

A estratégia desta query passa por criar uma hash table que fosse ser quase uma "*cópia*"de BUSINESSES, só que com um filtro relativo à cidade pedida. Para isso, depois de criar a *hash*, percorremos a *hash table* BUSINESSES e colocamos dentro da *hash* criada apenas os negócios da cidade pedida. De seguida, percorremos a *hash table* REVIEWS de modo a calcular o número médio de estrelas atribuídas aos negócios, e por fim, percorremos a *hash table* criada inicialmente, e colocamos dentro de uma TABLE os negócios que tiverem um número de estrelas médias maior ou igual ao número de estrelas pedido.

Query 6

"Dado um número inteiro n, determinar a lista dos top n negócios (tendo em conta o número médio de stars) em cada cidade. A informação associada a cada negócio deve ser o seu id, nome, e número de estrelas."

Nesta query, utilizamos o raciocínio da query anterior só que em vez de 1 cidade, temos n cidades. Assim, a ideia que tivémos foi criar n *hash tables*, e, como modo de organização, colocamos essas *tabelas de hash* dentro de uma *hash* global que tivesse como chave o nome da cidade. Depois, assim como na query 5, percorremos o BUSINESSES e, em seguida, o REVIEWS, com o mesmo propósito. Por fim, percorremos a *hash* criada, que por sua vez percorre cada uma das suas *hashes* de valor, e coloca por ordem (1^{o} por cidade, 2^{o} por valor médio de estrelas) dentro da TABLE o top pedido.

Query 7

"Determinar a lista de ids de utilizadores e o número total de utilizadores que tenham visitado mais de um estado, i.e. que tenham feito reviews em negócios de diferentes estados."

A solução começou por se criar uma *hash table* que guarda um inteiro (número de cidades visitadas) e uma `string` (cidade visitada) numa estrutura de dados HASHDATA. Já a chave será o ID do utilizador. Em seguida, percorremos a *hash table* REVIEWS e em cada REVIEW (com o acesso à HASHDATA através de um `g_hash_lookup` com o ID do utilizador na *hash table* criada), verificamos a sua existência. Se esse HASHDATA não existir, significa que é a primeira vez que encontramos este `user` e então adicionamos à hash table com o valor da cidade visitadas a 1 e colocamos a cidade na string (obtemos a cidade através do identificador do negócio e a hash table BUSINESSES). Se existir, então vemos se este utilizador já passou por 2 cidades; se sim não fazemos nada, pois este já é internacional; se não então comparamos a cidade visitada da hashdata e a cidade da crítica, e caso sejam diferentes aumentamos 1 ao número de cidades visitadas. Por fim é só percorrer a hash table criada e adicionar à TABLE os idenficadores dos utilizadores que tiverem 2 nas cidades visitadas. NB: Nota que como não pede o número de cidades visitadas, não nos vimos na necessidade de ver se o utilizador passou por mais que 2 cidades. Caso pedisse teriamos que usar um array de strings (cidades visitadas) e ir incrementando o número de cidades de visitadas, sempre que fosse uma cidade nova.

Query 8

"Dado um número inteiro n e uma categoria, determinar a lista dos top n negócios (tendo em conta o número médio de stars) que pertencem a uma determinada categoria. A informação associada a cada negócio deve ser o seu id, nome, e número de estrelas."

Aqui aplicamos um pouco do raciocínio da query 5, onde criamos uma hash table em que a chave eram os identificadores de negócio e o valor uma HASHDATA com o nome, número

total de estrelas e número total de reviews. Em seguida percorremos a hash BUSINESSES e colocamos os negócios dentro da hash criada que tiverem a mesma categoria pedida. Percorremos a hash REVIEWS para calcular o número total de estrelas/total reviews dos negócios. Por fim percorremos a hash criada para ordenar pelo maior número médio de estrelas, os negócios num top definido e colocamos dentro da TABLE.

Query 9

"Dada uma palavra, determinar a lista de ids de reviews que a referem no campo text."

A resolução desta query passou por percorrer o REVIEWS e colocar dentro da TABLE os IDS das críticas que contêm no seu texto a palavra pedida.

3.4 Testes de performance

Query 1	Query 2
Carregar os ficheiros →7.85 s	businesses_started_by_letter(sgr,'w') → 27.080 ms businesses_started_by_letter(sgr,'t') → 24.138 ms businesses_started_by_letter(sgr,'a') → 23.390 ms

Query 3	Query 7
business_info(sgr,"N3_Gs3DnX4k9SgpwJxdEfw") → 136.385 ms business_info(sgr,-l5w8_vwKDSUlpr9FSQoqA") → 130.396 ms business_info(sgr,"6fT0lYr_UgWSCZs_w1PBTQ") → 131.130 ms	international_users(sgr) → 719.160 ms

Query 4
businesses_reviewed(sgr, "q_QQ5kBBwlCcbL1s4NVK3g") → 144.893 ms businesses_reviewed(sgr, "RNm_RWkcd02Li2mKPRe7Eg") → 138.027 ms businesses_reviewed(sgr, "bUHweiErUJ36WGeNrPmEbA") → 137.753 ms

Query 5	Query 9
businesses_with_stars_and_city(sgr,4,"Portland")→212.109 ms businesses_with_stars_and_city(sgr,1,"Portland")→210.217 ms businesses_with_stars_and_city(sgr,4,"Atlanta")→214.209 ms	reviews_with_word(sgr,"and")→739.166 ms reviews_with_word(sgr,"well")→1572.931 ms reviews_with_word(sgr,"cook")→1790.292 ms

Query 6	Query 8
top_businesses_by_city(sgr,10)→607.508 ms top_businesses_by_city(sgr,50)→648.791 ms top_businesses_by_city(sgr,100)→672.220 ms	top_businesses_with_category(sgr,4,"Restaurants")→290.798 ms top_businesses_with_category(sgr,1,"Restaurants")→297.053 ms top_businesses_with_category(sgr,4,"Pet")→199.174 ms

Especificações da máquina onde os testes foram executados:

- Nome: *MSI GL63 8SE*
- Sistema operativo: *Ubuntu*
- Processador: *Intel Coffeelake i7-8750H + HM370*
- RAM: *16GB DDR4 2666MHz*
- Placa gráfica: *NVidea GeForce RTX 2060, com 6GB GDDR6*

3.5 Estratégias para melhorar o desempenho

Como dito anteriormente, um dos nossos problemas de desempenho era na leitura e no armazenamento dos ficheiros. Começamos por usar um buffer enorme fixo para a leitura de uma linha, mudando posteriormente para a *getline* que aloca espaço no buffer à medida que necessita. Já no que toca ao armazenamento, numa fase inicial guardávamos a informação em arrays, mas decidimos utilizar as *hash tables* fornecidas pela API GLIB para que o processo de procura fosse mais rápido. Falando das queries, houve uma que sofreu drásticas mudanças devido ao seu desempenho lento, a query 6 (3.3). A príncipio a sua resolução tinha as seguintes etapas: percorrer a hash BUSINESSES e em cada BUSINESS calcular o seu average star através da hash REVIEWS, e por fim realizar o top. Isto obviamente era extremamente ineficiente, pois percorriamos a hash REVIEWS quantos negócios tivessemos, logo decidimos por obtar por outra estratégia, mais elaborada. Primeiro criamos uma hash (em que as chaves são as cidades) que tivesse como valor outras mini hashes (em que as chaves são os identificadores) com valor de HASHDATA (uma struct auxiliar que neste caso teria o business_id, o business_name, um apontador para o número total de estrelas, e o número de reviews). Depois percorriamos a hash BUSINESSES e em cada BUSINESS criavamos uma HASHDATA, e através da sua cidade e do seu identificador inseriamos na respetiva hash table. Em seguida percorriamos a hash REVIEWS para calcular os average stars de todos os businesses guardados na criada hash table. Por fim percorremos a hash das cidades e colocamos por ordem o top dos businesses, como pretendido na query. Apesar de mais elaborada e se calhar confusa estratégia, a eficiência melhorou pois já não precisamos percorrer tantas vezes desnecessariamente a hash REVIEWS, ainda que tenhamos de abdicar de um pouco mais de memória ao criar hash tables de auxílio.

4 Conclusões

Face ao problema apresentado e analisando criticamente a solução proposta concluímos que cumprimos todas as tarefas, conseguindo atingir os objetivos definidos. No decurso do projeto recorremos aos conhecimentos adquiridos nas unidades curriculares de Algoritmos e Complexidade, Programação Imperativa, Laboratórios Informática II, bem como a decorrente Laboratórios Informática III, onde aprendemos conceitos novos como a Modularidade e Encapsulamento.

Todavia, entendemos que há alguns aspetos da nossa solução que, eventualmente, poderiam ser melhorados, mas que com a falta de tempo e/ou conhecimento não foi possível. Infelizmente, não tivemos tempo para testar, em termos de desempenho, mais soluções possíveis que pudessem fazer diferença a nível de tempo em todas as queries, por se calhar termos gasto esse tempo em coisas mais triviais, como à construção das nossas estruturas de dados, bem como à forma como iríamos ler e processar os dados dos ficheiros CSV, pois verificamos que estas duas ações iriam ter um peso determinante em termos de eficiência no trabalho.

Em suma, não obstante as potenciais melhorias que poderiam ser feitas no programa, os testes por nós realizados, nas nossas máquinas, atingiram um tempo de execução que consideramos bastante aceitável.

Referências

[1] Dpt informática. https://www.di.uminho.pt/.

[2] Getline. https://man7.org/linux/man-pages/man3/getline.3.html.

[3] Github. https://github.com/dium-li3/Grupo15.

[4] Glib foreach. https://developer.gnome.org/glib/stable/glib-Hash-Tables.html#g-hash-table-foreach.

[5] Glib hashtable. https://developer.gnome.org/glib/stable/glib-Hash-Tables.html.

[6] Glib lookup. https://developer.gnome.org/glib/stable/glib-Hash-Tables.html#g-hash-table-lookup.

[7] Glib manual. https://developer.gnome.org/glib/.

[8] Uminho. https://www.uminho.pt/PT.

[9] Yelp. https://www.yelp.pt.