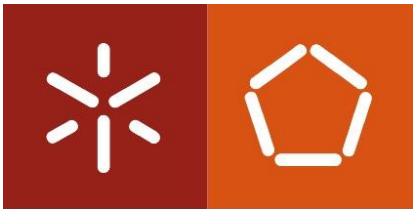

Universidade do Minho

Departamento de Informática

Laboratórios de Informática III (2º ano de MIEI)

Projeto C

Relatório de Desenvolvimento

Grupo 3

Ariana Lousada	Rui Armada	Miguel Gomes
(a87998)	(a90468)	(a93294)

19 de maio de 2021

Resumo

O presente documento expõe o projeto desenvolvido em linguagem C no âmbito da unidade curricular Laboratórios de Informática III. O principal objetivo deste projeto consiste em desenvolver uma aplicação de gestão de *reviews*, a partir da informação contida em alguns ficheiros de dados *.csv*. Neste relatório são descritas sucintamente todas as decisões tomadas pela equipa de trabalho, como a arquitetura global da aplicação e as estruturas de dados utilizadas.

Conteúdo

Capítulo 1

Introdução

Para o desenvolvimento deste projeto, foi necessário tomar algumas decisões importantes, nomeadamente a escolha das estruturas de dados a serem utilizadas. Uma vez que é necessário processar um volume considerável de dados, concluiu-se que seria de extrema importância utilizar uma estrutura de dados de rápida consulta. Visto isto, foi decidido pela equipa de trabalho utilizar tabelas de *hash*.

Apesar existir ao nosso dispor a biblioteca *glib* com tabelas de *hash* já definidas, a equipa de trabalho deparou-se com alguns problemas na interpretação da sua estrutura, o que levou à criação de estruturas de dados personalizadas. O principal objetivo consiste em atingir um compromisso de memória e de tempo de execução que permitam uma boa *performance* da aplicação em geral.

Capítulo 2

Desenvolvimento

2.1 Ficheiros CSV

Uma vez que a aplicação a desenvolver utiliza informação de vários ficheiros *CSV*, é necessário fazer *parsing* dos mesmos, guardando a informação mais relevante nas estruturas de dados desenvolvidas. Nomeadamente a aplicação irá utilizar informação de três ficheiros distintos:

- **Reviews** - cada linha é constituída pelo identificador do *user*, do *business* e da própria *review*, estrelas, cool, funny, data e texto.
- **Business** - cada linha é constituída pelo identificador, nome, cidade, estado e categorias do próprio *business*.
- **Users** - cada linha é constituída pelo identificador, nome e amigos do *user*.

Uma vez que não existe qualquer **query** que requer informação dos amigos do *user* e que os mesmos ocupam grande parte da memória total do ficheiro *CSV* dos mesmos, foi decidido não carregar esta informação para as estruturas de dados, uma vez que foi observada uma diminuição considerável no tempo de *parsing* dos ficheiros para a aplicação quando estes dados não eram armazenados. Também foi decidido apenas carregar os identificadores constituintes da review assim como as estrelas e o texto, uma vez que as restantes informações não iriam ser utilizadas pelo programa.

2.2 Estruturas de Dados

Para ser possível organizar a informação de uma forma ordeira e organizada, foi criada uma estrutura *SGR* que contém as três *hashtables* principais de **Users**, **Reviews** e **Business**.

```
typedef struct sgr{
  user_cat * user;
  business_cat * business;
  reviews_cat * reviews;
}* SGR;
```

Para o catálogo de *reviews* desenvolveram-se as seguintes **structs**,

```
typedef struct reviews_cat {
    l_review * hash_table_review[MAX_TABLE];
    size_t n_reviews;
}reviews_cat;

typedef struct l_review {
    reviews* reviews;
    struct reviews *head;
    struct reviews *last;
}l_review;

typedef struct reviews { //id, userID, businessID, stars, funny, cool, text
```

```
  char* review_id;
  char* user_id;
  char* business_id;
  float stars;
  char * text;
  struct reviews *next;
} reviews;
```

onde a *l_review* funciona como uma lista de *reviews* e *head* e *last* correspondem respetivamente à primeira e última review da tabela de *hash*. A *struct reviews* simplesmente é composta pela informação de cada *review* do ficheiro *CSV*. Esta tabela de *hash* foi organizada por ID de user e não por *ID de review*, uma vez que em algumas *querys* isto mostra-se mais vantajoso particularmente no acesso a certa informação necessária, como vai ser explicado na secção 2.4 deste documento.

Para o catálogo de *business* aplicou-se a mesma lógica que as tabelas de *hash* de *reviews*.

```
typedef struct business_cat {
    l_business* hash_table_business[MAX_TABLE];
    size_t n_business;

typedef struct l_business {
    business* business;
    struct business *head;
    struct business *last;
}l_business;

typedef struct business { //id, name, city, state, categories
        char* business_id;
        char* business_name;
        char* business_city;
        char* business_state;
        char* business_cat;
        size_t n_reviews;
        float m_stars;
        struct business *next;
} business;
```

Para cada *business* decidiu-se adicionar, para além das informações necessárias presentes no ficheiro *CSV*, o número total de *reviews* assim como o número médio de estrelas atribuídas ao *business* em causa. O número total de *reviews* é um contador que vai incrementando à medida que são adicionadas *reviews* do *business* atual, assim como o número de estrelas[1].

Foi decidido adicionar estes valores a cada *business* uma vez que melhoram o desempenho de algumas **queries**, como vai explicado em mais detalhe na secção 2.4 deste relatório.

Para o catálogo de *Users* foi utilizada uma lógica semelhante à do catálogo de *reviews*.

```
typedef struct user_cat {
    l_user* hash_table_user[MAX_TABLE];
    size_t n_users;

typedef struct l_user {
    user* user;
    struct user *head;
    struct user *last;
}l_user;

typedef struct user { //id, name, friends
    char* user_id;
    char* user_name;
    size_t n_reviews;
    struct user *next;
}user;
```

Tal como nos *business*, foi adicionado o número total de reviews feitas pelo *user* atual, uma vez que melhora o desempenho de algumas *queries*.

[1]Este número corresponde à média das estrelas atribuídas, que vai sendo calculado de acordo com inserções de *reviews*

Por fim, foi necessário desenvolver uma estrutura de dados para resposta das *queries*. Para isto foi desenvolvida a estrutura *TABLE*.

Através de uma análise dos valores de retorno de cada *query*, conseguimos observar o seguinte padrão:

- Query 2 - retorna informação a cerca de um *business*(lista de nomes e número total);
- Query 3 - retorna informação a cerca de um *business*(nome,cidade,estado,stars,e número total reviews).
- Queries 4 e 5 - retornam uma lista de *business*(nome e identificador apenas).
- Queries 6 e 8 - retornam informação a cerca de um *business*(identificador, nome e número de estrelas).
- Query 7 - retorna uma lista de identificadores de *users*.
- Query 9 - retorna uma lista de identificadores de *reviews*.

Com isto podemos dizer que a maior parte das queries retorna uma lista de *business*, com exceção da 7 e da 9, que retornam uma lista de identificadores.

Assim, organizou-se a estrutura *TABLE* da seguinte forma:

```
typedef struct TABLE2 {
    business * info;
    struct TABLE2 * next;

}TABLE2;

typedef struct TABLEIDS {
    char * id;
    struct TABLEIDS * next;
}TABLEIDS;

typedef struct {
    TABLE2 * table;
    TABLE2 * last;
    TABLEIDS *ids;
    TABLEIDS * lastID;
}*TABLE;
```

Com isto, a *TABLEIDS* foi utilizada para as *querys* 7 e 9 e a *TABLE2* para as restantes, de modo a diminuir a memória alocada por *query*.

2.2.1 Complexidade das estruturas e otimizações realizadas

Uma vez que foram desenvolvidas tabelas de *hash* específicas, foi necessário também desenvolver uma função de *hash*, para a qual se utilizou o seguinte algoritmo de *hashing*, dividindo o valor gerado no final pelo valor atribuído a *MAX_TABLE*, correspondente ao tamanho máximo de cada tabela de *hash*[2]:

```
size_t hash(const char *s){
  size_t hash, i,len = strlen(s);
  for(hash = i = 0; i < len; ++i){
    hash += s[i];
    hash += (hash << 10);
    hash ^= (hash >> 6);
  }
  hash += (hash << 3);
  hash ^= (hash >> 11);
      hash += (hash << 15);
      hash = hash%MAX_TABLE;
      return hash;
}
```

[2]Esta divisão é necessária de modo a que o valor que resulte do algoritmo esteja entre os índices disponíveis na tabela.

Esta *hash function* é utilizada maioritariamente para o armazenamento de dados e para a respetiva consulta[3].

Inicialmente, uma vez que foram utilizados *arrays* de tamanho estático para cada campo de informação contida nos ficheiros *CSV*, o programa requeria cerca de 4GB de memória para carregar toda a informação necessária. Visto isto, foi decidido utilizar *arrays* de tamanho dinâmico, de modo a tentar diminuir o consumo de memória. Estas modificações permitiram uma redução da memória utilizada de 4GB para 1.200GB, com tempo de carregamento de aproximadamente 24 segundos.

De modo a ser possível analisar a composição das várias estruturas de dados criadas, desenvolveram-se os seguintes esquemas:

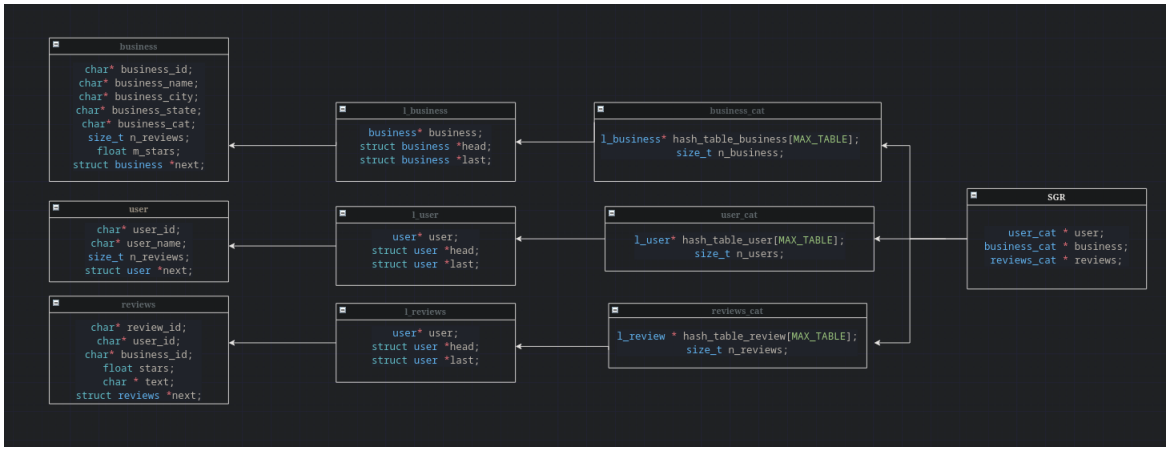

Figura 2.1: Esquema da estrutura SGR

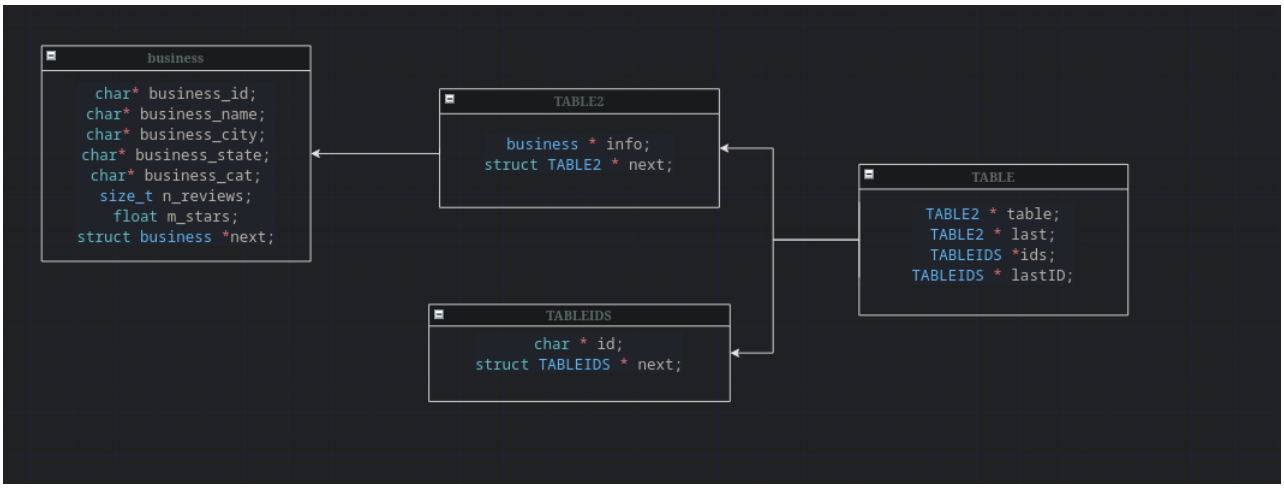

Figura 2.2: Esquema da estrutura TABLE

2.3 Modularização Funcional

O projeto é composto por vários módulos, que foram criados ao longo do desenvolvimento da aplicação, conforme as necessidades detetadas.

- controler.c - contém a função responsável pela inicialização do controler.
- getset.c - contém as funções *get* e *set* necessárias para aceder às estruturas de dados.

[3]Uma vez que é esta função que determina o *slot* onde a determinada *review/user/business* vai estar armazenada.

- hash.c - contém todas as funções necessárias para criação de *business*, *reviews* e *users*, para a inserção das mesmas e a função de *hash*, responsável por calcular os *slots* de cada entidade na tabela.
- load.c - contém todas as funções necessárias para o *parsing* dos ficheiros *CSV*.
- menu.c - contém a função responsável pela geração da *view* para interação com o utilizador.
- SGR.c - contém todas as funções responsáveis pela alocação e respetivo *free* de memória para a estrutura SGR. Também contém todas as funções das *queries*.
- aux.c - contém todas as funções auxiliares utilizadas pelas *queries*.
- structs.c - contém todas as funções de inserção e inicialização de estruturas relacionadas com a *TABLE*.
- interpreter.c - contém as funções desenvolvidas para o interpretador.
- pagination.c - contém as funções relacionadas com a paginação.

2.4 Queries e respetivas otimizações

2.4.1 Query 1

Uma vez que a primeira *query* apenas consiste no carregamento de dados, foi desenvolvida uma função para esse propósito, que carrega os dados para uma struct *SGR*. Com isto, também utiliza 3 funções, que são mais específicas para o tipo de dados que carregam para a aplicação. Como já foi referido anteriormente na secção 2.2.1, foi possível melhorar o desempenho desta *query* ao utilizar *arrays* de tamanho dinâmico. Após várias modificações e testes, verificou-se um tempo de execução de 24 segundos e uso de memória aproximado a 1.200GB no carregamento dos dados.

2.4.2 Query 2

Para a segunda *query* foi criada uma *TABLE*, na qual foram inseridos *business* à medida que foi encontrado um nome que começasse pela letra dada, utilizando a função *tolower* para que a pesquisa fosse *case insensitive*.

2.4.3 Query 3

Para a terceira *query* foi criada uma *TABLE* com um só *business*, que foi aquirido através da execução da função *hash* utilizando o seu identificador. Uma vez que foram adicionadas logo desde início o número total de *reviews*, bastou apenas aceder a esse valor e apresentá-lo ao utilizador.

2.4.4 Query 4

Para melhorar o desempenho desta *query*, foi decidido organizar o catálogo das *reviews* logo na parte de *parsing* pelos identificadores de *users* em vez de utilizar os identificadores das *reviews*. Esta organização, como seria de esperar, deu origem a um maior número de colisões; porém, estas colisões colocaram as *reviews* todas feitas por um dado utilizador no mesmo *slot*.

Com isto, para aceder a todas as *reviews* feitas por um dado utilizador, apenas basta fazer *hash* do identificador deste. Isto evita a necessidade de consultar as três tabelas de *hash* simultaneamente, sendo apenas necessário percorrer a tabela de *reviews* e a de *businesses*.

2.4.5 Query 5

Para a quinta *query*, foi adicionado o número total de *reviews* feitas a um dado negócio à informação do *business*, assim como o número total de estrelas a ele atribuídas, de modo a ser apenas necessário aceder ao valor das estrelas para obter o valor médio de estrelas.

Com isto, aplicando um raciocínio semelhante ao da *query* anterior foi possível obter os resultados pretendidos.

2.4.6 Query 6

Para a sexta query foi necessário desenvolver uma nova estrutra de modo a ser possível o armazenamento de cidades numa lista ligada.

```
typedef struct top_arr{
  business* b_city;
  struct top_arr *next;
}top_arr;

typedef struct city_arr{
  char * city;
  top_arr* top;
  top_arr* last;
  struct city_arr *next;
}city_arr;
```

À medida que se vão adicionando cidades, vai-se comparando também com o número médio de estrelas de um determinado negócio. No final apenas os n elementos de maior cotação são selecionados do *top_arr*, que é constituído por todos os elementos de uma dada cidade ordenados. Inicialmente, para organizar o *top_arr* foi utilizado o algoritmo *bubble-sort*; contudo, era pouco eficiente, uma vez que a *query* demorava cerca de um minuto a executar. De modo a melhorar o desempenho, utilizou-se um algoritmo de inserção ordenada em vez do *bubble-sort*, o que diminuiu o tempo de execução da *query* para aproximadamente 13 segundos.

2.4.7 Query 7

Para a sétima *query*, utilizou-se a parte da *TABLEIDS* para armazenar os identificadores dos *users* numa lista ligada. Uma vez que os estados de cada negócio são *strings* e que a comparação de *strings* num grande volume de dados(como o ficheiro das *reviews*) iria afetar significativamente o desempenho da procura e consulta de dados. Para isto, foi desenvolvida a função $hashingB(char*state)$ que transforma um estado na respetiva soma dos códigos $ASCII$(testando tambem se está em ordem alfabética, incrementando o valor caso isto se verifique, para evitar ambiguidades como por exemplo os estados "AB"e "BA", que iriam ser associados ao mesmo inteiro). Com a utilização desta função apenas são comparados inteiros, que é muito mais eficaz em termos de tempo de execução.

2.4.8 Query 8

Para a oitava *query* utilizou-se o mesmo raciocínio desenvolvido na *query* 6, apenas substituindo a lista ligada de cidades por uma lista ligada de categorias.

2.4.9 Query 9

Para a última *query*, tal como para a *query* 7, utilizou-se a *TABLEIDS* da *TABLE*. De modo a tentar diminuir o tempo de execução, desenvolveu-se um algoritmo de procura de uma palavra numa dada *string*[4], de modo a evitar o uso de funções como a *strcmp*, que iriam aumentar significativamente o tempo de execução, uma vez que é necessário percorrer todas as *reviews*. À medida que são encontradas *reviews* com a palavra dada no campo texto, os seus identificadores vão sendo adicionados à *TABLE*. No final obtêve-se um tempo de execução de aproximadamente 4 segundos, utilizando cerca de 0.6GB de memória.

2.5 Paginação

Para a paginação utilizou-se um array de pointers de *TABLEIDS* ou de *TABLE2*, dependendo da *query* em questão. Percorrendo a estrutura obtida pelas *queries*(*TABLE*), vão sendo guardados *pointers* de 10 em 10 slots da tabela(5 em 5, caso a *query* em questão retorne informação acerca de negócios).

Com isto, utilizando a opção "p"(*previous*) e "n"(*next*) conseguimos recuar ou avançar a página respetivamente. Para além disto, também é possível inserir o número pretendido e caso este exista, é consultado o pointer respetivo,

[4] A função $findWordR(char*searchW, char*text)$

apresentando 10 (ou 5) linhas do resultado da página correspondente. Caso esta não exista, é apresentada a última página. Também é possível sair da paginação inserindo "q" (*quit*) ou "n" (se o utilizador se encontrar na última página).

2.6 Arquitetura Final da Aplicação

Para uma melhor organização em termos de arquitetura do projeto, este foi organizado em *Model*, *Control* e *View*, dos quais o *Model* é o responsável por todas as operações de manipulação e armazenamento de dados, o *Control* funciona como uma ponte de ligação entre a *View* e o *Controler* e a *View* como uma interface para o utilizador.

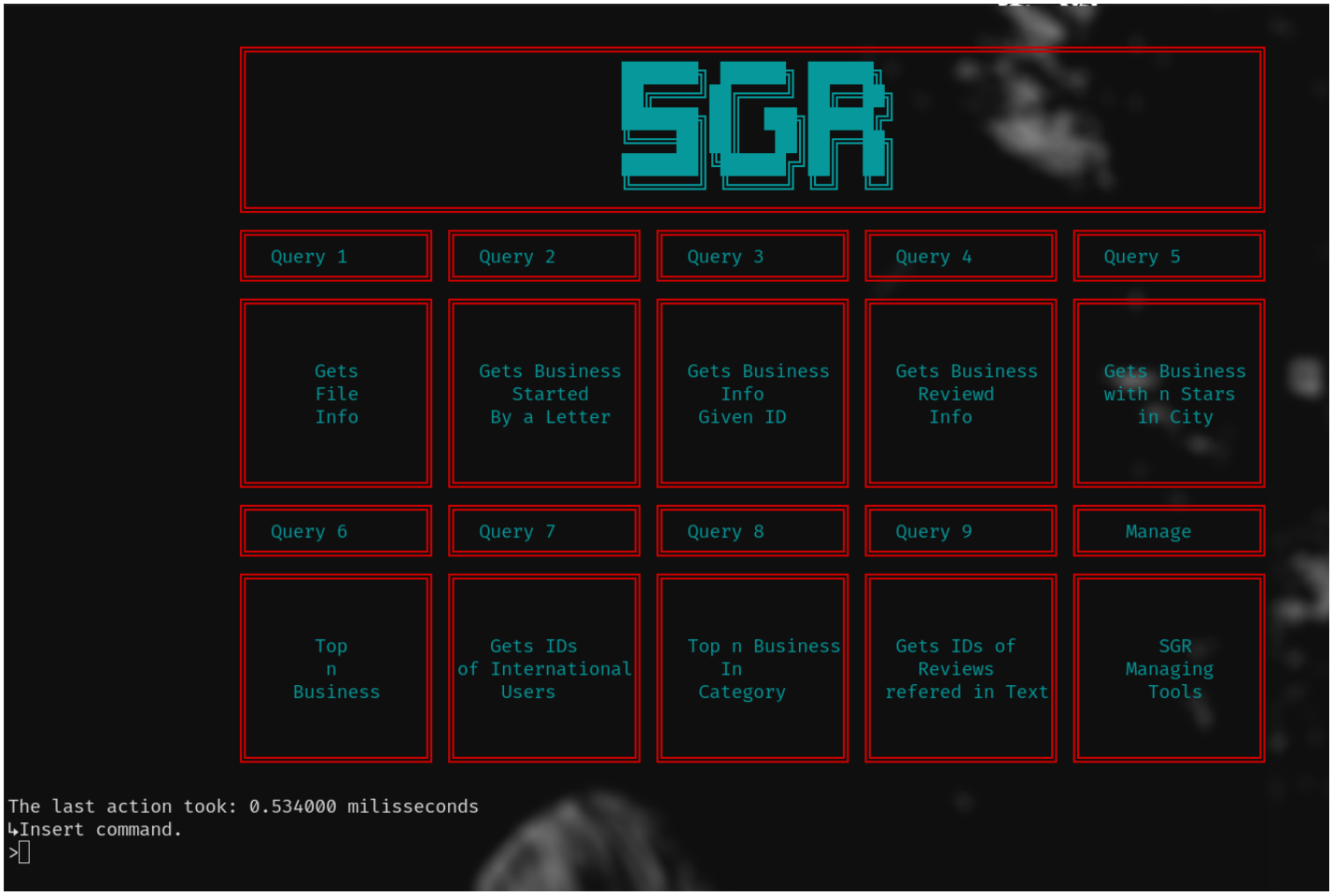

Figura 2.3: Interface

Capítulo 3

Análise de resultados e sugestões para melhoria do projeto

Com a realização de vários testes fomos capazes de detetar alguns problemas, nomeadamente na *query* 7 e na função *fromCSV* do interpretador.

Na query 7 observou-se uma repetição de dados de utilizadores, assim como alguns *users* que apenas visitaram um estado. Depois de uma análise exaustiva com recurso a testes chegámos à conclusão que isto deve-se às colisões presentes na tabela de *hash* das *reviews*.

Uma vez que foram utilizados os identificadores de *users* para organizar esta tabela, ficámos com as *reviews* de um dado *user* no mesmo *slot*. Porém, existem outras colisões para além das provocadas por este tipo de organização que resultam da nossa função de *hash*. Os utilizadores que foram colocados num dado *slot* por esta razão acabam por influenciar a pesquisa, que foi feita com a ideia que só existiria um identificador de *user* por *slot*.

Uma sugestão para corrigir esta *query* seria voltar a carregar os dados das *reviews*, organizando a tabela pelos identificadores de *reviews* (apesar de isto piorar significativamente o sua desempenho).

Ao desenvolver a função $fromCSV$ também foram detetados alguns problemas na leitura de dados dos ficheiros *CSV* para uma estrutura *TABLE*. Apesar de serem utilizadas todas as funções necessárias para a leitura de ficheiro, observou-se que a função apenas estava a ler a última linha.

Ao verificar as *memory leaks* com o *Valgrind*, detetámos alguns *bytes* de possível perda que não conseguimos detetar. Talvez uma resolução possível para este problema seria reescrever certas partes do código, principalmente em zonas de alocação de memória.

Por fim, para melhorar de uma maneira geral as colisões que ocorrem nas tabelas de *hash*, talvez o ideal seria utilizar um algoritmo de hash diferente, de modo a diminuir o número total de colisões em cada tabela.