

UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

Laboratórios de Informática III (2^o ano de MIEI)

Projeto Java

Relatório de Desenvolvimento

Grupo 3

Ariana Lousada
(a87998)

Rui Armada
(a90468)

Miguel Gomes
(a93294)

25 de junho de 2021

Resumo

O presente documento expõe o projeto desenvolvido em linguagem *Java* 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

1	Introdução	2
2	Desenvolvimento	3
2.1	Ficheiros CSV	3
2.2	Estruturas de Dados	3
2.3	Arquitetura do projeto	3
2.3.1	Diagrama de Classes	4
2.4	Complexidade das estruturas e otimizações realizadas	4
2.5	Modularização Funcional	5
2.5.1	Parser	5
2.5.2	Estatísticas	5
2.5.3	Classe de Testes	6
2.5.4	Armazenamento do Estado da Aplicação	7
2.6	Consultas interativas	7
3	Testes realizados e Resultados	10
4	Análise de resultados e problemas de implemetação	15
5	Conclusões finais	16

Capítulo 1

Introdução

O principal objetivo deste projeto consiste em consolidar os conhecimentos teóricos e práticos adquiridos na disciplina de Programação Orientada aos Objetos, construindo um programa de gestão de *Reviews*.

Utilizando várias bibliotecas disponíveis do *Java* e as várias estruturas auxiliares de dados desenvolvidas pela equipa de trabalho, foi possível atingir os principais objetivos estabelecidos no início da construção do projeto.

Neste documento irão ser explicadas todas as estruturas de dados e as dependências entre elas, assim como as decisões tomadas pela equipa de trabalho.

Capítulo 2

Desenvolvimento

2.1 Ficheiros CSV

Tal como o programa já desenvolvido na parte de C do projeto, a aplicação a desenvolver utiliza informação de vários ficheiros *CSV*, dos quais é necessário extrair informação. 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*.

2.2 Estruturas de Dados

O programa desenvolvido tem como principais classes **User**, **Review** e **Business**.

Um **User** é composto pelo seu nome, *set* de amigos e número total de *reviews* efetuadas.

Um **Business** é composto pelo nome, cidade, estado, *set* de categorias, número total de *reviews* associadas assim como o número total de estrelas a ele atribuídas.

Uma **Review** é composta pelo identificador do **User**, do **Business**, número de estrelas atribuído, *cool*, *funny* e *useful* (valores inteiros), data e texto da *Review* em si. Para juntar todos os *users*, *businesses* e *reviews* criaram-se um catálogo para cada: **User_cat**, **Business_cat** e **Review_cat** respetivamente.

Cada um destes catálogos contém um *Map*, no qual cada *Entry* é constituída pelo identificador e o respetivo *user*, *business* ou *review*.

2.3 Arquitetura do projeto

De modo a evidenciar facilmente a estrutura completa do programa criou-se um diagrama de classes com auxílio ao *IntelliJ*.

2.3.1 Diagrama de Classes



Figura 2.1: Diagrama de Classes do projeto

2.4 Complexidade das estruturas e otimizações realizadas

Tal como já foi referido na secção 2.2, para a informação contida nos ficheiros CSV foram construídas as classes `User`, `Business`, `Review`, `User_cat`, `Business_cat` e `Review_cat`.

Para além destas classes para dados, foi novamente construída a classe `Table`, de modo a armazenar os resultados de cada consulta interativa.

Depois de uma análise do que cada consulta interativa retornava como resultado, a equipa de trabalho viu como necessário criar as várias variáveis contidas na classe `Table`. Cada consulta irá apenas utilizar as variáveis necessárias para armazenar os seus resultados.

```
1 public class Table implements Serializable{
2     private static final long serialVersionUID = 1656334331841663697L;
3
4     private int queryN; //todas
5     private Set<String> codNames; //query 1
6     private int counter1; // query 1 e 2
7     private int counter2; // query 2
8     private Map<Integer, Triple<Integer,Integer,Float>> info; // query 3 e 4
9     private Map<Integer, HashMap<String,Integer>> info2; // query 6
10    private Map<String, Map<String,Integer>> info3; // query 7
11    private Map<String, Integer> info4; //query 8 e 5
12    private Map<String, Float> info5; //query 9
13    private Map<String, TableStateBus> info6; //query10
```

Como podemos observar, para a consulta dez é utilizada a classe `TableStateBus`.

A classe `TableStateBus` contém um *Map* no qual cada *Entry* é constituída por um nome de um estado e uma *TableBusClass*, que é composta por um *Map* em que cada *Entry* é constituída por um nome de um negócio e respetiva

classificação.

```
1 public class TableStateBus {  
2     private Map<String, TableBusClass> stateBus;  
  
1 public class TableBusClass {  
2     private Map<String, Float> busClass;
```

2.5 Modularização Funcional

Para a construção da aplicação utilizou-se a estrutura *MVC - Model View Controller*.

O *Model* é responsável por todas as operações de *backend*, como o cálculo de resultado das consultas interativas, testes de performance e valores estatísticos.

A *View* é responsável pela apresentação de informações ao utilizador. Todos os menus e resultados das consultas da aplicação são imprimidos no ecrã a partir deste módulo. O *Controller* estabelece uma ligação entre a *View* e o *Model*. É neste módulo que é captado e analisado o *input* do utilizador, que posteriormente é enviado para o *Model*.

Uma vez que nas secções 2.2 e 2.4 já foram expostas as estruturas responsáveis pelos dados da aplicação assim como dos resultados, vão ser expostos os restantes módulos da aplicação.

2.5.1 Parser

De modo a ser possível extrair toda a informação necessária dos ficheiros *csv*, desenvolveu-se uma classe **Parser**.

Durante a leitura dos ficheiros é também feita simultaneamente a verificação de dados. Tal como o pedido, considerou-se que:

- Um negócio só é válido se a linha do ficheiro em questão possuir cinco campos.
- Um utilizador é válido se possuir três campos.
- Uma *review* é válida se possuir 9 campos.

Nas *reviews* é também testado se os identificadores do utilizador e do negócio em causa existem. Caso não existirem a *review* é considerada como inválida.

Este método de teste para o catálogo de *reviews* funciona, uma vez que todos os negócios e utilizadores são importados para a aplicação anteriormente, o que evita situações como a eliminação de uma *review* antes de o utilizador ou negócio associado ser lido e armazenado.

2.5.2 Estatísticas

Tal como solicitado no enunciado do projeto, desenvolveu-se uma classe **Stats** que reúne vários dados estatísticos, como o número de reviews inválidas, número total de *reviews*, de *business* e de *users*, número de negócios avaliados, número de negócios não avaliados, número de *users* que fizeram pelo menos uma *review*, número de *users* sem *reviews*, número de *reviews* nulas (com os campos *useful*, *funny* e *cool* a zero), *reviews* e classificação média por mês, média global de todas as *reviews* e *users* por mês (ativos).

Todos os métodos de aquisição de dados estatísticos estão definidos no *Model*, que são posteriormente utilizados consoante o utilizador seleciona ou não a opção "3 - Stats" do menu inicial.

```
1 public class Stats{  
2     private Triple<String, String, String> files;  
3     private int wrong_reviews;  
4     private int number_reviews;  
5     private int number_businesses;  
6     private int number_users;  
7     private int number_reviewed_businesses;  
8     private int number_non_reviewed_businesses;
```

```

9 private int number_users_with_reviews;
10 private int number_users_with_no_reviews;
11 private int null_reviews;
12 private Map<Integer,Integer> reviews_per_month;
13 private Map<Integer,Float> average_per_month;
14 private double global_average;
15 private Map<Integer,Integer> users_per_month;

```

2.5.3 Classe de Testes

Tal como foi solicitado no enunciado, foi implementada uma classe de testes. Esta classe tem como objetivo testar diferentes funcionalidades e implementações na aplicação desenvolvida. Com isto, foram implementados dois tipos de testes: **Testes das Queries** e **Testes Unitários**.

Teste das Queries

Aqui foram realizados vários testes às diferentes *queries* presentes na aplicação. Foi criada a classe *Query_Testing* que possui apenas o método *Testing*. Neste método são invocadas as *queries* uma a uma, testando a sua integridade recorrendo a vários tipos de argumentos.

Quando a integridade da *Query* é testada, é também verificado o seu tempo de execução. O seguinte excerto de código exemplifica os testes realizados à *Query 2*.

```

1  /*
2      Query 2 – Valid Test
3  */
4  Crono.start();
5  Table query2_1 = m.query2(2000, 07);
6  Crono.stop();
7  System.out.println("Query 2 -> " + Crono.get_time_to_string());
8  if(t.isEmpty(query2_1)) {
9      System.out.println("Query 2: Recieved Empty Table");
10     System.out.println("Therefore, the test was invalid.");
11 } else {
12     System.out.println("Query 2: " + query2_1);
13 }
14
15 /*
16     Query 2 – Invalid Test
17 */
18 Crono.start();
19 Table query2_2 = m.query2(1998, 07);
20 Crono.stop();
21 System.out.println("Query 2 -> " + Crono.get_time_to_string());
22 if(t.isEmpty(query2_2)) {
23     System.out.println("Query 2: Recieved Empty Table");
24     System.out.println("Therefore, the test was invalid.");
25 } else {
26     System.out.println("Query 2: " + query2_2);
27 }

```

Testes Unitários

De seguida foram implementadas três classes distintas cujo o objetivo é testar pequenas porções de código para verificar a integridade dos três diferentes catálogos existentes no projeto. As classes são as seguintes:

- User_Test
- Business_Test
- Review_Test

Estas três classes possuem métodos que consistem em testar se um dado *Id* ou *nome* existe no catálogo. O seguinte excerto de código exemplifica os testes unitários realizados ao catálogo de utilizadores.

```
1 public class User_Test {
2
3     public boolean containsUser(User_cat u, String id) throws NonExisting_User_Exception {
4         return u.getU_catalog().keySet().contains(id);
5     }
6
7     public boolean existingName(User_cat u, String name) {
8         boolean rez = false;
9         for(Entry<String, User> entry : u.getU_catalog().entrySet()) {
10             if(entry.getValue().getUser_name() == name) {
11                 rez = true;
12             }
13         }
14         return rez;
15     }
16 }
17 }
```

2.5.4 Armazenamento do Estado da Aplicação

Tal como foi solicitado, foi implementada a salvaguarda do estado do programa durante a sua execução. Em qualquer momento durante a utilização da aplicação o utilizador poderá salvar o estado do programa até ao momento. Para isto, foram implementadas as funções *read* e *save*.

A função *save* recebe o nome que o utilizador deseja dar ao ficheiro de *backup* do programa e vai escrevendo toda a informação contida no programa até o utilizador pedir para salvarguardar o estado da aplicação.

A função *read* recebe o nome do ficheiro do qual será extraída toda a informação que compõe o *backup* da aplicação, importando do ficheiro objeto toda a informação. Os dados presentes da aplicação são então alterados de forma a corresponder à informação contida no ficheiro objeto pretendido, permitindo assim o carregamento de um *backup* realizado anteriormente pelo utilizador.

2.6 Consultas interativas

De modo a agrupar a informação da aplicação de acordo com os vários parâmetros propostos, foram desenvolvidas várias consultas interativas.

Consulta Interativa - 1

De modo a obter uma lista ordenada alfabeticamente com os identificadores dos negócios nunca avaliados utilizou-se um *TreeSet*, uma vez que cada elemento é automaticamente inserido por ordem alfabética.

Uma vez que cada negócio tem o número total de *reviews* associado, basta testar se este valor corresponde a zero e adicionar o identificador do negócio em causa ao *TreeSet*, caso isto se verifique.

Consulta Interativa - 2

Para calcular o número total de *reviews* e de utilizadores distintos num dado mês de um determinado ano, utilizaram-se dois contadores e uma lista. Para cada *review*, é em primeiro lugar verificada se a data está incluída no mês e no ano dados pelo utilizador. Se isto se verificar, o primeiro contador é incrementado e é verificado se o utilizador da *review* atual está na lista de identificadores de utilizadores. Caso não esteja, o identificador do utilizador é adicionado à lista e o segundo contador é incrementado.

Consulta Interativa - 3

Para esta consulta é necessário obter, dado um identificador de um utilizador, quantas *reviews* fez, quantos negócios distintos avaliou e que nota média atribuiu em cada mês. Para isto, utilizou-se uma matriz, dois *arrays* e um *Map*.

A matriz tem dimensões doze por dois, na qual cada linha corresponde a um mês e as colunas ao número total de *reviews* e de negócios(não distintos) respetivamente.

O primeiro *array* de dimensão doze armazena todas as estrelas que o utilizador atribui num determinado mês, enquanto que o segundo *array* também de dimensão doze que contém o número de negócios distintos avaliados por mês.

Para cada *review*, vão sendo incrementados os valores contidos na matriz e nos *arrays* se a *review* atual foi feita pelo utilizador dado.

Os negócios distintos são verificados através da sua inserção numa lista sem repetidos, incrementando também o valor do mês respetivo no segundo *array*.

No final vão ser inseridas doze entries no *map info*, cada uma constituída pelo número do mês e triplo associado, que contém o número total de *reviews*, número de negócios distintos avaliados e a classificação média atribuída.

Consulta Interativa - 4

Para esta consulta utilizou-se um raciocínio semelhante à consulta anterior. A única diferença consiste na procura de um dado negócio dado pelo utilizador e os utilizadores distintos que o avaliaram.

Consulta Interativa - 5

Para obter a lista de negócios que um dado utilizador avaliou, em primeiro lugar foi necessário percorrer o catálogo das *reviews*, armazenando os identificadores dos negócios pretendidos num *TreeSet*.

Depois disto, foi necessário aceder ao catálogo dos utilizadores, armazenando os nomes na *Table*, mais propriamente no *map info4*. Consultando quantas vezes um dado identificador aparece numa lista¹, este valor é adicionado a um *map* de um objeto *Table* com o nome de negócio a ele associado.

Por fim, criou-se um método de ordenação do *map info4* por valores(que neste caso correspondem aos contadores de quantas vezes cada negócio foi avaliado).

Consulta Interativa - 6

Para obter os resultados pretendidos, começou-se por percorrer o catálogo de *reviews*, de modo a armazenar os identificadores de utilizadores de cada negócio por ano.

De seguida, é aplicada uma inversão da ordem do *TreeSet*, inserindo um número X de elementos (inserido pelo utilizador) num objeto do tipo *Table*.

Consulta Interativa - 7

De modo a obter a lista dos três negócios mais famosos de cada cidade, criou-se um *Map* cujas *keys* correspondiam às diferentes cidades e *values* a objetos da classe *DataSet*. Nesta consulta, é apenas utilizado o *dataSet*, que corresponde a um *set* de pares. Cada par contém o nome do *business* e o número de *reviews* a ele associado.

Consulta Interativa - 8

Para obter a lista dos X utilizadores que avaliam mais negócios diferentes é necessário percorrer todo o catálogo de *reviews*. De modo a agrupar a informação, utilizou-se um *Map* com uma *String* como *key* (que corresponde ao identificador do utilizador) e um objeto da classe *DataSet* como *value*. Desta classe *DataSet* apenas se vai utilizar a variável *pair8*, que corresponde a um par de um inteiro e um objeto da classe *NamesBus*. Por sua vez esta classe contém um *set* de strings.

Assim, ao longo da análise do catálogo de *reviews* vai se utilizando esta estrutura, na qual o par de inteiro e *NamesBus*(*set* de strings) vai corresponder ao número de negócios distintos avaliados e à lista dos identificadores desses negócios(sem repetidos) respetivamente. Este par é associado ao utilizador respetivo.

¹Esta lista contém todos os identificadores de negócios com repetições

Ao mesmo tempo que se utiliza esta estrutura também se vai utilizar a variável `info4` (um Map de strings e inteiros), fazendo inserções ou substituições na informação relacionada com cada utilizador(número de negócios distintos avaliados).

Depois da agregação de todos os dados, ordena-se a variável `info4` por ordem decrescente dos *values*. Por fim a informação presente na `info4` é inserida pela mesma ordem num objeto da classe *Table*.

Consulta Interativa - 9

Para obter a informação pretendida, é utilizada uma lista de `Triple`(triplos). Cada `Triple` contém o identificador de um utilizador, o número de vezes que este avaliou o negócio em questão e a classificação média que lhe atribui como primeiro, segundo e terceiro elementos, respetivamente.

Quando o catálogo de *reviews* é totalmente percorrido, esta lista de triplos é organizada por ordem decrescente do segundo elemento, isto é, pelo número de avaliações. De seguida, são inseridos como pares (key,value) num novo Map os primeiros e terceiros elementos dos X triplos iniciais. Por fim, este map é transferido para um objeto da classe *Table*.

Consulta Interativa - 10

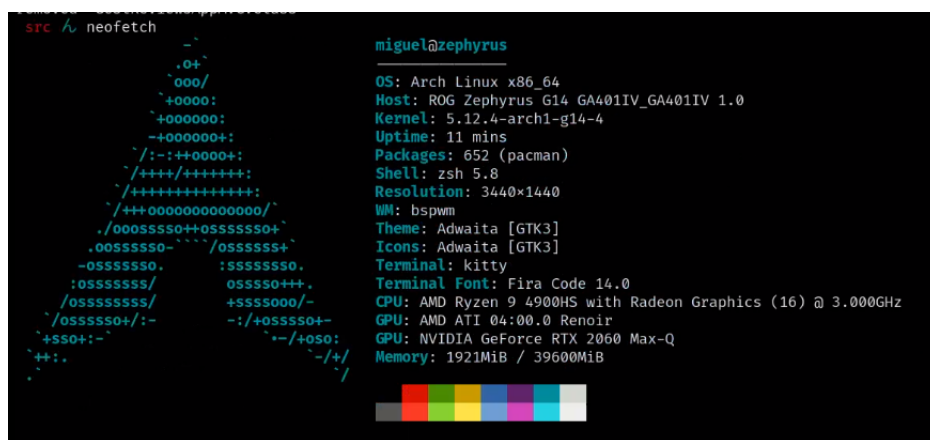
Para a última consulta foi necessário utilizar uma estrutura de dados mais complexa. De modo a agregar a média de classificação de cada negócio em cada cidade por estado, utilizou-se um Map com o nome do estado em questão como *key* e um objeto da classe `TableStateBus` como *value*. Este objeto contém um outro Map com o nome da cidade como *key* e um objeto da classe `TableBusClass` como *value*. Por sua vez este objeto contém um Map com o nome do negócio e classificação média a ele associada(esta classificação média é calculada no `Parser`, logo nesta consulta apenas é necessário aceder ao valor respetivo). De modo a reduzir o tempo de execução, esta informação vai sendo inserida num objeto da classe *Table* à medida que o catálogo dos negócios é percorrido.

Capítulo 3

Testes realizados e Resultados

Para a realização de vários testes de *performance* e integridade da aplicação desenvolvida, utilizou-se a aplicação desenvolvida na secção 2.5.3.

Para estes utilizou-se uma máquina com o sistema operativo *Arch Linux* e com as seguintes especificações.



```
src ~ neofetch

      .+~
     .0+
    .000/
   +0000:
  +000000:
 -+000000+:
  /:-:++0000+:
 /++++/+++++:
 /++++/+++++:
 /+++000000000000/`
./000SSSS0++0SSSSSS0+
.0SSSSSS0-`"/0SSSSSS+`
-0SSSSSS0.   :SSSSSS0.
:0SSSSSSS/   0SSSS0+++
/0SSSSSSSS/  +SSSS000/-
`/0SSSSS0+/-  -:/+0SSSS0+-
+SS0+:-      *-/+0S0:
++:.         ~-/+/-
.            -/

migueld@zephyrus
OS: Arch Linux x86_64
Host: ROG Zephyrus G14 GA401IV_GA401IV 1.0
Kernel: 5.12.4-arch1-g14-4
Uptime: 11 mins
Packages: 652 (pacman)
Shell: zsh 5.8
Resolution: 3440x1440
WM: bspwm
Theme: Adwaita [GTK3]
Icons: Adwaita [GTK3]
Terminal: kitty
Terminal Font: Fira Code 14.0
CPU: AMD Ryzen 9 4900HS with Radeon Graphics (16) @ 3.000GHz
GPU: AMD ATI 04:00.0 Renoir
GPU: NVIDIA GeForce RTX 2060 Max-Q
Memory: 1921MiB / 39600MiB
```

Figura 3.1: Especificações da máquina usada para testes

Parsing dos ficheiros csv

```
Do you intend to load the User FRIENDS ? (High Ram Usage)
Y - Yes | N - No
N
Do you intend to load the DEFAULT files ?
Y - Yes | N - No
Y
Files are being loaded.
Users LOADED
Businesses LOADED
Reviews LOADED
Elapsed Time: 35.662746322 s

##### Initialize #####
#####

1 - Load Database
2 - Interactive Queries
3 - Stats
4 - Benchmarks
5 - Load Status
0 - Exit

#####
```

Figura 3.2: Parsing dos ficheiros csv sem *friends* - RAM utilizada: aproximadamente 2.5GB

```
Do you intend to load the User FRIENDS ? (High Ram Usage)
Y - Yes | N - No
y
Do you intend to load the DEFAULT files ?
Y - Yes | N - No
y
Files are being loaded.
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.base/java.util.HashSet.<init>(HashSet.java:107)
    at Model.User.setFriends(User.java:145)
    at Model.User.<init>(User.java:29)
    at Model.Model_SGR.addUser(Model_SGR.java:158)
    at Controler.Parser.parse_users(Parser.java:227)
    at Controler.Parser.parse_full(Parser.java:333)
    at Controler.Controller.start(Controller.java:297)
    at GestReviewsAppMVC.main(GestReviewsAppMVC.java:7)
src  ↪ java GestReviewsAppMVC -Xmx30g -Xms521m
```

Figura 3.3: Parsing dos ficheiros com *friends* e opções de definição de RAM utilizadas

Testes das Queries

Na query 1 apenas são testados os tempos de execução e de *memory usage*.

```
Users LOADED
Businesses LOADED
Reviews LOADED
Parsing Time → Elapsed Time: 25.650667444 s
Used Memory: 2053Mb

Query 1 → Elapsed Time: 0.060091938 s
Used Memory: 8Mb
Query 1: Model.Table@153f5a29
```

Figura 3.4: Testes da *query* 1 e tempo de *parsing* dos ficheiros sem *friends*

Na query 2 são também verificados os tempos de execução e de *memory usage*, validando também o input. No primeiro teste são fornecidos um ano e mês válidos. O segundo teste consiste na execução da query com um ano e um mês que se encontrem fora do intervalo que consideramos válido.

```
Query 2 → Elapsed Time: 0.171006834 s
Used Memory: 0Mb
Query 2: Model.Table@7f560810
Query 2 → Elapsed Time: 2.201E-5 s
Used Memory: 0Mb
Query 2: Model.Table@69d9c55
```

Figura 3.5: Testes da *query* 2

Na query 3 são também verificados os tempos de execução e de *memory usage*, validando também o input. No primeiro teste é fornecido um User_ID (**37Hc8hr3cw0iHLoPzLK6Ow**) válido. No segundo teste é fornecido um user_id que não se encontre nos ficheiros da database e é utilizado o user_id inválido **1u**.

```
Query 3 → Elapsed Time: 0.172665125 s
Used Memory: 0Mb
Query 3: Model.Table@13a57a3b
Query 3 → Elapsed Time: 0.154919657 s
Used Memory: 0Mb
Query 3: Model.Table@7ca48474
```

Figura 3.6: Testes da *query* 3

Na query 4 são também verificados os tempos de execução e de *memory usage*, validando também o input. No primeiro teste é fornecido um Business_ID (**6iYb2HFDywm3zjuRg0shjw**) válido. No segundo teste é fornecido um business_id que não se encontre nos ficheiros da database e é utilizado o business_id inválido **1b**.

```
Query 4 → Elapsed Time: 0.162429602 s
Used Memory: 3Mb
Query 4: Model.Table@337d0578
Query 4 → Elapsed Time: 0.152150188 s
Used Memory: 0Mb
Query 4: Model.Table@59e84876
```

Figura 3.7: Testes da *query* 4

Na query 5 são também verificados os tempos de execução e de *memory usage*, validando também o input. No primeiro teste é fornecido um User_ID (**37Hc8hr3cw0iHLoPzLK6Ow**) válido. No segundo teste é fornecido um user_id que não se encontre nos ficheiros da database e é utilizado o user_id inválido **1u**.

```

Query 5 → Elapsed Time: 0.164397241 s
Used Memory: 0Mb
Query 5: Model.Table@5f375618
Query 5 → Elapsed Time: 0.143983679 s
Used Memory: 0Mb
Query 5: Recieved Empty Table
Therefore, the test was invalid.

```

Figura 3.8: Testes da *query* 5

Na query 6 são também verificados os tempos de execução e de *memory usage*, validando também o input. No primeiro teste é fornecido um inteiro positivo. O segundo teste consiste na execução da query com um inteiro negativo(inválido).

```

Query 6 → Elapsed Time: 0.96853464 s
Used Memory: 164Mb
Query 6: Model.Table@5dfcfece
Query 6 → Elapsed Time: 0.89837207 s
Used Memory: 200Mb
Query 6: Model.Table@23ceabc1

```

Figura 3.9: Testes da *query* 6

Na query 7 apenas são testados os tempos de execução e de *memory usage*.

```

Query 7 → Elapsed Time: 0.251106651 s
Used Memory: 19Mb
Query 7: Model.Table@58651fd0

```

Figura 3.10: Testes da *query* 7

Na query 8 são também verificados os tempos de execução e de *memory usage*, validando também o input. No primeiro teste é fornecido um inteiro positivo. O segundo teste consiste na execução da query com um inteiro que se encontre fora do intervalo que consideramos válido, neste caso um inteiro negativo.

```

Query 8 → Elapsed Time: 3.091709777 s
Used Memory: 144Mb
Query 8: Model.Table@7dc7cbad
Query 8 → Elapsed Time: 3.092141779 s
Used Memory: 175Mb
Query 8: Model.Table@d2cc05a

```

Figura 3.11: Testes da *query* 8

Na query 4 são também verificados os tempos de execução e de *memory usage*, validando também o input. No primeiro teste é fornecido um Business_ID (**6iYb2HFDywm3zjuRg0shjw**) válido. No segundo teste é fornecido um business_id que não se encontre nos ficheiros da database e é utilizado o business_id inválido **1b**. O terceiro teste consiste na execução da query com um inteiro que se encontre fora do intervalo que consideramos válido, neste caso um inteiro negativo. Por fim, o quarto teste recebe dois inputs inválidos.

```
Query 9 → Elapsed Time: 0.161781834 s
Used Memory: 0Mb
Query 9: Model.Table@2a2d45ba
Query 9 → Elapsed Time: 0.181205851 s
Used Memory: 0Mb
Query 9: Model.Table@2a5ca609
Query 9 → Elapsed Time: 0.186899043 s
Used Memory: 0Mb
Query 9: Model.Table@20e2cbe0
Query 9 → Elapsed Time: 0.17696397 s
Used Memory: 0Mb
Query 9: Model.Table@68be2bc2
```

Figura 3.12: Testes da *query* 9

Na query 10 apenas são testados os tempos de execução e de *memory usage*.

```
Query 10 → Elapsed Time: 109.522245856 s
Used Memory: 12Mb
Query 10: Model.Table@482f8f11
Elapsed Time: 109.523016951 s
```

Figura 3.13: Testes da *query* 10 e tempo de execução da globalidade dos testes

Algumas *queries* tendem a dar *output* de memória usada negativa. Isto deve-se à execução dos testes juntamente com os restantes processos do sistema operativo. Com isto em consideração, é possível que no final e/ou durante a execução das *queries* sejam feitas limpezas à memória, o que acaba por influenciar o valor da memória utilizada.

Capítulo 4

Análise de resultados e problemas de implemetação

Como podemos observar nos testes anteriormente expostos podem-se observar que algumas *queries* não apresentam os resultados esperados.

Isto, infelizmente, acontece em todas as queries nas quais foi necessário fazer algum tipo de ordenação da lista/-map/set dos resultados, uma vez que a equipa de trabalho viu-se com bastante dificuldade em aplicar este tipo de ordenação. Apesar de se ter tentado aplicar ordenação com *Streams*, *Collections* e *Comparators*, não foi possível atingir o objetivo pretendido.

Para além disto, na secção de armazenar e importar um estado da aplicação por ficheiro objeto, a equipa de trabalho deparou-se com algumas dificuldades sobretudo na importação de estado. É possível que seja algum erro relacionado com o **Serializable** mas infelizmente o grupo de trabalho não conseguiu corrigir. Contudo, é possível armazenar o estado da aplicação.

Também não foi possível fazer *parse* dos ficheiros com os *friends* dos utilizadores, uma vez que chega-se ao limite de memória, como se pode observar na figura 3.3. É possível que isto se deva à utilização de **HashSets** para guardar os identificadores dos *friends* de cada utilizador.

Capítulo 5

Conclusões finais

Uma vez que foi possível desenvolver o mesmo projeto (com algumas diferenças) em duas linguagens de programação diferentes, é possível fazer uma comparação geral entre as linguagens em questão, C e Java.

No desenvolvimento da aplicação em C, uma vez que a equipa de trabalho desenvolveu as suas próprias estruturas de dados (mais especificamente *hashtables*), estas observaram-se mais facilmente customizadas, visto que era possível resolver os vários problemas de uma forma escolhida pelo grupo de trabalho, como a resolução das colisões na própria *hashtable*. Isto mostrou-se benéfico para a eficiência da aplicação, uma vez que foi possível utilizar as colisões para obter dados mais rapidamente. Contudo, é necessário ter em conta algumas precauções ao utilizar C de modo a ser possível resolver *memory leaks*, que podem facilmente ocorrer se o programador não tiver os devidos cuidados.

No desenvolvimento da aplicação em Java, viu-se exatamente o oposto. Muito dificilmente o programador se depara com *memory leaks* e é uma linguagem de certa forma mais intuitiva que C. Contudo, o acesso à informação por si só é mais lento, uma vez que Java já possui uma maneira pré-definida de resolver as colisões em *hashtables* (Maps, no caso de Java). Isto nota-se sobretudo nas queries que requerem mais informação, uma vez que possuem um maior tempo de execução¹.

Com isto conclui-se que ambas as linguagens de programação possuem as suas respetivas vantagens e desvantagens e que devem ser utilizadas consoante o tipo de projeto em questão.

¹Uma vez que Java normalmente elimina as *keys* repetidas, não foi possível tirar proveito das colisões da maneira como se fez ao utilizar a linguagem C.