PROJETO C – LI3

2º Ano MIEI

2020/2021

Grupo 52

Bruno Filipe Miranda Pereira A93298

Francisco José Martinho Toldy A93226

João Pedro Fontes Delgado A93240

1. Introdução e desafios

O enunciado do projeto apresentava a tarefa de criar um sistema de gestão e consulta de recomendações de negócios na plataforma *Yelp*.

As exigências acrescidas deste trabalho face às cadeiras de C anteriores, nomeadamente a necessidade de garantir encapsulamento e modularidade, apresentaram alguns desafios no desenvolvimento. Obrigaram-nos a repensar alguns raciocínios que teríamos feito anteriormente para responder às questões colocadas neste projeto e a constituição de cada módulo.

Além da modularidade e encapsulamento, a biblioteca ***Glib*** também apresentou os seus desafios. Apesar de imensamente útil no desenvolvimento do projeto, apresentou uma curva de aprendizagem algo íngreme nas semanas iniciais, tempo que provavelmente foi recuperado ao não termos que desenvolver por nós próprios as versões de funções da biblioteca.

De um ponto de vista mais de testes ao programa, apostamos em testar o programa com os ficheiros dos professores inalterados (ou seja, evitamos a estratégia de testar com poucas linhas dos ficheiros *.csv*) . Isto levou a problemas de crashes num ou dois dos computadores de membros do grupo em fases em que o código não aplicava o número adequado de free's para reduzir ao mínimo a carga na memória. Nessa fase os testes foram feitos na máquina mais capaz do grupo.

2. Raciocínio

Desde o início do desenvolvimento do projeto que o grupo adotou a seguinte ideia para o funcionamento do programa: todo o "trabalho pesado" seria feito no início da execução do programa, ou seja, no momento em que a *Query* 1 fosse executada também seriam calculadas as estruturas de dados necessárias para o uso do programa. A razão para esta escolha passou pela ideia de tentar criar um programa cuja utilização fosse mais rápida possível para o utilizador. Ao sacrificar memória e mais tempo na *Query* 1, poderíamos acelerar o uso de *queries* seguintes ao ter os dados necessários já calculados.

Esta decisão apresentou os seus desafios, uma vez que a máquina necessária para fazer os testes de uso teria que ter mais memória do que normalmente é necessário, como iremos apresentar mais à frente.

Para auxiliar a rapidez de acesso e execução, teríamos que escolher estruturas de dados cujo tempo de procura fosse muito reduzido. Para tal recorremos à biblioteca Glib que foi recomendada pelos docentes e baseamos o

nosso trabalho em várias **GHashTables** e **GArrays**. Esta decisão foi baseada não só no facto de haver uma boa base de funções pré-definidas na Glib que facilitariam o desenvolvimento do trabalho, mas também na ideia de que o tempo de execução de *lookups* nas GHashTables seria bastante menor que em outras estruturas

O desenvolvimento foi feito de uma forma muito simbiótica. Aproveitando plataformas como o ***Discord*** para comunicação do grupo durante longas reuniões em que a programação era feita alternadamente nos computadores de cada membro com a edição a "seis mãos" recorrendo ao plug-in **Live Share** do **Visual Studio Code**. Além disso, houve também momentos de trabalho mais focado e individual para complementar essas frequentes reuniões.

3. MÓDULOS

3.1BUSINESS

Módulo com as funções relativas ao tipo de dados Business, que por sua vez é constituído por 4 *arrays* para armazenar o Nome, Id, Cidade, Estado e depois uma GHashTable para as categorias. O modo de armazenamento das categorias foi escolhido com base na *Query* 8 (ver análise das querys na secção seguinte).

Este módulo é constituído por várias funções auxiliares get para diversos parâmetros de um dado Business, de forma a preservar o encapsulamento. Existe ainda uma função de free da Struct Business e free de uma HashTable de Businesses (**freeCatalogoBusiness**).

No entanto, a função principal deste módulo é a **readFileBusiness**, responsável por transformar um ficheiro business.csv numa GHashTable. A função lê o ficheiro linha a linha, executando sempre uma verificação da validade de cada campo. Esta verificação é feita com os comprimentos dos vários campos: se algum for vazio ou (no caso do ID que sabemos ter um tamanho específico de 22 caracteres) ter um comprimento diferente do esperado, a linha é considerada inválida.

3.2REVIEWS

Módulo com as funções relativas ao tipo de dados Reviews, constituído por 4 Arrays (*review_id, business_id, user_id e data*) , 3 ints (*useful, funny, cool*) e 1 float (*stars*), além disso há uma GHashTable de texto processado. Esta

GHashTable é constituída por todas as primeiras ocorrências de cada palavra na Review, ou seja, elimina repetições, acelerando o processo de procura na *Query* 9. O módulo é complementado com os gets apropriados para manter o encapsulamento do tipo Review, uma função de cópia de uma GHashTable para o texto processado na função **getReview**.

Tal como no módulo Users e Business, há uma função de leitura de um ficheiro do tipo review.csv, em que a leitura é feita linha a linha. A validação de cada linha passa por confirmar os comprimentos dos id's e por 2 funções de checkDate e checkHour. O processamento do texto é feito na função auxiliar processText que recebe um array de texto e converte para uma GHashTable. O módulo contém ainda funções que serão mencionadas na secção seguinte.

3.3USERS

O tipo de dados User é constituído por 2 strings para armazenar o *user_id* e o nome e, com um raciocínio semelhante às categorias no módulo business, uma GHashTable de estados para auxiliar à Query 7. Foi tomada uma decisão de não armazenar de forma alguma a extensa lista de amigos do utilizador uma vez que isto foi considerado um desperdício de memória já que estes não eram utilizados em nenhuma Query, quer como parâmetros, quer como resultado a armazenar no tipo TABLE.

O módulo possui as funções auxiliares adequadas para manter o encapsulamento e libertar a memória.

O modo de leitura do ficheiro é semelhante aos outros 2 módulos na medida em que segue uma estrutura de leitura linha a linha, validação, inserção na GHashTable CatalogoUsers. A validação de cada linha antes de inserir é feita através de uma verificação dos comprimentos do campo *user_id* (o primeiro a ser verificado, devendo ter um tamanho exatamente de 22 caratéres) e do campo nome (que não deve estar vazio).

3.4SGR

Módulo onde está definido o tipo TABLE, funções apropriadas a esse tipo (nomeadamente gets e sets para preservar o encapsulamento) . Também para o tipo o tipo SGR (constituído por 3 GHashTables, uma para cada catálogo referente a User Reviews e Business) e a sua função **init** e **free**.

É neste módulo que estão as funções que executam cada **Query**.

3.5 auxStructs

Módulo auxiliar aos restantes. Encontra-se definido neste módulo o tipo BizScore, acompanhado da sua função free, de comparação e os gets necessários para preservação do encapsulamento quando usado noutros módulos do trabalho.

3.6 Interpretador

Intermediário entre o utilizador e os restantes módulos. O primeiro desafio foi perceber como processar o input do utilizador de forma a poder passar para uma função **callFunction**. A resposta para esse desafio foi um conjunto de funções.

1. a função **interp**, é responsável por, caso o utilizador insira vários inputs com a separação de ";" , essa linha é separada e colocada num Array definido para armazenar comandos.
2. Esse Array de comandos é então passado por uma das maiores funções do módulo, a função **checkCommand**, que processa cada comando com os delimitadores "=()[]" depois do comando ter sido submetido a uma função auxiliar **tiraEspacos**.
3. Cada comando processado é então submetido a várias condições que, consoante o número de **parametros** (neste momento " x = business_info(y,t)" corresponderia a 3 parametros (x | business_info | (y,t)) , strcmp com os nomes das queries e outros comandos, verifica a validade do input e executa a função apropriada através da função **callFunction**.
4. A função **callFunction** usa um switch que, consoante o número fornecido pela função **checkCommand**, executa um comando associado ao número utilizando também os argumentos (no exemplo anterior, agora equivalentes apenas ao y e t graças à utilização de uma auxiliar **breakParam**) conseguidos na função **checkCommand**

Durante todo a execução de funções de módulo há variadas mensagens de erro para situações de

- número errado de argumentos/argumentos insuficientes;
- argumentos inválidos
- falha ao carregar um ficheiro;

Cada atribuição feita pelo utilizador é armazenada numa estrutura chamada vars_interp. Nesta estrutura, encontram-se os campos ***SGR***, que é o sistema de recomendações carregado quando o utilizador chama a Query **load_sgr**, o nome da variável onde é guardado este sistema de recomendações e, por fim, as variáveis de qualquer query que seja parametrizada. Este último, consiste numa GHashTable em que como chaves são adicionados os nomes das variáveis e como respectivos valores uma tabela (tipo TABLE) com a informação relevante a essa variável. Desta forma, é possível verificar a existência de uma variável quando lhe é atribuída algum resultado de uma Query, para, deste modo, caso ela já tenha um

valor associado, libertar a informação que lá está guardada e atribuir o novo valor à mesma.

É necessário ainda mencionar que a função **show** pode encontrar dificuldades em imprimir toda a linha da tabela dentro do terminal com as dimensões standard. Isto deve-se ao facto de que, no processo de impressão, é calculado o tamanho máximo que uma coluna poderá ter e é esse o tamanho escolhido para todas as colunas.

4. Queries, TABLE e Interpretador

4.1 Query 1

Esta *Query* é responsável pelo load de todos os catálogos (do tipo GHashTable) que constituem o SGR. Para tal, são executadas as funções **readFileUsers**, **readFileBusiness** e **readFileReviews**. Além disso, e de acordo com a filosofia em que baseamos o nosso programa - de ter o máximo de dados tratados no início para (teoricamente) diminuir o tempo de execução das queries seguintes, é também executada a função **reviewsByCity**. Esta função, do **(módulo Stats)**, faz o tratamento dos dados, organizando os negócios por cidade e simultaneamente calculando as pontuações de cada negócio. Na GHashTable resultante as keys são os nomes das cidades e os valores armazenados são GArrays dos negócios organizados por ordem decrescente de pontuação (mais sobre isso na *Query* 5 e 6).

4.2 Query 2

Esta função devolve como resultado a tabela dos negócios começados por uma dada letra. Para tal é executada a função **get_businesses_by_letter** do módulo Business que tem como parâmetros o catálogo de negócios e a letra. Essa função percorre o catálogo de negócios e armazena numa GHashTable os negócios em que a primeira letra é a fornecida pelo utilizador. Esta GHashTable tem como key os ID’s do Business e como valor o nome.

4.3 Query 3

A função usa um simples **getBusiness** para conseguir todos os dados que tem que guardar na TABLE. Para conseguir a pontuação é executada a função **traverseReviewsForStars** (do módulo **Reviews**) que percorre o Catálogo de Reviews somando a pontuação de cada Review cujo **business_id** seja igual ao

fornecido pelo utilizador e incrementando um contador. No fim o valor devolvido pela função traverseReviewsForStars é obtido com uma divisão do valor cumulativo das stars e o contador. Os restantes parâmetros a ser devolvidos são conseguidos com o auxílio de gets apropriados para o tipo Business.

4.4 Query 4

A função utiliza a **traverseReviewsForID** do módulo Reviews que percorre catálogo de Reviews e, a cada Review cujo *user_id* corresponda ao *user_id* fornecido pelo utilizador, retira os dados do business correspondente através de um **getBusiness** e de um **getNameBusiness**. Esses dados são guardados numa GHashTable e convertidos para o tipo TABLE.

4.5 Query 5

A *Query* 5 é a primeira *Query* a aproveitar a vantagem de se ter feito o tratamento de dados logo no início. A única coisa que a função faz quando é executada é retirar o GArray guardado na posição da cidade fornecida pelo utilizador. Depois disso simplesmente percorre o GArray com um ciclo for e retira todos os negócios (juntamente com os dados importantes, guardados numa struct CityStars) com pontuação igual ou superior ou fornecida recorrendo a um **getBusiness**. Uma vez que o GArray está ordenado, o ciclo for faz o menor número possível de iterações para realizar esta tarefa, aumentando a rapidez de execução, em linha com a filosofia de desenvolvimento escolhida. A informação é depois convertida e guardada no tipo TABLE.

4.6 Query 6

A segunda *Query* com a vantagem do tratamento de dados no início do programa, desta vez com diferenças mais notáveis. A função, através de um ciclo *while*, vai iterando ao longo da GHashTable *starStats* guardada no SGR. Em cada iteração, é feito um ciclo for que percorre o Array, guardando na TABLE a informação pertinente (que está em structs **CityStars**) com gets apropriados dos top N negócios nessa cidade.

4.7 Query 7

Uma das únicas funções que faz tratamento de dados on-demand, começa por correr a função **traverseReviewsForState** do módulo Reviews. Essa função

percorre o catálogo de Reviews com um iterador e, com um **getBusiness** e *getStateBusiness*, consegue os dados necessários para executar a função **setState** do módulo **Users**. A **setState** adiciona ao campo *states* (previamente vazio) de um User o estado fornecido (o estado em que o User em questão fez a Review atual no iterador da função anterior. No final da execução da **traverseReviewsForState**, é executada a função **traverseUsersForState** do módulo Users, que guarda num GArray as IDs dos users que tenham mais do que 2 estados guardados no campo "estados" da struct User. Por fim, os dados são convertidos para o tipo TABLE.

4.8 Query 8

Com os dados já calculados no arranque do programa, a nível de ordenação por stars, a *Query* 8 faz a travessia dessa estrutura e armazena num simples GArray os top n negócios que possuam uma dada categoria na lista de categorias. Para o fazer, são usadas funções de gets para fazer o teste da existência da categoria num dado negócio e, em caso afirmativo, é criada uma nova estrutura Business para onde se faz a cópia do negócio analisado e posterior inserção num GArray. Assim, no GArray resultado, ficam as cópias de todos os negócios cuja lista de categorias contém a categoria solicitada.

4.9 Query 9

Esta Query beneficia, tal como a Query 5 e 6, de outra decisão tomada no início do desenvolvimento do trabalho, desta vez no modo de processamento e armazenamento do texto das Reviews, como explicado na secção anterior. A função começa por usar a **findReviewsText** do módulo Reviews. Essa função percorre o catálogo Reviews, fazendo lookups nas GHashTables que contém o texto processado de cada Review, guardando num GArray os IDs das Reviews que possuem a palavra passada como argumento da query. Esse GArray é depois convertido para o tipo TABLE.

4.10 TABLE

Constituído por três GPtrArrays, que são o *nome_campos*, onde são introduzidos os nomes dos campos que constam ordenados por uma ordem específica, o *info_campos*, que por sua vez possui a informação dos respetivos campos pela mesma ordem que consta no array *nome_campos*. Desta forma, é-nos possível imprimir no prompt pela ordem correta e assegurar que cada índice de ambos os Arrays relaciona o tipo de informação com o seu conteúdo. Finalmente,

o terceiro, *max_colunas_campos*, contém o tamanho máximo de cada coluna, valores que são usados na função show.

Cada Query executa uma conversão do resultado das funções para o tipo TABLE, essa conversão varia consoante o nível de colunas e a estrutura de dados a partir do qual está a ser feita a conversão (GArray ou GHashTable). Durante essa conversão é também calculado o tamanho máximo de cada coluna

5. Testes

Foram feitos testes de tempo para cada Query usando a máquina principal que foi usada para testar o programa ao longo do trabalho. Para cada Query foram feitas 3 tentativas, cujas médias de tempo de execução são as seguintes:

Query 1	35.634s	Query 6	0.014s
Query 2	0.071s	Query 7	4.8236s
Query 3	0.116s	Query 8	0.731s
Query 4	0.122s	Query 9	0.679s
Query 5	0.0023s		

Há um gasto de tempo inicial bastante acentuado, como seria de esperar, mas que parecer valer a pena quando se olha para a velocidade da Query 5 e 6 face às outras. Esse investimento inicial seria então, teoricamente, compensado ao longo da utilização prolongada do programa com os ficheiros carregados. O único outlier nos tempos de execução é a Query 7 uma vez que é executado on demand.

No que toca a memória gasta no decorrer da execução do programa, os valores máximos rondaram os 4.5GB, valores altos mas esperados, dado que grande parte do cálculo estatístico era feito no início da execução.

6. Conclusão/Discussão dos resultados

Em conclusão, o programa que desenvolvemos tem todas as especificações pedidas no enunciado a funcionar em condições satisfatórias para os membros do grupo.

Numa reflexão sobre o que gostaríamos de ter feito melhor ou de forma diferente no trabalho, é necessário referir alguns pontos, alguns dos quais que serão tidos em conta também no segundo projeto desta UC:

1. Gostávamos que as funções do interpretador, especialmente a **callFunction** e a **checkCommand** fossem mais curtas, talvez com mais tempo para experimentar com tentativas de separar o que é feito nessa função em funções mais curtas.
2. No que toca à organização do trabalho, como referido anteriormente, a biblioteca Glib foi uma espada de dois gumes: facilitou imenso o trabalho mas também implicou uma grande perda de tempo de trabalho no processo de adaptação às suas funções que poderia ter sido gasto em discussões sobre resposta aos desafios impostos pelo trabalho.
3. Gostávamos de ter tido mais tempo para trabalhar na Query 7 e perceber qual o impacto no tempo de execução de colocar esta Query com a mesma heurística que as restantes.
4. O interpretador integra os comandos **toCSV** e **fromCSV**, que servem para escrever uma TABLE para um ficheiro csv e ler um ficheiro para uma TABLE respetivamente. No ficheiro ".csv" os diferentes campos da TABLE são separados por um delimitador (delim) que é passado como argumento a ambas as funções. Um dos problemas que encontramos é que, caso esse delimitador seja um ';' o interpretador não vai ler corretamente os comandos passados, uma vez que esse mesmo caracter é usado para delimitar uma sequência de comandos no interpretador. Este ponto seria algo a melhorar se pudéssemos refazer o interpretador.
5. Por último, gostávamos de ter conseguido reduzir mais o espaço ocupado pela execução do programa sem sacrificar tempo de execução.

Nota sobre o git:

No decorrer da realização do projeto tivemos um problema com a utilização do github , um dos membros do grupo submeteu um commit para o git de um computador diferente do que normalmente costuma usar para trabalhar. Ao verificar se o commit tinha sido bem feito, reparamos que o utilizador que submeteu o commit não era o membro do grupo (com o nome de utilizador do git obrunofilipe), mas sim um outro utilizador (BrunoFilipe824). Com alguma investigação nas definições do git na máquina onde foi feito o commit reparamos que na verdade essas definições não estavam corretas, em vez do utilizador ser 'obrunofilipe' era 'BrunoFilipe824'. Como este membro do grupo utiliza token's gerados no site do git ,(por ter ativado a verificação a 2 fatores na sua conta), ao submeter o commit com o token gerado por si , mas com um utilizador diferente , o github aceitou o commit com o nome de utilizador errado, mesmo este não tendo acesso ao repositório criado pelos docentes.