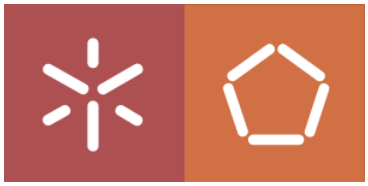

Universidade do Minho
Licenciatura em Engenharia Informárica

Laboratórios de Informática III
2022/2023

Trabalho Prático
Grupo 056

Realizado por:

Gonçalo Marinho	A90969
Luís Caetano	A100893
Maya Gomes	A100822



Gonçalo Marinho Luís Caetano Maya Gomes

3 de fevereiro de 2023

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Introdução

Para este projeto, que foi dividido em duas fases, foi necessário aplicar conhecimentos no que diz respeito à leitura e *parsing* de ficheiros `CSV` e, a partir desses ficheiros, ser capaz de criar coleções e estruturas de dados de forma a conseguir implementar estratégias eficientes e rápidas de realizar as nove *queries* propostas. Para este efeito, recorremos à biblioteca da linguagem C, `GLib`. Este relatório contem os passos e decisões tomados na realização das duas fases do projeto.

Inicialmente é apresentada uma descrição detalhada de como realizamos o encapsulamento e garantimos modularidade, seguida de uma explicação da implementação do *parsing* dos ficheiros, bem como a forma como fizemos o a validação dos mesmos. Posteriormente temos uma explicação das estratégias implementadas no desenvolvimento das *queries* e do modo interativo proposto.

Além disso, apresentamos os nossos resultados obtidos de forma a verificar a rapidez de cada *query* implementada, finalizando com uma breve conclusão.

2 Arquitetura do Projeto

2.1 Encapsulamento e modularidade

Inicialmente, tínhamos as estruturas definidas em todos os módulos, não possuíamos funções de *get* que retornam cópias dos valores/coleções, tínhamos dependências dentro do mesmo "nível"e inicializações na *main*, o que quebrava completamente o encapsulamento. Face a isto, de forma a obter um bom encapsulamento e modularidade no trabalho, o mesmo foi completamente reestruturado. Foram, para esse efeito, criados novos módulos, onde as dependências apenas são dos níveis abaixo em relação aos mais acima, nunca no mesmo "nível"ou de cima para baixo.
Posto isto, é nestes novos módulos que são implementadas coleções especificas, e todas as operações sobre estas, sendo estas estruturas privadas e as operações referidas públicas. Ou seja, apenas as funções no interior do módulo podem aceder a estas estruturas. No entanto, é sempre garantida a opacidade entre módulos através de cópias dos valores. Estas coleções geridas por módulos proporcionam aos módulos dependentes uma interface (API), de forma a que qualquer mudança interna nestas coleções não afete os módulos que são dependentes deste. Isto ocorre, por exemplo, com a estrutura *User* e as respetivas funções *get* que são as únicas com acesso à estrutura mencionada, devolvendo sempre cópias.
Para além disso, de forma a esconder a total implementação do módulo, nos *header files* declaramos apenas as funções que são necessárias noutros módulos, de "nível"inferior.
Este encapsulamento está demonstrado na imagem que se segue (Figura 1). É de salientar que, no diagrama, estão apenas presentes os módulos gerais, sendo que para a parte dos testes os módulos `main.c`, `conector.c`, `executer.c` e `queries.c` foram substituídos por outros idênticos, acrescentando apenas às funções as alterações necessárias, mantendo as dependências entre módulos. Deste modo, de forma a tornar a imagem mais percetível, os mesmos não lhe foram acrescentados.

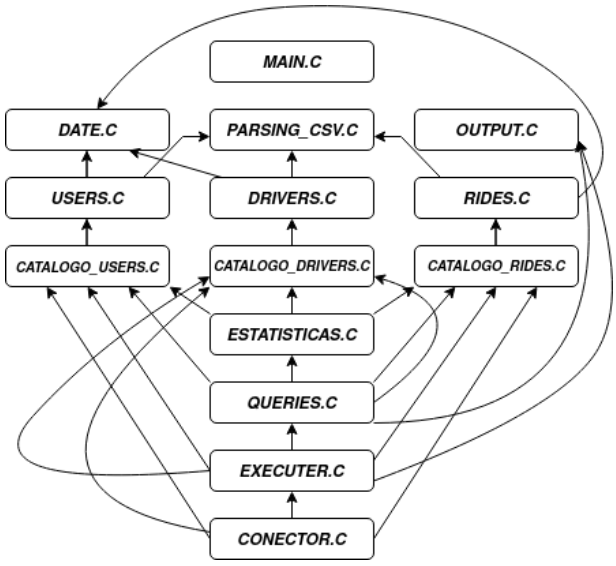

Figura 1: Diagrama do encapsulamento

3 Leitura e Tratamento de Dados

3.1 *Parsing* dos dados

Devido a problemas de encapsulamento e modularidade que explicamos anteriormente no relatório (2.1), realizamos uma reestruturação do modo de como fazíamos o *parsing* dos ficheiros `CSV`, e agora, em vez de termos uma função de *parsing* específica para cada ficheiro, de modo a reutilizarmos código, criámos uma função geral para *parsing* de ficheiros.
Deste modo, a função recebe como argumento uma função que vai ser específica de cada ficheiro. Posto isto, optamos inicialmente por criar um *array* de strings, composto pelos *tokens*, obtidos a partir de cada linha de dados dos ficheiros, os quais são de seguida tratados pela função passada como argumento.

3.2 Validação dos ficheiros de dados

De forma a fazer a validação de entradas de dados durante o *parsing* dos ficheiros `CSV` é passada como argumento uma função específica que tem, no seu corpo, uma função responsável pela validação dos dados de cada ficheiro.
Assim, verifica-se *token* a *token* se o seu campo é ou não nulo e, no caso de estar preenchido, se está conforme o pressuposto. Caso seja válido, é então criada a coleção específica do ficheiro, i.e `User`, `Driver` e `Ride`, e adicionada à sua própria estrutura de dados, contida num catálogo exclusivo.

4 Implementação

4.1 *Queries*

Anteriormente, para a implementação das *queries* tinhamos um catálogo por *query*, o que não era o mais modular e fazia com que a complexidade do código aumentasse conforme íamos aumentado o número de *queries*.
Para combater este problema, também esta parte do trabalho sofreu uma reestruturação, uma vez que implementamos o módulo *estatisticas.c* onde é criado um catálogo composto por todas as estruturas de dados necessárias para o armazenamento das informações relativas a cada *query*. Estes dados estão organizados em coleções que foram implementadas para fornecer a informação necessária para cada *query*. Para isto, existe a função `loadStats` que através de funções implementadas nos módulos dos catálogos de *catalogo_ users.c*, *catalogo_ drivers.c* e *catalogo_ rides*, vai buscar clones das coleções, nomeadamente da `Ride`, percorrendo a estrutura onde esta foi armazenada, de forma a guardar e atualizar toda a informação necessária sob a forma de várias coleções nas estruturas respetivas a cada *query*.
As estruturas de dados definidas no catálogo das estatísticas são apenas a `GHashTable` e o `GPtrArray`. A `GHashTable` foi escolhida para recolher informação nas *queries* 1 a 7, pois como para uma dada *key* a informação tem de ir sendo atualizada, é a estrutura mais rápida para detetar e devolver o campo que queremos atualizar. Já o `GPtrArray` foi a estrutura escolhida para que se pudesse ordenar as coleções que lhe são inseridas.
Todas as funções que podem aceder ao `CatalogoStats` são implementadas em *estatisticas.c* respeitando sempre a opacidade e encapsulamento.

4.1.1 *Query* 1

Nesta primeira *query*, tomando proveito de estruturas de dados das *queries* 2 e 3, acrescentamos às coleções, armazenadas nestas estruturas, parâmetros específicos da *query* 1, de forma a evitar a criação de uma estrutura e coleção específica apenas para a primeira *query*.
Assim, já na implementação da *query*, acedemos aos catálogos dos *users* e *drivers*, de forma a obter a informação relativa ao *name*, *gender* e *birth_ date*.
Para além disso, acedemos ao catálogo das estatísticas para obter a soma total do *score_ user* ou *score_ driver*, consoante o que a *query* pedir, o *n_ viag*, número de viagens, e o *total_ gasto*.
Uma vez recolhidas as cópias destes parâmetros, é calculada a avaliação média, que consiste apenas na divisão do *score* pela *n_ viag*. É também calculada a idade, sendo agora possível gerar o output.

4.1.2 *Query* 2

Nesta *query*, seguindo os princípios enunciados anteriormente, no preenchimento do catálogo das estatísticas, vai guardando e atualizando, na respetiva `GHashTable` (`topdriv`), a coleção `StatsDrivers` para cada driver (*id*), contendo nesta coleção a soma da avaliação que lhes vai sendo atribuída, assim como o número de viagens e a data da viagem mais recentemente efetuada, que é obtida através de uma comparação com a última data guardada.
De seguida, é utilizada a função `Ordena`, conjugada à função de comparação criada de modo a ordenar pela avaliação média, em caso de empate pela viagem mais recente, e caso se

mantenha empatado, pelo *id* do condutor. Assim percorre `topdriv` e vai adicionando ao `GPtrArray` (`Q2`) a coleção `StatsDrivers` nela existente, para esta estrutura de dados depois ser ordenada.
Uma vez invocada a *query* 2, vai ser percorrido `Q2` pelas N últimas posições. À medida que isto ocorre, esta vai aceder ao `CatalogoStats` para obter o *id*, a soma do *score* e o número de viagens. Além disso, também vai aceder ao `DriverCatalog`, para obter o nome do condutor a partir do seu *id*, e assim, gerar o output, calculando a avaliação média para cada posição.

4.1.3 *Query* 3

Esta *query* é análoga á anterior alterando a coleção para a `StatsUsers`, uma vez que esta terceira *query* exige parâmetros diferentes. A forma como as estruturas de dados são utilizadas é bastante semelhante, neste caso, temos agora a `GHashTable` (`topuser`) e o `GPtrArray` (`Q3`).
Para além disso a função de comparação associada à função `Ordena` também é diferente de modo a que `Q3` fique ordenado pela distância (*dist*), em caso de empate pela viagem mais recente (*data*) e em caso de empate pelo *username* do utilizador (*id*).
Ao invocar a *query* 3, `Q3` é percorrido nas N últimas posições, acedendo ao `CatalogoStats` para obter o *username* e a distancia percorrida. De seguida, acede ao `UserCatalog` para obter o nome do utilizador a partir do seu *username*, gerando assim o output.

4.1.4 *Query* 4

Para a *query* 4 foi apenas utilizada a `GHashTable` (`pmed`), em que é utilizada o nome da cidade como *key*, sendo guardado e atualizado a soma do preço da viagem (*preco*) assim como o número de viagens (*n_viagens*). Assim, quando a *query* 4 é invocada, basta fazer uma procura pela cidade passada como argumento na `GHashTable` e, caso exista, é calculado o preço médio e gerado o output.

4.1.5 *Query* 5

Analogamente às *queries* 2 e 3, nesta *query* também vai ser utilizada um `GHashTable` (`pordata`) e um `GPtrArray` (`Q5`). Em `pordata` vai-se adicionando e atualizando a coleção `DataStats`, sendo a sua *key* a data da viagem. Assim, à medida que aparece uma viagem para uma dada data, é adicionado o preço dessa viagem e incrementado em um o número de viagens nessa data.
De seguida, volta a ser invocada a função `Ordena` associada a uma função de comparação, que vai ser responsável por ordenar `Q5` pelas datas de viagem.
Uma vez invocada a *query* 5, esta vai percorrer `Q5`, tendo duas variáveis que vão somando o número de viagens e preço total a partir do momento em que a data coincide com o limite inferior passado como argumento até ao momento em que coincide com o limite superior. Por fim, é calculado o preço médio entre essas duas datas e gerado o output.

4.1.6 *Query* 6

Mais uma vez, também nesta *query*, é utilizada uma `GHashTable` (`dmed`) e um `GPtrArray` (`Q6`). Neste caso a coleção `DMed` vai sendo preenchida e adicionada a `dmed`, cuja *key* é uma

junção da cidade com a data da viagem. Assim, sempre que aparecer uma viagem num dado dia para determinada cidade, o valor da distância percorrida e o número de viagens serão atualizados.
Posteriormente, mais uma vez, é invocada a função `Ordena` com uma função de comparação, de modo a gerar `Q6` ordenado pela data da viagem. Quando invocada a *query* 6, esta vai atuar de forma idêntica à *query* 5 no que diz respeito a percorrer `Q6`, atualizando as suas duas variáveis quando a cidade passada como argumento corresponde à guardada na coleção. No fim é calculada a distância média e gerado o output.

4.1.7 *Query* 7

Novamente, na *query* 7, é utilizada a `GHashTable` (`topcc`) onde é guardada e atualizada a soma da avaliação e o número de viagens da coleção `TopCC` à medida que surge uma viagem de um dado condutor numa dada cidade, por este motivo usamos como *key* de `topcc` uma junção do *id* do condutor com a cidade em questão. Quando `topcc` estiver totalmente preenchida e atualizada é ainda invocada a função `Ordena` associada a uma função de comparação de modo a gerar o `GPtrArray` (`Q7`) ordenado pela avaliação média e, em caso de empate o *id* do condutor. Quando esta *query* é invocada, o output vai ser gerado para as últimas N posições de `Q7` em que a cidade guardada em `TopCC` coincide com a passada como argumento, bastando efetuar o cálculo da avaliação média, aceder a `DriverCatalog` para obter o nome do condutor a partir do *id* do mesmo e, assim, gerar o output.

4.1.8 *Query* 8

Para esta *query* foi apenas utilizado um `GPtrArray` (`Q8`), uma vez que não vão haver valores a serem atualizados, principal motivo para a utilização da `GHashTable` nas restantes *queries*. Deste modo, sempre que o género do condutor e do utilizador são idênticos, vai ser adicionado a `Q8` a coleção `TopViagem`. Uma vez preenchido, `Q8` vai ser ordenado a partir de uma função de comparação que o ordenará pela data de criação da conta dos condutores, em caso de empate pela dos utilizadores e, em caso de o empate persistir, pelo *id* da viagem.
Assim uma vez invocada a *query* 8 esta vai percorrer `Q8`, verificando se o género guardado em `TopViagem` num determinado índice de `Q8` corresponde ao que é passado como argumento e se a idade das contas do utilizador e condutor são maiores ou iguais à passada como argumento, gerando assim o output.
Por outro lado, achamos que uma otimização seria criar uma estrutura de dados para cada um dos géneros, em vez de apenas uma para os dois, uma vez que tem de percorrer um maior número de índices.

4.1.9 *Query* 9

Esta última *query* foi feita de forma análoga à anterior, utilizando também um `GPtrArray` (`Q9`) onde se vai preenchendo e adicionando a coleção `VG` sempre que o passageiro deu gorjeta.
De seguida, `Q9` é ordenado utilizando uma função de comparação de modo a que seja ordenado pela distância média, em caso de empate, as viagens mais recentes e em caso de persistência do empate o *id* da viagem.

Quando invocada a *query* 9, vai ser percorrida a estrutura `Q9` e sempre que para dado índice a data da viagem esteja dentro do intervalo passado como argumento vão-se buscar os parâmetros necessários para gerar o output.

4.2 Modo interativo

Para o modo interativo, foi criada uma coleção (`Paginacao`) que possui uma estrutura de dados com o intuito de armazenar a informação à medida que se vai formando um output nas *queries*, sendo escolhido o `GPtrArray` (`dados`), para além disso são utilizados também inteiros de modo a facilitar a paginação.
Deste modo, no fim de uma *query*, a coleção encontra-se preenchida ao que se segue um modo interativo que permite ao utilizador ver até 20 valores de output simultaneamente, permitindo avançar e recuar de página, ir diretamente para a última página e voltar para a primeira (Figura 2). Enquanto o utilizador quiser, pode continuar a executar *queries* até o mesmo pretender encerrar o programa (Figura 3).

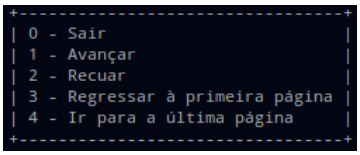

Figura 2: Menu de paginação

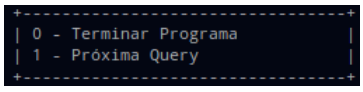

Figura 3: Menu final

Para fazer a paginação, aproveitamos os valores *liminf*, *limsup* e *pagina_ atual* da coleção mencionada anteriormente. Deste modo, uma vez que para imprimir os valores basta aceder aos índices do `GPtrArray`, implementamos funções que percorrem os valores do índice de 20 em 20, atualizando, conforme a escolha do utilizador, os limites inferior e superior para imprimir sempre os valores nesse intervalo, ou caso existam menos do que 20, imprime até ao valor da *length* da estrutura de dados. A *pagina_ atual* vai atualizando conforme a função chamada, incrementando, decrementando ou assumindo o valores específicos, "0"para a primeira página e *length*/20 para a última.
As interações com o utilizador são todas à base de leitura do input na linha de comandos (*scanf's*), avaliando as suas respostas. Para executar as *queries*, é necessário fornecer o número da mesma e, de seguida, os restantes argumentos tal como se encontram no enunciado do trabalho prático. No fim da sua execução, inicia-se o outro modo interativo explicado anteriormente, relativo à paginação e apresentação de resultados.

5 Testes funcionais e de desempenho

5.1 *Testes*

Para esta parte do trabalho foram utilizadas as bibliotecas `time.h` para obter o tempo de execução das *queries* e `sys/resource.h` para obter o *maximum memory resident size*. Deste modo, é necessário passar como argumento o nome da pasta onde se encontram os outputs esperados, havendo uma função que compara os ficheiros criados pelas *queries* implementadas com os ficheiros com o output pretendido, lendo linha a linha os dois ficheiros e comparando as strings. Esta função vai então verificar se a *query* executou dentro do tempo estipulado e se os ficheiros são iguais, originando o output consoante os resultados obtidos.
Devido a pequenos erros na representação de valores reais em C, por vezes acontece, nomeadamente nas *queries* 2 e 7, que a terceira casa decimal defira em 1 unidade, o que ao comparar strings leva a que assuma que o valor, e consequentemente a *query*, estejam errados, quando na verdade a mesma está bem executada.

5.2 *Memory Leaks*

Na primeira fase do trabalho, o método utilizado gerava imensos erros e *memory leaks* devido às dependências de dados entre estruturas de dados que tornavam difícil libertar a memória, problema que foi corrigido através de uma reestruturação do mesmo.
Atualmente o trabalho tem apenas as *memory leaks* associadas à utilização da biblioteca `GLib`, que são 0,019 MB. Ao longo do trabalho fomos sempre tentando prevenir a ocorrência das mesmas através da criação de funções para fazer *free* tanto das estruturas de dados como das coleções. Para além disso, sempre que foi necessária a utilização de memoria dinâmica, a implementação do código foi pensada de modo a poder libertar a memória alocada assim que possível.

5.3 Tempo de execução

Embora o objetivo seja conseguir que as *queries* executem o mais rápido possível, em algumas *queries* optamos por fazer um simples cálculo em vez de fazer vários durante o *load* de dados, pois consideramos que embora isto possa aumentar muito ligeiramente o tempo de execução da *query*, no quadro geral fazemos muito menos operações e diminuímos o tempo de execução a nível geral.
Na tabela seguinte (Tabela 1), para cada sistema operativo dos membros integrantes do grupo foi calculada para o *large dataset* a média do tempo de execução das *queries*.

Sistema Operativo	Processador	Armazenamento	RAM	*Query*	Tempo de Exec.
Linux Manjaro 5.3.19	i7-6500U	SSD	8GB	1	0,000075
				2	0,020011
				3	0,036240
				4	0,000054
				5	0,001048
				6	0,009042
				7	0,056412
				8	0,428733
				9	4,279060
Linux Mint 21	i7 4thGen	SSD	8GB	1	0,000041
				2	0,009356
				3	0,018268
				4	0,000036
				5	0,000615
				6	0,006201
				7	0,036551
				8	0,335438
				9	3,091138
Ubuntu 22.04	I5-8250U	SSD	12GB	1	0,000061
				2	0,026543
				3	0,062774
				4	0,000054
				5	0,000919
				6	0,009633
				7	0,036433
				8	0,457186
				9	4,2931245

Tabela 1: Tempos de Execução

6 Conclusão

Tendo como base a primeira fase deste trabalho prático considerámos que melhorámos o nosso trabalho consideravelmente, o que se pode verificar por exemplo, uma vez que temos zero *memory leaks*, ao contrário de na primeira fase que possuíamos imensas, para além de erros que surgiam quando utilizada a ferramenta *valgrind*.
Além disso, na versão final do projeto, conseguimos ter uma carga computacional praticamente toda no `load` de forma a que as *queries* apenas tenham de ir buscar a informação necessária, o que é bastante mais rápido. Alteramos ainda as coleções de modo a termos mais tipos que não exigem alocar mais memória dinâmica, o que também permitiu reduzir bastante o tempo de execução.
Por outro lado, consideramos que conseguimos implementar um código que contempla boas práticas de encapsulamento, modularidade e opacidade. No primeiro, alcançamos um bom encapsulamento tendo sempre em conta uma hierarquia entre módulos, em que apenas existem dependências de baixo para cima e nunca no sentido inverso ou no mesmo "nível". Em relação à modularidade, garantimos que, apenas é possível gerir um determinado módulo com as funções do mesmo e, para além disso, tornámos os detalhes de implementação dos módulos privados e públicas apenas as funções necessárias a outros módulos. Já no que toca à opacidade, entendemos que esteja assegurada uma vez que nunca são fornecidos os valores reais dos dados, mas sim cópias ou clones dos mesmos.
No entanto, mesmo com todas as otimizações efetuadas, reparámos que o nosso tempo de execução, em comparação com outros grupos, é superior, o que acreditamos que seria um pequeno desafio conseguir diminuir. Talvez isto fosse possível através de uma possível reutilização de estruturas de dados para as *queries*.
Por fim, consideramos que, com a realização deste trabalho conseguimos melhorar o nosso método de programação, tomando conhecimento de novas bibliotecas, neste caso a `GLib`. Além disso, foi ainda possível consolidar melhor os conceitos de encapsulamento, modularidade e opacidade, que serão certamente úteis em trabalhos futuros.