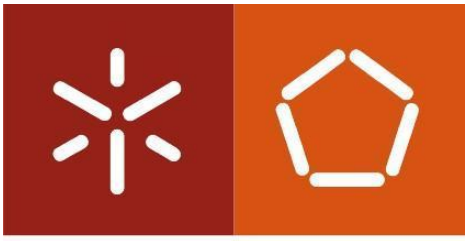

Universidade do Minho

Relatório Projeto LI3 fase 2

grupo 99

a100646, Diogo Ribeiro Vassalo de Abreu, DiogoAbreu19

a100610, José Afonso Lopes Correia, Ze-Limao

a100665, Tiago Nuno de Magalhães Teixeira, T-yago

Índice

Introdução

O presente relatório tem como objetivo apresentar a segunda fase do projeto realizado no âmbito da unidade curricular de Laboratórios de Informática III, ao longo do primeiro semestre, do segundo ano, da Licenciatura em Engenharia Informática da Universidade do Minho.

Este consiste na criação de um programa capaz de ler, armazenar e gerir toda a informação válida contida nos ficheiros *.csv* (*users.csv*, *drivers.csv*, *rides.csv*). Os dados são armazenados de forma a futuramente conseguir responder aos diferentes pedidos dos utilizadores da forma mais rápida e eficiente possível e, por fim, devolver os resultados.

A aplicação vai ter dois modos de execução (Batch e Interativo).

No modo Batch, o programa é executado com dois argumentos, o primeiro é um *"path"* para a pasta que contém os 3 ficheiros *.csv*, o segundo corresponde ao *"path"* para um ficheiro de texto que contém uma lista de comandos, sendo um comando por linha.
Um comando possui a seguinte estrutura: <query−id> [arg1...argN].

Estes comandos serão executados e o resultado da execução de cada comando X é escrito num ficheiro *commandX_output.txt*, que estará localizado na pasta "Resultados" da raiz da pasta "trabalho-pratico".

No modo interativo, o programa é executado sem argumentos, ou seja, apenas com o comando **./programa-principal**. Neste modo, é o utilizador que define tudo, desde a seleção da query até à seleção dos argumentos a executar para cada query. Tal como o nome indica, é utilizada uma interface gráfica para mostrar os resultados.

Arquitetura da Aplicação

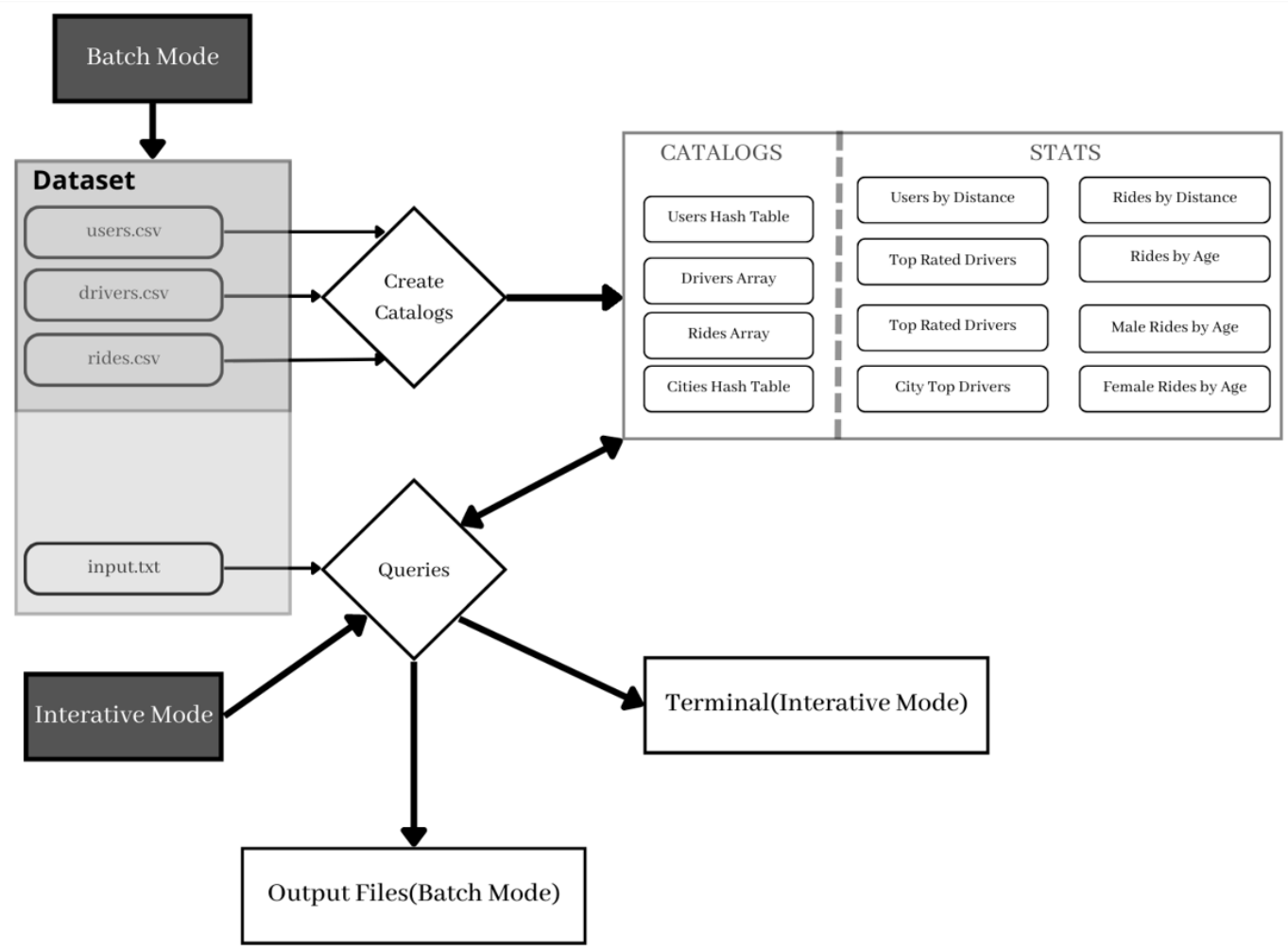

Figura 1- Arquitetura do projeto.

Tal como pode ser observado pela figura 1, o projeto parte ou o modo batch ou do modo interativo, o que depende do número de argumentos que a função "*main*" recebe.

Se o modo executado for o batch, então o processo começa por criar os catálogos a partir dos csv cujo caminho foi especificado como argumento, Juntamente com os catálogos, são criadas as estruturas auxiliares que vão ser usadas para responder as queries.

Após isso, o ficheiro de input é lido e interpretado, de modos que cada query e respondida e gera um ficheiro de output a seu respeito.

Caso não sejam passados argumentos para a função main, o modo interativo é acionado e de acordo com o ficheiro recebido pelo stdin também ele cria os catálogos e estruturas auxiliares, com a diferença que as queries nao sao lidas de um ficheiro mas sim do stdin que o utilizador escreve. O seu resultado e também demonstrado graficamente no terminal.

Desenvolvimento

Resumo do que foi realizado na 1ª fase do projeto

Na primeira fase do projeto foram criados os catálogos dos users e dos drivers, tendo ambos sido implementados sob a forma de uma hashtable.

No que toca aos rides, não era criada de todo uma estrutura para os catalogar: apenas se utilizava o conteúdo que era lido a cada linha para preencher os campos necessários nas estruturas dos users e dos drivers.

O parsing era ainda realizado de forma específica para cada ficheiro .csv e realizado pois em cada módulo que envolvia catálogos.

No que toca às queries, as estruturas auxiliares eram definidas no próprio ficheiro da query e eram criadas a cada chamada da query.

Em relação ao encapsulamento, os catálogos não estavam de todo encapsulados. já qie os apontadores para as estruturas principais eram passados de ficheiro para ficheiro livremente e as suas definições não eram opacas entre ficheiros.

Os catálogos devolviam uma referência direta para a sua estrutura. Dessa forma, era perfeitamente possível um utilizador do programa usar essa função para eliminar ou alterar propositadamente os valores armazenados nos catálogos.

De modo a preencher, por exemplo, o catálogo dos users com informações obtidas através da leitura do ficheiro das rides, eram ainda utilizadas funções violadoras do encapsulamento, como é o exemplo da função "*inc_aval_media_user*" que incrementava o valor de um user por um inteiro passado por argumento. O problema evidente é, mais uma vez, a violação do encapsulamento, na medida em que qualquer utilizador do programa podia intencionalmente alterar o valor da avaliação média de um user a seu desejo.

Implementações relativas à segunda fase do projeto

Estruturamento dos catálogos

Face aos novos inputs da segunda fase, o projeto foi alterado de forma a acomodar de forma mais eficiente os dados armazenados nos catálogos.

Users

A estrutura de dados utilizada para armazenar os users manteve-se: uma hashtable, mais especificamente uma hashtable importada da "glib", uma vez que continua a ser a forma mais eficiente de indexar um usuário através do seu id, que neste caso é do tipo string. Atendendo a estes requisitos, incluímos no nosso código a biblioteca *glib2.0* e utilizamos as funções associadas a *hashtables* da mesma, de nomear a função de criação e a de procura de um valor.

Quanto à struct individual de cada user, essa foi alterada na medida que existiam campos a mais que gastavam memória e não eram intrinsecamente necessários durante todo o funcionamento do programa. Exemplos destes campos são a "avaliação_total_user" acompanhada do campo "avaliação_média_user" que foi eliminado mantendo-se apenas o primeiro, que é agora atualizado no momento apropriado para o bom funcionamento do programa.

Estruturas auxiliares dos users

Através de cópias dos dados dos users catalogados, é criado um array auxiliar de uma struct definida unicamente para preencher os requisitos de ordenação da query 3.

Assim sendo, são utilizadas as funções "getters" para preencher com o array com cópias dos campos desejados do catálogo sem referências diretas à estrutura opaca do catálogo.

Após isso, o array é ordenado através do uso da função “qsort”, que se revelou ser a mais eficaz para esta estrutura, e é colocado no catálogo dos users, de forma a poder depois ser consultado pelo módulo da query 3.

Drivers

No que toca aos drivers, foi realizada uma alteração em relação à sua estrutura, passando de se utilizar uma hashtable para um array.

Isto porque, ao contrário do que acontece com os users, o id dos drivers não passa de um número que é sequencialmente maior ao longo do seu ficheiro “.csv”.

Para além disso, os inputs inválidos mantêm este invariante de ordem dos ids dos drivers. Ultimamente, isto permite-nos apagar o campo “id” da estrutura individual de cada driver, e ao invés disso relacionar as propriedades dos índices de um array com o id do driver para assim conseguir obter a posição de um driver qualquer, com um certo id, no array.

De uma forma geral, a implementação de um array foi bastante útil na medida que nos permitiu fazer buscas de drivers em tempo constante, ao contrário do que acontecia com a hashtable.

Estruturas auxiliares dos drivers

Utilizando as funções de obtenção de cópias dos catálogos dos drivers, é criado um array ordenado de acordo com os requisitos da query 2.

Mais uma vez, utilizamos o quicksort para realizar a operação de *sorting* e posteriormente adicionamos o array ao catálogo.

Rides

Ao contrário do que acontecia antes, as rides têm agora também o seu próprio catálogo. Tal como para os drivers, foi escolhida para estes a estrutra de um array.

No entanto, neste caso, a indexação das rides a partir dos seus ID’s não é tão simples. Isto porque , apesar dos ids das rides poderem não ser sequenciais, as viagens são guardadas sequencialmente, sem “lacunas” entre si, para evitar perdas de memória.

Assim sendo, a solução encontrada foi guardar o id na estrutura de cada ride, de forma a poder consultá-la mais tarde. A maior preocupação passa então, por perceber se certas funções precisam de receber o ID da ride ou a posição onde esta se encontra no array das rides, detalhe que está exaustivamente documentado na implementação do projeto.

Estruturas auxiliares das rides

Para auxiliar nas queries, como vai ser explicado mais tarde, foram criados um array para ordenar as viagens por distâncias e dois arrays nos quais estão incluídas as viagens nas quais o user e o driver são do mesmo género, um para o género masculino e outro para o feminino.

Cada um destes arrays é ainda ordenado em função da data de criação do user, do driver e do id da ride.

Cities

Nesta fase do projeto decidimos criar um novo catálogo chamado "*cities*".

Dado que múltiplas queries necessitavam de informação proveniente de apenas uma das cidades do catálogo das rides, surgiu a ideia de agrupar os dados por cidade de forma a tornar a procura dos dados desejados numa cidade particular mais eficiente.

Assim sendo, optamos por utilizar uma vez mais uma hashtable, uma vez que é conveniente usar a função "hash" para, aqui também, indexar as informações de uma cidade sob a sua string literal.

Ao contrário do que acontece nas estruturas singulares dos usuários, rides e drivers, em que a maioria dos campos são de armazenamento das informações lidas no csv, neste catálogo todos os membros da estrutura "cidade" contida na hashtable foram escolhidos pensando unicamente em responder com eficiência às queries que pretendem informação sobre uma destas estruturas.

Assim sendo, o módulo das cidades é todo ele praticamente um catálogo auxiliar.

Cada cidade tem as informações relativas ao total gasto na mesma, o número de viagens e ainda dois arrays, um com as avaliações médias dos drivers naquela cidade em particular, e outro com os índices onde encontrar as viagens realizadas naquela cidade no array das viagens.

Parsing

Foi implementada uma função genérica de parsing de um ficheiro do tipo “.csv”, de forma a se poder reutilizar o código para cada módulo que assim o faz.

Ao receber um “*function pointer*” para uma função que cria, a partir dos tokens selecionados, uma estrutura singular para cada módulo, evitamos a repetição sem utilidade de código.

De notar que, as estruturas que criam unidades singulares estão apenas definidas no próprio módulo do catálogo de onde provém. Isto é, no caso dos users, por exemplo, a função “create_user” que cria um user a partir da informação lida pelo parser não está definida no header file do ficheiro dos uma vez que a sua implementação está apenas no “users.c” e por isso não pode ser utilizada fora deste módulo, o que contribui para o encapsulamento desta estrutura.

Inputs inválidos

Para resolver a questão do tratamento dos dados inválidos presentes no .csv foram criadas funções que verificam para um determinado catálogo, se a linha lida pela função de parsing é efetivamente válida de acordo com os critérios contidos no enunciado do projeto.

Para exemplificar, como os requisitos para uma estrutura individual ser válida diferem de estrutura para estrutura, o ficheiro “*rides.c*” possui uma função “*is_valid_ride*” que valida uma linha obtida pelo parser. Esta e as outras semelhantes funções são ambas chamadas aquando da criação da estrutura auxiliar.

Assim, para o caso dos rides e dos users, em que não são utilizadas as propriedades do índice do array, a estrutura individual destes não é criada de todo, quando uma linha é inválida. No caso dos drivers, como a posição do array em que se encontra um driver é dada pelo seu id, então para um driver inválido é alocado espaço, mas coloca-se o seu *status* como invalido, de forma a não interferir nos resultados das queries.

Além disso, o facto de ser realizada uma verificação dos tokens antes de estes serem atribuídos aos campos das estruturas permite fazer algo interessante: No caso do estado da conta, por exemplo, se o tokens correspondente a esta string é válido então não precisamos de guardar toda a string no catálogo, mas apenas o primeiro caracter, “a” ou “i”, de “active” ou “inactive” respetivamente, permitindo assim poupar alguma memória.

Resolução das Queries

Query 1

Para a resolução da *query1*, no caso dos *users* continuamos a aproveitar o facto de termos inicialmente criado uma *hashtable* para utilizar a função ***g_hash_table_lookup*** incluída da biblioteca ***glib.h***.

No caso dos drivers, no entanto, a maneira de indexar um driver no seu array é agora procurar no array pela posição cujo valor é o seu id.

Assim sendo, responder aos campos pedidos pela query passa apenas por chamar as funções "*getters*" definidas nos seus módulos e imprimi-las no ficheiro desejado.

Query 2

Foi mantida a ideia de criar um array ordenado, mas desta vez, ao contrário do que acontecia na primeira fase, o array contém apenas os campos necessários para a ordenação, sendo os restantes campos pedidos pelas queries calculados a partir do id do driver e das funções getters presentes em seu módulo.

Os campos desta estrutura são também eles preenchidos com cópias dos campos originais, presentes no array dos drivers. Para a ordenação de arrays manteve-se a escolha do *qsort*.

Além disso, a sua criação está agora acompanhada do módulo dos drivers e não no ficheiro das queries, com o propósito de manter o seu encapsulamento e de tornar o módulo da query mais conciso e útil.

Query 3

Tal como anteriormente, são ainda feitas cópias dos dados que se encontram na hashtable dos utilizadores, preenchemos uma estrutura criada para o propósito da *query3* e depois ordenamos em função do que é pedido no enunciado, também com uso do **qsort**.

Query 4

Uma vez criada a estrutura cidades, obter o preço médio das viagens (sem considerar gorjetas) numa determinada cidade tornou-se bastante intuitivo: para uma qualquer cidade pedida pela função, caso ela exista na hashtable das cidades, então basta calcular o preço médio nessa cidade, que já estará pois calculado.

Query 5

Esta query difere da quatro por restringir um intervalo de tempo. Tal como referido na primeira fase do projeto, as datas são convertidas utilizando a função "convert_to_day" definida no módulo "*dates*" de forma a representar o número de dias que passaram desde até à data de referência como um unsigned short int.

Assim sendo, responder à query 5 passa por percorrer o array das rides e realizar comparação de inteiros para perceber se a data de uma ride específica está ou não no intervalo desejado.

No final, com o valor total do preço total e o número total de viagens realizadas, procede-se a uma divisão do tipo *double* para obter o preço médio.

Query 6

Determinar a distância média percorrida, numa determinada cidade num certo intervalo de tempo é mais fácil do que aquilo que foi feito para a query 5.

Isto porque, tal como foi referido anteriormente, existe um array com as posições do array das viagens onde podem ser encontradas todas as viagens feitas naquela cidade em concreto.
Assim sendo, os dados estão restritos a uma cidade em particular e por isso basta utilizar as funções "getters" disponibilizadas no módulo das rides para obter os dados referentes àquelas viagens em específico e verificar se as datas dessas viagens se encontram no intervalo de tempo desejado para calcular a média.

Query 7

Para calcular os top "N" condutores de uma determinada cidade, recorremos novamente à utilização da estrutura das cidades para restringir os dados através da mesma.

Para calcular a avaliação média de um driver numa cidade específica, cada cidade possui um array cujo comprimento é o número de drivers existentes.

Desta forma, sempre que uma nova viagem válida aparece, a avaliação que o driver recebeu na mesma é adicionada, no array presente na estrutura da cidade na qual foi realizada a viagem, na posição cujo valor é o id do driver.

No final, cada array (de cada cidade) é ordenado usando uma combinação do qsort com o foreach(), da biblioteca glib, permitindo assim obter um array ordenado para cada cidade. Com este implementado, basta usar as funções "getters" para obter as informações pedidas na query.

Query 8

Tal como foi referido anteriormente, existem dois arrays no catálogo das viagens, um para o sexo feminino e outro para o sexo masculino, que guardam ambos os ids do driver e do user, ambas as datas de criação de conta e o id da ride para cada viagem realizada em que o sexo do condutor e do utilizador são o mesmo.

Desta forma, para qualquer chamada da query basta procurar num dos arrays, que está já restringido por género e ordenado de acordo com os parâmetros pedidos pela query, pelas viagens em que a idade de ambos é maior ou igual à passada por argumento.

A comparação de datas é feita mais uma vez a partir da data convertida para unsigned short int pela função "convert_to_day".

Assim sendo, a idade passada como argumento para a função da query é convertida e o seu significado torna-se o número de dias que passaram desde o momento em que uma pessoa, cuja conta tem essa idade, criou a conta, até à data de referência.

Logo, como seria de se esperar, para verificar se um condutor e um utilizador têm contas à pelo menos uma idade arbitrária "x" basta verificar se as suas datas de criação da conta são, em valor, menores ou iguais à data de referência, também esta convertida, significando isto, no sentido literal, que as contas têm "menos dias" do que o "número de dias" que a idade passada como argumento tem.

Query 9

Para auxiliar a resolução desta query, recorremos ao array que está catalogado no ficheiro das viagens e que contém as informações "id da ride", "data da ride" (convertida como uns. short int), "distância da ride" sobre a viagem, assim como o índice do array das viagens no qual se encontra essa viagem.

Tal como esperado, no array estão apenas contidas as viagens nas o user deu uma gorjeta e o array está ordenado de acordo com os critérios pedidos pela query.

Assim sendo, a resposta a uma chamada desta query passa por percorrer o array ordenado e selecionar dele as viagens em que a data se encontra no intervalo passado como argumento. Para tal, usa-se, mais uma vez, a função de conversão de datas.

Surge um problema, no entanto, já que o exercício nos obriga a retornar a data da viagem como uma string. Assim sendo, utilizamos uma função de conversão reversa que transforma a data (em uns. short int) de volta para uma string.

Modo Interativo

O modo inicia com o pedido ao utilizador pelo caminho para a pasta com os ficheiros csv que pretende analisar. A partir deste caminho, são criadas todas as estruturas de dados necessárias para a execução de cada query (invisível ao utilizador) e este depara-se com um menu de queries. O utilizador escolhe a opção desejada entre as 9 queries ou mesmo sair do modo interativo. Ao escolher a query desejada, o utilizador deve introduzir os argumentos necessários à execução da query de acordo com a informação apresentada após a seleção da query desejada.

Para as queries que envolvem apenas resultados pequenos tal como a apresentação de um perfil de um utilizador ou driver, preço médio das viagens numa determinada cidade… o output é apresentado num simples retângulo. É depois possível ao utilizador continuar na mesma query premindo **C** ou então voltar ao menu das queries premindo **Q**.

Para as queries que envolvem resultados grandes, é apresentada uma tabela com 30 linhas por página, onde o utilizador pode navegar pelas diferentes páginas premindo **N** para avançar para a seguinte e **B** para retroceder à anterior.

Encapsulamento e modularidade

Como este projeto já possui uma dimensão considerável, era essencial modular o projeto de forma a simplificar a sua estrutura, possuir um melhor controlo sobre os dados e também facilitar o seu futuro desenvolvimento em equipa. Para trabalhar no projeto tivemos sempre em conta os princípios da modularidade (divisão do código-fonte em unidades separadas coerentes) e do encapsulamento (garantia de proteção e acessos controlados aos dados).

No que toca a **modularidade** dividimos o código-fonte em vários módulos, de forma a tornar o programa mais organizado e de modos que seja mais útil para o consumidor final. Para tal, seguimos a estratégia de dividir problemas em subproblemas menores e de criar funções que os resolvem. Além disso, tivemos ainda o cuidado de criar soluções genéricas que podem ser reutilizadas durante todo o código.

Já no que toca ao **encapsulamento**, foi tido em grande cuidado, durante todo o projeto, que nenhuma função ou declaração de estrutura compromete a infraestrutura dos dados guardados nos catálogos.

Assim sendo, as próprias estruturas dos catálogos contribuem em grande parte para a preservação do encapsulamento.

Na main do programa acontecem apenas três coisas:

1 - Criação dos catálogos

2 - Resolução das queries

3 - Limpeza da memória associada aos catálogos

Assim sendo, e de forma a conseguir enviar de forma mais simples os catálogos que vão ser usados na resolução das queries para a função "queries_handler", foi criada uma estrutura que contém todos os catálogos (figura 1).

```
struct catalogs {
Catalog_Users* catalog_users;
Catalog_Drivers* catalog_drivers;
Catalog_Rides* catalog_rides;
Catalog_Cities* catalog_cities;
}
```

Figura 2: Definição da estrutura dos catálogos.

Isto torna-se particularmente interessante pois a criação de um módulo "catalogs" insere também novas funções potencialmente úteis para o usuário final, como por exemplo as funções que devolvem apontadores para os catálogos individuais.

A principal preocupação na criação de catálogos foi manter a integridade dos dados contidos em cada catálogo. Para isso, tivemos muito cuidado ao definir a estrutura individual de cada um.

```
struct catalog_users {
GHashTable * hash_users;
User_Distance_Data* top_N_users;
int top_N_users_length;
};
```

Figura 3: Definição da estrutura do catálogo dos users

Tal como se pode observar pela figura 3, a estrutura do catálogo serve como uma forma de encapsular os apontadores para as estruturas; neste caso, dos users, mais especificamente da hashtable dos users e do array com os users ordenados usado pela query 3.

Esta estrutura, assim como a dos outros catálogos, é definida apenas no próprio ficheiro ".c", estando no seu "header file" apenas um "typedef struct …". Isto significa que qualquer módulo que importe este não consegue aceder aos seus campos. Ultimamente, isto traduz-se no facto de que um utilizador do programa não consegue modificar os catálogos, por exemplo, editando as informações de um utilizador ou condutor propositadamente.

Por esta razão, todas as funções do projeto que tratam de relacionar módulos recebem apenas como argumentos os apontadores para os catálogos e não apontadores para as estruturas catalogadas em si.

Os nomes de cada função foram pensados para corresponder à sua funcionalidade literal, tornando assim o código mais legível e intuitivo, de forma a facilitar a sua compreensão por outros programadores que possam trabalhar no mesmo, futuramente.

Documentação

A documentação foi realizada com o uso do Doxygen. Assim sendo, basta utilizar o ficheiro "Doxyfile" e correr o comando "doxygen" para obter, em html ou Latex, uma página com todas as funções definidas e organizadas por ficheiros, alfabeticamente, entre outros.

Além disso, o Doxygen permite visualizar gráficos "dotgraphs" de dependências de ficheiros baseado na leituras das linhas "include ..".

Makefile

O Makefile foi usado para facilitar a compilação do programa, o que tornou muito mais prática a execução de testes ao longo do desenvolvimento do programa, pois com apenas um único comando, **make**, conseguimos criar os ficheiros executáveis (programa-principal e programa-testes), que necessitam de todos os ficheiros objetos (*.o)* resultantes da compilação dos respetivos ficheiros de código-fonte (*.c)*. E posteriormente com o comando **make clean** conseguimos remover tanto os executáveis como os ficheiros objetos usados na criação dos mesmos, e ainda remover também a pasta com os ficheiros de output.

Foram ainda criados dois novos comandos, que facilitam ainda mais a utilização do programa, o comando **make run-programa-principal**, que compila todos os ficheiros, cria o executável e executa o mesmo. E para o caso de querermos testar os resultados obtidos, o tempo, a memória utilizada e possíveis *memory leaks*, utilizamos o comando **make run-programa-testes**, que compila todos os ficheiros necessários, cria o respectivo executável e executa-o.

Sem o *Makefile* para cada vez que quiséssemos testar o programa, implicaria compilar individualmente cada ficheiro *de* código-fonte alterado, e posteriormente criar o executável com a junção de todos os ficheiros objetos.

Testes funcionais e de desempenho

Os testes têm como objetivo verificar de forma rápida e prática a validade dos resultados obtidos nas queries e também calcular a memória utilizada na execução do programa, assim como os tempos de processamento (obtidos com as funções da biblioteca *time.h*).
Para isto, precisaríamos de um novo executável (programa-testes), este programa-testes vai executar o programa normalmente, porém desta vez calcula os tempos de

processamento das queries e por fim chama uma nova função que compara os ficheiros obtidos com os valores de referência, calculando a percentagem de ficheiros idênticos por Query.

Tempos

Relativamente ao custo computacional, vemos como o mesmo se comporta pela tabela abaixo:

	Regular (1)	Large (1)	Regular (2)	Large (2)	Regular (3)	Large (3)
Users	0,163943s	1,553278s	0,168954s	1,899534s	0,152678s	1,486423s
Drivers	0,013853s	0,146438s	0,148329s	0,152933s	0,013456s	0,142342s
Rides	1,000143s	10,805333s	1,014934s	15,238329s	1,001788s	10,994435s
Aux	1,2037549s	19,758923s	1,270323s	22,234951s	1,103967s	19,238864s
Query 1	0,000256s	0,010343s	0,000347s	0.016542s	0,000173s	0,007060s
Query 2	0,000029s	0,008622s	0,000032s	0,010432s	0,000050s	0,008091s
Query 3	0,000030s	0,017243s	0,000034s	0,019984s	0,000024s	0,017540s
Query 4	0,000081s	0,000804s	0,000084s	0,000846s	0,000068s	0,000790s
Query 5	0.063932s	9,788932s	0,066745s	9,808939s	0,053538s	8,016104s
Query 6	0,015349s	2,496384s	0,017003s	3,204362s	0,016216s	2,829082s
Query 7	0,000059s	0,016758s	0,000069s	0,021986s	0,000050s	0,014909s
Query 8	0,009123s	0,198364s	0,0099832s	0,231202s	0,009373s	0,200136s
Query 9	0,003643s	8,237543s	0,003923s	8,786742s	0,044053s	8.024355s

Tabela 1: Tempos do programa em que cada valor foi calculado com uma amostra de 5 valores diferentes e em que “1”, “2” e “3” correspondem a 3 computadores diferentes com configurações de software e hardware a seguir especificadas.

Computador 1:

OS	Ubuntu 22.04.1 LTS 64 bits
Host	Lenovo IP GAMING 3 15ARH05
CPU	AMD® Ryzen 5 4600H (6) 3.0GHz
Memory	16gGB DDR4 2133MHZ
Solid State Drive	128GB PCIe

Computador 2:

OS	Windows 11 with WSL
Host	DELL 5300 2-in-1
CPU	Intel Core i5-8265 (4) 3.9GHZ
Memory	16GB DDR4 2400MHZ
Solid state drive	1TB PCIe

Computador 3:

OS	Ubuntu 22.04.1 LTS 64 bits
Host	Lenovo Legion 5 15ACH6H
CPU	AMD Ryzen 7 5800H (8) 4.4GHz
Memory	2x8GB DDR4-3200
Solid state drive	1TB PCIe

Após a análise da tabela, podemos verificar que o programa executa muito rapidamente, independentemente da query escolhida (milésimas depois dos 0 segundos). Deste modo, podemos afirmar que o programa é eficiente em termos de custo computacional.

Principais dificuldades

Entre os maiores entraves aquando da realização do projeto estão, sem dúvida:

- A preocupação constante com a preservação do encapsulamento, já que foi algo novo que esta cadeira nos introduziu. Percebemos, no entanto, a sua importância e significado, e temos agora a habilidade de, durante o processo de criação de um módulo ou de uma funçao, perceber se a mesma compromete os dados do programa que devem estar encapsulados, assim como se é possível aceder a estruturas fora dos módulos em que tal se pretende fazer.
- Os comprometimentos de desempenho e memória, já que por vezes nos encontramos em situações em que temos de decidir perante abdicar de tempo para uma implementação menos custosa em termos de memória ou o inverso.
- As dependências de dados, já que correlacionar dados de diferentes módulos nem sempre é intuitivo e torna-se especialmente complicado de resolver quando nos deparamos, por exemplo, com casos de incluídos cíclicos.

Conclusões

Por fim, após a realização do projeto de Laboratórios de Informática III, pudemos adquirir um maior conhecimento acerca da linguagem C, mais concretamente nos termos da salvaguarda de dados, gestão e sua manipulação através de funções da biblioteca *glib2.0* .

As funcionalidades aplicadas permitiram uma gestão eficiente dos dados, garantindo a sua segurança e integridade. Além disso, as ferramentas usadas permitem uma experiência intuitiva e relativamente simples para o usuário.

A utilização do git foi uma peça fundamental para o trabalho, permitindo-nos coordenar na realização das diferentes tarefas através da criação de diferentes branches para cada um dos elementos do grupo.

Para terminar, o excelente trabalho de grupo e empenho de todos permitiu que alcançássemos todos os objetivos propostos em ambas as fases: resolução de todas queries, modo batch e modo interativo, executáveis **programa-principal** e **programa-testes** modularidade e encapsulamento, qualidade do código, makefile e este relatório no prazo previsto e dentro dos requisitos da memória e tempo de execução.