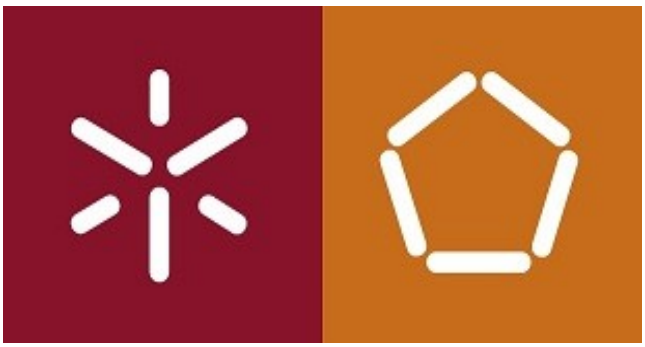


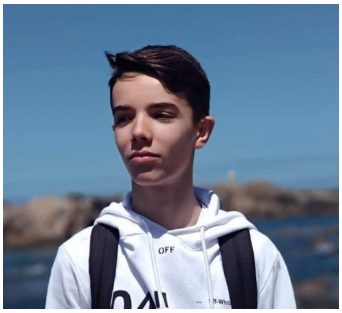
Universidade do Minho

Licenciatura em Engenharia Informática

LI3 - Gestão de dados da Uber

Grupo 11

Ano Letivo 2022/2023



João Lopes (a100594)

Luís Barros (a100693)

João Vale (a100697)

Índice

Introdução

Este relatório tem como objetivo expor o pensamento, processo de desenvolvimento e tomadas de decisão do projeto que foi realizado no âmbito da unidade curricular de Laboratórios de Engenharia Informática III, no segundo semestre do segundo ano, da Licenciatura em Engenharia Informática da Universidade do Minho. Este visa o desenvolvimento de um programa com capacidade de processar, armazenar e organizar os dados fornecidos a partir de ficheiros csv(s), com o objetivo de responder a uma série de questões da forma mais eficiente possível em termos de tempo de execução e memória ocupada.

1.1 Descrição do Problema

São fornecidos três ficheiros csv(s): drivers.csv, rides.csv e users.csv, em que cada linha pode corresponder a informações válidas ou inválidas relacionadas a um condutor, viagem ou utilizador, respetivamente.

O objetivo é processar estes dados e determinar estruturas adequadas, capazes de armazenar estes dados ocupando o menor espaço possível e, simultaneamente, capazes de fornecer uma resposta em tempo útil quando solicitado um pedido por parte de um utilizador.

Importante realçar que a solução deve ter em conta o encapsulamento e modularidade do programa.

1.2 Análise da solução

Ao longo do desenvolvimento do projeto tivemos sempre em vista responder ao problema inicial da melhor forma, debatendo e discutindo de forma construtiva entre os elementos do grupo todas as ideias e alternativas antes de qualquer tomada de decisão. Desta forma, asseguramos que cada alteração e adição no código contribuísse para a melhor resposta possível ao problema, culminado no programa final que apresentamos.

Assim sendo, as estruturas de dados utilizadas foram escolhidas com foco em obter o melhor desempenho em termos de memória e rapidez do programa, sem esquecer a modularidade e encapsulamento do programa.

Módulos

Esta capítulo abordará os métodos, estratégias e ideias adotadas no nosso projeto para solucionar o problema. Discutiremos, portanto, de forma detalhada, a estratégia eleita para recolha dos dados, as estruturas para o armazenamento dos mesmos e o funcionamento de cada querie, bem como o raciocínio por trás das mesmas.

2.1 Catálogo

Estrutura, que contém outras 8 estruturas e ainda 2 inteiros, funcionando como base de dados do programa, guardando todas as informações necessárias à resolução de cada uma das queries quando solicitadas pelo utilizador. O facto de guardarmos os dados todos numa só estrutura, facilita o acesso aos mesmos.

Cada uma das estruturas presentes no catálogo será abordada de seguida.

```
struct catalogos {
	int sizeArrDrivers;
	int sizeArrTrips;

	Users users;
	Drivers arrDrivers;
	Trips arrTrips;
	DriversClass arrDClass;
	UsersDist arrUDist;
	OrdScoreDriversByCity ordScoreDriversByCity;
	DriversUsersSameGender driversUsersSameGender;
	CityOrdByDate cityOrdByDate;
};
```

catalogo.c

2.2 Drivers

Neste módulo encontramos todas as funções e estrutura necessárias ao armazenamento de todos os drivers, sendo que apenas guardamos os drivers válidos. Segue-se a estrutura responsável por guardar a informação de um driver.

```
struct drivers {
        char *name;                                     // Nome
        unsigned char age;                              // Idade
        char gender;                                    // Género
        char classe;                                    // Classe do veículo
        bool status;                                    // Status
        unsigned short int creation;                    // Data de criação da conta
        unsigned short int lastTrip;                    // Data da última viagem
        StackIdTrips trips;                             // Ids das viagens realizadas
};
```

drivers.c

Em relação a cada driver, o ficheiro drivers.csv, fornece-nos informações, sobre o seu id, nome, data de nascimento, género, classe de automóvel que possuí, matrícula, data de criação da conta e o seu status.

Tendo em conta as queries a serem respondidas, constatamos que não seria necessário guardar a informação da matrícula e, que seriam mais vantajoso guardar a idade do driver em substituto da sua data de nascimento por completo, poupando assim memória.

No que toca aos tipos de dados associados a cada informação, estes foram igualmente escolhidos visando ocupar o menor espaço possível, sem nunca comprometer o bom funcionamento do programa. Escolhemos, portanto, para cada parâmetro da estrutura o seguinte tipo:

- Char ou unsigned char, para a idade, género e classe, pois a idade não faria sentido ser superior a 256 (unsigned char varia entre 0 e 256), o género apenas pode tomar dois valores (decidimos associar o valor 1 se for do sexo masculino e 0 se for do feminino) e a classe apenas aceita 3 valores válidos (concordamos em associar o carácter ‘b’ ao modelo basic, ‘g’ ao modelo green e ‘p’ ao modelo premium);

- Bool, para o status, pois este pode tomar apenas dois valores válidos (decidimos associar o valor 1 se for ativo e 0 se for inativo);

- Unsigned short int, para a data de criação da conta e a data da última viagem, pois verificamos que era possível converter cada uma destas informações para um valor que corresponde ao número de dias a partir do início do ano 2000, correspondendo o valor 0 ao dia 01/01/2000. Desta forma, ocupamos apenas 2 bytes por cada informação, sendo bastante mais eficiente do que se guardássemos em int ou string, tornando-se ainda mais relevante essa diferença quando estamos perante ficheiros com milhares de drivers.

- Char *, para o nome, pois não encontramos nenhum outro tipo ou conversão que fosse mais vantajoso em termos de memória.

- StackIdTrips, para guardar os ids das viagens realizadas, mas aprofundaremos mais esta estrutura de seguida.

Contudo, esta estrutura é apenas capaz de guardar as informações de um e, apenas um, driver. Para guardar as informações de todos os drivers, optamos por guardá-los num array de apontadores para structs drivers, sendo que cada posição do array corresponde ao id do driver subtraindo-lhe uma unidade. Desta forma, poderão existir posições do array que se encontrem vazias, não alocando memória para estas e atribuindo-lhes valor NULL, devido a existirem drivers inválidos.

A estrutura que se encontra, portanto, guardada no catálogo, é este array de apontadores para structs drivers.

2.3 Users

Neste módulo encontramos todas as funções e estrutura necessárias ao armazenamento de todos os users, sendo que apenas guardamos os users válidos. Segue-se a estrutura responsável por guardar a informação de um user.

```
struct user {
        char *name;                     // Nome
        char gender;                    // Género
        unsigned char age;              // Idade
        unsigned short int creation;    // Data de criação da conta
        bool status;                            // Status
        unsigned short int lastTrip;    // Data última viagem
        StackIdTrips trips;                     // Ids das viagens realizadas
};
```

users.c

Em relação a cada user, o ficheiro users.csv, fornece-nos informações, sobre o seu username, nome, género, data de nascimento, data de criação da conta, método de pagamento e o seu status.

Novamente, à semelhança do decidido nos drivers, entendemos que seria melhor apenas guardar a idade dos users, em vez da sua data de nascimento por inteiro, pois esta não se verifica ser relevante para responder às queries, ocupando assim menos memória.

Em relação aos tipos de dados escolhidos para cada informação, estes foram escolhidos seguindo a mesma linha de pensamento usada nos drivers, que procura ocupar sempre a menor quantidade de memória possível, sem nunca perder ou comprometer o armazenamento correto dos dados. Deste modo, para melhor compreender a escolha destes, volte para a página anterior onde debatemos o uso de cada tipo por pontos enumerados, aplicando e adaptando adequadamente a informação dos drivers aos users.

Ao contrário dos drivers, a nosso ver, não é viável guardar as informações todas dos users num array como feito anteriormente. Optamos, em alternativa, por usar uma hashTable da biblioteca glib, sendo as keys os usernames dos users e associando a estes uma struct user com o resto das informações necessárias. Desta forma, alocamos apenas a memória necessária, pois apenas reservamos espaço para os users válidos, e torna muito mais eficiente a busca de informações sobre um user quando solicitado.

A estrutura que se encontra, portanto, guardada no catálogo, é a estrutura users que contém a hashTable com todos os users e as suas informações. É a seguinte:

```
struct users {
	GHashTable *users;			// HashTable dos users
};
```

users.c

2.4 Trips

Neste módulo encontramos todas as funções e estrutura necessárias ao armazenamento de todas as viagens, sendo que apenas guardamos as viagens válidas. Segue-se a estrutura responsável por guardar a informação de uma viagem.

```
struct trips {
	unsigned short int date;			// Data
	int driver;				// Id do driver
	char *user;				// Username do user
	char *city;				// Cidade
	unsigned int distance;			// Distancia percorrida
	unsigned char scoreUser;		// Avaliacao user
	unsigned char scoreDriver;		// Avaliacao Driver
	double tip;				// Gorjeta
};
```

rides.c

Em relação a cada viagem, o ficheiro rides.csv, fornece-nos informações, sobre o seu id, data em que ocorreu, driver e user que participaram nela, cidade onde teve lugar, distância percorrida, a avaliação do driver e do user e um comentário relacionado com a mesma.

Mais uma vez, visando economizar memória, decidimos não guardar o comentário, pois este não fornece informação necessária a nenhuma querie.

Em relação aos restantes dados, escolhemos tipos de dados que acreditamos serem os mais adequados aos objetivos do projeto. Escolhemos usar unsigned char para a avaliação do user e do driver, pois esta pode tomar apenas valores inteiros, compreendidos entre 0 e 5. Os motivos para a escolha dos restantes tipos é idêntica à dos drivers e users.

À semelhança dos drivers, escolhemos guardar as viagens todas num array de apontadores para structs trips, sendo que a posição no array corresponde ao id da viagem subtraindo-lhe uma unidade. Isto é possível, pois cada viagem é caracterizada por um id único e, portanto, suficiente para caracterizá-la e distingui-la das restantes. Novamente, tal como nos drivers, apenas guardamos as viagens válidas e, por conseguinte, não alocamos espaço para as restantes, igualando os apontadores destas a NULL.

A estrutura que se encontra, portanto, guardada no catálogo, é este array de apontadores para structs trips.

2.5 DriversExtensions

Este módulo tem como objetivo guardar estruturas de dados e funções que derivem da estrutura inicial do drivers, anteriormente referida, de modo a responder mais eficientemente às queries, com o preço de ocupar mais memória. Neste caso, apenas desenvolvemos uma estrutura auxiliar, com o objetivo de reduzir drasticamente o número de instruções quando chamada a querie 2 múltiplas vezes.

2.5.1 DriversClass

Estrutura guardada no catálogo, criada com o objetivo de evitar a repetição de múltiplas instruções e cálculos, aquando de múltiplas chamadas da querie 2, com o preço de ocupar mais espaço de memória.

```
struct driversClass {
	int id;				// Id do driver
	double class;				// Classificação media
};
```

driversExtensions.c

Sabendo que a querie 2 procura listar os N condutores com maior classificação (apenas condutores ativos) , deduzimos que seria vantajoso ter uma estrutura que guardasse já os condutores ativos ordenados por ordem de classificação média, visto que um variação no valor N nada interferiria na sua ordem, mas sim apenas no número de impressões no ficheiro de resultados. Optamos, então, por criar um array ordenado por ordem decrescente de classificação dos drivers ativos (em caso de empate, ordenamos pela viagem mais recente e, se necessário, pelo id do condutor) e guardar o mesmo no catálogo, sendo este um array de apontadores para structs driversClass. Neste, somente guardamos dois parâmetros, pois o id é suficiente para obter o resto das informações necessárias à correta execução da querie 2, através do acesso ao array com todas as informações dos condutores disponibilizado no catálogo, poupando assim memória.

De salientar, que de modo a facilitar a criação deste array usamos um array auxiliar de apontadores para a estrutura auxiliar, struct driversClassAux, que se distingue da struct driversClass, por guardar também a data da última viagem do driver, facilitando assim a ordenação do array com recurso à função qsort da biblioteca padrão C. Após ordenar, apenas copiamos o conteúdo necessário para o array que vai ser guardado.

```
typedef struct driversClassAux {
	int id;				// Id do driver
	double class;			// Classificação media
	unsigned short int lastTrip;	// Data da ultima viagem
} *DriversClassAux;
```

driversExtensions.c

2.6 UsersExtensions

Módulo semelhante ao DriversExtensions, mas aplicado aos users. Da mesma forma, apenas possuí uma estrutura auxiliar, destinada a atomizar o tempo de execução de múltiplos pedidos da querie 3.

2.6.1 UsersDist

Estrutura guardada no catálogo, seguindo os mesmos princípios da driversClass, aplicada aos users e focando na querie 3.

```
struct usersDist {
        char *username;                         // Username
        int distance;                           // Distancia percorrida
};
```

usersExtensions.c

Sabendo que a querie 3 procura conhecer os top N utilizadores com maior distância viajada, seguimos a mesma linha de pensamento da querie 2 e guardamos um array ordenado por ordem decrescente de distância total percorrida pelos users ativos (em caso de empate, ordenamos pela viagem mais recente e, se necessário pelo username do utilizador por ordem crescente) no catálogo, sendo este um array de apontadores para structs usersDist. Da mesma forma, optamos por esta estratégia, visto que o array ordenado é um invariante, não se alterando com o valor do N. De igual modo, apenas guardamos duas informações, pois a partir do username, que é o identificador de um user, conseguimos ter acesso a todas as outras informações do mesmo, poupando ,mais uma vez, espaço de memória.

Utilizamos, de igual modo, um array auxiliar de apontadores para structs usersDistAux, que tem como objetivo facilitar a ordenação do array pela função qsort da biblioteca padrão C, contendo também informação sobre a data da ultima viagem realizada pelo user. No final deste processo, apenas copiamos a informação necessária para o array que se pretende guardar.

```
typedef struct usersDistAux {
        int distance;                           // Distancia percorrida
        char *username;                         // Username
        unsigned short int lastTrip;            // Data da ultima viagem
} UsersDistAux;
```

usersExtensions.c

2.7 RidesExtensions

Módulo que reúne funções e estruturas, destinadas a melhorarem a eficiência das queries 4, 5, 6, 7, 8 e 9, aquando de repetidas chamadas das mesmas. Ao contrário dos DriversExtensions e UsersExtensions, este contém 3 estruturas auxiliares que serão guardadas no catálogo.

2.6.1 OrdScoreDriversByCity

Estrutura desenvolvida focando na melhor performance da querie 7, evitando a repetição de instruções desnecessárias.

```
typedef struct driversCity {
        int id;                         // Id do driver
        double class;                   // Classificação media do driver numa cidade
} DriversCity;
```

ridesExtensions.c

Estrutura base semelhante à struct driversClass do módulo driversExtensions, contudo, com a diferença que na presente estrutura, guardamos a classificação média de cada condutor por cidade, alterando significativamente o método como organizamos a informação. Optamos por organizar os dados numa hashTable de cidades, que a cada uma corresponde um array de condutores (apenas ativos) ordenados por ordem decrescente da sua classificação média na respetiva cidade (em caso de empate, ordenamos pelo id do condutor, de forma decrescente).

Apostamos nesta estratégia, dado que o cálculo desta estrutura é bastante demorado e trabalhoso, sendo vantajoso guardar a informação, sofrendo apenas o custo destes cálculos 1 vez. De realçar, que apenas é possível tirar partido desta, pois a querie 7 não implica nenhuns cálculos extras, apenas imprimindo as N posições do array da cidade correspondente, tornando esta bastante eficiente em termos de tempo de execução, ao custo de ocupar bastante espaço de memória. Mais uma vez, guardamos o id do condutor de forma a ter acesso às suas informações sem necessidade de as guardar outra vez.

No que toca à criação desta estrutura, auxiliámo-nos de um array de structs driversScoreCityAux, que facilitam no cálculo da avaliação média dos drivers por cidade, guardando o número de viagens do condutor na respetiva cidade e a classificação total, sendo apenas necessária a realização de uma divisão ao fim de percorrer o array das viagens (que se encontra no catálogo). Após este cálculo, apenas ordenamos o array de acordo com as indicações e copiamos a informação necessária para o array que se pretende guardar. Este processo é repetido para cada cidade da hashTable.

```
typedef struct driverScoreCityAux {
        int id;                         // Id do driver
        int NumberTrips;                        // Numero de viagens numa cidade
        double scoreDriver;                             // Score total do driver numa cidade
} DriverScoreCityAux;
```

ridesExtensions.c

O catálogo contém, portanto, a estrutura ordScoreDriversByCity, que, por sua vez, contém a hashTable das cidades com os respetivos arrays de structs driversClass.

```
struct ordScoreDriversByCity {
	GHashTable *cities;			// HashTable das cidades
};
```

ridesExtensions.c

2.6.2 DriversUsersSameGender

Estrutura responsável por guardar os dados de forma a facilitar o trabalho da querie 8.

```
typedef struct arrayOrdGender {
	unsigned short int DriverCreationDate;	// Data da criação da conta do driver
	unsigned short int UserCreationDate;		// Data da criação da conta do
user
	int idDriver;					// Id do driver
	char *usernameUser;				// Username do user
} *ArrayOrdGender;
```

ridesExtensions.c

Visando responder o mais eficientemente possível à querie 8, concordamos em criar uma estrutura composta por dois arrays de structs arrayOrdGender e ainda dois inteiros responsáveis por guardar o número de elementos em cada um. Um contém apenas as viagens realizadas por condutores e users do sexo masculino e o outro apenas por elementos do sexo feminino, estando ambos ordenados de forma às contas dos condutores mais antigos aparecerem primeiro (empatando, ordenamos de forma aos utilizadores mais antigos aparecerem primeiro e ,se necessário, por ordem crescente do id da viagem). Desta forma, aquando de uma chamada da querie 8, já disponibilizamos dos dados e apenas é necessário fazer algumas verificações de forma a verificar a condição dada como argumento. O id do condutor e o username do utilizador servem para aceder às informações relacionadas com os mesmos, sem necessitar de ocupar espaço desnecessário.

Com a finalidade de originar esta estrutura, utilizamos como suporte a struct array e a struct sameGender, sendo que a primeira guarda um array de apontadores para structs sameGender e dois inteiros, correspondentes ao espaço alocado e ao número de elementos guardados. Acreditamos ser útil, visto que não é possível prever que tamanho terão os arrays sem antes percorrer todos os dados, permitindo assim que realoquemos memória quando necessário e, ainda, facilita a ordenação do array com a ajuda da função qsort da biblioteca padrão C, sendo depois apenas necessário copiar do dados necessários para o array que iremos guardar na struct arrayOrdGender e, inserir este no catálogo.

```
typedef struct sameGender {
	int idTrip;                                  // Id da viagem
	int idDriver;                                // Id do driver
	char *usernameUser;                          // Username do user
	unsigned short int DriverCreationDate;       // Data da criação da conta do driver
	unsigned short int UserCreationDate;         // Data da criação da conta do
user
} SameGender;
```

ridesExtensions.c

```
typedef struct array
{
	int sp;                                      // Numero de elementos no array
	int size;                                    // Espaço alocado para o array
	struct sameGender **array;                   // Array de elementos do mesmo sexo
} Array;
```

ridesExtensions.c

2.6.3 CityOrdByDate

Última estrutura armazenada no catálogo, usada pelas queries 4, 5, 6 e 9, para reduzir o seu tempo de execução, com o custo de ocupar mais memória, mas compensando largamente em termos de tempo de execução para múltiplas chamadas destas queries pelo utilizador.

```
typedef struct cityTrips {
	int sp;                                      // Numero de elementos no array
	int *ids;                                    // Array de ids de viagens
} CityTrips;
```

ridesExtensions.c

Analisando o objetivo das 4 queries, concluímos que todas partilhavam uma condição comum, todas beneficiavam ou de ter as viagens ordenadas por data de acontecimento ou de ter as viagens distribuídas por cidade, ou ambas. Desta forma, de modo a encontrar um equilíbrio e conciliar ambas as condições, concordamos em guardar os dados numa hashTable de cidades, que a cada uma corresponde uma struct cityTrips, que por sua vez guarda um array de ids de viagens ordenado da mais antiga para a mais recente, permitindo assim aceder às informações das viagens a partir do id e organizando já as viagens de forma a poupar trabalho às queries mencionadas.

Acelerando o processo de criação desta estrutura, usamos duas estruturas auxiliares, a struct cityTripsAux e a struct idsDatesAux, sendo que a primeira armazena um array de structs idsDatesAux e dois inteiros, relativos ao espaço alocado para o array e ao número de elementos nele guardado. A segunda tem como função guardar o id da viagem e a data em que ocorreu, de forma a facilitar a ordenação do array pela função qsort da biblioteca padrão C. Por fim, apenas necessitamos de copiar os ids das viagens do array ordenado para o array de structs cityTrips, ocupando assim somente o espaço necessário.

```
typedef struct idsDatesAux {
	int id;                                  // Id da viagem
	unsigned short int date;                 // Data da viagem
} IdsDatesAux;
```

ridesExtensions.c

```
typedef struct cityTripsAux {
	int sp;                                  // Numero de elementos no array
	int size;                                // Espaço alocado para o array
	struct idsDatesAux *idsDates;            // Array de viagens
} CityTripsAux;
```

ridesExtensions.c

A estrutura que se encontra no catálogo é a struct cityOrdByDate e esta contém a hashTable das cidades com os arrays de ids de viagens.

```
struct cityOrdByDate
{
	GHashTable *cities;                      // HashTable das cidades
};
```

ridesExtensions.c

2.7 Auxiliares

Este módulo contém diversas funções necessárias nos outros módulos referidos, sendo a maior parte de fácil compreensão.

De salientar, a função pushData e decodeData, que convertem uma data em string para unsigned short int e uma data em unsigned short int para string, respetivamente. Sendo que o modo como funcionam é o inverso uma da outra. A pushData simplesmente conta o número de dias a partir do ano 2000 até à data recebida como argumento, correspondendo, portanto, ao valor 0 a data 01/01/2000. Por sua vez, a decodeData, começa por subtrair 365 ou 366, dependendo do ano ser bissexto ou comum, ao valor recebido até este ser inferior ao número de dias de um ano e, depois simplesmente calcula o dia e o mês com os dias que sobraram.

Execução

Este capítulo explora os principais módulos responsáveis por recolher os dados e devolver as respostas aos inputs recebidos, quer esteja no modo batch ou iterativo.

3.1 Parsing

Módulo que contém a função que lê todos os ficheiros. A função em questão, definida como parsing, recebe um ficheiro como argumento e percorre todas as linhas guardando-as num array de void pointers, sendo que as duas primeiras posições do array guardam inteiros, sendo a primeira o tamanho do espaço alocado e a segunda o número de linhas guardadas. As restantes destinam-se a guardar todas linhas dos ficheiros recebidos, excluindo o cabeçalho. Após percorrer o ficheiro por inteiro e guardado todas as linhas, envia o array que originou para outras funções responsáveis por organizar os dados e armazená-los.

3.2 Outputs

Módulo que abrange as funções responsáveis pelo tratamento das linhas de input, bem como pela execução da respetiva querie e impressão dos resultados nos ficheiros correspondestes.

Usamos uma estrutura auxiliar, struct query, de forma a guardar as informações da linha recebida, possuindo num campo o número da querie e noutro os parâmetros que especificaram a sua execução. Além disso, aproveitado que todas as queries recebem o mesmo número e tipos de argumentos, criamos um array de apontadores para funções possuindo todas as queries. Assim, aquando da chamada da querie N, apenas acedemos à posição do array N-1, executando a querie pedida.

```
struct query {
	short int num;			// numero da querie a ser executada
	char **args;			// argumentos
};
```

outputs.c

Importante realçar que de forma a distinguir se nos encontramos no modo batch ou iterativo, optamos por passar um argumento extra às queries, um booleano, que especifica onde se pretende escrever o resultado. Concordamos, então, que quando o argumento fosse True, estaríamos a executar o programa no modo batch e quando fosse False estaríamos no modo iterativo, permitindo assim imprimir o resultado num ficheiro (FILE *) ou numa janela (WINDOW *), respetivamente.

Em relação às funções responsáveis por executar cada querie, estas operam da seguinte forma.

3.2.1 Querie 1

Opera de uma forma muito simples, avaliando apenas se pretendemos imprimir um condutor ou um utilizador, e caso este exista, vai buscar a informação necessária às estruturas exploradas anteriormente e calcula as restantes, sem nunca violar o encapsulamento através do uso de funções auxiliares disponibilizadas pelos ficheiros onde as estruturas têm origem.

3.2.2 Querie 2 e 3

Limitam-se a ir buscar ao catálogo, respetivamente, o array de condutores ordenado por classificação e o array de utilizadores ordenado por distância total percorrida, percorrendo esse array e imprimindo a informação necessária relativa aos primeiros N elementos do array, usando as estruturas que guardam toda a informação dos condutores e dos users para obter as informações que estas estruturas não contenham, respetivamente.

3.2.3 Querie 4, 5, 6 e 9

pesar de terem objetivos diferentes, funcionam de forma muito semelhante, sempre percorrendo alguma parte da hashTable das cidades, que a cada uma corresponde um array de ids de viagens ordenados da viagem mais antiga para a mais recente.

A mais simples é a querie 4, que se limita a percorrer um array de uma cidade e calcular a média do preço das viagens com o auxílio da estrutura que guarda todas as informações das viagens.

Por sua vez, a querie 6 realiza o mesmo, mas relativamente à média da distância percorrida e limitando-se no tempo, recebendo um intervalo de datas. Em alternativa a percorrer o array todo procurando pela data de início do intervalo, usamos uma espécie de binary search, que torna esta busca muito mais eficiente, percorrendo depois o intervalo entre as duas posições calculadas pelo binary search e calculando a média da distância.

Por fim, a querie 5 e 9, ainda que respondam a perguntas completamente diferentes, usam o mesmo método para calcular o intervalo de valores relevante, ambas usando a estratégia semelhante ao binary search da querie 6 mas aplicada a todas as cidades. Enquanto que a querie 5 limita-se a percorrer o intervalo desejado de todas as cidades e a calcular a média do preço das viagens, a querie 9 copia para um array auxiliar os ids de todas as viagens que se encontram no intervalo recebido e em que o utilizador deu gorjeta de forma a poder ordenar os dados. Esta utiliza a função qsort da biblioteca padrão C para fazê-lo e organiza-os por ordem decrescente de distância percorrida (em caso de empate, as viagens mais recentes aparecem primeiro e, se necessário, ordena-os por ordem decrescente do id da viagem). Depois, simplesmente imprime o array todo no local pretendido.

3.2.4 Querie 7

Funciona de forma semelhante à querie 2, mas usa a estrutura do catálogo que guarda a hashTable das cidades, que a cada uma corresponde um array de condutores ordenado por classificação, limitando-se, depois, apenas a ir buscar o array da cidade recebida como argumento e a imprimir as informações necessárias das primeiras N viagens.

3.2.5 Querie 8

Bastante simples, dependendo do género recebido como argumento vai buscar ao catálogo o array correspondente, de elementos do mesmo sexo ordenado por data de criação da conta. Depois, limita-se a imprimir os dados necessários das viagens em que a conta do condutor e do utilizador têm mais e X anos, com auxílio das estruturas que guardam todas as informações dos condutores e utilizadores.

3.3 Iterativo

Módulo referente às funções responsáveis pelo modo iterativo. Quando o programa se encontra neste estado, o programa é executado sem argumentos adicionais e, assim, os resultados passam a ser apresentados no ecrã em vez de num ficheiro de texto. Com este objetivo, recorremos à biblioteca <ncurses.h> para construir a interface gráfica do programa, juntamente com a biblioteca <locale.h> de forma a ser possível imprimir no ecrã wide chars, uma vez que, sem esta, não conseguiríamos imprimir caracteres como “ã”, “í”, “à”, entre outros.

O utilizador quando escolher executar o programa neste modo, o programa vai lhe pedir para inserir o caminho para os ficheiros de dados e, enquanto este for inválido, o programa imprime uma mensagem de erro, que desaparece quando o utilizador pressiona uma key, voltando a pedir novamente o caminho para os ficheiros csv(s). Quando este for válido, o programa gera as estruturas de dados respetivas, igual a se nos encontrássemos no modo batch e, de seguida, apresentamos o menu principal no qual o utilizador pode navegar utilizando as setas verticais ou “W” e “S” para selecionar a querie que pretende executar ou terminar o programa. Selecionada uma querie, é pedido ao utilizador para inserir os argumentos da mesma, verificando sempre se o input tem o formato desejado. Dito isto o programa executa a querie e imprime no ecrã os resultados, sendo que se não couberem todos recorremos a paginação, permitindo que o utilizador alterne entre páginas através das setas horizontais ou “A” e “D”, permitindo visualizar todos os resultados. Para voltar ao menu principal, o utilizador simplesmte necessita de pressiona a tecla Enter.

Programas

Este capítulo faz referencia aos dois programas que é possível executar, o programa principal e o programa de testes.

4.2 Programa Principal

Programa que pode ser executado de duas formas diferentes consoante o número de argumentos recebidos, sendo estes o modo batch e o modo iterativo, tendo o segundo sido já abordado.

Em relação ao modo batch, ao executar o programa o utilizador passa como primeiro argumento o caminho para os ficheiros de dados e como segundo argumento um ficheiro com os inputs a executar. Neste estado, os resultados são todos guardados em ficheiros, numa pasta criada aquando o *MakeFile*, tendo todos o nome "command%d_output.txt", onde %d representa o número da linha do ficheiro de input.

Em ambos os modos, ao fim de executar todas as linhas do ficheiro de input (modo batch) ou de o utilizador escolher terminar o programa (modo iterativo), libertamos a memória reservada para todas as estruturas de dados.

4.2 Programa Testes

Este programa tem uma base muito semelhante à do programa principal, com a diferença que avalia simultaneamente a veracidade e exatidão dos nossos resultados para o input de 500 linhas dos testes automáticos quando fornecido como argumento o dataset grande sem dados inválidos.

Com o intuito de armazenar todos dados necessários aos testes, recorremos a um *array* com 9 índices (um para cada querie), em que cada um guarda um *array* dinâmico de *structs testes_aux* (um para cada input da sua respetiva querie), sendo esta responsável por guardar o número da linha do input, de forma a identificá-la, o tempo que a querie demorou a executar, um *bool que determina a correta execução da querie e* uma *string* para informar, caso necessário, o tipo de erro.

```
typedef struct testes_aux {
        short int number_input;
        double time;
        int lines;
        bool valid;                             // True = sem erros ; False = tem erros
        char *errors_info;                              // Guarda informação sobre os erros
} Testes_aux;
```

automatic_tests.c

Os tipos de erros que podem ocorrer são os seguintes:

- No caso de o output ser diferente, informa a primeira linha onde o erro ocorre;
- Caso um dos ficheiros esteja vazio e não seja suposto, ou vice-versa, informa qual dos dois ocorreu e o que seria suposto ocorrer;
- Se o tempo exceder os 10 segundos, avisa que ultrapassou o limite de tempo;
- Por fim, caso não haja problemas, simplesmente não passa informação nenhuma.

De forma a determinar os erros, o programa percorre todos os caracteres dos nossos outputs e do output correto, identificando se houver algum diferente e guardando a respetiva linha.

4.2.1 Estatísticas de Execução

O programa de testes para além de avaliar a correta execução do programa face à leitura de diferentes inputs, também calcula o tempo de execução de cada querie para cada input, apresentando no fim uma aproximação do tempo de execução de cada uma. Esta aproximação é feita através de uma média pesada, visto que existem queries que têm número de linhas de output variáveis, sendo assim a aproximação mais exata do que se utilizássemos uma média normal.

Seguem-se os resultados dos tempos de execução em diferentes máquinas.

Querie	Input[1]	Output[2]	Tempo[3]		
			Máquina 1	Máquina 2	Máquina 3
1	290	276	0.000044	0.000042	0.000027
2	10	13602	0.002221	0.001958	0.001219
3	10	27171	0.003501	0.003471	0.002334
4	25	20	0.237446	0.003471	0.153062
5	50	45	0.091628	0.089197	0.003677
6	50	45	0.007067	0.006601	0.055998
7	30	27187	0.001456	0.001345	0.000793
8	5	18621	0.013700	0.009180	0.007166
9	30	27452	0.009269	0.012048	0.004762

1: Número de inputs que tem a respetiva querie.
2: Número de linhas de output que cada querie gerou no total.
3: Média pesada do tempo de execução por querie.

Máquina	Marca	Distribuição	OS	CPU	Memória
1	HP	Pop!_OS	Linux	i7 8th 1,8GHz 8 cores	16GB RAM 256GB SSD
2	Lenovo	Ubuntu	Linux	i7 8th 1,8GHz 8 cores	8GB RAM 512GB SSD
3	Asus	Manjaro	Linux	i7 10th 2,2GHz 16 cores	16GB RAM 512GB SSD

Conclusão

Tendo em conta os desafios propostos e o resultado final, concluímos que alcançamos os objetivos traçados com um bom desempenho global. Todavia, sentimos que podíamos ter implementado funcionalidades extra de modo a enriquecer o nosso programa.

Em suma, é indiscutível que conseguimos uma melhoria bastante significativa face à primeira fase do projeto e, por isso, estamos orgulhosos do trabalho realizado e satisfeitos com o projeto apresentado.