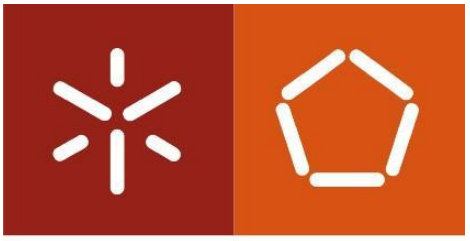

Universidade do Minho

Relatório Projeto LI3

FASE 1

GRUPO 99

a100646, Diogo Ribeiro Vassalo de Abreu, DiogoAbreu19

a100610, José Afonso Lopes Correia, Ze-Limao

a100665, Tiago Nuno Magalhães Teixeira, T-yago

Índice

Introdução

O presente relatório tem como objetivo apresentar a primeira 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), porém nesta primeira fase só trataremos do modo Batch.

Neste modo, 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".

Desenvolvimento

Estruturamento do projeto

Foi necessário averiguar quais seriam as melhores estruturas de dados para conseguirmos resolver os nossos problemas, no que tocava a armazenamento, gestão de dados e velocidade de execução.

Users e Drivers

Definimos estruturas para ambos *users* e *drivers*. Isto é conveniente, na medida que nos permite guardar as informações de cada *user* e *driver* de forma organizada, para além de tornar mais fácil a obtenção de um certo campo de qualquer um dos dois. Ultimamente, isto resulta ainda num código mais legível e num encapsulamento mais conciso.

Optamos, assim, por adicionar um campo na *struct* para cada caraterística presente nos ficheiros *.csv* dados. Para além disso, acrescentamos ainda, dentro da *struct* de cada um, tipos que são úteis para a resposta das *queries*, por motivos que serão explicados neste mesmo capítulo.

Na fase de armazenar os dados presentes nas structs, tentamos otimizar ao máximo a memória, guardando a informação no mínimo de espaço possível. Um exemplo disso foi a criação da função **convert_to_day** que converte uma data (inicialmente lida em string) num unsigned short int (que ocupa um espaço mínimo de até 16 bits), que posteriormente ainda facilita a comparação de datas para, por exemplo, verificar qual delas é mais recente.

Optamos por usar tabelas de Hash para os *users* e para os *drivers* devido à possibilidade de utilizar o *username* como chave da *hashtable* dos *users* e o *id* como chave para a *hashtable* dos *drivers*. Quanto aos valores associados a cada *key* da *hashtable*, definimos em ambas as tabelas de Hash um apontador para as *structs* do respetivo ficheiro a ser tratado.

Assim, utilizamos uma combinação das funções *getline* e *strsep* incluídas na biblioteca *"string.h"* para ler *token* a *token* a informação de cada linha e preencher o seu respetivo campo na *struct* do *user* ou *driver* em questão logo após a leitura desse mesmo dado ser feita.

O que mais nos atraiu nesta *feature* das tabelas foi a possibilidade de utilizar a função *hash* para obter o valor (neste caso o apontador para o *struct*) associado a uma chave

qualquer evitando assim ter que percorrer todos os dados (no pior caso) e conseguindo um tempo linear para cada *lookup* realizado.

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 dado.

Rides

O tratamento das *rides* não foi o mesmo dos tipos anteriores.

Nesta primeira fase realizamos as três primeiras *queries* do enunciado fornecido pelos professores sem guardar na memória toda a data associada às *rides*.

Ao invés disso, optamos por ler linha a linha o ficheiro das *rides* e imediatamente calcular os valores necessários para as *queries* à medida que percorremos o ficheiro, respeitando sempre o encapsulamento. Isto dá-nos a possibilidade de libertar a memória associada a cada *ride* no final da sua leitura, pelo que evitamos ter um momento do programa em que está alocada toda a memória necessária para armazenar a informação das *rides* na sua totalidade.

Isto mostrou-se bastante eficiente não só no que toca à gestão da memória mas também porque nos permitiu realizar apenas uma leitura completa do ficheiro *rides.csv*, que como sabemos, é o mais extenso que nos foi fornecido.

Adicionalmente, esta implementação permite executar as três *queries* sem a necessidade de consultar mais do que uma vez qualquer dado presente neste mesmo ficheiro.

Parsing

A função de *parsing* tem como objetivo ordenar a execução das funções necessárias ao funcionamento do programa, de modo a que depois, ao executar cada *query*, tenha toda a informação necessária quer nos catálogos, quer nas estatísticas (onde tudo o que é relacionável entre os ficheiros *.csv* se encontra), de modo a que seja possível aceder a tudo o que seja necessário para a resolução de cada *query* mais eficientemente, melhorando assim a *performance* do programa.

A função parsing ordena a execução do programa da seguinte forma:

1. Cria os catálogos dos users e drivers
2. Junta a informação relacionável com os outros ficheiros nos catálogos dos users e drivers
3. Lê o ficheiro de input e de acordo com o primeiro digito de cada linha, aponta para a query pedida

Resolução das Queries

Query1

Para a resolução da *query1* aproveitamos o facto de termos inicialmente criado uma *hashtable* para *users* e *drivers* para utilizar a função **g_hash_table_lookup** incluída da biblioteca **glib.h**. Isto foi particularmente vantajoso, já que o input que recebemos para a realização desta *query* pelo *parser* é precisamente a chave de cada tabela criada por nós no início do programa, pelo que o podemos utilizar como argumento para a função de **lookup** diretamente, resultando isto num código mais conciso e ajudando na legibilidade do mesmo.

Assim sendo, responder aos campos nome e género do *output* implica apenas a chamada da função **lookup**. Quanto aos campos ”avaliação média” e “número de viagens”, decidimos adicionar dois novos campos nas structs dos dois tipos de dados: “avaliação total” e “número de viagens” que são modificados à medida que o ficheiro *rides.csv* é lido. Assim, a resposta para o número de viagens está já calculada no final da leitura deste ficheiro, enquanto que a avaliação média é calculada pelo quociente dos valores avaliação total e número de viagens.

Query2

Para a resolução da *query2* optamos por criar um *array de structs*, de forma a conseguirmos assim aproveitar as *structs* já criadas e de fácil acesso por *id* que se encontram na *hashtable*.

Com isto em mente, começamos por obter todos as *structs* do tipo *driver* presente na *HashTable*. Para evitar um maior gasto de memória, e já que nos eram apenas pedidas três informações sobre os *drivers*, achamos útil a criação de uma nova estrutura para esta *query*, que apenas contém os campos necessários para a execução da mesma e que é preenchida pela *structs* que já estão, nesta altura, na tabela.

Após isso, utilizamos o algoritmo **qsort** da linguagem para ordenar os *structs* criados para o propósito desta *query* criando condições que resultam na ordenação desejada.

Query3

A resolução da *query3* foi bastante semelhante à *query2*. Tal como anteriormente, fizemos fetch aos dados que se encontram na hash, 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**.

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 à modularidade, para não ter tudo num só ficheiro, e também de forma a resolver vários problemas em simultâneo, dividimos o os ficheiros de código-fonte da seguinte forma:

main.c ;
parser.c;
users.c; drivers.c; rides.c;
query1.c; query2.c; query3.c;
datas.c(função que converte uma data para um número).

Para manter a segurança e integridade dos nossos dados, criamos funções de encapsulamento, para que num certo ficheiro pudéssemos utilizar estruturas de dados que não pertencessem a este ficheiro, desta maneira um ficheiro só tem acesso a coisas que foram declaradas no mesmo e garantimos que não existia nenhuma exposição a qualquer estrutura que não fosse a do próprio ficheiro na qual estava inserida ou um apontador para algum dado cuja acesso devesse ser interdito ao utilizador. Para conseguir fazer isto os ficheiros estão interligados pelos *headers files*.

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.

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 o ficheiro executável, que necessita 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 ficheiros objetos usados na criação do executável como também os ficheiros de output.

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.

Conclusões

Por fim, após a realização desta primeira fase 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* .

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 de nós.

A maior dificuldade sentida após a realização desta primeira fase foi a gestão de memória. A libertação de memória após a alocação da mesma foi uma autêntica dor de cabeça em relação aos catálogos.

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

Output	CPU Time (s)
1 (Query 1)	0.001039
2 (Query 1)	0.000729
3 (Query 1)	0.000847
4 (Query 1)	0.000968
5 (Query 1)	0.000800
6 (Query 1)	0.000785
7 (Query 2)	0.002108
8 (Query 3)	0.045984

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

Para terminar, o excelente trabalho de grupo e empenho de todos permitiu que alcançássemos o objetivo para a primeira fase: resolução de 3 queries (no mínimo), modularidade e encapsulamento, qualidade do código, makefile e este relatório.