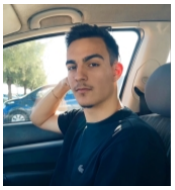
Relatório da Segunda Fase

Laboratórios de Informática III

Maurício Pereira
A95338



Gabriel Silva
A97363

Grupo 7
https://github.com/dium-li3/grupo-07

Departamento de Informática
Universidade do Minho
Janeiro 2023

Índice

1 Resumo

Este relatorio diz respeito a ultima fase (II) do projeto de Laboratorios de Informatica III e tem como objetivo mostrar as metas e formas de resoluçao dos exercicios propostos e tambem as dificuldades encontradas nestes.

Em comparação com a fase anterior anterior podemos dizer que esta foi bastante mais complexa de se realizar e consequentemente mais motivadora, porém foram encontrados alguns problemas na mesma. Explorou a nossa criatividade e capacidade de superar dificuldades o que nos levou a descobrir aspetos da linguagem C e de programação no geral que nunca imaginavamos.

2 Avanços relativos à Fase I

Após a apresentação da I fase, a nossa realidade relativamente ao estado do nosso projeto alterou. Como futuros profissionais vimos este acontecimento como um fator muito importante para o desenvolvimento constante do nosso trabalho, acreditamos que o progresso está visível e facilmente comprovável.

Começamos então por garantir o encapsulamento e modelação de todos os dados, visto que após discussões com os Docentes concluímos facilmente que este era o fator mais decisivo. Seguindo este objetivo, alteramos a arquitetura inicial da aplicação relativamente à criação das *hashtables*, que se sustentava num ficheiro que continha as três principais estruturas deste projeto, passando agora a ser composta por um ficheiro para cada uma delas.

Corrigimos também os problemas que tinham sido apontados na nossa Makefile.

Um outro avanço relativamente à primeira fase de entrega centra-se no encapsulamento das *hashtables* visto que estas eram tornadas acessíveis em outros ficheiros que não o da sua criação através do uso da primitiva ***extern*** o que compromete o encapsulamento das mesmas. Face a este problema as hashtables passaram as ser criadas no ficheiro *interpreter.c* onde são posteriormente acessadas nos diferentes ficheiros sendo passadas como argumentos das funcões que as invocam.

Adotamos também uma nova forma de calcular campos que são frequentemente necessários/pedidos. Fazemos então o parsing dos users e dos drivers e posteriormente ao longo do parse das rides guardamos no user e driver respetivo dados como a avaliação total, o total gasto/auferido, a distância total, o número de viagens, a avaliação média, e a última ride realizada. Anteriormente a esta implementação, a informação pedida por cada querie era calculada ao longo da realização da mesma, deste modo apenas precisamos de a pedir. A implementação desta estratégia contribuiu para um melhor desempenho do programa reduzindo o tempo de execução de queries mais simples como a querie 1 e 4, por exemplo, visto que todos os dados referentes às mesmas já se encontram cálculados e guardados de antemão.

3 Modularização e reutilização de código

Modularização e reutilização de código são conceitos fundamentais no desenvolvimento de software. A modularização consiste em dividir o código em módulos separados, cada um com uma função específica e bem definida. Isso torna o código mais fácil de ser entendido, mantido e testado. Além disso, permite a reutilização de módulos em projetos futuros, economizando tempo e esforço.

A reutilização de código se refere ao uso de módulos já existentes em vez de escrever novamente o mesmo código. Isso não só poupa tempo, mas também garante que o código já testado e confiável seja utilizado. Além disso, a reutilização de código pode ajudar a manter a consistência em projetos maiores e mais complexos.

Ao longo do desenvolvimento deste projeto foi sempre dado o devido cuidado à manutenção da modularização e reutilização do código havendo ainda espaço para possíveis melhorias no futuro, como por exemplo, a criação de um módulo específico ao parsing de dados, contrário ao utilizado no nosso programa onde o parsing de cada ficheiro *csv* é realizado no ficheiro da estrutura respetiva.

4 Encapsulamento e abstração

Como referido nos *Avanços relativos à Fase I*, tomamos em consideração as observações dos Docentes, pelo que modificamos no seu todo o método de encapsulamento e abstração dos dados.

Primeiramente tínhamos o parsing dos catálogos concetrado num só ficheiro, tal como a criação das tabelas. Acedíamos às tabelas e ao conteúdo desejado de forma direta, ou seja, não tínhamos controlo algum sobre os dados que estavam a ser manipulados.

Tivemos então de criar os ficheiros *users.c*, *drivers.c* e *rides.c* para parsing dos dados e posterior criação das *hashtables*. Fizemos funções *get* e *set*, que nos permitem aceder e alterar, respetivamente, os dados desejados em cada *value* das tabelas. Deste modo, temos total controlo sobre aquilo que pode ou não ser acedido a partir de uma determinada zona.

5 Estruturas de dados utilizadas

Devido ao facto de ser livre o uso da biblioteca *glib2* utilizamos as *HashTables* e as *Listas* que esta biblioteca nos proporciona. Optamos por esta solução, e não pela criação das nossas próprias tabelas, pois sabiamos de antemão que estas estrututas eram bastante mais eficientes (tanto a procura por *keys*, como a inserção de novos elementos).

6 Exercícios e Resolução

6.1 Parsing de dados/Validação de campos

O parsing de dados evoluiu um pouco ainda que a base estabelecida no final da primeira fase persistisse. O parsing, que estava até então confinado ao ficheiro *main.c* onde eram criados todos os catálogos passou agora a estar disperso pelos diferentes ficheiros correspondentes a cada catálogo. Isto veio com a necessidade de uma melhoria do encapsulamento e modularização do código até então desenvolvido. Desta forma o parsing de cada ficheiro é efetuado num ficheiro próprio e, aquando do parsing do ficheiro das *rides*, passaram a ser também calculados alguns valores importantes às queries. A validação dos campos não ficou implementada a 100% visto que as validações dos campos ***car class*** da estrutura *driver* e ***Acc status*** da estrutura *user* inválidos acabam por passar na validação ainda que erroneamente.

6.2 Querie 1

Procuramos o *id/username* fornecido nas nossas tabelas e, como já temos todos os dados necessários guardados nas estruturas, basta fazermos os acessos aos dados de interesse e imprimi-los para o ficheiro de output.

6.3 Querie 2

Primeiramente passavamos os dados da *drivers table* para um array auxiliar, sendo a key o *id* do driver e o value seria o *driver*. Posteriormente ordenamos o array pelos parâmetros pedidos, mas não estávamos a ter sucesso. Utilizamos a função *qsort* para este efeito.

Vimo-nos então obrigados a mudar de estratégia. Neste momento estamos a criar uma nova tabela com os *drivers* ordenados de forma correta (teoricamente, pois mesmo assim a ordenação não está a correr como o esperado).

6.4 Querie 3

Alvo de uma estratégia similiar à querie 2, assim como todas as queries TopN (no enunciado dao outro nome). Este método acatava com *leaks* de memória que alcançavam os 19MB de dados. Face a este problema optamos, posteriormente pela conversão das entradas da *hashtable users* numa *GList*, à qual seria feita a ordenação dos valores. Nesta fase a querie já produzia um *output*, no entanto, os valores apresentados não eram os corretos, pelo que a ordenação não estava a ser realizada corretamente. Dito isto, numa última tentativa de corrigir esta querie alterada a abordagem passando a transformar a hashtable numa hashtable iterativa, ainda que, sem resultado visto que o problema pareceu persistir ao nível da função de comparação utilizada na ordenação.

6.5 Querie 4

Iteramos a ***rides table*** com a função *g_hash_table_iter_init*, da biblioteca *glib*, e a cada iteração é comparada a cidade da *ride* em questão com a cidade objetivo: se forém iguais, é calculado o seu preço.

Neste momento precisamos de verificar qual a classe do veículo utilizado nesta viagem. Assim, basta aceder ao *driver* pretendido através do seu ***id***, que o podemos obter pela viagem e verificar a classe do seu carro. Para classes e distâncias de viagens diferentes teremos custos diferentes. O preço total e o número de viagens são guardados e atualizados a cada iteração e, deste modo, é possível calcular o preço médio das viagens na cidade objetivo.

6.6 Querie 5

Primeiramente, é efetuado o parsing das datas atribuindo cada valor a um inteiro. Em seguida, é utilizada a função *g_hash_table_iter_init*, da biblioteca *glib*. A cada iteração a data de cada *ride* é comparada com as datas fornecidas no input da querie de modo a verificar-se se esta se encontra no intervalo de datas pretendido e no caso deste último é feito o acesso ao *driver*, que corresponde ao ***driver id*** da *ride* acedida previamente, que se encontra na *hashtable* ***drivers table***.

Tendo acesso ao *driver* é acedido o campo ***car class*** e dependendo do tipo de carro é efetuado o cálculo do custo dessa viagem e guardado o valor num *int* ***preco viagem***. É também incrementada a varíavel *int* ***n viagens*** que representa o número de viagens que se encontram no intervalo de tempo estipulado pelas datas recebidas na querie.

Dado como terminadas a pesquisa das keys da ***rides table*** é calculado o custo médio das viagens e guardado na varíavel *int* ***preco medio*** sendo, por último, impresso este valor no ficheiro para o qual o caminho está contido, como dito anteriormente, no *pointer* ***filepointer***.

6.7 Querie 6

O processo de execução desta querie é, essencialmente, o mesmo da querie5, com a pequena excessão de que, aquando da validação que permite averiguar se a data de uma *ride* se encontra no intervalo de tempo fornecido na querie é também verificada em simultâneo se a cidade onde a *ride* em questão foi providenciada é a mesma do que a cidade que fora fornecida previamente como argumento da querie. Após esta verificação o processo de execução toma um rumo similar ao da querie5 novamente.

6.8 Querie 7

A estratégia utilizada para este exercício é muito similar à da *querie 2*, a pequena diferença é o uso de uma tabela auxiliar para guardar todas os *drivers* que satisfaçam os requisitos.

A partir desta tabela criamos uma nova tabela ordenada (teoricamente), consoante os parâmetros pedidos, com as *rides* desejadas.

6.9 Querie 8

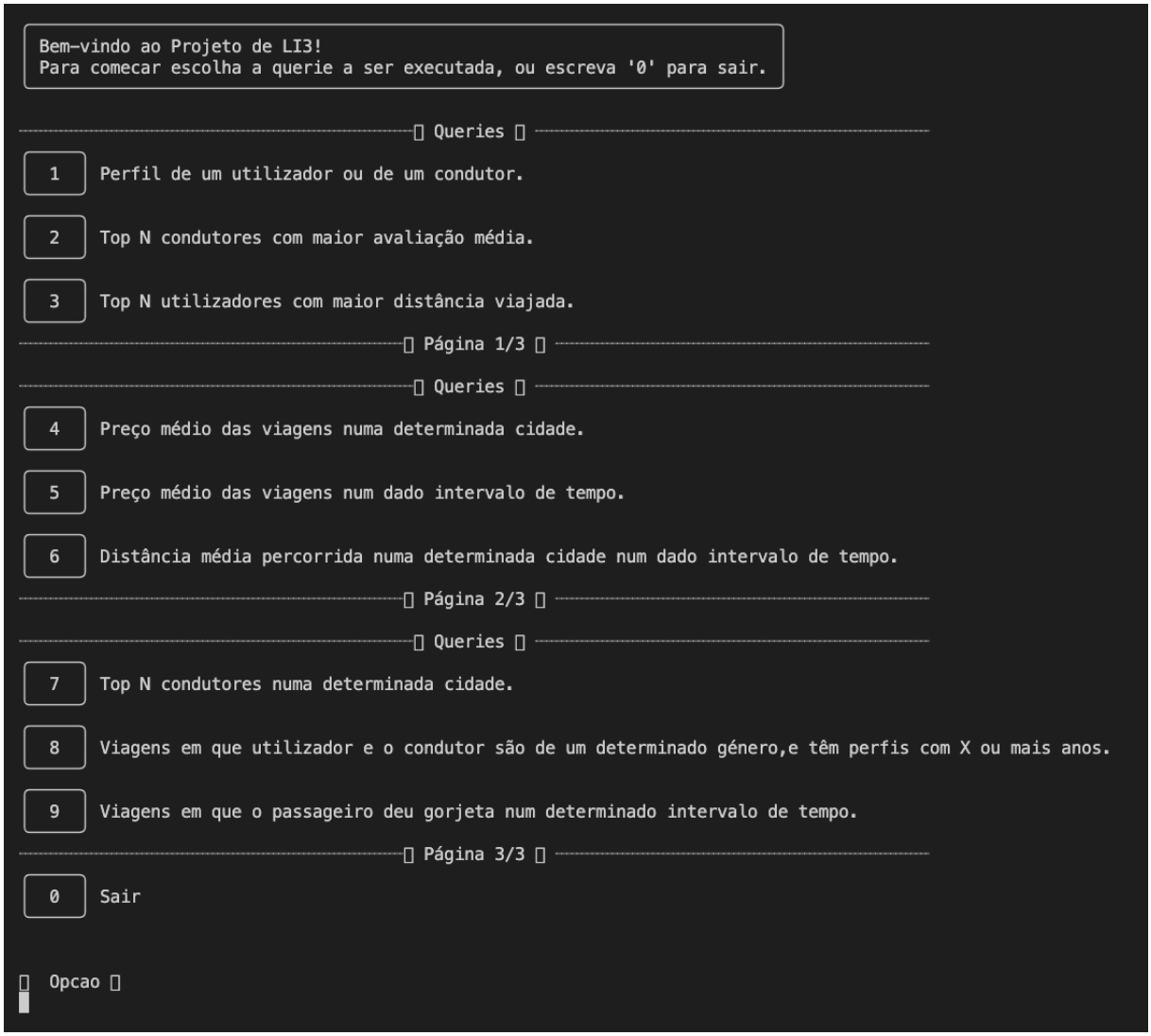
Estrutura bastante similar à *querie 7*, mas guardamos as *rides* desejadas numa nova tabela para depois ser possível ordenar. Visto que tivemos um problema com as funções de ordenação e, como o nosso tempo já não era muito, optamos por dar prioridade a outros pontos do projeto.

6.10 Querie 9

A implementação elaborada para esta querie é muito similar à querie 5, cada *ride* que verifique a condição estabelecida, onde a ***data*** da ride se encontra no intervalo estabelecido pelo *input* da querie e o cliente deu gorjeta, é adicionada à tabela auxiliar.

Esta tabela posteriormente dá origem a uma outra ordenada (teoricamente), com as *rides* desejadas. Visto que tivemos um problema com as funções de ordenação e, como o nosso tempo já não era muito, optamos por dar prioridade a outros pontos do projeto.

7 Interface


Bem-vindo ao Projeto de LI3!
Para comecar escolha a querie a ser executada, ou escreva '0' para sair.
Queries
1 Perfil de um utilizador ou de um condutor.
2 Top N condutores com maior avaliação média.
3 Top N utilizadores com maior distância viajada.
Página 1/3
Queries
4 Preço médio das viagens numa determinada cidade.
5 Preço médio das viagens num dado intervalo de tempo.
6 Distância média percorrida numa determinada cidade num dado intervalo de tempo.
Página 2/3
Queries
7 Top N condutores numa determinada cidade.
8 Viagens em que utilizador e o condutor são de um determinado género,e têm perfis com X ou mais anos.
9 Viagens em que o passageiro deu gorjeta num determinado intervalo de tempo.
Página 3/3
0 Sair
Opcao

8 Conclusão

Finalizando, apesar das adversidades encontradas aquando da realização do projeto, tendo em conta o volume de trabalhos que os integrantes desta equipa se encontravam envolvidos, consideramos que conseguimos atingir a maior parte dos objetivos propostos.

Ainda assim um dos objetivos cruciais para a resolução dos exercícios de top N não ficou totalmente concluído, ficando a faltar a ordenação dos *outputs*. As funções de ordenação estão implementadas e sofreram uma exausta análise não tendo sido, no entanto, encontrado o erro, apesar das diferentes estratégias adotadas - entre as quais, passar todos os dados das tabelas para um array auxiliar e posteriormente ordenar utilizando a função *qsort*.

O processo de aprendizagem sobre a utilização da biblioteca GLib é demorado e algo complexo, assim como a utilização das listas e tabelas de hash da mesma.