Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática
Laboratórios de Informática III 2022/23

Relatório Fase 2

(Grupo 73)

Diogo Filipe Oliveira da Silva (a97941)

Rafael Conde Peixoto (a96807)

Índice

1 Introdução

Nesta segunda fase do trabalho melhoramos o trabalho desenvolvido na primeira fase.

Decidimos adicionar duas novas coleções e alterar a estrutura de dados de uma já existente com o objetivo de melhorar o desempenho. Implementamos também 8 das 9 queries pedidas no enunciado.

O trabalho prático pode ser resumido em duas partes:

- Modo Bach – Neste modo carrega-se três ficheiros de dados *.csv's* para memória, para futuramente a informação ser processada (parsing), com o objetivo de responder a diversas questões. O output das instruções é retornado na forma *.txt* dentro da pasta *Resultados.*

- Modo Interativo – O modo interativo é um mecanismo de interação gráfica onde neste modo pede-se ao utilizador para indicar o path para os *csv's* onde será realizado o parsing dos mesmos e em seguida redirecionado para o menu principal.

  Já no menu principal é dada ao utilizador uma lista de instruções(queries) disponíveis e o output é apresentado no terminal.

Em ambos os modos o nosso objetivo principal é respeitar as regras de encapsulamento e modularidade ao máximo.

2 Modo Interativo

Quando o utilizador inicia o programa é lhe apresentado um menu com todas as funcionalidades(queries) disponíveis onde o mesmo deve escolher o valor numérico que está definido para a instrução pretendida.

Demonstração:

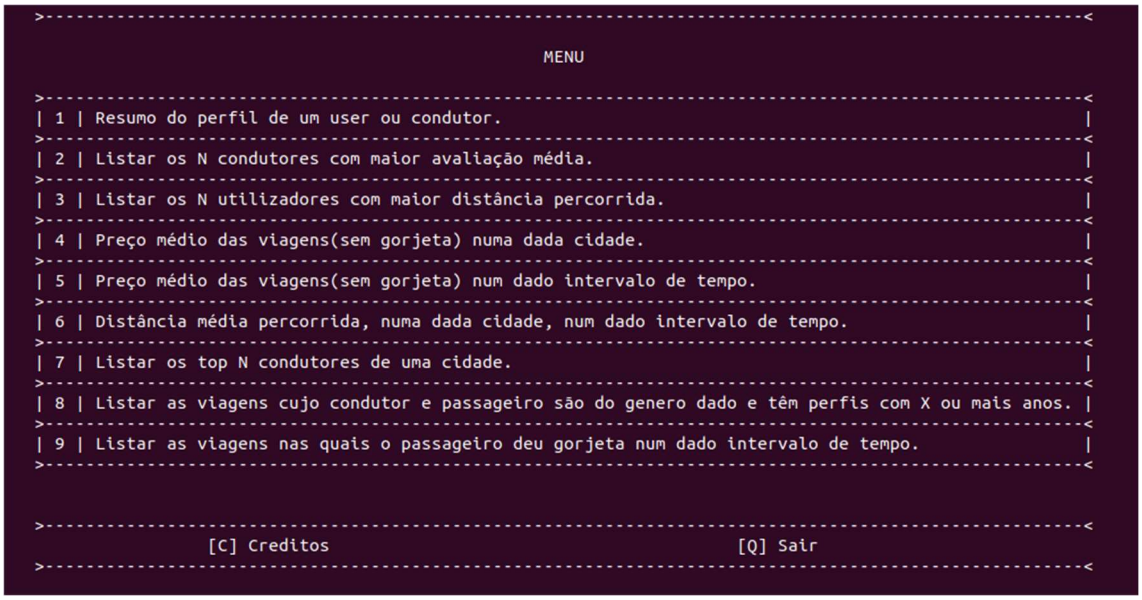

Figura 1 - Menu principal

Após a escolha da opção pretendida é pedido ao utilizador para inserir os argumentos respetivos a cada instrução. Nas queries onde o output não é um valor numérico/uma linha o modo interativo apresenta um sistema de paginação onde é possível avançar(P), retroceder(A) ou escolher a página (nº da página), assim como uma opção para voltar(V) ao menu principal.

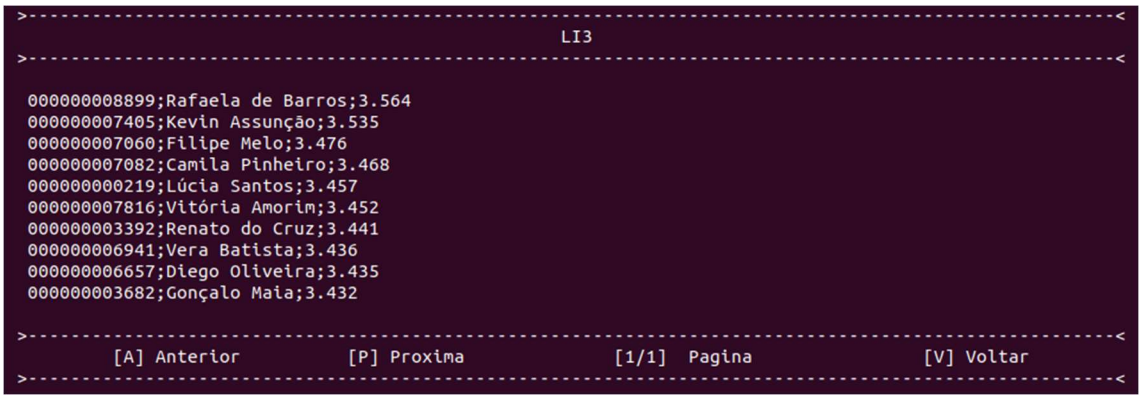

Figura 2 - Paginação

3 Estruturas de dados

Nesta segunda fase decidimos criar duas novas coleções StatusDriver e StatusUser onde foram implementadas duas hashtables. Decidimos usar hashtables para estas coleções devido ao seu tempo constante Ơ(1). A hashtable StatusDriver é uma hashtable *open addressing* com tamanho igual ao array dos drivers enquanto a StatusUser é *closed addressing/chaining* com tamanho de 1.000.000.

Alteramos também a coleção rides de uma hashtable para um array com tamanho inicial de 5.000.000 ordenado pela data das viagens onde usamos procura binaria devido ao facto do seu tempo de procura ser mais eficiente, Ơ(log n) no pior caso. Sentimos a necessidade desta alteração em consequência da inutilidade de estarem ordenadas pelo id uma vez que nenhuma das queries pedia o id de viagem, mas sim intervalos de tempo.

Nas coleções do drivers e users decidimos manter as hashtables, no caso da coleção drivers as colisões são inexistentes devido a não existência de drivers(condutores) com o mesmo id então usamos uma hashtable *open addressing* com tamanho inicial de 10.000. Já no caso do users existem colisões entre os usernames então optamos por uma hashtable *closed addressing/chaining* com tamanho fixo de 1.000.000.

Decidimos definir nós mesmos as hashtables/arrays dinâmicos para uma melhor compreensão do seu funcionamento, ao contrário da primeira fase onde usamos arrays estáticos.

4 Modularidade e Encapsulamento

Para respeitar os princípios da modularidade e encapsulamento definimos estruturas opacas/genericas, sempre que é necessário informação do tipo char * realizamos uma copia (strdup), de forma a esconder informações dos outros módulos.

5 Queries

Query 1

A query 1 recebe um id que tanto pode ser de um user como de um driver. A grande diferença entre a primeira fase do trabalho e da segunda é que antes percorria a coleção Rides toda para calcular a avaliação média, nº de viagens e o total aferido/gasto. Já nesta fase ela acede apenas à coleção StatusDriver/StatusUser para obter esses dados adquirindo um tempo constante.

Query 2

A query 2 recebe um número N e retorna o top N drivers daquele dataset. Ela faz uma copia do array StatusDriver e ordena-a através de 3 quicksort's, no fim seleciona os primeiros N elementos desse array.

Query 3

A query 3 recebe um número N e retorna o top N users daquele dataset. Ela faz uma copia do array StatusUser e ordena-os através de 2 quicksort's e um selection sort, no fim seleciona os primeiros N elementos desse array.

Query 4

A query 4 recebe o nome de uma cidade e retorna o preço médio da mesma. A query percorre a coleção Rides toda.

Durante a travessia do array sempre que aparece a cidade for igual á cidade daquela viagem incrementamos a distância e o número de viagens.

Retorna o preço médio caso a cidade apareça pelo menos uma vez, -1 caso não apareça nenhuma.

Query 5

A query 5 recebe duas datas e retorna o gasto médio. Começamos por fazer procura binaria da data de início no array da coleção dos Rides, quando obtemos a posição de arranque percorre o array enquanto a data na posição atual for menor ou igual á data do fim. Retorna o gasto médio

caso exista uma viagem nesse intervalo de tempo, -1 caso não apareça nenhuma.

Query 6

A query 6 recebe duas datas e o nome de uma cidade e retorna a distância média percorrida nessa cidade no intervalo de tempo definido.

Começamos também por fazer procura binária da data de início do array da coleção Rides, quando obtemos a posição de arranque percorremos o array enquanto a data na posição atual for menor ou igual á data do fim.

Durante a travessia do array sempre que aparece a cidade for igual á cidade daquela viagem incrementamos a distancia e o numero de viagens.

Query 8

A query 8 recebe um género e uma idade do perfil e retorna a lista das viagens onde o género do user e driver é igual e têm ambos perfis com X ou mais anos.

Começamos por percorrer o array da coleção rides todo e sempre que os géneros do condutor e do user são iguais ao género inserido no input verificamos se ambas as contas têm X ou mais anos de idade. Caso essa condição se confirme adicionamos a um array.

No fim ordenamos através de um quicksort e um selection sort. Retorna NULL caso não exista nenhuns perfis com essas características, caso contrário retorna a lista dos perfis.

Query 9

A query 9 recebe duas datas e retorna todas as viagens onde o condutor recebeu tip.

Começamos por por fazer procura binária da data de início do array da coleção Rides, quando obtemos a posição de arranque percorremos o array enquanto a data na posição atual for menor ou igual á data do fim.

Durante a travessia do array sempre que aparecer uma tip com valor maior que zero adicionamos ao array. No fim ordenamos através de dois quicksorts e um selection sort. Retorna NULL caso não existam viagens, caso contrário retorna a lista de viagens.

Medição de desempenho

Medição de tempo

Para calcular o tempo médio de execução de cada query vamos realizar 10 execuções onde iremos retirar os outliners(maior e menor valor).

O método que usado para medir o tempo foi usando a biblioteca "time.h". Durante a medição do tempo foi usado como base ambos os datasets com entradas invalidas e os inputs usados nos testes automáticos sendo ignorado qualquer output nulo.

Metodo usado para a medição do tempo(por exemplo query 5):

```
clock_t start, end;
double cpu_time_used;
start = clock();
...
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
printf("Fim da Q5 - %f segundos \n", cpu_time_used);
```

Testes Regulares

	1	2	3	4	5	6	8	9
A*	0.00001	0.00533	0.10215	0.26993	0.00903	0.00179	0.88049	0.01849
B*	0.00001	0.00534	0.10331	0.25570	0.01220	0.00296	0.77481	0.01453
C*	0.00001	0.00767	0.14923	0.52047	0.07136	0.02751	1.31814	0.11565

Tempo de execução A*: 8.11565s
Tempo de execução B*: 9.43757s
Tempo de execução C*: 15s

Testes Largos

	1	2	3	4	5	6	8	9
A*	0.00001	0.10070	5.47879	3.17640	0.38128	0.10409	11.14926	0.00761
B*	0.00008	0.30686	5.28600	3.45483	0.34798	0.20988	16.50426	0.05817
C*	0.00001	0.12299	8.35485	5.36297	0.75037	0.56068	16.86887	0.03363

Tempo de execução A*: 249.53592s
Tempo de execução B*: 354.81730s
Tempo de execução C*: 393s

A*: HP Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz, 2592 Mhz, 6 Núcleos. SSD 256Gb Ubuntu
B*: HP Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz, 2496 Mhz, 4 Núcleos. SSD 512Gb WSL
C*: Pc usado nos testes automáticos (dockerfile),(valores aproximados)

Medição de memória

Durante a execução do programa para os datasets regulares com entradas invalidas os computadores A* e B* obtiveram uma ocupação máxima de memoria de 366,24Mb (valores obtidos através do valgrind) com 25.749.864 allocs e free's já para o computador dos testes automáticos registou-se 399Mb usados. Durante a execução com os datasets largos com entradas invalidas os computadores A* e B* obtiveram uma ocupação máxima por volta dos 3500Mb. Já para o computador dos testes automáticos registou-se 3531Mb usados.

Conclusão

Durante a realização deste trabalho aprendemos a importância do uso da modularidade e encapsulamento. Os resultados obtidos na query 8 não atingiram as nossas espectativas.

Infelizmente, optámos por não definir a query 7 para melhorar o código que já tínhamos em questões de memória e desempenho.

Algo que nos prejudicou no desempenho foi o não uso da biblioteca *glib*, por escolha própria.

No modo interativo podíamos ter definido uma função que definisse o tamanho da janela 110x30 ao invés de ajustar manualmente, mas como não havia problema optamos por não alterar.

Para terminar, sentimo-nos satisfeitos com o trabalho em geral, mas infelizes por não terminar a query 7.