Universidade do Minho
Escola de Engenharia

# Relatório de Laboratórios de Informática III

## Guião 3

2ºano - 1ºsemestre

Trabalho realizado por:
Joana Branco (A96584)
Joana Pereira (A97588)
Marta Sá (A97158)

Braga, 7 de fevereiro de 2022

## Introdução

Para este guião foi-nos proposta a resolução de alguns problemas relativos ao programa em desenvolvimento, entre eles, a implementação de um novo mecanismo de interação, a execução de testes funcionais e de desempenho para cada *query* e, por fim, uma nova forma de gestão de dados, uma vez que utilizamos ficheiros de maior dimensão.

De acordo com estas etapas a serem resolvidas, o nosso *software* é dividido em três partes gerais: a parte responsável pela execução do código até este ponto já desenvolvido, isto é, do código capaz de analisar um ficheiro de entrada e, a partir dele, executar as *queries* apresentadas no guião anterior, imprimindo cada *output* num ficheiro de saída, a parte responsável pelas mesmas *queries*, mas desta vez de maneira mais interativa, ou seja, exibindo um menu que permite ao utilizador escolher a *query* que deseja executar e os respetivos *inputs*, dispensando assim, a existência de comandos num ficheiro e, por fim, a parte encarregue de realizar os testes de desempenho de cada *query*, evidenciando o tempo de execução médio de cada uma e verificando, através da comparação com os *expected files*, se os resultados estão corretos. Desta forma, assumindo a existência do ficheiro "commands.txt" na pasta entrada, as duas primeiras partes são executadas a partir do comando ./guiao-3 e a última, juntamente com os *expected files*, com o comando ./expectedF e ./testsQ, respetivamente.

Concluindo, para esta fase do trabalho, adquirimos como objetivo a concretização de um programa mais interativo, visualmente apelativo e com um bom desempenho, capaz de elaborar os resultados esperados, respeitando os conceitos de modularidade e de encapsulamento.

## Modularidade

A modularidade tem como objetivo separar o código fonte de um software em várias partes de forma a permitir um melhor entendimento e manutenção do mesmo. Desta forma, tornamos o programa mais flexível implementando vários *headers files* para as várias componentes do código, cada uma contendo os tipos de dados necessários e as API 's, isto é, as funções responsáveis pela interação com o módulo. De facto, cada *header file* encontra-se relacionada não apenas com o respetivo ficheiro de implementação, onde inserimos o código de cada função e a especificação das estruturas de dados, mas também com outras entidades externas, pelo que é importante expor o mínimo de informação possível em cada módulo.

## Encapsulamento

Na criação de softwares é comum utilizar o conceito de encapsulamento quando se desenvolve código modular. Este conceito permite tornar o módulo opaco em termos de implementação e de estrutura interna, uma vez que, a informação é ocultada ao código que executa o módulo. Para além disso, o encapsulamento funciona como uma propriedade que garante uma maior segurança e um acesso a dados mais controlado.

No nosso projeto, começamos por impedir os acessos diretos aos diversos módulos, determinando as assinaturas das funções e das estruturas de dados a serem exportadas (*Hash Tables*, Listas Ligadas e Árvores) nos *header files* que lhes competem. Por conseguinte, nos ficheiros de implementação, encontram-se definidos os campos das estruturas de dados, bem como o código de cada função. Por sua vez, algumas destas funções estão restritas ao próprio módulo através do uso da palavra-chave *static* na sua declaração. Desta forma, contribuímos para uma maior segurança no código, uma vez que conseguimos impedir que uma identidade externa modifique a representação interna do mesmo.

Por fim, tratamos de utilizar métodos *getters* (identificadas pela palavra *get*), encarregues de ler um valor, e setters (identificadas pela palavra *change*), encarregues de atualizar um valor para resolver o problema de acesso a dados cuja estrutura interna é desconhecida pelo ficheiro. Uma parte das funções *getters* são responsáveis por devolver uma cópia da estrutura para a qual apontam, uma vez que trabalham com apontadores, impedindo assim que a representação interna desta seja modificada.

## Mecanismo de Interação

De forma a tornar o programa mais interessante, foi implementado um menu de interação, visualmente atrativo, entre o utilizador e a máquina. Este menu é inicializado quando o ficheiro de entrada, "commands.txt", está vazio e, por sua vez, apresenta não só a lista de *queries* disponíveis para execução, como também a opção de saída do programa. Desta forma, o utilizador tem liberdade na escolha da *query* e nos *inputs* que a mesma deve receber. Por fim, os valores resultantes destas operações, os *outputs*, são guardados em ficheiros de saída e, posteriormente, impressos no terminal do utilizador[1].

A prática deste mecanismo passou pela análise dos ficheiros de entrada e, em específico nas *queries* 5 a 10, pela identificação dos *inputs* e inicialização de um *array* de tamanho t[2]. Este último, funciona como suporte para navegar nos ficheiros com os *outputs* gerados aquando da seleção das *queries* e, consequentemente, para imprimir o resultado final. Tendo em conta esta funcionalidade, a estrutura de dados indicada armazena no respetivo índice o número acumulado de caracteres que existem nas 25 linhas de cada página (exceto na *query* 10 que retém, no máximo, N linhas por página). Desta forma, o valor do *offset* é obtido através do acesso ao índice do array, o que permite, por intermédio da função

[1] Estes ficheiros são obtidos a partir de uma versão melhorada dos mecanismos utilizados no guião 2.
[2] Para a maioria das *queries*, a variável t (número de páginas) é calculada a partir do valor do *input* N dividido por 25 (número regular de linhas). No entanto, esta regra não se verifica na *query* 10 que atribui a t o resultado da divisão de n (total de linhas no ficheiro gerado) por N (valor do *input*) nem na *query* 7 que, de forma semelhante, atribui a t o quociente entre N (total de linhas no ficheiro gerado) e 25 (número regular de linhas).

*fseek (),* navegar num ficheiro. Assim, uma vez selecionada a *query* desejada, o utilizador pode observar o resultado destes processos e ainda tomar uma decisão relativamente à próxima ação, isto é, o usuário além das opções sugeridas como o avanço, o recuo e o salto para uma página em específico, pode também escolher avançar diretamente para a última página ou até voltar ao menu inicial.

## Testes Funcionais e de Desempenho

Com o objetivo de verificar o desempenho do programa desenvolvido, foram criados testes que permitem validar o resultado e avaliar o tempo médio necessário para a conclusão de cada *query*, respeitando a modularidade e o encapsulamento das componentes do sistema.

A validação é feita através da comparação dos resultados com os *expected files* que são responsáveis por guardar os *outputs* esperados. De facto, para gerar os *expected files* como ficheiros de saída, adaptou-se a estratégia utilizada pela função principal do programa, isto é, adicionou-se um novo parâmetro, v, à função *executeQuery ()* que indica a cada *query* o destino dos *outputs* correspondentes. Para além disso, descartou-se a apresentação do menu, uma vez que, neste caso, utilizamos comandos definidos no ficheiro de texto "*commands.txt*" (para os *expected files*) e comandos definidos na própria função da *query* a ser testada (no módulo *testQuery.c*).

Uma vez que os *expected files* são gerados (por intermédio da *Makefile*), podemos começar a correr os testes destinados a cada *query*. Por um lado, os testes das 4 primeiras *queries* dependem apenas do tempo de execução calculado nas funções principais dos módulos *users.c, commits.c e repos.c*, com a intenção de descartar o trabalho realizado ao elaborar as *hash tables* e assim manter um tempo útil para cada uma delas. Por outro lado, os restantes testes dependem das 10 execuções feitas de forma cíclica e dos valores dos inputs[3] onde, excecionalmente, fez-se variar o valor de N para a *query* 5, fixando-o para as outras. Ainda nesta *query*, a variação do valor de N teve como objetivo avaliar a tendência central do resultado obtido e ainda confirmar, por intermédio dos *expected files*, que o top N mantém-se.

De forma geral, o tempo de execução de cada *query* em cada uma das máquinas foi calculado com o auxílio da biblioteca *<time.h>* que disponibiliza variáveis e funções inicializadas em diferentes momentos do programa, essenciais para os testes funcionais. Para além disso, é importante realçar que, tal como sugerido, retirou-se o valor máximo e mínimo dos tempos obtidos e só depois é que se calculou a média dos valores restantes.

[3] Como se trata de um módulo de testes, optamos por não verificar os *inputs* pelo que estes são assumidos como válidos e no formato esperado.

## Gestão de Dados

Apesar do aumento relevante do tamanho dos ficheiros de entrada, neste guião, não foram feitas muitas mudanças ou novas implementações relativamente à gestão de dados, dado que as estruturas de dados e os algoritmos elaborados para responder às *queries* foram, a nível geral, bem concebidos durante o guião 2[4]. De facto, o nosso *software* obtém um bom tempo de execução utilizando *hash tables* para armazenar em memória os dados analisados aquando do arranque do programa, listas ligadas e árvores para reter, também em memória, a informação tratada durante a execução das *queries*. Esta informação é, por sua vez, guardada em ficheiros de saída. Para além disso, este resultado deve-se à aplicação de funções que tratam e armazenam dados nas respetivas estruturas antes da utilização das *queries* (*dateWithUser ()* e *insere_message ()* no módulo *commits.c*).

Desta forma, as otimizações feitas basearam-se principalmente na modificação da *query* 9 que deixou de utilizar um *array* para passar a utilizar um campo na própria estrutura do *user* com o intuito de contabilizar o total de *commits* feitos por ele no repositório de um amigo. Para além disso, aproveitamos os dados guardados em disco no final da execução de cada *query* para que, neste guião, seja possível utilizá-los durante a elaboração do menu e dos testes funcionais[5].

Concluindo conseguiu-se organizar os dados de forma a permitir uma resposta por parte das *queries* em tempo útil, respeitando um funcionamento interno opaco para quem pretenda utilizar os módulos. É também relevante indicar que, a nível de custo computacional, a função *strdup () e* as *hash tables* ocupam grande parte do armazenamento em questão e, de forma a manter padrões razoáveis de usabilidade, aplicou-se várias vezes a função *free ()* durante o código para que alguns ficheiros sejam libertados.

## Correções

Neste capítulo tratamos de mencionar as correções feitas no *software* relativas aos guiões anteriores, visto que alguns mecanismos neles desenvolvidos são utilizados nesta fase do projeto.

No que toca ao guião 1, como os ficheiros de entrada contêm linhas inválidas, melhoramos os processos utilizados para autenticá-las, nomeadamente, aqueles responsáveis pelas datas (*validDate ()*) e pelos parâmetros (*validPar ()* e *validList ()*).

No guião 2, para além de incorporar os critérios de modularidade e encapsulamento (referidos nos tópicos a eles destinados), implementamos algumas medidas:

- Como referido anteriormente, aperfeiçoamos o desempenho da *query* 9, ao tornar possível incrementar um novo campo na estrutura de dados dos *users* (*totalC*). Inicialmente, a *query* utilizava um *array* para armazenar e incrementar o índice correspondente ao *user* à medida que este cumpria determinada condição.

[4] A nível geral, durante este guião, mudamos apenas o fluxo de dados a serem tratados (alguns *outputs* não eram corretos) e não a forma como as estruturas que os tratam são construídas.
[5] Nos testes funcionais, este processo foi útil para comparar os *expected files* com os ficheiros produzidos.

- Corrigimos o problema mencionado na conclusão do guião 2, relativamente à impressão dos *outputs* das *queries* em ficheiros de saída.
- Corrigimos o método de comparação de *strings,* passando a utilizar a função *strcmp ()*.
- Corrigimos o resultado final de quase todas as *queries* o que acabou por consumir bastante tempo, uma vez que foi necessário refazer completamente algumas delas.
- Passamos a apresentar os *outputs* da maneira que foi sugerida durante a primeira defesa.
- Assumindo que o número de *inputs* é corretamente atribuído à *query*, implementámos uma medida que apenas permite a execução do programa quando os *inputs* são válidos.[6]

## Resultados Obtidos

| PC | Query | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | | 6 | 7 | 8 | 9 | 10 |
| **1** | 0.000002 | 1.86 | 0.000001 | 0.000003 | 0.73 | 0.90 | 1.44 | 0.06 | 0.17 | 3.18 | 2.60 |
| **2** | 0.000002 | 1.11 | 0.000001 | 0.000003 | 0.65 | 0.84 | 1.19 | 0.05 | 0.16 | 2.30 | 1.11 |
| **3** | 0.000001 | 1.18 | 0.000002 | 0.000003 | 0.67 | 0.85 | 1.39 | 0.07 | 0.18 | 2.44 | 1.62 |
| **N** | - | - | - | - | [25,50, 75,100] | 100 | 100 | - | 100 | 100 | 100 |
| **Data/ Lang** | - | - | - | - | 2010-01-01 até 2015-01-01 | | Python | 2014-04-25 | 2016-10-05 | - | - |

[6] Ao introduzir estes mecanismos de validação, achamos por bem expandi-los para o módulo encarregue de gerar o menu interativo de modo que o programa aguarde que o utilizador insira *inputs* válidos ou que aborte quando estes não são do tipo esperado (o *assert ()* falha).

| PC | SO | CPU | Memória | Disco | Método de instalação |
|---|---|---|---|---|---|
| **1** | Ubuntu 20.04.3 LTS (64 bit) | Intel® Core™ i7-9750H CPU @ 2.60GHz | 7,8 GiB | 53,7 GB | Virtual Machine |
| **2** | Ubuntu 20.04.3 LTS (64 bit) | Intel® Core™ i7-6820HK CPU @ 2.70GHz × 8 | 31,3 GiB | 1,5 TB | Dual Boot |
| **3** | Ubuntu 21.04 LTS (64 bit) | Intel® Core™ i7-7500U CPU @ 2.70GHz × 4 | 7,7 GiB | 1 TB | Dual Boot |

## Discussão de Resultados

Para as *queries* de 1 a 4 o tempo médio de execução é imediato, visto que, como já foi referido no tópico "Testes Funcionais e de Desempenho", o trabalho realizado para elaborar as *hash tables* dos *users* e dos *commits* foi descartado[7], possibilitando um tempo de execução inferior a 5 segundos. Comparando os resultados dos três PC's é possível observar que os valores não diferem muito o que permite concluir que estas primeiras quatro *queries* têm um bom desempenho em diferentes ambientes.

Na *query* 5, de forma a deixar os testes mais completos, acrescentou-se a possibilidade de variar o valor de N, o que permite concluir que *outputs* se mantêm pela mesma ordem. Além disso, como esperado, o tempo de execução médio para os diferentes N's utilizados é menor. Neste caso, os resultados obtidos pouco variam entre máquinas e estes valores apresentam uma quantia relativamente baixa tendo em conta que a *query* percorre apenas a *hash table* dos utilizadores, armazena os dados em Listas Ligadas e apenas acede a uma função externa ao próprio módulo.

Na *query* 6 testou-se 10 vezes, para o mesmo valor de N, a capacidade de determinar o número utilizadores com mais *commits* em repositórios de uma determinada linguagem. De acordo com os resultados obtidos, o tempo médio de execução foi ligeiramente superior, comparativamente a outras *queries*, e a diferença entre os diferentes computadores foi pouco significativa. Isto justifica-se pelo facto da *query* 6 percorrer a maior das *hash tables,* a dos *commits*, e a dos *repos*. Apesar disso, o tempo de execução é aceitável para o tamanho do *input*, uma vez que conseguimos uma boa gestão de dados ao armazená-los em listas ligadas.

Por sua vez, a *query* 7, em média, demora relativamente pouco tempo. O foco, neste caso, centra-se apenas na visita ao ficheiro dos *repos*, visto que durante a inicialização da *hash* dos *commits*, fomos atualizando o campo *date* da estrutura de dados dos *repos* com a data do *commit* mais atual no respetivo repositório (função *changeRepos ()*). Assim, na query 7 para decidir o *output*, bastou comparar a data do *input* com o campo *date* do repositório a analisar, o que torna esta *query* muito mais simples e rápida. Além disso, para a concretização destes testes usamos sempre o mesmo *input*, repetindo os referidos 10 vezes. Refletindo

[7] O trabalho realizado pela *hash* destinada aos repositórios não foi descartado, dado que não influencia muito no tempo de execução.

sobre os três valores diferentes podemos concluir que esta *query*, também, apresenta um bom comportamento em diferentes ambientes.

Na *query* 8[8], os resultados do tempo de execução médio são muito próximos, em especial nos computadores 1 e 3 que apresentam o mesmo valor. Tal como referido na *query* anterior, aqui é preciso proceder ao acesso de um ficheiro de entrada e inserir os dados numa estrutura própria para, posteriormente, ser ordenada o que torna o programa não lento. Neste caso, os valores de N e da data inserida pelo utilizador foram fixados a um determinado valor e o teste foi feito executando a *query* repetidamente 10 vezes.

Já na *query* 9 estavam a ser sentidas algumas dificuldades a gerar a *query* de forma cíclica pelo que inicialmente estavam a ser usados diferentes valores consoante a máquina utilizada. Esta ocorrência deveu-se a problemas na gestão de memória, uma vez que esta não era libertada. Felizmente o problema foi resolvido através da utilização de *frees* e os testes passaram a ser igualmente executados. Esta *query* é ligeiramente mais demorada em comparação com as restantes. Isto deve-se ao facto de percorrer a *hash* dos *commits*, a maior das *hash*, enquanto verifica se o *owner* do repositório é amigo do utilizador e de seguida percorrer também a *hash* dos *users* enquanto insere os dados necessários numa lista ligada para, posteriormente, serem impressos. Contudo, apesar de todas as adversidades sentidas, foi possível executar a *query* em tempo útil.

Por fim, na 10 com o objetivo de determinar os top N utilizadores com as maiores mensagens por cada repositório percorreu-se a *hash* do *repos* com o auxílio de uma estrutura de dados onde guardávamos o *id*, o *login*, o tamanho da mensagem (anteriormente calculado durante a elaboração da *hash table* dos *commits*) e o *id* do repositório. Para facilitar a impressão de apenas um repositório por página, na parte do menu, o ficheiro onde se guardou os *outputs* ficou com algumas linhas em branco, de modo a completar sempre N linhas por repositório. Tendo isto em conta, o tempo médio de execução foi ligeiramente superior nesta *query*, uma vez que o ficheiro de saída contém um elevado número de linhas.

Em conclusão, no que toca à *performance* antes *versus* depois, podemos afirmar que houve uma evolução significativamente positiva, visto que fomos capazes de melhorar a gestão de dados tanto a nível de armazenamento como a nível de libertação dos mesmos.

[8] É importante mencionar que, nesta *query, contabilizamos* o tipo de linguagem "None" como *output*.

## Conclusão

A elaboração deste guião permitiu explorar novos conhecimentos relativos à questão visual, desenvolver os conceitos de modularidade e de encapsulamento e otimizar o desempenho do programa.

Tendo como retrospetiva os resultados obtidos pela execução do programa, podemos afirmar que este ocorreu de forma positiva e, apesar das dificuldades sentidas no guião 2, conseguimos ultrapassar essa situação e otimizar o seu comportamento. No entanto, há alguns aspetos que poderiam ser melhorados, tais como uma implementação mais eficaz de *frees* das estruturas de dados. Uma vez que esta não alterava substancialmente a memória ocupada na execução do programa, não temos a certeza de que este processo afeta positivamente o projeto.

Outro aspeto passível a ser melhorado, passa por descartar os parâmetros dos ficheiros dos *users*, dos *repos* e dos *commits* que não foram utilizados para a execução do programa.

Contudo, apesar de alguns percalços durante a elaboração deste guião podemos afirmar que estamos satisfeitas com o resultado produzido.