Relatório da Fase 2 - Grupo 03

Francisco Macedo Ferreira (A100660)
Júlio José Medeiros Pereira Pinto (A100742)

5 de fevereiro de 2023

Conteúdo

1 Introdução

Este relatório tem como intuito abordar a segunda fase do projeto da UC Laboratórios de Informática III do ano letivo 2022/2023. Nesta fase como objetivos o trabalho necessitava a implementação de todas as queries , de um modo de operação interativo (incluindo menu de interação e um módulo de paginação para apresentação de resultados longos) e evolução de aspetos relacionados a modularidade, encapsulamento e qualidade de código.

Este relatório irá abranger as decisões tomadas pelo grupo tal como o método de raciocínio para desenvolvimento do mesmo.

2 Alterações realizadas

2.1 Ajustes apontados pelos docentes

Após a apresentação da primeira fase do trabalho foram levantados alguns pontos por parte dos docentes.

- Originalmente, no catálogo, tínhamos todos os *Arrays* e *Hashtables* das várias estruturas juntas, porém foi levantado o ponto que de que poderia constituir uma quebra de modularidade. Assim sendo, tornámos o catálogo num principal, que guarda os diferentes catálogos das estruturas, catálogos estes opacos ao catálogo principal (será abordado mais abaixo).
- Também tínhamos ideias de explorar o uso de programação paralela, mas isso foi invalidado, já que a máquina de testes roda em *single-core*.
- Foi nos avisado alguns ficheiros temporários de funções utilitárias não eram uma solução ideal e então foram também corrigidas ao longo do desenvolvimento do projeto.
- Separámos também a lógica de *tokenização* de pontos e vírgulas do *parse* do input e apresentação de output, fazendo com que vários formas de input e output sejam facilmente implementadas.
- A sugestão de trocar a função *fgets* pela *getline* na leitura dos CSVs e inputs de queries não foi implementada devido a que, para além da função não existir por padrão em alguns compiladores, um utilizador malicioso pode escrever um comentário grande o suficiente para que se esgote a memória. No entanto, o tamanho do *buffer* do *fgets* foi aumentado para 8192 para acomodar estruturas que tenham possíveis nomes grandes.

2.2 Ajustes de fator escala (Dataset maior)

Da forma como tínhamos o projeto implementado, quando usado o dataset maior, o programa mesmo assim continuava a ter um desempenho dentro dos limites de tempo com grandes margens. As estruturas de dados e estratégias para a resolução de queries foram então adequadas, pelo que não alteramos a implementação das queries já feitas (1, 2, 3, 4 e 5). As previsões de possíveis alterações mencionadas na fase 1 não foram necessárias de serem implementadas, exceto o *lazy loading* que foi útil no modo interativo (será abordado mais abaixo). No entanto, foram feitas algumas melhorias que acabaram por otimizar estas funções como resultado disso.

3 Pipeline

Neste momento, o nosso programa funciona da seguinte maneira:

1. Os ficheiros dos datasets são lidos conforme o *input* do programa (seja pedido no modo interativo ou passado como parâmetro).
2. Consoante o tipo de estrutura, a partir de um módulo de leitura, um *TokenIterator* é passado para o *parser* da estrutura correspondente. Essa *iterator* permite esconder a implementação de leitura de linhas separadas por ponto e vírgula. O *parser* da estrutura vai avançando o *iterator* até recolher os dados necessários e a estrutura é criada se todos os campos forem válidos.
3. Após cada criação de estrutura, essa é adicionada no catálogo correspondente e informação necessária para resolução de queries de outras estruturas são atualizadas.
4. No fim da leitura do datasets, os catálogos são indexados conforme as exigências das queries (Apenas quando o *Lazy loading* está desativado).
5. No caso do modo *Batch*, os ficheiros de queries são lidos. No caso do modo Interativo a *query* (ou comando do programa) a ser executada é pedida continuamente ao utilizador. Cada query é então passada para módulo de resolução de queries.
6. As queries são computadas acedendo ao catálogo principal. O seu resultado é enviado, separado por *tokens*, para um módulo de *buffer* de output. Este módulo abstraí a escrita para as diferentes formas de output: ficheiro separado por ponto e vírgula no modo *Batch* ou terminal com possível paginação no modo Interativo.
7. A memória do catálogo é libertada e o programa termina a execução.

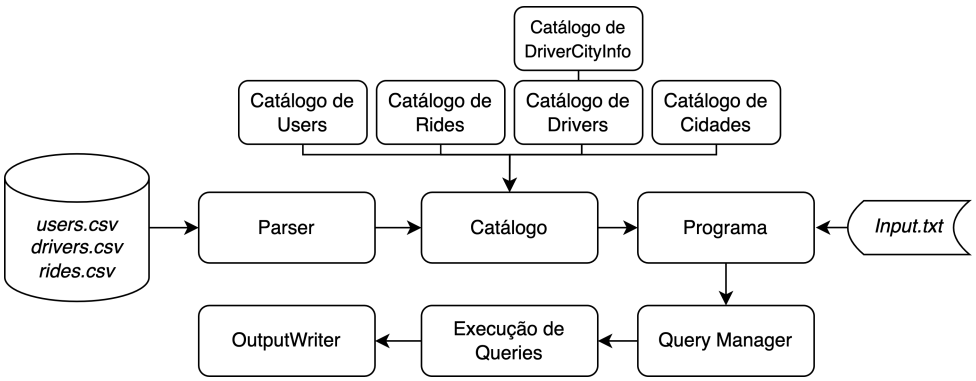

Figura 1: Representação do pipeline

4 Queries Implementadas

Para a segunda fase do trabalho foram aplicadas todas as queries em falta (Queries 1 a 5 realizadas na fase anterior). Foram implementadas as queries 6, 7, 8 e 9. A implementação das queries anteriores não foi alterada, apenas foi otimizado algumas partes do catálogo para que a indexação dos dados fosse mais rápida.

4.1 Query 6

De maneira a conseguirmos calcular o preço médio dentro de uma data usamos o *array* de rides que está ordenado por data. Com ele fazemos *binary search* para encontrar a data do começo (ou a próxima maior caso não exista) e percorremos o array, linearmente, até a segunda data, fazendo média dos preços das viagens. O preço de cada viagem foi calculado e guardado na ride durante a inserção nos catálogos para evitar *lookups* às informações do condutor.

4.2 Query 7

Para a query 7 tivemos de recorrer a uma nova estrutura de dados separado dos condutores: *DriverCityInfo*. Essa estrutura de dados guarda o score de um condutor numa cidade específica. Conforme a inserção de rides no catálogo, vamos somando o score do condutor daquela ride a um valor acumulado dentro da tal estrutura de dados. Para cada cidade, existe um array de

DriverCityInfo e uma *hashtable* temporária para fazer com que na inserção não seja necessário percorrer o array inteiro para encontrar (ou ver que ainda não existe) o *DriverCityInfo* onde queremos somar o score de tal ride. Quando surge uma nova cidade, é criado um novo *array* e uma nova *hashtable* temporária.

Não foi possível, por exemplo, guardar um array na própria estrutura de dados do condutor porque não sabemos a quantidade de cidades que existem no dataset até o lermos por completo, pelo que teríamos que fazer *reallocs* muito constantes, o que impactaria muito a performance.

Assim, com esta implementação, conseguimos fazer com que cidades sejam lidas e adicionadas dinamicamente de forma eficiente. Quando a query é chamada, todos os arrays são ordenados (independente da cidade) e pegamos nos primeiros n elementos do array da cidade pedida e imprimimos o resultado formatado.

4.3 Query 8

De maneira a fazer a query 8 decidimos adicionar dois *arrays* ao Catálogo de Viagens. Estes *arrays* são preenchidos conforme a inserção das viagens. Cada array guarda referências a viagens em que o utilizador e o condutor têm o mesmo género para cada um, masculino e feminino. No fim da inserção, os dois arrays são organizados, conforme os requisitos da query, pela data de criação da conta do condutor e em caso de empate, pela data de criação da conta do utilizador e em caso de empate novamente, pelo id da ride. Na resolução da query, percorremos o *array* dependendo do género pedido, e se as idades da conta do utilizador e do condutor forem superiores à idade pedida, adiciona-o ao array de resultados. Este acaba quando a idade da conta do condutor for inferior à idade pedida, visto que o array está ordenado por essa idade da conta do condutor. Então não haverá mais condutores com conta de idade superior à pedida, no resto do *array*.

4.4 Query 9

Por último, tal como na query 6, utilizamos o *array* de rides ordenado por data e fazemos *binary search* pela primeira data. Percorremos o *array* linearmente a partir daí, e gerámos o resultado com as viagens em que gorjeta é superior a 0 até chegarmos ao final do *array* ou à segunda data.

5 Modo Interativo

Para implementação do modo interativo, após alguma discussão, decidimos avançar com um no estilo de um *Terminal* emulado dentro do programa baseado em comandos que o utilizador pode executar. Para isso usamos a biblioteca *readline* para leitura e histórico de comandos.

5.1 Lista de Comandos

O modo interativo tem vários comandos que podem ser executados. Esses comandos foram implementados de forma genérica, de modo seja possível adição de novos muito facilmente.

Escrevendo ***help*** no *terminal* aparece uma lista com todos os comandos.

Para execução de *queries*, o utilizador pode seguir dois métodos: escrever a query no formato da mesma como se fosse um comando ou então pode utilizar o comando ***file*** e passar-lhe um ficheiro de input, semelhante ao modo *batch*. Ao escrever uma query com o número de argumentos incorreto, uma mensagem de uso e uma pequena explicação da query são mostradas.

Para visualização do output do comando *file*, existem os comandos ***cat*** e ***list-output***, que leem e listam ficheiros de outputs na pasta de Resultados, respetivamente.

Também existe o comando ***clear*** que limpa as linhas do terminal e ***restart*** que recomeça o programa, pedindo um novo catálogo.

Caso o utilizador pretenda fechar o programa basta escrever o comando ***exit***.

5.2 Paginação

A pedido de implementação, para queries que tenham um output superior a um valor estipulado de 10 linhas, um menu de paginação surge onde é possível percorrer as páginas, escrevendo ***n*** para avançar para a próxima página, ***p*** para recuar uma página e ≪***número***≫ para ir até uma página específica. Ao trocar de página, todas as linhas da página atual são apagadas do terminal (via recurso a caracteres ANSI) e as próximas são mostradas. Escrevendo ***q*** para sair desse menu, o *footer* onde se troca a página é apagado e a página apresentada atualmente é mantida no *terminal*.

5.3 Histórico de Comandos

Através da utilização da biblioteca *readline/history* foi possível facilmente a criação de um histórico de comandos, em que, como num *terminal* normal, ao clicar nas setas do teclado, é possível navegar pelos últimos comandos digitados.

6 Estruturas de Dados

6.1 Catálogo

Como foi dito na parte de ajustes, tivemos a necessidade de separar o nosso catálogo num catálogo único para os utilizadores, condutores e viagens. Existe um catálogo principal/controlador para acedê-los e controlar dados comuns entre eles.

Em cada um dos catálogos guardámos pelo menos um *array*.

No Catálogo dos Utilizadores guardamos um *array*, ordenado pela distância total percorrida pelo utilizador. Uma *hashtable* para a procura rápida de um utilizador pelo username e ainda também outro array para indexarmos os utilizadores por ID (ID gerado no momento de inserção do utilizador no catálogo).

No Catálogo dos Condutores guardamos um array, ordenado pelo *score* do condutor e um array para indexado pelo ID do condutor para rápido acesso à estrutura a partir dele. Para além destes, este também inclui outro catálogo (*CatalogDriverCityInfo*), utilizado para a realização da query 7, para guardar as informações de cada condutor para uma cidade específica.

No Catálogo das Viagens guardamos um *array* ordenado pela data da viagem. Também é guardado um *array* de *arrays* de viagens por cada cidade, cada índice do *array* principal correspondendo ao ID da cidade e mais dois *arrays* para guardar as viagens onde o utilizador e o condutor têm o mesmo género, um *array* para cada, utilizados para a realização da query 8.

Por fim, também existe um Catálogo de Cidades para atribuição de IDs inteiros a cidades. Será abordado na secção das otimizações.

6.2 *Lazy*

De maneira a tornar o programa mais eficiente decidimos criar uma estrutura de dados já presente ou semelhante em outras linguagens de programação. Para otimizar o tempo de carregamento dos dados, implementamos o '*Lazy*' que é uma estrutura que permite aplicar uma função a um valor quando este é requisitado.

A função e o valor são passados como parâmetros na criação da estrutura.

A estrutura foi implementada da forma mais genérica possível, para que possa ser usada para qualquer tipo de dados.

Isto permite então que, por exemplo, várias funções de *sort* sejam aplicadas apenas quando necessário, pelo que quando uma query nunca é chamada, as estruturas necessárias não sejam indexadas desnecessariamente. Também torna o carregamento no modo interativo mais leve.

Este *lazy sorting* é feito por defeito, mas para dar escolha ao utilizador, pode ser desativado passando *flag –lazy-loading=false* como argumento ao programa (esta *flag* pode estar em qualquer posição não interferindo com os datasets passados como argumento).

6.3 *Program*

De forma a abstrair os possíveis modos de execução (*batch* e *interativo*) foi criado um módulo de ***Program*** que guarda um catálogo e o estado do programa. É aqui que as *flags* do programa são também controladas. Este módulo é responsável pelo tratamento de input e execução de comandos.

6.4 *OutputWriter*

Para diferenciar diferentes formas de *output* existe um módulo que controla para onde vai o output gerado pelas queries. A query escreve cada *token* do resultado para este ***OutputWriter*** separadamente, e notifica quando uma linha de informação termina.

Dependendo da forma de como se cria esta estrutura, o *OutputWriter* pode escrever:

- Diretamente para um ficheiro, separando os *tokens* por ponto e vírgula. Usado no modo *batch* e num comando do modo interativo.
- Para um *array* de *strings*, que a função de escrita, adiciona uma *string* a esse *array*. Usado no modo interativo para fácil gestão de linhas na parte da paginação.
- Para vazio, usado na execução de testes de alguns testes unitários.

Mais tipos de implementação podem ser adicionados, visto que basta criar esta estrutura com diferentes apontadores para funções de escrita.

6.5 *TokenIterator*

Com a mesma ideia do *OutputWriter*, uma estrutura foi implementada de modo a abstrair a leitura de *datasets*. Este *TokenIterator* tem como objetivo, no caso da leitura de CSVs, avançar para o próximo ponto e vírgula, numa linha, para que o *parser* das estruturas não tenha lógica de interpretação de *inputs*.

Para já, esta estrutura suporta apenas leitura de CSVs, mas devido à generalização desta, pode ser expandida futuramente para leituras de qualquer forma de *input*.

7 Otimizações

Apesar das estruturas previamente implementadas permitirem a resolução das queries dentro dos limites de tempo, com grandes margens, decidimos mesmo assim otimizá-las para além do que o que já tínhamos. Para tal, alterámos certas estruturas de dados e certas formas de guardar informações de maneira a atingir-se uma melhor utilização de memória e performance.

7.1 IDs de cidades

Ao invés de se guardar em cada condutor e viagem uma cópia em string da cidade, passámos a guardar um ID inteiro único para cada. Isto diminuiu muito o uso de memória e o número de *strdups* necessários. Este ID é gerado e guardado, no Catálogo das Cidades, conforme se lê novas cidades do dataset.

Por consequência, o tempo necessário para a alocação e libertação das strings é reduzido, tal como o número de bytes necessários para cada condutor e viagem. Isto implica também uma maior localidade espacial, dando uma melhor performance a todas as queries.

7.2 IDs de users

No mesmo contexto, alterámos também a maneira como as viagens guardam os utilizadores. Atribuindo a cada utilizador um ID inteiro único, permite-nos então poupar na quantidade de espaço necessário pelas estruturas das viagens e evita-se que um username do utilizador seja alocado várias vezes pelas viagens Esse ID é gerado conforme a leitura de utilizadores. A identificação do user a partir do ID, é feita através de um array cujo índices são os tais IDs gerados com valores apontadores para estruturas de utilizadores. Devido à query 1, ainda é mantido a *hashtable* que faz ligação dos usernames em string às estruturas de users.

7.3 Data

Inicialmente a data era implementada numa struct com 3 inteiros de 32 bits. Esta pôde ser compactada em apenas um inteiro de 32 bits de forma codificada. Da direita para a esquerda, os primeiros 5 bits da eram reservados para o dia, os próximos 5 bits reservados para o mês, e o resto reservado para o ano. A ordem dos bits foi escolhida para que comparações entre datas seja uma simples subtração. A estrutura é codificada e descodificada através de operações *bitwise*.

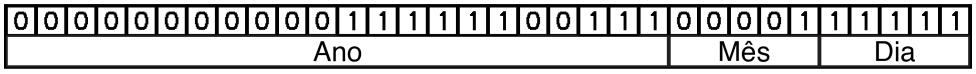

Figura 2: Representação binária da data 31/01/2023

Com isto, para cada ride foi possível poupar 8 bytes e as comparações são muito mais rápidas. Os melhoramentos no uso de memória e performance foram notórios no dataset large, que para 10 milhões de rides foram poupados 80MBs, aumentando também assim a localidade espacial.

7.4 Funções nativas

Com recurso a um *profiler* observámos que funções como *strtok*, *strtol*, *strtod* estavam a demorar alguns meios segundos. Estas são chamadas algumas milhões de vezes. Por isso, implementando funções mais simples e otimizadas para as nossas necessidades, reduzimos grande parte do tempo gasto nessas funções.

7.5 Remoção de campos não usados

Observámos que alguns campos não estavam a ser utilizados em nenhuma query, pelo que foram ignorados na criação das estruturas. Campos esses são, por exemplo, a matrícula do driver e os comentários da ride.

7.6 Estruturas de dados (padding e otimização de memória)

De forma a gastar ainda menos memória, os campos das estruturas foram reorganizados de modo a reduzir o *padding* dessas. Também foram substituídos alguns inteiros de 32 bits por inteiros com menos bits onde fosse possível.

Com isso, cada estrutura está a ocupar:

- Ride: 48 bytes
- User: 64 bytes (mais o espaço dos valores do username e name)
- Driver: 48 bytes

8 Testes Unitários

Os testes unitários foram algo que demos muita atenção no desenvolvimento deste software. Para isso a sua concretização, usámos uma combinação de scripts e código.

Ao chamar *make test*, o programa de testes compila juntamente com as sources do programa principal e é executado. Com ele, recorremos à biblioteca de testes da GLib que permite adicionar testes unitários muito facilmente. Testes unitários estes que testam:

- O encoding, decoding e parsing da estrutura de **Data** implementada.
- Funcionalidade gerais do ***Lazy***, ***OutputWriter*** e ***TokenIterator***.
- O ***Parser*** do user, driver e ride para leitura de linhas de CSVs apenas válidas/apenas inválidas (recorrendo aos datasets dado pelos docentes).
- **Performance** (limite estipulado de 1 segundo para cada query) para o dataset regular.

Caso qualquer uma destes testes falhe, é nos informado quais falharam e o programa retorna um error code diferente de 0. Se não tiver falhado, o script prossegue a fazer testes ***end-to-end*** com o programa principal. O programa principal é executado para os diversos datasets e ficheiros de

queries (dataset regular input 1 e 2 e dataset large), e a cada execução é comparado os ficheiros de output da pasta Resultados com uma de resultados corretos.

Durante a execução, também é verificado o **pico de uso de memória**, que se exceder os valores estipulados de 200MB para os datasets regulares ou 2000MB para o dataset large, o teste é dado como falhado. A verificação do pico de memória é feita com um script que chama o comando ***time*** passando parâmetros adequados conforme o sistema operativo onde é executado (Linux ou MacOS suportados).

8.1 Fugas de memória

A verificação de fugas de memória do programa são separados dos testes principais, pelo que, para a execução desse é feita através de *make leaks*. Este chama o *valgrind* no Linux ou o *leaks* no MacOS (*Valgrind* ainda não é suportado nos novos M1s) para o programa principal com o dataset regular. Esta separação foi feita para evitar o grande tempo de execução do *valgrind* quando for apenas necessário correr os testes.

8.2 Actions do Github

De maneira a reunir todos estes testes, tirámos partido das Actions do GitHub, que, a cada commit, eram rodados: *make*, *make test* e *make leaks* no GitHub. Isto foi muito útil e conveniente, já que, para além de garantir o funcionamento correto do programa ao longo do seu desenvolvimento, como estas são executadas na mesma distribuição Linux do que a da plataforma de testes de LI3, ainda nos informava se o programa rodaria corretamente aí.

Os datasets necessários para a execução deles, são obtidos através de um link para um zip num Google Drive. É feito o download deste e *cacheado* nas Actions do GitHub para futuras execuções.

9 Testes de Desempenho

Comparando o desempenho do programa para os vários datasets em diferentes máquinas, temos na Tabela 2 os resultados obtidos para as especificações dos PCs na Tabela 1.

A metodologia adotada para a medição foi a seguinte:

- O programa foi compilado com as flags *-O3 -flto -funroll-loops*.
- De forma a reduzir a variância entre execuções são feitas algumas execuções de aquecimento.
- Para os datasets regulares, o resultado é obtido a partir da média de 20 execuções com 5 execuções de aquecimento.
- Para os datasets grandes, o resultado é obtido a partir da média de 5 execuções com 2 execuções de aquecimento.
- Os ficheiros de input são os mesmos utilizados na plataforma de testes.
- As médias e desvios padrões do tempo de execução foram obtidos usando a ferramenta *Hyperfine*.

As especificações dos computadores e resultados das medições são então:

	PC 1	PC 2	PC 3
CPU	M1 Pro 8-core (6p/2e)	Intel i7-8550U 4-core	Ryzen 7 7700X 8-core
RAM	16GB LPDDR5 6400MHz	8GB DDR4 2400MHz	32GB DDR5 5600MHz
Disco	500GB NVME	500GB NVME	1TB NVME
OS	MacOS Ventura 13.0.1	ArcoLinux 6.0.9	W11 (WSL2 Ubuntu 22)
Compilador	Clang 15.0.5 (ARM64)	GCC 12.2.0	GCC 11.3.0

Tabela 1: Especificações dos PCs

Datasets	PC 1	PC 2	PC 3
Regular Dataset (with invalid entries)	557.6ms ± 4.7ms	1466.0ms ± 31.0ms	540.4ms ± 2.9ms
Regular Dataset (without invalid entries)	576.3ms ± 7.5ms	1507.0ms ± 49.0ms	559.6ms ± 5.2ms
Large Dataset (with invalid entries)	9.176s ± 0.064s	23.218s ± 0.267s	9.072s ± 0.077s
Large Dataset (without invalid entries)	9.559s ± 0.077s	24.119s ± 0.456s	9.579s ± 0.095s

Tabela 2: Tempos de execução para os Datasets

9.1 Uso de memória

Garantir que o programa não tenha fugas de memória foi um objetivo prioritário pelo que foi devidamente atingido.

Na tabela seguinte, apresentámos os picos de uso memória para os diferentes datasets nos diferentes PCs, medidos com a ferramenta */usr/bin/time*.

Datasets	PC 1	PC 2	PC 3
Regular Dataset (with invalid entries)	92 MB	113 MB	113 MB
Regular Dataset (without invalid entries)	96 MB	116 MB	116 MB
Large Dataset (with invalid entries)	987 MB	1114 MB	1120 MB
Large Dataset (without invalid entries)	1019 MB	1150 MB	1155 MB

Tabela 3: Pico de uso de memória nas diferentes máquinas

Desconfiamos que alterações entre os PCs serão devido a diferentes implementações de alocação de memória por parte do sistema operativo, ou de estratégias de alocação por parte do compilador. Fazendo uma pequena investigação das diferenças, testámos no PC 1 (MacOS) o uso de memória para os mesmos datasets, só que usando *GCC 12.2.0* como compilador. O uso de memória registado com o novo compilador foi ainda mais baixo (94, 97, 925, 964 MB), portanto as mudanças devem dever-se ao alocador de memória do MacOS, a uma diferente implementação da ferramenta de medição de pico de memória ou à natureza da arquitetura ARM.

Com estes resultados podemos notar que as otimizações feitas foram bem conseguidas. O programa está a rodar mais rápido e com menos memória (com as 9 queries implementadas) do que na primeira fase (com apenas 5 queries implementadas).

10 Outros

10.1 *CSV Generator*

Como no início da segunda parte do projeto ainda não nos tinham sido disponibilizados os datasets grandes, fizemos um script rápido em *Python* para gerar datasets de grande escala. Contudo, talvez por inexperiência nossa ou pela natureza da linguagem, os datasets grandes demoravam mais que tempo útil a serem gerados. Portanto, fizemos um programa simples em *Rust* para o mesmo objetivo. Com a implementação nesta linguagem, já conseguimos gerar datasets de maior tamanho em tempo útil e prosseguir no projeto enquanto os datasets originais não eram lançados pelos docentes.

10.2 Documentação

Para a documentação, usámos o padrão do Doxygen no código. De forma a assegurar que todas as funções ficassem devidamente comentadas, a cobertura da documentação pode ser gerada a partir de *'make generate-doxygen'*.

11 Conclusão

Com o desenvolvimento desta segunda fase, conseguimos, tal como na primeira fase, consolidar o nosso conhecimento de C, com conhecimentos mais avançados relacionados a performance, gestão de memória e encapsulamento e modularidade. Mesmo assim, apesar de mais familiarizados com estes conhecimentos, acreditamos que o uso de uma linguagem de baixo nível como C, sem suporte a objetos, causou por vezes alguma confusão na aplicação destes conceitos.

Porém, perante os desafios apresentados, acreditamos que o nosso trabalho teve um desempenho muito satisfatório. Aplicando conhecimentos teóricos e práticos adquiridos durante a UC. A nossa dedicação e empenho resultaram num bom desempenho na unidade curricular e nos preparou para futuros desafios académicos.