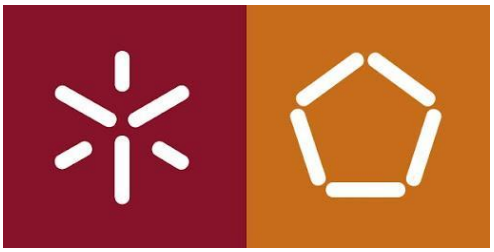
Universidade do Minho
Licenciatura em Engenharia Informática



LABORATÓRIOS DE INFORMÁTICA III

Guião 2

Trabalho realizado por :

GRUPO 10

Nuno Guilherme Cruz Varela - a96455
Gabriela Santos Ferreira da Cunha - a97393
Miguel de Sousa Braga - a97698

A96455
Guilherme Varela

A97393
Gabriela Cunha

A97698
Miguel Braga

1. Introdução

No âmbito da unidade curricular de Laboratórios de Informática III, foi-nos pedida, nesta segunda fase do trabalho, a implementação de uma aplicação que obedecesse à seguinte arquitetura:

- um *parsing* de dados, que trata da leitura dos dados de três ficheiros (“users-g2.csv”, “commits-g2.csv” e “repos-g2.csv”), previamente filtrados pela primeira fase do trabalho;
- um interpretador de comandos, responsável por ler um ficheiro de comandos (“commands.txt”), interpretar cada um e executar a respetiva *query* associada a este;
- três catálogos, para cada ficheiro, que são módulos de dados onde devemos guardar convenientemente os dados provenientes dos ficheiros.

Neste guião, focamo-nos, principalmente, na modularização e abstração do código, sem desprezar, também, a eficiência do mesmo.

2. Organização do Projeto

Nesta segunda fase, o projeto foi organizado de forma a respeitar as boas práticas de modularidade e encapsulamento do código, de modo a melhorar a organização e a facilitar o eventual trabalho de manutenção futuro. O código foi assim dividido em 4 partes: *parsing* dos dados, realização dos catálogos (*users*, *commits* e *repos*), interpretação de comandos e *queries*.

A secção de *parsing* lê os dados para memória, guardando os dados necessários à resolução das *queries* nos respetivos catálogos de cada ficheiro. Não faz qualquer tipo de filtragem aos dados inválidos, pois esse era o propósito do guião-1. De modo a evitar passagens desnecessárias nas novas estruturas criadas, este módulo foi construído de modo a calcular, em tempo de *parsing*, informações pedidas nas *queries* estatísticas (1 a 4).

A secção de interpretação de comandos lê cada linha do ficheiro de comandos, chamando a *query* respetiva com os dados disponibilizados.

A secção das *queries* é constituída por 6 ficheiros, um para cada *query* parametrizável. Cada *query* é independente, recebendo os apontadores para os catálogos, tendo acesso às *APIs*.

Todos estes módulos e as suas funções encontram-se devidamente documentados, aumentando a organização e contribuindo para uma mais fácil interpretação do código.

3. Estruturas de dados

Depois de bastante ponderação relativamente à forma como as *queries* poderiam ser resolvidas, chegamos à conclusão que seria importante guardar os dados numa estrutura que fosse bastante eficiente em termos de acesso (inserção e procura) com base nos *ids* (*repo_id*, *commiter_id*, *author_id*). A maior parte das queries não estatísticas envolvia cruzamento entre os vários catálogos, de modo a relacionar informação, por exemplo *author_ids* (disponíveis nos commits) com *logins* (disponíveis nos *users*).

Uma maneira de resolver este problema seria recorrendo a árvores binárias equilibradas, organizadas por *ids*, tal como foi feito no guião-1, resultando numa complexidade logarítmica quer para a pesquisa, quer para a inserção.

Uma opção alternativa passaria pelo uso de *hashtables*, onde se inseriam dados de utilizadores, *commits* e repositórios no lugar especificado pelo valor da hash calculado para o *id*. Uma das vantagens desta abordagem consiste em reduzir o tempo de inserção e pesquisa para constante. Apesar de este poder ser linear para operações de duplicação da tabela ou quando esta apresenta um fator de carga bastante alto, de forma amortizada o tempo médio seria constante.

Reconhecemos, contudo, que esta abordagem não é a mais eficiente para resolver algumas *queries*, nomeadamente as que envolvem datas, em que uma representação em árvore ordenada por datas tornaria o processo de filtragem mais eficiente. No entanto, a utilização desta estrutura traria tempos de procura menos eficientes para as restantes *queries*.

Optamos então por esta segunda alternativa. Para a implementação desta estrutura de dados nos diferentes catálogos, recorremos à biblioteca “glib”, onde já se encontram definidas grande parte das funções que iríamos utilizar ao longo do trabalho.

Para os catálogos dos *users* e dos repositórios, a estrutura de dados consistia numa *hashtable* organizada por *ids* (*user_ids* e *repo_ids*, respetivamente). Já para o catálogo dos *commits*, esta seria ligeiramente diferente. Nas *queries* não estatísticas, seria útil conseguirmos percorrer de uma forma rápida todos os *commits* de um dado repositório. De forma a resolver este problema, concebemos a nossa *hashtable*, como uma tabela de listas ligadas de *commits* associados a um determinado *repo_id,* que seria a nossa chave. Conseguimos assim, percorrer facilmente ($\Theta(1)$) toda a lista de *commits* de um dado repositório.

Estes três módulos foram implementados de forma a funcionarem de forma independente, tendo em vista os objetivos de abstração, modularidade e encapsulamento dos dados e do código. Como tal, cada um deles fornece a respetiva *API*, com funções básicas de alocação de estruturas, pesquisa e inserção. De modo a guardar a informação necessária lida a partir de cada ficheiro, utilizamos as seguintes *structs*, guardadas nas *hashtables* dos seus catálogos correspondentes:

- user;
- repo;
- commit.

3.1 Struct user

Um *user* é composto pelo seu *login*, tipo e dois apontadores para *hashtables* que guardam os *ids* das listas *following* e *followers*. De notar que não possuímos um campo para o *id* do utilizador, visto que este será a chave da entrada na *hashtable* e, desta forma, conseguimos poupar 4 bytes. Decidimos, também, utilizar um char de forma a identificar o seu tipo, dado que utilizamos apenas 1 byte.

```
struct user{
    char *login;
    GHashTable *following;
    GHashTable *followers;
    char type;
};
```

Figura 1 - Struct de um user

3.2 Struct repo

Um *repo* possui campos para o *id* do criador, descrição e linguagem do repositório. Tal como na *struct user*, não possui um campo para o *id* do repositório, uma vez que vamos guardá-lo na chave de cada entrada da *hashtable*, e, portanto, poupamos memória utilizada.

```
struct repo{
    char *owner_id;
    char *description;
    char *language;
};
```

Figura 2 - Struct de um repositório

3.3 Struct commit

Um commit é constituído pelo seu *author id*, *committer id*, data em que foi realizado e tamanho da mensagem do *commit*. Aplicamos também a estratégia referida em cima para o *id* do repositório onde foi publicado o *commit*.

```
struct commit{
    char *author_id;
    char *committer_id;
    char *commit_at;
    int msg_ln;
};
```

Figura 3 - Struct de um commit

4. Módulos de dados adicionais

- **counter_hash.c:**
  Este módulo auxiliar serve de apoio à implementação das queries 5, 6, 8 e 9. É responsável pela implementação de *hashtables* com contadores associados a uma determinada estrutura. A inserção é feita na *hashtable* e a ordenação e consequente impressão é feita em listas ligadas. A transformação para listas ($\Theta(N)$) representa um custo computacional adicional neste problema. Contudo, este *overhead* é compensado pela rápida inserção na *hashtable.*

5. Parsing / Queries estatísticas

Na fase inicial do programa, fazemos um parsing dos 3 ficheiros fornecidos, de modo a guardar a informação útil nos catálogos de cada ficheiro e, ao mesmo tempo, calculamos as *queries* estatísticas (1 a 4) para tornar o código mais eficiente.

Mais concretamente, primeiramente, durante a travessia do ficheiro dos *users* procedemos ao cálculo do número de bots, organizações e utilizadores. Também inserimos todos os *ids* dos bots numa *hashtable* que nos irá ser útil para calcular mais tarde a *query* 3. Na travessia do ficheiro dos *commits*, conseguimos retirar o número de *commits*, útil para a query 4 e, ainda, podemos contar o número de colaboradores presentes. Já na última travessia do ficheiro dos repositórios, conseguimos concluir a *query* 3, uma vez que os *ids* dos bots se encontram guardados em memória, e extraímos o número de repositórios para a *query* 2.

6. Queries parametrizáveis

Para a resolução das *queries* parametrizáveis, decidimos utilizar uma estratégia muito semelhante em todas, que passa por recorrer a uma *hashtable* temporária para guardar informações úteis e, no final, inserir ordenadamente numa lista para devolver o *output* pedido. Optamos por usar esta estrutura temporária devido aos seus acessos rápidos e em tempo constante, facto que nos favorece, dado que em muitas das *queries* necessitamos de ir à *hashtable* para incrementar um valor. Não utilizamos árvores binárias, uma vez que têm tempos de acesso superiores em relação à *hashtable* e, como a cada vez que incrementamos, tínhamos de equilibrá-la, seria um processo muito custoso. Portanto, optamos por utilizar primeiramente a *hashtable,* seguida da inserção ordenada numa lista para depois devolver o *output*.

- **Query 5:**
  A *query* 5 é responsável por devolver o top N de utilizadores com mais *commits* num determinado intervalo de datas. Para tal, decidimos criar uma *hashtable* temporária em que cada entrada guarda uma *struct* que contém o contador de *commits* e o *login* do utilizador e é identificada pelo *id* do utilizador. Começamos, então, por percorrer o catálogo dos *commits*, através de uma função *for_each* que aplica uma função auxiliar a cada entrada da *hashtable*. Portanto, criamos a *q5_func* que vai ser aplicada a um apontador para o início de uma lista. Assim, vai verificar para cada *commit* da lista se está dentro do intervalo de datas e, em caso afirmativo, recorremos à função *increment* do módulo *counter_hash.c,* que trata de incrementar o número de *commits* na estrutura. Uma vez que não é possível ordenar uma *hashtable,* optamos por introduzir ordenadamente numa lista o resultado final da *hashtable*, com o objetivo de imprimir os N primeiros elementos desta lista.

- **Query 6:**
  Esta query pede o top N de utilizadores com mais *commits* em repositórios de uma determinada linguagem. Para tal, começamos por percorrer o catálogo dos repositórios, de modo a verificar se o repositório é da linguagem passada como parâmetro à *query*. Em caso afirmativo, vamos procurar os *commits* associados a este repositório ao catálogo dos *commits*. Portanto, à semelhança da *query* 5 criamos uma *hashtable* temporária, com o objetivo de guardar numa *struct* os utilizadores e

vamos incrementando a quantidade de *commits* de cada um. No final, recorremos ao módulo *counter_hash.c* para inserir de forma ordenada as *structs* guardadas na *hashtable* e, posteriormente, devolver os N utilizadores com mais *commits*.

- **Query 7:**
  Nesta *query* era pretendido obter a lista de repositórios sem *commits* a partir da data do *input*. Para isto, criamos uma *hashtable* para guardar os dados necessários para a impressão no ficheiro de *output*: descrição e o *repo_id* dos repositórios inativos a partir dessa data. Aplicando uma função *for_each* com a função auxiliar - que contém as operações necessárias para a *query* - ao catálogo dos repositórios, para cada repositório fazemos a pesquisa de *commits* no catálogo dos *commits* referentes a esse *repo_id*. Com isto, se não existirem *commits* então adicionamos a informação desse repositório à *hashtable* final. Se existirem, então verificamos se são ou não depois da data (através da função *check_commit_after*): caso sejam, também adicionamos essa informação do repositório à *hashtable* final. Por fim, imprimimos os dados presentes na *hashtable*.

- **Query 8:**
  A resolução desta *query* consistia em apresentar o top N de linguagens mais utilizadas a partir de uma determinada data, obtida no *input*.
  Assim, utilizamos, mais uma vez, uma *hashtable* para armazenar os dados e a função *for_each*, que nos permite aplicar uma função que recolha a linguagem e procure *commits* associados a esse *repo_id* a cada entrada do catálogo dos repositórios. Se não existirem *commits* ou a linguagem do repositório for "None", passamos para a verificação da próxima entrada da *hashtable*. Caso exista algum *commit* associado, verificamos se esse mesmo *commit* é posterior à data do *input* e, caso isto se confirme, colocamos os dados na *hashtable* final e incrementamos o contador dessa linguagem, recorrendo ao módulo *counter_hash*.
  Por fim, e uma vez que devemos apresentar o top N, passamos os dados de cada entrada da *hashtable* para uma lista e procedemos à impressão das N linguagens mais utilizadas.

- **Query 9:**
  Esta *query* pedia o top N de utilizadores com mais *commits* em repositórios cujo owner_id está contido quer na lista de *followers*, quer na lista de *following* do *user* que faz o *commit*. A resolução desta *query* passou por uma travessia no catálogo dos *commits*, com vista a incrementar numa *hashtable* o contador referente a esse utilizador. Para cada *commit*, caso o dono do repositório associado pertencesse à lista de *following* e *followers* do *committer_id/author_id*, o número de commits desse utilizador seria incrementado na *hashtable*. De modo a apresentar os N utilizadores com mais *commits* nesses repositórios, recorremos ao módulo *counter_hash*, responsável pela ordenação e output do top N de utilizadores.

- **Query 10:**
  Na *query* 10 eram pedidos o N utilizadores com as maiores mensagens de *commit* em cada repositório. Mais uma vez, nesta *query* fica evidenciada a importância de ter todos os *commits* do mesmo repositório de uma forma contígua na mesma estrutura. Para além disso, de forma a otimizar a impressão da informação relevante para o ficheiro de *output*, a lista dos *commits* já se encontra ordenada de forma decrescente de acordo com o tamanho da mensagem de *commit*. Limitamo-nos assim a imprimir apenas os N primeiros *users* (*owner* e/ou *commiter*) de cada repositório. Uma dificuldade encontrada na implementação desta *query* teve a ver com o encapsulamento da estrutura de dados dos *commits*. Como a travessia da lista para efeitos de impressão necessitava de dados do catálogo dos *users*, duplicamos a lista dos N primeiros *users* de cada repositório, ordenados pelo tamanho da mensagem. Tal esforço computacional é necessário de modo a manter o encapsulamento do catálogo dos *commits*.

7. Testes ao código

De modo a testar a funcionalidade das queries, a eficiência do código e a libertação da memória alocada, foram realizados vários testes com datasets mais pequenos. Estes testes tentaram cobrir todos os possíveis casos, incluindo os menos frequentes (por exemplo, committer id diferente de author id). Mais uma vez, foi utilizada a ferramenta valgrind, de forma a identificar e corrigir eventuais *memory leaks* do código.

```
==12672== HEAP SUMMARY:
==12672==     in use at exit: 18,612 bytes in 6 blocks
==12672==   total heap usage: 25,850,625 allocs, 25,850,619 frees, 567,071,424 bytes allocated
==12672==
==12672== LEAK SUMMARY:
==12672==    definitely lost: 0 bytes in 0 blocks
==12672==    indirectly lost: 0 bytes in 0 blocks
==12672==      possibly lost: 0 bytes in 0 blocks
==12672==    still reachable: 18,612 bytes in 6 blocks
==12672==         suppressed: 0 bytes in 0 blocks
==12672== Reachable blocks (those to which a pointer was found) are not shown.
==12672== To see them, rerun with: --leak-check=full --show-leak-kinds=all
```

Figura 4 - Valgrind do programa a executar todas as queries

8. Grafo de dependências

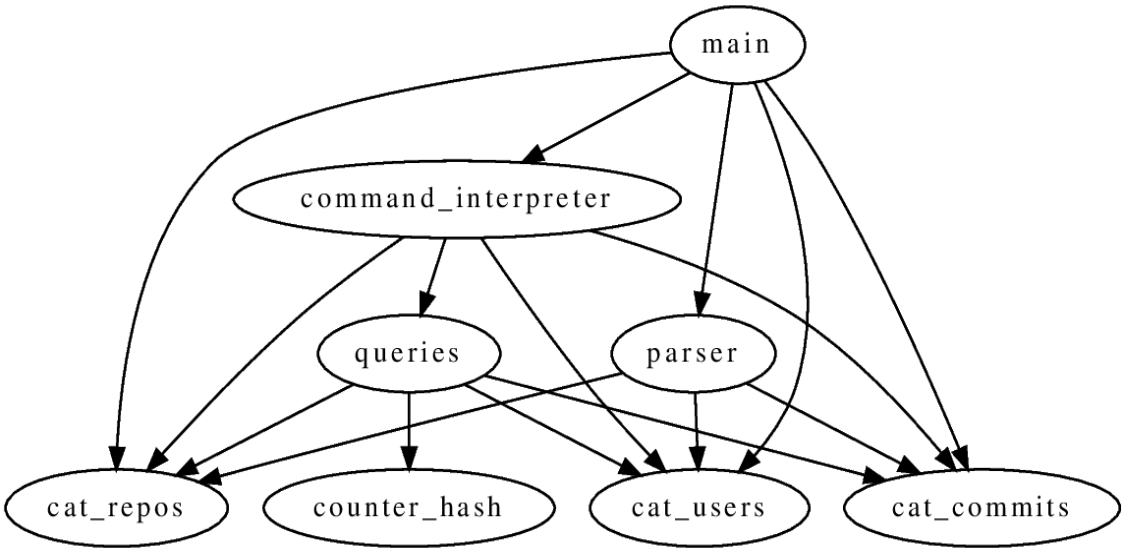


Figura 5 - Grafo de dependências

Este grafo mostra as dependências existentes entre os vários ficheiros do nosso trabalho, numa estrutura onde todos os ficheiros das *queries* se encontram abstraídos num único nodo do grafo. Como podemos ver, os catálogos são totalmente independentes, uma vez que não há nenhuma seta a sair dos nodos dos catálogos. Também podemos notar que as *queries* estão apenas dependentes do módulo de apoio *counter_hash* e dos catálogos.

9. Conclusão

Com este trabalho, fomos levados a explorar vantagens e desvantagens de diferentes estruturas de dados, com o objetivo de responder da forma mais eficiente às *queries* pedidas. Temos noção de que não há soluções perfeitas e que vantagens de uma certa estrutura para uma *query*, pode representar desvantagens para outras *queries*. Assim, a nossa implementação constitui um compromisso entre as várias alternativas possíveis.

Para além disso, ganhamos mais sensibilidade para aspetos do código como encapsulamento, modularidade e abstração. Entendemos a importância de escrever código mais seguro, mais robusto e mais fácil de manter, pois isso irá poupar trabalho mais tarde, seja a resolver erros, seja a adicionar novas funcionalidades.