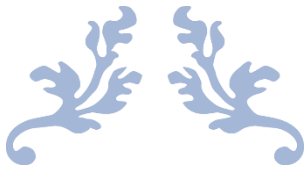

LABORATÓRIOS DE INFORMÁTICA III

Relatório Guião 1, Grupo 73

8 DE NOVEMBRO DE 2021

GUILHERME SAMPAIO, A96766
TIAGO OLIVEIRA, A97254
MIGUEL NEIVA, A92945

Licenciatura em Engenharia Informática

Conteúdo

1. Introdução

No âmbito da unidade curricular Laboratórios de Informática III, o presente relatório tem como objetivo apresentar os métodos de trabalho usados no Guião 1 do projeto e expor certos problemas que possam ter condicionado o desempenho do programa e, além disso, também os procedimentos e técnicas usadas para colmatar esses mesmos defeitos.

Deste modo, este guião tinha como primeira meta criar um programa na linguagem C que lesse determinados ficheiros (formato CSV) e descartasse registos que não estivessem de acordo com o que era pedido. Outra das finalidades desta parte do trabalho era o cruzamento dos ficheiros resultantes do primeiro programa, já descrito anteriormente, e filtrar dados inválidos a partir dessa interseção.

No entanto, surgiram alguns problemas como uso excessivo de memória e otimização de código.

2. Guião 1

2.1 Exercício 1

2.1.1 Enunciado e explicação.

O exercício 1 consiste em analisar os dados presentes nos ficheiros que nos foram apresentados gerando, assim, um ficheiro de saída contendo apenas os dados válidos. Dados estes, que são considerados válidos de acordo com algumas condições enunciadas.

2.1.2 Planeamento e estratégias a usar.

Com um pensamento no futuro, inicialmente, decidimos por optar por uma validação baseada em estruturas de dados, isto é, existe uma estrutura (*User, Repo, Commit*) para cada ficheiro de entrada, que possuem um parâmetro booleano de verificação (*is_valid*) que nos vai indicar se o registo é válido ou não.

Para além disso, no *parsing* das linhas dos ficheiros optamos por utilizar funções como *strdup* e *strsep*.

Mais tarde, apercebemo-nos (demasiado tarde) que esta abordagem não foi deveras a melhor e é discutida no ponto 2.1.3 desta secção.

Ou seja, de uma maneira geral, para todos os ficheiros, foi criada uma função principal que itera pelas linhas do ficheiro aplicando-lhes uma função booleana secundária, mas importante, que nos validava a linha ao verificar o parâmetro *is_valid* de cada estrutura.

É uma abordagem simples, porém não executada da melhor forma.

2.1.3 Problemas e resolução.

Este exercício embora pareça simples, ainda tem algo que se lhe diga, no caso a gestão de memória e a otimização do código.

Começando pelos problemas de memória, por ter sido usada a *strdup* para a verificação de todos os campos de uma linha, bastante memória ficava alocada, mesmo dando *free()* sempre no final da execução da função.

Este é um problema e o problema que deitou fora toda e qualquer chance de pontuação, pois para inputs significativamente maiores o programa não só demora mais tempo a correr como acaba por ocupar a *RAM* inteira sendo forçado o encerramento do mesmo.

Uma resolução para este problema seria não utilizar a *strdup* assim como a *strsep*, trocando-as por apenas a *strtok_r*. A vantagem da *strtok_r* é que nos dá a possibilidade de guardar o resto das linhas mesmo após esta ser dividida no nosso *token*. Desta maneira, não seria preciso alocar memória para uma cópia das linhas antes de a passar pela strsep.

Em relação à otimização do código, acredito que hajam maneiras de tornar o código muito mais eficiente e menos pesado para a memória do computador assim como para o *CPU*. Conseguiríamos alcançar este objetivo se a estratégia usada fosse a descrita acima (uso do *strtok_r*).

2.2 Exercício 2

2.2.1 Enunciado e explicação.

O exercício 2 consistia em cruzar alguns dos dados dos ficheiros gerados pelo exercício 1 de modo a, mais uma vez, eliminar os registos inválidos, neste caso, apenas registos do tipo *commits* e *repos*.

Para isso foram-nos dadas 4 condições que determinam se um registo é inválido ou não:

1. Remoção de *commits* que refiram utilizadores (*committer_id*, *author_id*) não existentes.
2. Remoção de *commits* cujo repositório não exista.
3. Remoção de repositórios que refiram utilizadores não existentes.
4. Remoção de repositórios que não contenham qualquer *commit*.

Assim, traduzindo em procedimentos, temos:

1. Comparar o *committer_id* e *author_id* (*commits.csv*) com os *ids* (*user.csv*), removendo sempre os que se encontram nos *commits*;
2. Verificar se o *repo_id* (*commits.csv*) existe nos *ids* (*repos.csv*);
3. Comparar o *owner_id* (*repos.csv*) com os *ids* (*users.csv*), descartando sempre os repositórios;
4. Procurar pelos *ids* (*repos.csv*) nos *repo_id;*

E foi por estes procedimentos que a nossa implementação deu frutos.

2.2.2 Planeamento e estratégias a usar.

A primeira coisa que nos surgiu à cabeça assim que formulamos as instruções mencionadas no ponto 2.2.1 foi que, para podermos comparar estes dados todos, de maneira eficiente, temos de os guardar em algum lado.

Como tal começamos por procurar uma estrutura de armazenamento de dados que tivesse tanto um tempo baixo de inserção, como de procura. Destas estruturas as que mais se fizeram sentir entre nós foram:

- Árvore AVL
- *Hash Table*

Decidimos utilizar *hash tables* na resolução do exercício. **Porquê?**

Em primeiro lugar, nas *hash tables* a "*memory footprint*" é baixa, sendo que, apenas as inserções que colidem é que precisam de um *pointer*, enquanto numa árvore AVL são precisos 2 por inserção. Outra vantagem é que o tempo de procura numa *hash table* é esperado ser O(1), ou seja, constante contrariamente a uma árvore AVL onde é O(log n), sendo deste modo, significativamente mais alto.

Isto não quer dizer de todo que uma árvore AVL é ineficaz, simplesmente não se adequa a este paradigma, e seria útil, por exemplo, num caso de procura de dados ordenados.

(Depois de termos decidido a estrutura (*GHashTable - glib2.0*) surgiu o nosso primeiro problema que é discutido no ponto 2.2.3 Problemas, resolução. Resumidamente, caso existisse mais de uma *hash table* no programa, os valores eram perdidos / não adicionados.)

Mas como é que vamos fazer a leitura dos dados e saber o que escrever nos ficheiros de saída?

Como já foi assinalado as instruções definidas no ponto 2.2.1 foram seguidas à risca, portanto a nossa estratégia foi a seguinte:

I. Guardar os *ids* do ficheiro *users-ok.csv*
II. Guardar os *ids* do ficheiro *repos-ok.csv*:
III. Comparar e eliminar registos inválidos no ficheiro commits-ok.csv:
IV. Comparar e eliminar registos inválidos no ficheiro repos-ok.csv:

Desta maneira, conseguimos chegar ao resultado pretendido sem fazer travessias pelos ficheiros desnecessárias, sem gastar enormes quantidades de memória e de uma maneira organizada e limpa.

2.2.3 Problemas e resolução.

O exercício 2, relativamente ao exercício 1, foi muito mais acessível. O único problema que encontramos foi inicialmente ao tentar introduzir *hash tables* no programa.

O problema: Sempre que mais do que uma *hash table* era criada no programa, a *hash table* anterior perdia a "forma", isto é, os valores ou não eram adicionados corretamente, ou todos os valores tomavam o valor do último elemento a ser adicionado.

Logo à partida este mistério gritava *"scope",* já que a *hash table* recebe os *pointers* dos elementos e não os elementos em si. Após muito tempo (aproximadamente uma semana) descobrimos como o resolver.

A resolução: De modo a não perder os elementos da *hash table*, introduzimos um *array* auxiliar que, de certa forma, vai "transportar" os elementos ao longo do código. Então, ao ler os parâmetros desejados do ficheiro, primeiro adicionávamos o *pointer* do elemento ao *array*, e seguidamente adicionávamos o endereço do elemento no *array* à *hash table*.

Desta forma, podemos adicionar quantas *hash tables* ao código quantas quisermos sem ter de preocupar com o seu conteúdo.