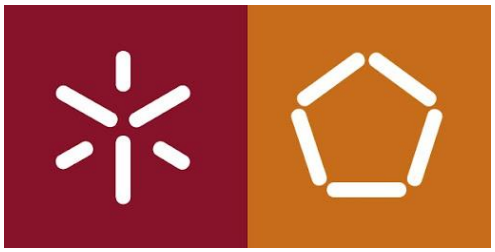
Universidade do Minho
Licenciatura em Engenharia Informática



LABORATÓRIOS DE INFORMÁTICA III

Guião 1

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

Nesta fase do trabalho de Laboratórios de Informática III foi-nos proposto implementar um parser em C que validasse as linhas dos 3 ficheiros, com o formato *csv,* que nos foram disponibilizados: commits, users e repositórios do GitHub. Para complementar esta validação, tivemos ainda de efetuar o cruzamento entre os ficheiros, removendo as linhas incompatíveis. Assim, focamo-nos, principalmente, na modularidade e eficiência do código, na abstração e encapsulamento das estruturas de dados e na utilização reduzida de memória.

2. Procedimento

Com o objetivo de dar resposta ao primeiro problema, começamos por implementar um parser para cada ficheiro que validasse cada linha separadamente, utilizando um contador para a identificação do campo a validar. Como certos campos eram semelhantes em termos de forma de verificação, criamos várias funções, com vista a validar cada tipo diferente de campo (inteiros, strings, listas, datas, bool e tipo). No entanto, esta implementação precisava de 3 funções (uma para cada ficheiro), visto que os ficheiros tinham estruturas diferentes a nível dos campos. Para resolver este entrave, procedemos à criação de uma função que abstraísse todos os ficheiros (f. *parse*). Esta função recebe uma lista de apontadores para as funções validadoras já criadas anteriormente. Este tipo de abordagem contribuiu, a nosso ver, para uma melhor organização, abstração e modularidade do código.

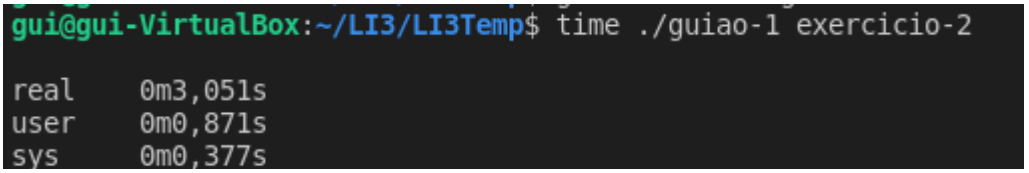
Ainda neste primeiro problema, tínhamos utilizado a biblioteca *time.h* para recolher a data e o fuso horário do sistema. Todavia, apercebemo-nos de que esta abordagem não estava a ser eficiente, resultando num elevado tempo de execução do sistema. Decidimos então deixar o limite superior para a data como sendo a data da entrega do guião (10-11-2021).

Após a validação das linhas de cada ficheiro, prosseguimos com o *cross-checking* dos *ids* dos diferentes ficheiros. Ao contrário do primeiro exercício, neste tivemos de armazenar a informação numa estrutura de dados em memória. O principal desafio desta fase foi a escolha de uma estrutura de dados eficiente em termos de tempo e espaço.

A nossa primeira abordagem consistiu em recorrer a arrays desordenados, utilizando pesquisa linear para verificar a existência dos ids nesse array. Dois problemas surgiram: requerimento de um elevado espaço de memória contígua para armazenar as N entradas do array e um elevado tempo de execução.

1. Tempo de execução utilizando procura linear num array desordenado

Face a este problema decidimos implementar um método de pesquisa binária num array previamente ordenado. Para isso, recorremos à função *qsort* da biblioteca *stdlib.h.* Houve um melhoramento considerável em termos de tempos de execução, no entanto o problema da memória persistia.

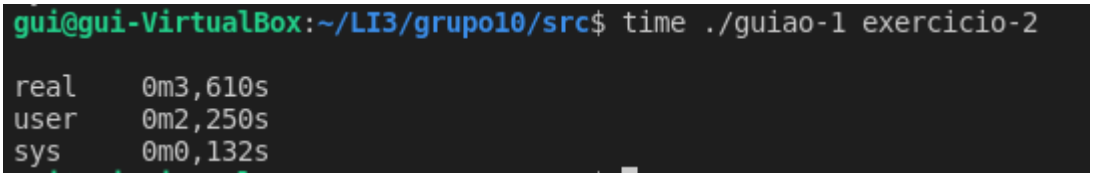


2. Tempo de execução utilizando procura binária num array ordenado

Então, decidimos utilizar uma estrutura de dados não contígua que fosse, contudo, eficiente na procura dos elementos. A escolha recaiu nas árvores binárias balanceadas (AVLs). Temos noção de que, em termos de criação da estrutura, esta acaba por ser menos eficiente devido à carga do balanceamento. No entanto, acreditamos que a longo prazo acabaríamos por ter proveitos, devido à rápida procura de complexidade temporal $O = (log\ N)$, uma vez que a cada iteração eliminamos (para efeitos de pesquisa) um dos lados da árvore.

Outras otimizações foram inseridas ao longo do desenvolvimento do projeto. Em primeiro lugar, na validação do ficheiro "commits-ok" com base nos users válidos, verificamos em primeiro lugar a igualdade dos *author_id* e *committer_id*. Deste modo, pudemos evitar mais uma pesquisa na árvore dos ids dos *users*, caso fossem iguais. Para além disso, à medida que fazíamos esta passagem, íamos inserindo os repositórios que continham qualquer tipo de commit numa árvore binária, com o objetivo de, seguidamente, filtrar o "repos-ok". Deste modo, poupamos uma passagem extra no ficheiro "commits-ok".

Obtivemos, então, os seguintes tempos de execução:



3. Tempo de execução utilizando AVL trees

Uma outra possibilidade analisada foi a utilização de *hash tables* para o armazenamento dos dados a cruzar. Esta acaba por ser uma estrutura ligeiramente mais eficiente do que as AVL, visto que a complexidade temporal é $\mathcal{O}(1)$ - pesquisa através da hash - contudo, é também mais propícia a colisões. Assim, decidimos não alterar o código em árvores binárias, por este estar razoavelmente eficiente e por desconhecermos, na íntegra, o que nos será proposto futuramente no trabalho.

3. Ferramentas Utilizadas

Para ajudar no desenvolvimento e na correção de erros do código, utilizamos três ferramentas auxiliares: valgrind, gprof e gdb. Através do primeiro, conseguimos avaliar se o nosso programa estava a alocar e a libertar corretamente a memória necessária. Com o segundo, analisamos que funções estavam a ocupar a maior fatia dos recursos computacionais gastos pelo nosso programa. Com a terceira, conseguimos resolver grande parte dos erros funcionais do nosso código. Estas três ferramentas foram fulcrais para o desenvolvimento do nosso programa.

```
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 5,664,142 allocs, 5,664,142 frees, 333,913,832 bytes allocated

All heap blocks were freed -- no leaks are possible
```

4. Resumo da *heap* no exercício 1

```
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 4,663,436 allocs, 4,663,436 frees, 489,743,187 bytes allocated

All heap blocks were freed -- no leaks are possible
```

5. Resumo da *heap* no exercício 2

4. Conclusão

Em jeito de conclusão, neste trabalho ficou evidenciado a importância de uma boa eficiência na pesquisa de valores em estruturas. Através da comparação dos tempos de execução, conseguimos ponderar entre diferentes alternativas, escolhendo a que mais se adequa aos nossos objetivos. Para um projeto em grande escala, com milhões de entradas em cada ficheiro, torna-se imprescindível a boa organização e gestão da memória.