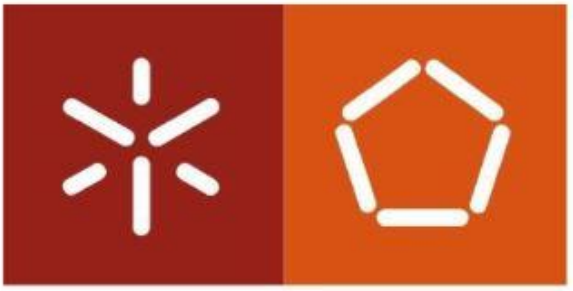


UNIVERSIDADE DO MINHO

MIEI/ LEI+MEI

UC Laboratórios de Informática *III* - 2021/2022

| Grupo 11

Luís Fernandes (A88539)

David Duarte (A93253)

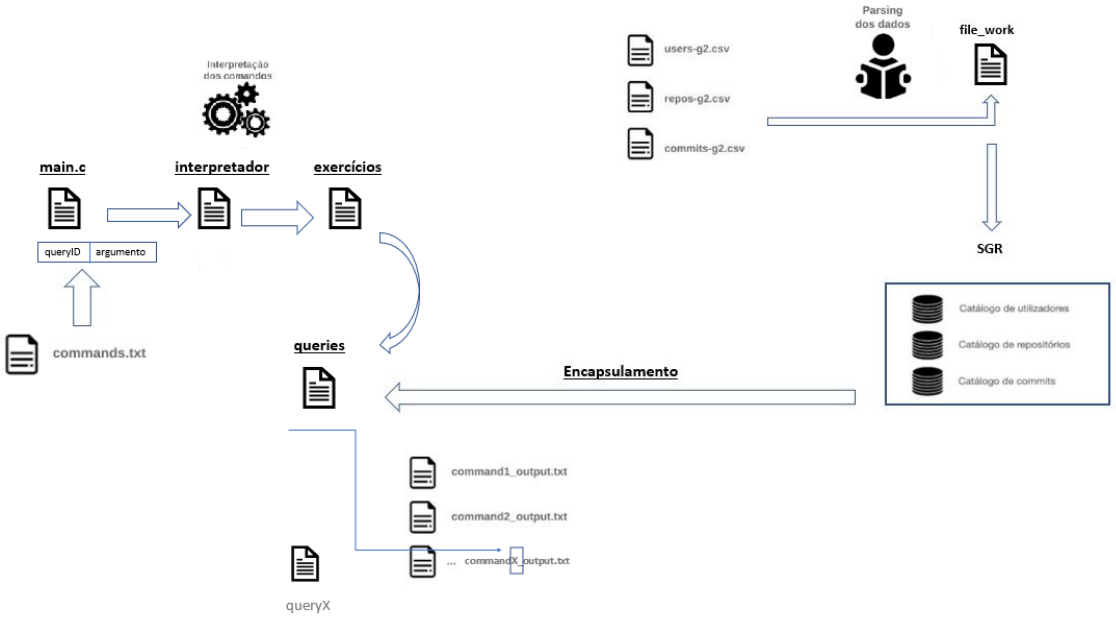
João Castro (A97223)

Introdução

O desenvolvimento do guião 2, no âmbito da UC de Laboratórios de Informática III, teve como principais objetivos a aplicabilidade de conceitos como a modularidade e encapsulamento. Este guião, por outro lado, visava na implementação de estruturas de dados dinâmicas, e a definição de estratégias tendo em mente o custo computacional das mesmas. O trabalho em causa permitiu-nos também a aumentar o nosso conhecimento em relação à linguagem C e entrar em contacto com ferramentas destinadas para correr e compilar programas como a make.

Arquitetura da Aplicação

A nossa aplicação está a ser organizada em MVC (Model/View/Controller), de modo a que o nosso código esteja separado de acordo com o tipo de trabalho que o mesmo irá realizar. Deste modo, o código correspondente ao parsing de dados, implementação das nossas estruturas de dados e queries encontra-se na diretoria Model.
Na diretoria Controller é onde se encontram os ficheiros responsáveis pela interpretação dos comandos.



Estruturação da arquitetura da nossa aplicação

Como podemos observar na figura acima, a leitura e tratamento dos dados dos três ficheiros é realizada pela ‘file_work.c’, sendo que a nossa struct SGR irá conter os três catálogos. Estes catálogos irão conter os registos dos ficheiros validados e outras estruturas de dados que serão essenciais para a implementação das queries através do processo de encapsulamento. Por outro lado, os nossos ficheiros ‘main.c’,‘interpretador.c‘ e ‘exercicios.c’

irão proceder à interpretação dos comandos dado pelo ficheiro ‘commands.txt’ de modo a que sejam executadas as respetivas queries com os seus argumentos(se tiverem).

Descrições das API’s & estratégias adotadas

Num primeiro momento, criamos a nossa struct SGR com os 3 catálogos e procedemos à sua inicialização. Após essa fase a nossa função ‘fill_sgr’ irá chamar as funções de parsing no ficheiro file_work.c de modo a ler os dados dos ficheiros de entrada.

```
struct sgr{
   CCOMMITS catalogo_commits;
   CREPOS  catalogo_repos;
   CUSER catalogo_users;
};
```

Struct SGR

Em ler_file_user para além de definirmos um apontador de leitura e um buffer de modo a fazer o parsing de cada linha de users-g2.csv, decidimos definir como estratégia criar variáveis que contassem cada type de utilizador. Assim, por cada user lido dependendo do tipo de user que se tratava incrementávamos nas respetivas variáveis(get_Type-por encapsulamento). Ao mesmo tempo que líamos cada user adicionávamos um user à nossa Hashtable de users do catálogo de users. No final da função devolvemos, desta maneira, o catalogo_commits.

```
struct catalogo_user{
    GHashTable *GUser;
    int bots;
    int org;
    int user;
};
```

Struct catalogo_user

```
typedef enum type{
    BOT = 0,
    ORGANIZATION = 1,
    USER = 2,
    INVALID = -1
}TIPO;
```

Tipos de dados de cada utilizador

Como podemos constatar a nossa struct de catalogo_user é constituída por uma Hashtable de users e com o número de cada tipo de utilizador. Assim, através da técnica de encapsulamento, obtínhamos o número de cada type de user e escrevíamos num novo ficheiro txt de acordo com o índice da query em causa(query1->command1.txt).

Em ler_file_commits definimos novamente um apontador de leitura e um buffer de modo a realizar o parsing de dados do file commits-g2.csv. Como o header não nos interessava, fizemos um primeiro fgets. Caso o apontador fosse NULL então devolveria NULL, caso contrário, o nosso buffer2 teria uma cópia do buffer, e o commit teria a estrutura de uma linha de commit. Em seguida, verificámos se o user e se o id do repositório se

encontravam na hashtable de users construída anteriormente através das funções pré-definidas da glib (GContainsKey). Na hipótese das condições serem válidas adicionávamos o commit num GArray.

O nosso grupo chegou à conclusão de que não seria o mais indicado construir uma hashtable de commits. Isto deve-se ao facto do elevado número de colisões que as operações de procura poderiam acarretar. Um ficheiro de commits pode ter vários commits do mesmo user, pelo que usar hashtable que possuísse como chaves os repos_id não seria muito prudente.

Assim, de modo a resolver as colisões, caso o user ou o repos_id existissem na hashtable, procedíamos à construção de um novo GArray através do get do array de commits já feitos na hashtable de users de um dado commiter_id. Assim, colocávamos o commit em causa nesse array e colocávamos esse array na hashtable de users, por outras palavras, pegávamos no garray que estava lá dentro, adicionávamos o novo commit e voltávamos a inserir na hashtable(funções - set_CC_GHashTable e GHAddCommit). Caso, não existisse era criado um garray, consequentemente, o commit seria adicionado no garray, e finalmente, o garray seria inserido na hashtable com a chave indicada.

De modo a garantir que a contagem do número de repositórios contendo bots como colaboradores fosse correta, temos de verificar se já existia algum commit feito pelo utilizador em causa através do commiter_id, de modo a contabilizar apenas um repositório em que o utilizador é bot. Se esse user não existisse e se o seu type fosse bot, adicionariamos +1 a nbots. Caso se tratasse de um user a variável nusers era incrementada. No sentido de calcular o valor de repositórios, apenas uma vez, já que o mesmo repo_id podia aparecer em várias linhas no ficheiro commits.csv, tivemos de proceder à verificação de já existir algum commit feito ou não através da função GContainsKey dado pela chave repo_id e pela hashtable de repos. Se ainda não existir, adicionámos mais um a nrepos.

. Por cada linha de commits que adicionámos à GArray, incrementávamos o ncommits, que correspondia no fundo ao número de linhas do ficheiro(total de commits).

```
struct catalogo_commits {
    GArray *GCommits;
    GHashTable *GUsers;
    GHashTable *GRepos;
    int nrepos;
    int nusers;
    int nbots;
    int total_commits;
    GHashtable *hashOcor;
};
```

Struct catalogo_commits

Deste modo, a nossa struct de catalogo_commits possui um GArray onde se encontram os commits válidos, e uma hashtable de users e repositórios. Por outro lado, possuímos 4 variáveis: nrepos- número total de repositórios; nusers- número total de utilizadores; nbots-número total de utilizadores de tipo bot; total_commits-número total de commits.

Através do encapsulamento realizámos os gets necessários nas queries de modo a extrair os dados estatísticos pedidos.

Para além disso, podemos averiguar que a nossa struct catalogo_commits possui uma hashtable ‘hashOcor’. Esta hashtable apresenta como chaves os commiter_id de cada commit presente no ficheiro csv. No entanto, é imporante referir que esta hashtable não apresenta a mesma chave mais do que uma vez, já que isso iria conduzir novamente a um número elevador de colisões(visto que um utilizador pode realizar mais do que um commit). Desta forma, por cada commit bem construído, para cada chave seria incrementado o número de ocorrências, que no fundo corresponderia ao número de commits de cada utilizador.

O nosso grupo chegou à conclusão que a implementação de estruturas de dados como Hashtables seria uma mais-valia para a realização deste projeto. No que toca à análise de complexidade, sabemos que as hash tables possuem um tempo de procura constante(run time complexity : O(1)). Portanto, num pior caso estaríamos a falar de um tempo linear, algo que é muito raro de acontecer(run time complexity : O(N)). Por outro lado, a biblioteca da glib, onde podemos recorrer de funções já pré-definidas.

Porém, também é importante referir que as hash tables consomem muita memória, até mais do que é necessário. Porventura, teríamos outras estruturas de dados como árvores binárias que são mais eficientes a nível de memória, e que apenas alocam espaço que é necessário para os nodos. Numa outra perspetiva, se quiséssemos extrair os valores num dado intervalo de keys isso seria impossível em hash tables, sendo que seria algo concretizável através das árvores binárias. Contudo, árvores binárias correm em tempo logarítmico, o que acaba por demorar mais tempo. Ainda que esse aumento fosse mais lento, a implementação e uso de árvores binárias na altura pareceu-nos pouco adequada a este trabalho. Assim, como podemos concluir, existem sempre pontos que são mais vantajosos e outros não, pelo que o nosso grupo optou pela escolha que achou ser mais adequada no momento.

Struct catalogo_repos

Na figura representada em cima vemos que a nossa struct catalogo_repos é constituída por uma hash table de repositórios e por uma hash table que por cada key correspondente a uma string(tipo de linguagem) conta em quantos repositórios a mesma é utilizada. Esta hash table GRepos é construída verificando previamente se o utilizador dado pela chave owner_id se encontra na hashtable de users.

Como é possível observar no nosso projeto, também recorremos de outras técnicas, nomeadamente, a escrita em ficheiros auxiliares. Nesta técnica, enquanto era feito o parsing dos dados, extraímos os dados que achamos relevantes, como por exemplo, na função ‘ler_file_commits’ em que temos o nosso pointer escritaAux para o ficheiro objeto commits-aux, onde são escritos os commiter_id.