Relatório Guião 1- Laboratórios de Informática III

I-Introdução

Este primeiro guião está separado em dois exercícios, sendo que o primeiro tinha como objetivo a eliminação dos registos que não respeitem os formatos indicados e consequentemente gerar os ficheiros que contenham apenas os registos válidos. Por sua vez, o segundo tinha como propósito o cruzamento de dados entre os vários ficheiros, gerados no exercício 1, de forma a filtrar dados considerados inválidos, produzindo assim os ficheiros finais.

II-Exercício 1

A fim de respeitar os requisitos impostos para o exercício 1, nomeadamente verificar que em cada um dos campos de cada ficheiro, a informação estava correta. Para tal, decidimos fazer funções de *parsing* de cada linha para cada um dos três ficheiros originais, que utilizando uma variável k inicializada a 1, avaliava a validade de cada token, criado através da função *strsep*, em que caso o token não cumprisse os requisitos impostos, esta variável k passava a valer 0. A partir do momento em que *k* se torna 0, a linha é eliminada e é realizado o *parsing* da linha seguinte.

Começamos, então, pela validação dos campos do ficheiro *users.csv* uma vez que era o mais semelhante ao que ficheiro que tínhamos estudado na aula anterior, em seguida validamos os ficheiros *commits.csv* e *repos.csv*. Para cumprir os critérios impostos para cada ficheiro, começamos por validar se os campos pela ordem que apareciam na linha.

De forma geral, para os campos em que nos era exigido que fossem inteiros não negativos, ao longo dos três ficheiros, fizemos a função *isNumber* e *checkPos* que com o auxílio da função pré-definida *atoi* (definida na biblioteca *stdlib.h*). Em seguida, de forma a validar os campos que deveriam estar no formato string, havia dois tipos de critérios: em alguns deles o único requisito era que a string fosse não nula e para tal criámos a função *checkString* que descartava os casos em que o campo fosse NULL, '\0' ou '\n'; enquanto que no segundo caso o campo só poderia ter determinadas strings, em que tivemos de criar algumas funções, *checkT* e *isBool*, que utilizando a função *strcmp* (definida na biblioteca *string.h*) avaliavam a sua validade.

Para validar o campo de created_at, devido ao facto de apresentar o mesmo formato em todos os ficheiros, usámos uma função geral denominada *checkData*, que retorna 1 caso seja válida e 0 caso contrário. Para esta função, nós primeiramente seguimos um processo que consistia em validar elemento a elemento da string, mas depois como o código ficava um pouco complexo de se perceber, optamos por criar um struct que armazena 6 inteiros para cada componente da data (ano, mês, dia, hora, minuto, segundo) e com isto criamos uma função converte que a string numa struct deste tipo (*stringToData*). Caso a conversão seja possível, analisamos os respetivos campos para avaliar se cumprem os critérios. Começamos por analisar os casos mais gerais e à medida que avançamos aumentamos a especificidade: em primeiro verificamos as condições de falha mais simples: como é o caso de o dia ser igual a -1, o mês não ser um inteiro entre 1 e 12, a data não ser posterior a 7 de abril de 2005, em que caso se verificasse a função retorna 0 de forma imediata não. Em seguida, verificamos que os meses do ano não poderiam ter mais dias do que é suposto, bem como o caso especial dos anos bissextos. Averiguamos que o campo das horas tem de conter um inteiro entre 0 e 23 e os

minutos e os segundos devem ser um inteiro entre 0 e 59. Por fim, fomos verificar a condição de que a data não deve ser superior à atual e, para tal, criámos uma função denominada *getTime* que obtém a data atual (com recurso à função *ctime* da biblioteca *time.h*) e retorna o resultado no formato desejado através da função *separateTime*. Por outro lado, caso não seja possível converter a string para esta estrutura de dados, ou seja, um dos parâmetros não seja número inteiro, a função retorna NULL e automaticamente o nosso programa considera a linha a ser analisada como inválida.

É importante salientar, que no caso do ficheiro repos.csv foi necessário criar uma função (*calcDifDatas*) que verifica que o campo *updated_at* não corresponde a uma data inferior à do *created_at*.

No ficheiro *users.csv* também foi necessário verificar nos campos respetivos aos *followers* e *following* que para além de serem obrigatoriamente inteiros não negativos, correspondem ao tamanho das listas *follower_list* e *following_list*, respetivamente. Para confirmar que estes números eram os mesmos, recorremos à função *checkList* que tem como argumento a string das diferentes listas e o inteiro que correponde ao campo de follower ou following. Já dentro desta função, convertemos a string da lista numa struct que criámos chamada *Array_f* (que tem um campo para um array de inteiros - *array_f* e um campo para um inteiro correspondendo ao seu tamanho - *size*) através da função *stringToArray* e depois verificamos apenas se o inteiro que foi passado como argumento corresponde ao valor de array_list.size. Caso esta condição não se verifique, a função retorna 0 e a linha é removida.

III-Exercício 2

Para o exercício 2, decidimos começar por criar uma estrutura de dados, chamada *Array_IDS*, que armazenaria os *ids* específicos dos ficheiros resultantes do exercício 1, num conjunto de três arrays dinâmicos de inteiros, dependendo do ficheiro, para além dos três array em si, a estrutura também armazena o tamanho de cada um destes arrays. O campo *id_users* armazena os ids presentes no ficheiro *users-ok.csv*, o campo *id_repos* armazena os ids dos repositórios do ficheiro *repos-ok.csv* e por último, o campo *id_commits* guarda os ids dos repositórios do ficheiro *commits-final.csv*.

Em seguida, partimos para a obtenção dos elementos do campo *id_users* no ficheiro *users-ok.csv* usando a função *getUsersID* e ordenamos esse array com a função *qsort* definida na biblioteca *stdlib.h* que utiliza como auxiliar a função *cmpfunc*, esta função compara o valor de dois *void pointers*, convertendo-os em *integer pointers*, subtraindo-os.

Fizemos o mesmo procedimento para o campo *id_repos* mas no ficheiro *repos-ok.csv*.

Neste ponto, começamos a filtrar as linhas do ficheiro *commits-ok.csv* usando a função *verifyCommitID*, através da função *verifyIDLine*, que recebe como parâmetros a linha a filtrar, uma linha auxiliar que é igual ao estado inicial da linha a filtrar e uma variável com o tipo *Array_IDS*. A função *verifyIDLine* vai buscar os três primeiros valores da linha dada e guarda-os em três variáveis inteiras locais, e depois usando o algoritmo de *Binary Search* (no nosso código surge como *pesquisaBinaria*) é avaliado se o primeiro e o segundo campo da linha existem no array *id_users* e também se o terceiro campo da linha existe no array *id_repos.* Caso a linha seja válida, adicionamos a respetiva linha no ficheiro *commits-final.csv* e adicionamos ao array *id_commits* o elemento correspondente à linha válida.

No final da função anteriormente referida temos o *id_commits* completo e pronto para ser ordenado com a função *qsort*.

Para filtrar o ficheiro *repos-ok.csv* criamos a função *verifyRepoID* que para cada linha invoca a função *analyseLineRepo*, que recebe como parâmetros a linha que pretendemos filtrar, uma cópia do estado inicial desta linha e uma variável com o tipo *Array_IDS*. Esta função invoca outras duas que resumidamente, vão buscar o primeiro e o segundo elemento da linha a filtrar e no caso de o primeiro campo verificamos se esse elemento existe no array *id_commits*, no caso do segundo campo vai verificar se o elemento existe no array *id_users,* usando mais uma vez a função *pesquisaBinaria* para ambas as situações. Caso a função *verifyRepoID* retorne 1, escrevemos a linha válida no ficheiro *repos-final.csv*.

Por fim, uma vez que concluimos que os ficheiros *users-ok.csv* e *users-final.csv* seriam iguais, criamos apenas uma função denominada *usersFinal* que se limita a copiar as linhas do primeiro ficheiro para o segundo.