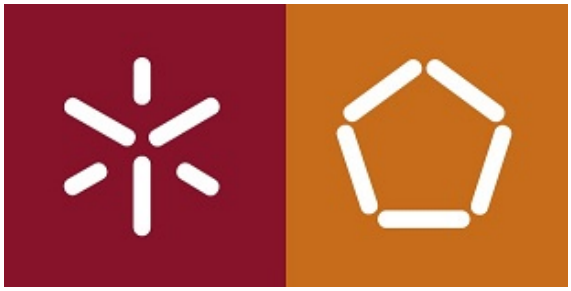

Universidade do Minho
Departamento de Informática

Github - Gestão de dados

LEI - Laboratórios de Informática 3

Grupo 9

10 de Novembro, 2021

Filipa Gomes
(a96556)

Ricardo Oliveira
(a96794)

João Loureiro
(a97257)

Introdução

Neste relatório descrevemos o nosso processo de desenvolvimento do guião 1, um sistema de gestão de usuários, commits e repositórios da plataforma Github, guardados em ficheiros *.csv*.

Exercicio 1

0.1 Estruturas de Dados

Para ser possível organizar a informação de uma forma ordeira e organizada, foram criadas três estruturas principais de **Users**, **Repos** e **Commits**, que depois são organizadas em arrays dinamicos para fácil e rápida implementação.

```c
typedef struct Repos{
  int cmt_id;
  int owner_id;
  char *full_name;
  char *license;
  char *has_wiki;
  char *description;
  char *language;
  char *default_branch;
  int cr_date[6];
  int updated_at[6];
  int forks_count;
  int open_issues;
  int stargazers_count;
  int size;
}Repo;
```

```c
typedef struct Commits{
      int repo_id;
      int author_id;
      int committer_id;
      int cmt_date[6];
      char *message;
}Commit;
typedef struct Users{
      int id;
      char *login;
      char *type;
      int c_date[6];
      int followers;
      int *flr_id;
      int following;
        int *flg_id;
        int pub_gist;
        int pub_repo;
 }User;
```

0.2 Importação de dados a partir de ficheiros

Foram criadas várias funções para importar os dados no projeto algumas delas são as funções *parse_x* e com o auxílio de uma função *build_x*, onde x é o tipo a ser lido do ficheiro *.csv*.

```c
Commit *parse_commits(int *i, char *entrada){
    Commit *commits= malloc(sizeof(Commit));
    FILE *r = fopen(entrada, "r");
    char *buff = malloc(sizeof(char) * 200000);

    while (fgets(buff, 200000, r)){
        build_commits(&commits[*i], buff, i);}
}

int build_commits(Commit *data, char *line, int *i){
    char *dt;
    data->repo_id = verify_int(strsep(&line, ";"));
    data->message = strdup(strsep(&line, "\n"));
}
```

*Codigo com omissões.

0.3 Verificação de campos

A verificação dos campos do ficheiro e a sua validação é efetuada em simultâneo com a sua construção do seu *type* usando as funções *verify_int*, *build_date* e usando a função especifica a cada tipo *verify_x*, onde x é o tipo a ser construido no momento.

0.4 Exportação de dados validados

Sendo o objetivo do exercício 1 a validação dos dados guardados nos ficheiros *.csv* e a sua exportação para ficheiros "*corretos*", foi necessário criar a função *save_x*, onde x é o tipo a ser guardado, que recebe um array de *x e o escreve num ficheiro com o nome "*ok-x.csv*".

```
int save_commits(Commit *u, int i, char *saida){
    FILE *us = fopen(saida, "w");
    fprintf(us, "repo_id;author_id;committer_id;commit_at;message\n");
    int j = 0;
    while (j < i){
        fprintf(us, "%d;%d;%d;%d-%02d-%02d %02d:%02d:%02d;%s\n", u[j].repo_id, u[j].
    author_id, u[j].committer_id, u[j].cmt_date[0], u[j].cmt_date[1], u[j].cmt_date[2],
    u[j].cmt_date[3], u[j].cmt_date[4], u[j].cmt_date[5], u[j].message);
        j++;
    }
    fclose(us);
    return 0;
}
```

*exemplo do *save_commits* que é guardado com o formato "*repo_id;author_id;committer_id;commit_at;message*"

Exercicio 2

0.5 Combater o problema

No exercicio 2 foi nos pedido que fosse feita a confirmação de dados entre ficheiros, por exemplo verificar se um commit tinha sido efetuado por um utilizador real e se não para eliminar o mesmo. Mas isso apresentava um outro problema por detrás, as nossas estruturas de dados não foram desenhadas para serem acedidas assim com tanta diferença de "*indices*", o que tornaria o nosso programa extremamente lento e custoso em termos de *CPU Time*. A nossa solução foi a utilização de *Hash_tables*. As *Hash_tables* utilizadas são apenas arrays de *int* com listas ligadas para evitar perda de dados nas colisões.

```
#define MAX_TABLE 300000

/** Struct para representar commits na hashtable*/
typedef struct commit_entry{
  int id;
  struct commit_entry *next;

}commit_entry;

/** Representa uma hashtable */
typedef struct hash_table {
  user_entry *users[MAX_TABLE];
  repo_entry *repos[MAX_TABLE];
  commit_entry *commits[MAX_TABLE];

}Hash_table;
```

0.6 Gestão dos dados

Chegamos também á conclusão de que o ficheiro users-ok.csv não ia sofrer alterações neste exercício, pelo que o qual bastava apenas guardar os registos na *Hash_table*, fazendo do users-final.csv o mais rápido de exportar. Foram implementadas várias funções com o intuito de resolver o problemas de repositórios e commits inválidos.

De modo a inserir os registos de ids nas *Hash_tables*, criamos uma funcão *insert_x*, onde x representa o tipo a ser inserido que fizesse esse mesmo *insert*, mais tarde esses dados seriam depois utilizados para fazer a construção do tipo de struct, repositório ou commit. A *parse_hash_x*,

que semelhante a *parse_x*, interpreta os ficheiros de entrada (desta vez os ficheiros resultantes do exercício 1), e a *build_hash_x*, onde x representa o tipo a ser construido, que guarda os registos repartidos nos seus respetivos campos da struct, verificando antes com o auxilio da *Hash_table*, se o registo continua a ser válido de acordo com as novas restrições do exercício 2. Os dados são depois guardados da mesma forma que no exercício 1, porém com a função *save_hash_x*, onde x é o tipo a guardar.

```
void insert_repo(int repo, Hash_table *table){
    size_t pos = hash(repo);
    repo_entry *new_repo = table->repos[pos];
    repo_entry *x = calloc(1,sizeof(struct repo_entry));
    x->id = repo;
    if (new_repo == NULL) {
        table->repos[pos] = x;
    } else {
        while ( new_repo->next != NULL) {
            new_repo = new_repo->next;
        }
        new_repo->next = x;
    }
}

Repo *parse_hash_repos(Hash_table *table, int *i, char *entrada)
{
    Repo *repos= malloc(sizeof(Repo));
    int  at = 1;
    FILE *r = fopen(entrada, "r");
    char *buff = malloc(sizeof(char) * 500000);

    if (r == NULL){
        printf("Error when opening %s!\n", entrada);
        return NULL;
    }else{fgets(buff, 500000, r);}

    while (fgets(buff, 500000, r)){
        build_hash_repos(&repos[*i],table, buff, i);
        if ((*i) == at){
            repos = realloc(repos, sizeof(Repo) * (*i)*2);
            at = 2 * (*i);
        }
    }
    free(buff);
    fclose(r);

    return repos;
}
```

*exemplo do *insert_repos* e do *parse_hash_repos*.

0.7 Desafios na resolução do Guião 1

Como grupo encontramos muitas dificuldades em implementar as Hash Tables da biblioteca *glib*, logo tentamos procurar outra solução, o que nos custou algum tempo de trabalho e dedicação. A conclusão do exercício 2 foi a mais complexa em termos de compreensão do que era pedido e como implementar eficientemente. De resto, o Guião 1 foi relativamente fácil de implementar.