Relatório do projeto em C de LI3

João Pimentel A80874 Jaime Leite A80757
Bruno Veloso A78352

Abril 2018

Universidade do Minho
Departamento de Informática
Laboratórios de Informática 3

Mestrado Integrado em Engenharia Informática

Conteúdo

1 Introdução

Este relatório aborda a resolução do projeto prático de "Laboratórios de Informática 3", na linguagem "C". O mesmo consiste em desenvolver um programa para analisar dados presentes em ficheiros *XML* relativos a conteúdo presente no *StackOverflow*.

Para realizar esta tarefa foi necessário criar três módulos: um para parsing/load dos dados necessários para realizar as tarefas pretendidas; outro para trabalhar sobre as estruturas por nós definidas (*USER*, *QUESTION*,*ANSWER*, *TAGS*); e, por fim, um módulo com o código relativo à resolução das queries fornecidas pela equipa docente.

Ao longo do relatório serão abordadas as decisões tomadas na implementação do projeto, nomeadamente, quais as estruturas utilizadas para criar cada um dos módulos, o porquê das escolhas feitas e as suas *API's*.

2 Módulos

Nesta secção são apresentados excertos de código comentado, tal como a explicação das soluções usadas e as divisões do projeto.

2.1 Estrutura de Dados (*struct*)

Para guardar toda a informação necessária lida a partir dos ficheiros, foram usadas quatro *HashTables*, uma para cada estrutura de dados definida.

Sejam às estruturas que tiveram de ser definidas de raíz:

1. *USER*;
2. *QUESTION*;
3. *ANSWER*;
4. *TAGS*;

Tendo em conta que o tempo de procura e inserção numa *HashTable* é (*O(1)*), foi escolhida como o principal tipo de dados do trabalho. Sendo que, em caso de ser necessário fazer alterações sobre algum elemento da mesma, é utilizado um clone, para não corromper os dados.

2.1.1 TCD_community

A community tem as quatro *HashTables* já mencionadas acima, no formato *GHashTable**, ou seja, *HashTables* implementadas pela *Glib*. Sendo que a *HashTable respostas* é composta por elementos do tipo *ANSWER*, *questões* por *QUESTION*, *users* por *USER* e *tags* por *TAGS*, como seria expectável.

```
typedef struct TCD_community{
    GHashTable* respostas; /*Hash com as respostas */
    GHashTable* questoes; /*Hash com as questoes */
    GHashTable* users; /*Hash com os users */
    GHashTable* tags; /* Hash com as tags */
}TCD_community;
```

2.1.2 USER

Um *USER* é composto pelos seu *ID*, *username*, contagem de *posts* (perguntas e respostas), a sua biografia e reputação.

```
typedef struct user {
    gpointer id; /* Apontador para o ID do user */
    char username[256];  /* String que contem o username do user */
    int post_count;  /* Contador de posts */
    char short_bio[16384];  /* Biografia do utilizador */
    int reputation; /* Valor da reputacao */
}*USER;
```

2.1.3 QUESTION

Uma *QUESTION* tem o seu *ID* e do respetivo autor, a data de criação (no formato *Date* e em forma de inteiro), título, *tags*, os votos e a contagem de respostas.

```
typedef struct question {
    gpointer id_user; /* Apontador para o ID do autor */
    Date postagem; /* Data da postagem em formato Date */
    int date; /* Data da postagem em formato int => exemplo:
        20181209 = 2018-12-9 */
    gpointer id; /* Apontador para o ID da questao */
    char titulo[256];  /* String que contem o titulo da questao */
    char tags[128]; /* String que contem todas as tags no formato <
        tag>*/
    int votos; /* Contador de votos da questao*/
    int respostas_count; /* Contador das respostas */
}*QUESTION;
```

2.1.4 ANSWER

Uma *ANSWER* possui, também, a data em dois formatos, o seu *ID*, do autor e da respetiva questão, o número de comentários, votos (*Up* e *Down*), o *score* e a quantidade de favoritos.

```
typedef struct answer{
  Date postagem; /* Data da postagem em formato Date */
  int date; /* Data da postagem em formato int => exemplo: 20181209
       = 2018-12-9 */
  gpointer id_resp; /* Apontador para o ID da resposta */
  gpointer id_question; /* Apontador para o ID da questao */
  gpointer id_user; /* Apontador para o ID do autor */
  int comentarios; /* Numero de comentarios */
  int votesUp; /*Votos positivos*/
  int votesDown; /*Votos negativos*/
  int score; /* Score da resposta */
  int favoritos; /* Numero de favoritos*/
}*ANSWER;
```

2.1.5 TAGS

Uma *TAG* possui apenas duas características: uma *String* com a *TAG* em questão (exemplo: "ssh") e *ID* da *TAG* em questão, consoante o valor referente no ficheiro *XML*.

```
typedef struct tags {
    gpointer id; /* ID de uma TAG consoante o ficheiro XML */
    char tag[256]; /* String que contem a tag em questao */
}*TAGS;
```

2.2 Parsing/Load

Esta secção descreve a realização de uma das partes fundamentais do projeto, o *parse* dos ficheiros e *load* do seu conteúdo essencial para a memória. Para isto, foram usadas funções já existentes na biblioteca *libxml2*.

2.2.1 Confirmação da possível utilização de ficheiro

Este excerto de código verifica se o ficheiro fornecido está em formato *XML* e se possui conteúdo antes de processar a sua informação. Caso alguma destas propriedades não se verifique, este retorna um valor de erro, caso contrário o ficheiro será processado.

```
1     xmlNodePtr cur = xmlDocGetRootElement(doc);
2
3     /* ... */
4
5     if (doc == NULL ) {
6         fprintf(stderr,"Document␣not␣parsed␣successfully.␣\n");
7         return 0;
8     }
9
10    if (cur == NULL) {
11        fprintf(stderr,"empty␣document\n");
12        return 1;
13    }
```

2.2.2 parseQuestion

Esta é a função mais complexa deste módulo, já que faz a inserção de tanto questões, como respostas nas respetivas HashTables.

É utilizada, por diversas ocasiões, a função "xmlGetProp" para encontrar a característica pretendida, uma de cada vez.

No momento em que todos os parâmetros de uma struct já estão guardados, é criada uma nova para acrescentar à HashTable utilizando funções definidas no módulo struct (newQuest, newAnswer).

É ainda incrementado o número de posts de um user nesta função.

```
1    uri = xmlGetProp(cur, (xmlChar*)"Id");
2    id = atoi((char*) uri);
3    xmlFree(uri);
4 /* ... */
5
6    q = newQuest(id, title, votos, respostas, data, tags,id_user);
7    g_hash_table_insert (questoes, GINT_TO_POINTER(q->id),q);
8    incrementa_posts(tq->users, q->id_user);
```

2.2.3 parseUser

Esta função funciona da mesma forma que a anterior, só que gera um USER, partindo do mesmo princípio de procura dos parâmetros e criação de uma struct completa.

```
cur = cur->xmlChildrenNode;

    while (cur != NULL) {
        if ((!xmlStrcmp(cur->name, (const xmlChar *)"row"))) {
      uri = xmlGetProp(cur, (xmlChar*)"Id");
      id = atoi((char*) uri);
      xmlFree(uri);
      uri = xmlGetProp(cur, (xmlChar*)"DisplayName");
      if (uri == NULL) strcpy(name, "NULL");
      else strcpy (name,(char*)uri);
      xmlFree(uri);
      uri = xmlGetProp(cur, (xmlChar*)"AboutMe");
      if (uri == NULL) strcpy(bio, "NULL");
      else strcpy (bio,(char*)uri);
      xmlFree(uri);
      uri = xmlGetProp(cur, (xmlChar*)"Reputation");
      reputation = atoi((char*)uri);
      xmlFree(uri);
      u = newUser(id, reputation, name, posts, bio);
      g_hash_table_insert (users,GINT_TO_POINTER(u->id),u);
         }
  }
```

2.2.4 parseVotes

A função parseVotes atua sobre o ficheiro Votes.xml e acrescenta votos (Up, Down e favoritos) ao contador dos mesmos de cada answer, pois é o único tipo de dados nos quais são utilizados os votos.

```
    while (cur != NULL){
        if ((!xmlStrcmp(cur->name, (const xmlChar *)"row"))){
        uri = xmlGetProp(cur, (xmlChar*)"PostId");
        i = atoi((char*)uri);
        xmlFree(uri);
        item_ptr = g_hash_table_lookup(tq->respostas,
            GINT_TO_POINTER(i));
        if (item_ptr != NULL) {
          uri = xmlGetProp(cur, (xmlChar*)"VoteTypeId");
              j = atoi((char*)uri);
              xmlFree(uri);
            if (j == 2) incrementa_votesUp (item_ptr);
              else if (j == 3) incrementa_votesDown (item_ptr);
              else if (j == 5) incrementa_favs (item_ptr);
            }
       }
```

2.2.5 parseTags

A função parseTags corre o ficheiro Tags.xml, criando structs do tipo TAGS, que apenas têm em consideração a String da mesma e o seu ID. Em seguida é possível ver a forma como funciona a mesma.

```
    cur = cur->xmlChildrenNode;

    while (cur != NULL) {
```

```
4           if ((!xmlStrcmp(cur->name, (const xmlChar *)"row"))) {
5               uri = xmlGetProp(cur, (xmlChar*)"Id");
6               id = atoi((char*) uri);
7               xmlFree(uri);
8               uri = xmlGetProp(cur, (xmlChar*)"TagName");
9               if (uri == NULL) strcpy(tag, "NULL");
10              else strcpy (tag,(char*)uri);
11              xmlFree(uri);
12              u = newTag(id,tag);
13              g_hash_table_insert (tags,GINT_TO_POINTER(u->id),u); //
                    mais eficiente
14  }
```

2.3 Queries

Nesta secção serão mencionados os modos como foram executadas as queries fornecidas e qual o algoritmo desenvolvido para atingir ao resultado esperado.

2.3.1 Init

A *query init* apenas aloca memória para a estrutura *TCD_community*, inicializando as *HashTable* a *NULL*. Serão, depois, alteradas para o seu valor final depois de aplicadas as funções de *load*.

2.3.2 Load

Esta *query* possui a carga de trabalho mais elevada do programa, já que carrega para memória todas os dados necessários à realização das outras *queries*, correndo os ficheiros de *Users*, *Posts*, *Votes* e *Tags*. De modo a amortizar o tempo de *loading*, foi decidido que certos parâmetros que podiam ter sido acrescentados às structs, como por exemplo, os *ID's* de todos os *posts* de um *user*, não o foram, pois implicariam um aumento no período de execução e este aspeto nem sempre é necessário à realização das interrogações.

2.3.3 Info From Post

Dado o identificador de um *post*, a função retorna o título do *post* e o nome de utilizador do autor. Se o *post* for uma resposta, a função deverá retornar informações da pergunta correspondente. Para isto foi necessário encontrar as informações em questão nas *HashTables*, existentes na *TAD_community*, usando funções de *lookup*.

2.3.4 Top Most Active

Tendo em conta que o *output* da *query* em questão deveria ser uma *LONG_list* com *N ID's* dos *users* mais ativos, ou seja, com mais *posts* (perguntas e respostas), bastou "clonar"os *users* para uma *GList* e ordená-la pelo parâmetro *post_count*. Posteriormente, são retirados os *N* elementos com mais posts da referida *GList* para a *LONG_list*.

2.3.5 Total Posts

Como é pedido que seja devolvido um par de valores, que representem o número de perguntas e respostas num dado intervalo de tempo, a resolução começou por converter as datas dadas para inteiros, de forma a serem feitas comparações mais eficientes. Dando uso à execução de uma função *foreach* sobre a *Hash* das questões e outra sobre a das respostas, em que se verifica se a data do post está contida entre dois valores acima mencionados e adiciona uma unidade ao respetivo contador caso se verifique a condição.

2.3.6 Questions with tag

Uma vez que as tags de uma determinada *QUESTION* estão guardadas na forma de *String*, sendo divididas entre si por "<>", ou seja, uma questão *Q* que tenha a tag "ssh" e "system_call" ficará com o parâmetro Q -> tags = "<ssh><system_call>". Assim, a solução evidente passou por concatenar um < antes da tag dada e um > depois, ficando algo como: <tag>. Sendo assim, usando o mesmo método acima mencionado para validar se a questão em análise pertence ao intervalo, só foi necessário dar uso à função *strstr* definida na biblioteca de *C*, de modo a ver se quereríamos guardar uma certa pergunta.

2.3.7 Get User Info

Para a *query* em questão foi adotado um método bastante simples, em que, primeiramente, foi encontrado o utilizador do *input*, usando um método de *lookup* na *Hash* de *users* consoante o *ID* do mesmo. Em seguida, foram criadas duas *GLists* (uma para questões e outra para respostas), em que seriam colocados todos os *posts* do *USER* em análise, ordenados por cronologia inversa. Deste modo, bastou definir um ciclo de inserção do *ID* do *post* num *array*, tendo em consideração a data mais recente.

2.3.8 Most Voted Answers

De modo a obter as *N* respostas com mais votos, num certo intervalo de tempo, foi utilizada uma *struct* auxiliar que contém a data inicial e final do intervalo (em forma de inteiro), bem como uma *GList*, inicialmente vazia, e a sua dimensão. Usando um *foreach* sobre a *Hash* das respostas, é testado se uma *ANSWER* pertence ao intervalo e, caso aconteça, é adicionada à *GList*, ordenadamente, consoante o número de votos. No final, basta retirar os *N ID's* de respostas com mais votos presentes na lista ligada.

2.3.9 Most Answered Questions

Foi utilizado um método em que, percorrendo a *HashTable* das questões, seriam adicionadas, a uma nova *Hash*, as questões cuja data de publicação se enquadrasse no intervalo temporal dado, tendo, ainda, um contador de respostas, iniciado a zero. Em seguida, era percorrida a *HashTable* que continha as

respostas, em que se analisava se a pergunta a que respondia existia na *Hash* gerada acima. Caso isso acontecesse, seria incrementada uma unidade no contador. Finalmente, a *Hash* que sofreu alterações é transformada numa *GList*, ordenada pelo contador acima menciona, sendo fácil retirar as N questões mais respondidas.

2.3.10 Contains Word

Para resolver a *query* seguinte, foi pensado numa forma semelhante à *query 4*, uma vez que foram acrescentados espaços à palavra em procura (exemplo: "droid" -> " droid "), de modo a evitar casos em que uma palavra pudesse ocorrer dentro de outra, como no caso de "android" e "droid". Além disso, foi copiada a *String* do título da questão em análise para uma *String* auxiliar, em que todos os caracteres que não fossem alfabéticos são convertidos em espaços e são, ainda, concatenados dois espaços, um no início e outro no fim. Finalmente, bastou validar se a palavra ocorria no título de uma questão dando uso à função *strstr*, onde, caso acontecesse, seria guardada numa *GList* ordenada pelas datas, à qual, no final, são copiados para uma *LONG_list* os *ID's* das N questões mais recentes.

2.3.11 Both Participated

Para conseguir obter uma lista com os *posts* em que ambos os utilizadores do *input*, foi decidido que as questões dos mesmos seriam guardadas em duas *HashTables* e que as respostas seriam guardadas na forma da pergunta correspondente. Assim, utilizando uma função que confirma se os elementos da *Hash* de menor dimensão ocorrem na maior, é possível gerar uma *GList* com as questões em que ambos os users participam, ordenada em termos temporais, de forma rápida e eficiente, graças à complexidade associada às *HashTables*.

2.3.12 Better Answer

De modo a ser obtida a melhor resposta a uma questão, ou seja, segundo o enunciado, a resposta que tenha o valor de ***(score × 0.45) + (reputação × 0.25) + (votos × 0.2) + (comentários × 0.1)*** mais elevado, o método adotado foi percorrer a *Hash* de respostas para encontrar quais estavam relacionadas com a questão do *input*, calculando os valores necessários. Ao encontrar um valor maior que o que estava guardado, o *ID* da resposta é, então, guardado na *struct* auxiliar.

2.3.13 Most Used Best Rep

Sabendo que o output pedido era uma lista com os *ID's* das N *tags* mais usadas pelos N *users* com melhor reputação, o algoritmo de resolução passou por criar *GLists* auxiliares que estivessem ordenadas pelos parâmetros mais convenientes: uma com os utilizadores ordenados pela reputação; outra com as questões dos N utilizadores com melhor reputação, no determinado intervalo de tempo; e uma

com as *TAGS* e respetivos contadores, inicializados a zero. De notar que foram acrescentados < > à *String* da *tag*, de modo a facilitar a procura (método utilizado na *query 4*). Percorrendo a *GList* das questões para cada *TAG*, é possível incrementar ao valor do contador quando encontra uma correspondência. Finalmente, os *ID's TAGS* mais utilizadas são colocados numa *LONG_list*, sendo que, as que tiverem zero ocorrências, não aparecerão.

2.3.14 Clean

Sabendo que a gestão de memória é uma característica importante deste projeto e que foi um aspeto ao qual foram feitas várias implementações consoante os resultados obtidos com o uso da ferramenta "*valgrind*", as quatro *HashTables* foram destruídas usando a função "*g_hash_table_destroy*" sobre cada uma, definida na *GLib*, de modo a libertar o seu conteúdo em memória.

3 Conclusão

Em suma, é opinião dos elementos do grupo que o projeto foi concluído com sucesso. Através do *parsing* dos ficheiros *XML* foi possível extrair toda a informação considerada relevante para os desafios propostos, de forma otimizada, guardando a mesma em estruturas de dados pensadas e implementadas minuciosamente. Sabendo que o objetivo, definido pelos 3 alunos, passava por criar estruturas simples e eficazes, tendo em conta os resultados obtidos, é credível que este objetivo foi atingido, pelo que o projeto foi uma forma de aprendizagem bastante eficaz para melhorias nos conhecimentos de *algoritmia e complexidade*, bem como "manipulação" de dados em grandes quantidades, em tempos não muito elevados.

4 Bibliografia

https://developer.gnome.org/glib/