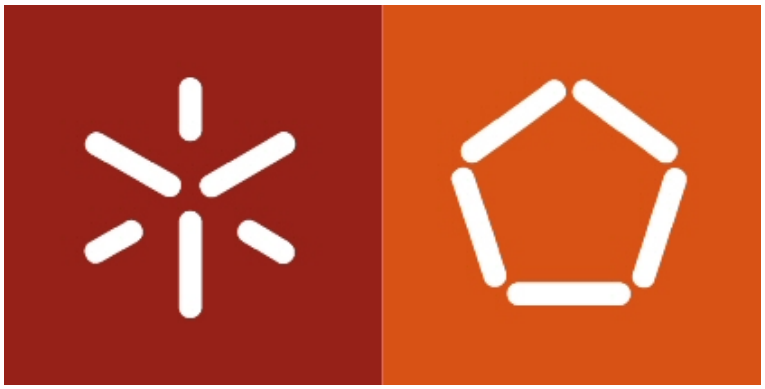

Relatório do Projeto de Laboratórios de Informática III em LaTeX

2.º ano de MIEInf

Universidade do Minho

Artur Ribeiro (A82516)
Davide Matos (A80970)
Francisco Freitas (A81580)
Inês Alves (A81368)

Departamento de Engenharia Informática

Abril de 2018

Conteúdo

1 Introdução

No âmbito da Unidade Curricular Laboratórios de Informática III, do Mestrado Integrado em Engenharia Informática da Universidade do Minho, este projeto tem como objetivo o desenvolvimento de um sistema capaz de processar ficheiros XML que armazenam as várias informações utilizadas pelo *Stack Overflow*.

Uma vez processada esta informação, pretendia-se que fosse possível executar um conjunto de questões específico de forma eficiente. Esta aplicação foi desenvolvida na linguagem de programação C, conforme solicitado no enunciado do projeto.

Devido à grande quantidade de dados, este projeto tornou-se um desafio, uma vez que, ao terem que passar no nosso programa milhões de dados, tornar-se-ia fundamental escolher bem as esturuturas de dados a utilizar de modo a que as questões propostas fossem respondidas de forma eficiente.

- O que é o Stack Overflow?

O *Stack Overflow* é, atualmente, uma das comunidades de perguntas e respostas mais utilizadas por *developers* em todo o mundo. Nesta plataforma, qualquer utilizador pode colocar questões que serão depois respondidaspor outros utilizadores. Estas respostas estão sujeitas a um sistema de votações que tende a favorecer as melhores respostas dado que as respostas são apresentadas por ordem decrescente do número de votos.

Cada utilizador acumula pontos de reputação que são ganhos sempre que uma das suas respostas é favoravelmente votada *(voted up)* e perdidos sempre que uma das suas respostas é votada desfavoravelmente *(voted down)*.

Segundo informação oficial, a plataforma tem cerca de 8.4 milhões de utilizadores e já se colocaram cerca de 15 milhões de questões, às quais se responderam 24 milhões de vezes.

A informação resultante da utilização da plataforma é extremamente útil e valiosa, não só devido ao conteúdo das perguntas e respostas mas também aos metadados originados a partir dos mesmos. Através deles é possível inferir, por exemplo, quais são os utilizadores mais ativos ou quais são os temas (identificados por tags) mais comuns.

A análise destes metadados pode, no entanto, ser um processo bastante demorado e custoso devido ao grande volume de dados com os quais se tem de lidar e às operações necessárias para cruzar as diferentes informações disponíveis.

2 Estruturas de dados usadas

Na primeira fase deste projeto houve necessidade de executar 3 tarefas:

1. Estudar a biblioteca *libxml2*

   *L*ibxml é uma biblioteca na linguagem de programação C que implementa funções de leitura, criação e manipulação de dados XML. A partir desta biblioteca foi possível extrair a informação necessária para dar resposta às questões propostas. Foi também importante o tutorial[1] presente no site desta biblioteca, orientando-nos na execução do método responsável pela extração da informação do ficheiro XML (*p*arse).

2. Estudar a biblioteca *GLib*

   GLib é uma biblioteca de uso geral que disponibiliza vários tipos de dados, macros, funções para conversão de tipos, utilitários para manipulação de strings e de arquivos, entre outros.

   Esta biblioteca foi utilizada neste protejo como forma de reutilização de, neste caso, *Hash Tables* e *AVL Trees*, uma vez que estas, se fossem implementadas de raíz, resultariam num "desperdício" de tempo e recursos.

3. Implementação de uma estrutura para armazenamento dos dados, *TCD_community*;

4. Responder às 11 *queries* propostas no enunciado do projeto

   Por fim, esta tarefa concentrava-se em aplicar conhecimentos de Programação Imperativa. O principal objetivo desta tarefa, para além da evidente resposta às questões, era também que estas questões fossem respondidas num reduzido espaço de tempo, explorando assim, o mais possível, a eficiência das estruturas escolhidas pelo grupo.

Para a realização deste projeto escolhemos usar três *AVL Tree* e uma *Hash Tabel*, estuturas abordadas nas aulas de Algoritmos e Complexidade, Unidade Curricular do 1.º semestre deste ano. Para além destas estruturas criamos também uma estrutura *ARRAY*.

Cada uma das *AVL Tree* foi ordenada de forma diferente, de modo a tirar o máximo proveito destas e dar resposta às *q*ueries no menor tempo possível.

Já a *Hash Tabel* foi uma estrutura auxiliar utilizada somente para a *query* 11, organizando as *TAGS*. Nesta fase, foi importante o contacto com a biblioteca *GLib*, já que, com o manual referente às tabelas de Hash, todas as funções de criação, inserção, procura e remoção ligadas a esta estrutura já estariam definidas, sendo o foco principal a resposta eficiente à *Query* referida.

Por fim, a estrutura *ARRAY* guarda por ordem descrescente os IDs dos *Users* com mais *posts*.

[1] http://xmlsoft.org/tutorial/ar01s03.html

- Nodo da AVL Tree de Posts ordenada pelo ID

```
struct post {
  long id;
  char * title;
  long ownerId;
  Owner autor;
  short  typeId;
  long parentId;
  DATA creation_date;
  Tags tag;
  int score;
  int comentCount;
  int answerCount;
};
```

Figura 1: Nodo da AVL de Posts ordenada por ID

id: guarda o ID do post correspondente;
title: array que guarda o título do post em questão;
ownerId: guarda o ID do *U*ser que criou post;
autor: apontador para uma estrutura *Owner*;
typeId: guarda a informação sobre o tipo do post (Tipo 1 - Pergunta; Tipo 2 - Resposta);
parentId: guarda o ID da pergunta à qual foi dada resposta. Se o post for uma pergunta, este parametro vai ser 0;
creation_date: guarda a data de criação do post;
tag: lista ligada que guarda todas as tags do post;
score: guarda o *score* do post;
comentCount: guarda o número de comentários do post;
answerCount: guarda o número de respostas dadas ao post; Se o post for uma resposta, este parametro vai ser 0.

- Estrutura Owner

```
struct owner {
  char* name;
  long id;
  int reputation;
};
```

Figura 2: Estrutura Owner

name: string que contém o nome do User que publicou o post.
id: guarda o ID do *User* em questão;
reputation: guarda a reputação do *User*.

- Estrutura Data

```
struct data{
    int dia;
    int mes;
    int ano;
};
```

dia: dia em que foi publicado o *post*;
mes: mês em que foi publicado o *post*;
ano: ano em que foi publicado o *post*.

Figura 3: Estrutura Data

- Nodo da AVL Tree de Users ordenada pelo ID

```
struct users{
    long id;
    int reputation;
    char* name;
    char* bio;
    int numposts;
    Post *lista_posts;
};
```

id: guarda o ID do *User*;
reputation: guarda a reputação do *User*;
name: *s*tring que guarda o nome do *User*;
bio: *string* que guarda a informação sobre o *User*;
numposts: guarda o número de posts do *User*;
lista_posts: array de apontadores para uma estrutura *P*ost, ordenados por cronologia inversa.

Figura 4: Nodo da AVL de Users ordenada por ID

- Nodo da AVL Tree de Posts ordenada por data

```
struct myAVL{
    int altura;
    struct myAVL *dir, *esq;
    struct myAVL *pai;
    DATA key;
    int numposts;
    Post* lista_posts;
    int isleft;
};
```

altura: guarda a altura da árvore;
dir, esq: ramos da árvore;
pai: apontador para o pai do nodo. Se o nodo em questão for a raíz da árvore, este apontador é *NULL*;
key: parametro de ordenação da *AVL Tree*;
numposts: guarda o número de posts que esse nodo possui;
lista_posts: array de apontadores para uma estrutura *Post*, ordenados por cronologia inversa.
isleft: informa se o nodo está à esquerda ou à direita do pai deste nodo. Se o nodo em questão for a raíz da árvore, este parametro toma o valor de -1.

Figura 5: Nodo da AVL de Posts ordenada por data

- Hash Table de TAGS ordenada por ID

```
struct mytags{
  long id;
  char* name;
};
```

id: guarda o ID da *TAG* em questão;
name: string que contém o nome da *TAG*.

Figura 6: Hash Table de TAGS ordenada por ID

- Estrutura ARRAY

```
struct arraygeral{
  int tamanho;
  int used;
  int *array1;
  long *array2;
};
```

tamanho: guarda o tamanho do *array*;
used: guarda o número de *slots* ocupados da estrutura;
array1: guarda o número de *posts*;
array2: guarda o ID dos *Users* por ordem do número de *posts*

Cada posição destes *arrays* guarda o ID do *User* e o número de *posts* deste.

Figura 7: Estrutura ARRAY

3 Modularização funcional e Abstração de dados

- Encapsulamento

O encapsulamento permite-nos que só exista uma forma de aceder aos nossos dados, através *"getters"*, e uma forma de os alterar, através de *"setters"*.

Estas funções permitem devolvem sempre uma cópia da informação contida na nossa estrutura, evitando que dados específicos sejam acedidos ou usados diretamente, o que traz uma maior segurança, uma vez que evita que um programa se torne tão interdependente que uma pequena mudança tenha grandes efeitos colaterais.

```
char * get_title(Post p) {
    return mystrdup (p->title);
}
```

Figura 8: Função *get_title*

No entanto, o nosso trabalho possui partilha de apontadores entre algumas estruturas com o objetivo de evitar informações repetidas em memória.

4 Estratégias seguidas em cada uma das interrogações

- Query 1

A estratégia seguida para responder a esta interrogação consiste numa simples procura na *AVL Tree* de Posts ordenada por ID, sendo que, a partir da estrutura *Owner*, conhecemos o nome do autor do *post* e o título é retirado do nodo da *AVL Tree* referida. Se o tipo do Post for 2 (ou seja, o post é uma resposta), faz-se uma nova procura da pergunta correspondente, através do parentID.

- Query 2

Para esta *Q*uery, e como foi dito anteriormente, foi criada uma estrutura *ARRAY* que guarda por ordem descrescente os IDs dos *Users* com mais *posts*. Posto isto, a resolução desta *Query* passa por simplesmente retornar os N primeiros elementos desta estrutura.

- Query 3

Para dar resposta a esta *Query* corremos a *AVL Tree* de Posts ordenada por Data, da menor para a maior data, incrementando o número de respostas ou o número de perguntas, dependendo do tipo do post em questão.

- Query 4

Nesta *Query* corremos do fim para o início a *AVL Tree* de Posts ordenada por Data, com o objetivo de retornar já os resultados ordenados por cronologia inversa, sem ser necessário implementar um algoritmo de ordenação. Nesta travessia são verificadas as *TAGs* e, caso a TAG passada como argumento esteja no post, o ID desse post é adicionado à *LONG_list*.

- Query 5

Como estratégia para resolver esta *Query* realizamos uma procura na *AVL Tree* de Users ordenada por ID, onde retornamos a *bio* e os IDs dos 10 primeiros posts do array lista_posts, previamente ordenado por cronologia inversa.

- Query 6

Nesta *Query* percorremos a *AVL Tree* de Posts ordenada por Data, guardando os IDs das respostas que estão no intervalo passado como argumento. Após esta travessia, é realizado um merge Sort do array dos IDs, segundo o parâmetro score. Por fim, retornamos os N primeiros do array resultante, ou seja, o array já ordenado.

- Query 7

Para esta *Query* foi realizada uma travessia da *AVL Tree* de Posts ordenada por Datas.

Nesta travessia, se o tipo do post for 1 (ou seja, é do tipo pergunta), o ID desse post é guardado e inicializa-se a 0 um número associado a esse ID, que mais tarde será o seu número de respostas. Se o tipo do post for 2, vai ser verificado no array de IDs de perguntas se o parentID deste post já existe[2]. Caso exista, é incrementado o tal valor referido no parágrafo anterior. Caso contrário, podemos concluir que a pergunta correspondente a esta resposta não está compreendida no intervalo em análise, por isso nada é feito.

[2]Só são consideradas respostas compreendidas no intervalo de tempo passado como argumento

- **Query 8**

Como método de resolução desta *Query* efetuamos uma travessia da *AVL Tree* de Posts ordenada por Datas, do fim para o início, implementando a função pré-definida *strstr* da palavra passada como argumento e do título do post, ou seja: *strstr*(palavra, título). Esta função só se aplica aos posts tipo 1, uma vez que a *Query* se refere apenas a perguntas. De outra maneira também não faria sentido, uma vez que as respostas não possuem título. No entanto, esta função conta também palavras contidas noutras. A decisão de usar esta função teve como base os testes disponibilizados na plataforma de ensino usada.

O caso de paragem acontece quando obtemos N posts que contêm a palavra referida. Como a travessia é feita do fim para o início, garantimos que devolvemos os N posts mais recentes.

- **Query 9**

Nesta *Query* foi realizada uma procura pelos IDs de ambos os *Users* na *AVL Tree* de Users ordenada por ID.

A partir desta, para ambos os IDs passados como argumentos, analisamos o *array* com apontadores para os seus posts. De seguida utilizamos 2 estruturas auxiliares (uma para cada *User*) ordenadas por data, em que é guardada a data da pergunta e o ID da mesma. Estas estruturas foram passadas para uma função (*getNcomuns*) cujo objetivo foi comparar se o ID da pergunta de uma estrutura se encontra também na outra. Caso o post for do tipo resposta, é utilizada a função *getquestionID* com o parentID para ser conhecida a pergunta que deu origem à resposta em questão.

No fim são devolvidos os N primeiros IDs presentes em ambas as estruturas.

- **Query 10**

Para responder a esta *Query* implementamos uma procura do ID passado como argumento na *AVL Tree* de Posts ordenada por ID. De seguida, guardamos a creation_date e o answerCount do post. A partir desta *creation_date*, corremos a *AVL Tree* de Posts ordenada por Data até que o número de respostas encontradas seja igual ao *answerCount*. Em cada resposta encontrada aplicamos a fórmula que se encontra no enunciado desta *Query*, devolvendo o ID da melhor resposta.

- **Query 11**

Por fim, para dar resposta à última *Query* realizamos 2 travessias da *AVL Tree* de Posts ordenada por Data.

A partir da primeira travessia retiramos os N *Users* com melhor reputação no intervalo de datas passado como argumento. Estes N *Users* são guardados num array.

Já na segunda travessia, em todos os posts que pertencerem aos utilizadores obtidos na primeira travessia, são contabilizadas quais e quantas *TAGs* foram usadas para esse post. Estas *TAGs* são ordenadas noutro *array* por ordem decrescente do número de vezes que foram usadas e são retornadas as N primeiras deste *array*.

Nome da Função	Tempo de execução (s)
STR_pair_info_from_post	0.000025
LONG_list_top_most_active	0.000001
LONG_pair_total_posts	0.009977
LONG_list_questions_with_tag	0.105284
USER_get_user_info	0.000044
LONG_list_most_voted_answers	0.052707
LONG_list_most_answered_questions	19.329192
LONG_list_contains_word	0.007055
LONG_list_both_participated	0.000516
LONG_better_answer	0.000117
LONG_list_most_used_best_rep	23.627971

Tabela 1: Tempos de Execução no Pior Caso

5 Estratégias para melhoramento de desempenho

Uma solução para melhorar o desempenho do nosso projeto seria pré-calcular e guardar no *TCD_community* determinadas funções úteis para a realização de algumas *queries* e assim aumentar a velocidade de resposta destas.

6 Conclusão

Ao longo do projeto, as dificuldades encontradas foram bastantes e diversificadas.

A realização deste projeto, para além de permitir o desenvolvimento das capacidades de raciocínio e programação do grupo, serviu como primeiro contacto no que toca ao uso de bibliotecas deste género, análise de grandes quantidades de informação e manuseamento de trabalho e implementações já existentes.

De facto, o maior obstáculo deste projeto, além da programação em larga escala, ao contrário do que acontecia nas Unidades Curriculares anteriores, foi o estudo de diversas bibliotecas, e a obtenção de uma linha de raciocínio que tivesse como prioridade a eficiência de resposta, que seria um dos maiores objetivos.

Já no fim do projeto, os testes disponibilizados pelos docentes às implementações propostas foram essenciais à comprovação das mesmas, e o uso do *Valgrind* tornou-se essencial no que toca a *memory leaks*.