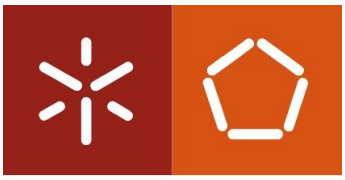

Universidade do Minho

Departamento de Informática

Projeto de Laboratórios de Informática III
Grupo 1

Catarina Machado (a81047) Cecília Soares (a34900)
João Vilaça (a82339)

5 de Maio de 2018

Resumo

O presente relatório descreve o projeto realizado no âmbito da disciplina de *Laboratórios de Informática III* (LI3), ao longo do segundo semestre, do segundo ano, do Mestrado Integrado em Engenharia Informática da Universidade do Minho.

O objetivo do projeto foi desenvolver um sistema capaz de processar a informação contida em ficheiros XML para responder a um conjunto de interrogações de forma eficiente, utilizando, para isso, a linguagem de programação C. Neste documento descrevemos sucintamente o tipo concreto de dados e as estruturas de dados usadas, a modularização funcional e abstração de dados a que recorremos, as estratégias seguidas em cada uma das interrogações e as estratégias utilizadas para melhoramento de desempenho.

Conteúdo

1 Introdução

O presente relatório foi elaborado no âmbito da unidade curricular *Laboratórios de Informática III* (LI3), ao longo do segundo semestre, do segundo ano, do Mestrado Integrado em Engenharia Informática da Universidade do Minho, e tem como objetivo descrever as tarefas desenvolvidas para criar um sistema capaz de processar as informações contidas em ficheiros XML para responder a um conjunto de interrogações de forma eficiente, utilizando a linguagem de programação C.

1.1 Descrição do Problema

Os ficheiros em formato XML que deveriam ser processados continham informação referente ao website *Stack Overflow*, o qual foi criado em 2008 por Jeff Atwood e Joel Spolsky para que programadores e entusiastas possam expor as suas dúvidas, funcionando como um fórum em que os usuários fazem perguntas e respondem a dúvidas dos seus pares. Além disso, podem ainda votar em questões e respostas que considerem pertinentes, obtendo pontos de reputação e medalhas pela qualidade da sua intervenção.

Em concreto, o trabalho prendia-se em extrair a informação necessária dos vários ficheiros, de forma a conseguirmos responder a 11 interrogações relacionadas com o conteúdo dos mesmos da forma mais eficiente possível, isto é, tendo especial atenção ao tempo de execução do programa, bem como ao encapsulamento dos dados.

1.2 Ficheiros XML

De todos os ficheiros XML colocados à nossa disposição (Votes, Tags, Users, PostLinks, Posts, PostHistory, Comments e Badges) e que se destinavam a dar resposta a um conjunto de interrogações, decidimos que apenas iríamos precisar de carregar algumas informações contidas nos ficheiros Tags.xml, Users.xml e Posts.xml.

No ficheiro Tags.xml retiramos somente o identificar da tag (ID) e o nome da mesma (TagName). No ficheiro Users.xml, recolhemos a informação relativa ao identificador do utilizador (ID), à sua reputação (Reputation), ao seu nome (DisplayName) e ao seu perfil (AboutMe).

Por último, quanto ao ficheiro Posts.xml extraímos o identificador do post (ID); o tipo de post (PostTypeId); o utilizador que publicou o post (OwnerUserId); o título do post (Title); as suas tags (Tags); a pontuação obtida (Score); o número de comentários que foram feitos (CommentCount); no caso de ser uma resposta, o número da pergunta a que se refere (ParentId) e, finalmente, a sua data de criação (CreationDate).

1.3 Concepção da Solução

Para resolvermos este problema foram cruciais três momentos. Numa primeira fase, definimos a estrutura de dados que consideramos que melhor solucionaria o nosso problema. Posteriormente, analisamos as vantagens e desvantagens das diferentes alternativas para fazer a leitura dos dados e a recolha da informação relevante. A nossa opção recaiu na utilização da API SAX para fazer o *parser* da informação, dado que permite o acesso serial ao conteúdo de um documento XML de forma orientada a eventos, aquilo a que chamam *event-based parser* para os documentos XML. Finalmente, na última etapa do projeto concentramo-nos em responder às diferentes *queries*.

O código subjacente à solução proposta pode ser encontrado no repositório:

https://github.com/dium-li3/Grupo1.

O restante deste relatório está organizado da seguinte forma: a Secção 2 descreve as estruturas de dados adoptadas, ao passo que a Secção 3 apresenta e discute a solução proposta para a resolução do problema. O relatório termina com conclusões na Secção 4, onde é também apresentada uma análise crítica dos resultados obtidos.

2 Organização dos Dados

Para desenvolvermos este trabalho adotamos as seguintes estruturas de dados:

2.1 Tipo de Dados Concretos

```
typedef struct TCD_community {
  GHashTable * users;
  GHashTable * questions;
  GHashTable * answers;
  GList * questionsList;
  GList * usersList;
  GPtrArray * day;
  GHashTable * tags;
} TCD_community;
```

Os dados necessários para responder às queries foram armazenados na estrutura `TCD_community`, a estrutura de dados principal do nosso trabalho. Esta estrutura armazena outras 7 estruturas que serão explicitadas mais tarde. De modo a respondermos às queries da forma mais eficiente possível utilizamos estruturas de dados já existentes e, para isso, recorremos ao glib, uma biblioteca que fornece simples estruturas de dados para programas em C.

Em primeiro lugar, para as estruturas **users**, **questions**, **answers** e **tags** utilizamos uma GHashTable para armazenar os dados.

A razão para essa escolha deve-se ao facto de tantos os IDs dos users, como os IDs das questions, answers e tags não estarem armazenados nos ficheiros de forma sequencial, nem serem contíguos (existem "buracos" entre os respetivos números identificadores), pelo que se utilizássemos os IDs como sendo os índices de um array haveria muita memória desperdiçada, e, por isso, não vimos nenhuma vantagem em utilizar um GPtrArray ou uma lista, nem mesmo utilizando algoritmos auxiliares, como por exemplo o algoritmo de procura binária.

No caso das árvores binárias, o tempo de procura e inserção no caso médio é logarítmico e no pior caso linear, o que poderia ser uma opção viável, porém, concluímos que utilizando GHashTable, ou seja, hash tables, e recorrendo aos IDs dos users, das questions e das answers, e ao nome da tag no caso das tags, como meio de procura dos elementos (chave da hash) conseguimos um tempo médio de inserção e procura (os dois métodos que mais utilizamos) constante, e no pior dos casos conseguimos um tempo linear ao tamanho do array, o que se traduz numa forma muito mais rápida e simples de trabalharmos o problema pois conseguimos aceder aos elementos da estrutura de dados muito eficientemente com o recurso a funções já existentes no glib.

No entanto, no caso das perguntas é também utilizada um segunda estrutura, GList, criada aquando da primeira vez que é preciso utilizar perguntas ordenadas por cronologia. A utilização desta estrutura permite que a inserção de perguntas seja feita numa GHashTable, com complexidade média O(1), e, apenas no fim, sejam ordenadas por data na GList, reduzindo a complexidade.

Utilizamos também uma segunda estrutura GList para os utilizadores, igualmente criada aquanda da primeira vez que é preciso utilizar os utilizadores ordenados por reputação.

Na estrutura **day** utilizamos um GPtrArray porque consideramos que seria a forma mais eficiente de percorrer os intervalos de tempo. Deste modo, a estratégia que utilizamos foi criar um array de apontadores onde o índice 0 corresponde à data de criação do stackoverflow (15/9/2008), o índice 1 ao dia seguinte, e assim sucessivamente. Assim, para percorrermos um intervalo de tempo basta fazermos 2 cálculos: 1- determinar qual é o índice da data inicial do intervalo no array (e para isso utilizamos uma função do glib (g_date_days_between) que nos diz quantos dias passaram entre 2 datas; neste caso, sabendo quantos dias passaram desde o início do stackoverflow até à data pretendida sabemos qual é o índice dessa data). 2- Descobrirmos também qual é o número de dias do intervalo de tempo. Depois disso, vamos ao GPtrArray, ao índice da data inicial, e vamos percorrendo as restantes datas até à data final bastando para isso incrementar 1 ao índice inicial até fazermos um número de iterações igual ao número de dias do intervalo de tempo. Com isto,

conseguimos aceder às informações de qualquer dia e, consequentemente, de qualquer intervalo de tempo de uma forma muito rápida e direta, com um tempo de procura constante (O(1)).

2.2 Estruturas de Dados Complementares

```
typedef struct users {
  long user_id;
  char * shortbio;
  char * username;
  int reputation;
  int n_posts;
  GArray * last_posts;
} users;
```

A estrutura de dados `users` contém o ID do utilizador, a sua short bio, o seu nome, a sua reputação, o número total de posts desse utilizador (i.e. perguntas e respostas), bem como um GArray contendo todos os posts do utilizador (perguntas e respostas) no formato *postAndDate*. Esta estrutura encerra informações necessárias para responder às interrogações 1, 5, 8 e 10.

```
typedef struct questions {
  long post_id;
  postDate pd;
  long user_id;
  char * title;
  char * tags;
  int n_answers;
  int n_answer_votes;
  GPtrArray * answers;
} questions;
```

A estrutura de dados `questions` contém o ID da pergunta, a sua data no formato *postAndDate*, o ID do utilizador que publicou a pergunta, o seu título, as suas tags, o número total de respostas que obteve, o número total de votos das suas respostas, bem como um GPtrArray com todas as respostas de cada pergunta no formato formato *answers*. Esta estrutura encerra informações necessárias para responder às interrogações 1, 4, 7, 8 e 11.

```
typedef struct answers {
  long user_id;
  long answer_id;
  long parent_id;
  int score;
  int comment_count;
} answers;
```

A estrutura de dados `answers` contém o ID do utilizador, o ID da resposta, o ID da pergunta a que essa resposta se refere, a sua pontuação e os seus comentários. Esta estrutura encerra informações necessárias para responder às interrogações 1, 6, 7, 9 e 10.

```
typedef struct day {
  int day;
  int month;
  int year;
  int n_questions;
  int n_answers;
  GPtrArray * questions;
  GPtrArray * answers;
} day;
```

A estrutura de dados `day` contém uma data e as informações relativas à mesma, nomeadamente o seu dia, mês, ano, o número de respostas e o número de perguntas efetuadas nesse dia, um GPtrArray com todas as perguntas desse dia no formato *questions* e um GPtrArray com todas as respostas desse dia no formato *answers*. Esta estrutura encerra informações necessárias para responder às interrogações 3, 4, 6, 7 e 11 (interrogações que dizem respeito a intervalos arbitrários de tempo).

```
typedef struct postAndDate {
    long post_id;
    int year, month, day, hour, min, sec, mili;
} postAndDate;
```

A estrutura de dados `postAndDate` contém o ID do post e o ano, mês, dia, hora, minuto, segundo e milissegundo em que esse post (pergunta ou resposta) foi efetuado. Esta estrutura encerra informações necessárias para responder à interrogação 7.

```
typedef struct tags {
  int id;
  char * nameTag;
  int value;
} tags;
```

A estrutura de dados `tags` contém o ID que identifica a tag, o nome da tag e ainda uma variável que servirá para armazenar o número de ocorrências dessa tag durante um intervalo de tempo. Esta estrutura encerra informações necessárias para responder à interrogação 11.

3 Implementação

3.1 Modularização Funcional

O trabalho é composto por vários módulos, tendo sido os que a seguir se descrevem criados por nós, consoante as necessidades que fomos detectando.

- main.c - main do programa.
- 00load.c - contém as funções necessárias para carregar e processar os dados necessários dos ficheiros XML.
- answers.c - todas as funções necessárias para aceder à estrutura de dados answers.
- day.c - todas as funções necessárias para aceder à estrutura de dados day.
- questions.c - todas as funções necessárias para aceder à estrutura de dados questions.
- struct.c - todas as funções necessárias para inicializar e aceder à estrutura de dados TCD_community.
- tags.c - todas as funções necessárias para aceder à estrutura de dados tags.
- users.c - todas as funções necessárias para aceder à estrutura de dados users.
- postDate.c - todas as funções necessárias para aceder à estrutura de dados postDate.
- 01TitleUserName.c - responde à interrogação n.º 1.
- 02TopMostActive.c - responde à interrogação n.º 2.
- 03totalPostDate.c - responde à interrogação n.º 3.
- 04questionsWithTag.c - responde à interrogação n.º 4.

- 05UserInfo.c - responde à interrogação n.º 5.
- 06mostVotedAnswers.c - responde à interrogação n.º 6.
- 07mostAnsweredQuestions.c - responde à interrogação n.º 7.
- 08titlesWithWord.c - responde à interrogação n.º 8.
- 09bothParticipated.c - responde à interrogação n.º 9.
- 10BetterAnswer.c - responde à interrogação n.º 10.
- 11mostUsedBestRep.c - responde à interrogação n.º 11.
- 12clean.c - liberta o espaço de memória antes de encerrar o programa.

Os módulos por nós criados podem ser vistos como unidades interdependentes que se complementam, cada uma com objetivos específicos, que interagem e que estão ligadas entre si apenas por funções (única interface), garantindo o encapsulamento dos dados.

Cada módulo tem uma única funcionalidade, um objetivo específico, como, por exemplo, aceder a uma estrutura de dados ou responder a uma query. De facto, com a divisão física dos ficheiros tentamos distribuir as tarefas do nosso programa de modo a garantir a abstração dos dados, bem como facilitar a reutilização do código.

Essa interdependência ou coesão entre os módulos é bem patente, já que os diferentes módulos necessitam interatuar para serem executados. Por exemplo, todas as interrogações necessitam do módulo struct.h, que, por sua vez, depende dos módulos interface.h, users.h, questions.h, postDate.h, day.h e tags.h, os quais dependem de vários outros e assim sucessivamente.

Contudo, apesar de haver uma interdependência entre os módulos, eles estão fracamente ligados na medida em que a única ligação entre eles é através da interface de cada módulo, isto é, recorrendo somente às diferentes funções disponibilizadas e nunca acedendo diretamente às estruturas de dados criadas.

3.2 Abstração de Dados

Conforme descrevemos na secção anterior, uma das estratégias adotadas para garantir a abstração dos dados foi através da modularização funcional do código. Aliado a isso, o encapsulamento de todos os dados foi garantido por ***forward declaration*** de todas as estruturas definidas, sendo os seus atributos somente acessíveis através de funções "getters e setters". De facto, os ficheiros headers criados impossibilitam que os atributos das diversas estruturas de dados criadas sejam acedidos diretamente. Por exemplo, a estrutura de dados que contém todos as informações relevantes e referentes ao usuário foi definida no ficheiro users.c da seguinte forma:

```
typedef struct users {
  long user_id;
  char * shortbio;
  char * username;
  int reputation;
  int n_posts;
  GArray * last_posts;
} users;
```

sendo que o acesso aos elementos de users só é possível através das funções "getters e setters", como por exemplo, *getUserId* ou *setUserId*, já que o ficheiro header users.h possui apenas a declaração *typedef struct users * Users*.

3.3 Queries

Finalmente, a última etapa do nosso projeto prendeu-se com a resposta às diversas *queries*, as quais passamos a explicar neste capítulo.

Query 1

"Dado o identificador de um post, a função deve retornar um par com o título do post e o nome (não o ID) de utilizador do autor. Se o post for uma resposta, a função deverá retornar informações (título e utilizador) da pergunta correspondente."xw

Na resposta a esta query, começamos por verificar se o ID do post passado como parâmetro para a função identifica uma pergunta. No caso da resposta ser negativa verificamos se esse ID identifica alguma resposta contida na GHashTable `answers`.

Na hipótese de ser uma pergunta, com o auxílio da estrutura de dados `questions` conseguimos obter o título do post e o ID do utilizador, através do qual, procurando na GHashTable users conseguimos saber se o usuário existe e, caso exista, obter todas as suas informações relevantes e, posteriormente, extrair o seu nome através da função getUsername. Desta forma, conseguimos devolver o par pretendido, o título do post e o nome do utilizador.

No caso do ID passado como parâmetro para a função identificar uma resposta, determinamos a que pergunta esta se refere e seguimos o procedimento descrito no parágrafo anterior.

Query 2

"Função que devolve o top N utilizadores com maior número de posts de sempre. Para isto, são considerados tanto perguntas quanto respostas dadas pelo respectivo utilizador."

```
typedef struct totalPosts {
  long user_id;
  int n_posts;
} totalPosts;
```

Para respondermos a esta interrogação criamos a estrutura de dados `totalPosts`. Esta é composta por um long que identifica o utilizador e um inteiro que traduz o número de posts publicados por aquele mesmo utilizador. Estas duas informações vão constar de cada posição de um GArray entretanto criado e que irá ser preenchido com a informação resultante de percorrer toda a GHashTable `users`. De seguida ordenamos o referido GArray, o qual passará a conter todos os users, bem como o total de post que cada um publicou, por ordem decrescente do número de posts. Por último, inserirmos os N utilizadores com mais posts de sempre na lista que é devolvida por esta função. Convém referir que se dois utilizadores tiverem o mesmo número de posts publicados, ocritério de ordenação adotado foi por ordem decrescente de ids.

Query 3

"Dado um intervalo de tempo arbitrário, obter o número total de posts (identificando perguntas e respostas separadamente) neste período."

Uma vez que na nossa estrutura de dados `day` já temos uma variável que nos diz o número total de perguntas e o número total de respostas efetuadas num determinado dia, para sabermos o número total de cada uma delas durante um intervalo de tempo criámos duas novas variáveis na nossa função da query 3: 'n_questions' e 'n_answers', e incrementámo-las com o valor das perguntas e respostas efetuadas, respetivamente, durante os dias passados como parâmetro.

No final de percorridos todos os dias, adicionamos as duas variáveis da nossa função a um LONG_pair, e retornamo-lo.

Query 4

"Dado um intervalo de tempo arbitrário, retornar todas as perguntas contendo uma determinada tag. O retorno da função deverá ser uma lista com os IDs das perguntas ordenadas em cronologia inversa."

Para esta query, tirando partido da nossa estrutura de dados `day`, que tem um GPtrArray com os apontadores das perguntas do dia em questão, começamos por percorrer cada pergunta desse array da última data passada como parâmetro e comparamos as suas tags com a tag passada como argumento. Deste modo, se a pergunta tiver a tag desejada inserimos o seu ID num array dinâmico. Vamos percorrendo os dias da "Date end" para a "Date begin".

No final de consultarmos todas as perguntas do intervalo de tempo temos o array dinâmico preenchido, e copiamos os IDs para uma LONG_list. Esta LONG_list já tem os IDs das perguntas ordenadas por cronologia inversa uma vez que fomos preenchendo o array dinâmico do dia mais recente para o dia mais antigo.

Query 5

"Dado um ID de utilizador, devolver a informação do seu perfil (short bio) e os IDs dos seus 10 últimos posts (perguntas ou respostas), ordenados por cronologia inversa."

Para responder a esta query fazemos uma procura por ID na GHashTable * `users` que devolve a estrutura do utilizador na qual está presente a informação do seu perfil, char * shortbio, e um vetor, GArray * last_posts, com os seus últimos posts que são ordenados por cronologia inversa e do qual, em seguida, se copiam os 10 primeiros elementos.

Query 6

"Dado um intervalo de tempo arbitrário, devolver os IDs das N respostas com mais votos, em ordem decrescente do número de votos; O número de votos deverá ser obtido pela diferença entre Up Votes (UpMod) e Down Votes (DownMod)."

Nesta query utilizamos um GPtrArray auxiliar. Ao percorrer os dias do intervalo de tempo fornecido como parâmetro inserimos no array auxiliar os apontadores das respostas que iam aparecendo.

Depois de percorrer todos os dias temos um array auxiliar com todas as respostas efetuadas nesse intervalo de tempo. Consequentemente, ordenamos o array pelo número de votos (no inicio do array está a resposta com mais votos e no fim a resposta com menos votos).

Para sabermos as N respostas com mais votos, consultamos o nosso array auxiliar e retiramos dele o ID das primeiras N respostas que aparecem.

Assim, temos a LONG_list pedida, por ordem decrescente do número de votos.

Query 7

"Dado um intervalo de tempo arbitrário, devolver as IDs das N perguntas com mais respostas, em ordem decrescente do número de respostas."

Para a resolução desta query consideramos o score total das respostas, e como critério de desempate a ordenação decrescente dos IDs.

Tal como na query 6 também recorremos a um array auxiliar, porém desta vez com apontadores para perguntas.

Utilizamos o mesmo raciocínio da query 6, mas agora fomos percorrendo o intervalo de tempo e adicionando ao nosso array auxiliar os apontadores para as perguntas. No fim do array preenchido, ordenamo-lo segundo o critério de maior número de respostas.

Retiramos os primeiros N elementos do array e devolvemos os determinados IDs numa LONG_list, que se encontra então ordenada decrescentemente segundo o número de respostas.

Query 8

"Dado uma palavra, devolver uma lista com os IDs de N perguntas cujos títulos a contenham, ordenados por cronologia inversa"

Nesta query, são copiadas todas as perguntas para uma lista, GList * l, que é iterada e na qual, em todas as perguntas até ao fim da lista ou até encontrar o número total de perguntas pedido, é verificado se o seu título contém a palavra dada e, em caso positivo, copia-se o ID para a estrutura a ser devolvida.

Query 9

"Dados os IDs de dois utilizadores, devolver as últimas N perguntas (cronologia inversa) em que participaram dois utilizadores específicos. Note que os utilizadores podem ter participado via pergunta ou respostas"

Nesta query, são copiadas todas as perguntas para uma lista, GList * l, que é iterada e na qual, em todas as perguntas até ao fim da lista ou até encontrar o número total de perguntas pedido, é verificado se o utilizador 1 criou ou respondeu a este post e, em caso positivo, verifica-se se o utilizador 2 criou ou respondeu a este post e, se também isto for verdade, copia-se o ID do post para a estrutura a ser devolvida.

Query 10

"Dado o ID de uma pergunta, obter a melhor resposta. Para isso, deverá usar a função de média ponderada abaixo: (score da resposta x 0.45) + (reputacao do utilizador x 0.25) + (votos recebidos pela resposta x 0.2) + (comentarios recebidos pela resposta x 0.1)"

Antes de mais, começamos por verificar se o ID da pergunta existe, pois caso não seja o ID de uma pergunta a função devolverá -1.

No caso de o ID ser válido, averiguamos quantas respostas tem essa mesma pergunta, através da função getNAnswers, e calculamos para cada resposta a sua média ponderada. Para o calculo da média de cada resposta socorremo-nos de várias funções. Em primeiro lugar, procuramos na GHashTable dos `users` o user que obtivemos em cada posição do GPtrArray `answers`, para de seguida conseguirmos obtermos a sua reputação. Posteriormente, obtivemos o score e número de comentários de cada resposta e calculamos a média ponderada com esses três fatores (ignoramos os votos já que o score é igual aos votos). Finalmente, verificamos qual a resposta com melhor média ponderada para depois devolvermos o seu ID.

Query 11

"Dado um intervalo arbitrário de tempo, devolver os identificadores das N tags mais usadas pelos N utilizadores com melhor reputação. Em ordem decrescente do número de vezes em que a tag foi usada."

Para a resolução desta query consideramos os N utilizadores com maior reputação no geral, e como critério de desempate a ordenação decrescente dos IDs. Também no que diz respeito ao número de ocorrências das tags, em caso de empate utilizamos a ordem decrescente dos IDs das tags como critério de desempate.

Para esta query utilizamos 1 array auxiliar para guardar apontadores de tags.

Tirando partido da GList * `usersList`, presente na nossa estrutura `TCD_community`, que possuí os apontadores para users ordenados consoante a sua reputação (da maior para a menor), criamos uma nova GList * mas desta vez com apenas os N primeiros elementos da GList * usersList. Deste modo, temos uma GList com os N users com melhor reputação.

Depois disso, percorremos todos os dias do intervalo de tempo tendo em atenção as perguntas, mais precisamente as suas tags. Se a pergunta tiver sido efetuada por um user com melhor reputação (ver se existe o user que fez essa pergunta na nossa GList auxiliar de users), pegamos

nas tags dessa pergunta (ainda todas juntas numa só string) e separamo-las de modo a termos todas as tags individuais.

Na nossa estrutura de `tags` temos uma variável 'value', com o valor 0, que é utilizada somente para esta query. Incrementamos essa variável para cada uma das tags contidas na pergunta. Adicionamos o apontador para essa tag ao nosso array auxiliar (tendo em atenção se já a tínhamos adicionado ou não ao array).

No final de percorridos todos os dias, todas as perguntas e respetivas tags temos um array de apontadores de tags, com todas as tags que apareceram durante esse intervalo de tempo e que foram feitas por algum dos N users com melhor reputação, com o seu respetivo número de ocorrências (armazenado na variável 'value').

Ordenamos esse array pelo número de ocorrências (do maior para o menor) e colocamos numa LONG_list os primeiros N identificadores das tags desse array.

3.4 Estratégias para melhorar o desempenho

Um dos objetivos deste projeto é o desempenho. Para o garantirmos começamos por estudar a melhor forma de ler e processar os dados armazenados nos ficheiros XML e, pesadas todas as vantagens e desvantagens, optamos por fazer o *parser* dos dados com recurso ao SAX *(Simple API for XML)*, em detrimento do DOM *(Document Object Model)*.

O DOM é um modelo que representa documentos XML numa estrutura em forma de árvore, designada de árvore DOM. Apesar do mesmo ser mais simples de utilizar, tornava o programa substancialmente mais lento, visto que consumia mais memória porque os ficheiros XML a serem processados eram grandes, resultando na construção de uma árvore DOM com toda a informação contida nos mesmos, a qual permanecia na memória enquanto estivesse a ser utilizada. Acresce que as diversas funcionalidades que o DOM possui, tais como, navegar pelos nós, remover, editar e apagar os nós da árvore DOM, acabam por gerar uma sobrecarga da memória sendo, concomitantemente, desnecessárias para a realização do projeto.

Por seu turno, a SAX efectua o *parser* de ficheiros em formato XML, definindo funções que são executadas quando determinado evento ocorre - "callbacks". O consumo de memória é reduzido, comparativamente com o DOM, uma vez que a memória utilizada corresponde somente às informações que estão sendo processadas pelas "callbacks". De facto, corridos alguns testes com a DOM verificamos que o carregamento das estruturas com o mesmo demorava praticamente o dobro do tempo do que com a SAX, factor determinante para a adoção da mesma.

A segunda medida que utilizamos para melhorar o desempenho do programa foi a criação da estrutura de dados que armazena o tempo. Tendo em conta que cinco das onze queries consideravam um intervalo de tempo, utilizamos um GPtrArray em detrimento de uma Hashtable, que não tem os dados ordenados, porque consideramos que seria a forma mais eficiente de percorrer os intervalos de tempo. Com esta solução, basta-nos saber qual o índice do array onde se inicia o intervalo de tempo e de quantos dias é esse mesmo intervalo para percorrermos o GPtrArray, conseguindo manter um custo baixo e constante das procuras dos intervalos de tempo.

A terceira medida adotada foi o método de resolução da query 2. Antes de nos decidirmos pela nossa implementação, testamos outras possibilidade para termos a certeza de que esta solução seria a que obteria melhor desempenho.

Em primeiro lugar, testamos a possibilidade de termos uma GList, duplamente ligada, com a informação contida na estrutura totalPosts e que contivesse apenas N elementos. A solução passaria por ordenar a lista sempre que "visitássemos" a GHashTable users. Acontece que o tempo desta implementação era substancialmente superior, na ordem dos 1.1s, 6.2s, 17s e 150s quando pedidos os 100, 500, 1.000 e 5.000 utilizadores com mais posts de sempre, respetivamente.

Outra tentativa de resolver esta query foi criar um GArray também com os dados da estrutura totalPosts, só que este GArray teria somente N posições. O GArray seria ordenado sempre que "visitássemos" a GHashTable users e quando estivesse cheio, os valores que obteríamos da GHashTable users seriam sempre colocados na N-ésima posição e reordenado o GArray por ordem decrescente do número de posts, isto até acabarmos de iterar sobre a GHashTable users. Todavia, esta solução mostrou-se ainda mais ineficiente, alcançando valores na ordem dos 3.8s,

23.9s, 50s e 294s quando pedidos os 100, 500, 1.000 e 5.000 utilizadores com mais posts de sempre, respetivamente.

Assim, optamos pela implementação da query 2 descrita na secção Queries, visto que o seu tempo é de cerca de 0.195s, 0.196s, 0.197s, 0.196s, 0.199s e 0.196s quando pedidos os 100, 500, 1.000 e 5.000, 10.000, 50.000 utilizadores com mais posts de sempre, respetivamente.

A quarta medida adotada foi nas queries 8, 9 e 11, que utilizam uma GList auxiliar que está presente na estrutura principal do trabalho (`TCD_community`). No caso da query 8 e 9, aquando da primeira chamada de qualquer uma desta duas funções, a função getQuestions é também invocada e devolve a lista de todas as perguntas ordenada por data. Esta lista é necessária para a resolução destas queries uma vez que em ambas as queries é necessário percorrer as perguntas por cronologia inversa (a pergunta mais recente primeiro) e esta GList dá-nos esse resultado e uma forma prática de o percorrer.

Da mesma forma, na query 11, é chamada a função getUsers. Esta função ordena os utilizadores do programa por ordem decrescente de reputação, o que é necessário para a resolução desta query uma vez que é pedida as tags mais usadas, num determinado intervalo de tempo, pelos N utilizadores com mais reputação global. É de notar que esta função apenas ordena os utilizadores se a query 11 for invocada, e que, caso seja invocada, apenas faz uma vez a ordenação, pois após a primeira utilização desta função a GList com os utilizadores ordenados por reputação fica armazenada na nossa estrutura principal e, nas próximas chamadas da query 11, basta utilizar essa GList sem problema, pois já se encontra calculada.

Esta estratégia ajuda na performance do programa uma vez que apenas precisamos de fazer o cálculo destas ordenações uma vez por programa e que só o fazemos caso seja necessário.

4 Conclusões

Face ao problema apresentado e analisando criticamente a solução proposta concluímos que cumprimos todas as tarefas, conseguindo atingir os objetivos definidos. No decurso do projeto socorremo-nos dos conhecimentos adquiridos nas unidades curriculares de Algoritmos e Complexidade, Programação Imperativa, bem como Arquitetura de Computadores de forma a equacionarmos a melhor solução para o problema apresentado, conseguindo obter resultados bastante satisfatórios face ao que nos foi pedido.

Todavia, entendemos que há alguns aspetos da nossa solução que, eventualmente, poderiam ser melhorados. Com efeito, infelizmente, não tivemos tempo para testar, em termos de desempenho, todas as soluções possíveis e que pudessem fazer diferença a nível de tempo em todas as queries.

Na verdade, dedicamos especial atenção à construção das nossas estruturas de dados, bem como à forma como iríamos ler e processar os dados dos ficheiros XML, pois verificamos que estas duas ações iriam ter um peso determinante em termos de eficiência no trabalho.

Em suma, não obstante as potenciais melhorias que poderiam ser feitas no programa, os testes por nós realizados, nas nossas máquinas, atingiram um tempo de execução que consideramos bastante aceitável.