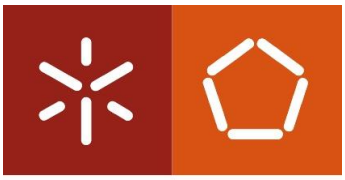

Universidade do Minho

Departamento de Informática

Projeto de Laboratórios de Informática III
Grupo 1

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

12 de Junho 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 Java. 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 **3**
1.1 Descrição do Problema 3
1.2 Ficheiros XML 3
1.3 Conceção da Solução 3

2 Organização dos Dados **3**
2.1 Tipo de Dados Concretos 4
2.2 Estruturas de Dados Complementares 4

3 Implementação **6**
3.1 Modularização Funcional 6
3.2 Abstração de Dados 7
3.3 Queries 8
3.4 Estratégias para melhorar o desempenho 11

4 Interface Gráfica **12**

5 Conclusões **14**

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 Java.

1.1 Descrição do Problema

Os ficheiros em formato XML que deveriam ser processados continham informação referente ao website *Stack Overflow*.

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, ao encapsulamento dos dados, bem como à modularização do código.

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.

Do ficheiro Tags.xml retiramos somente o identificador da tag (ID) e o nome da mesma (TagName). Do 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 Conceção 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 5, 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

```
public class TCD_Community implements TADCommunity {
    private Map<Long, Users> users;
    private Map<Long, Posts> posts;
    private Set<Question> questionsSet;
    private Set<Users> usersSet;
    private List<Long> usersPostList;
    private Map<LocalDate, Day> days;
    private Map<String, Long> tags;
```

Os dados necessários para responder às queries foram armazenados na classe `TCD_Community`, que contém os apontadores para as principais estruturas de dados do nosso trabalho, bem como os métodos necessários para aceder a essa mesma classe de forma a garantir o encapsulamento dos dados.

Em primeiro lugar, **users**, **posts**, e **tags** utilizamos a interface map para armazenar os dados, o que nos permite implementar quer uma HashMap quer uma TreeMap.

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 as interfaces **List** ou **Queue**.

De facto, a nossa opção recaiu sobre a classe HashMap para armazenar cada um desses dados porquanto o tempo de performance esperado é constante para a maioria das operações, como add(), remove(), contains(). Assim sendo, é significativamente mais rápida do que uma TreeMap, visto que o tempo médio de procura numa HashMap é O(1), ao passo que uma TreeMap tem uma performance média de O(log n) para as mesmas operações acima referidas.

Apesar de com a utilização da TreeMap conseguirmos reduzir o espaço de memória utilizado (em comparação com a HashMap), porque esta somente usa o espaço de memória necessário para armazenar os items, ao contrário da HasMap que usa região de memória contígua, entendemos que a HashMap é a escolha correta visto que sabemos a quantidade de objetos que a nossa coleção irá armazenar e não os queremos extrair na ordem natural. Assim, tendo em conta que a nossa tarefa se centra na performance do nosso programa, priorizamos o desempenho em detrimento do consumo de memória, pelo que a HashMap é a melhor escolha.

Os sets **questionsSet** e **usersSet** e a **usersPostList** foram especialmente criados por razões de eficiência, que passaremos a explicar em Secção 3.4, para auxiliar nas respostas às queries 8 e 9, o primeiro set, 11, o segundo set, e 2, a referida lista.

Na estrutura **days** também utilizamos a interface map para armazenar os dados do dia. Neste map, a chave é o dia em questão no formato LocalDate e o valor é um objeto da classe **Day**. Na Secção 3.4 explicaremos também o motivo desta decisão. Esta estrutura é utilizada para as queries que têm como parâmetros intervalos de tempo, nomeadamente a query 3, 4, 6, 7 e 11.

2.2 Estruturas de Dados Complementares

```
public class Users {
    private long user_id;
    private String shortbio;
    private String username;
    private int reputation;
    private int n_posts;
    private TreeSet<Posts> posts;
```

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

conjunto que contém todos os posts do utilizador (perguntas e respostas) ordenados por data. A escolha recaiu sobre uma TreeSet porque .... FALTA!!!!! Esta estrutura encerra informações necessárias para responder às interrogações 1, 5, 8 e 10.

```
public abstract class Posts implements Comparable<Posts> {
    private long user_id;
    private long post_id;
    private LocalDate pd;
    private int postType;
```

A estrutura de dados `Posts` é uma classe abstrata que contém três variáveis de instância, user_id, que se refere ao ID do utilizador que publicou a pergunta, post_id, número identificador do post, pd que se refere à data do post e o postType que identifica o tipo de post, 1 se é uma pergunta e 2 se é uma resposta. Estas informações são comuns quer às instâncias da classe *Questions* quer às da classe *Answers*, daí que esta classe seja hierarquicamente superior a estas duas. Convém notar que a classe `Posts` é uma classe abstrata, para que não se possa instanciar e para que, no futuro, caso seja necessário, se torne mais fácil criar novas classes de posts com o mínimo de modificações no código.

```
public class Question extends Posts {
    private LocalDate pd;
    private String title;
    private String tags;
    private int n_answers;
    private int n_answer_votes;
    private List<Answer> answers;
```

Esta classe contém a sua data no formato *LocalDate*, o seu título, as suas tags, o número total de respostas que obteve, o número total de votos das suas respostas, bem como uma lista com todas as respostas daquela pergunta. Esta estrutura encerra informações necessárias para responder às interrogações 1, 4, 7, 8, 10 e 11.

```
public class Answers extends Posts {
    private long parent_id;
    private int score;
    private int comment_count;
```

Esta classe tem três variáveis de instância, o parent_id que identifica a que pergunta aquela resposta se refere, o score da resposta e o número de comentários que esta obteve. Esta estrutura encerra informações necessárias para responder às interrogações 1, 6, 7, 9 e 10.

```
public class Day {
    private int n_questions;
    private int n_answers;
    private List<Question> questions;
    private List<Answer> answers;
```

A classe day foi criada com quatro variáveis de instância, n_questions e n_answers que se referem à quantidade de perguntas e respostas feitas naquela data, uma lista de perguntas e uma lista de respostas de determinada data. Esta classe foi assim desenhada de forma a conseguirmos obter, de forma célere, as informações necessárias para responder às queries 3, 4, 6, 7 e 11.

3 Implementação

3.1 Modularização Funcional

O trabalho é composto por três pacotes, commom, engine e li3, nos quais organizamos as nossas classes de acordo com as suas funcionalidades.

No *package* commom encontramos, tal como o nome indica, as classes comuns, as que constituem os alicerces, o sustentáculo do nosso programa. Aí definimos as seguintes classes:

- *Users* - classe que define o estado e o comportamento de um utilizador.
- *NumeroPostsComparador* - classe que estabelece uma ordem de comparação para a classe Users.
- *NumeroVotosRespostas* - classe que estabelece uma ordem de comparação para as Respostas, tendo em conta o critério da ordem decrescente do número de votos. Em caso de empate, prevalece a resposta que tenha um ID maior.
- *NumeroRespostasPergunta* - classe que estabelece uma ordem de comparação para as Perguntas, tendo em conta o critério da ordem decrescente do número de respostas. Em caso de empate, prevalece a pergunta que tenha um ID maior.
- *UsersComparator* - classe que estabelece uma ordem de comparação para os Utilizadores, tendo em conta o critério da ordem decrescente da reputação. Em caso de empate, prevalece o user que tenha um ID maior.
- *PostComparator* - classe que estabelece uma ordem de comparação para os Posts, tendo em conta o critério da ordem decrescente da data dos mesmos. Em caso de empate, prevalece o post que tenha um ID maior.
- *Posts* - classe abstrata que define as variáveis de instância comuns a uma instância de *Answers* ou *Question*.
- *Question* - classe que descreve o estado e o comportamento de uma pergunta.
- *Answer* - classe que descreve o estado e o comportamento de uma resposta.
- *Tag* - classe que descreve o estado e o comportamento de uma tag.
- *MyLog* - classe criada para gerar instâncias que irão identificar os resultados e os tempos obtidos pelas queries.
- *Pair* - classe que caracteriza um par, cujas instancias serão usadas para dar resposta a queries.
- *NoPostIdException* - Classe de exceção criada para sinalizar as situações em que o id em causa não representa nenhum post.
- *NoQuestionIdException* - Classe de exceção criada para sinalizar as situações em que o id em causa não representa uma pergunta.
- *NoAnswersException* - Classe de exceção criada para sinalizar as situações em que determinada pergunta não tem associada a si nenhuma resposta.
- *NoUserIdException* - Classe de exceção criada para sinalizar as situações em que determinada id não tem associada a si nenhum User.

Por seu turno, no pacote engine agrupamos as classes que fazem o parse dos dados necessários e a estrutura *TDC_Community*. Desta forma, tentamos concentrar neste pacote as classes que são o motor, que põem em marcha, o nosso programa.

Nessa medida, definimos neste *package* as seguintes classes:

- *Load* - classe que contém os métodos necessários para fazer o load dos ficheiros xml.
- *TCD_Community* - classe que contém os métodos necessários para descrever o estado e comportamento da estrutura de dados TCD_Community e que contém os métodos que respondem às onze queries.
- *SAXParsePosts* - classe que contém os métodos necessários para fazer o parse dos dados que serão imprescindíveis para as classes *Posts Answers* e *Question*.
- *SAXParseTags* - classe que contém os métodos necessários para fazer o parse dos dados que serão imprescindíveis para a classe *Tags*
- *SAXParseUsers* - classe que contém os métodos necessários para fazer o parse dos dados que serão imprescindíveis para a classe *Users*

Finalmente, no pacote li3 estão presentes a classe Main, o interface do sistema de queries, a classe de testes e a classe que implementa a nossa interface gráfica.

- *Main* - Main do programa.
- *TADCommunity* - Interface que dispõe dos métodos para carregar os ficheiros xml e dar resposta a cada uma das queries.
- *QueriesTests* - Classe para testar todas as queries.
- *GUI* - Classe que implementa uma interface gráfica para o nosso programa.

Os packages por nós criados podem ser vistos como unidades interdependentes que se complementam, cada um com objetivos específicos, que interagem e que estão ligados entre si apenas por métodos, garantindo o encapsulamento dos dados.

Acresce que, cada classe tem uma única funcionalidade, um propósito específico, como, por exemplo, definir o comportamento e o estado de um utilizador ou de um post, ou ainda fazer o parser dos ficheiros xml ou criar uma exceção. De facto, com a divisão física das classes e destas em diferentes pacotes, tentamos distribuir e organizar 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 pacotes e as classes é bem patente, já que os diferentes pacotes necessitam interatuar para serem executados. Por exemplo, as classes do pacote engine necessitam de importar o *package* commom, o qual contém as classes basilares do programa.

Contudo, apesar de haver uma interdependência entre as classes, elas estão fracamente ligadas na medida em que a única ligação entre estas é através dos diferentes métodos disponibilizados em cada uma, ou seja, o acesso às variáveis de instância é sempre privado, apenas podendo ser feito recorrendo a métodos.

3.2 Abstração de Dados

Conforme descrevemos na secção anterior, uma das estratégias adotadas para garantir a abstração dos dados foi através da modularização do código. Aliado a isso, tentamos garantir a capacidade de reutilização do código, o encapsulamento dos dados, bem como a possibilidade de alterar os dados sem grande impacto no programa, tendo em mente as seguintes preocupações:

- Os métodos apenas acedem às variáveis de instância locais ao módulo.
- As diferentes classes não exibem nenhum tipo de informação que permita do "exterior" conhecer a sua implementação. Efetivamente, apenas se conhecem os métodos que definem o comportamento de um objeto daquela classe, o qual é somente acessível através de uma API.
- Somente na classe Main temos código Input/Output.

Desta forma, a nossa estrutura de dados passa a ser opaca ao utilizador, tendo subjacente a ideia de *Data Hiding*, a qual foi implementada pelo acesso aos atributos somente por intermédio de métodos de instância e pela definição do método clone (deep clone) em cada uma das classes, que é chamado sempre que é necessário devolver uma cópia dos objetos e não um apontador para os mesmos.

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."

Na resposta a esta query, começamos por verificar se o ID do post passado como parâmetro para o método identifica um post, pois caso isso não aconteça é lançada uma exceção, `NoPostIdException`.

Caso o ID identifique um objeto da classe Posts, então verificamos que tipo de post é, ou seja, se é uma resposta ou uma pergunta. Na hipótese de ser uma pergunta, verificamos qual o ID do utilizador que fez aquela pergunta para o procurar na HashMap correspondente, de forma a conseguirmos obter o seu nome. Se o usuário não existir na HashMap `users` o método devolve um par nulo. Caso o usuário exista, extrai-se o nome deste, bem como o título do post. Se o título do post e o nome do usuário existirem ambos, o método devolve o respetivo par, caso contrário devolve um par nulo.

No caso do ID passado como parâmetro para o método 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."

Para respondermos a esta query criamos a classe `NumeroPostsComparador`, a qual define um critério de ordenação dos utilizadores por ordem decrescente do número total de posts publicados e, na possibilidade de dois usuários terem o mesmo número de posts, estabelece a ordem crescente de IDs dos usuários.

Ademais, criamos uma lista, `usersPostList`, onde armazenamos os utilizadores pela ordem estabelecida pela classe `NumeroPostsComparador`. Esta lista é preenchida aquando da primeira vez que é preciso utilizá-la, sendo que as respostas posteriores a esta query apenas têm de a limitar às N primeiras posições, aumentando-se assim a performance desta query. Este processo é feito através do método `initUsersPostsList`, o qual preenche devidamente a referida lista, caso esta esteja vazia. Por último, resta mencionar que, por questões de eficiência, utilizamos sempre iteradores internos para percorrer todos os usuários e ordená-los, bem como para para devolver os IDs dos N usuários com mais posts.

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: 'number_q' e 'number_p', 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, criamos um objeto `Pair` com as duas variáveis da nossa função, 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 uma List com as 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 `ArrayList`. Vamos percorrendo os dias da LocalDate "begin" para a "end".

No final de consultarmos todas as perguntas do intervalo de tempo temos o ArrayList preenchido, e retornamo-lo. Este ArrayList já tem os IDs das perguntas ordenadas por cronologia inversa uma vez que fomos preenchendo-o 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."

Em primeiro lugar, verifica-se se a HashTable possui o User com id passado como argumento, em caso negativo é lançada uma exceção em como esse utilizador não existe. Em caso afirmativo, copia-se o objeto da HashTable ao qual se retira a biografia e os posts. Em seguida, usa-se o método stream nos posts que serão ordenados recorrendo ao uso do PostComparator, dos quais se retiram os primeiros 10. Faz-se um map a esses 10 resultados para retirar os seus PostId e coleciona-se tudo para um List.

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 `ArrayList` auxiliar. Ao percorrer os dias do intervalo de tempo fornecido como parâmetro inserimos no ArrayList auxiliar **as** as respostas que iam aparecendo.

Depois de percorrer todos os dias temos um ArrayList 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), com recurso à classe `NumeroVotosRespostas`.

Através do método 'subList', selecionamos somente as N primeiras respostas que aparecem.

Depois disso, inserimos no ArrayList **ids** somente os IDs nas respostas e retornamo-lo.

É de referir que no caso das duas respostas terem o mesmo número de votos o critério de desempate a ordenação decrescente dos IDs.

Query 7

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

Utilizamos o mesmo raciocínio da query 6, mas agora fomos percorrendo o intervalo de tempo e adicionando ao nosso `ArrayList` auxiliar as perguntas. No fim do array preenchido, ordenamo-lo

segundo o critério de maior número de respostas, e em caso de empate, utilizamos o critério de ordenação decrescente dos IDs. Para a ordenação, recorremos ao Comparator da classe `NumeroRespostasPergunta`.

Retiramos os primeiros N elementos do ArrayList e devolvemos os respetivos IDs num `ArrayList`, que se encontra então ordenado 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"

É criada uma estrutura de posts auiliar ordenada caso a mesma ainda não tenha sido criada. Sobre essa estrutura é feito um stream, ao qual se filtram as perguntas que não tenham as palavras passadas como argumento. Retiram-se os 10 primeiros elementos, e usa-se um map para retirar os POstId que se guardam para uma List.

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"

É criada uma estrutura de posts auiliar ordenada caso a mesma ainda não tenha sido criada. É criado um iterator que percorre essa estrutura para verificar os N primeiros posts em que ambos os utilizadores criaram ou participaram num post.

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 convém ressaltar que esta query lança duas exceções: `NoAnswersException`, quando a pergunta não tem nenhuma resposta, e `NoQuestionIdException`, quando o ID passado como parâmetro não identifica uma pergunta.

No caso de o ID ser válido, averiguamos quantas respostas tem essa mesma pergunta, através do método getNAnswers, e calculamos para cada resposta a sua média ponderada. Para o calculo da média de cada resposta socorremo-nos de vários métodos.

Em primeiro lugar, obtemos cada uma das respostas da pergunta em causa, iterando sobre a List `answers`, posteriormente procuramos na HashMap `users` o user que deu aquela resposta, para de seguida conseguirmos obter a sua reputação. Posteriormente, obtemos 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.

Tirando partido do Set `usersSet`, presente na nossa classe `TCD_community`, que possui os users ordenados consoante a sua reputação (da maior para a menor), selecionamos somente os N primeiros elementos e criamos um `Map` chamado "users_rep". Este Map possui como chave o Id do user, e como valor o "apontador"para o utilizador.

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 no map "users_rep"um user com a chave (Id) do utilizador que fez a pergunta), pegamos nas tags dessa pergunta (ainda todas juntas numa só string) e separamo-las de modo a termos todas as tags individuais.

Assim, adicionamos os IDs dessas tags a um `ArrayList` chamado ident_tags. (Temos um map "tags"na nossa classe `TCD_community` em que ao nome da tag corresponde o respetivo identificador, o que nos ajuda agora a colecionar esses Ids).

No final de percorridos todos os dias, todas as perguntas e respetivas tags temos um ArrayList "ident_tags", com todos os identificadores das tags (com repitação) que foram aparecendo durante esse intervalo de tempo e que foram feitas por algum dos N users com melhor reputação).

Deste modo, criamos um novo `Map`, chamado "maptags", onde a cada um dos identificadores das tags (presentes no ArrayList "ident_tags") corresponderá o número de repetições da tag.

Por último, ordenamos esse map consoante o número de repetições das tags, e devolvemos os identificadores das N primeiras tags do map ordenado.

3.4 Estratégias para melhorar o desempenho

Para melhorarmos o desempenho do nosso programa, uma das estratégias utilizadas foi a utilização de coleções que nos permitem aumentar a eficiência do nosso programa, na medida em que preenchemos cada uma das coleções somente na primeira vez que é preciso utilizar:

- usuários ordenados por número de posts, conforme o método `initUsersPostsList`;
- perguntas ordenadas por datas, conforme o método `initQSet`;
- usuários ordenados por reputação, conforme o método `initUSet`.

O `questionsSet` é utilizado nas queries 8 e 9, já o `usersSet` é usado na query 11 e, finalmente, a `usersPostList` guarda os dados para dar resposta à query 2. Assim, sempre que estas coleções se encontrem devidamente preenchidas a resposta às referidas queries terá uma complexidade de O(1).

Para determinar que coleção seria mais eficiente em cada caso concreto analisamos o tempo de execução do programa com a utilização de TreeSets e ArrayList nos diferentes casos.

Antes de nos decidirmos pela nossa implementação, testamos outras possibilidade para termos a certeza de que esta solução seria a que obteria melhor desempenho. No que se refere à query 2, testamos a possibilidade de termos uma TreeSet a armazenar os usuários por ordem decrescente de posts e um ArrayList, corremos a query 100 vezes para cada uma das referidas hipótese obtivemos os tempos descritos nos gráficos infra.[1].

Figura 1: Resultados da implementação da query 2 com usersPostsSet.

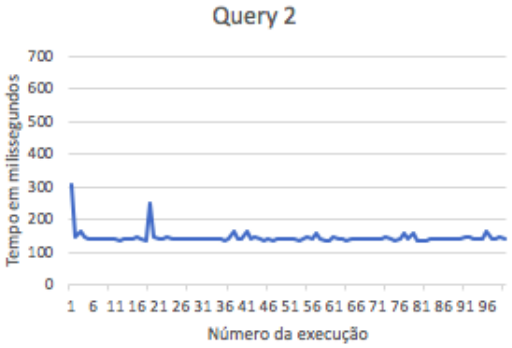

Figura 2: Resultados da implementação da query 2 com usersPostsList.

[1]A máquina utilizada para medir os tempos de execução foi um MacBook Air com Processador 1,6 GHz Intel Core i5, Memória 8 GB 1600 MHz DDR3 e disco flash de 121 GB

Conforme podemos observar a partir dos gráficos, a implementação da query 2 utilizando uma TreeSet de usuários ordenada por ordem decrescente de número de posts, é menos eficiente, já que em média leva 198,76 milissegundos(ms) a preencher devidamente a TreeSet e a responder à query 2, ao passo que com uma lista, o mesmo processo, leva em média 144,45 ms,o que equivale a um ganho de cerca de 50 ms, pelo que optamos pela implementação da query 2 descrita na Secção 3.3.

Assim como para a query 2, fizemos o mesmo teste para a query 11, tendo como fator diferenciador o `usersSet` implementado, averiguando se seria mais vantajoso recorrer ao uso de um TreeSet ou de um ArrayList. Deste modo, utilizando o mesmo procedimento de correr o programa 100 vezes e guardando os respetivos resultados construimos os gráficos em seguida:

Figura 3: Resultados da implementação da query 11 com usersSet.

Figura 4: Resultados da implementação da query 11 com usersList.

Deste modo, concluimos que, no caso da query 11, ao contrário da query 2, é mais vantajoso recorrermos ao uso de Sets ao invés de Lists.

Outro solução que implementamos para melhorar o desempenho do nosso programa foi o uso da classe `Day`. Esta classe, que armaneza o número de perguntas, o número de respostas e duas listas, uma com as perguntas e outra com as respostas efetuadas naquele dia, permite-nos percorrer o intervalo de tempo passado como parâmetro nas queries que se referem a tempo, nomeadamente a query 3, 4, 6, 7 e 11, de uma forma muito rápida e eficaz.

Isso deve se ao facto de possuirmos um Map (o map **days**), onde a cada LocalDate corresponde um objeto da classe `Day`. Assim, partindo da data inicial passada como parâmetro conseguimos obter de forma imediata as informações relativas a esse dia, variando de acordo com o pedido na query e, somando 1 dia à data, até chegar à data final passada como argumento, vamos obtendo todas as informações necessárias, de todos dias do intervalo de tempo, para responder à query com uma complexidade de O(1), uma vez que sabemos sempre qual é a chave desejada no map **days**.

4 Interface Gráfica

Implementamos também uma interface gráfica, presente na classe GUI, com a ajuda da biblioteca JavaFX.

Quando o programa é aberto aparece a imagem em seguida, e o utilizador deverá colocar o seu nome e o path (completo) onde se encontra a pasta com os ficheiros XML.

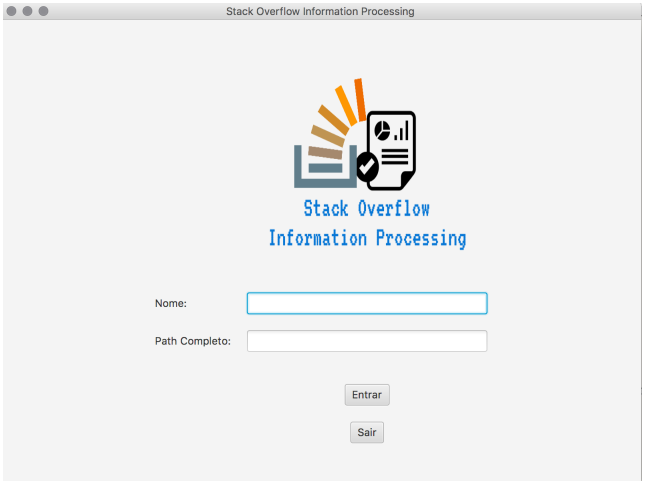

Figura 5: Stack Overflow Information Processing.

Depois disso aparece a seguinte dashboard, onde o utilizador deverá escolher a query que pretende correr. Em baixo, à esquerda, aparece o tempo (em milissegundos) que o load demorou.

Figura 6: Stack Overflow Information Processing- Queries.

Assim, selecionando a query pretendida, abre a seguinte dashboard, por exemplo da Query 7:

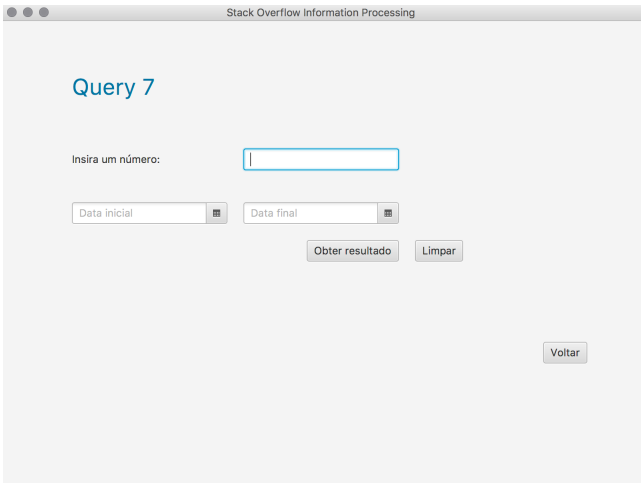

Figura 7: Query 7.

O utilizador deverá preencher os dados dos argumentos necessários e clicar em "Obter resultado"de modo a obter os resultados desejados.

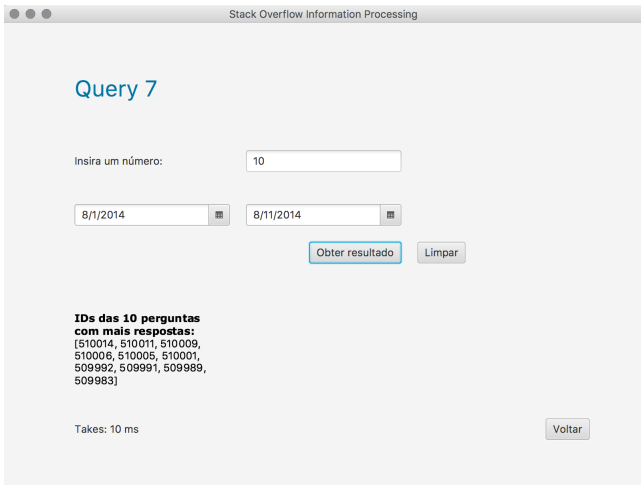

Figura 8: Query 7- resultado.

A interface gráfica foi implementada separadamente ao programa, pelo que o programa pode ser corrido tanto pelo main do programa fornecido pelos professores como através da classe GUI.

5 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 Orientada a Objetos, 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.

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.