universidade de aveiro

ASSIGNMENT 1 - RELATÓRIO

Simple Document Indexer

Recuperação de Informação

Curso: Engenharia de Computadores e Telemática

Departamento de Eletrónica, Telecomunicações e Informática

09 de Outubro de 2017

Professor:

Prof. Sérgio Matos

Autores:

Miguel Oliveira nº 72638

Tiago Henriques nº 73046

universidade de aveiro

Índice

1 – Introdução

A primeira tarefa do trabalho prático desenvolvido no âmbito da unidade curricular de Recuperação de Informação consiste em criar um mecanismo de indexação simples de vários documentos. Este será composto em vários módulos: corpus reader, document processor e indexer.
Com este relatório, pretende-se apresentar o diagrama de classes, as classes implementadas e respetivos métodos, referir as bibliotecas externas utilizadas, explicar o modo como o código está estruturado e o que é necessário fazer para executar o programa. No fim, são apresentados os resultados da execução do programa referindo os tempos médios de simulação. São ainda apresentadas as respostas às perguntas que estão na Secção 4. do enunciado do projeto.

universidade de aveiro

2 - Diagrama de Classes

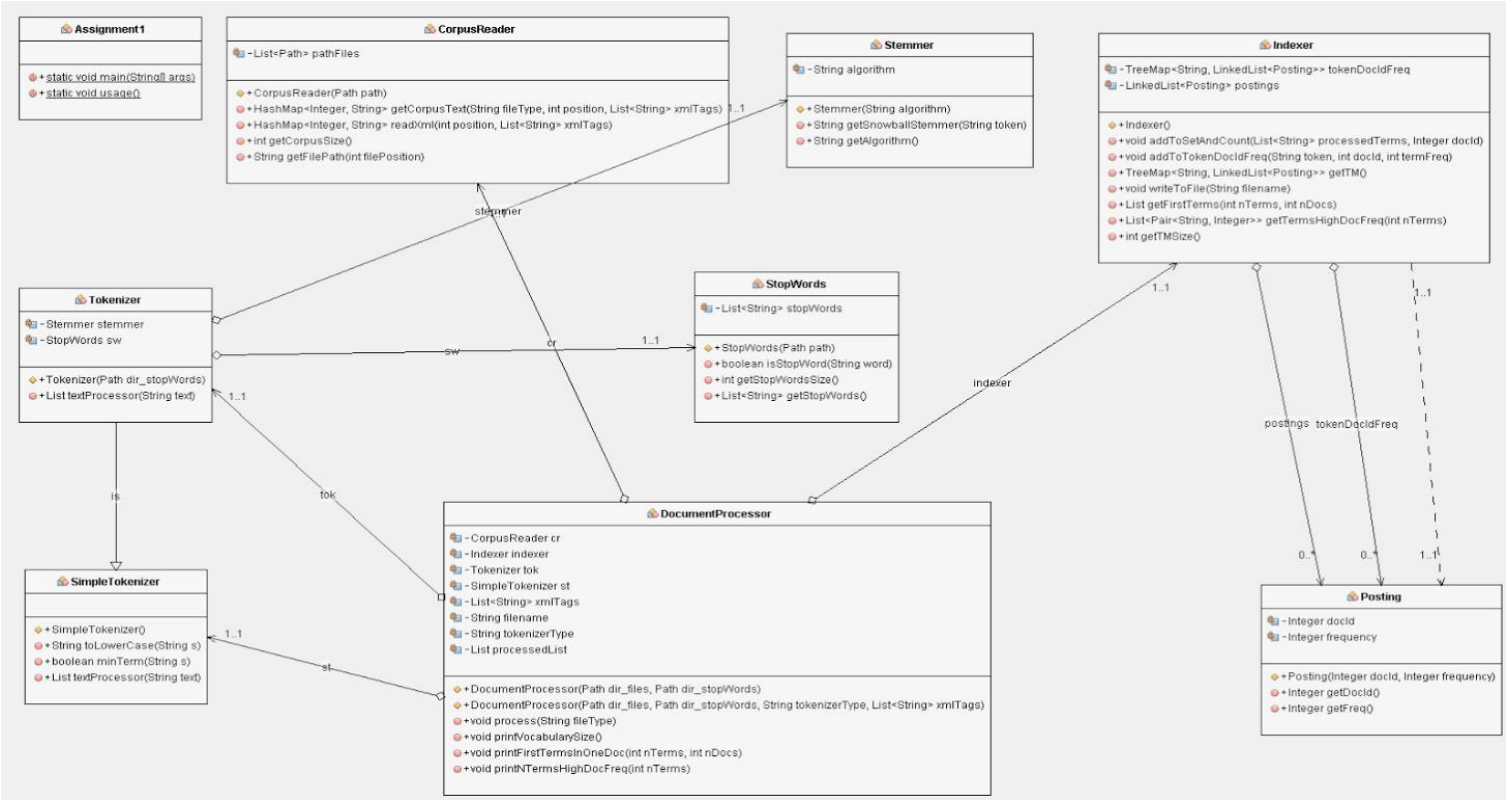

Figura 1 - Diagrama de Classes onde estão presentes as classes e os seus respetivos métodos

3 – Classes e Métodos

3.1 – Corpus Reader

Este módulo é responsável por localizar os documentos a processar e posteriormente pela sua leitura. Caso o corpus seja do tipo XML, então irá ser invocado o método responsável por este tipo de documentos, onde será retornado o texto (corpo + título) e o ID do documento, para posteriormente poder ser processado pelo Tokenizer.

Métodos e a sua descrição:

- **getCorpusText(String fileType, int position, List<String> xmlTags):** Método usado para obter o texto de cada ficheiro, onde recebe o tipo de ficheiros a serem lidos e invoca a função readXml.

- **readXml(int position, List<String> xmlTags):** Método usado para ficheiros do tipo xml que retorna uma HashMap em que a chave é o identificador do documento e o valor é o texto desse documento.

- **getCorpusSize():** Método que retorna o tamanho do corpus;

- **getFilePath(int filePosition):** Método que retorna o caminho para cada ficheiro em forma de String;

3.2 – SimpleTokenizer

Este módulo tem a finalidade de fazer a separação dos espaços em branco, converter os caracteres para minúsculas, remover todos os caracteres não alfanuméricos e manter apenas tokens que tenham 3 ou mais caracteres. Isto servirá para de seguida proceder à indexação dos tokens, correspondendo a lista de caracteres já processados ao vocabulário a ser usado pelo Indexer.

Métodos e a sua descrição:

- **toLowerCase(String s):** Função que para cada palavra da lista de tokens, aplica a conversão desse caracter para caracter minúsculo;

- **minTerm(String s):** Função booleana que verifica se as palavras possuem 3 ou mais caracteres;

- **textProcessor (String text):** Função responsável por aplicar as filtragens ao texto proveniente de um documento. Neste caso, são retirados todos os sinais de pontuação e caracteres não alfanuméricos. De seguida, é verificado se cada termo tem 3 ou mais caracteres e de seguida, convertido para caracter minúsculo.

3.3 –Tokenizer

Esta classe estende o módulo descrito imediatamente acima, o SimpleTokenizer. Portanto, tem a finalidade de aplicar as transformações do SimpleTokenizer, visto que herda os métodos desta classe, com o acréscimo de aplicar algumas filtragens adicionais como a verificação de "stop words" e a aplicação da "Stemmização", portanto esta classe instancia quer a classe StopWords quer a classe Stemmer. Este método é mais complexo do que o SimpleTokenizer, aplicando uma filtragem mais abrangente às palavras dos documentos e, desta forma, resultando num vocabulário de palavras menos extenso do que o vocabulário resultante do SimpleTokenizer.

Métodos e a sua descrição:

- **textProcessor(String text):** Função responsável por aplicar as filtragens ao texto proveniente de um documento. Neste caso, são retirados todos os sinais de pontuação e caracteres não alfanuméricos. De seguida, é verificado se cada termo tem 3 ou mais caracteres e de seguida, convertido para caracter minúsculo. Este método, como foi referido anteriormente, faz uma filtragem mais abrangente, verificando ainda se o termo em questão é "stop word" e aplicando "stemmização".

3.4 – StopWords

Módulo que recebe para leitura as palavras do ficheiro "stopWords.txt" de forma a identificar as "stop words". Contém um método booleano, isStopWord, que recebe os vários tokens, e irá verificar se o token em causa é ou não stop word, sendo este token descartado para a indexação caso seja de facto uma "stop word".

Métodos e a sua descrição:

- **isStopWord(String word):** Função booleana que verifica se um token é ou não uma "stop word";

- **getStopWordsSize():** Função que retorna o tamanho da lista de stop words;

- **getStopWords():** Função que retorna a lista de stop words;

3.5 – Stemmer

Esta classe tem como função o tratamento automático das diferentes formas morfológicas das palavras, de forma a representar palavras semelhantes por meio de uma palavra principal. A implementação do Stemmer, em si, recorre ao Porter Stemmer através do URL seguinte: http://snowball.tartarus.org/download.html. Para que esta biblioteca seja reconhecida, foi necessário incluir no projeto e nas dependências do Maven, o jar do snowball (snowball.jar), que se situa na pasta "Dependencies".

Métodos e a sua descrição:

- **getSnowballStemmer(String token):** Este método trata da implementação do Stemmer e retorna uma palavra resultante do processo de Stemmização. Recebe como argumento um token, onde é aplicado o algoritmo definido. É retornado o token transformado ou o original, caso não tenha sofrido qualquer alteração.

- **getAlgorithm():** Este método retorna o algoritmo de stemming usado no processo de "tokenização", no nosso caso é o "porter".

3.6 – Posting

Esta classe tem com principal objetivo ser uma estrutura de dados de modo a ser usada no Indexer. Ou seja, esta classe é constituída apenas pelo ID do documento em que o token aparece e a sua frequência (número de vezes que ocorre o termo no mesmo documento).

Métodos e a sua descrição:

- **getDocId():** Função que retorna o ID do documento;

- **getFreq():** Função que devolve a frequência do termo no documento;

universidade de aveiro

3.7 – Indexer

3.7.1 – Estrutura de dados usada

A estrutura de dados pensada e implementada tem como objetivo principal o aumento do desempenho da pesquisa, recorrendo à indexação dos termos de um dado conjunto de documentos. Neste caso, a estratégia utilizada será o "Inverted Index" que consiste em, para cada termo, guardar uma referência com a identificação do documento em que está presente, bem como o seu número de ocorrências nesse documento (frequência). Nesta primeira fase do projeto, considerámos que a melhor estrutura de dados a adotar seria:

- **LinkedList<Posting> postings:** Este tipo de dados herda as características de uma LinkedList composta por elementos do tipo Posting que são caracterizados por um identificador do documento (docId) e a respetiva frequência do token nesse documento.

- **TreeMap<String, LinkedList<Posting>> tokenDocIdFreq:** Este tipo de dados herda as características de uma TreeMap, em que a chave corresponde ao token e o valor corresponde à LinkedList referida anteriormente.

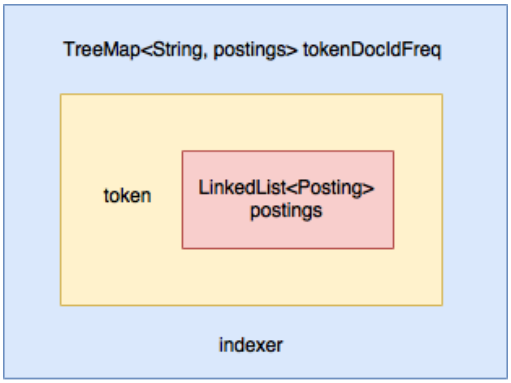

Figura 2 - Ilustração da estrutura de dados TreeMap<String, LinkedList<Posting>> tokenDocIdFreq

Figura 3 - Representação da estrutura interna de LinkedList<Posting> postings

Esta classe é responsável pela indexação dos termos. O texto de cada documento, já processado pelo Tokenizer, vai ser enviado em forma de lista juntamente com o identificador do documento para esta classe através da função addToSetAndCount. Este método converte a lista de termos validados para um set de strings e para cada token, procede-se à contagem da sua frequência. É então invocado, para cada termo do set, o método addToTokenDocIdFreq que recebe como parâmetros o identificador do documento, o token e a respetiva frequência. Aqui, caso a TreeMap não contenha esse token, é criado um novo elemento do tipo Posting(token, freq) e é adicionado esse Posting à LinkedList. Por fim, é associada a LinkedList à TreeMap. Caso contrário, caso a TreeMap contenha esse token, significa que a chave da TreeMap já existe. Neste caso, é necessário retornar a LinkedList que contém esse token, e só depois criar e adicionar um novo Posting à LinkedList. No fim, é necessário associar novamente a LinkedList à TreeMap.

Por fim, a função writeToFile vai gerar o ficheiro de saída, que tem o nome de "fileout.txt" localizado no diretório "output" do projeto. Este ficheiro de texto vai ser escrito com um termo por linha com o formato previamente indicado.

Métodos e a sua descrição:

- **addToSetAndCount (List<String> processedTerms, Integer docId):** A lista de termos processados é convertida num set de strings para se proceder à contagem da frequência de cada termo. É então invocado, para cada termo, o método addToTokenDocIdFreq, que irá ser responsável por adicionar os termos e a frequência à LinkedList e por fim, associar a LinkedList à TreeMap.

- **addToTokenDocIdFreq(String token, int docId, int termFreq):** Este método recebe um token, o identificador do documento e a frequência desse token. Caso a TreeMap não contenha esse token, é criado um novo elemento do tipo Posting(token, freq) e é adicionado esse Posting à LinkedList. Por fim, é associada a LinkedList à TreeMap. Caso contrário, caso a TreeMap contenha esse token, significa que a chave da TreeMap já existe. Neste caso, é necessário retornar a LinkedList que contém esse token, e só depois

criar e adicionar um novo Posting à LinkedList. No fim, é necessário associar novamente a LinkedList à TreeMap.

- **getTM():** Método que retorna a TreeMap<String, LinkedList<Posting>>.

- **writeToFile(String filename):** Método responsável por gerar o documento de texto com o formato previamente especificado (um termo por linha): termo,doc id:term freq,…

3.8 – DocumentProcessor

Nesta classe, são instanciadas várias classes: CorpusReader, SimpleTokenizer, Tokenizer e Indexer. Durante a instanciação do CorpusReader, é criada uma lista com os caminhos de todos os ficheiros da coleção a ser lida. Ainda com o auxílio desta classe, são percorridos todos os documentos e é obtido o identificador de cada documento bem como todo o texto presente nesse documento. Em seguida, o texto irá ser enviado para um dos Tokenizers (passado por argumento na main, irá ser explicado mais abaixo) com a finalidade de fazer o processamento desse texto. Caso seja utilizado o tokenizer complexo, é instanciada a classe Stop Words que gera um array de strings com todas as palavras consideradas “stop words” e é também instanciada a classe Stemmer.

O método responsável pelo processamento do texto em ambos os tokenizers é o textProcessor que vai começar por colocar numa lista de strings os termos do texto do documento ignorando os sinais de pontuação e todos os caracteres não alfanuméricos. Em seguida, a lista é submetida a uma série de filtragens onde apenas são guardados os termos com 3 ou mais caracteres e feita a conversão para minúsculas. No caso do tokenizer complexo, a lista vai necessitar de filtragens extra, ou seja, cada termo é então, submetido a uma verificação, ou seja, se está contido ou não na lista de “stop words” através do método isStopWord. Caso seja “stop word”, é ignorada da lista. A seguir, é aplicado o stemming usando a classe Stemmer. Esta classe é responsável por obter o stemmer especificado, neste caso, o “porter” stemmer. Depois de tudo isso, é retornada a lista resultante de todas estas filtragens.

A lista resultante é, então enviada juntamente com o identificador do documento para o Indexer. Numa primeira fase é feita a contagem dos elementos da lista através do método addToSetAndCount e só posteriormente, serão adicionados os termos à TreeMap através do método addToTokenDocIdFreq. No fim da iteração, através do método writeToFile, definido na classe Indexer, vai-se proceder à escrita no ficheiro no formato indicado.

A classe ainda contém os métodos printVocabularySize, printFirstTermsInOneDoc e printNTermsHighDocFreq que têm a finalidade de responder às questões apresentadas no enunciado do projeto, na secção 4.

Métodos e a sua descrição:

- **process(String fileType):** Este método é chamado na classe main (Assigment1), e é responsável por iterar sobre toda a coleção dos documentos de modo a fazer o processamento do texto dos documentos.

- **printVocabularySize():** Método que imprime o tamanho do vocabulário;

- **printFirstTermsInOneDoc(int nTerms, int nDocs):** Método que percorre a lista dos termos e imprime os N primeiros termos (passados como parâmetro nesta função - nTerms) que aparecem num dado número de documento (passados como parâmetro desta função - nDocs) e são ordenados alfabeticamente.

- **printNTermsHighDocFreq(int nTerms):** Método que imprime os N termos (passados como argumento - nTerms) com maior frequência de repetição em todos os documentos.

3.9 – Assigment1 (main)

Classe principal e executável do projeto. Aqui, é instanciado o DocumentProcessor e é invocado o método process desse módulo.

4 - Diagrama geral de fluxos

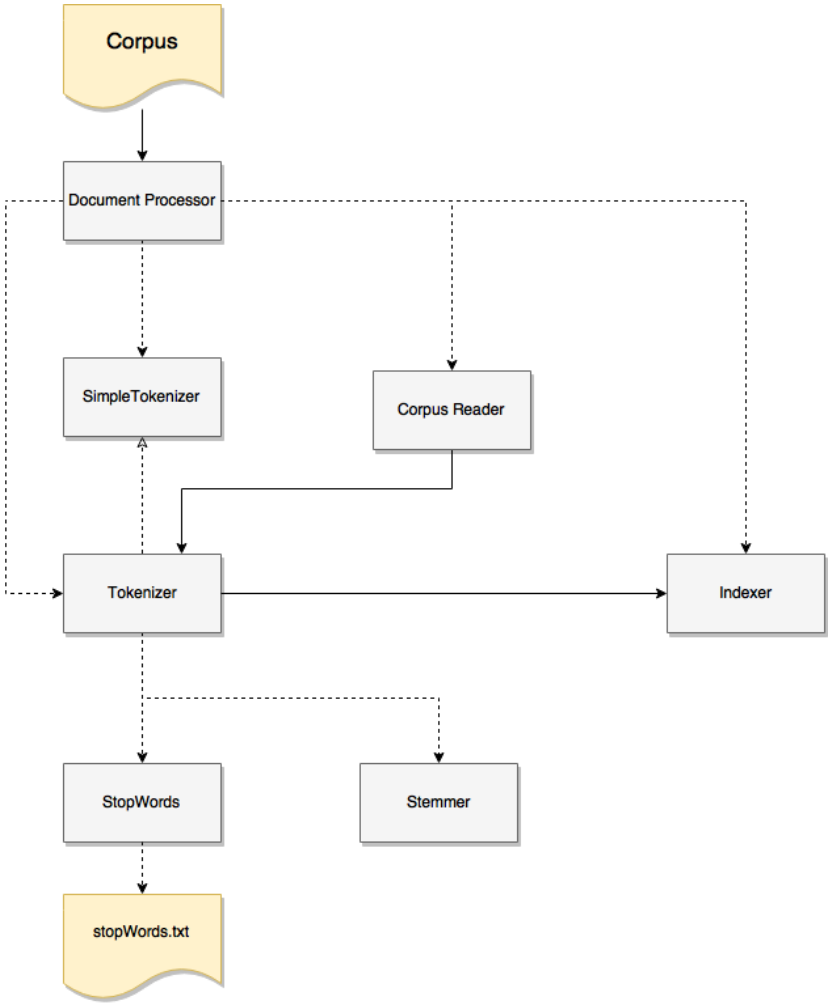

Figura 4 - Imagem que representa o fluxo de execução do programa

A figura anterior mostra-nos de forma esquemática o fluxo de dados no nosso projeto. Este fluxo já foi explicado anteriormente na descrição da classe DocumentProcessor.

6 - Estrutura do código

A estrutura do nosso projeto Maven encontra-se organizada da seguinte forma:

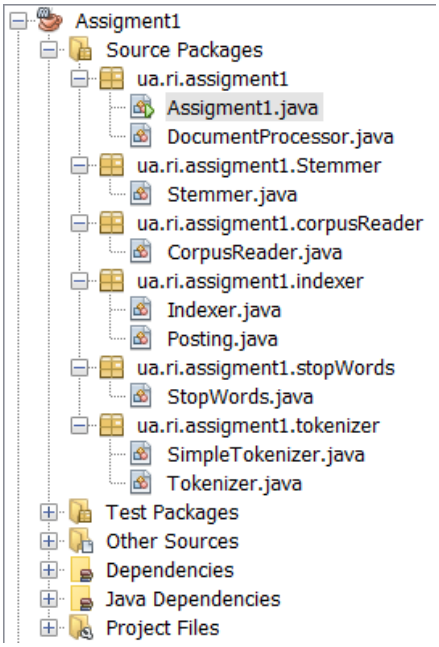

Figura 5 - Descrição da estrutura do nosso projeto Maven

No diretório raiz do nosso projeto Maven é possível encontrar três diretórios:

• **data**: Diretório onde se encontram a coleção de corpus utilizados. Para esta fase do projeto apenas foi utilizado o cranfield. Ainda neste diretório existe um ficheiro stopWords.txt que contém todas as palavras consideradas “stopwords”.

• **output**: Este diretório vai conter o ficheiro gerado resultante do processo de indexação. O ficheiro vai ter o nome “fileout.txt”.

• **Dependencies**: Neste diretório estão colocadas as bibliotecas compiladas necessárias para este projeto (biblioteca .jar do Snowball).

7- Bibliotecas Externas

The Porter stemming algorithm: http://snowball.tartarus.org/.

8- Execução

Para executar o nosso projeto é necessário a seguinte abordagem:

• Compilar e executar o nosso projeto recorrendo ao Netbeans.

• Podem ser passados, por argumento, o tipo de tokenizer que se pretende correr e as tags xml que irão ser usadas para pesquisa do identificador do documento, título e texto (DOCNO, TITLE e TEXT).

Modo de uso: java Assignment1 <tokenizerType> <DOCNO> <TITLE> <TEXT>

Por defeito, é considerado o tipo de Tokenizer complexo e são consideradas as tags xml dos documentos do corpus atual, por isso, apenas é necessário correr o comando: java Assignment1 **ou** java Assignment <tokenizerType>

9- Resultados

Para obtermos os tempos que do processo de indexação e respetiva geração e escrita em ficheiro, foi utilizado o seguinte hardware:

• **Processador**: Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz

• **Memória**: 8.00 GB

• **Sistema Operativo**: Windows 10, 64bits

Para o corpus fornecido, cranfield.zip, foram obtidos os seguintes tempos:

Simulação	Tempo de execução (s)
1	2.269
2	2.302
3	2.279
4	2.278

O tempo médio de execução foi: 2.282 segundos.

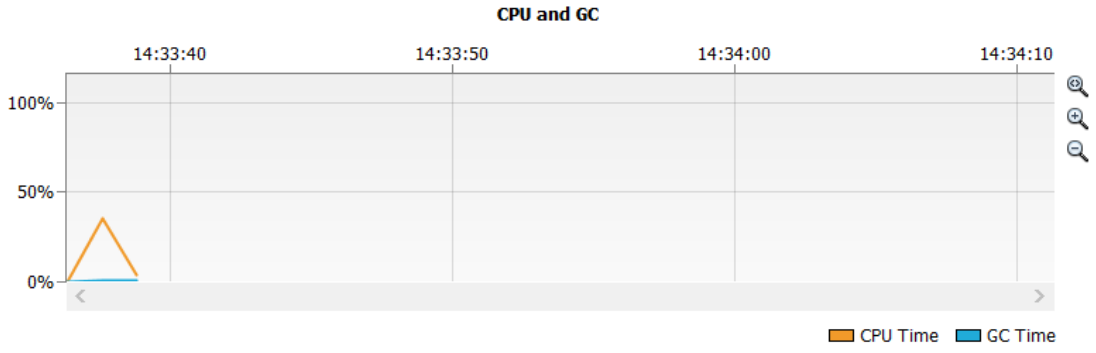

Figura 6 - Descreve a CPU Time durante a execução do projeto

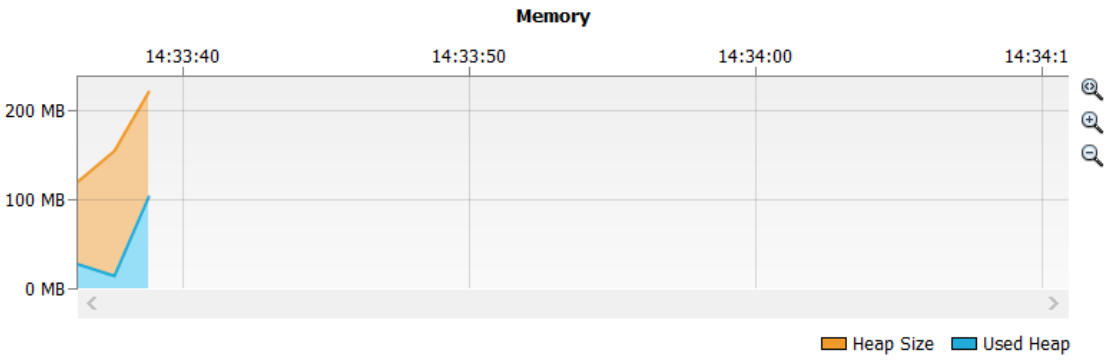

Figura 7 - Traduz o consumo de memória na execução do programa

Assim, por análise do gráfico podemos concluir que a quantidade de memória utilizada foi de 100 MB.

universidade de aveiro

10 – Respostas às questões Secção 4.

Nesta secção, iremos descrever os resultados obtidos nas perguntas **a), b), c), d)** da alínea **4** do assigment1 relativamente ao Simple Tokenizer e ao Tokenizer.

Para o **Simple Tokenizer**, os resultados foram os seguintes:

```
Document Processor starting to process files ...
Using Simple Tokenizer class to process terms ...
################################
Vocabulary size: 7133
################################
Printing the 10 first terms (in alphabetic order) that appear in 1 doc(s):
abbreviated
ablatedlength
ablative
abovementioned
absent
absorbing
abundantly
academic
accelerates
accelerators
################################
Printing the 10 terms with higher document frequency:
the -> 1391
and -> 1323
for -> 1144
are -> 1029
with -> 1010
that -> 807
flow -> 701
this -> 655
from -> 621
results -> 597
```

Para o **Tokenizer**, os resultados foram os seguintes:

```
Document Processor starting to process files ...
Using Tokenizer class to process terms ...
################################
Vocabulary size: 4429
################################
Printing the 10 first terms (in alphabetic order) that appear in 1 doc(s):
abbrevi
ablatedlength
abovement
absent
abundantli
academ
acceleromet
accentu
access
accident
################################
Printing the 10 terms with higher document frequency:
flow -> 729
result -> 692
number -> 571
pressur -> 551
effect -> 541
us -> 514
present -> 507
boundari -> 467
obtain -> 464
method -> 455
```

11 - Repositório de desenvolvimento

Para o desenvolvimento deste trabalho foi utilizado a plataforma **CodeUA** de forma a potenciar o trabalho de equipa. O repositório é o seguinte: http://code.ua.pt/projects/ir/repository.

12 – Comentários

- Inicialmente tínhamos optado por utilizar a seguinte estrutura de dados, HashMap<String, HashMap< Integer, Integer>>, para o processo de indexação, porém desta forma, para além de não ser a mais eficiente, não fazia a ordenação dos termos. Assim, passámos a usar a estrutura TreeMap < String, Linkedlist <Posting>>, pois desta forma podemos potenciar a tarefa de pesquisa nos trabalhos futuros.

- De modo a tornar a nossa implementação mais modular, optamos por definir um argumento que indica o tipo de ficheiros que o Corpus possui, neste caso, XML. A nossa implementação também permite especificar as tags dos documentos XML a processar, para o caso em que os documentos deste tipo mudem de estrutura. No entanto, se o tipo de ficheiros do Corpus mudar, apenas teremos de especificar o tipo desses ficheiros. No momento, como apenas temos documentos xml no nosso corpus, apenas temos implementado o processamento de ficheiros deste tipo.

13 - Conclusão

Na elaboração deste assignment pretendeu-se aplicar os conceitos teóricos iniciais de Recuperação de Informação de modo a consolidá-los, tendo em conta, a elaboração de um projeto modular e utilizando estruturas de dados que potenciasse a eficiência do nosso primeiro Indexer.

Dado que esta é a primeira abordagem do projeto global da unidade curricular, temos como principal preocupação de nas seguintes tarefas aperfeiçoar e corrigir os erros que possam existir, de modo a construir um projeto o mais eficiente e coeso possível.