universidade de aveiro

ASSIGNMENT 2 - RELATÓRIO

Simple Searcher

Recuperação de Informação

Curso: Engenharia de Computadores e Telemática

Departamento de Eletrónica, Telecomunicações e Informática

24 de outubro de 2017

Professor:

Prof. Sérgio Matos

Autores:

Miguel Oliveira nº 72638

Tiago Henriques nº 73046

universidade de aveiro

Índice

1 – Introdução

Com base no Assignment 1, onde foi criado um primeiro mecanismo de indexação (Indexer), o Assignment 2 tem como objetivo a criação de um simple searcher que utiliza o indexer elaborado anteriormente.
Neste relatório será feita uma descrição detalhada das classes e métodos usados, o "data flow" do funcionamento geral da execução, bem como as instruções e as estruturas usadas na elaboração do programa. Serão ainda referidas as alterações efetuadas em relação à primeira iteração.
Por fim, serão expostos os resultados da execução do programa como o tempo e a memória total usada durante a execução do mesmo.

2 – Alterações em relação à primeira entrega

A execução do programa da primeira entrega, Assignment1.java, mantém-se igual, mas agora especificamos um nome diferente para o ficheiro a ser gerado como output. Para gerar um ficheiro de saída relativo ao tokenizer simples é necessário correr o executável da primeira entrega (Assignment1.java) da seguinte forma: "*java Assignment1 simple*". Para gerar um ficheiro de saída que corresponda ao tokenizer complexo, é necessário correr o ficheiro executável da primeira entrega da seguinte forma: "*java Assignment1*" ou "*java Assignment1 tokenizer*", ou seja, por default, é considerado o tokenizer complexo.
Na eventualidade de ser utilizado o tokenizer simples, o ficheiro gerado vai ter o nome de "*fileout_simple.txt*". Caso o tokenizer a ser utilizado seja o complexo, o ficheiro gerado vai ter o nome "*fileout_tok.txt*". Ambos os ficheiros vão ser colocados no diretório com o nome "*output*". O motivo desta mudança prende-se com o facto de agora na segunda entrega ser necessário gerar dois ficheiros saída para cada tipo de tokenizer e o ficheiro de entrada ter de corresponder ao tipo de tokenizer a ser utilizado.

3 - Diagrama de Classes

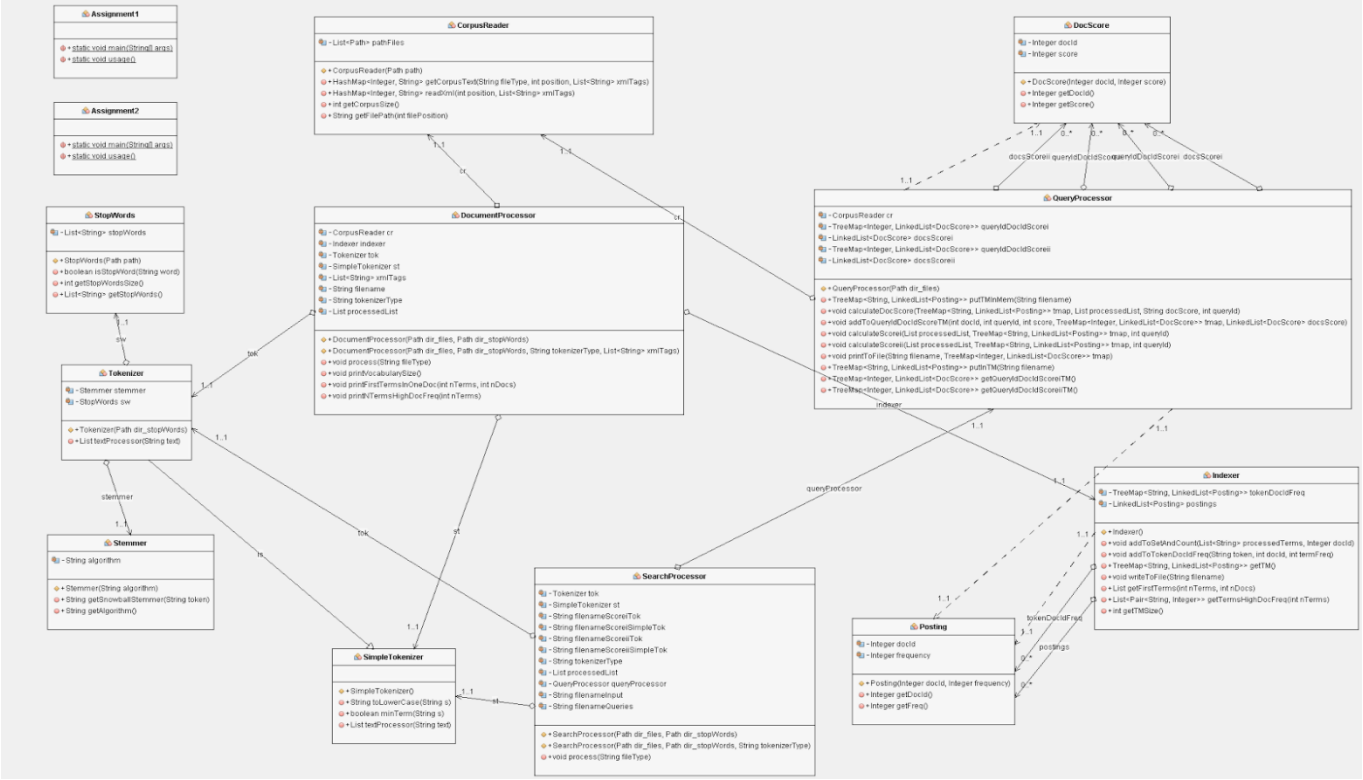

Figura 1 - Diagrama de Classes onde estão presentes as classes e os seus respetivos métodos

3 – Classes e Métodos

3.1 – Assignment2 (main)

Classe principal e executável do projeto. Aqui, é instanciado o SearchProcessor que iremos descrever mais abaixo.

3.2 – SearchProcessor

Esta classe tem como objetivo de instanciar todo o fluxo associado à pesquisa, isto é, é responsável por ler o ficheiro que contém as queries, neste caso, “cranfield.queries.txt”. As queries vão ser lidas

linha a linha e é realizada a tokenização do conteúdo utilizando quer o Tokenizer quer o SimpleTokenizer, dependendo dos resultados que o utilizador queira. Aqui, são aplicadas as mesmas transformações de tokenização, usando as classes StopWord e Stemmer se for utilizado o tokenizer complexo, e são calculado os dois tipos de scoring. Por fim, são escritos os resultados associados em ficheiros diferentes da seguinte forma: "fileout_docscorei_tok.txt" para o primeiro tipo usando o Tokenizer complexo, e "fileout_docscoreii_tok.txt" para o score do segundo tipo usando na mesma o Tokenizer. Caso seja utilizado o tokenizer simples, os ficheiros terão o nome "fileout_docscorei_simple.txt" e "fileout_docscoreii_simple.txt".

Métodos e a sua descrição:

- **process(String fileType)**: Este método é responsável por instanciar o método putTMInMem(String filename) da classe QueryProcessor, onde vai ler o Indexer do disco. De seguida, é feita a leitura das queries linha a linha, onde é utilizado, para cada query, o mesmo método de processamento de texto usado na primeira iteração. É devolvido uma lista de termos já processados e posteriormente procede-se ao cálculo do score tendo por base o indentificador da query e a lista resultante de termos da query, através da invocação da função calculateDocScore(...) da classe QueryProcessor. Por fim, após iterar sobre todas as queries, os resultados dos dois tipos scoring são escritos em disco, através da função printToFile(...) da classe QueryProcessor, em ficheiros separados dependendo do tipo de tokenizer utilizado.

3.3 – DocScore

Esta classe serve como estrutura de dados composta pelo identificador do documento e o valor de score respetivo e é usada na classe QueryProcessor.

Métodos e a sua descrição:

- **getDocId**(): Método que retorna o docId, ou seja, o identificador do documento.
- **getScore():** Método que retorna o score de um termo num determinado documento.

3.4 – QueryProcessor

Esta classe é instanciada no SeachProcessor que é responsável pelo fluxo de execução do programa.
Esta classe tem como objetivo implementar os métodos associados ao carregamento do Indexer do disco e ao cálculo do score, ou seja, é nesta classe que primeiramente são carregados os termos existentes no Indexer criado na tarefa anterior para posteriormente serem utilizados para o cálculo do score. O score pode ser de dois tipos, cálculo do número de palavras da query que existem num documento ou cálculo da frequência total das palavras da query presentes num documento.
Nesta classe, é então implementado o cálculo do primeiro tipo de scoring usando o método calculateScorei, e o segundo tipo de scoring usando o calculateScoreii.
Por fim, através da função printToFile(...), os resultados de cada tipo de scoring são armazenados em disco pelas escritas destes em ficheiros distintos.

Métodos e a sua descrição:

- **putTMInMem**(String filename): Método que instancia o método putInTM(String filename), onde é definido o caminho do ficheiro de onde vai ser lido o indexer do disco. Na nossa implementação, este ficheiro está armazenado na pasta 'outputs' e é designado por "fileout_tok.txt" caso se queira indexar em memória o ficheiro previamente gerado pelo tokenizer complexo. Caso se queira indexar em memória o ficheiro previamente gerado pelo tokenizer simples, este ficheiro terá o nome "fileout_simple.txt". Este método retorna uma Treemap que contém os termos e a lista de Postings carregados do ficheiro previamente gerado pelo indexer.

- **calculateDocScore**(TreeMap<String, LinkedList<Posting>> tmap, List processedList, String docScore, int queryId): Método que tem como objetivo escolher a opção de scoring. A eleição da opção é feita através da String docScore definida nos parâmetros de entrada. Caso seja igual a "scorei", o scoring é baseado no número de palavras da query que aparece num documento e será instanciada a função calculateScorei(). Caso seja igual a "scoreii", o scoring neste caso resulta da frequência total das palavras da query presentes num documento e é invocada a função calculateScoreii().

- **addToQueryIdDocIdScoreTM**(int docId, int queryId, int score, TreeMap<Integer, LinkedList<DocScore>> tmap, LinkedList<DocScore> docsScore): Este método é chamado no fim das funções calculateScorei(...) e calculateScoreii(...) após ser calculado o valor de score para cada documento dada uma determinada query. Este método recebe como argumento o identificador do documento, o identificador da query, o valor de score e o respetivo treemap e lista ligada para aquele tipo de score. Esta função tem um comportamento similar à função addToTokenDocIdFreq(...) da primeira iteração. Aqui, é criado um novo elemento do tipo DocScore(docId, score). Caso a treeMap não contenha a chave do identificador de query passado como argumento, é criado uma nova lista ligada e é imediatamente adicionado a essa lista o elemento DocScore previamente criado. Por fim,

é colocado na treeMap como chave o identificador da query e como valor a lista ligada de doc Scores. Por outro lado, caso o identificador de query já exista como chave da treeMap, é retornada a lista ligada onde esse identificador de query é chave e só depois é adicionado o elemento DocScore. Por fim, é colocado na treeMap como chave o identificador da query e como valor a lista ligada de doc Scores.

- **calculateScorei**(List processedList, TreeMap<String, LinkedList<Posting>> tmap, int queryId): Método em que após os tokens do ficheiro onde se localizam as queries, neste caso, "cranfield.queries.txt" terem sido lidas e tratadas de acordo com o Tokenizer usado no Indexer, é procedido ao cálculo do score. Neste caso, é verificado o número de palavras da query que existam num documento. É então percorrida a lista de termos já processados da query e ao mesmo tempo verificados os termos lidos do indexer, definidos na chave do TreeMap. Caso a TreeMap contenha o termo da query, são de seguida verificados os documentos onde aparece o termo da query e guardado o identificador do documento e é então incrementado o score associado a esse documento. Por fim é invocado método addToQueryIdDocIdScoreTM(...) que já foi explicado mais acima.

- **calculateScoreii**(List processedList, TreeMap<String, LinkedList<Posting>> tmap, int queryId): Método em que após as palavras do ficheiro das queries, "cranfield.queries.txt", terem sido lidas e tratadas de acordo com o Tokenizer usado no Indexer, é procedido ao cálculo do score. Neste caso resulta da frequência total das palavras da query presentes num documento, ou seja, é incrementado a frequência definida nos Postings. É então percorrida a lista de termos da query e ao mesmo tempo verificados os termos lidos do indexer, definidos na chave do TreeMap. Caso a TreeMap contenha o termo da query, é de seguida incrementado a frequência do termo associado ao documento onde ocorre, sendo posteriormente guardado o identificador do documento associado. Por fim é invocado método addToQueryIdDocIdScoreTM(...) que já foi explicado mais acima.

- **printToFile**(String filename, TreeMap<Integer, LinkedList<DocScore>> tmap): Função que trata de escrever num ficheiro o valor de score resultante. Na nossa implementação os resultados dos dois tipos de scoring existentes são definidos em ficheiros diferentes, ou seja, são gerados dois ficheiros distintos e armazenados no diretório "output" da estrutura de ficheiros do projeto. A estrutura da escrita nos ficheiros obedece à seguinte forma: Qi doc_id doc_score.

- **putInTM**(String filename): Método que tem como principal objetivo funcionar como "index reader", onde carrega o indexer do disco. Ou seja, irá ler o ficheiro definido nos parâmetros de entrada linha a linha. Numa primeira fase, é feito o split por vírgulas, para aceder aos elementos de cada linha. De seguida, é realizado o split por dois pontos (":") de forma a aceder ao termo do indexer e à lista ligada de Postings associados (docId e a frequência). No fim, é utilizada a estrutura de dados TreeMap<String, LinkedList<Posting>>, onde a chave será o termo do indexer lido e atualizada a LinkedList de Postings correspondentes.

- **getQueryIdDocIdScoreiTM**(): Método que retorna a TreeMap getQueryIdDocIdScoreiTM associada ao primeiro tipo de score (scorei).

- **getQueryIdDocIdScoreiiTM**(): Método que retorna a TreeMap getQueryIdDocIdScoreiiTM correspondente ao segundo tipo de score (scoreii).

universidade de aveiro

Estrutura de dados usada:

A estrutura de dados pensada e implementada tem como objetivo principal o aumento do desempenho da pesquisa. Optámos por utilizar as mesmas estruturas de dados usadas na primeira iteração do trabalho, pois desta forma o pensamento para a pesquisa é feito de forma semelhante. Considerámos que a melhor estrutura de dados a adotar seria:

- **LinkedList docsScorei:** Este tipo de dados herda as características de uma LinkedList composta por elementos do tipo DocScore que são caracterizados por um identificador do documento (docId) e o respetivo score (score do tipo 1) do token nesse documento.

- **TreeMap<Integer, docsScorei> queryIdDocIdScorei:** Este tipo de dados herda as características de uma TreeMap, em que a chave corresponde ao identificador da query e o valor corresponde à LinkedList referida anteriormente.

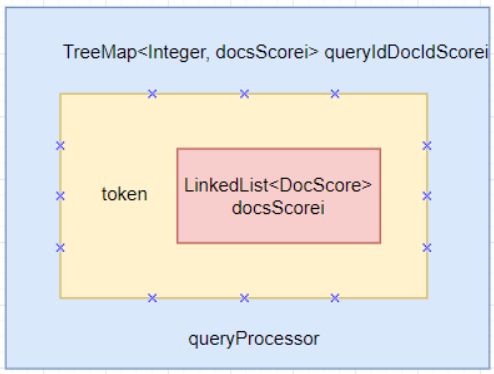

Figura 2 - Ilustração da estrutura de dados TreeMap<Integer,docsScorei> queyIdDocIdScorei

- **LinkedList docsScoreii:** Este tipo de dados herda as características de uma LinkedList composta por elementos do tipo DocScore que são caracterizados por um identificador do documento (docId) e o respetivo score (score do tipo 2) do token nesse documento.

- **TreeMap<Integer, docsScoreii> queryIdDocIdScoreii:** Este tipo de dados herda as características de uma TreeMap, em que a chave corresponde ao identificador da query e o valor corresponde à LinkedList referida anteriormente.

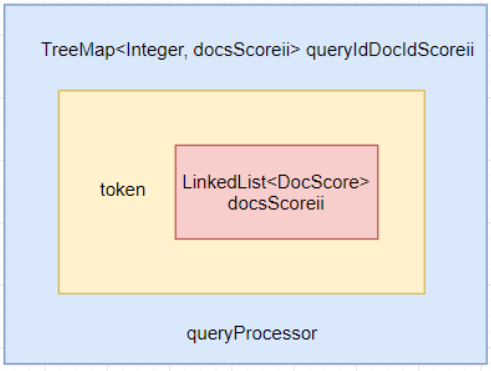

Figura 2 - Ilustração da estrutura de dados Tremap<Intger, docsScoreii> queryIdDocIdScoreii

4 - Diagrama geral de fluxos

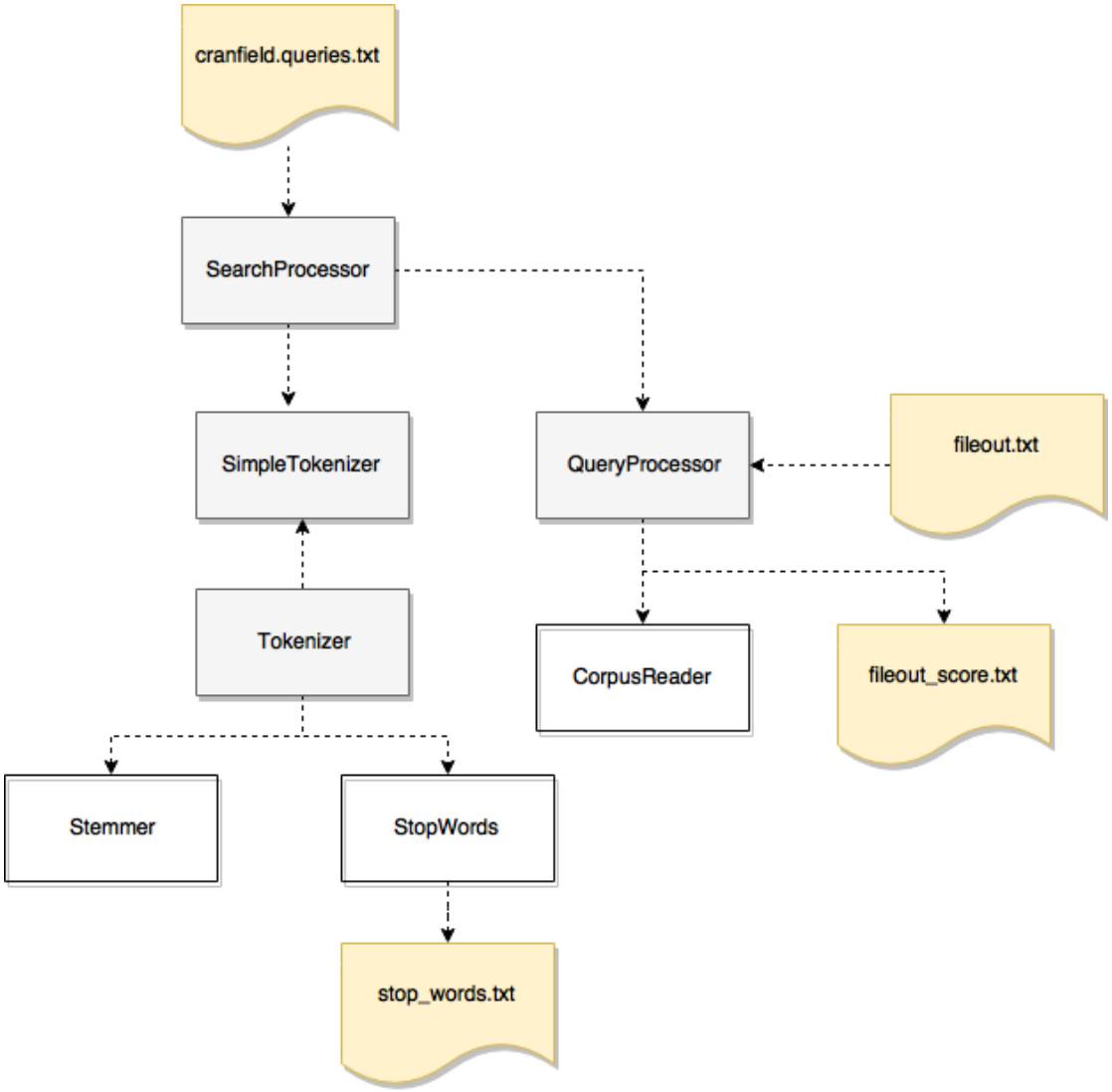

Figura 4 - Imagem que representa o fluxo de execução do programa

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 SearchProcessor.

universidade de aveiro

5 - Estrutura do código

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

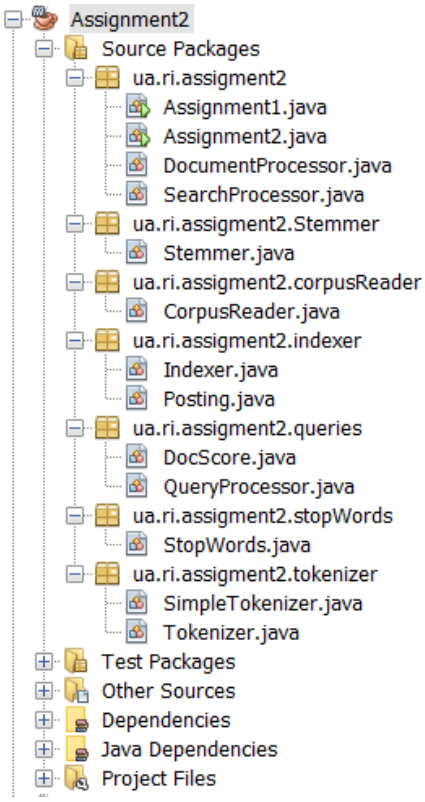

Figura 3 - Descrição da estrutura do nosso projeto Maven

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

- **data**: Este diretório, diferentemente ao da primeira iteração do trabalho, está agora dividido em diretórios. No diretório "cranfield", encontram-se a coleção de corpus utilizados. Para esta fase do projeto apenas foi utilizado o cranfield. No diretório "queries", encontra-se o ficheiro das queries considerado. Neste caso, apenas foi considerado o ficheiro com o nome "cranfield.queries.txt". Na raiz do diretório, existe um ficheiro stopWords.txt que contém todas as stopwords consideradas.

- **output**: Este diretório, diferentemente ao da primeira iteração do trabalho, vai agora conter mais ficheiros gerados. Este diretório vai conter o(s) ficheiro(s) gerado na primeira iteração resultante do processo de indexação. Este(s) ficheiro(s) terá(ão) o nome "fileout_simple.txt" caso tenha sido utilizado o simple tokenizer e o nome "fileout_tok.txt" no caso de ter sido utilizado o tokenizer complexo. Os ficheiros gerados nesta segunda iteração vão também ser colocados neste diretório. Serão gerados dois ficheiros para cada tipo de tokenizer utilizado, um para cada tipo de scoring. Os ficheiros terão o nome "fileout_docscorei_tok.txt" e "fileout_docscoreii_tok.txt" no caso de ser lido o ficheiro indexado pelo tokenizer complexo. No caso de ser lido o ficheiro indexado pelo tokenizer

simples, os ficheiros terão o nome “fileout_docscorei_simple.txt” e fileout_docscoreii_simple.txt”.

- **Dependencies**:. Neste diretório estão colocadas as bibliotecas compiladas necessárias para este projeto (Para este caso foi usado o jar do Snowball). (Relativo à primeira iteração do trabalho).

7- Bibliotecas Externas

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

De referir, que esta biblioteca externa apenas foi utilizada na primeira iteração do projeto.

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 corre.
  Modo de uso: java Assigment2 <tokenizerType>. No entanto, por defeito, é considerado o tipo de Tokenizer complexo, por isso, apenas é necessário correr o comando: java Assigment2. De notar que o tipo de tokenizer apenas aceita a string <tokenizer> que corresponde ao Tokenizer complexo ou a string <simple> que corresponde ao Simple Tokenizer. Caso seja introduzida uma outra string, nenhum tipo de tokenizer será reconhecido e será apresentada uma mensagem no terminal com o formato que deve ser utilizado para correr o programa.

universidade de aveiro

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 sample cranfield.zip e cranfield.queries.txt, foram obtidos os seguintes tempos:

Simulação	Tempo de execução (s)
1	6.793
2	6.379
3	6.275
4	6.413s

O tempo médio de execução foi: 6,465 segundos.

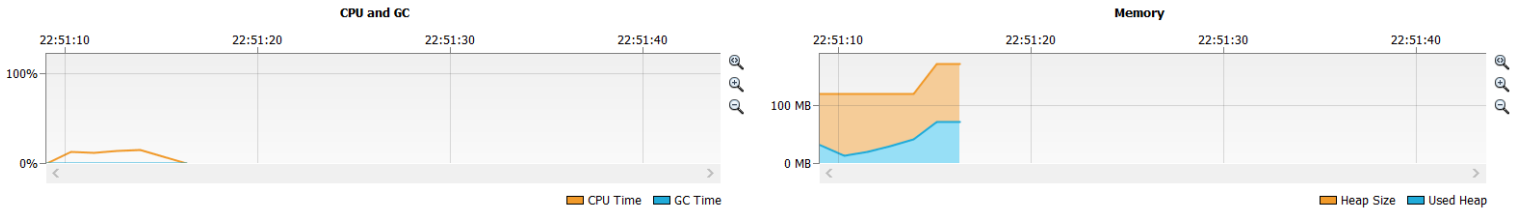

Figura 4 - Descrição de medidas de desempenho: CPU, Memória e Garbage Collection

Assim, por análise do gráfico podemos concluir que a quantidade de memória utilizada foi de aproximadamente 150 MB.

10 - 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.

11 - Conclusão

Na realização do Assignment2 pudemos rever a elaboração do programa da primeira iteração, onde verificamos questões de modularidade do código e onde tentaram ser corrigidos para uma versão mais correta do nosso ponto de vista. Neste relatório, cujo procede à descrição das novas classes e novos métodos, bem como ao fluxo do programa, não deixa de referenciar a contínua utilização de classes e métodos do Assignment1.
Assim, pretendemos ao longo das iterações produzir um trabalho cada vez mais modular e eficiente com base nos trabalhos realizados anteriormente.