**Montagem e avaliação de um mecanismo de recuperação de informação em memória**

Relatório do trabalho elaborado na disciplina Busca e Recuperação de Informação da COPPE/UFRJ, em julho de 2008

Trabalho realizado por:

Hua Lin Chang Costa – DRE 108079525

Marcelo Schots de Oliveira – DRE 108080063

Conteúdo:

1. Introdução

2. Projeto da Solução  
*2.1.* Pré-processamento dos Dados  
*2.2.* Estrutura de Indexação

3. Análise dos resultados  
*3.1.* Cálculo tf-idf  
*3.2.* Cálculo de Similaridade

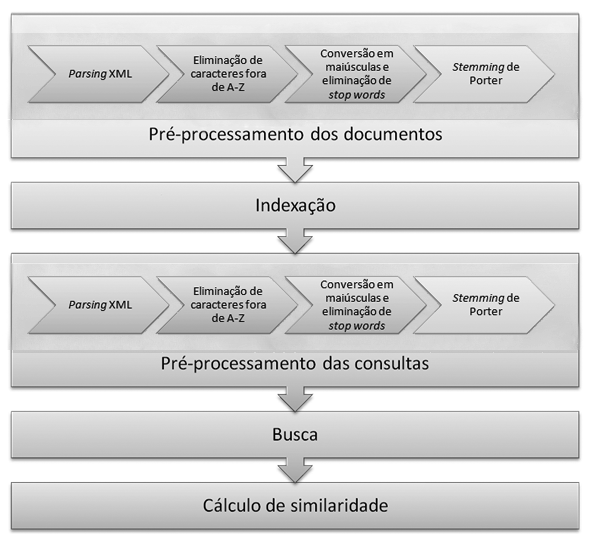
4. Resultados do Mecanismo de Busca

5. Anexos  
A. Funções responsáveis pelo pré-processamento do texto  
B. Classe responsável pela indexação de termos  
C. Classe responsável pela busca no mecanismo de indexação, cálculo de tf-idf, cálculo de similaridade e ranqueamento.

**1. Introdução**

Este relatório apresenta uma visão geral das etapas de montagem e avaliação de um mecanismo de recuperação de informação em memória. Este mecanismo consta das etapas de pré-processamento das informações textuais (contidas nos dados e nas consultas), busca e indexação. O mecanismo foi desenvolvido em Java.

**2. Projeto da Solução**  
  
A figura a seguir consta de um diagrama explicativo da solução projetada.



As seções a seguir apresentam como foram estruturadas as etapas deste processo efetuado pelo mecanismo.

**2.1. Pré-processamento dos Dados**

    A chamada aos módulos descritos a seguir é feita através do método preProcessText(String sourceText). Este método, por sua vez, fará a chamada a cada um dos módulos (cada módulo corresponde a um método) na ordem definida no diagrama da solução.  
O mecanismo de pré-processamento das informações é composto dos seguintes módulos:

* *Parsing* XML

    Este módulo realiza uma análise (*parsing*) do arquivo XML que contém as informações da coleção de dados.  
    Essas informações serão adicionadas em uma lista de objetos conforme o tipo de informação obtida: será gerada uma lista de objetos do tipo Record (quando forem lidas do arquivo XML as informações do registro) ou do tipo Query (quando forem lidas do arquivo XML as informações de busca).  
  
**Entrada:** o arquivo XML, informado através do caminho (*path*) no qual o mesmo se encontra  
**Saída:** uma lista de objetos do tipo associado à informação lida, conforme o método chamado: para o tipo Record, o método Record.parseRecordFromXML(String filePath)), e para o tipo Query, o método Query.parseQueryFromXML(String filePath)).

* Eliminação de caracteres fora de A-Z

    Este módulo elimina todo e qualquer caractere que não pertença ao alfabeto latino (ou seja, que não esteja entre as 26 letras de A e Z), se existir.  
    Também é feita neste módulo a remoção de ocorrências de caracteres de espaço (*whitespace characters*) em excesso (considera-se como excesso a ocorrência de um ou mais caracteres de espaços adjacentes a outro caractere de espaço). Este passo é importante para o módulo de conversão em maiúsculas e eliminação de *stop words*.  
  
**Entrada:** um objeto String composto pelas palavras  
**Saída:** um objeto String que contém apenas letras maiúsculas e minúsculas (formando as palavras) e caracteres de espaço (separando uma palavra da outra)

* Conversão em maiúsculas e eliminação de *stop words*

    Este módulo faz com que todo e cada caractere em letra minúscula seja convertido em letra maiúscula e, em seguida, elimina as *stop words* encontradas no objeto String de entrada. A chamada a este módulo requer o pré-carregamento de uma lista de *stop words*, através do método loadListOfStopWords(String filePath).  
  
**Entrada:** um objeto String composto pelas palavras  
**Saída:** uma lista de objetos String, onde cada String representa uma palavra, e cada palavra pertencente à lista está em maiúsculas e não é uma *stop word*

* *Stemming* de Porter

    Este módulo executa o *stemming*, que extrai o radical das palavras de forma a efetuar uma busca mais abrangente, contendo um maior número de termos sem o viés de prefixos e sufixos que tendem a tornar a busca muito específica em alguns casos. A classe Stemmer foi obtida no endereço <http://tartarus.org/~martin/PorterStemmer/java.txt> e adaptada ao projeto.  
  
**Entrada:** uma lista de objetos String na qual cada String é uma palavra  
**Saída:** uma lista de objetos String na qual cada String é um radical

**2.2. Estrutura de Indexação**

    A estrutura do mecanismo de indexação foi baseada no conceito de lista invertida, no qual uma chave secundária é associada a uma lista de chaves primárias. Neste trabalho, cada chave (radical) contém uma ligação para os documentos que a contém.  
    É utilizada uma estrutura *hash* em memória. Tal estrutura foi definida de forma que, para cada radical, é associada uma lista, indicando: a) em que documentos este radical aparece; e b) quantas vezes este radical aparece em cada documento. Sendo assim, o radical é a chave da estrutura *hash*.

**3. Análise dos resultados**

**3.1. Cálculo tf-idf**

O mecanismo de busca realiza a consulta na estrutura de indexação a partir dos termos resultantes do pré-processamento da pergunta. O mecanismo de busca retorna todos os documentos e possuem pelo menos um dos termos contidos na pergunta, realizando assim uma consulta através do OU de termos da consulta.

Com o objetivo de atribuir pesos a cada uma das palavras em comum entre a pergunta e os documentos retornado, foi utilizado o cálculo de tf-idf.

As seguintes definições foram utilizadas:

**tf** (*term frequency*): freqüência de cada termo de dentro de um documento



Onde:

a = número de vezes que um termo aparece em um documento

b = número termos dentro de um documetno

idf (the inverse document frequency) : inverso da freqüência de um termo numa coleção de documentos.



Onde:

N = número de documentos na coleção

n = número de documentos da coleção que possuem determinado termo

Assim a seguinte a matriz pode ser obtida: termos X documentos. Que relaciona cada termo da pergunta (eixo y) com o seu peso dentro de um documento (eixo x).

  
Onde:

t = número de termos na pergunta.

d = número de documentos que contém pelo menos um termo da pergunta

x = tf-idf de determinado termo dentro de determinado documento

**3.2. Cálculo de Similaridade**

Com o objetivo de ranquear os resultados de uma consulta, foi utilizado um cálculo de similaridade entre vetores.

As seguintes definições foram utilizadas:



Onde:

A = peso dos termos em uma pergunta, consideramos para uma pergunta o peso de todos os termos sendo 1.

B = peso dos termos em um documento, consideramos para cada pergunta as colunas na matriz Termos X Documentos

Assim:





Logo:

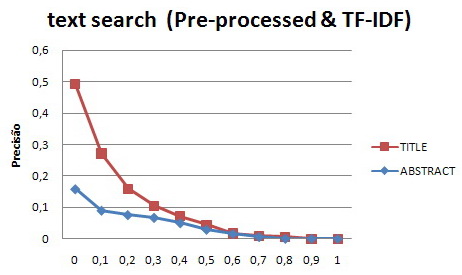
   
 Com isso obtemos um valor de similaridade de cada documento da matriz *‘Termos X Documentos’* com a pergunta em questão. Assim esse conjunto de documentos pode então ser ordenado (ranqueado), através de seus valores de similaridade.

**4. Resultados do Mecanismo de Busca**

  O mecanismo de busca foi avaliado utilizando-se apenas dos campos TITLE e ABSTRACT. O gráfico abaixo demonstra o resultado obtido através dessa avaliação.

O valor de precision recall foi obtido através da comparação dos documentos retornados pela consulta com os documentos definidos como relevantes nos documentos XML de entrada.

Os valores ainda foram interpolados para melhor visualização do gráfico.

****

**5. Anexos**

**A. Funções responsáveis pelo pré processamento do texto**

**A.1. Remoção de Caracteres diferentes de letras**

/\*\*

\* Removes all the non-letter characters from the input string, if any.

\* Also removes excessive whitespace occurrences (when there's one or more

\* whitespace adjacent to another whitespace).

\* **@param** source - a string

\* **@return** the content of the input string with only letter and whitespace characters

\*/

**public** String removeNonLetterCharacters(String source) {

String result = source;

// replaces all non-letter characters by a whitespace

result = result.replaceAll("[^a-zA-Z]"," ");

// removes excessive whitespace occurrences

result = result.replaceAll("[ ]+"," ");

**return** result;

}

**A.1. Remoção de Stop Words**

/\*\*

\* Removes stop words (defined in the list of stop words),

\* if any, from the input string.

\* A list of stop words <i>must</i> be loaded by the method

\* <code>loadListOfStopWords(String filePath)</code>

\* before calling this method, otherwise there will be

\* no stop word removed from the input string.

\* **@param** source - a string

\* **@return** a list of all the words from the input string,

\* except the stop words

\*/

**public** List<String> removeStopWords(String source) {

String[] words = source.split(" ");

List<String> listOfWords = **new** Vector<String>();

**for** (**int** i = 0; i < words.length; i++) {

String word = words[i].toUpperCase();

// this algorithm is not considering composite terms

**if** (!listOfStopWords.contains(word)) {

listOfWords.add(word);

}

}

**return** listOfWords;

}

/\*\*

\* Loads a list of stop words from the input file path

\* so that these words can be removed thereafter by the method

\* <code>removeStopWords(String source)</code>.

\* The file must have <i>only one word</i> per line.

\* **@param** filePath - the file path to the list

\* **@return** true if loaded successfully, false otherwise

\*/

**public** **void** loadListOfStopWords(String filePath) {

**try** {

// open the file using the path defined in filePath parameter

FileInputStream fileInputStream = **new** FileInputStream(filePath);

// get the object of DataInputStream

DataInputStream in = **new** DataInputStream(fileInputStream);

BufferedReader bufferedReader = **new** BufferedReader(**new** InputStreamReader(in));

String stringLine;

// read the file line by line

**while** ((stringLine = bufferedReader.readLine()) != **null**) {

listOfStopWords.add(stringLine.trim());

}

// close the input stream

in.close();

} **catch** (FileNotFoundException e) {

System.*err*.println("Error: " + e.getMessage());

e.printStackTrace();

} **catch** (IOException e) {

System.*err*.println("Error: " + e.getMessage());

e.printStackTrace();

}

}

**B. Classe responsável pela indexação de termos**

**B.1. Classe Indexing**

**package** br.ufrj.cos.disciplina.bri.indexing;

**import** java.util.Hashtable;

**import** java.util.List;

**import** java.util.Vector;

**import** br.ufrj.cos.disciplina.bri.indexing.model.RadixInfo;

**public** **class** Indexing {

/\*\*

\* The hash will be composed as follows: The radix will be the key of the

\* hash The associated element will be a list of RadixInfo, where RadixInfo

\* objects contains information about the document where a radix occurs and

\* how much times this radix occurs.

\*/

**private** Hashtable<String, List<RadixInfo>> hash;

/\*\*

\* Default constructor method.

\*/

**public** Indexing() {

hash = **new** Hashtable<String, List<RadixInfo>>();

}

/\*\*

\* Adds a certain radix and its information.

\* **@param** radix - the radix to be added

\* **@param** info - the information about the radix

\*/

**public** **void** addToHash(String radix, RadixInfo info) {

**if** (hash.containsKey(radix)) {

// hash key already exists, just add the information to the list

(hash.get(radix)).add(info);

} **else** {

// create the list, add the information and put it on the hash

List<RadixInfo> list = **new** Vector<RadixInfo>();

list.add(info);

hash.put(radix, list);

}

}

/\*\*

\* **@return** the hash table

\*/

**public** Hashtable<String, List<RadixInfo>> getHash() {

**return** hash;

}

}

**B.2. Classe RadixInfo**

**package** br.ufrj.cos.disciplina.bri.indexing.model;

**public** **class** RadixInfo {

// shows the document id where a radix occurs

**private** **int** documentId;

// shows the frequency of a radix term in the document

**private** **double** tf;

// specifies the section of the document (TITLE, ABSTRACT)

**private** String part;

/\*\*

\* Default constructor method.

\* **@param** documentId - indicates what document contains the radix

\* **@param** numberOfOccurrences - show the number of occurrences of the radix in the document

\*/

**public** RadixInfo(**int** documentId, **double** tf, String part) {

**this**.documentId = documentId;

**this**.tf = tf;

**this**.part = part;

}

/\*\*

\* **@return** the documentId where a radix occurs

\*/

**public** **int** getDocumentId() {

**return** documentId;

}

/\*\*

\* **@param** documentId the documentId to set

\*/

**public** **void** setDocumentId(**int** documentId) {

**this**.documentId = documentId;

}

/\*\*

\* **@param** tf - the term frequency to set

\*/

**public** **void** setTf(**double** tf) {

**this**.tf = tf;

}

/\*\*

\* **@return** the term frequency

\*/

**public** **double** getTf() {

**return** tf;

}

/\*\*

\* **@return** the part

\*/

**public** String getPart() {

**return** part;

}

/\*\*

\* **@param** part the part to set

\*/

**public** **void** setPart(String part) {

**this**.part = part;

}

@Override

**public** String toString() {

**return** "Record: " + documentId + " - tf: " + tf + " - Where: " + part;

}

}

**C. Classe responsável pela busca no mecanismo de indexação, cálculo de tf-idf, cálculo de similaridade e ranquamento.**

**package** br.ufrj.cos.disciplina.bri.algorithm;

**public** **class** VectorSearch {

Hashtable<Integer, List<Double>> docMatrix;

/\*\*

\* Default constructor method.

\*/

**public** VectorSearch() {

docMatrix = **new** Hashtable<Integer, List<Double>>();

}

/\*\*

\* Executes vector search, obtaining ranking values.

\* **@param** questionTerms

\* **@param** documents

\* **@param** numberOfDocs

\* **@return** a list with ranking values

\*/

@SuppressWarnings("unchecked")

**public** List<Integer> vectorSearch(Set<String> questionTerms, Indexing documents, **int** numberOfDocs) {

**int** position = 0;

// creating a temporary list of double values

// so that it won't be necessary

// creating a loop inside another loop

List<Double> temporaryList = **new** ArrayList<Double>();

**for** (**int** i = 0; i < questionTerms.size(); i++) {

temporaryList.add(0.0);

}

**for** (Iterator<String> iteratorQuery = questionTerms.iterator(); iteratorQuery.hasNext();) {

String term = iteratorQuery.next();

**double** idf;

**if** (documents.getHash().containsKey(term)) {

**double** numberOfRelevantDocs = documents.getHash().get(term).size();

idf = numberOfDocs / numberOfRelevantDocs;

idf = Math.*log*(idf) / Math.*log*(2);

List<RadixInfo> listDocInfo = documents.getHash().get(term);

**for** (Iterator<RadixInfo> iteratorDoc = listDocInfo.iterator(); iteratorDoc.hasNext();) {

RadixInfo doc = iteratorDoc.next();

**double** tfIdf = 0.0;

tfIdf = doc.getTf()\*idf;

**if**(!docMatrix.containsKey(doc.getDocumentId())) {

docMatrix.put(doc.getDocumentId(), **new** ArrayList<Double>(temporaryList));

}

docMatrix.get(doc.getDocumentId()).add(position, tfIdf);

}

}

position++;

}

Hashtable<Integer, Double> ranking = **new** Hashtable<Integer, Double>();

**for**(Enumeration<Integer> n = docMatrix.keys(); n.hasMoreElements();) {

**int** key = n.nextElement();

ranking.put(key, calculateSimilarity(docMatrix.get(key)));

}

// ordering by similarity rating

ArrayList tempRanking = **new** ArrayList(ranking.entrySet());

Collections.*sort*(tempRanking, **new** SimilarityComparator());

List<Integer> result = **new** ArrayList<Integer>();

Iterator rankingIterator = tempRanking.iterator();

**while**(rankingIterator.hasNext()) {

Map.Entry e = (Map.Entry)rankingIterator.next();

result.add((Integer)e.getKey());

}

System.*out*.println("Consulta: " + result);

**return** result;

}

/\*\*

\* Calculates similarity between document and question vectors.

\* **@param** documentsWithRelevantTerms

\* **@return** the similarity value

\*/

**public** **double** calculateSimilarity(List<Double> documentsWithRelevantTerms) {

// similarity calculation is a result of dividing

// the dot product between document and question vectors

// by the product of multiplying its isolated norm values

// let D be the document vector and Q the question vector:

// sim = ( Q · D ) / ( ||Q|| \* ||D|| )

// here, the question vector Q is (1, 1, 1, ... , 1)

// (all its components equals 1)

// so, the dot product can be just a sum of the

// values of the document vector components

// furthermore, the norm calculus is as follows:

// norm(V) = || V || = sqrt(sum(xi)^2), i=1..n

// where xi are the vector V components and

// n is the number of components

// for vector Q (whose all components values equals 1),

// || Q || = sqrt(n)

// for vector D, default calculation is used

// sum = dot product

**double** sum = 0.0;

// norm of document vector

**double** normDoc = 0.0;

// norm of question vector

**double** normQuestion = 0.0;

**for** (**int** i = 0; i < documentsWithRelevantTerms.size(); i++) {

sum += documentsWithRelevantTerms.get(i);

normDoc += Math.*pow*(documentsWithRelevantTerms.get(i), 2);

}

// as explained, only the number of components is needed

normQuestion = Math.*sqrt*(documentsWithRelevantTerms.size());

// finishing document vector calculation

normDoc = Math.*sqrt*(normDoc);

// calculating similarity

**double** similarity = (sum / (normDoc \* normQuestion));

**return** similarity;

}

/\*\*

\* Inner class for helping vector search.

\*/

@SuppressWarnings("unchecked")

**static** **class** SimilarityComparator **implements** Comparator {

@SuppressWarnings("unchecked")

**public** **int** compare(Object obj1, Object obj2) {

**int** result = 0;

Map.Entry e1 = (Map.Entry) obj1;

Map.Entry e2 = (Map.Entry) obj2;// Sort based on values.

Double value1 = (Double) e1.getValue();

Double value2 = (Double) e2.getValue();

**if** (value1.compareTo(value2) == 0) {

Integer word1 = (Integer) e1.getKey();

Integer word2 = (Integer) e2.getKey();

// sort string in an ascending order

result = word1.compareTo(word2);

} **else** {

// sort values in a descending order

result = value2.compareTo(value1);

}

**return** result;

}

}

}