Master 1 Informatique 2013-2014

Mise un place d'un logiciel de recherche documentaire via la méthode booléenne en Java

Université Angers - Faculté des Sciences

Enseignant :Bernard Levrat
Groupe : Dorian Coffinet, Thibault Gauthier, Yassine Badih

I) Environnement de développement

L'ensemble du projet est codé dans un environnement Linux. Les machines utilisent un environnement Ubuntu. Nous avons choisi de codé le projet à l'aide du langage JAVA 7. Il nous semblais plus facile de le réaliser dans ce langage car chacun d'entre nous est à l'aise dans ce dernier. De plus nous avons utilisé un logiciel de "versionning" pour le projet à l'aide de Git et GitHub. Il nous a paru essentiel d'avoir ce type d'outils lorsqu'on travail à plusieurs sur un projet.

II) Répartition des tâches

a) Développement itératif

Pour arriver à réaliser le projet dans le temps imparti nous avons découper le projet en plusieurs phases. Ces derniers sont scindés en différentes tâches :

1. Lecture des fichiers
    - Lecture de l'ensemble des texte du corpus

2. "Stemming" pour un fichier donné
    - Création du parser (Stopword)
    - Création du "Stemmer"

3. Gestion du dictionnaire
    - Représentation du dictionnaire
    - Écriture du dictionnaire

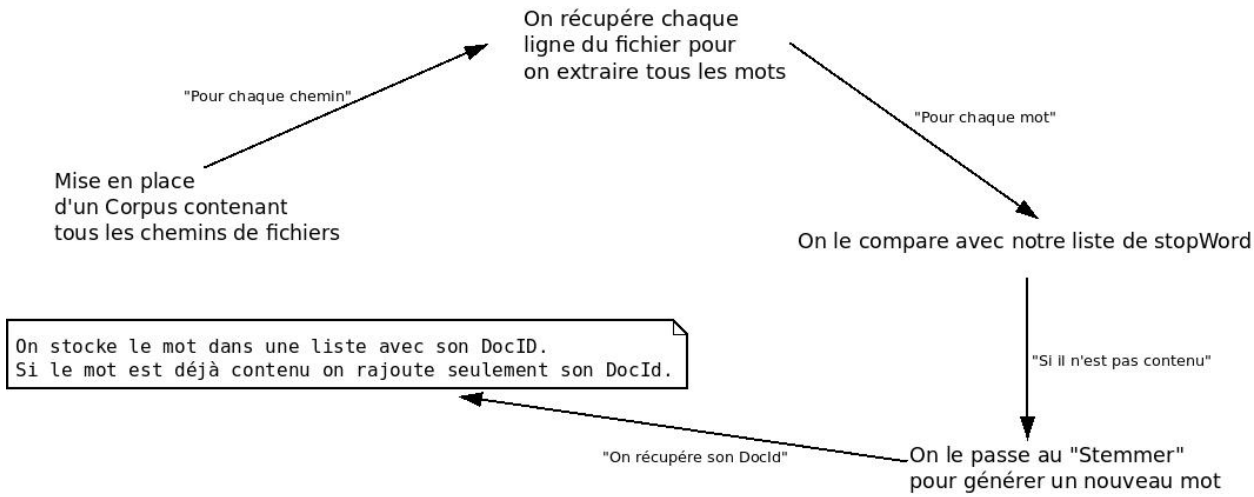
4. Gestion de la "Query" utilisateur
    - Création d'un algorithme booléen
    - Création d'un algorithme par défault

5. Interface graphique
    - Création d'une interface graphique

Entre chaque phases nous avons fait un travail de mise en commun pour relier chaque code. Ceci nous a permis à chaque étape d'avoir une version stable et fonctionnel.

III) Fonctionnement du projet

a) Génération du dictionnaire



Schémas de génération du dictionnaire

Le dictionnaire crée est stocké sous forme de fichier .txt. Il se présente sous la forme : `motStemmer[DocID1, DocID2, DocID3].`
Pour un corpus de 364 documents d'un poid total d'environ 110Mo nous obtenons un dictionnaire avec 128383 entrées d'un poid d'environ 15Mo.

b) Les méthodes utilisées

Nous avons utilisé deux méthodes pour la recherche documentaire via notre programme:

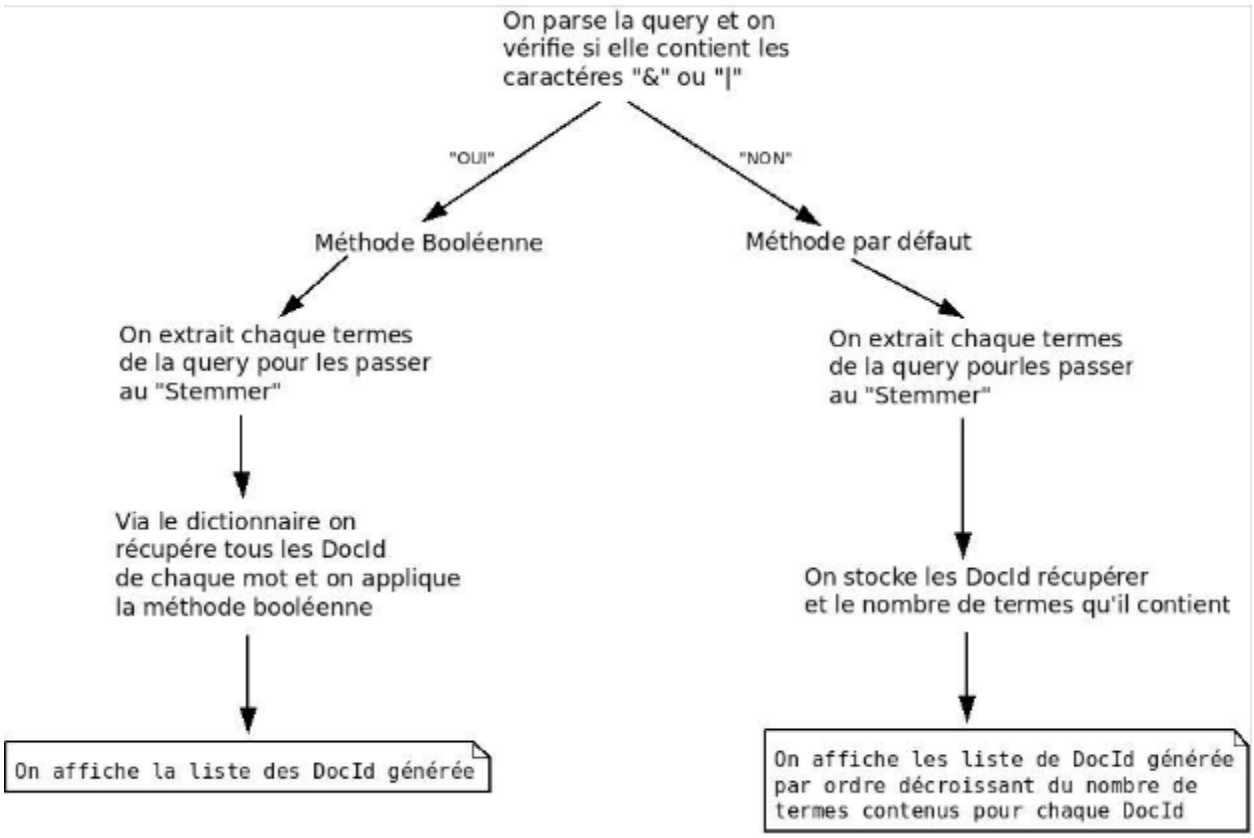
1. Méthode Booléenne : Cette méthode reprend les fonctionnalités vu en cours pour faire la recherche. Nous avons implémenté un algorithme pour la recherche avec "ou" et un algorithme avec le "et". A la fin de la recherche nous affichons l'ensemble des documents classés par DocId croissants.

2. Méthode par défaut : Cette méthode est utilisé si il n'y a pas de mot "et" ou "ou" dans la query. L'algorithme opère ainsi :

    - Il va chercher pour chaque mots tous les DocID qui lui sont associés dans le dictionnaire => Contruction de hash avec en clef le mot et en valeur les DocID.

- Il créé une hash(H_{docID}) prenant le DocID en clef et en valeur le nombre de fois ou le DocID apparaît en valeur des hash crées à l'étape précédente.
- On trie H_{docID} par leur valeur de manière croissante.

c) Déroulement du traitement d'une Query



Schémas de traitement d'une query

Le schéma explique les deux scénarios possibles lorsque l'utilisateur lance une recherche. Le résultat de la recherche booléenne avec un "|" (un OR) est le même que le résultat de la méthode par défaut, seul l'ordre d'affichage est différent. La méthode par défaut affiche les documents par ordre décroissant du nombre de termes de la query contenus de le fichier tandis que la recherche booléenne affiche les documents par ordre alphabétique.

IV) Améliorations et tests

a) Optimisation du code

L'optimisation s'est portée sur l'articulation entre la lecture des fichiers du corpus, la phase de parsing et la phase de stemming. Au début le temps de génération du dictionnaire prenait environ 30 minutes. C'était du au fait que pour chaque fichiers on les passaient entièrement au parser puis au stemmer, ensuite nous passions le résultat pour qu'il soit ajouté au dictionnaire. De plus, pour chaque fichiers nous avions un nouveau parser et stemmer. Pour améliorer le temps d'exécution nous avons donc fait un seul parser et stemmer. Ainsi nous lisons lignes par lignes le fichier puis nous passons chaque mots au stemmer qui va l'ajouter au dictionnaire. Ce processus nous permet maintenant de générer le dictionnaire en environ 20 secondes. Nous allons donc 90 fois plus vite, ce qui est un gain de temps considérable.

b) Tests sur les algorithmes

Pour voir la véracité de nos différents algorithmes de recherche nous avons effectué quelques tests. Pour les illustrer nous allons utiliser les mots: "Shannon Wright". Voici les résultats obtenu avec les différents algorithmes.

Query	Méthode Booléen(M_B)	Méthode par défaut (M_D)
Shannon	∅	44
Wright	∅	158
Shannon Wright	∅	174
Shannon|Wright	174	∅
Shannon&Wright	28	∅

Tableau de comparaison de résulats sur les mots Shannon et Wright

Ici on constate :

- Les résultats de "*Shannon Wright"* et *"Shannon|Wright"* sont identiques. Cela est du au fait que la recherche par défaut va chercher tous les documents ou sont présents *"Shannon"* ou *"Wright"* ce qui se ramène au "ou" mais la méthodes par défaut affiche les résultats différemment.

- Concentrons nous sur le résultat de *“Shannon|Wright”* qui est de 174. Pour vérifier si le résultat est bon il faut faire le calcul suivant :
  $(M_D(Shannon) + M_D(Wright)) - M_B(Shannon\&Wright) = (158 + 44) - 28 = 174$.
  Les deux nombres de résultat sont identiques.

c) Amélioration à apporter

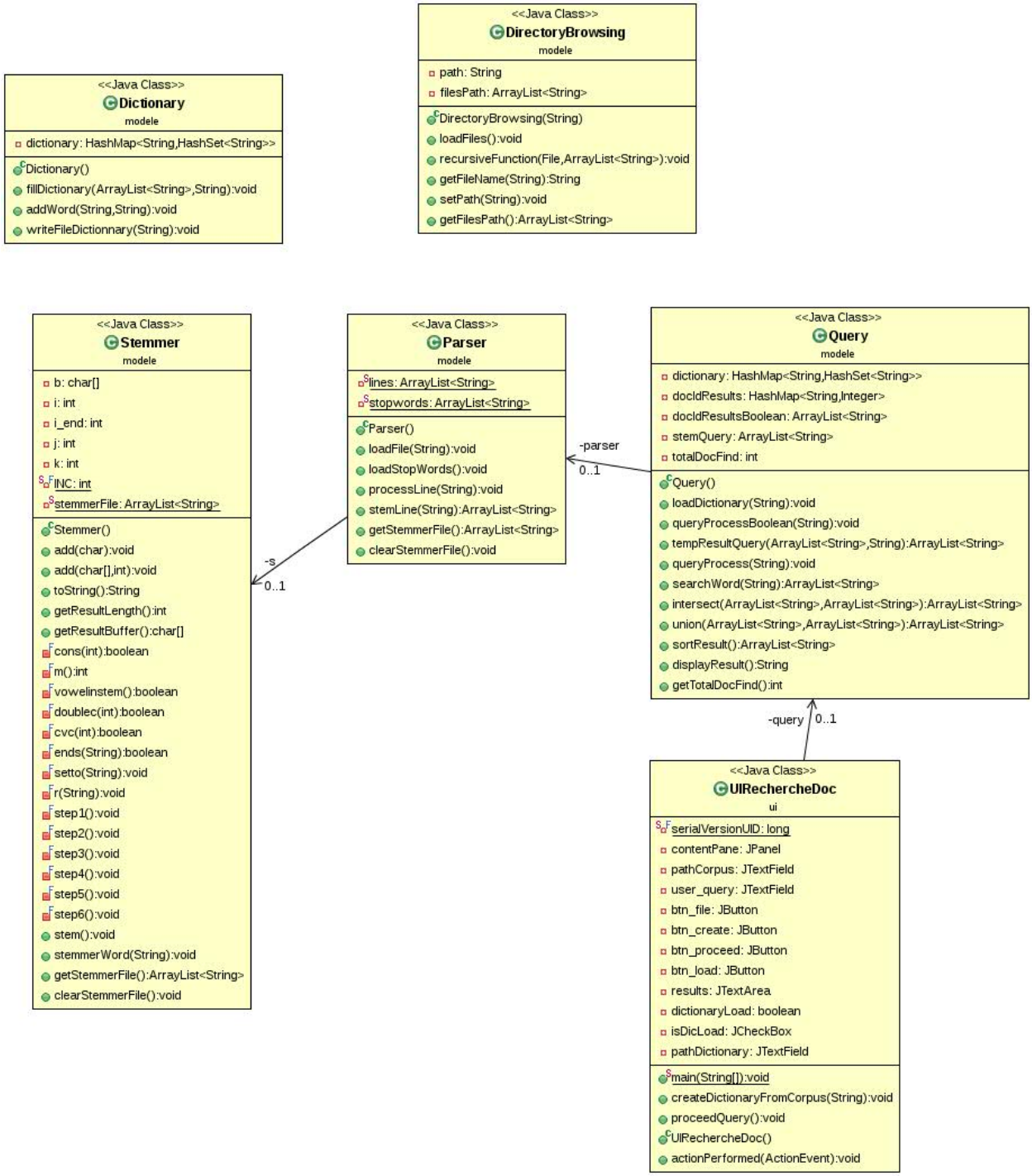
Ici nous avons codé une application avec une méthode booléenne. On peut améliorer cette méthode avec l’acceptation de l’opérateur “non”. L’acceptation de bi-words. On peut aussi implémenter une méthode vectorielle pour améliorer l’efficacité de la recherche.

Annexes :

Vous pouvez récupérer le projet ici :

- http://github.com/geeknbar/projet_recherche_documentaire.
- Il comprend le code ainsi que la JavaDoc.

Diagramme UML disponible à la page suivante


<<Java Class>>
Dictionary
modele
dictionary: HashMap<String,HashSet<String>>
Dictionary()
fillDictionary(ArrayList<String>,String):void
addWord(String,String):void
writeFileDictionnary(String):void
<<Java Class>>
DirectoryBrowsing
modele
path: String
filesPath: ArrayList<String>
DirectoryBrowsing(String)
loadFiles():void
recursiveFunction(File,ArrayList<String>):void
getFileName(String):String
setPath(String):void
getFilesPath():ArrayList<String>
<<Java Class>>
Stemmer
modele
b: char[]
i: int
i_end: int
j: int
k: int
INC: int
stemmerFile: ArrayList<String>
Stemmer()
add(char):void
add(char[],int):void
toString():String
getResultLength():int
getResultBuffer():char[]
cons(int):boolean
m():int
vowelinstem():boolean
doublec(int):boolean
cvc(int):boolean
ends(String):boolean
setto(String):void
r(String):void
step1():void
step2():void
step3():void
step4():void
step5():void
step6():void
stem():void
stemmerWord(String):void
getStemmerFile():ArrayList<String>
clearStemmerFile():void
-s
0..1
<<Java Class>>
Parser
modele
lines: ArrayList<String>
stopwords: ArrayList<String>
Parser()
loadFile(String):void
loadStopWords():void
processLine(String):void
stemLine(String):ArrayList<String>
getStemmerFile():ArrayList<String>
clearStemmerFile():void
-parser
0..1
<<Java Class>>
Query
modele
dictionary: HashMap<String,HashSet<String>>
docIdResults: HashMap<String,Integer>
docIdResultsBoolean: ArrayList<String>
stemQuery: ArrayList<String>
totalDocFind: int
Query()
loadDictionary(String):void
queryProcessBoolean(String):void
tempResultQuery(ArrayList<String>,String):ArrayList<String>
queryProcess(String):void
searchWord(String):ArrayList<String>
intersect(ArrayList<String>,ArrayList<String>):ArrayList<String>
union(ArrayList<String>,ArrayList<String>):ArrayList<String>
sortResult():ArrayList<String>
displayResult():String
getTotalDocFind():int
-query
0..1
<<Java Class>>
UIRechercheDoc
ui
serialVersionUID: long
contentPane: JPanel
pathCorpus: JTextField
user_query: JTextField
btn_file: JButton
btn_create: JButton
btn_proceed: JButton
btn_load: JButton
results: JTextArea
dictionaryLoad: boolean
isDicLoad: JCheckBox
pathDictionary: JTextField
main(String[]):void
createDictionaryFromCorpus(String):void
proceedQuery():void
UIRechercheDoc()
actionPerformed(ActionEvent):void