UNIVERSITÉ D'AVIGNON ET DES PAYS DE VAUCLUSE

M2 ILSEN – 2019/20
UE Ingénierie du document et de l'information
UCE Indexation & recherche

Vincent Labatut

TP 2 | Construction de l'index

L'objectif de ce TP est de compléter les classes créées lors du TP 1, afin de pouvoir créer des index à partir des corpus fournis. Là encore, il est conseillé de déboguer votre code source sur un petit nombre de textes extraits de cette collection, pour des raisons de rapidité (dossier `wp_test` du projet `Common`). Lorsque votre programme semble fonctionner correctement, alors vous pouvez l'appliquer au corpus entier (`wp` ou `springer`).

1 Construction

On veut implémenter l'algorithme présenté en cours pour construire un fichier inverse simple. Pour rappel, il s'agit de :

1. (Obtenir une liste de paires (terme, docID) représentant le corpus ;)
2. Trier cette liste par terme puis docID ;
3. Fusionner les occurrences multiples relatives au *même document* ;
4. Grouper les occurrences multiples relatives à des *documents différents* afin de produire les listes de postings.

Exercice 1

Dans le package `processing`, la classe `Builder` est chargée d'implémenter l'algorithme décrit ci-dessus. Dans cette classe, complétez la méthode `int filterTokens (List<Token> tokens)` qui reçoit la liste triée des tokens normalisés et en supprime les occurrences multiples relatives au même document (point 3 de l'algorithme).

> **Attention :** comme au TP précédent, pensez à utiliser la classe `Iterator` pour parcourir la liste tout en ayant la possibilité d'en supprimer des éléments.

La méthode renvoie un entier correspondant au nombre de termes trouvés dans la liste à la fin du traitement. Cela signifie donc que vous devez compter le nombre de tokens consécutifs dont les types diffèrent (et non pas tous les tokens). Cette valeur sera utilisée plus tard pour déterminer la taille de l'index lors de son instanciation.

Complétez la méthode `main` de manière à tester votre fonction sur des données bidon initialisées manuellement.

> **Remarque :** votre méthode doit effectuer un traitement de complexité linéaire. Vous ne devez donc pas utiliser des structures supplémentaires comme des maps. Vous devez tirer parti du fait que la liste reçue est triée, ce qui signifie que les doublons contenus dans la liste sont forcément placés consécutivement. Le filtrage et le calcul du nombre de termes doivent être effectués dans une même (et unique) boucle.

Exercice 2

Dans la classe `Builder`, complétez la méthode `int buildPostings(List<Token> tokens, AbstractIndex index)` qui reçoit la liste triée filtrée des tokens normalisés, ainsi qu'un index vide index (préalablement instancié dans une autre méthode).

La méthode doit remplir le champ `data` de l'index avec le contenu de la liste reçue (point 4 de l'algorithme). Chaque terme doit apparaître dans un objet `IndexEntry` avec la liste des postings associés.

La méthode renvoie le nombre total de postings placés dans les listes associées aux différentes entrées constituant l'index. Cette valeur est renvoyée seulement à titre de contrôle : si le traitement est correct, elle devrait être égale à la longueur de la liste `tokens` reçue.

Remarque : pour `ArrayIndex`, rappelons que les objets `IndexEntry` doivent être rangés dans le tableau dans l'ordre alphanumérique, afin de permettre d'effectuer des recherches dichotomiques.

Exercice 3

Dans la classe `Builder`, complétez la méthode `AbstractIndex buildIndex (List<Token> tokens, LexiconType lexiconType)`, qui prend en paramètre la liste des tokens *normalisés* et renvoie un objet représentant l'index construit. Pour arriver à ce résultat, la méthode doit appliquer l'algorithme présenté au début de cette section.

Le point 2 de cet algorithme stipule que la liste de tokens doit d'abord être triée. Vous devez pour cela utiliser la méthode `Collections.sort(List<T> list)`. Notez que `sort` va utiliser votre méthode `Token.compareTo` pour réaliser le tri. En cas de problème de tri, l'erreur viendra donc probablement de `compareTo`, et non pas de `sort`.

Pour les points 3 et 4, vous devez bien entendu appeler les méthodes `filterTokens` et `buildPostings`.

Le paramètre `lexiconType` reçu par la méthode permet de contrôler la structure de données utilisée pour représenter le lexique. Ce paramètre est du type énuméré `LexiconType`, qui est prédéfini dans la classe `ArrayIndex`. Il va directement déterminer le type d'index renvoyé par la méthode : `ArrayIndex` (pour la valeur `LexiconType.ARRAY`), `HashIndex` (`HASH`) ou `TreeIndex` (`TREE`).

La méthode doit produire une sortie texte de la forme suivante (les valeurs indiquées correspondent à `wp_test`) :

```
Sorting tokens...
631928 tokens sorted

Filtering tokens...
167502 tokens remaining, corresponding to 38218 terms

Building posting lists...
167502 postings listed, lexicon type=ARRAY
```

2 Indexation

Exercice 4

Dans la classe `AbstractIndex`, complétez la méthode `AbstractIndex indexCorpus(TokenListType tokenListType, LexiconType lexiconType)`, chargée de réaliser l'indexation, i.e. d'enchaîner la segmentation (tokénisation), la normalisation et la construction, sur le corpus contenu dans le dossier est obtenu via `FileTools.getCorpusFolder()`. Pour cela, la méthode doit instancier et appliquer successivement trois objets de classes `Tokenizer`, `Normalizer` et `Builder`. Les deux derniers doivent être stockés dans les champs appropriés d'`Index`, pour être utilisés ultérieurement. Une fois l'index créé, il doit être affiché.

Le paramètre `lexiconType` permet de contrôler la représentation du lexique : il faut simplement le transmettre à `buildIndex`. Le paramètre `tokenListType` est du type énuméré `TokenListType`, qui est prédéfini dans `AbstractIndex`. Il est destiné à contrôler le type de liste utilisé pour construire l'indexation, afin de comparer les performances. Sa valeur peut être `ARRAY` (on utilise une `ArrayList`) ou `LINKED` (`LinkedList`).

La méthode doit produire une sortie texte de la forme (où la partie notée `[...]` correspond à la sortie texte de `Builder.buildIndex`, déjà décrite auparavant) :

```
Tokenizing corpus...
631928 tokens were found

Normalizing tokens...
```

```
631928 tokens remaining after normalization

Building index...
  [...]
There are 38218 entries in the index, token list=LINKED
```

Comme précédemment, les valeurs indiquées ci-dessus correspondent à `wp_test`. Pour `wp`, on obtient 7 612 885 tokens avant et après normalisation, 2 177 994 tokens après filtrage, correspondant à 142 792 termes, 2 177 994 postings, et finalement 142 792 entrées dans l'index.

Exercice 5

Dans la racine du projet, allez dans la classe `Test1`. Complétez la méthode `testIndexation` chargée de construire et afficher un index en utilisant la méthode `AbstractIndex.indexCorpus`.

Complétez enfin la méthode principale `main` pour 1) indiquer le nom du corpus dans `Configuration.setCorpusName`, et 2) réaliser l'appel de `testIndexation`, afin d'afficher votre index.

Exemple : pour `corpus`, le début de l'index affiché devrait prendre la forme suivante :

```
0. <0 [755] ( 0 5 12 15 18 21 25 37 42 43 45 46 48 49 51 52 55 56 64 68 ... )>
1. <00 [52] ( 51 169 219 233 235 251 401 442 478 490 538 597 672 694 751 ... )>
2. <000 [798] ( 5 7 9 10 11 22 30 31 32 34 48 51 52 53 59 63 68 71 72 74 ... )>
3. <0000 [6] ( 148 149 1068 2330 2571 2711 )>
4. <00000 [1] ( 1068 )>
5. <000000 [1] ( 2232 )>
6. <0000000 [3] ( 367 916 2700 )>
7. <00000000 [4] ( 346 1068 2424 2771 )>
8. <00000001 [1] ( 160 )>
9. <00000011b [1] ( 2656 )>
10. <00000080 [1] ( 1022 )>
```

Exercice 6

Dans la classe `FileTools`, complétez la méthode `List<String> getFileNamesFromPostings(List<Posting> postings)` qui reçoit une liste de postings et renvoie la liste des noms de fichier correspondant aux docIds, pour le corpus localisé au chemin indiqué par `FileTools.getCorpusFolder()`. La méthode doit ouvrir le dossier du corpus et y lire les noms des fichiers qu'il contient.

Exemple : pour une liste 1,2,3, la méthode renvoie :

- `002637bd-64b5-424d-b3b1-6dd0331847b2.txt`
- `004233de-a0e7-4c25-b634-e0681e9174b0.txt`
- `005b5d65-b27a-4362-b7b7-d4c6fdf2a147.txt`

> **Remarque :** l'ordre dans lequel Java liste les fichiers contenus dans un dossier peut varier d'un système à l'autre, d'où la nécessité de les trier comme expliqué lors du TP 1.

Exercice 7

Utilisez la méthode `getFileNamesFromPostings` pour effectuer une vérification basique de votre index. Pour cela :

1. Prenez au hasard quelques entrées de l'index affiché précédemment ;
2. Récupérez les noms des fichiers correspondant à leurs postings ;
3. Ouvrez les documents concernés et vérifiez s'ils contiennent bien les termes associées entrées sélectionnées.

3 Durée de traitement

Exercice 8

Pour évaluer la performance de votre programme, il serait bon de mesurer le temps mis pour effectuer les différentes étapes. Pour cela, le plus simple est d'utiliser la fonction

`System.currentTimeMillis()`, qui renvoie la date actuelle exprimée en millisecondes.

On peut l'utiliser de la façon suivante :

```
long start = System.currentTimeMillis();
{  // début du traitement dont on veut mesurer la durée
   ...
   // fin du traitement concerné
}
long end = System.currentTimeMillis();
System.out.println("Duréee mesurée : "+(end-start)+" ms");
```

Modifiez les méthodes `AbstractIndex.indexCorpus` et `Builder.buildIndex` afin d'afficher les temps de traitement nécessaires aux différentes étapes du processus d'indexation. Vous devez obtenir un affichage similaire à l'exemple ci-dessous, qui correspond à `wp` :

```
Tokenizing corpus...
7612885 tokens were found, duration=13843 ms

Normalizing tokens...
7612885 tokens remaining after normalization, duration=13352 ms

Building index...
Sorting tokens...
7612885 tokens sorted, duration=6915 ms

Filtering tokens...
2177994 tokens remaining, corresponding to 142792 terms, duration=1719 ms

Building posting lists...
2177994 postings listed, lexicon=ARRAY, duration=893 ms
There are 142792 entries in the index, token list=LINKED, duration=9528 ms

Total duration=36723 ms
```

> **Remarque :** les temps peuvent varier en fonction de la machine, bien entendu, mais les différences relatives entre les étapes devraient être du même ordre (sinon cela signifie qu'une étape n'a pas été implémentée de façon optimale, voire est erronée).

Exercice 9

Comparez les résultats obtenus en faisant varier :

- Le type de liste utilisé pour stocker les tokens lors de l'indexation, grâce au paramètre de type `TokenListType`.
- La structure de données utilisée pour représenter le lexique, grâce au paramètre de type `LexiconType`.

4 Stockage

Dans les TP à venir, nous aurons besoin de l'index à chaque exécution. Il serait long et peu pratique de le reconstruire à chaque fois. Nous allons plutôt le stocker dans un fichier en utilisant le système de sérialisation de Java.

Notez que pour pouvoir enregistrer un objet en utilisant le mécanisme de sérialisation de Java, toutes les classes concernées doivent implémenter l'interface `Serializable`, qui a la propriété de ne contenir aucune méthode. Dans notre cas, cela concerne bien sûr `AbstractIndex`, mais aussi `Tokenizer` et `Normalizer`, puisque un index possède des champs de ces types-là. Cela concerne aussi les classes `IndexEntry` et `Posting` à cause du champ `data` de l'index.

> **Remarque :** utiliser la sérialisation de Java n'est certainement pas la solution la plus performante, que ce soit en termes de robustesse, de temps de traitement et d'espace occupé par le fichier généré. Cependant, cette approche offre l'avantage d'être rapide à mettre en

place. Mais elle ne serait pas utilisée dans un contexte plus réaliste, notamment en production.

Exercice 10

Dans la classe `AbstractIndex`, complétez la méthode `void write()` de manière à ce qu'elle enregistre l'index dans le fichier dont le chemin est renvoyé par `FileTools. getIndexFile()`.

Votre méthode doit générer une sortie texte de la forme suivante (la durée affichée est seulement un exemple, bien entendu) :

```
Writing the index
Index written, duration=17253 ms
```

Dans `Test1`, modifiez la méthode `testIndexation` de manière à ce qu'elle enregistre l'index produit. Puis, exécutez-la et vérifiez l'existence du fichier contenant l'index, qui devrait être localisé dans le dossier `data` du projet `Index1`.

Remarque : pour enregistrer un objet `object` dans un fichier appelé `fileName`, en Java, on ouvre d'abord un flux approprié :

```
File file = new File(fileName);
FileOutputStream fos = new FileOutputStream(file);
ObjectOutputStream oos = new ObjectOutputStream(fos);
```

Puis on écrit l'objet :

```
oos.writeObject(object);
```

Et on n'oublie pas de refermer le flux :

```
oos.close();
```

Exercice 11

Dans la classe `AbstractIndex`, complétez la fonction `AbstractIndex read()` qui lit l'index dans le fichier dont le chemin est renvoyé par `FileTools.getIndexFile()`.

Votre méthode doit produire une sortie texte de la forme suivante :

```
Loading the index
Index loaded, duration=21494 ms
```

Dans `Test1`, modifiez la méthode `main` de manière à ce qu'elle lise l'index enregistré dans le fichier de l'exercice précédent. Affichez le contenu de l'index chargé, et comparez-le à l'affichage obtenu juste après la création (avant l'enregistrement). Les deux affichages devraient être exactement les mêmes, puisqu'il s'agit du même index.

Remarque : pour lire un objet dans un fichier nommé `fileName`, en Java, on ouvre d'abord un flux approprié :

```
File file = new File(fileName);
FileInputStream fis = new FileInputStream(file);
ObjectInputStream ois = new ObjectInputStream(fis);
```

Puis on lit l'objet (on doit connaître sa classe à l'avance) :

```
MyClass result = (MyClass) ois.readObject();
```

Et on n'oublie pas de refermer le flux :

```
ois.close();
```