Rapport MAAIN

Victor Fontaine

21 Mars 2019

Ce rapport décrit comment nous avons implémenté un moteur de recherche sur les pages de Wikipedia FR (disponible sous format xml dans le fichier *frwiki-20190120-pages-articles.xml*) en utilisant l'algorithme **PageRank** de Google. Nous avons utilisé le langage Java pour ce projet.

Pour réaliser ce moteur de recherche, il nous faut tout d'abord collecter tout les titres des pages séléctionnées et leurs associer un identifiant unique pour ensuite créer la matrice d'adjacence du graphe des pages (sous la forme CLI, cf section 1) ; cela est fait par le **Crawler**.

Ensuite il faut parcourir les pages séléctionnées pour en extraire tout les mots ainsi que leurs fréquences d'apparition dans la page, et stocker ses données ; nous verrons cela dans la section 2 **L'indexation**.

Enfin, il faut calculer le **PageRank** (section 3) de chaque page en fonction de la matrice d'adjacence du graphe des pages pour finalement pouvoir rechercher une suite de mots et trier les URL correspondant au résultat de la recherche par ordre de *PageRank* décroissants.

Nous verrons alors (section 4) comment produisons nous une réponse, sous la forme d'une liste d'URL wikipedia, pour une recherche donnée qui contient un ou plusieurs termes à rechercher.

Finalement, nous verrons les difficultés rencontrées lors du projet, ainsi que les améliorations possibles.

1 Le Crawler

Premièrement, nous avons implémenter le **crawler** dont la mission est de parcourir toutes les pages wikipedia à disposition, et, à partir de mots clés pré-défini [1], sélectionné les pages qui contiennent ces mots et stocker tout les titres des pages concernées dans une $Hashtable < String, Integer >$ nommée ht_titles (qui est une variable globale). Chaque entré de cette *Hashtable* est constituée d'un *String* qui est le titre de la page, et d'un *Integer* qui est l'identifiant correspondant à cette page. Cette *Hashtable* est créer par la fonction :

```
Hashtable<String, Integer> createHashtableTitles(String file,
String[] wanted)
```

1. Dans notre cas, nous avons utilisé les mots *mathématiques* , *informatique* , *sciences* et *étudiant* pour la pré-sélection des pages.

qui enregistre ces données dans un fichier *HtTitle.data* grâce à la fonction `void storeHtTitle()` quand elle a fini son exécution. Nous avons donc *parser* le fichier xml, et à chaque nouvelle page (i.e qui ce qui se trouve dans la balise `<page>`) nous extrayions le titre grâce à un $PatternMatcher$ (car celui-ci se trouve dans une balise `<title>`) de la manière suivante :

```
Pattern p = Pattern.compile("<title>(.+?)</title>");
Matcher m = p.matcher(line);
if(m.find()){
    title = m.group(1);
}
```

Ensuite nous analysons les mots contenus dans la page pour voir si celle-ci contient un des mots pré-défini (cf note page 1), et ci c'est le cas nous ajoutons le titre de cette page avec un nouvel identifiant (incrémentale partant de 0) dans *ht_titles* : `ht_titles.put(title,current_id_title);`

Lors de l'indexation (cf section 2), nous allons retenir l'identifiant de la page actuellement parcouru et nous allons *parser* les liens sortants en utilisant une expression régulière et un $PatternMatcher$ comme ceci :

```
Pattern p = Pattern.compile("\\[\\[(.+?)\\]\\]");
Matcher m = p.matcher(line);
while(m.find()){
    // Verification que le lien pointe vers une page de ht_titles
    // Remplissage de la matrice d'adjacence sous la forme C.L.I
}
```

Quand nous trouvons des liens entre deux pages de *ht_titles* (i.e pas un lien "mort"), nous stockons ce lien dans une `ArrayList<Integer>` nommée *liens* jusqu'à avoir parcouru toute la page courante, puis nous ajoutons ces liens aux tableaux C, L et I qui représente la matrice d'adjacence du graph des pages. Cette matrice étant creuse, le stockage sous forme CLI est plus approprié. Tout cela se fait en une seule lecture du fichier xml en ayant au préalable les titres des pages séléctionnées, à savoir *ht_titles* (qui est obtenu avec une première lecture du fichier ou en lisant des données enregistrés au préalable).

2 L'indexation

Une fois tout les titres de pages enregistrés, nous allons reparcourir le fichier xml pour en extraire chaque mots (qui se trouve dans une balise `<text>`, et en dehors des balises `&lt ... &gt;`) des pages seléctionnées (i.e dont le titre figure dans la $Hashtable$*ht_titles*) ainsi que calculer leurs fréquences dans chacune des pages. Pour chaque ligne, nous allons supprimer les *stopwords* (via la fonction `String cleanStopWords(String s)`)[2] et nous allons la normaliser grâce à la fonction `String normalize(String s)`. Cette fonction permet

2. nous avons utiliser une liste de stop words disponible ici : https ://github.com/stopwords-iso/stopwords-fr/blob/master/stopwords-fr.txt que nous avons adpaté à notre besoin

d'enlever les accents, la ponctuation ainsi que tout ce qui se trouve entre des balises &lt..> (et autres). Pour stocker ses données, nous allons créer une `Hashtable <String,Hashtable<Integer,Double>>` nommé *dict* dont chaque entré est de la forme un *String* corréspondant au mot en question, et une $Hashtable < Integer, Double >$ associant l'identifiant ($Integer$) des pages dans lesquelles le mots apparait à la fréquence d'apparition du mot dans cette page ($Double$). Cela se fait via la fonction :

```
Hashtable<String,Hashtable<Integer,Double>> createDictionnary(
    String file, Hashtable<String, Integer> ht_titles)
```

Une fois cette $Hashtable$ créer, nous allons en extraire les N mots les plus fréquents, dont la fréquence d'apparition est supérieur à une valeur minimale pré-définie. Dans notre cas, après avoir réalisé des tests pour déterminer une bonne valeur, nous avons choisis que $FreqMin = 0.000001$ et que $N = 20000$, i.e que nous auront au plus 20.000 mots dans le dictionnaire et de plus tous ces mots auront au moins une page dans laquelle sa fréquence d'apparition est au moins de 0.000001. Cette sélection est faite par la fonction :

```
Hashtable<String,Hashtable<Integer,Double>> selectNBestFreq(
    Hashtable<String,Hashtable<Integer,Double>> dict,
    Hashtable<Integer,Integer> ht_nb_words)
```

Cette fonction renvoi une `Hashtable<String,Hashtable<Integer,Double>>` qui n'est pas encore le dictionnaire final, car il faut encore le trier par $PageRank$ décroissant (cf section 3 PageRank). Nous avons maintenant toutes les données nécessaire pour calculer le $PageRank$ de chaque page.

3 Page Rank

Nous allons maintenant calculer le $PageRank$ de chaque page sélectionnées (qui sera stocké dans un tableau : `Double[] rank`).

Premièrement nous initialisons le vecteur $rank$ à $1/\#pages$ pour chaque page :

```
int n = ht_titles.size();
Double [] rank = new Double[n];
for(int k = 0; k < n; k++){
    rank[k] = 1.0/n;
}
```

de cette manière, nous avons un vecteur de taille n dont chaque entrée vaut $1/n$. Ensuite nous fixons un *epsilone* (encore une fois nous avons fait des tests pour déterminer une bonne valeur à prendre pour *epsiolne* et nous avons choisis que $epsilone = 0.001$), et nous recalculons le vecteur $rank$ en faisant le produit matricielle entre l'ancien vecteur $rank$ et la matrice d'adjacence des pages. De cette manière, le vecteur $rank$ va converger vers le vecteur $rank$ souhaité. Nous arretons de recalculer le vecteur $rank$ quand la distance entre l'ancien vecteur $rank$

et le nouveau vecteur *rank* est plus petite que *epsilone*. Dans notre cas, cela représente en moyenne une trentaine d'itérations. La distance entre les deux vecteurs est calculée par la fonction `Double distance(Double[] v, Double[] w)` qui calcule la **distance euclidienne** entre deux vecteurs :

$$distance(u, v) = \sqrt{\sum_{i=1}^{n}(u_i - v_i)^2} \tag{1}$$

Si nous avons une page qui ne contient aucun lien sortant qui pointe vers d'autres pages sélectionnées, nous simulons des liens sortant en choisissant k nombres aléatoires et en mettant un poids de $1/k$ pour toutes les pages dont l'identifiant fait parti des nombres aléatoires. Une fois le calcul terminé nous stockons le résultat dans la variable globale *rank*.

Nous pouvons alors revenir à notre création du dictionnaire : Il aura comme type `Hashtable<String, Integer[]>` dont chaque entrée est un *String* corréspondant à un mot, et un tableau d'entiers ($Integer[]$) qui est la liste des identifiants des pages dans lesquelles le mot apparait, trié par ordre de *PageRank* décroissant. Nous n'avons plus besoin de leurs fréquences d'apparition ici. Ce tri se fait par la fonction :

```
Integer[] sortPagesByRank(Hashtable<Integer,Double> ht)
```

qui renvoi un tableau contenant les identifiants des pages trié par ordre de *PageRank* décroissant. Cette fonction est appelée par la fonction :

```
Hashtable<String, Integer[]> sortDictionnary(
    Hashtable<String,Hashtable<Integer,Double>> dictionnary,
    Double[] rank)
```

qui prend en argument le dictionnaire que nous avons précédemment produit, (qui contient les mots ainsi que leurs fréquences d'apparitions dans chaque pages où il apparait) et qui le tri en fonction du *PageRank* des pages. Comme dis précédemment, dans ce dictionnaire nous n'avons pas les fréquences d'apparitions car nous n'en n'avons pas besoin par la suite. Au final on obtient un dictionnaire (qui est une variable globale nommée *dictionnary*) qui contient les mots et leurs liste des pages dans lesquelles ils apparaisent trié par *PageRank* décroissant.

C'est ce dictionnaire que l'on utilisera par la suite pour faire nos recheches, on peut donc le stocker une fois qu'il est calculer pour eviter de le recalculer lors des exécutions futures :

```
FileOutputStream f_dictionnary =
        new FileOutputStream("Dictionnary.data");
ObjectOutputStream o_dictionnary =
        new ObjectOutputStream(f_dictionnary);
o_dictionnary.writeObject(dictionnary);
o_dictionnary.close();
f_dictionnary.close();
```

De manière générale, une fois les données calculer (le *dictionnary*, la table *ht_titles*, le vecteur *rank* ..) on les stockera dans des fichiers (d'extension *.data*) via la fonction `void storeDatas()` pour ne pas avoir à les recalculer par la suite, mais pour pouvoir les lire à partir des fichiers. Cela est possible par la fonction `void readDatas()` qui rempli nos données à partir des fichiers correspondant.

On pourra donc charger des données (option *read* dans le programme) que l'on à calculer auparavant pour éviter de devoir tout refaire à chaque fois. De cette manière, il nous suffit d'executer une fois notre programme (option *init*) pour calculer et stocker nos données, puis nous pouvons les utiliser dans la suite pour faire nos recherches rapidement. En effet, la recherche se fait en quelques millisecondes, tandis que la création de *ht_titles* et *dictionnary* prend plusieurs heures quand on a beaucoup de pages à analyser.

4 Recherche

Nous avons maintenant tout ce qu'il nous faut pour faire des recherches sur nos pages. En effet, nous avons le *dictionnary* qui contient les mots, et leurs listes de pages rangées par *PageRank* décroissant.

La fonction suivante permet de faire une recherche comportant un seul mot, *word*, qui, à partir du dictionnaire *dict*, renvoye la liste des identifiants des pages comportant ce mot, triée par *PageRank* décroissant :

```
ArrayList<Integer> researchSimple(String word,
    Hashtable<String, Integer[]> dict)
```

Ensuite il nous suffit de passer cette liste d'identifiant à la fonction :

```
void printURLS(ArrayList<Integer> list,
    Hashtable<Integer, String> ht_titles)
```

qui va récuperer le titre correspondant de chaque identifiants, et le passer à la fonction :

```
public static String titleToURL(String title){
    title = title.replaceAll(" ","_");
    title = "https://fr.wikipedia.org/wiki/"+title;
    return title;
}
```

pour traduire les titres en URL wikipédia. La fonction `printURLS` va alors afficher tout ces URL, en mettant en haut de la liste les pages dont le titre comporte un mot recherché, si de telles pages existent (car on les considère plus importante que les autres). De plus, `printURLS` affiche 10 URL à la fois, et demande à l'utilisateur si il veut afficher les 10 prochaines, ou arreter l'affichage et lancer une nouvelle recherche (cela permet une meilleure visibilité). Le temps pris par la recherche est calculé en millisecondes en faisant la différence de l'heure obtenue par `System.currentTimeMillis()` avant et après la recherche, et est affiché.

Il est également possible de faire une recherche comportant plusieurs mots séparés par des espaces grâce à la fonction :

```
ArrayList<Integer> researchMultiple(ArrayList<String> words,
    Hashtable<String, Integer[]> dict, Double[] rank)
```

qui, comme `searchSimple` renvoi la liste des identifiants des pages résultant de la recherche. Lors d'une recherche, on découpe le *String* lu en entrée pour obtenir une `ArrayList<String>` qui contient tout les mots de la recherche, après avoir enlever les *stopwords* grâce à la fonction `cleanStopWords`. C'est cette liste que nous passons à la fonction $researchMultiple$: nous lançons alors une recherche simple sur le premier mot pour obtenir une première liste d'identifiant de page. Ensuite, tant que l'intersection entre la liste d'identifiant obtenu jusqu'alors et celle que l'on obtient en lançant une recherche simple sur le mot suivant n'est pas vide, on calcule cette intersection avec la fonction :

```
public static ArrayList<Integer> intersection(
ArrayList<Integer> list1, ArrayList<Integer> list2){
    ArrayList<Integer> result = new ArrayList<Integer>();
    for (Integer t : list1) {
        if(list2.contains(t)) {
            result.add(t);
        }
    }
    return result;
}
```

Finalement, on obtient la liste des identifiants des pages qui contiennent tout les mots de la recherche [3], trié par $PageRank$ décroissant. Il ne nous reste plus qu'à les afficher avec la fonction `printURLS` décrite précédemment.

5 Difficultés rencontrées

Durant le développement du projet, nous avons testé nos résultats petit à petit sur une petite partie du gros fichier xml qui ne contient qu'environ 30.000 pages (ce qui correspond au 5.000.000 premières lignes du gros fichier xml). Malheureusement notre algorithme ne passe pas à l'echelle pour le gros fichier qui contient toutes les pages wikipédia françaises : il met trop de temps à indexer tout les titres des pages séléctionnées ainsi que tout les mots qu'elles contiennent. Nous avons essayer d'améliorer cela en parcourant une seule fois le fichier, mais il s'est avéré que ce n'était pas possible car il nous fallait déjà avoir indexé les titres des pages pour savoir si un lien était mort (i.e qu'il ne pointait pas vers une page séléctionnée) ou non.

Nous avons fait quelques petites améliorations de *parsing* (par exemple le fait de ne pas traiter les lignes qui commencent par |) mais malgrès tout, nous

3. éventuellement la liste vide, résultant d'une recherche qui n'a rien donnée

avons un temps d'execution trop long que nous etimons, après nos mesures, à 400 heures (3 pages traités par secondes) si l'on utilise le gros fichier qui contient environ 4.600.000 pages.

De plus, lors du développement du site web, comme nous avions fait notre algorithme en Java, nous avons essayé d'utiliser les *servletJava* qui permettent de faire communiquer un site (sous Apache Tomcat) et une application Java, en l'occurence notre algorithme de recherche, ce qui n'est pas compliqué dans l'idée : il suffit de faire étendre la classe java à `HttpServlet` et d'implémenter la focntion *doPost*() qui traite les données recu par l'application, envoyé par le site avec la méthode *POST*, puis à répondre la liste des URLS et les afficher dans notre page html, avec un petit peu de CSS pour enjoliver le tout. Pourtant cela nous a posé un problème que nous n'avons pas réussi à résoudre (après plusieurs heures dessus) : nous n'avons pas réussi à créer le lien entre notre site et l'application qui devait se faire par les lignes suivantes dans le fichier web.xml :

```
<web-app >
  <servlet>
    <servlet-name>SearchEngineServlet</servlet-name>
    <servlet-class>pkg.SearchEngine</servlet-class>
  </servlet>
   <servlet-mapping>
    <servlet-name>SearchEngineServlet</servlet-name>
    <url-pattern>/search</url-pattern>
  </servlet-mapping>
  <load-on-startup>1</load-on-startup>
</web-app>
```

Nous avons donc abandonné l'idée du site web, et avons opté pour une présentation interactive plus oldschool : le terminal.

6 Améliorations possibles

Beaucoup d'améliorations sont possible, nous en sommes loin de Google qui traite, archive, classifie ect.. des milliards de pages dans toutes les langues et tout les alphabets. De plus, ils n'utilisent plus l'algorithme *PageRank* depuis plusieurs années déjà, cela veut dire qu'il y a surement une manière plus optimale pour le faire.

Malgrès tout, nous aurions pu proposer une correction aux fautes de frappes, en fonction des mots les plus proches du mot éronné ; une autocomplétion en fonction des recherches les plus pertinentes ; un historique des recherches ...

Après coup nous avons réaliser qu'il aurait été beaucoup plus simple de faire l'algorithme en JavaScript pour ensuite générer le site web (ainsi que ces possibles améliorations) en quelques lignes de code.