Rapport de DAAR4 : Choix A
Moteur de recherche d'une bibliothèque

Réalisé par :

BENALLA ANIS
GEORGE MATHIEU
GOMEZ PIERRE

7 février 2021

Table des matières

1 Définition du problème

L'objectif du projet est de créer un moteur de recherche pour une bibliothèque sous la forme d'une application web. Il doit être capable de rechercher efficacement des ouvrages parmi un grand nombre de documents disponibles de grande taille, ici 1664 livres d'au moins 10 000 caractères. Nous avons également veillé à implémenter un critère d'ordonnancement des résultats, ici `Closeness Centrality`.

2 Notre implémentation

L'application est accessible depuis n'importe quel navigateur web, que ce soit sur mobile ou non. Notre projet est séparé en deux entités principales, le client et le serveur. Notre client a été réalisé avec Angular, on a notamment utilisé Bootstrap pour la mise en page et l'interfaçage. Le client est lancé sur le port 4200. Pour le serveur, on se sert d'une API REST développée avec Spring Boot en Java 8. L'API se lance sur le port 8081.

Notre application suit une architecture MVC. Les vues sont gérées par le client Angular, le contrôleur est dans le serveur Spring, les modèles sont des objets sérialisés.

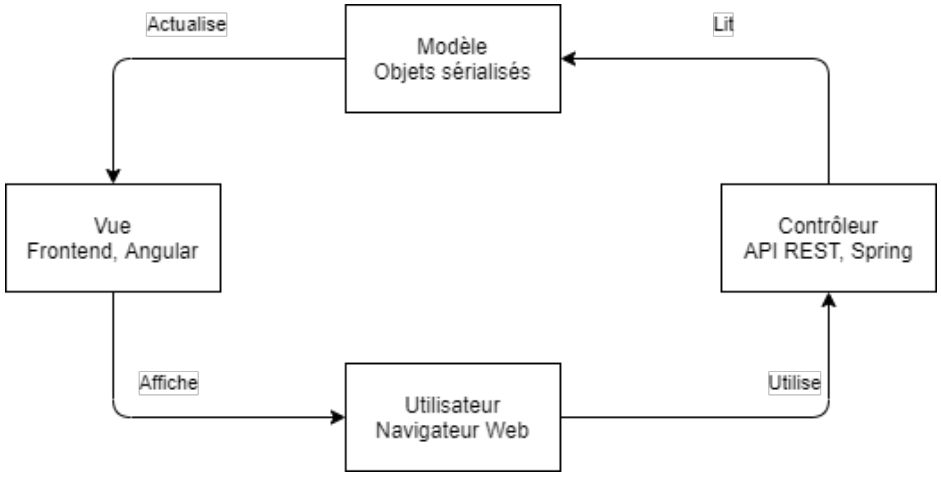

FIGURE 1 – Architecture de l'application

D'une part, nous avons implémenté les deux fonctionnalités principales définies par le sujet : une recherche basique via mot-clef basée sur le contenu des textes, ainsi qu'une recherche avancée pouvant utiliser des Expressions Régulières (*Regex*) pour la recherche de mot-clef. Pour la recherche avancée, nous avons adapté notre projet de DAAR1 : clone de la commande *egrep* qui permet la recherche par *Regex*.

Nous avons également intégré deux fonctionnalités supplémentaires plus spécifiques. Une recherche par auteur ainsi qu'une recherche par titre. Les deux recherches principales sont plus focalisées sur le contenu des textes. On considère cependant que les mots présents dans le titre ainsi que les noms des auteurs sont dans la plupart des cas plus importants que les mots-clefs contenus dans les textes. On a donc souhaité avoir des recherches plus concrètes concernant ces deux critères.

3 Rendu visuel

Pour l'interface utilisateur de l'application, nous avons décidé d'utiliser Angular (version 11.0.7) car il s'agit de l'un des frameworks web les plus utilisés avec JQuery et ReactJS.
Il permet de créer facilement une Application Single Page (SPA) ce qui fluidifie l'expérience utilisateur car il n'y a pas de redirections. Il est basé sur le langage TypeScript qui est un sur-ensemble statiquement typé de JavaScript.

Nous souhaitions réaliser une interface de moteur de recherche simple sur laquelle il était possible de choisir le type de recherche utilisé à l'aide d'onglets en haut de la page. Cliquer sur un type de recherche devait afficher la barre de recherche associée et les résultats trouvés devaient être lister en dessous.

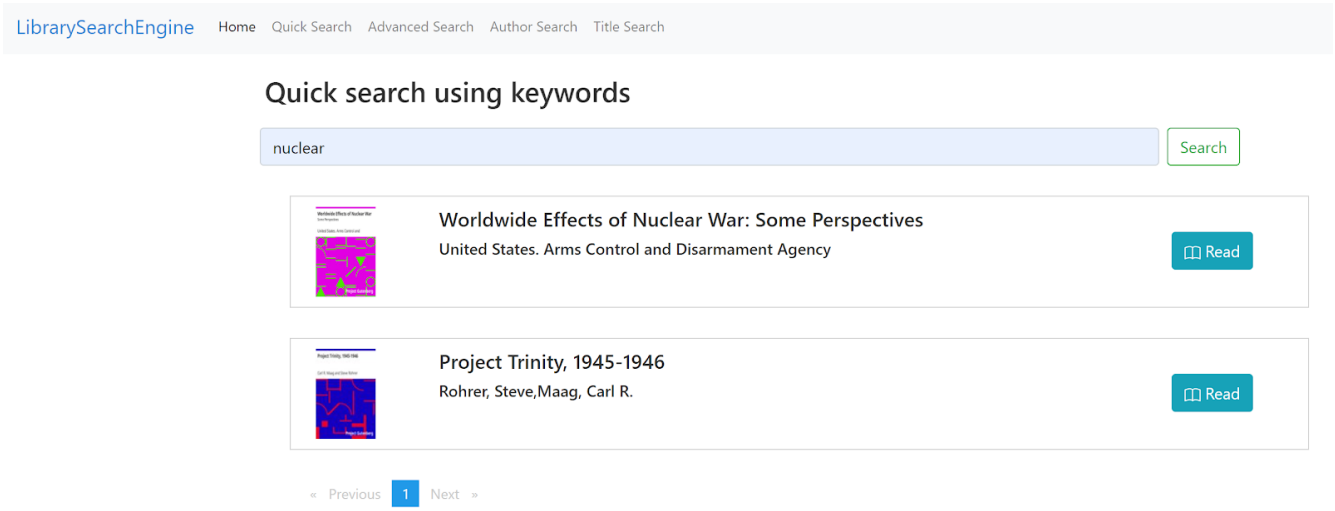

FIGURE 2 – Rendu du client web

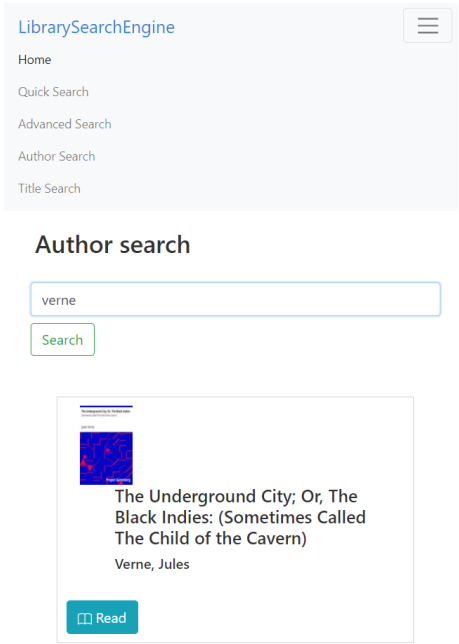

FIGURE 3 – Rendu du client format mobile

Nous avons tiré parti de Bootstrap pour les styles utilisés dans le template html, notamment pour la NavBar sur laquelle apparaissent les onglets de recherche.
Pour lister les résultats, nous nous sommes servis de la directive structurelle d'Angular **ngFor* qui permet de créer une balise html pour chaque élément d'une collection, ici la liste des résultats.
Enfin, afin de gérer un nombre potentiellement important de résultats lors d'une recherche, nous avons utilisé le module ngx-pagination qui permet d'afficher seulement un certain nombre d'éléments par page. Ce module fournit également un outil de navigation entre les différentes pages de résultats, que nous avons placé en bas de la page.

4 Construction de la couche Data

4.1 Notre bibliothèque

Les documents que nous utilisons sont les livres hébergés sur le site **Gutenberg** avec les identifiants allant de 1 à 1664.

Pour construire notre bibliothèque, nous avons utilisé **Gutendex**. C'est une API web qui permet de récupérer les metadata des documents présents sur Gutenberg. Cela nous évite d'avoir à envoyer des requêtes directement à Gutenberg et parser les réponses qui sont des pages HTML. On utilise `com.google.gson` pour parser les réponses JSON de Gutendex et récupérer les informations dont nous avons besoin, ici le lien pour récupérer le contenu du livre. En Java on utilise la classe `RestTemplate` de Spring qui permet de faire des requêtes HTTP.

On note qu'il existe plusieurs formats de textes qui varient selon l'encodage du document (`UTF-8`, `ASCII`). Pour récupérer les textes on doit ainsi chercher l'objet `format` dans la réponse JSON puis les différents éléments `text/plain` possibles. Cela nous donne le lien vers le contenu pur du document hébergé sur Gutenberg, on peut ensuite le récupérer avec une requête HTTP. Ci-dessous un extrait de code montrant comment on parse les réponses JSON.

```
// utils.graph.GsonManager.java
/** Get the plain text url of a Book */
public static String getBookContentURL(JsonObject jo) throws IOException {
      JsonElement plainText = new JsonObject();
       JsonElement el = jo.getAsJsonObject("formats").get("text/plain");
       if (el != null) {
            plainText = el;
       } else if ((el = jo.getAsJsonObject("formats")
                  .get("text/plain; charset=us-ascii")) != null) {
            plainText = el;
      } else if ((el = jo.getAsJsonObject("formats")
            .get("text/plain; charset=utf-8")) != null) {
              plainText = el;
      }
       return plainText.getAsString();
}
```

On a également remarqué que plusieurs documents étaient disponibles dans des archives au format `.zip`. On a créé une classe `Zip.java` permettant de récupérer le contenu du zip et de l'ajouter à notre bibliothèque.

4.2 Nos différents index

Pour effectuer une recherche rapide sans pour autant perdre trop d'informations. Nous utilisons différents index qui permettent un accès rapide aux données nécessaires selon le type de recherche, et cela sans avoir à parcourir tous les documents de la bibliothèque.

Nous utilisons au total 5 index pour le moment :

— `keywords` (335 Ko), qui manipule les mots-clefs des documents
— `titles` (72 Ko), utilisé pour la recherche par titre.
— `authors` (40 Ko), utilisé pour la recherche par auteur.
— `books` (341 Ko), utilisé pour renvoyer les données d'affichage au client (couverture, lien pour la lecture)
— `closeness` (33 Ko), permet d'accéder à la valeur de Closeness Centrality de chaque document

4.2.1 Structure de données utilisées

Nos index sont tous stockés via des `HashMap`, elles permettent un accès aux données en O(1) ce qui est très performant pour des données de grande taille.

Ils sont tous conservés dans des objets sérialisables dans le système de fichiers où le serveur est lancé. On a préféré ce système à une base de données plus traditionnelle car nous n'avons pas besoin de modifier ces données. Ce sont des données construites au préalable et chargées une seule fois au démarrage du programme. Une fois le serveur lancé, les données sont accessibles dans l'environnement d'exécution. La faible taille des index permet de les charger en très peu de temps.

```
16:05:40.700 [main] INFO com.sorbonne.daar.DaarApplication - Keywords loaded in 765 ms with 6587 keywords
16:05:40.779 [main] INFO com.sorbonne.daar.DaarApplication - Titles loaded in 56 ms with 1577 titles
16:05:40.844 [main] INFO com.sorbonne.daar.DaarApplication - Authors loaded in 65 ms with 612 authors
16:05:40.911 [main] INFO com.sorbonne.daar.DaarApplication - Closeness graph loaded in 67 ms with 1664 books
16:05:41.145 [main] INFO com.sorbonne.daar.DaarApplication - Books data loaded in 234 ms with 1636 books
```

FIGURE 4 – Chargement des index au lancement de l'application

Les temps de chargement sont influencés d'une part par la taille de l'index, mais également par le nombre d'entrées qu'il contient. On note que certains identifiants ne correspondaient à aucun document, ou alors ces documents ne contenait aucune metadonnée. C'est pour cela qu'on observe des tailles d'index différentes pour `books` par exemple, qui devrait avoir 1664 entrées.

Ci-dessous un extrait de code montrant comment les index sont chargés à partir du système de fichier.

```
// DaarApplication.java , Chargement de l'index des mots clefs
ObjectInputStream ois = new ObjectInputStream(new FileInputStream("keywords.ser"));
MotCleMap mcm = (MotCleMap) ois.readObject();
ois.close();
keywords = mcm.getMotCleMap();
LOGGER.info("Keywords loaded");
```

4.2.2 Construction

Pour construire nos index, nous avons également utilisé Gutendex comme pour la construction de la bibliothèque. On utilise les metadata des documents issus de Gutenberg. Ces données sont suffisantes pour construire les index `titles`, `authors` et `books`.

— Les index `titles` et `authors` ont une structure similaire, respectivement `HashMap<String, Integer>` et `HashMap<String, List<Integer>>`. Pour `titles`, les clés de la map sont les titres, la valeur l'identifiant du livre associé. On considère qu'un titre correspond à un seul document, cela peut être modifié en transformant les valeurs en `List<Integer>`. Pour `authors`, les clés sont les noms des auteurs et les valeurs sont les identifiants des livres qu'ils ont écrits. Ces index sont sauvegardés sous la forme d'objets sérialisés, respectivement `titles.ser`et `authors.ser`.

— L'index `books` est une `HashMap<Integer, Book>`. Il permet d'accéder aux métadonnées des ouvrages, tel que le titre ou encore la couverture. Il est principalement utilisé pour l'affichage côté client. Ci-dessous, la structure utilisée pour représenter les documents dans le système de fichier.

```java
// Book.java
public class Book implements Serializable{
   int id;
   String title;
   List<String> authors;
   String content; // lien externe vers le texte au format HTML
   String image; // lien externe vers la couverture du document
}
```

— Notre index `keywords` est plus complexes à construire. Il se présente sous la forme d'une `HashMap <String, List<Integer>>`. Les clés de la map sont les mots clefs issus des documents. Les valeurs sont les listes des identifiants des livres où ils sont présents. Pour récupérer les mots clefs de chaque livre, nous avons utilisé **Apache Lucene**. C'est une bibliothèque développée par la Fondation Apache qui permet d'extraire des mots clefs dans un document, elle est utilisée par **ElasticSearch**.

Elle permet de construire une liste des mots-clefs d'un texte tout en conservant leur fréquence. On peut donc obtenir une liste avec les mots-clefs d'un document ordonnée selon leur fréquence d'apparition dans le texte. On choisit de conserver les 50 mots-clefs les plus fréquents dans chaque texte pour ne garder que les mots les plus informatifs sur le contenu du document. Cela nous aide à obtenir des recherches plus rapides que si nous conservions tous les mots. Les mots clefs sont également récupérés en conservant leur racine principale. Ainsi, les mots `work`, `works` ou encore `worked` seront considérés comme le même mot ce qui permet de conserver efficacement l'information présente dans le texte.

Nous utilisons également une liste des mots usuels ne devant pas être comptabilisés car trop communs (déterminants, pronoms, etc.). Apache Lucene en propose une par défaut mais on observait trop de mots parasites alors nous avons utilisé une liste personnalisable. Cette liste est présente dans notre archive de rendu à `daar/stopWords.txt`. Nous avons donc pu également rajouter nos propres mots à interdire tels que `Gutenberg`, qui était présent dans la quasi-totalité des textes, ou encore les chiffres. Enfin, nous conservons dans notre index que les mots ayant moins de 30 occurrences dans la bibliothèque. Un mot présent dans de nombreux livres n'apporte que peu de contexte permettant de rechercher un ouvrage. Par exemple, le mot `work` était présent dans 1256 documents sur les 1664 disponibles, savoir qu'il est présent dans un texte est donc peu informatif.

L'index est conservé en tant qu'objet sérialisé dans le système de fichiers dans le fichier `keywords.ser`.

5 Algorithmes principaux

5.1 Aho-Ullman

Dans le cas d'une recherche sous forme d'expression Regex, nous réutilisons l'algorithme d'Aho-Ullman réaliser pour le premier projet DAAR1. Dans un premier temps, l'expression est transformée en arbre de syntaxe abstraite (AST)

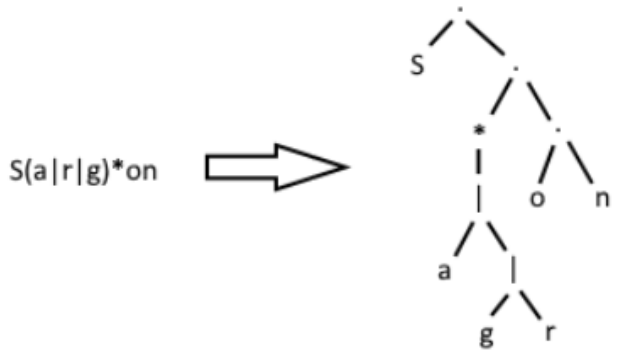

FIGURE 5 – Transformation d'une ERE en AST

L'AST est représenté par un type récursif `RegExTree` qui stocke le code ASCII de sa racine et une liste de sous-arbres. L'AST est ensuite traité pour être transformé en automate non-déterministe avec epsilon-transition selon la méthode d'Aho-Ullman (NFA).

Pour chacun des symboles d'opération d'une expression régulière, il existe une construction du NFA correspondant, on prend pour exemple le symbole étoile :

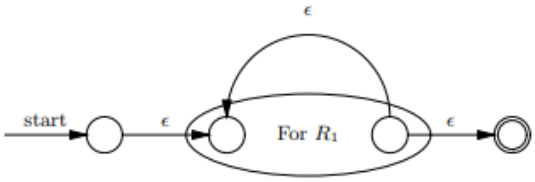

FIGURE 6 – Construction de l'automate pour le symbole plus

L'étape suivante consiste à transformer notre NFA avec epsilon transition en un automate fini déterministe (DFA), contrairement au NFA ou plusieurs possibilités de transitions peuvent exister simultanément pour un état et un symbole d'entrée donné, le DFA permet d'avoir une seule transition possible pour un état et un symbole d'entrée donné.

La dernière étape avant d'entamer la recherche, consiste à transformer le dfa en un DFA minimal, en effet pour chaque DFA il existe un unique DFA minimal équivalent avec un nombre d'états inférieur, le but est de trouver les états équivalents et de les fusionner.

Pour effectuer la recherche, on applique l'algorithme à chacun des mots-clefs présents dans l'index. On applique le DFA sur chaque caractère du mot en partant de l'état initial du graphe. Si on se situe sur un noeud final à la fin du mot, alors ce mot est validé comme correspondant au mot-clef recherché.

5.2 Knuth-Morris-Pratt (KMP)

Dans le cas où la chaîne recherchée est réduite à une suite de concaténations, nous pouvons appliquer KMP. Cet algorithme est plus rapide mais ne peut pas être appliqué à des expressions régulières. Nous l'utilisons dans le cadre de la recherche avancée si l'expression recherchée ne contient pas de regex. Le but de cet algorithme est d'éviter de comparer le motif recherché à des parties du texte qui ont déjà été comparés avec le motif.

Un traitement est effectué au préalable sur la chaîne à chercher pour obtenir un tableau de correspondance partielle. Ce tableau est construit en cherchant les plus longs préfixes qui sont aussi suffixes. Il indique d'où repartir lorsque le texte ne correspond pas au motif. Cela évite d'explorer du texte qu'on a déjà exploré et dont on sait qu'il n'apportera pas de nouvelles informations. On applique ici l'algorithme à chaque clé dans l'index de mots-clefs `keywords` pour chercher des correspondances.

mamamia	0|0|1|2|3|0|0
Sargon	0|0|0|0|0
" his "	0|0|0|0|1

FIGURE 7 – Tableau de correspondance partielle

Les valeurs du tableau sont utilisées au cours de la recherche via KMP. Lorsqu'on trouve une chaîne semblable au motif commençant à l'index `m` puis un caractère ne faisait pas partie du motif à l'index `i`, on regarde la valeur de la case précédente pour savoir d'où repartir. Soit une ligne L, on recommence à partir de `L[m + i - T[i-1]]`.

Si on a validé `n` caractères et que `n` est égal à la taille du motif, cela signifie que la clé possède une chaîne correspondant au pattern. On valide alors la clé et on ajoute ses livres correspondants à notre liste de résultats.

5.3 Jaccard

L'indice de Jaccard permet de déterminer si deux textes sont similaires ou non en se basant sur leurs contenus. On note A et B deux textes distincts, l'indice se calcule selon la formule suivante :

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

On divise le nombre de mots en communs dans les deux textes par le nombre de mots total contenus dans les deux textes. On utilise des `Set<String>` pour stocker tous les mots des textes car ils ne contiennent pas de doublons, ce qui est cruciale pour le calcul de l'indice de Jaccard, on utilise le nombre de mots distincts. On utilise la méthode `intersection` contenu dans le package `com.google.common`, elle permet de créer un Set avec les éléments en communs de deux Sets donnés en paramètre. Cela nous permet d'obtenir le nombre de mots en communs dans les deux textes. Enfin, on cherche à obtenir la distance de Jaccard qui se calcule simplement selon la formule $1 - J(A,B)$.

On cherche ensuite à créer un graphe de Jaccard. Ce graphe contient les distances de Jaccard entre chaque livres. C'est donc une matrice symétrique de taille 1664 x 1664, ce qui nécessite 2 768 896 calculs de distance. Bien que le calcule soit efficace pour deux textes simples, même de taille conséquente, la création de la matrice devient extrêmement chronophage.

On remarque que la taille du texte influence énormément le temps d'exécution du calcul de similarité. Celui-ci pouvait varier entre 20 secondes pour une ligne de la matrice (1664 appels) pour un fichier de `200 Ko` et 20 minutes pour un fichier de `44 Mo`.

L'algorithme de création peut être facilement optimisé car la matrice est symétrique, lorsqu'on a déjà la valeur opposée on a juste à la récupérer au lieu de recalculer Jaccard.

```
// utils.graph.Jaccard.java
// Construction du graph de Jaccard
for (int i = 0; i < 1664; i++) {
       // We get the first text
       String content1 = readFile("data", i, StandardCharsets.US_ASCII);
      for (int j = 0; j < 1664; j++) {
            if (j < i) {
                  // We already have the opposite value
                jArray[i][j] = jArray[j][i];
            } else if (i == j) {
                  // It's the same document, the distance is 0
                jArray[i][j] = 0f;
            } else {
                  // We get the 2nd text and compute the jaccard distance
                  String content2 = readFile("data", j, StandardCharsets.US_ASCII);
                  jArray[i][j] = distanceJaccard(content1, content2);
               }
      }
}
```

Cette optimisation nous permet de générer la matrice en 13h45 au lieu des 27h que nous avions initialement estimé. La matrice est ensuite sérialisée dans un fichier `jaccard.ser` d'une taille de `20,3 Mo`. Le fichier n'est pour le moment pas chargé à l'exécution du programme comme les autres index sérialisés car il n'est pas utilisé directement par l'application. Il est pour le moment uniquement utilisé pour générer notre index de Closeness Centrality pour l'ordonnancement.

Si on souhaite ajouter une fonctionnalité de suggestion qui exploite le graphe de Jaccard, il sera cependant nécessaire de le charger ce qui est également chronophage. La matrice met environ 40 secondes à se charger contre moins d'une seconde pour les autres index.

La similarité de Jaccard a toutefois ses avantages et ses inconvénients. D'une part son calcul est rapide et simple à exécuter. D'autre part, on perd une quantité élevée d'information due au fait que les mots ne sont comptabilisés qu'une seule fois, on ne tient pas en compte leur nombre d'occurrences. Cela se voit notamment via la présence de mots usuels ayant peu d'impact sur le contexte du livre (déterminants, pronoms, etc.). Plus un document est long, plus il aura de mots de ce type qui vont le rapprocher des autres sans pour autant avoir de lien direct. On remarque également que les documents que nous utilisons ont quasiment tous sans exception un gros bloc de texte concernant Gutenberg. Cela a pour effet de rapprocher les documents. Dans l'ensemble cela n'affecte pas la cohérence des distances de Jaccard car ils ont tous des blocs de textes similaires.

D'autres indices existent tels que le `Cosine Similarity` qui se calcule en transformant les textes en vecteurs puis en mesurant les angles entre les vecteurs. Le calcul est plus coûteux mais l'information obtenue est davantage précise.

5.4 Closeness Centrality

L'indice de Closeness Centrality dans un graphe permet de calculer la distance d'un noeud par rapport aux autres. Appliqué dans le graphe de Jaccard, cela nous permet de savoir si un noeud donne beaucoup d'informations sur l'ensemble des autres noeuds. Il se calcule selon la formule suivante :

$$Closeness(x) = \frac{1}{\sum_{i=1}^{1664} d(y,x)}$$

`d(y,x)` représente la distance dans le graphe de Jaccard entre les livres `x` et `y`. On calcule l'inverse de la somme des distances d'un livre dans le graphe de Jaccard. Ainsi, plus un livre est proche des autres, plus son indice de Closeness Centrality est élevé.

On conserve les indices des livres dans une `HashMap<Integer, Float>`, les clés sont les identifiants des livres et les valeurs sont leur indice de closeness correspondant. La map est utilisée une fois la recherche effectuée, pour ordonner les résultats en privilégiant les livres avec un indice de closeness plus élevé.

```java
/**
* daar.services.BookService.java
* Order the ids of a result Set based on the closeness graph
*/
public void orderResults(List<Integer> ids) {
    List<Integer> orderedIndexes = new ArrayList<>(DaarApplication.closeness.keySet());
    Collections.sort(ids, Comparator.comparing(id -> orderedIndexes.indexOf(id)));
}
```

Un des principaux avantages de Closeness Centrality est sa simplicité de calcul. Cependant, en pratique, il comporte certains inconvénients, notamment par rapport à la pertinence des résultats. La liste est figée une fois créée en prenant en compte tous les livres. Ainsi les mêmes documents seront toujours mis en avant par l'algorithme, même s'ils sont uniquement pertinents par rapport à des ouvrages qui ne correspondent pas au contexte de recherche actuelle. Cela se voit particulièrement quand des livres de langues différentes sont présents.

Dans notre bibliothèque, les livres écrits en anglais sont très largement majoritaire. Ils possèdent ainsi des distances faibles entre eux. Cependant il existe aussi des livres dans d'autres langues tel que le français. Par exemple, si on fait une recherche par auteur pour `"Verne"`, on aura toujours les ouvrages écrits par Jules Verne en premier et ses livres en français en dernier.

6 Tests

6.1 Tests fonctionnels

6.1.1 Recherche par mots clefs

Pour tester que la recherche par mots clefs fonctionne correctement, on utilise directement les ressources de notre index.

Dans notre index, le mot clef "nuclear" correspond aux identifiants 548 et 684. Ce sont deux ouvrages en liens direct avec le nucléaire, respectivement Project Trinity et Worlwide effects of Nuclear War. Quand on effectue la recherche avec le mot clef "nuclear" via la recherche basique de notre application, on obtient bien ces deux documents. De plus, on peut également effectuer cette même recherche via la recherche avancée, on obtient toujours ces deux ouvrages. Enfin, en modifiant la chaîne pour former une expression regex "nucl(ea)*r", la recherche avancée renvoie toujours les deux même livres.

6.1.2 Recherche par auteur

La recherche par auteur se teste de manière intuitive. On saisit la chaîne "verne" le moteur de recherche nous renvoie tous les livres de Jules Verne présent dans notre index. Quand on saisit balzac, on obtient tous les livres de Balzac.

6.1.3 Recherche par titre

Cette recherche fonctionne de manière similaire à la recherche par mots clefs. Quand on saisit la chaîne "jekyll", le moteur nous renvoie Dr. Jekyll and Mr. Hyde. Quand on rentre la chaîne "dr. jekyll", le moteur nous affiche les documents dont le titre contient soit "jekyll", soit "dr.".

6.2 Tests de performance

6.2.1 Aho-Ullman et KMP

Dans un premier temps, nous avons effectué des tests de performances pour les algorithmes de Aho-Ullman et KMP. Nous avons fait nos tests sur A History of Babylon, from the Foundation of the Monarchy to the Persian de Leonard William King, disponible sur gutenberg.org (n° 56667). Le texte fait 13300 lignes pour 750 000 caractères.

Nous avons fait varier la taille du document pour avoir différentes mesures. Dans un premier temps, on cherche à effectuer une comparaison en temps entre la recherche avec les deux méthodes de DFA et la recherche par KMP. On choisit comme exemple "Sargon" une expression composée uniquement de concaténation pour que KMP fonctionne. Cette expression ne correspond qu'à un seul mot dans le texte. De plus, elle commence par une majuscule et donc possède peu d'autres mots comparables. Ainsi, la boucle de recherche détecte que le premier caractère ne correspond pas très rapidement dans la plupart des cas, ce qui baisse le temps de recherche. Enfin, le fait que le mot soit composé uniquement de concaténations rend l'optimisation du DFA inutile, on devrait obtenir des temps très similaires pour les deux DFA. Pour la recherche par DFA, le document possède m lignes, on explore chaque ligne n fois avec n variant selon la taille du motif recherché. Plus le DFA est parcouru, plus l'algorithme est ralenti, on obtient ainsi des temps plus élevés si le motif commence par des caractères qui sont très présents dans le texte. La complexité en temps est donc en O(mn). La complexité en temps de KMP est en O(m). On s'attend donc à avoir des temps plus rapides pour KMP.

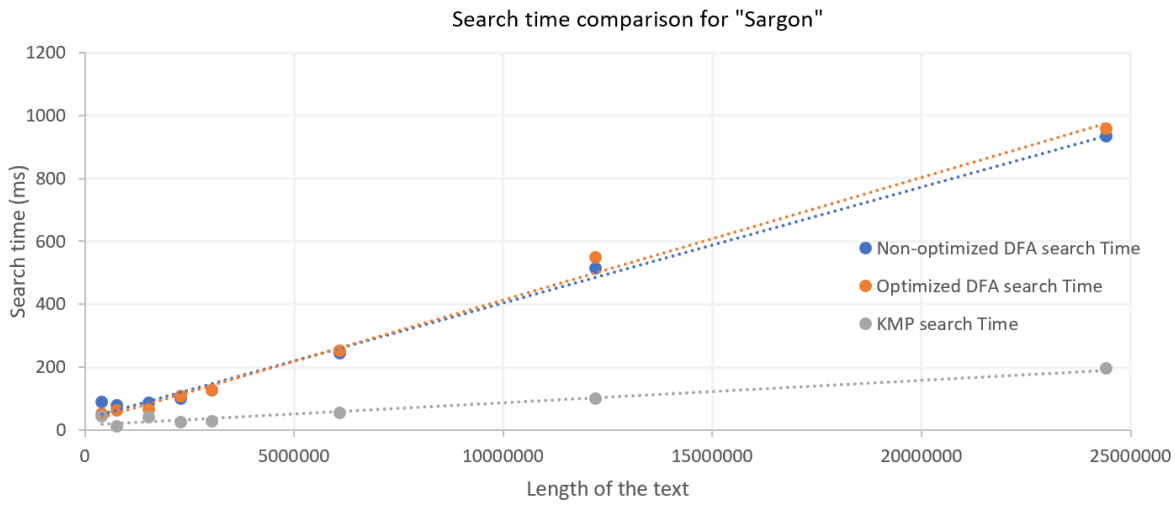

FIGURE 8 – Comparaison entre DFA et KMP

Conformément à notre analyse, on obtient des temps extrêmement proches pour les deux DFA : environ 940ms, moins de 20ms d'écart entre les deux pour 24 millions de caractères. Pour le même texte, KMP effectue la recherche en environ 190ms.

Enfin, on cherche à comparer l'efficacité entre un DFA optimisé ou non. On choisit un exemple qui, contrairement au précédent, va effectuer beaucoup plus de comparaison : `" h(e|is) "`. Le premier caractère est un espace suivit d'un `h` minuscule, deux caractères très présents ce qui va forcer l'algorithme à parcourir le DFA davantage que pour `"Sargon"`. On s'attend ainsi à observer des temps de recherche bien supérieurs que pour l'exemple précédent. Enfin, le DFA minimisé possédant moins de nœuds, on devrait logiquement avoir des mesures plus rapides pour celui-ci.

FIGURE 9 – Comparaison entre un DFA minimisé ou non

On obtient un temps de recherche d'environ 2000ms pour la recherche avec un DFA optimisé pour 24 millions de caractères. On observe des écarts entre 5 à 10 % à l'avantage du DFA optimisé.

6.2.2 Recherche Basique et Avancée

Nous avons également effectué des tests de performances pour comparer nos recherches basique et avancée. La recherche basique n'utilise pas d'expressions régulière, nous allons donc la comparer avec KMP qui est utilisé dans la recherche avancée. Dans le cadre de la recherche simple par mots clefs, nous utilisons la méthode `contains()` de `java.lang.String` pour faire nos comparaisons entre la chaîne entrée par l'utilisateur et notre index. On s'attend à avoir des résultats moins optimaux qu'avec KMP. Pour nos tests, nous utilisons la chaîne de caractère `"a"` car c'est celle qui possède le plus d'occurrences dans notre bibliothèque, 1560. Plus une chaîne a d'occurrences dans notre index, plus la recherche est longue. Par exemple, on observe une recherche d'environ **264 ms** pour la chaîne `"a"` contre un temps de **2 ms** pour la chaîne `"nuclear"` qui est beaucoup plus concrète et ne correspond qu'à deux documents.

FIGURE 10 – Comparaison entre KMP et la recherche basique

Pour 193 mots clefs dans notre index (10 livres), on observe des temps de recherche quasiment instantanés pour les deux méthodes. On obtient en moyenne **2 ms**. Pour 1179 mots clefs (100 livres), on obtient une fois encore des résultats très similaires, environ **12 ms** pour les deux méthodes. Pour des données plus conséquentes, on obtient des résultats légèrement différents. Pour un index contenant 6587 mots clefs (1664 livres), on observe environ **260 ms** pour la recherche basique et **240 ms** pour la recherche avancée avec KMP, soit une différence de **8%**.

7 Conclusion

Ce projet nous aura permis de découvrir le fonctionnement d'un moteur de recherche et de connaître différents critères d'ordonnancement permettant d'organiser les résultats. Cela a été une occasion de travailler avec une couche data conséquente composée de différents index. La création de ses index s'est avéré être une partie particulièrement chronophage à laquelle nous ne nous attendions pas forcément. Ces index sont dédiés à des ressources spécifiques, qui nous permettent d'effectuer des recherches rapides et efficaces.

En termes de perspective d'amélioration de notre application, on pourrait penser à un système de suggestion exploitant davantage le graphe de Jaccard, en se basant sur les recherches précédentes. On pourrait également implémenter différents critères d'ordonnancement tel que `Betweeness` qui propose des résultats plus pertinents que Closeness, ou encore `Pagerank`.