Sorbonne Université

Développement des Algorithmes d'Application Réticulaire

5 Février 2024

Projet final
-
Moteur de recherche d'une bibliothèque

Auteurs :
Daniel Sima
Walter Abeles

Spécialité :
Science et Technologie
du Logiciel
(STL)

Table des matières

1 Partie introductive

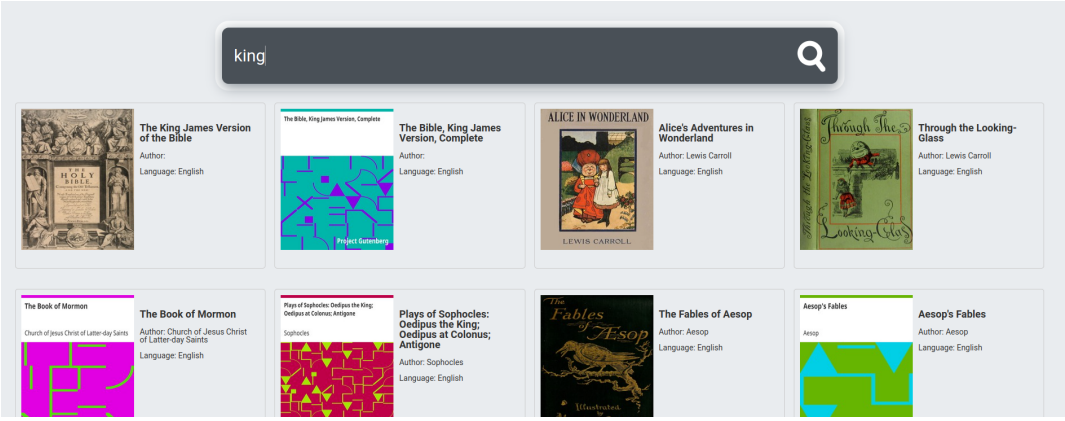

FIGURE 1 – Page d'accueil de l'application

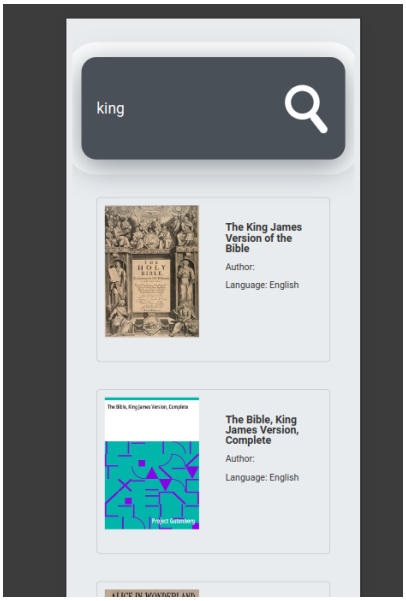

FIGURE 2 – Page d'accueil de l'application sur mobile

Dans la cadre de l'UE de DAAR, ce quatrième projet consiste en l'implémentation d'un moteur de recherche pour une bibliothèque sous forme d'une application web/mobile. La bibliothèque est fournie depuis le site Project Gutenberg [1] avec 1664 livres de prévu ayant au moins 10 000 caractères.
L'idée est de faire une version d'un moteur de recherche type Google dans ses premières versions.

1. https://www.gutenberg.org/

1.1 Analyse fonctionnelle

1.1.1 Use case

Un moteur de recherche de livres peut être utilisé dans divers contextes pour répondre à différents besoins utilisateur, parmi ces derniers on peut révéler :

Recherche générale

— Les utilisateurs peuvent effectuer des recherches générales pour trouver des livres en fonction de mots-clés, de titres, d'auteurs ou de genres.
  — Exemple : Un lecteur cherche des livres liées à l'indépendance des Etats-Unis.

Recherche Avancée

— Offrir des fonctionnalités de recherche avancées pour permettre aux utilisateurs de filtrer les résultats en fonction d'une expression régulière afin de couvrir plus de mots intéressants dans la recherche.
  — Exemple : Un lecteur souhaite trouver des ouvrages avec des mots difficiles, il utilise une RegEx afin de couvrir les tous les mots possibles.

Suggestions

— Offrir la possibilité d'accéder aux suggestions par rapport à un livre donnée.
  — Exemple : Un lecteur choisit un livre de science-fiction et se voit afficher des livres similaires en bas de la page.

Accès au format textuel du livre

— Offrir la possibilité d'accéder au format textuel du livre afin que les utilisateurs puissent lire le livre.
  — Exemple : Un lecteur choisit un livre donné et après recherche il souhaite le lire.

1.1.2 User-Stories

Pour développer ce site de recherche, nous avons décrit plusieurs user-stories représentant les comportements des utilisateurs sur les différentes pages du frontend. Ce procédé nous permet de développer et vérifier les fonctionnalités requises pour une utilisation efficace et agréable de l'application.

Ces user-stories se composent d'un descriptif des actions successives adoptées par l'utilisateur sur le site : - "En tant qu'utilisateur, je souhaite pouvoir rechercher avec un ou plusieurs mots les livres présents dans la base de données."
- "En tant qu'utilisateur, je souhaite pouvoir rechercher avec un ou plusieurs expressions régulières les livres présents dans la base de données."
- "En tant qu'utilisateur, je souhaite pouvoir consulter plus de détails concernant le livré recherche, notamment l'auteur, la langue du document, le texte contenu dans le livre." - "En tant qu'utilisateur, je souhaite pouvoir connaître les livres qui sont similaires au livre que j'ai sélectionné, sous la notion de suggestions."

On répertorie ensuite les fonctionnalités nécessaires à inclure sur le site, ici la barre de recherche, le bouton pour lancer la recherche, les résultats à afficher, etc...

1.2 Technologies utilisées

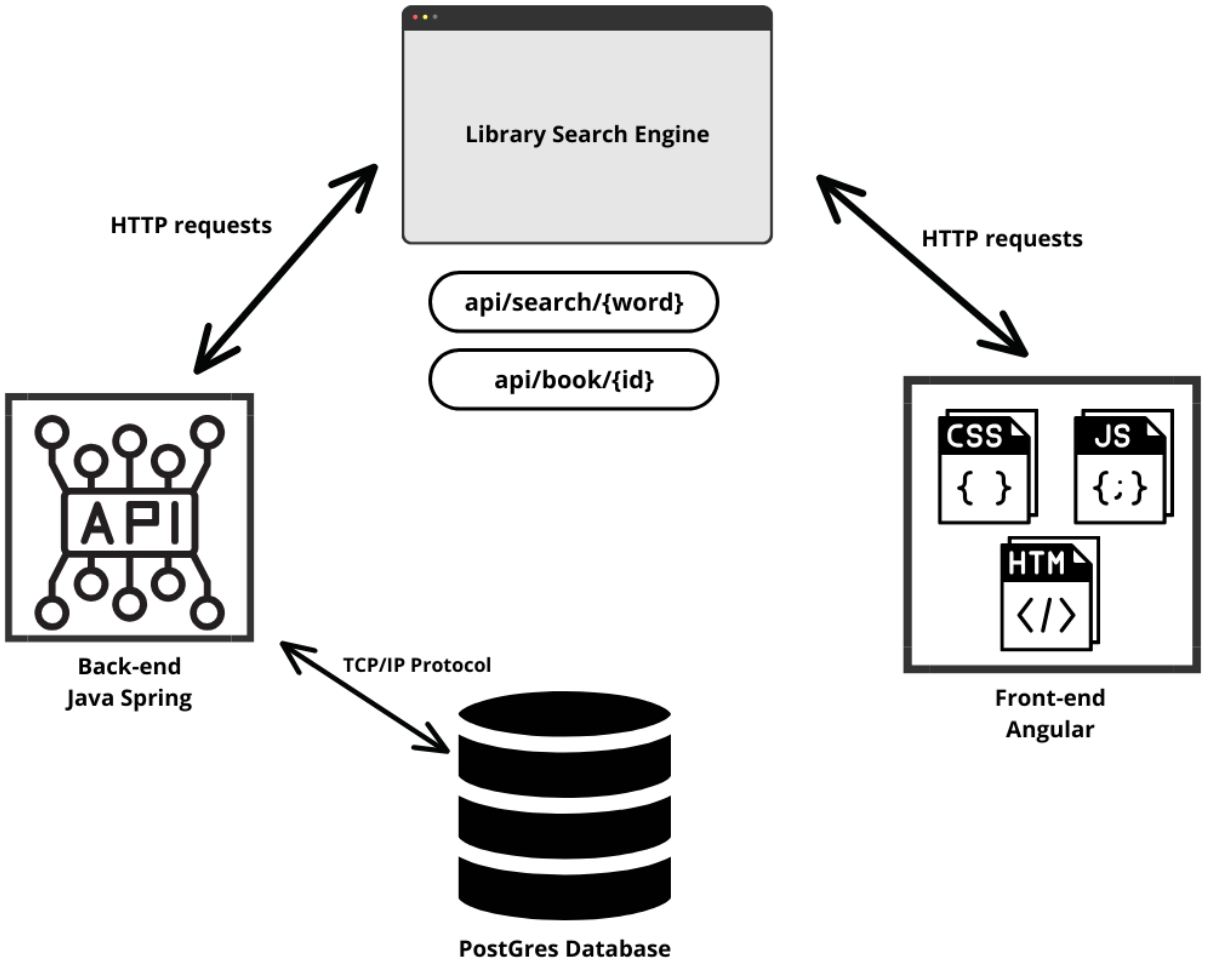

FIGURE 3 – Architecture du projet

Ce projet est réalisé sous forme d'application à trois niveaux. Le frontend, développé avec la technologie Angular, est accessible localement sur le port 4200. Il communique avec une API Rest développée avec Spring Boot en Java à l'aide de requêtes Http sur le port 8080.
L'API communique avec une base de données Postgres SQL en utilisant un protocole sur TCP/IP part le port 5432.
La couche data a été réalisée en Python avec les scripts fournis lors du TME9.

1.3 Bilan de ressources déployées

Nous avons réalisé un diagramme de Gantt pour structurer l'organisation de notre projet en plusieurs parties importantes :

FIGURE 4 – Description de Gantt

2 Partie technique

L'implémentation des différentes use-cases sera détaillé dans cette partie, en passant par la récupération des données depuis une API externe, leur filtrage et stockage dans le backend puis la mise en place de contrôleur pour réaliser les requêtes demandés par le utilisateur via le front-end.

2.1 Couche data

2.1.1 Récupération des données

Les documents que nous utilisons sont les livres en anglais hébergés sur le site **Gutenberg** avec au moins 10 000 mots.
Pour construire notre bibliothèque, nous avons utilisé **Gutendex**[2]. C'est une API web qui permet de récupérer les metadata des documents présents sur le site **Gutenberg** sous format JSON. Cela nous évite d'avoir à envoyer des requêtes directement à Gutenberg et parser les réponses qui sont des pages HTML.
Nous avons utilisé un script Python pour télécharger les 1664 premiers livres contenant au moins 10 000 caractères en parcourant les réponses JSON de Gutendex par Id des livres.

```python
import requests

guntendexURL = "https://gutendex.com/books/"
url_book = "https://www.gutenberg.org/cache/epub/47/pg47.txt"

# Get the list of books
response = requests.get(guntendexURL)
nb_words = 0
nb_texts_downloaded = 0
i = 0

while nb_texts_downloaded < 1664:
    url_book = "https://www.gutenberg.org/cache/epub/"+str(i)+"/pg"+str(i)+".txt"
    response = requests.get(url_book)
    if(response.status_code == 200):
        if(response.text == "Not Found"):
            print("Book "+str(i)+" not found")
            continue

        words = response.text.split()
        nb_words = len(words)
        print("Book "+str(i)+" has "+str(nb_words)+" words")
        if nb_words > 10000:
            with open("books/book"+str(i)+".txt", "w") as f:
                f.write(response.text)
            print("Book "+str(i)+" downloaded")
            nb_texts_downloaded += 1
        else:
            print("Book "+str(i)+" too short")
    else:
        print("Book "+str(i)+" not found")
    i += 1
```

FIGURE 5 – Script pour télécharger les livres depuis Gutendex

2. https://gutendex.com

2.1.2 Construction de la base

Une fois les livres téléchargés, nous avons filtré les mots qui ne sont pas pertinents pour la recherche textuelle. En effet, ces **stopwords**[3] peuvent être des conjonctions de coordinations, des pronoms ou toute autre sorte de mots qui sont récurrents dans ce genre d'ouvrages. Ce filtrage est effectué au moment de la génération de la base.
Dans les bases relationnelles il n'est pas possible d'ajouter plusieurs éléments dans une ligne afin de garantir l'atomicité des valeurs, nous avons relié plusieurs tables entre elles :

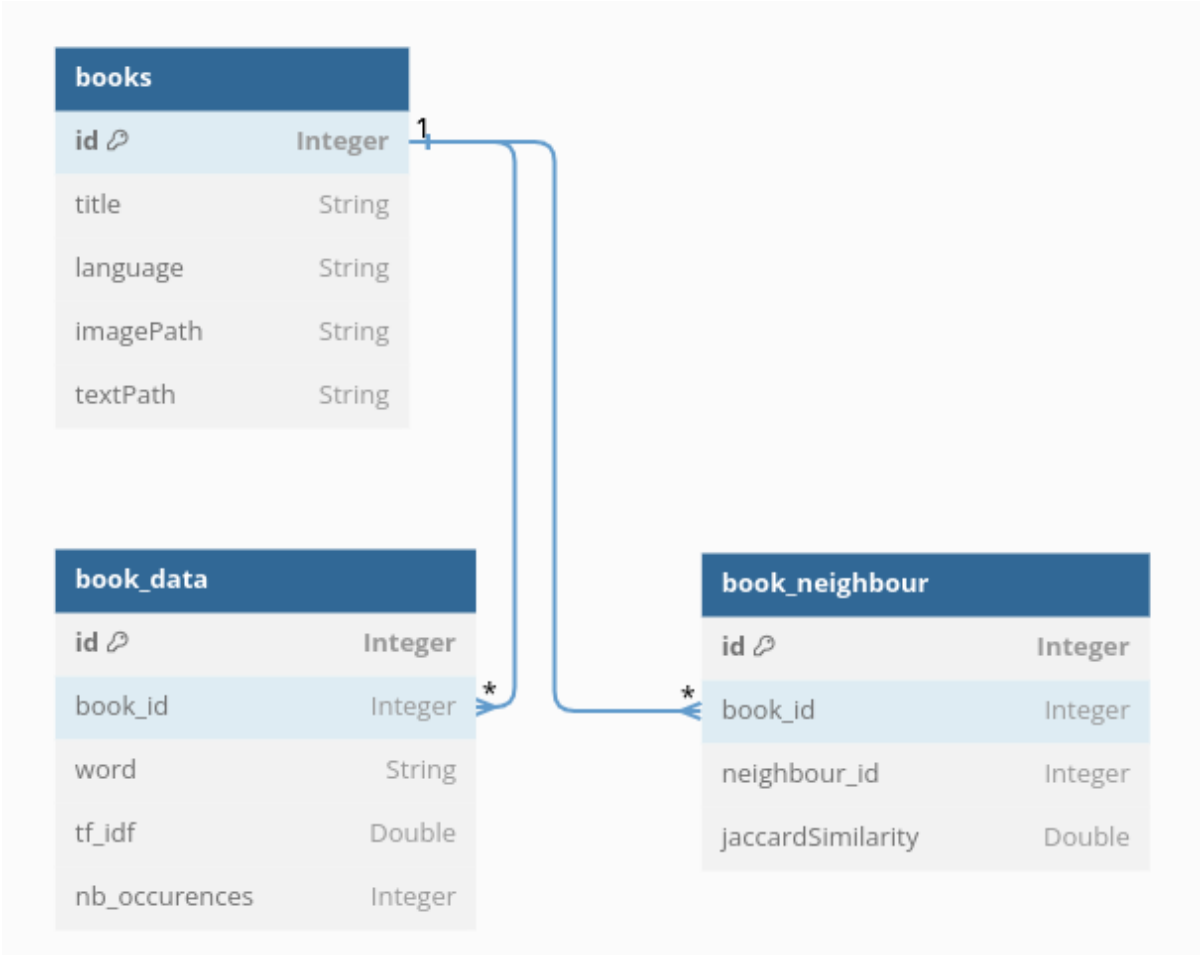

FIGURE 6 – Récapitulatif des liens entre les tables

Ci-dessus nous pouvons voir les différents liens entre les tables existantes, la table **books** comprends tous les livres dans la base de données, et elle est relié avec des realtions `One to many` à la table **book_data** et **book_neighbour**.
La table **book_data** correspond à un mot d'un livre avec différentes caractéristiques comme le `tf_idf` correspondant au **Term Frenquency * Inverse Document Frequency (TF*IDF)** et le nombre d'occurences `nb_occurences` du mot dans le livre avec comme clé étrangère `book_id`.
La table **book_neighbour** correspond à une arête du graphe de Jaccard avec comme poids la Jaccard Similarity `jaccardSimilarity`, elle relie un sommet `bool_id` à son voisin `neighbour_id`.

3. https://gist.github.com/larsyencken/1440509

2.2 Implémentation des fonctionnalités

2.2.1 Fonctionnalité explicite de “Recherche"

Pour effectuer une recherche simplifiée c'est-à-dire qui n'est pas une expression régulière, on utilise la formule **Term Frenquency * Inverse Document Frequency (TF*IDF)** [4] qui permet de déterminer dans quelles proportions certains mots d'un document texte, d'un corps de document ou d'un site web peuvent être évalués par rapport au reste du texte. Cette formule était utilisée dans l'une des première version du moteur de recherche **Google**. Elle utilise le critère de fréquence et peut être utilisée pour l'optimisation On-Page afin d'augmenter la pertinence d'un site web pour les moteurs de recherche, sans que la densité de mots-clés n'y joue un rôle unique.

La Term Frenquency TF permet de mesurer l'importance relative d'un mot dans un document en calculant son nombre d'occurrences dans le document et en divisant le résultat par le nombre total de mots dans le texte. Nous allons donc parcourir dans un premier temps tous les textes et compter le nombre d'occurrences de chaque mot qui ne sont pas des stop words. Cette étape est faite dans `BookConfig` via le `@Bean CommandLineRunner commandLineRunner(...)` qui sera exécuté a chaque fois que le sevrer et lancé et que la base de données et vide. Après avoir chargé les livres avec la fonction `load_book_TF()` de la classe `Utils` nous appliquons la formule suivante pour chaque mot de chaque livre :

$$TF = \frac{\text{number of times the term appears in the document}}{\text{total number of terms in the document}}$$

FIGURE 7 – Term Frenquency

L'Inverse Document Frequency IDF mesure la signification d'un terme en fonction de sa distribution et de son utilisation dans l'ensemble des documents. Plus un terme a de potentiel, plus l'**IDF** est élevée. Idéalement, un terme apparaît très fréquemment dans quelques textes seulement. Les mots qui apparaissent dans presque tous les documents ou très rarement n'ont que peu d'importance. On obtient le score par la formule suivante :

$$IDF = log(\frac{\text{number of the documents in the corpus}}{\text{number of documents in the corpus contain the term}})$$

FIGURE 8 – Inverse Document Frequency

On prend le logarithme car pour un gros corpus, la valeur de ce terme peut exploser. Une fois les deux termes calculés, on calcule sa valeur par :

4. https://www.learndatasci.com/glossary/tf-idf-term-frequency-inverse-document-frequency/

$$TF\text{-}IDF = TF * IDF$$

FIGURE 9 – TF*IDF

2.2.2 Fonctionnalité explicite de “Recherche avancée"

Dans le cas d'une recherche sous forme d'expression RegEx, nous réutilisons l'algorithme d'Aho-Ullman réalisé pour le premier projet DAAR.
Dans un premier temps, l'expression est transformée en arbre de syntaxe abstraite (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 (Voir Projet 1) [5]. Finalement, l'automate non déterministe est transformé en automate déterministe (DFA). Nous utilisons cet automate qui n'est pas minimal pour effectuer la recherche par RegEx.

Pour effectuer la recherche, nous appliquons l'algorithme pour générer l'automate déterministe une seule fois puis pour chaque livre nous prenons chacun des mots-clefs présents dans l'index. Pour ce faire, on applique la recherche sur le DFA pour chaque mot en partant de l'état initial du graphe. Si on se situe sur un nœud final à la fin du mot, alors ce mot correspond au mot-clef recherché.

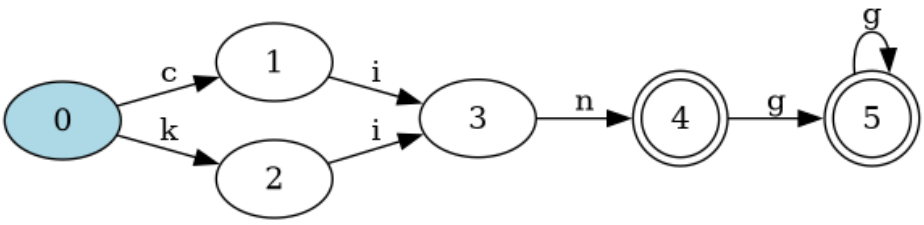

FIGURE 10 – Exemple pour la recherche du mot (k|c)ing*

2.2.3 Fonctionnalité implicite de suggestion

Une fois la recherche terminée, l'utilisateur pourra choisir le livre qu'il préfère et voir les informations le concernant, à cette occasion, des suggestions de livres lui sont présentés en bas de la page. Ces suggestions sont obtenues a travers la **Jaccard Similarity** :

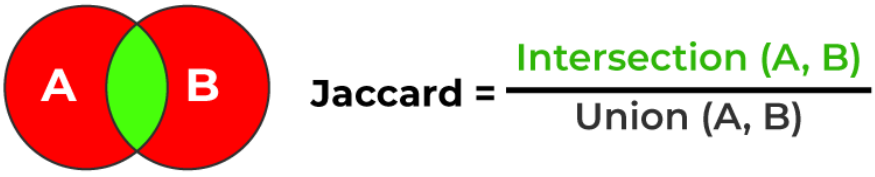

FIGURE 11 – Formule **Jaccard Similairty** de deux textes A et B

5. https://github.com/Daniel-Sima/Clone-of-the-egrep-UNIX-command

Ceci se fait également dans `BookConfig` via le `@Bean CommandLineRunner commandLineRunner(...)`, après l'obtention de **TF*IDF** ou nous appliquons la formule `jaccard_similarity(Book book1, Book book2)` de la classe `Utils` pour chaque livre de la base en faisant attention à ne pas répéter pour livres déjà calculés dans la boucle.
Nous décidons qu'un livre est une bonne suggestion d'un autre livre si la **Jacard Similarity** de deux livres est > 0.2.

2.2.4 Une fonctionnalité implicite de classement

Dans notre implémentation, la fonctionnalité de classement des mots importants se fait en fonction des occurrences du mot dans le texte en utilisant la méthode **TF*IDF** décrite précédemment. Nous ordonnons les livres résultat de la recherche par les mots qui ont le plus grand **TF*IDF**.

Nous aurions également pu l'implémenter en utilisant la **Closeness Centrality** mais la complexité temporelle est trop élevé en parcourant le graphe de Jaccard et en calculant les plus courts chemin de deux sommets distants.
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, s'il est "proche". Malheureusement la complexité temporelle lié au parcours du graph de Jaccard pour appliquer cet algorithme prend un temps considérable si nous l'appliquons à une base 1664 livres. Néanmoins nous allons vous expliquer son fonctionnement dans cette section.

Il se calcule selon la formule suivante :

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

$d(y, x)$ représente la distance la plus courte 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 muliplié au nombre de noeuds dans le graphe de Jaccard - 1. 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.

3 Tests

Cette partie traite des tests sur les fonctionnalités et également les tests de performance sur les différentes méthodes de recherche et sur la proposition de suggestions. Pour les établir nous utilisons une base de tests contenant 25 livres, les 25 premiers récupérés qui ont plus de 10 000 mots avec le script Python. Par manque de temps nous n'avons pas pu attendre le chargement de plus de livres, pour charger ces 25 livres cela a pris 40 minutes environ.

3.1 Tests de fonctionnalité

3.1.1 Recherche par mot

Pour tester que la recherche par mots clefs fonctionne correctement, on utilise directement la table d'indexage.
En cherchant le mot "constitution" sur les 25 livres de la base de données nous tombons sur 11 livres :

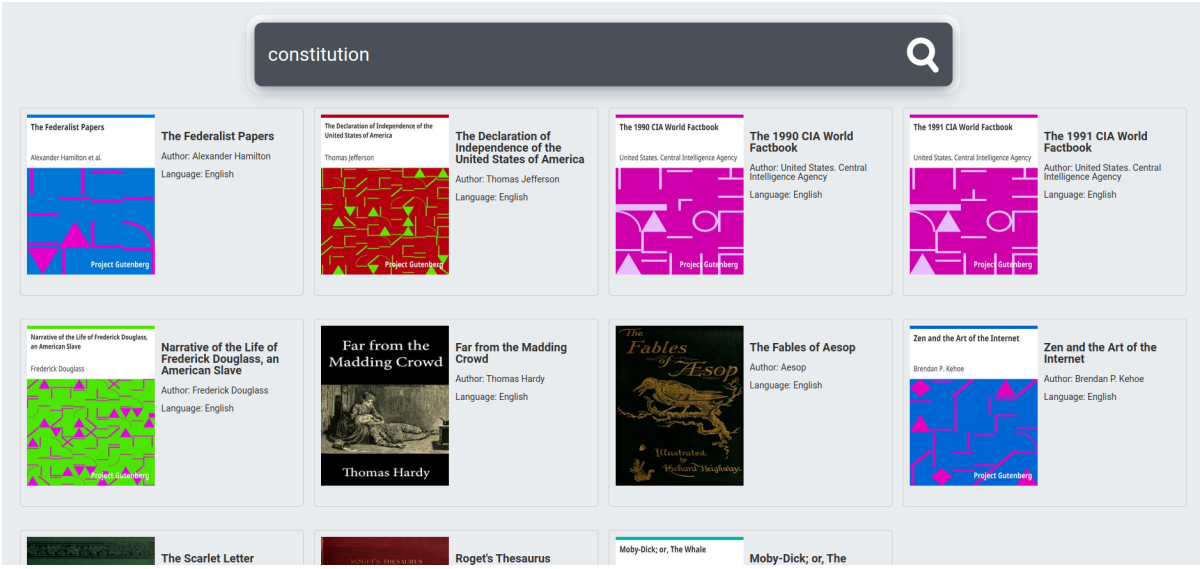

FIGURE 12 – Exemple pour la recherche du mot "constitution" sur les 25 livres de la base

En faisant une recherche pour le même mot mais avec la RegEx "c(o|x)nstitu*(t|l)ion" sur les 25 lives, nous tombons sur les mêmes 11 lives :

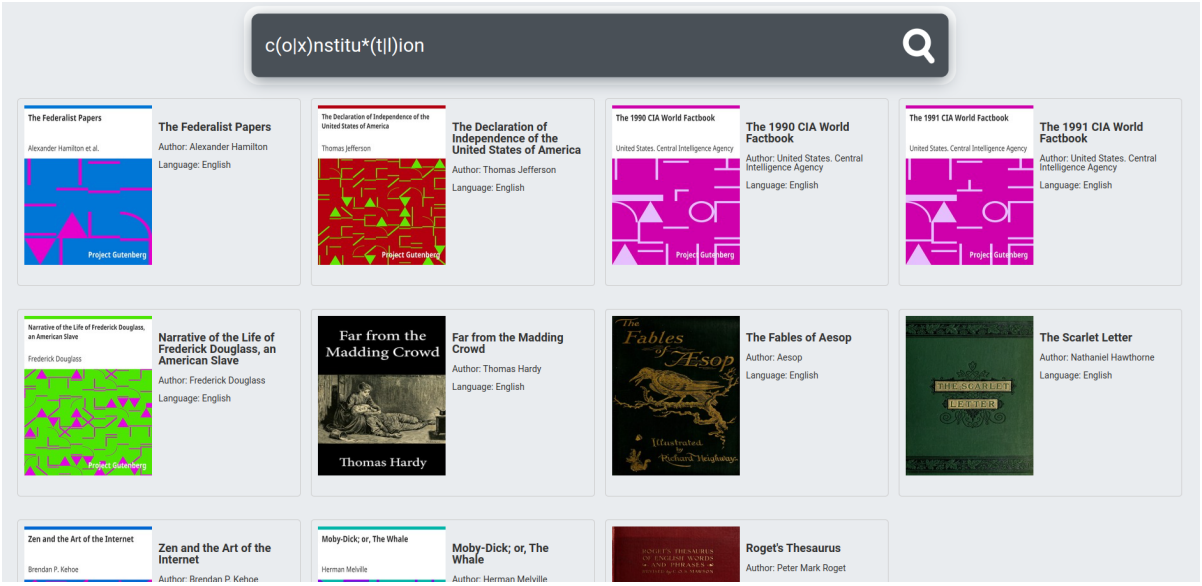

FIGURE 13 – Exemple pour la recherche avancée avec la RegEx "c(o|x)nstitu*(t|l)ion" sur les 25 livres de la base

La fonctionnalité implicite de classement par nombre d'occurrences du mot-clef dans le document est également vérifiée puisque nous retrouvons dans les deux cas la même suite de livres, les dernières livres ont le même **TF*IDF** et s'interchangeant.

3.1.2 Recherche par auteur et titre

La recherche par auteur se teste de manière intuitive. En saisissant la chaîne "Newton" le moteur de recherche nous renvoie tous les livres de Newton présent dans notre index. De le même manière pour le titre, le comportement est le même que pour les autres chaînes mais celles-ci sont spéciales puisque plus rares.

3.2 Tests de performance

3.2.1 Comparaison recherche Basique et Avancée

Nous avons également effectué des tests de performances pour comparer nos recherches basique, c'est-à-dire qui n'utilise pas les expressions régulières, et avancée. Pour cela nous avons testé les performances pour les algorithmes de Aho-Ullman et TF*IDF sur un mot clé.

Pour tester les performances dans le contexte de la recherche basique, nous avons pris le mot "constitution" et nous avons effectué la recherche sur un nombre croissant de textes pour observer le temps de réponse de chaque algorithme en fonction du nombre de livres. A noter que si on utilise une expression régulière, par exemple "c(o|x)nstitu*(t|l)ion", la complexité temporelle du processing avec Aho-Ullman reste inchangée. De ce fait les résultats sont sensiblement les mêmes lorsqu'on veut comparer les temps de calculs pour cet algorithme avec ceux obtenus en utilisant l'algorithme de TF*IDF.
Pour cela, nous utilisons la base de donnée pré-processed, c'est-à-dire que les stopwords ont été retirés des textes. On effectue la recherche sur un nombre de documents fini, c'est-à-dire un nombre de caractères fini.

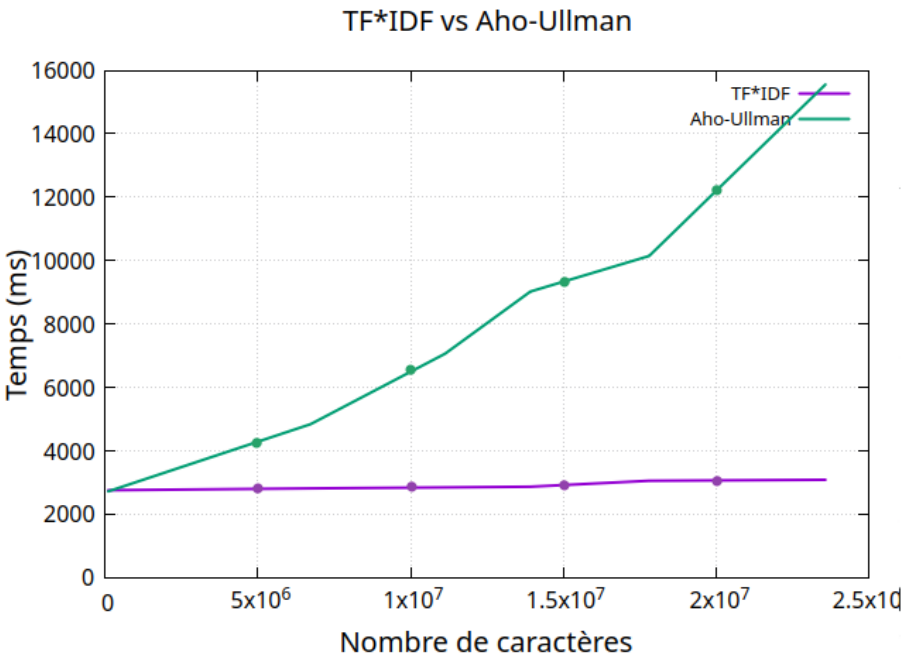

FIGURE 14 – Aho-Ullman vs TF*IDF

Chaque point représente un nombre de livres analysés qui augmente de manière linéaire de à 25. On observe que l'application de l'algorithme Aho-Ullman à des temps de calcul qui augmente de manière exponentielle au fur et à mesure que le nombre de caractères augmente. Ceci dû au fait que l'on doit analyser un grand nombre de livres qui peuvent avoir un nombre de caractère très élevé.
L'algorithme TF*IDF quant à lui est stable. Ceci s'explique, car il analyse directement les tables qui n'ont pas de doublons ni de stopwords.

L'algorithme TF*IDF est beaucoup plus performant lorsqu'il s'agit d'une recherche de mot basique néanmoins il ne permet pas une analyse des expressions régulières et

4 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.
Le manque de temps ne nous a pas permis d'exploité la couche data dans sa totalité.

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.
La recherche non pas de livres mais directement sur internet avec un Crawler web pourrait être egalement interessant.