Projet final – CHOIX A. Moteur de recherche d'une bibliothèque

Etudiants :
Abbas Mahfouz
Lea El Abboud

Enseignants :
Binh-Minh Bui-Xuan
Alfred Deivassagayame
Arthur Escriou
Guillaume Hivert
Suxue Li

Table des matières

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Plan du rapport . . . 1

2 Fonctionnalités et Technologies **2**
2.1 User Story . . . 2
2.1.1 Recherche de livres . . . 2
2.1.2 Exploration des résultats . . . 2
2.2 L'interface utilisateur . . . 2
2.3 Technologies et structure du projet . . . 3

3 Couche Data **5**
3.1 Traitement de textes de la base Gutenberg . . . 5
3.2 Recherche . . . 5
3.3 Suggestion . . . 5
3.4 Processus d'indexation . . . 5
3.5 Recherche d'images . . . 7

4 Les algorithmes principaux **8**
4.1 Algorithmes de recherche . . . 8
4.1.1 Aho-Ullman et Hopcroft . . . 8
4.1.2 Algorithme de Knuth–Morris–Pratt (KMP) . . . 9
4.2 Algorithme de classement des résultats . . . 9
4.2.1 Distance Jaccard . . . 9
4.2.2 Recherche du plus court chemin . . . 9
4.2.3 Betweenness centrality . . . 10
4.2.4 Closeness centrality . . . 10
4.2.5 Méthode de classement . . . 10

5 Résultats expérimentaux **11**

6 Conclusion **12**

1 Introduction

Dans le cadre du Projet Final – CHOIX A du module DAAR, nous avons conçu et implémenté une application web permettant la recherche et la navigation dans une base de données de livres numériques obtenus à partir de la base de Gutenberg [1]. Cette application intègre plusieurs fonctionnalités telles que la recherche par mot-clé, la recherche avancée via expressions régulières, un classement des résultats basé sur des indices de centralité et la suggéstion de contenus similaires.

1.1 Motivation

Le projet Gutenberg rassemble plus de 75,000 livres numériques non-soumis aux droits d'auteurs aux États-Unis téléversé par des contributeurs libres et rendu disponible aux lecteurs gratuitement. Cependant, la navigation parmi les ouvrages de cette collection ne support actuellement que la recherche par mot clés parmi les métadonnées des livres. Ce projet vise à rendre disponible une fonctionnalité de recherche plein-texte permettant aux usagers de trouver des livres dont le contenu textuelle correspond à leurs requêtes.

1.2 Plan du rapport

Dans ce rapport, nous présenterons dans un premier temps les fonctionnalités de l'application, avant de détailler l'architecture technique de notre solution. Nous expliciterons ensuite les algorithmes et méthodes employés pour le traitement des requêtes et l'indexation des documents. Enfin, nous conclurons en exposant les tests effectués et les perspectives d'amélioration envisagées pour ce projet.

2 Fonctionnalités et Technologies

2.1 User Story

L'application offre une expérience intuitive permettant aux utilisateurs de rechercher et d'explorer des livres facilement.

2.1.1 Recherche de livres

L'utilisateur peut effectuer une recherche en saisissant un texte dans la barre de recherche. Il a le choix entre deux modes :

— **Recherche par mot-clé (Keyword)** : Il entre un mot et l'application affiche tous les livres contenant ce terme.

— **Recherche avancée (Regex)** : L'utilisateur peut utiliser des expressions régulières (RegEx).

2.1.2 Exploration des résultats

Une fois la recherche effectuée, une liste de livres correspondants est affichée. L'utilisateur peut alors :

— Cliquer sur un livre pour afficher ses détails (titre, auteur, image, sujets abordés).

— Explorer des suggestions de livres similaires affichées sous la fiche détaillée.

2.2 L'interface utilisateur

Concernant l'interface utilisateur, nous avons opté pour un design simple, moderne et surtout intuitif :

Une page d'accueil (Figure 2.1) permettant d'accéder à notre bibliothèque digitale biblioshelf en cliquant sur le bouton "Let's get started".

FIGURE 2.1 – Page d'accueil

Une page de recherche (Figure 2.2) où l'utilisateur peut effectuer sa recherche.

En cliquant sur l'un des livres obtenus, l'utilisateur accédera à sa fiche détaillée (Figure 2.3) ainsi qu'à une sélection de livres suggérés.

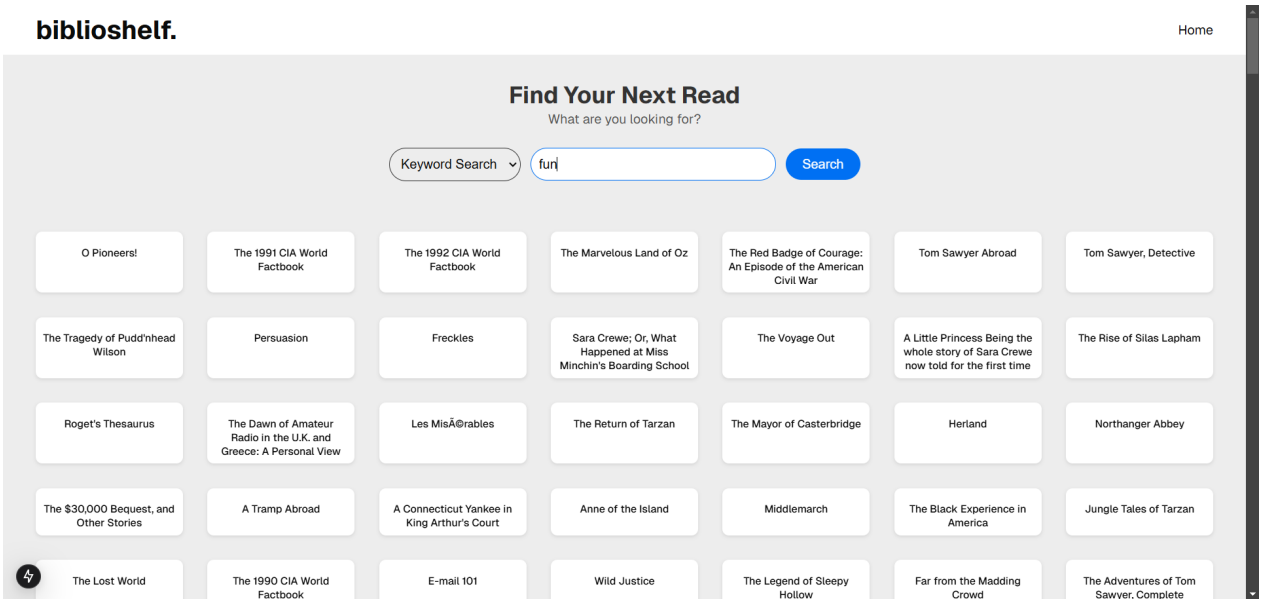

FIGURE 2.2 – Extrait de la liste des livres obtenus après la recherche du mot "fun".

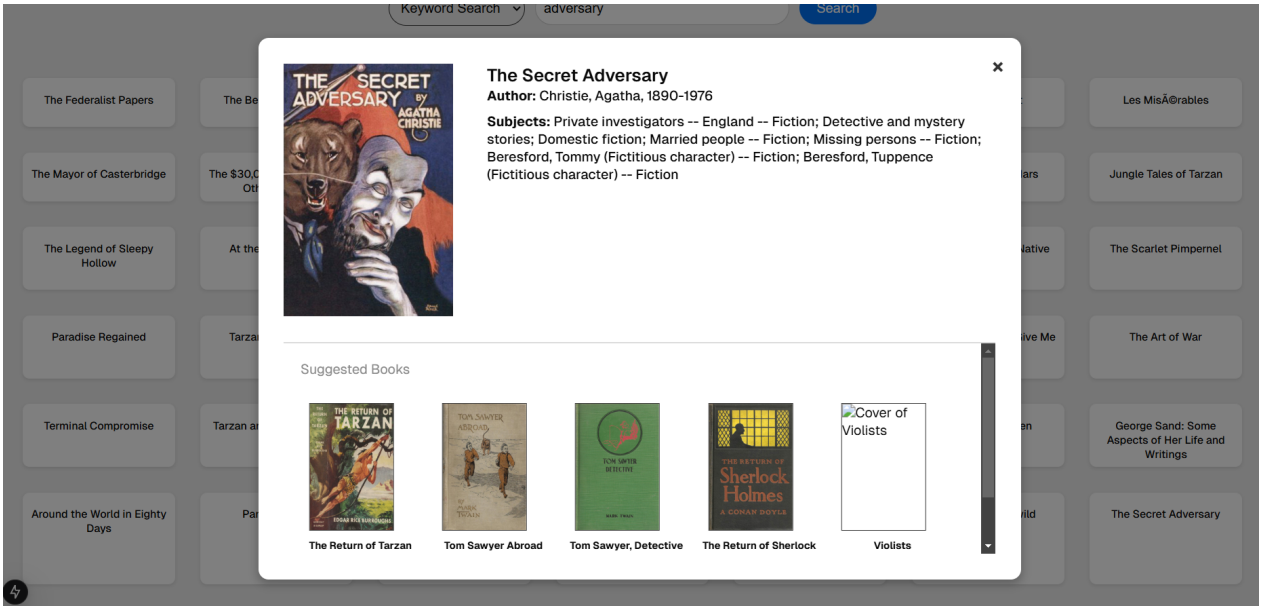

FIGURE 2.3 – Détails supplémentaires sur le livre sélectionné et suggestions

2.3 Technologies et structure du projet

Pour ce projet, nous avons utilisé **Flask** pour le serveur et **Next.js** pour le front-end. Flask, un micro-framework Python, a été choisi pour sa simplicité, permettant de créer une API rapidement et efficacement. Next.js, quant à lui, offre une interface utilisateur fluide, qui améliore la rapidité d'affichage des résultats. Cette combinaison assure une bonne communication entre le front-end et le back-end.

Pour cela notre projet est divisé en deux répertoires :

1. **Backend** : contient **bookService** et **imageService**
   - **bookService** :
     - Contient les fichiers nécessaires pour la recherche, notamment :
       - l'index global ;
       - les index individuels ;
       - le fichier de métadonnées ;
       - les données de distance entre les index.
     - L'ensemble de ces données peut être régénéré via les fonctions d'initialisation.

— Contient également :
  — le dossier de tests ;
  — le script permettant de générer les livres en texte intégral ;
  — les scripts pour les mesures de performance.
— **imageService** :
  — Contient le script permettant de lancer le serveur d'images.

2. **Frontend** : contient le répertoire **biblioshelf**
   — Contient les fichiers nécessaires pour générer la page d'accueil et la page de recherche.

L'architecture du projet en microservice en pratique correspond aux appels vers les APIs illustré dans la figure 2.4. Les requêtes sont traités aux niveaux des différents serveurs commençant par la transmission de la requête du client vers le bookService, qui rend en format JSON la liste des ouvrages correspondant à la recherche. Le client, munit de l'information sur quels ouvrages à afficher, fait des requêtes vers le service d'images afin d'obtenir les liens de la couverture et d'autres images pour inclure dans les informations de chaque livre.

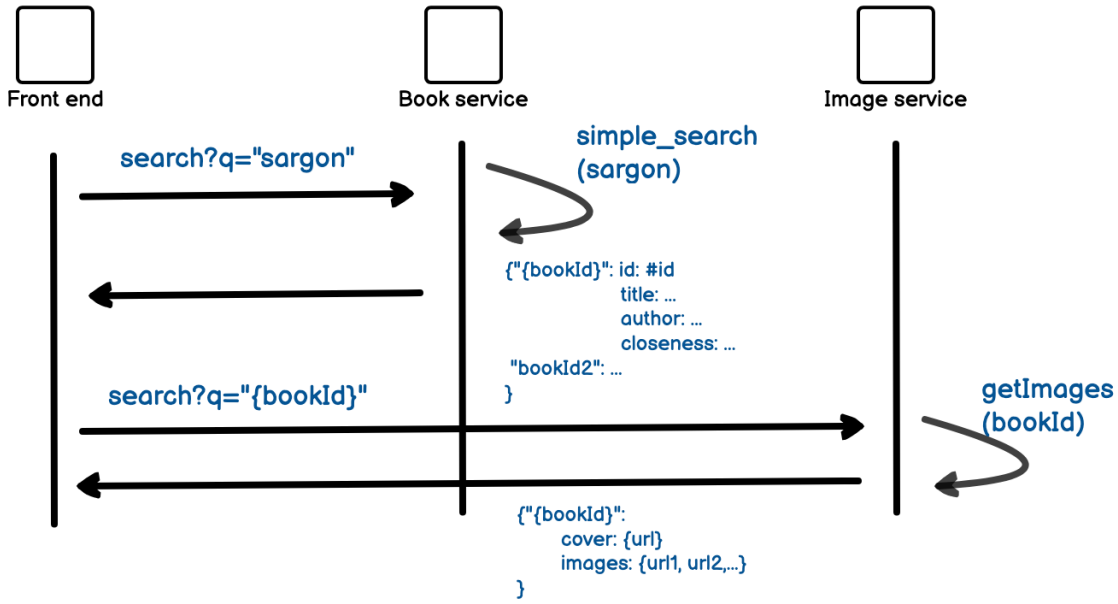

FIGURE 2.4 – Appels des APIs entre les micro-services et le client

3 Couche Data

3.1 Traitement de textes de la base Gutenberg

Pour faire une recherche à travers les documents de la base Gutenberg, restreint à 1664 exemplaires, nous avons d'abord introduit une fonction récupérant le fichier de métadonnées de la base du serveur gutenberg (mis à jour régulièrement), à partir duquel nous avons choisi le nombre voulu d'indices. Ensuite, nous avons utilisé la librairie `gutenberg` de Python pour accéder aux textes complets de chaque indice, servit par un des miroirs de la base recommandées pour les accès par machine, pour construire sa table d'indexage inversée, contenant la liste des mots, normalisés en minuscule et sans ponctuation, tout en mettant à jour un index inversé global contenant tout les mots trouver dans tout les textes.

Afin d'améliorer l'efficacité du moteur de recherche, nous avons effectuer un pré-calcul de la matrice de distance de Jaccard entre l'ensemble des tables d'indexage stocké ainsi que de la closeness centrality de chacune.

3.2 Recherche

La recherche est une fonction prenant en entrée une requête utilisateur et selon son choix procédant en une recherche simple ou d'expression régulière. La recherche simple rend l'ensemble des documents contenu dans l'entrée correspondante de la table d'indexage inversée. La recherche par expression régulière itère sur l'ensemble des clés de la table d'indexage globale et rend l'ensemble des indices de livres correspondant à la recherche. Les résultats sont rendu avec leurs indice de closeness au front-end, qui peut alors les ordonnées selon cet indice.

3.3 Suggestion

Selon le livre consulté dans la web-app, une requête est envoyée au serveur pour compiler une liste de 10 livres à suggérer, choisie selon le degré de ressemblance entre leurs thèmes et ceux du livre choisi. Cette sélection se fait en calculant la distance Jaccard entre les thèmes ainsi que la propriété "bookshelves" fournie par les méta-données de la base Gutenberg. Par exemple, la Figure 2.3 montre un livre d'Agatha Christie, célèbre autrice du genre policier, et en livre suggérés se joignent des livres d'autres ouvrages du même genre.

3.4 Processus d'indexation

Pour chaque livre de la base Gutenberg ajouté à notre base locale, l'entièreté de son texte est parcouru et pour chaque mot, un index local inversé pour le livre traité est mis à jour pour refléter soit la présence d'un nouveau mot soit une nouvelle occurrence d'un mot déjà reconnu comme une clé. Ce même parcours met également à jour un index inversé global contenant l'ensemble des mots de tout les textes traités par notre application avec les identifiants des livres dans lesquels ils apparaissent. Les mots servant comme clé sont uniformisés en minuscule et ne contiennent pas de ponctuation, sauf pour les tirets des compositions.

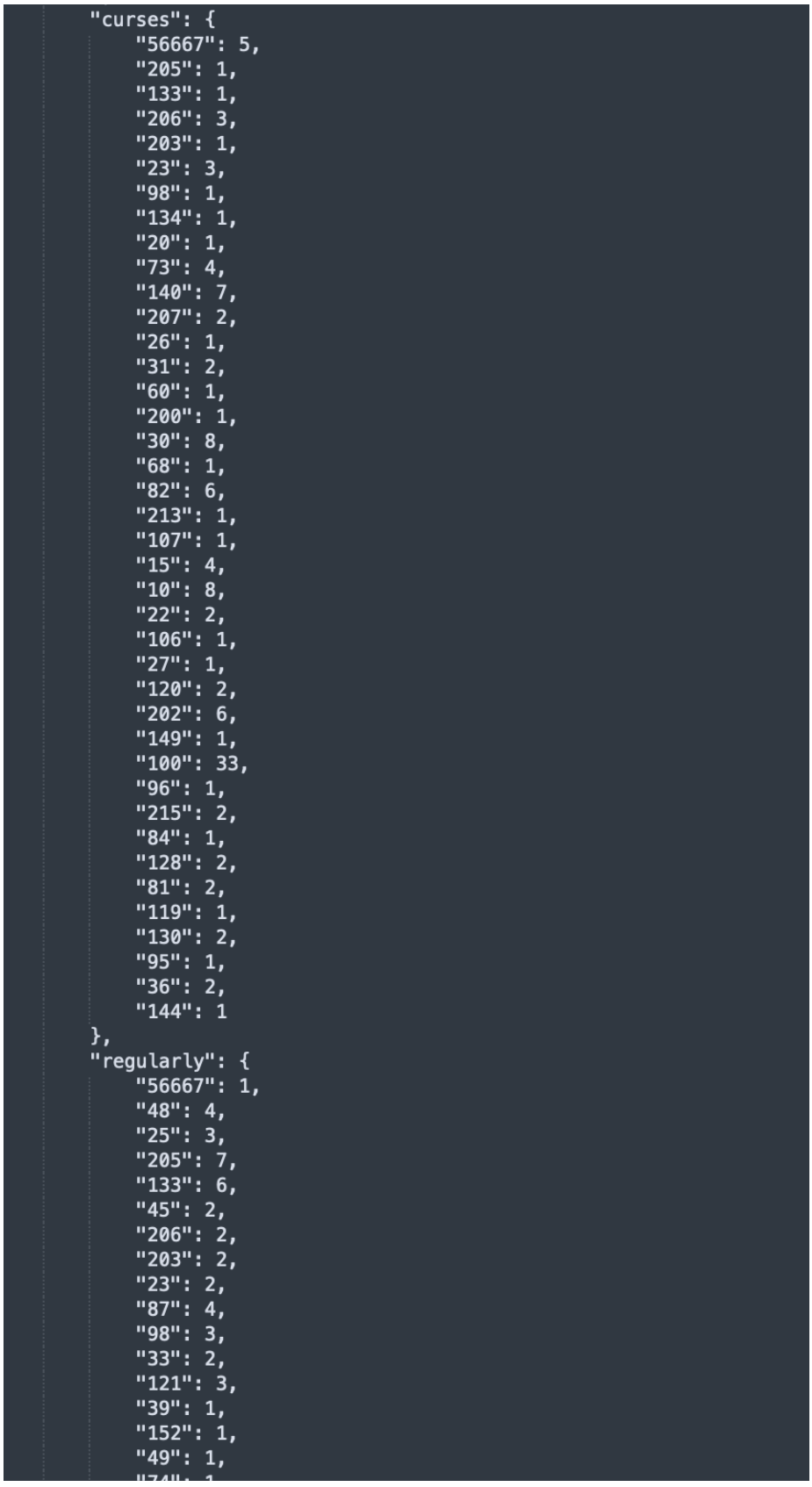

FIGURE 3.1 – Extrait de l'index inversée global

Dans 3.1 un extrait de l'index global montre la structure du fichier, qui inclut pour chaque mot apparaissant dans au moins un des textes indexée, l'identifiant du livre dans lequel il apparaît et le nombre d'occurrences dans chaque texte. Les indexes inversée de chaque texte suivent une structure similaire, sauf que pour chaque mot ils contiennent une liste de lignes dans laquelle ils apparaissent. La préservation des indexes individuelles est utile pour mesurer les distances entre les documents et établir leur centralités selon ces distances.

3.5 Recherche d'images

L'association d'un livre à ses contenus multimédia (pour le moment limités aux images), se fait d'abord par la transformation de son identifiant en un lien vers sa localisation dans le serveur Gutenberg. Cette transformation est offerte par une fonction de la bibliothèque Python `gutenberg`. Ayant obtenu le lien initial, la navigation vers le dossier contenant les images, si elles existent, suit une règle de chemin *hard-coded* et la page web correspondante est récupéré dans son entièreté. La page web, en format textuel, est ensuite traitée par une recherche du motif "f=\".+\.jpg\">" correspondant à un lien vers une image en format JPEG. L'ensemble des résultats est ensuite divisé selon les images apparaissant dans le texte ou les images de couvertures, identifié par le mot "cover" dans leur lien. Ces résultats sont ensuite rendus et transformés dans la web app en des liens traversables par l'application web pour afficher l'image. Cependant, le miroir utilisé pour accéder au contenu de la base Gutenberg ne contenait pas l'ensemble des images accessibles par l'hébergeur principal, résultant en des vides occasionnels en termes d'images.

La figure 3.2 visualise une réponse de l'API du service d'image à la requête avec l'identifiant de livre 56667 (*A History of Babylon, From the Foundation of the Monarchy to the Persian Conquest*), démontrant sa récupération de la couverture et de l'ensemble des images du texte.

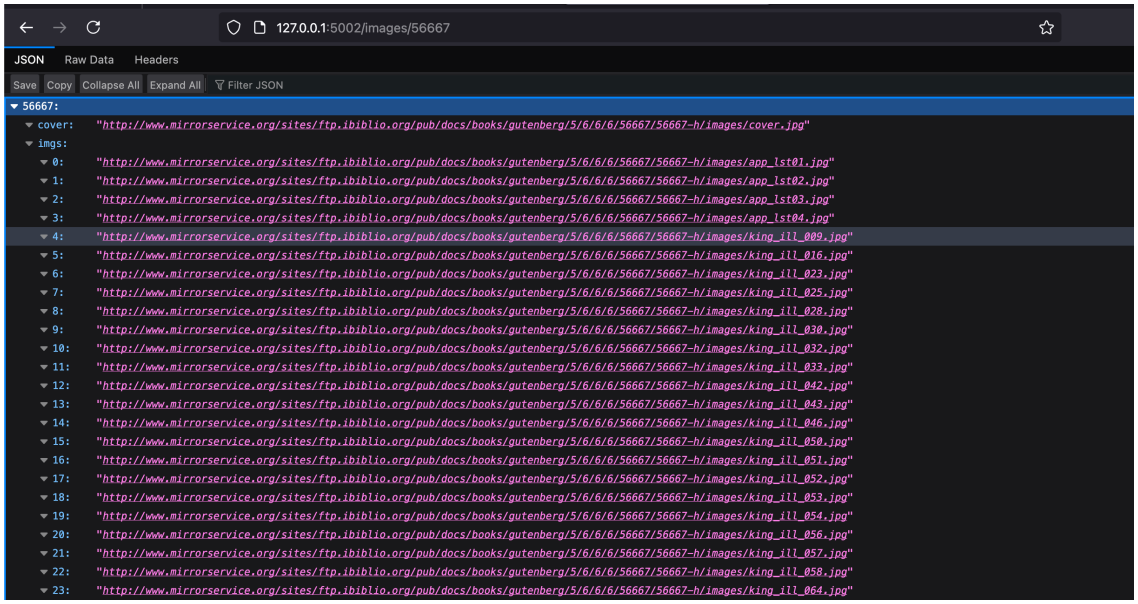

FIGURE 3.2 – Réponse du serviced d'image à une requête

4 Les algorithmes principaux

Dans le développement de notre moteur de recherche pour bibliothèque numérique, nous avons mis en place des algorithmes étudiés en cours, notamment **Aho-Ullman** et **Knuth-Morris-Pratt (KMP)** pour la recherche, un indice de centralité comme **Closeness centrality** pour le classement des résultats, ainsi que **Jaccard** pour la suggestion de documents, améliorant ainsi l'efficacité et la pertinence des réponses aux utilisateurs.

4.1 Algorithmes de recherche

L'application utilise plusieurs algorithmes pour assurer une recherche efficace et rapide selon le type de requête effectué par l'utilisateur. Pour une recherche par expression régulière, nous appliquons l'algorithme d'**Aho-Ullman**, qui repose sur la transformation d'une RegEx en un automate. En revanche, pour une recherche par mot-clé, nous utilisons l'algorithme de **KMP**, qui optimise la recherche de chaînes de caractères en évitant les comparaisons inutiles.

4.1.1 Aho-Ullman et Hopcroft

Utilisé dans le cas d'une recherche sous forme de RegEx, l'algorithme d'**Aho-Ullman** repose sur la transformation progressive d'une expression régulière en plusieurs structures : un arbre syntaxique (**RegEx Tree**), un automate fini non déterministe (**NFA**), et un automate fini déterministe (**DFA**), et enfin un **DFA minimisé** [2][3].

Conversion RegEx → RegEx Tree

L'expression régulière saisie par l'utilisateur est d'abord convertie en un arbre syntaxique.

Chaque opération est représentée par un nœud interne de l'arbre, tandis que les feuilles correspondent aux caractères de l'expression. Les principales opérations sont :

— **Concaténation** (.) : représentée par un nœud avec deux enfants.

— **Alternation** (|) : représentée par un nœud avec deux enfants indiquant les deux chemins possibles.

— **Etoile de Kleene** (*) : représentée par un nœud avec un seul enfant correspondant à l'opération répétée.

Par exemple pour l'expression `"a|bc*"` nous obtenons : `|(a,.(b,*(c)))` illustré par la Figure 4.1

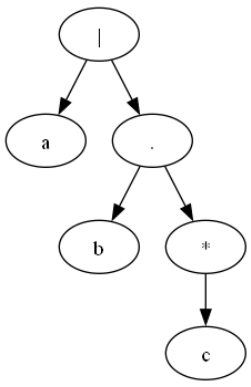

FIGURE 4.1 – RegEx Tree de l'expression `a|bc*`

Conversion Regex Tree → NFA

L'arbre syntaxique est ensuite transformé en automate NFA

— Pour une feuille contenant un caractère `A`, un NFA simple est créé avec un état initial, un état final et une transition `A` entre eux.

— Pour une concaténation `(A.B)`, on relie l'état final de `A` à l'état initial de `B` par une transition ε.

— Pour une alternation `(A|B)`, on crée un nouvel état de départ et un nouvel état d'arrivée, avec des transitions ε vers `A` et `B`, et des transitions ε reliant leurs états finaux à l'état d'arrivée.

— Pour une étoile de Kleene (`A*`), on crée un nouvel état initial et un nouvel état final, reliant l'état initial à l'état final par une transition ε et ajoutant des transitions ε vers `A` ainsi qu'une boucle de `A` à lui-même.

Conversion NFA → DFA

Pour améliorer l'efficacité, le NFA est transformé en un DFA. Cette construction repose sur la méthode de construction par sous-ensembles.[4]

1. **Représentation des états** : Chaque état du DFA correspond à un groupe d'états du NFA.
2. **État initial** : Le DFA démarre avec un état initial formé par l'état initial du NFA et tous les états atteignables via des transitions ε.
3. **Transitions** : À chaque symbole lu, le DFA regroupe tous les états accessibles depuis les états actuels du NFA et crée un nouvel état.
4. **Construction itérative** : Le processus est répété jusqu'à ce que toutes les transitions aient été explorées et que plus aucun nouvel état ne soit généré.
5. **États acceptants** : Un état du DFA est acceptant si l'un des états du NFA qu'il regroupe l'est.

Conversion DFA → DFA minimisé

Une fois le DFA construit, une phase de **minimisation** est appliquée pour réduire le nombre d'états. Pour cette conversion, nous avons utilisé l'algorithme de **Hopcroft**, qui est plus efficace que celui d'**Aho-Ullman**.

L'algorithme fonctionne par divisions successives : il commence par séparer les états (acceptants et non-acceptants), puis affine progressivement ces groupes en considérant les transitions entre les états. Le processus se poursuit jusqu'à ce qu'il n'y ait plus de groupes à diviser. Le résultat final est un DFA minimal, plus compact, mais qui reconnaît exactement le même langage que l'automate d'origine.

4.1.2 Algorithme de Knuth–Morris–Pratt (KMP)

Dans le cadre de la recherche de livre par mot-clé (chaîne réduite à une suite de concaténations), nous utilisons l'algorithme KMP [5].

La particularité de l'algorithme **KMP** est d'éviter de comparer plusieurs fois les mêmes parties du texte en cas d'échec de correspondance. Pour cela, un **prétraitement** est effectué sur le motif afin de construire une **table de correspondance partielle**. Cette table identifie les plus longs préfixes du motif qui sont aussi des suffixes, permettant de savoir où reprendre la recherche sans revérifier inutilement les caractères déjà comparés. Cela accélère la recherche par rapport aux méthodes plus naïves.

4.2 Algorithme de classement des résultats

4.2.1 Distance Jaccard

Afin de calculer la closeness centrality des différents textes, nous établissons une métrique de distance entre deux textes, précisément entre leurs tables d'indexage, en prenant la distance Jaccard entre eux. Cette distance identifie le nombre de mots en communs entre les deux textes.

$$D_J(t_1, t_2) = \frac{\left| t_1 \cap t_2 \right|}{\left| t_1 \cup t_2 \right|} \tag{4.1}$$

Le calcul de l'ensemble des distances se fait en temps quadratique, notant que cet opération se fait une fois à l'initialisation, varie selon le nombre de mots uniques dans chaque texte et nécessite globalement que le calcul du triangle supérieur des valeurs par symétrie de la matrice de distance.

4.2.2 Recherche du plus court chemin

Ayant une matrice de distance, elle peut être transformer en un graphe en définissant un seuil pour qu'il y ai une arête entre deux sommets. Le calcul des chemins les plus courts entre tout les sommets s'est fait par l'application itérative pour chaque sommet de l'algorithme de **Dijkstra** [6] : Cet algorithme prend en entrée un graphe G de sommets V, où chaque sommet v a pour voisins $N(v)$, ainsi qu'un sommet à partir de laquelle recherche commence. Il rend les chemins les plus court de ce sommet à tout les autres du graphe sous la forme de longueur du chemin le plus court entre v et chaque autre sommet ainsi que la liste de sommets précédents pour arriver à chacun des sommets par un chemin plus court.

Algorithm 1 Djikstra plus court chemin

```
Q ← {∀v ∈ G}
dist ← ∀v ∈ G  dist{v} = ∞
Q{source} = 0
while | Q |> 0 do
    u = Q.pop(v)  s.t.  d{v} < d{v'} ∀v ≠ v ∈ Q        ▷ pop l'élément ayant la distance minimale
    for v in Q ∩ N(u) do
        alt = dist{u}+alt
        if alt ≤ dist{v} : then                        ▷ ≤ permet d'avoir plusieurs chemin
            dist{v}=alt
            prev{v}+=alt
        end if
    end for
end while
```

4.2.3 Betweenness centrality

La **betweenness** d'un sommet mesure sa présence parmi l'ensemble des chemins les plus courts entre tout les sommets d'un graphe. Il s'agit du rapport entre le nombre de plus courts chemins pathsu, v, t entre v et t passant par u sur le nombre de plus courts chemins entre u et t.

$$\textbf{Betweenness Centrality } (v) = \sum_{u \neq t \neq v \ \in \ G} \frac{\#\text{paths}(v, u, t)}{\#\text{paths}(u, t)} \tag{4.2}$$

Le retraçage des plus courts chemins à partir du rendu de **Djikstra** se fait par un algorithme récursif `reconstructPath(src,dst,prev)`, qui prend les sommets source et destination et la tableau de précédence rendu par **Djikstra** :

Algorithm 2 reconstrucPath

```
paths ← {}
if src ∈ prev(dst) then
    return prev(dst)
else
    for v ∈ prev(dst) : do
        paths += ( {v} + reconstructPath(src,v,prev))
    end for
end if
```

4.2.4 Closeness centrality

Le degré de closeness d'un sommet v se mesure en calculant le rapport entre la taille des sommets du graphe et la somme de sa distance, par le plus court chemin, de tout les autres sommets.

$$\textbf{Closeness Centrality } (v) = \frac{\mid V \mid -1}{\sum_{v \neq u} d(v, u)} \tag{4.3}$$

La distances des plus courts chemins entre les paires de sommets est rendu par l'algorithme de Djikstra en choisissant l'un d'entre eux comme point de départ.

4.2.5 Méthode de classement

Après l'implémentation des méthode de calcul de betweeness et de closeness, le classement ultimement fut décidé par la mesure de closeness de chaque texte par rapport à tout les textes indéxés.

5 Résultats expérimentaux

Le bon fonctionnement de l'application fut vérifié par un test d'intégration consistant à charger la page, naviguer à la page de recherche entre une requête dans le mode recherche simple, vérifier les résultats, vérifier que le visionnage détaillé des livres fonctionne et que les recommandations sont cohérentes puis répéter ce processus avec une recherche avancé. La cohérence de nos résultats fut vérifier en comparant les indices de textes retournés avec les résultats de la commande système `egrep` lancé sur un ensemble de fichiers des contenus plein-textes des mêmes livres indexés par notre application (télécharger par un script uniquement pour des raisons de tests). La performance de l'application fut également mesuré en comparant au temps de réponse de `egrep`. Pour ces tests, un ensemble réduit de textes fut choisi et stocké dans un dossier dans lequel une commande `egrep` fut lancé sur plusieurs motifs listé dans les figures ci-dessous et des appels vers le serveur faisant des recherches avancés furent lancés aussi pour comparé la précision des réponses du serveur et sa vitesse.

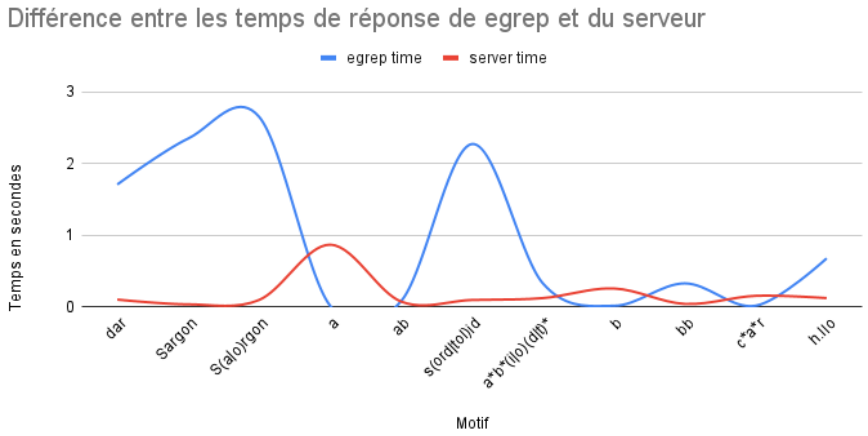

FIGURE 5.1 – Temps de calculs pris par `egrep` et le serveur pour chercher différents motifs

Nous pouvons voir qu'à part du motif "a" le serveur rend des résultats plus vite que `egrep`, que nous pouvons créditer par la récupération des valeurs de l'index global qui évite de parcourir les mots répétés, réduisant ainsi l'échelle des mots à parcourir.

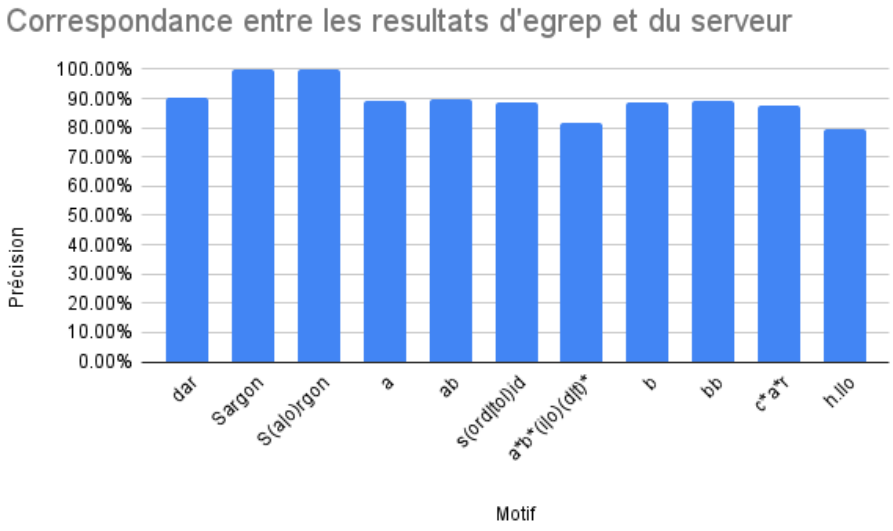

FIGURE 5.2 – Correspondance entre les résultats rendu par `egrep` et le serveur pour chercher différents motifs

Par contre, en terme de précision le serveur ne rend pas exactement les mêmes réponses que `egrep`, que les tests confirment comme des omissions de résultats.

6 Conclusion

L'application web développée permet d'effectuer une recherche en plein texte dans un sous-ensemble des livres de la base Gutenberg, tout en incluant des fonctionnalités de classement des résultats selon leur pertinence, mesurée par l'indice de **closeness centrality**. Elle propose également des suggestions de livres similaires en fonction des thèmes déclarés dans les métadonnées des ouvrages.

L'efficacité de la recherche textuelle repose sur un pré-calcul des indices de chaque livre, permettant une recherche simple en temps constant et une recherche avancée dont le temps de traitement dépend du nombre de mots uniques plutôt que du nombre total de mots.

L'ensemble des fonctionnalités de ce projet a été divisé en micro-services, notamment en séparant la recherche textuelle et la recherche d'images. Cependant, plusieurs perspectives d'amélioration demeurent. Concernant l'indexation, celle-ci pourrait évoluer d'une liste de mots complets vers des arbres de suffixes. En termes de calcul de distance entre les tables d'indexation, d'autres mesures comme `TF-IDF` pourraient être envisagées. En termes de fonctionnalités, l'accès aux textes des livres retournés par la recherche pourrait également être mis en place.

Pour l'interface utilisateur, il serait aussi intéressant d'ajouter une "sidebar" permettant de filtrer davantage les résultats, par exemple en sélectionnant les livres obtenus par la recherche en fonction de leurs sujets ou d'un auteur particulier.

Bibliographie

[1] Project Gutenberg https://www.gutenberg.org/

[2] Lecture 1 - Searching a RegEx (regular expressions) https://www-npa.lip6.fr/~buixuan/files/daar2024/daar1.pdf

[3] Foundations of Computer Science, Chapter 10 : Patterns, Automata, and Regular Expressions - Al Aho and Jeff Ullman http://infolab.stanford.edu/~ullman/focs/ch10.pdf

[4] Wikipédia, "Construction par sous-ensembles", disponible sur https://fr.wikipedia.org/wiki/Construction_par_sous-ensembles

[5] Lecture 2 - Searching a string (factor of a word) https://www-npa.lip6.fr/~buixuan/files/daar2024/daar2.pdf

[6] Wikipédia, "Dijkstra's algorithm", disponible sur https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm