CHOIX A : moteur de recherche d'une bibliothèque

Rapport de Projet

Clément GRESH
Yajie LIU
Zimeng ZHANG

Développement des Algorithmes d'Application Réticulaire (DAAR) 09/04/2023
M2 STL-INSTA
Sorbonne Université

Table des matières

Table des figures

I. INTRODUCTION

Le but de ce projet est de créer une application web permettant de faire des recherches par mots-clés ou par expressions régulières (REGEX) sur une bibliothèque, i.e. une base de données (BDD) conséquente de documents textuels. Pour créer cette application, il a fallu mettre en place une BDD ainsi que développer un *backend* et un *frontend* (Partie II).

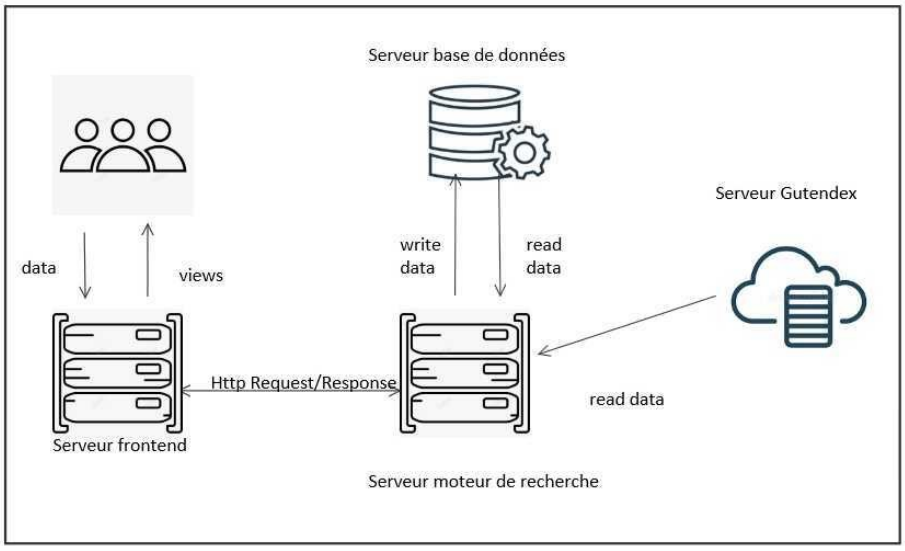

Figure 1: architecture de l'application

A son premier lancement, l'application récupère plus de 1500 documents textuels de la base Gutenberg. Ceux-ci sont traités au fur et à mesure de leur ajout : les mots-clés qu'ils contiennent sont extraits grâce à des algorithmes de recherche (Partie III) et sont enregistrés dans la BDD. Un graphe de Jaccard est également créé et stocké. Celui-ci permet l'implémentation des fonctionnalités de classement et de suggestion (Partie IV). A chaque nouveau lancement, l'application récupère dans la BDD les informations qui lui sont nécessaires et le *frontend* permet d'envoyer des requêtes pour effectuer des recherches.

Enfin, des tests (Partie V) ont été effectués pour évaluer les performances du moteur de recherche et sa viabilité.

II. ARCHITECTURE

1. Base de données

Le système de gestion de BDD utilisé pour ce projet est MongoDB. Ses principaux avantages sont qu'il est orienté documents et qu'il ne nécessite pas de schéma prédéfini des données. Les schémas peuvent donc directement être définis et modifiés dans le *backend*. Dans le cas d'une modification, il suffit pour générer à nouveau la BDD de la supprimer puis de relancer l'application.

La table *library* contient tous les documents téléchargés à partir de la base Gutenberg.

Les tables IndexTable, TitleIndex et AuthorIndex contiennent respectivement les mots-clés trouvés dans le corps, le titre et la liste des auteurs des livres. Il n'est possible de faire des recherches par REGEX que sur les mots-clés de IndexTable.

La table *Term2Keyword* contient la racine des mots-clés trouvés. Par exemple, les différentes conjugaisons d'un verbe ne sont rattachées qu'à un seul et unique mot-clé.

Enfin, la table BookGraph contient le graphe que forment les livres (les poids des arêtes étant les distances de Jaccard) ainsi que leurs centralités de proximité.

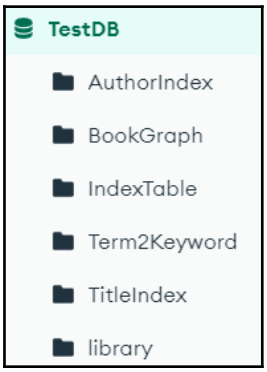

Figure 2: tables de la BDD

2. Backend

C'est le langage *Java* et le *framework Spring*, associé à la configuration *Spring Boot*, qui ont été choisis pour coder le *backend* de l'application. Ils permettent en effet de créer rapidement une nouvelle application grâce, par exemple, aux composants gérant les interactions avec la BDD. L'architecture en termes de classes est la suivante :

- le **contrôleur** définit les *endpoints* à utiliser et appelle le service quand il reçoit une requête du *front*. Il renvoie ensuite le résultat au *front* sous la forme d'un objet de la classe *ResponseEntity* qui permet de complètement définir la réponse HTTP.
- le **service** demande les données dont il a besoin aux différents répertoires, les traite et les renvoie au contrôleur.
- les **répertoires** sont en fait des interfaces qui se contentent d'étendre la classe générique *MongoRepository* en lui donnant les types des entités à récupérer et leurs clés primaires.
- les **entités** (e.g. *IndexTableData*) permettent de définir les tables de la BDD MongoDB, ainsi que les objets à lui envoyer et à y récupérer.

De plus, les classes *DownloadBooks*, *ProccessBook* et *BookGraph* ont pour mission de créer la BDD lors du premier lancement de l'application, puis de récupérer les listes de mots-clés et le graphe de Jaccard lors des lancements suivants.

3. Frontend

Les outils utilisés pour le *front* sont *Vue.js*, *Element Plus* et *Axios*. La bibliothèque de base de *Vue.js* se concentre sur la couche de visualisation, ce qui facilite sa prise en main. *Element Plus* quant-à-lui fournit un ensemble riche de composants d'interface utilisateur tels que des formulaires ou des fenêtres contextuelles. Enfin, *Axios* permet la conversion automatique de données dans des formats tels que JSON, XML ou encore FormData.

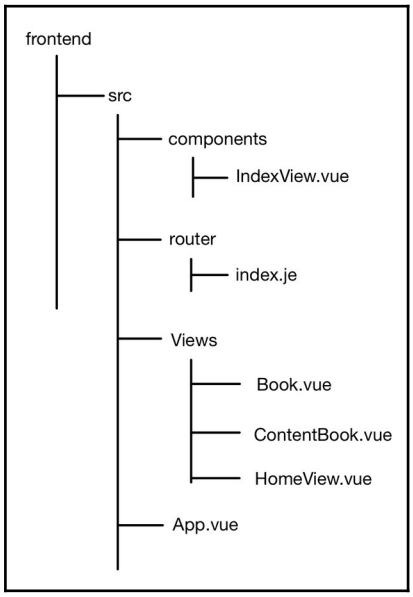

Figure 3: architecture frontend

Via la page d'accueil, il est possible d'effectuer des recherches par mots-clés sur le contenu, le titre, ou l'auteur d'un livre. Il est également possible d'utiliser des expressions régulières (limitées aux symboles '*', '+' et '|') pour les recherches sur le contenu. L'option de recherche avancée est désactivée si « author » ou « title » est sélectionné. Une recherche renvoie jusqu'à 10 documents ainsi que des recommandations liées aux trois d'entre eux les plus pertinents.

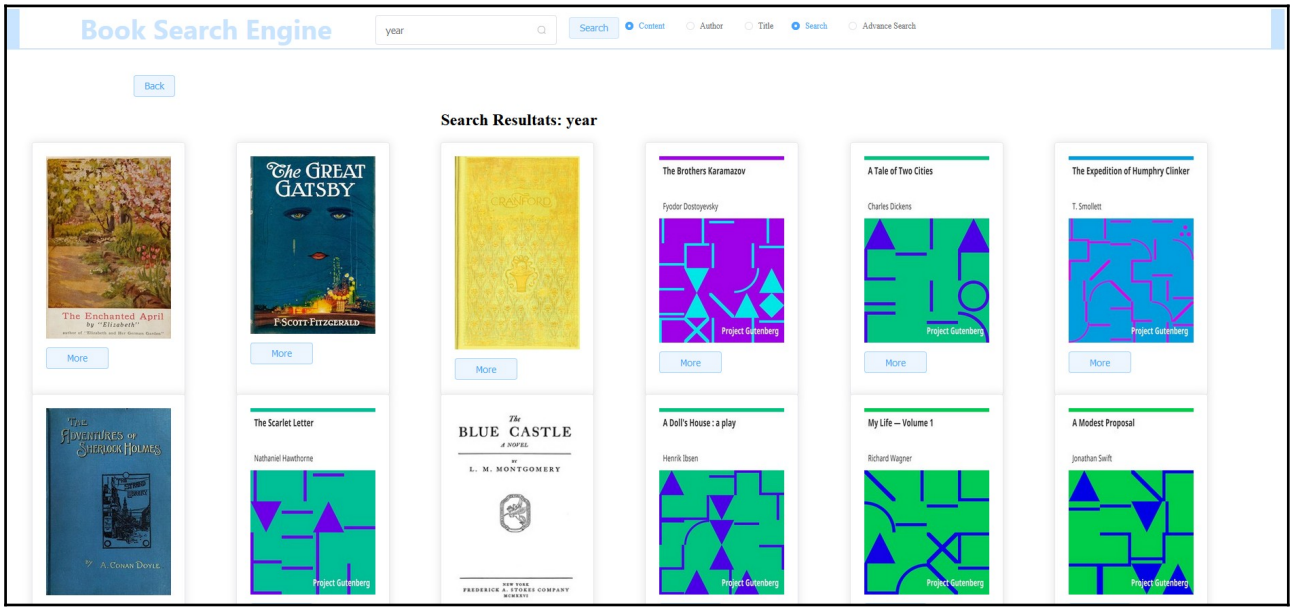

Figure 4: page de recherche

La sélection d'un livre permet d'accéder à une page contenant des informations détaillées sur celui-ci, ainsi qu'un lien vers l'intégralité de son contenu. Sont regroupés également des suggestions d'autres documents en lien avec le livre sélectionné.

III. RECHERCHE

1. Recherche par REGEX

La recherche par expressions régulières (REGEX) se compose de 5 étapes (d'après le chapitre 10 du livre *Foundations of Computer Science* dont seront extraits certains exemples présentés dans ce rapport) :

- déterminer la REGEX avec des parenthèses (ce qui revient à construit un arbre).
- déterminer un automate fini non-déterministe ayant des ε-transitions.
- déterminer un automate fini déterministe.
- minimiser cet automate (étape optionnelle non abordée ici).
- effectuer la recherche dans un texte.

Dans les 4 premières étapes, on ne prend en entrée que la REGEX et la réponse de l'étape précédente. C'est seulement à la 5ème étape que l'on utilise le texte à rechercher. Cette dernière étape est également la plus importante en termes de complexité temporelle. Il est peu utile d'essayer d'optimiser les autres. L'algorithme a, selon les cas, une complexité en $O(n^2)$ ou $O(n*k)$, où n et k sont respectivement les tailles du texte à chercher et de la REGEX. Il n'est pas possible d'obtenir une complexité linéaire.

La réponse de cet algorithme peut être :

- un booléen.
- le nombre d'occurrences (c'est le cas de notre algorithme)
- une liste de numéros de ligne (c'est le cas pour *egrep*).
- une liste de numéros de ligne correspondant à la REGEX ainsi que le numéro du 1er caractère.

a) REGEX parenthésée

Obtenir une REGEX complètement parenthésées revient à obtenir un objet de classe RegexTree ayant une racine et une liste de sous-arbres.

Pour cela, on crée une liste d'arbres correspondant chacun à un caractère de la REGEX. On parcourt cette liste jusqu'à ce qu'elle ne contienne plus qu'un seul arbre contenant tous les autres. On traite dans l'ordre : les parenthèses, '*' pour les caractères facultatifs, la concaténation (i.e. une suite de lettres), '|' pour l'alternative.

```
public static RegExTree parse(ArrayList<RegExTree> result) throws Exception {
  while (containParenthese(result)) result=processParenthese(result);
  while (containEtoile(result)) result=processEtoile(result);
  while (containConcat(result)) result=processConcat(result);
  while (containAltern(result)) result=processAltern(result);
```

Figure 5: analyse de REGEX (classe RegEx)

Remarques :

- On se limite à ces éléments (pas de '+', de '.', etc.)
- Les règles de précédence pour analyser une REGEX peuvent être trouvées ici.

Exemple

En entré, la REGEX : *a|bc**

En sortie, l'arbre ci-contre correspondant à l'expression : (a|(b(c*)))

Le point représente la concaténation (et non une *wild card*).

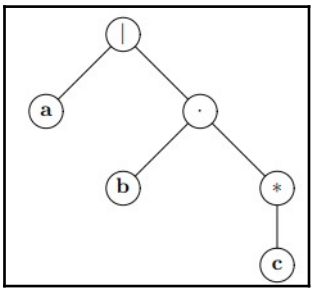

Figure 6: REGEX tree

b) Automate fini non-déterministe (NDFA)

On cherche maintenant à obtenir un NDFA, i.e. un automate ayant des ε-transitions pouvant être franchies sans conditions. On utilise pour cela l'arbre obtenu à l'étape précédente et l'algorithme d'Aho & Ullman.

On parcourt celui-ci de façon *Bottom-Up* en consommant d'abord ses feuilles. Si, en remontant, on rencontre '*', cela signifie que le caractère peut être présent 0, 1 ou plusieurs fois. Il faut donc ajouter des ε-transitions. Avec '|', il faut ajouter deux ε-transitions correspondant aux deux choix possibles. L'automate final ne peut avoir qu'un seul état initial et un seul état final.

Exemple : avec *(a|(b(c*)))* en entrée.

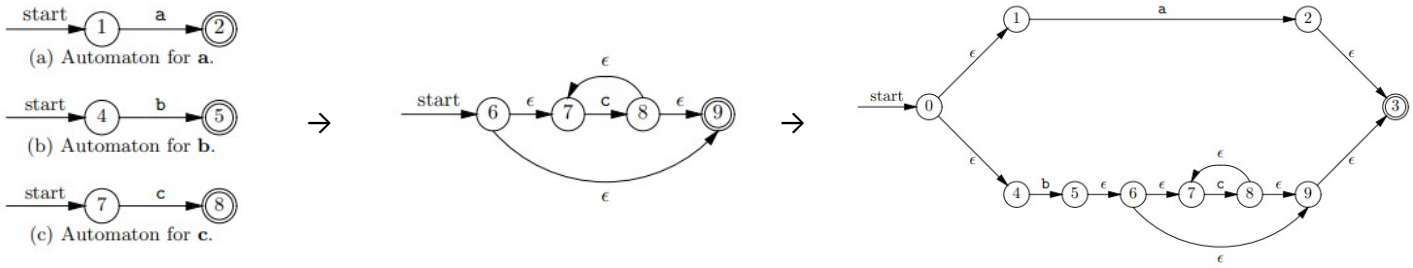

Figure 7: étapes d'obtention d'un NDFA

Dans le code, cet automate est représenté par un objet de la classe DFA contenant :

- un tableau à double entrée pour les transitions :

  tab[i][k] = j signifie qu'on a une transition de *i* vers *j* via le caractère *ASCII k*.
- un set d'entiers listant les état finaux (l'état initial est toujours 0).

c) Automate déterministe fini (DFA)

On veut maintenant se débarrasser des ε-transitions et ajouter des états pour transformer l'automate en automate déterministe. Dans celui-ci, il ne peut y avoir qu'un seul état initial mais il peut y avoir plusieurs états finaux.

On part de l'état initial 0. On le regroupe avec les états ayant les mêmes-propriétés par ε-transition (i.e. il n'est rien besoin de consommer pour y accéder). On regarde ensuite quels états sont accessibles par une transition (correspondant à un caractère). On réitère ce processus avec ces nouveaux états jusqu'à stabilité du système : aucune transition possible ou des transitions donnant seulement accès au même ensemble d'états.

A chaque étape, on vérifie si les états regroupés ensemble constituent l'état initial du nouvel automate (i.e. l'un d'eux est l'état 0) ou l'un de ses états finaux (i.e. un ensemble stable par transition). Ces regroupements d'états forment les états du nouvel automate.

Exemple

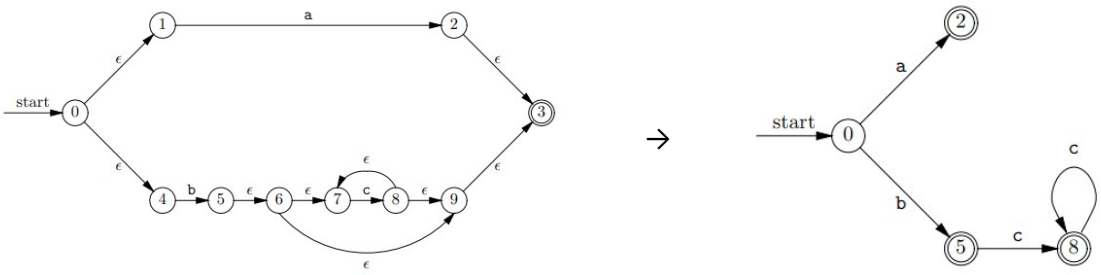

Figure 8: passage d'un NDFA à un DFA

d) Recherche dans le texte

Un texte matche une REGEX si et seulement si, en partant de l'état initial du DFA obtenu à partir de la REGEX, on arrive à l'état final en suivant les transitions correspondant aux lettres du texte.

Exemple : pour le DFA de la partie précédente et la chaîne de caractères « *babybcc* », les positions qui matchent sont 1, 2, 3 et 5.

Dans le code, on parcourt donc chaque ligne du texte puis chaque caractère de la ligne en vérifiant si celui-ci permet de passer à un nouvel état de l'automate et, le cas échéant, s'il s'agit d'un état final. Si l'on se trouve au $i^{ème}$ caractère de la ligne, il va, dans le pire cas, falloir parcourir la presque totalité des *k* caractères de la REGEX avant de se rendre compte de la non-correspondance du texte avec celle-ci et de devoir recommencer au caractère *i+1*. D'où la complexité en O(n*k), et même en $O(n^2)$ dans le cas de *wild card* '*.*' combinées avec '*'.

2. Recherche simple

Dans le cas de la recherche de mots-clés simples, i.e. correspondant à une suite de concaténation, on peut utiliser l'algorithme KMP (Knuth–Morris–Pratt). Celui-ci a en effet l'avantage d'être en temps linéaire. Ce cas correspond en fait à la grande majorité des recherches effectuées (sur Google par exemple). Cet algorithme utilise une table de retenues (*carryover*) qui permet de se décaler d'un certain nombre de caractères sur la ligne lorsqu'un caractère ne correspond pas à la REGEX.

Algorithme

Soit *F* la chaîne de caractères recherchée et *k* sa longueur. Soit *R* la table de retenues (de taille *k+1*).

$$d(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

Figure 9: distance de Jaccard

Étape 1 : *R[0] = -1* et *R[k] = 0*.

Pour tout autre *i*, *R[i]* est la longueur du plus long suffixe non trivial étant aussi préfixe dans *F[0, i-1]*.

Étape 2 : pour tout i, si :

- R[i] ≠ -1
- F[R[i]] = F[i] alors R[i] = -1
- R[R[i]] = -1

Étape 3 : pour tout i, si :

- R[i] ≠ -1
- F[R[i]] = F[i] alors R[i] = R[R[i]]
- R[R[i]] ≠ -1

Exemple :	*F =*	*[m*	*a*	*m*	*a*	*m*	*i*	*a]*
Étape 1	*R =*	*[-1*	*0*	*0*	*1*	*2*	*3*	*0*	*0]*
Étape 2	*R =*	*[-1*	*0*	*-1*	*1*	*-1*	*3*	*0*	*0]*
Étape 3	*R =*	*[-1*	*0*	*-1*	*0*	*-1*	*3*	*0*	*0]*

Une fois la table de retenues obtenue, il reste à parcourir chaque ligne du texte. Pour cela, on compare les caractères courants de la ligne et de *F :*

- s'ils correspondent, on passe aux suivants. Si on a atteint le bout de *F*, on a trouvé un match.
- s'ils ne correspondent pas et qu'il s'agit du $i^{ème}$ caractère de *F*, on se décale de *-R[i]* sur la ligne et on reprend au début de *F*. Cela évite souvent d'avoir à analyser tous les caractères d'une ligne plusieurs fois. En particulier, quand *R[i]=1*, cela permet de passer au caractère suivant de la ligne plutôt que de revenir en arrière.

IV. CLASSEMENT & SUGGESTIONS

1. Distance de Jaccard

a) Calcul

La distance de Jaccard est une grandeur permettant de mesurer la ressemblance entre deux ensembles. Elle est définie comme 1 moins le ratio entre :

- le cardinal de l'intersection de ces deux ensembles.
- le cardinal de leur union.

Cette distance est donc comprise entre 0 (tous les éléments sont communs aux deux ensembles) et 1 (aucun élément n'est commun). Dans le cas de la recherche dans une bibliothèque, la distance de Jaccard entre deux documents est déterminée en fonction du nombre de mots-clés qu'ils ont en commun. Cela permet de créer un graphe non-orienté dont :

- les sommets représentent les livres.
- les arêtes ont pour poids la distance de Jaccard entre les 2 sommets qu'elles relient.

b) Graphe de Jaccard

Dans le code, ce graphe est représenté par un objet de la classe *BookGraph* contenant une matrice d'adjacence. Chaque ligne de cette matrice représente un sommet et contient une liste de paires (livre, distance) correspondant aux arêtes et à leurs poids. La matrice ainsi que la liste sont en fait des *HashMap*, ce qui permet d'accéder aux données en temps constant.

```
Map<Integer, Map<Integer, Float>> adjacencyMatrix = new HashMap<>();
```

Figure 10: matrice d'adjacence en Java

Le peuplement de cette matrice se fait au fur et à mesure de l'ajout de nouveaux livres à la base de données (BDD). A tout instant, on connaît tous les mots-clés présents dans la BDD ainsi que la liste des livres qui les contiennent. Lors de l'ajout d'un livre, on détermine la liste des mots-clés de celui-ci. Pour déterminer l'intersection entre deux livres, il suffit alors de :

- parcourir la liste des mots-clés du nouveau livre,
- déterminer s'il est contenu par l'autre livre (déjà présent dans la BDD).

On calcule ensuite l'union entre ces deux livres à l'aide de la formule $|A \cup B| = |A| + |B| - |A \cap B|$ puis on calcule leur distance de Jaccard.

Enfin, on ajoute à la matrice d'adjacence :

- à chaque ligne existante, le nouveau livre avec la distance correspondante.
- une nouvelle ligne pour celui-ci et contenant la liste des autres livres associés à leurs distances.

```java
public void addBook(Integer newBook, Map<Integer, Float> jaccardDistance){
    adjacencyMatrix.putIfAbsent(newBook, jaccardDistance);
    for(Map.Entry<Integer, Map<Integer, Float>> entry : adjacencyMatrix.entrySet()){
        entry.getValue().putIfAbsent(newBook, jaccardDistance.get(entry.getKey()));
    }
```

Figure 11: ajout d'un nœud dans le graphe en Java

c) Fonctionnalités de suggestion

Deux fonctionnalités de suggestion ont été implémentées.

La première est liée à une recherche par mots-clés. Elle suggère la liste des sommets dans le graphe de Jaccard qui sont voisins (i.e. qui ont la plus petite distance moyenne) avec les trois documents textuels les plus pertinents qui contiennent les mots-clés.

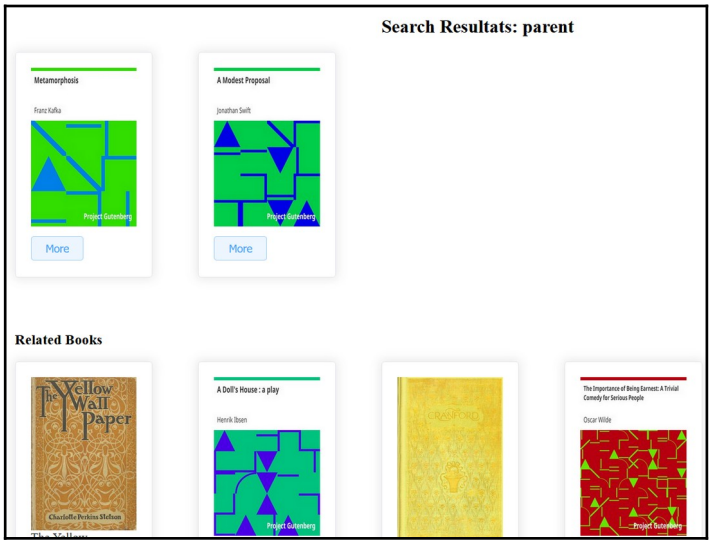

Figure 12: suggestions à partir d'une recherche

La seconde est liée à un livre et renvoie simplement les sommets les plus proches dans le graphe.

Figure 13: suggestions à partir d'un livre

2. Centralité

La centralité permet de mesurer l'importance relative de chaque nœud dans un graphe. Des exemples d'application sont l'identification :

- des personnes les plus influentes d'un réseau social
- des nœuds clés d'une infrastructure (e.g. internet ou un réseau urbain)
- des foyers d'infection de certaines maladies.

a) Centralité de proximité

La centralité de proximité (*closeness centrality*) permet de détecter les nœuds capables de diffuser efficacement de l'information dans un réseau. Elle est calculée en fonction de l'inverse de la distance moyenne à tous les autres nœuds. Une centralité de proximité élevée signifie donc une distance moyenne relativement faible avec les autres nœuds.

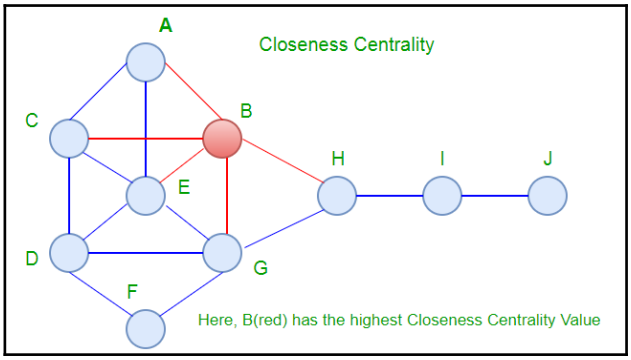

Figure 14: illustration de la centralité de proximité

Pour un nœud *u*, la centralité de proximité normalisée est : $$c_p(u) = \frac{nombre\ de\ noeuds}{\sum_{v \in graph, v \neq u} d(u,v)}$$

Il a été fait le choix dans notre projet d'utiliser la centralité de proximité comme critère de classement. En effet, celle-ci est souvent utilisée pour mesurer l'importance des mots dans un document. Lors d'une recherche par mots-clés, le classement se fait donc :

- d'abord en fonction du nombre de mots-clés présents dans les documents.
- puis, pour un nombre de mots-clés donnés, par centralité de proximité décroissante.

```java
public void computeCloseness(){
    int size = adjacencyMatrix.size();
    adjacencyMatrix.forEach((key, value) -> {
        float sum = 0F;
        for(float d : value.values()) { sum += d; }
        closenessCentrality.put(key, (size - 1) / sum);
    });
}
```

Figure 15: calcul de la centralité de proximité en Java

b) *PageRank*

Le critère *PageRank* (PR) permet de mesurer l'influence relative des nœuds dans un graphe. Un lien avec un nœud ayant un score élevé contribue plus au score qu'un lien avec un nœud ayant un score faible. Un score élevé indique donc la présence de connexions avec d'autres nœuds ayant un score élevé. Par exemple, avec Google, un site web cité par de nombreux sites web bien référencés aura un score élevé.

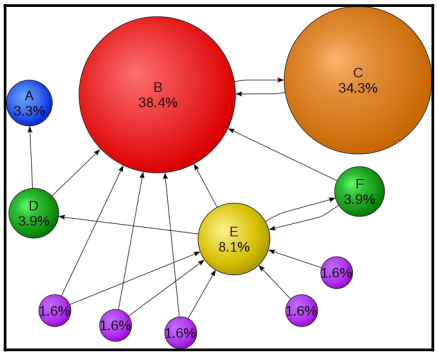

Figure 16: illustration de PageRank

Son calcul se fait avec la formule suivante : $PR(A)=(1-d)+d\times\frac{PR(T1)}{C(T1)}+\ldots+d\times\frac{PR(Tn)}{C(Tn)}$

où *A* est un élément en lien avec les éléments *T1* à *Tn*, *d* est un facteur d'amortissement (en général 0.85) et *C(T)* est le nombre de liens qu'a *T*.

c) Centralité intermédiaire

La centralité intermédiaire (*betweeness centrality*) permet de déterminer l'influence qu'a un nœud sur la circulation de l'information à l'intérieur du graphe. Elle peut servir à identifier les nœuds servant d'intermédiaire entre différentes parties du graphe.

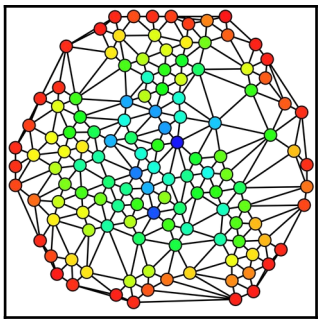
Figure 17: illustration de la centralité intermédiaire

Exemple : graphe ci-contre, nœuds colorés du rouge (centralité intermédiaire la plus faible) au bleu (la plus forte).

Son calcul se fait avec la formule suivante : $c_i(u)=\sum_{u\neq v\neq t}\frac{\#paths(u,s,t)}{\#paths(s,t)}$ où :

#paths(s, t) est le nombre de plus courts chemins entre *s* et *t*.

#paths(u, s, t) est le nombre de plus courts chemins entre *s* et *t* et passant par *u*.

V. TESTS DE PERFORMANCE

1. Démarrage de l'application

Des tests de performance ont été effectués sur le moteur de recherche en fonction du nombre de livres présents dans la BDD.

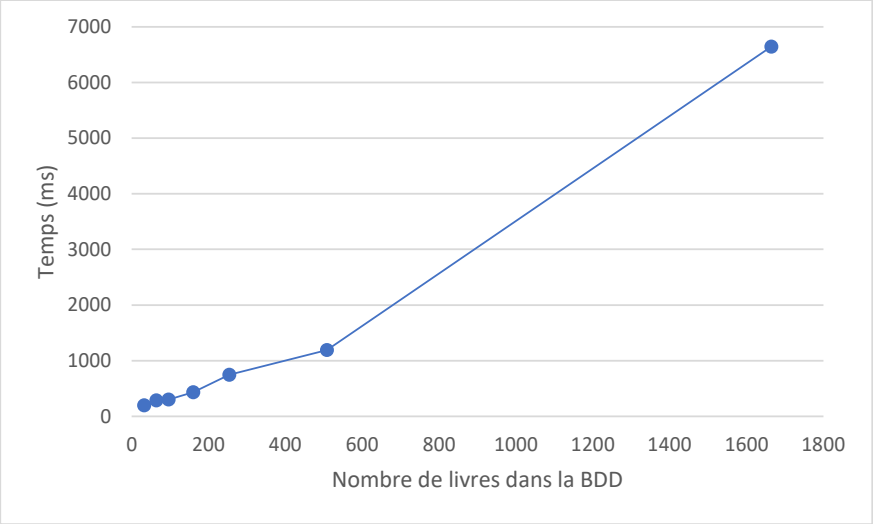

Figure 18: Temps de chargement de la BDD au démarage de l'application

Il semble qu'au démarrage de l'application, le temps pour charger les éléments nécessaires à son fonctionnement est linéaire par rapport au nombre de documents présents dans la BDD. De plus, ce temps est raisonnable : moins de 10 secondes pour plus de 1500 livres. L'application semble donc tout à fait viable.

2. Recherches

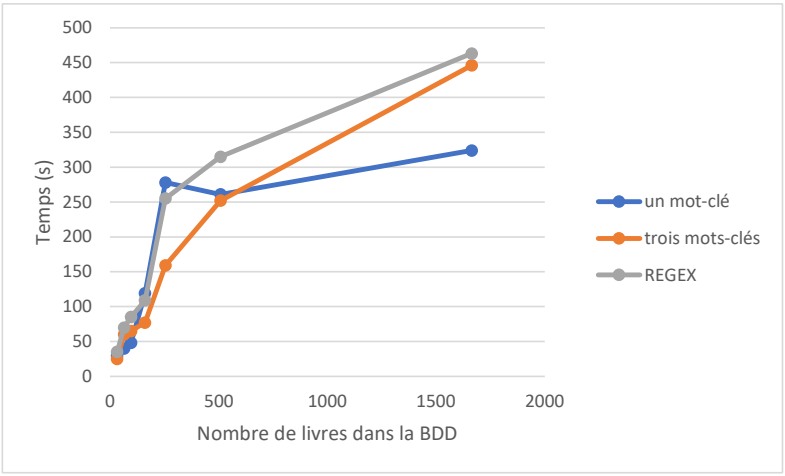

Figure 19: Temps de recherche

Les temps de recherche, eux, semblent être en temps logarithmiques par rapport au nombre de documents de la BDD. Comme précédemment, ces temps sont très raisonnables (moins d'une seconde pour plus de 1500 livres) et ce, que ce soit pour :

- la recherche d'un mot-clé.
- la recherche de 3 mots-clés.
- la recherche par REGEX (*g(o|a)v**).

VI. CONCLUSION

L'application web développée lors de ce projet a nécessité le développement de trois parties distinctes mais communicantes : la base de données, le *backend* et le *frontend*. Cette application permet d'accéder à une bibliothèque de documents textuels et regroupe les fonctionnalités suivantes :

- une recherche simple par mots-clés sur le contenu, le titre ou l'auteur d'un document.
- une recherche avancée par expressions régulières sur le contenu d'un document.
- un classement implicite des documents retournés par nombre de mots-clés présents et centralité de proximité décroissante.
- la suggestion d'autres documents lorsqu'un livre est sélectionné.
- la suggestion d'autres documents lors d'une recherche (en fonction des trois livres retournés les plus pertinents).

La recherche par REGEX a nécessité l'application de l'algorithme d'Aho-Ullman. Celle par mots-clés, l'algorithme KMP. Le classement implicite et les suggestions ont nécessité la création d'un graphe de Jaccard et le calcul de la centralité de proximité (*closeness centrality*) pour chaque nœud du graphe (i.e. pour chaque livre).

Les tests de performance ont quant-à-eux montré des complexités temporelles enviables (linéaires ou logarithmiques) qui permettent de penser que cette application est tout à fait viable.

Ce projet a mélangé des compétences de gestion des données, d'algorithmique et de développement *back* et *front*. Il nous a permis d'avoir une vision plus claire du fonctionnement d'une application web et en particulier d'un moteur de recherche. De ce point de vue, il a été très instructif.

D'autres fonctionnalités qu'il serait encore possible d'ajouter à l'application sont :

- le calcul pour chaque livre des deux autres indices vus en cours (*PageRank* et centralité intermédiaire) et la possibilité pour l'utilisateur de choisir celui qui lui convient le mieux.
- la suggestion de documents en fonction des recherches précédentes de l'utilisateur, voire en fonctions des recherches effectuées par les autres utilisateurs.
- améliorer encore les performances pour passer à des grandeurs de l'ordre de la dizaine ou centaine de millisecondes (au lieu de la seconde).