



Sorbonne Université
Faculté Des Sciences Et Ingénierie

CHOIX A. Moteur de recherche d'une bibliothèque

Dans le cadre de l'UE de DAAR

Auteurs :

Hongyu YAN
Liuyi CHEN

Professeur :

B.M. Bui-Xuan
Escriou Arthur
Hivert Guillaume

Master 2 STL 2022/2023

5 février 2023

Table des matières

1	Introduction	2
2	Architecture général	2
3	Couche client	2
3.1	L'interface de recherche	2
3.2	L'interface de la suggestion du livre	4
4	Couche data	4
4.1	La construction de la bibliothèque	4
4.2	Structure de données	4
4.2.1	Book	4
4.2.2	Keyword	5
4.2.3	Graph Ranking	5
5	Algorithmes principaux	5
5.1	Aho-Ullman	5
5.1.1	Présentation théorique de l'algorithme	5
5.1.2	Structure de données	6
5.1.3	Complexité	6
5.2	Knuth-Morris-Pratt (KMP)	6
5.2.1	Analyse et présentation théorique de l'algorithme	6
5.2.2	Complexité	7
5.3	GraphRanking	7
5.3.1	Jaccard	7
5.3.2	Closeness Centrality	8
6	Tests	9
6.1	Tests fonctionnels	9
6.1.1	Recherche par mots clefs	9
6.1.2	Recherche par expression régulière	9
6.1.3	Fonctionnalité de suggestion	9
6.2	Tests de performance	9
7	Conclusion	10

1 Introduction

Le but de ce projet est de créer un moteur de recherche pour une bibliothèque sous la forme d'une application web. Ce moteur de recherche aide l'utilisateur à trouver le bon document sur la base de mots-clés ou sur l'expressions régulières. Il suggère également des documents en fonction de l'historique de recherche de l'utilisateur. A l'aide du site <https://www.gutenberg.org/> qui stocke des dizaines de milliers de documents dans différents formats, nous récupérons des livres dans notre moteur de recherche. La bibliothèque contient donc au moins 1664 livres et chaque livre contient au moins 10^4 mots.

Cette application web Full Stack est construite avec ReactJS pour le Frontend et Java Spring Boot pour le Backend.

2 Architecture général

L'application peut être utilisée sur n'importe quel navigateur web. Elle suit une architecture MVC : les vues sont gérées par le client ReactJS, le contrôleur est dans le serveur Spring, les modèles sont des objets sérialisés. La partie client (frontend) qui a été construite avec ReactJS et se lance sur le port 3000. La partie serveur (backend) qui utilise une API REST développée avec Spring Boot en java 11 et se lance sur le port 8080. De plus, afin d'envoyer une requête HTTP du front-end au back-end, nous devons utiliser la bibliothèque Axios.

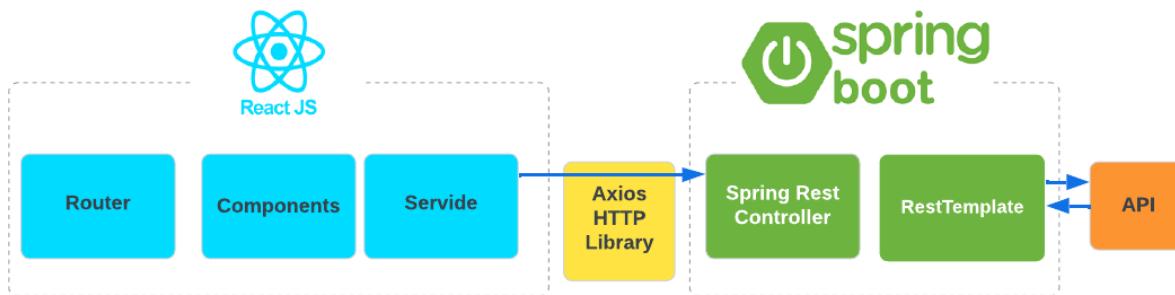


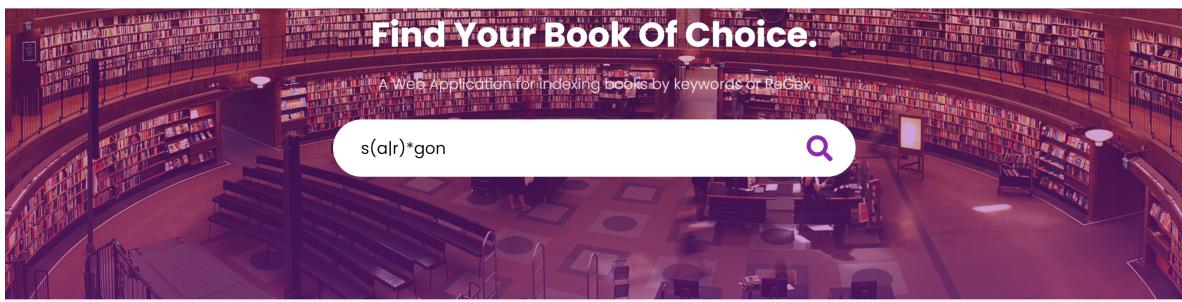
FIGURE 1 – Architecture de l'application

Nous implémentons les deux fonctions principales définies par le thème : une recherche de base par mots-clés basée sur le contenu du texte, et une recherche avancée qui peut utiliser des expressions régulières (Regex) pour la recherche de mots-clés. Dans la recherche comme une expression Regex. Nous réutilisons l'algorithme Aho-Ullman réalisé pour le premier projet DAAR. De plus, nous utilisons *Jaccard index* et *Closeness centrality* pour mettre en œuvre des recherches sous forme de caractères ainsi que des recommandations de livres similaires.

3 Couche client

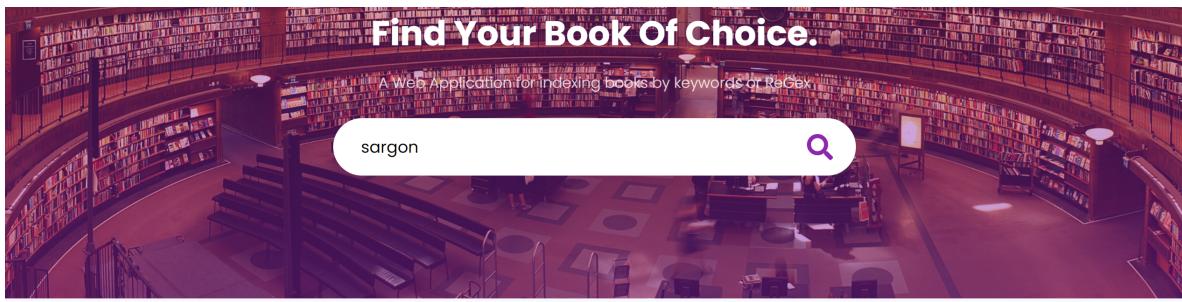
3.1 L'interface de recherche

Les résultats de recherche donnés par l'application sont les mêmes à travers des expressions régulières et des recherches de chaînes de caractère. Voici un exemple de recherche :



YOUR SEARCH RESULT

FIGURE 2 – Recherche par mots clefs



YOUR SEARCH RESULT

FIGURE 3 – Recherche par expression régulière

3.2 L'interface de la suggestion du livre

The screenshot shows a web page for a book titled "The Sketch-Book of Geoffrey Crayon". At the top left is a "Go Back" button. To the right is a decorative book cover image with the title "THE SKETCH BOOK" and a landscape illustration. To the right of the image is the book's title, "The Sketch-Book of Geoffrey Crayon", followed by author information ("Authors: Irving, Washington") and language ("Languages: en"). Below this is a section for "Subjects" listing various categories like "Americans -- England -- History -- 19th century, Catskill Mountains Region (N.Y.) -- Fiction, England -- Social life and customs -- 19th century, Fantasy fiction, American, Hudson River Valley (N.Y. and N.J.) -- Fiction, Irving, Washington, 1783-1859 -- Travel -- England". A blue link "link to the content" is also present. At the bottom, there is a section titled "SIMILAR BOOKS" featuring four smaller book covers: "Mosses from an Old Manse" by Nathaniel Hawthorne, "Twice-told tales" by Nathaniel Hawthorne, "Our Old Home: A Series of English Sketches" by Nathaniel Hawthorne, and "Knickerbocker's History of New York, Complete" by Washington Irving.

FIGURE 4 – Fonctionnalité de suggestion

4 Couche data

4.1 La construction de la bibliothèque

Le programme commencera à télécharger des livres du site Gutenberg. Les livres sont stockés sous forme de documents textuels. La taille minimum de la bibliothèque est 1664 livres et la taille minimum d'un livre est 10000 mots. Les livres téléchargés sont stockés dans le répertoire *books* avec leur identifiant comme nom de fichier.

Il existe une API web Gutendex qui permet de récupérer les metadata des documents présents sur Gutenberg. En Java, on utilise la classe RestTemplate du framework Spring qui permet de faire des requêtes HTTP pour récupérer les livres et parser les réponses JSON à l'aide d'une classe *bookList* qui représente une liste d'informations sur les livres dans la base de données obtenue en interrogeant l'API sur /books (par exemple gutendex.com/books).

4.2 Structure de données

Nous utilisons principalement la structure de données **Map** pour stocker des données, elle permet un accès aux données en O(1) ce qui est très performant pour des données de grande taille.

4.2.1 Book

On définit une classe sérialisable *Book* et une bibliothèque de type **HashMap<Integer, Book>**, qui stocke l'identifiant du livre téléchargé et son objet livre sérialisable correspondant.

Les livres sont donc tous conservés dans le système de fichiers où le serveur est lancé. On préfère 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.

4.2.2 Keyword

On définit une classe sérialisable *KeywordDictionary* pour les mots-clés. La classe *Keyword* (mot-clé) est composé d'une racine, des mots ayant cette racine et son nombre d'occurrences dans le texte.

```

1 public class KeywordDictionary implements Serializable {
2     // the word and its stem
3     private HashMap<String, String> word2stem;
4     // the frequency of keywords appearing in each document
5     private HashMap<String, HashMap<Integer, Double>> keywordTable;
6     // the keywords contained in the article and its frequency
7     private HashMap<Integer, HashMap<String, Double>> keywordBookTable;
8 }
```

- word2stem : stocker le mot et sa racine correspondante
- keywordTable : stocker le mot-clé et le nombre de fois qu'il apparaît dans différents livres
- keywordBookTable : stocker le texte et ses mots-clés et le nombre de fois qu'ils apparaissent

4.2.3 Graph Ranking

Nous utilisons trois structures pour représenter la relation entre les livres.

- *HashMap<Integer, HashMap<Integer, Double>> jaccardMatrice* :
Distance Jaccard d'un livre à d'autres livres en fonction de leurs tables d'index. La clé est l'identifiant du livre. La valeur est un *HashMap<Integer, Double>*. Son clé est aussi l'identifiant du livre et son valeur est la distance Jaccard de ces deux livres.
- *HashMap<Integer, ArrayList<Integer>> jaccardMatriceNeighbor* :
Les livres et leurs voisins. Deux livres sont voisins si leur *jaccardDistance* est inférieure à 0,5. La clé est l'identifiant du livre et la valeur est une liste de ses voisins.
- *LinkedHashMap<Integer, Double> closenessCentrality* :
Les livres et leurs closeness centrality dans la bibliothèque. Nous utilisons la structure de données **LinkedHashMap** pour stocker les livres par ordre décroissant selon leur centralité de proximité

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 réalisé pour le premier projet DAAR1. L'algorithme transforme d'abord l'expression en arbre de syntaxe abstraite (AST).

5.1.1 Présentation théorique de l'algorithme

Dans cette partie, nous avons 4 étapes principales.

- Etape 1 : D'abord, nous transformons le motif en un arbre syntaxique.

- Etape 2 : Puis en un automate fini non déterministe avec epsilon-transition selon la méthode Aho-Ullman.
- Etape 3 : Nous le transformons en un automate fini déterministe.
- Etape 4 : Nous entrons une expression régulière et utilisons les étapes ci-dessus pour obtenir un automate fini déterministe afin de vérifier si elle peut être reconnue par le texte entrant.

5.1.2 Structure de données

1. RegExTree : Il représente l'arbre syntaxe obtenu de la première étape. Ses enfants sont stockés dans un ArrayList.
2. NDFAutomaton : Un automaton représentant un automate fini non déterministe. Cette structure de données contient un tableau bidimensionnel des transitions de toutes les transitions de l'automate, à l'exception des epsilon transitions. Bien entendu, elle contient également un tableau de epsilon transitions.
3. DFAutomaton : Un automaton représentant un automate fini déterministe. Il a un tableau à deux dimensions pour stocker les transitions et un HashSet afin de stocker les états finaux.

5.1.3 Complexité

En supposant que la longueur du texte est n et que la longueur de l'expression régulière est k , la complexité est $O(n*k)$. Parce que chaque position dans le texte peut être appariée k fois.

5.2 Knuth-Morris-Pratt (KMP)

Pour faire correspondre une suite de concaténations à une chaîne, l'approche la plus générale consiste à rechercher la chaîne entière un par un. Si l'un des caractères de la chaîne ne correspond pas au caractère du motif, nous déplaçons tout le motif à l'endroit suivant. Néanmoins, la complexité de cette recherche est exponentielle.

5.2.1 Analyse et présentation théorique de l'algorithme

Lors que la chaîne recherchée est réduite à une suite de concaténations, nous pouvons appliquer KMP. Dans cet algorithme, nous construisons un tableau nommé *carryOver* pour faire avancer notre modèle. Le tableau *carryOver* nous indique comment nous pouvons déplacer notre motif. Cela peut nous éviter de déplacer le curseur vers l'arrière. Jusqu'à ce que tous les motifs soient trouvés ou que la chaîne atteigne la fin, l'algorithme s'arrête.

Nous définissons que le tableau *carryOver* a la longeur pârallèle celle du motif. Nous utilisons la sous-chaîne du motif pour construire le tableau *carryOver*. Pour chaque sous-chaîne, nous trouvons la longueur maximale qui rend le préfixe et le suffixe de la sous-chaîne identiques. Par exemple :

Supposons maintenant que la chaîne de texte S corresponde à la position i et que la chaîne de motif P corresponde à la position j . La correspondance est effectuée comme suit.

1. le caractère actuel est apparié avec succès (c'est-à-dire que $S[i] == P[j]$), les deux tels que $i++$ et $j++$ et continuer à appairer le caractère suivant.
2. la correspondance de caractères actuelle échoue (c'est-à-dire que $S[i] != P[j]$), alors on laisse i inchangé et $j = carryOver[j]$. Cela implique qu'en cas de non-concordance, la chaîne de motifs est décalée de $j - carryOver[j]$ vers la droite par rapport à la chaîne de texte S .

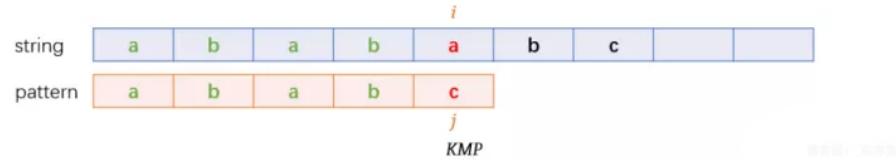


FIGURE 5 – La correspondance du caractère actuel a échoué

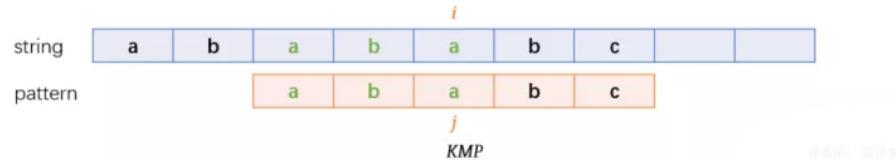


FIGURE 6 – Le pointeur de motif(pattern) j se déplace vers la position correcte

5.2.2 Complexité

Supposons que la longueur du texte soit m et que la longueur du motif soit n .

Dans notre fonction `kmpCarryOver`, nous utilisons une seule boucle `for` pour construire le tableau `carryOver`. Nous n'avons besoin de parcourir le motif qu'une seule fois, donc cela fait la complexité en temps n'est que de $O(n)$.

Pour la phase de recherche d'un motif dans une chaîne, la complexité est $O(m)$ où m est la longueur de la chaîne parce que le pointeur vers la chaîne de caractères ne revient jamais en arrière.

Par conséquent, la complexité temporelle totale devrait être $O(m+n)$, ce qui est linéaire.

5.3 GraphRanking

5.3.1 Jaccard

L'indice de Jaccard permet de déterminer si deux documents sont similaires ou non en se basant sur leurs contenus. L'algorithme calcule la similarité de Jaccard entre chaque paire de documents, qui est définie comme la taille de l'intersection des deux ensembles divisée par la taille de l'union des deux ensembles :

$$d(D1, D2) = \frac{\sum_{(w, k_1) \in D1 \wedge (w, k_2) \in D2} \max(k_1, k_2) - \min(k_1, k_2)}{\sum_{(w, k_1) \in D1 \wedge (w, k_2) \in D2} \max(k_1, k_2)}$$

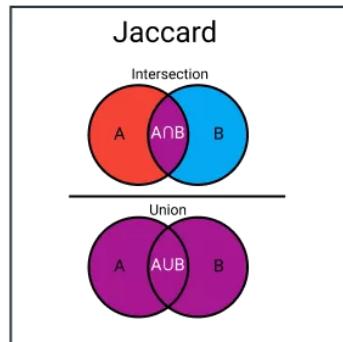


FIGURE 7 – Jaccard Distance

$d(D1, D2)$ représente la distance de Jaccard entre le livre D1 et le livre D2. $k1$ est le nombre d'occurrence du mot w dans le livre D1 et $k2$ est celui dans le livre D2.

On considère que si $d(D1, D2) < 0.6$, alors les livres D1 et D2 sont voisins. Nous stockons le livre et ses voisins dans un *HashMap<Integer, ArrayList<Integer>* qui sera utilisé pour la fonction de suggestion.

Le Jaccard Distance présente ses avantages et ses inconvénients. D'une part, le calcul est rapide et facile à exécuter. D'autre part, Il est fortement influencé par la taille des données. Les grands ensembles de données ont un impact important sur l'exposant, car ils peuvent augmenter considérablement les unions tout en gardant les intersections similaires. De plus, nous perdons une grande quantité d'information à cause du fait que les mots ne sont comptés qu'une seule fois, nous ne tenons pas compte du nombre d'occurrences.

5.3.2 Closeness Centrality

La centralité de proximité(Closeness Centrality) est une mesure de la centralité d'un nœud dans un graphe, basée sur sa distance moyenne à tous les autres nœuds. En d'autres termes, elle mesure la proximité d'un nœud par rapport à tous les autres nœuds du graphe. Appliquée 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 :

$$crank(v) = \frac{n - 1}{\sum_{u \neq v} d(u, v)}$$

$d(u, v)$ représente la distance dans le graphe de Jaccard entre les livres u et v. Et n représente le nombre total de livres dans la bibliothèque. Ainsi, plus un livre est proche des autres, plus son indice de Closeness Centrality est élevé.

On conserve les indices des livres dans une *LinkedHashMap<Integer, Double>*, 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.

Les noeuds présentant une centralité de proximité élevée ont des distances moyennes courtes par rapport à tous les autres nœuds, ce qui signifie qu'ils peuvent atteindre tous les autres nœuds rapidement. Un document présentant une centralité de proximité élevée dans le graphe des documents peut être recommandé à un utilisateur, car il est similaire à de nombreux autres documents et donc susceptible d'intéresser l'utilisateur.

Algorithm 1: ClosenessCentrality

```

Data: G : Un graphe G = (V, E)
Result: Un tableau de centralité de proximité pour chaque sommet dans G
init une liste des centralités
foreach  $u \in V$  do
    distanceTotal  $\leftarrow 0$ 
    foreach  $v \in V$  do
        if  $u \neq v$  then
            | distanceTotal += Jaccard( $u, v$ )
    enregistrer la distance totale pour  $u$ 
n  $\leftarrow$  nombreDeSommet(G)
foreach  $u \in V$  do
    crank( $u$ ) =  $(n-1) / \text{distance total}$ 
    ajouter crank( $u$ ) à la liste des centralités
return la liste des centralités

```

1. Pour chaque document, additionner les scores de similarité de Jaccard entre ce document et tous les autres documents du réseau.
2. La centralité de proximité d'un document est alors définie comme la réciproque de la somme des scores de similarité de Jaccard pour ce document.
3. Les scores de centralité de proximité est normalisés en divisant chaque score par le score maximal possible, qui est le nombre de documents dans le réseau moins un.

6 Tests

6.1 Tests fonctionnels

Pour la partie test, nous définissons des méthodes dans la classe **LibrarySearchEngineApplicationTests** pour tester et confirmer si la fonction est correctement implémentée à l'aide de JUnit5. Nous utilisons des documents textuels téléchargés et nos structures de données qui enregistrent des données comme des *testbeds*.

6.1.1 Recherche par mots clefs

Dans notre index, le mot clef "sargon" correspond aux résultats obtenus quand on effectue la recherche avec le mot clef "sargon" via la recherche basique de notre application.

6.1.2 Recherche par expression régulière

Semblable à la méthode de recherche précédente on peut effectuer la même recherche via la recherche avancée. Enfin, en modifiant la chaîne pour former une expression regex "s(alr)*gon", la recherche avancée renvoie toujours les mêmes résultats.

6.1.3 Fonctionnalité de suggestion

Nous avons bien les mêmes résultats en faisant le test avec par exemple l'identifiant 100 et via l'application web.

6.2 Tests de performance

Ci-dessous nous présentons la performance des deux stratégies sur petit texte et sur long texte. Ici, nous étudions le cas où le motif est réduit à une suite de concaténations. L'axe horizontal indique les différents motifs à chercher et l'axe vertical indique le temps en millisecondes.

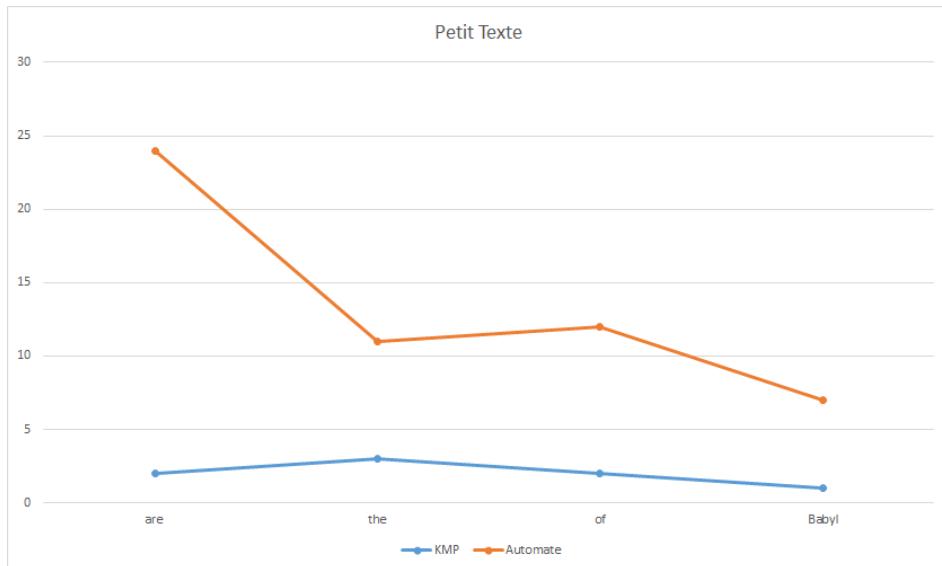


FIGURE 8 – Temps passé à rechercher différents motifs sur un petit texte selon les deux stratégies

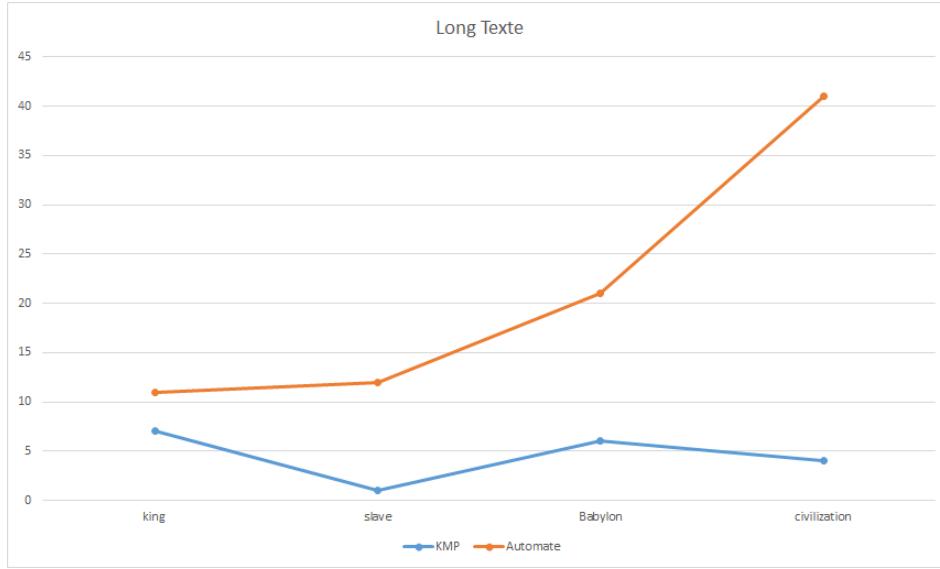


FIGURE 9 – Temps passé à rechercher différents motifs sur un long texte selon les deux stratégies

7 Conclusion

Ce projet nous a permis de découvrir comment un moteur de recherche fonctionne et de connaître les critères de centralité dans le graphe de Jaccard tels que closeness, betweenness, ou pagerank. Nous avons appris à comparer la diversité et la similarité des échantillons. Nous avons également eu l’occasion de travailler avec une importante couche de données en utilisant des fichiers .ser pour stocker les objets serialisés sur le serveur. Ils sont conçus pour des ressources spécifiques, ce qui nous permet de faire des recherches rapides et efficaces.

En termes de perspective d’amélioration de notre application, nous pouvons utiliser des séparateurs de mots (analyzer plus avancés et plus rapides tel que **elasticsearch**). On pourrait également implémenter différents critères d’ordonnancement tel que Betweeness qui propose des résultats plus pertinents que Closeness, ou encore Pagerank.