

**Projet 4 – projet final.**  
**CHOIX A. Moteur de recherche d'une  
bibliothèque.**

Étudiants :

- AZOUZ Yanis
- CHIBANI Hanae
- KABENE Said

Enseignant:

- Binh Xuan

## TABLE DES MATIÈRES

Introduction.....	2
Context du projet .....	2
Outils utilisés .....	3
Architecture .....	3
Implémentation des algorithmes .....	4
- Algorithme d'Aho Ullman.....	6
- Algorithme de Jaccard et critère de centralité.....	7
- Graphe de Jaccard .....	8
- Indexation des données .....	8
Conclusion .....	10

### ***I. Introduction***

Les moteurs de recherche (search engine) de nos jours sont des applications web qui permettent aux internautes de trouver rapidement des informations, des produits et des services en ligne.

Ces applications effectuent des recherches en fonction des expressions saisies par les internautes et affichent les résultats correspondants. Ainsi, les moteurs de recherche offrent aux internautes un moyen simple et rapide d'accéder à une grande quantité d'informations en ligne. Malgré la consommation d'un tel outil, un grand nombre de processus logiciels se produisent. Les moteurs de recherche tentent de trouver les réponses les plus fiables en les pondérant et en les classant selon divers critères de pertinence.

A travers ce projet, Nous avons mis au point une application fonctionnant tel un moteur de recherche qui offre la possibilité d'effectuer des recherches dans la bibliothèque Gutenberg. De ce fait comment construire un moteur de recherche efficace ?

Dans un premier temps, nous définirons le contexte et les objectifs à remplir, puis aborderons l'architecture et les outils, ainsi que la conception des algorithmes de l'application. Après cela, nous testerons l'application, analyserons les résultats obtenus et énumérerons les difficultés qui se sont présentées durant ce projet.

## ***II. Contexte du projet***

Durant ce projet, Nous avons été chargés de créer une application web qui permet aux utilisateurs de rechercher des livres dans la base de données de Gutenberg, celle-ci devra permettre à ses utilisateurs d'effectuer différentes recherches à savoir :

- **Une fonctionnalité explicite de “Recherche”** : L'application retournera la liste des documents textuels dans laquelle la chaîne de caractères saisie par l'utilisateur ("S") est incluse dans la table d'indexation lorsqu'il recherchera des livres avec un mot-clé.

- **La fonctionnalité "Recherche avancée"** permet aux utilisateurs de rechercher des livres en utilisant une expression régulière (RegEx) entrée par l'utilisateur.
- **Une fonctionnalité implicite de classement :** Après une recherche, l'application fournira une liste de documents triés en fonction d'un certain critère de pertinence, tels que le nombre d'occurrences du mot-clé dans le document, l'indice de centralité décroissante dans le graphe de Jaccard, etc.
- **Une fonctionnalité implicite de suggestion :** celle-ci retournera une liste des sommets les plus proches des deux ou trois documents textuels les plus pertinents contenant le mot-clé

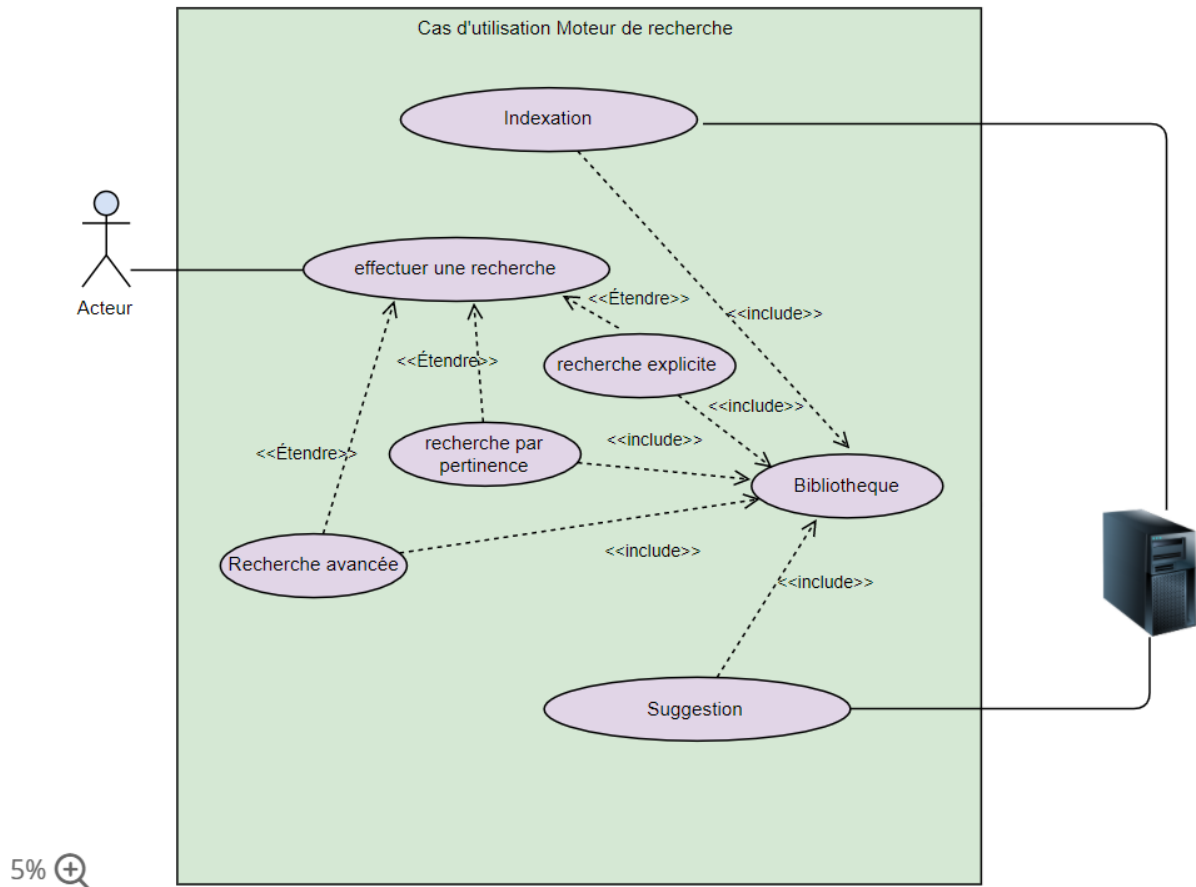
### ***III. Outils utilisés***

En raison de la variété de langages disponibles, il peut être difficile de décider quel outil utiliser. La qualité du résultat final dépend de ce choix, et après une analyse comparative, nous avons trouvé un langage qui satisfaisait pleinement nos exigences. De ce fait, nous avons choisi de travailler avec **DJANGO** et utiliser **ELASTICSEARCH** pour l'indexation des données.

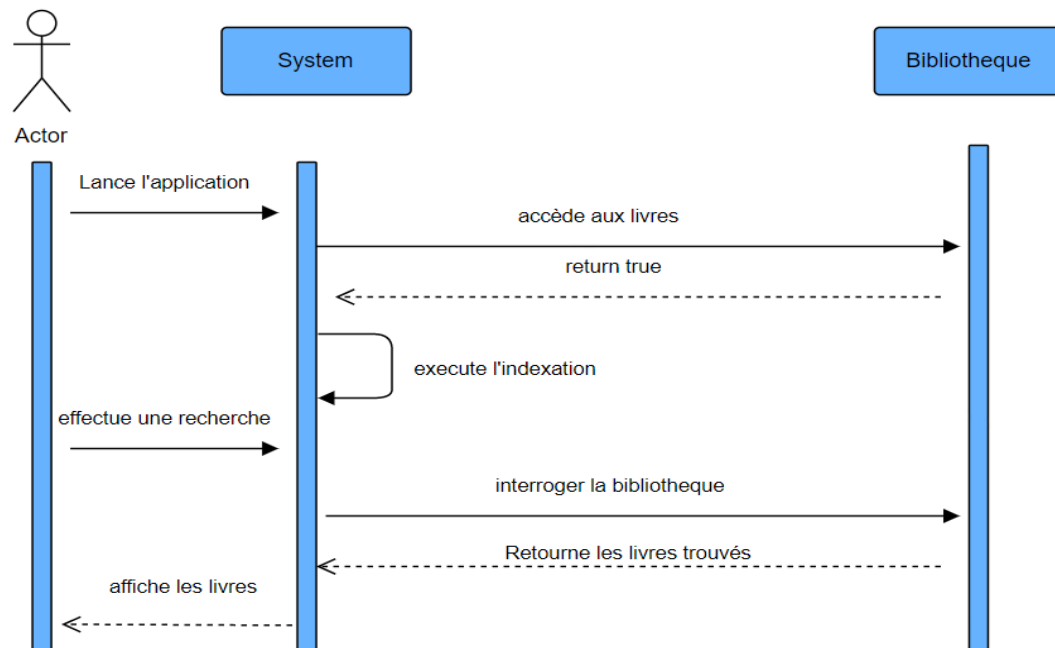


### ***IV. Architecture***

**Vue fonctionnelle :** Une analyse fonctionnelle a été effectuée à ce stade dans le but de déterminer une dimension correcte des caractéristique du produit, celle-ci consiste à distinguer les besoins fonctionnels qui vont décrire les besoins attendus, les acteurs qui interagissent avec notre produit et enfin une description entre ces acteurs et le produit à l'aide du diagramme de classe.



## Vue dynamique



## **V. Implémentation des algorithmes**

### **A. Algorithme d'Aho Ullman**

Dans cette section, nous allons décrire l'algorithme qui est utilisé par l'outil Regex pour afficher les résultats selon une expression régulière entrée par l'utilisateur, une fonctionnalité que nous avons implémentée dans notre moteur de recherche.

```
chaîne1 = ""           # Chaîne 1
chaîne2 = ""           # Chaîne 2
longueur_max = 0       # Longueur maximale de sous-chaîne
position_max = 0       # Position de la sous-chaîne maximale dans la chaîne 1

# Boucle principale
for i in range(len(chaîne1)):
    for j in range(len(chaîne2)):
        longueur = 0    # Longueur de la sous-chaîne courante
        position = i    # Position de la sous-chaîne courante dans la chaîne 1

        # Boucle de comparaison de caractères
        while (chaîne1[position] == chaîne2[j]):
            longueur += 1
            position += 1
            j += 1

        # Mise à jour de la sous-chaîne maximale
        if longueur > longueur_max:
            longueur_max = longueur
            position_max = i

# Affichage du résultat
print("La plus longue sous-chaîne commune est :")
print(chaîne1[position_max : position_max + longueur_max])
```

### **B. Algorithme de Jaccard et critère de centralité**

#### **1. Graphe de Jaccard**

La distance de Jaccard est une métrique employée pour évaluer la ressemblance entre deux échantillons (documents). Elle est calculée à l'aide de la formule suivante :

$$d(D_1, D_2) = \frac{\sum_{(w,k_1) \in D_1 \wedge (w,k_2) \in D_2} \max(k_1, k_2) - \min(k_1, k_2)}{\sum_{(w,k_1) \in D_1 \wedge (w,k_2) \in D_2} \max(k_1, k_2)}$$

La distance de Jaccard est calculée en divisant la somme des différences entre le nombre d'apparitions maximal et le nombre d'apparitions minimal d'un mot commun entre deux documents par la somme des nombres d'apparitions maximums d'un mot commun entre deux documents.

Ce ratio est compris entre 0 et 1 et est obtenu en comptant le nombre d'occurrences d'un mot donné dans l'autre document, puis en créant une liste avec les mots en commun entre les deux documents et leur nombre d'occurrences respectives, avant de calculer la distance selon la formule appropriée.

Nous avons mesuré la distance entre chaque paire de documents existante et créé un graphe géométrique où chaque document est un sommet. Les arêtes entre deux sommets sont établies si la distance de Jaccard entre les deux documents est inférieure à un seuil que nous avons défini après avoir examiné le graphe Jaccard.

Notre seuil est équilibré pour obtenir un résultat représentatif : un seuil trop élevé produira un graphe peu dense et peu de documents pertinents ; un seuil trop bas considérera trop de documents comme pertinents et la notion de pertinence sera donc réduite.

En outre, “closeness centrality” est une méthode pour évaluer à quel point un nœud est relié aux autres nœuds d'un réseau. On calcule cette valeur en tenant compte de la distance entre le nœud et tous les autres nœuds du réseau et en la normalisant pour prendre en compte la taille du réseau. Plus un nœud est proche des autres, plus sa centralité de proximité est grande. Cette métrique est calculée pour chaque sommet à l'aide de la formule de Jaccard. Un document qui n'est pas très différent des autres documents est alors considéré comme pertinent.

Pour améliorer la gestion et la lisibilité du graphe de Jaccard, nous avons utilisé les bibliothèques Pyvis et Networkx de python. Chaque nœud du graphe sera doté de l'ID du livre, du titre, de l'auteur et d'une liste de voisins qui permettra de proposer des documents. La visualisation du graphe nous a également aidé à déterminer une valeur appropriée pour le seuil de distance.

### **C. Construction du graphe de Jaccard**

Le graphe de Jaccard est construit juste après exécution de l'application, après lecture des documents dans le répertoire bibliothèque, nous utilisons la méthode **sent.tokenize()** de la bibliothèque **NLTK** pour extraire une liste de phrases de chacun des livres. ces informations sont utilisées dans Les fonctions **titreLivre(txt)**, **auteurLivre(txt)** et **idLivre(txt)** utilisent ces phrases pour extraire respectivement le titre, les auteurs et l'identifiant du document.

La fonction **distance(s1,s2)** prend des contenus textuels de deux documents comme argument. Elle décompose le texte en un ensemble de mots alphanumériques à l'aide de la méthode **word\_tokenize()** de la bibliothèque **nlk**. Ensuite, elle crée une liste qui contient les mots qui se trouvent dans les deux listes en excluant les doublons. Pour chaque terme de cette liste intersection, elle récupère le nombre de fois qu'il apparaît dans chaque document et calcule la distance entre les deux documents selon la formule indiquée.

La valeur retournée par cette fonction est utilisée dans la fonction **dic\_crank()** afin de créer un lien entre les documents. Elle retourne un dictionnaire qui comprend : la valeur **crank** pour chaque ID de document, une liste des voisins pour chaque ID de document, les auteurs de chaque document et les titres.

Ces données permettent d'afficher le titre et l'auteur des documents suggérés.

### **VI. Indexation des données**

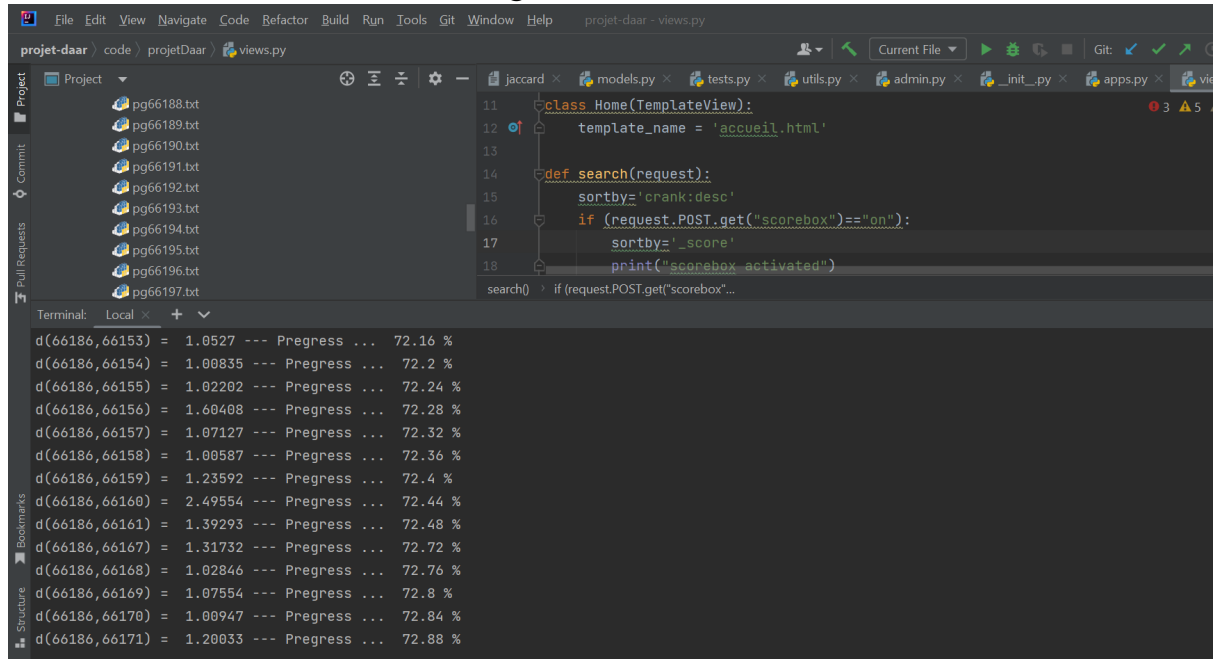
Pour l'indexation des données nous utilisons Elastic. Après que toutes les données soient indexées nous récupérons ces dernières dont notre moteur de recherche, et les avons mises sous un format json grâce à la fonction **livre\_json()**.

Celle-ci prend en argument les données des documents et crée un dictionnaire avec les champs **title**, **author**, **id**, **crank**, **content**, **voisin**, **titre\_voisin**, **auteur\_voisin**. Une fois le dictionnaire transformé sous format json à l'aide de la méthode **dumps()** du module **json**, nous avons indexé ces données dans un index nommé **livre** à l'aide de la méthode **index()** d'Elasticsearch.



## VII. Test

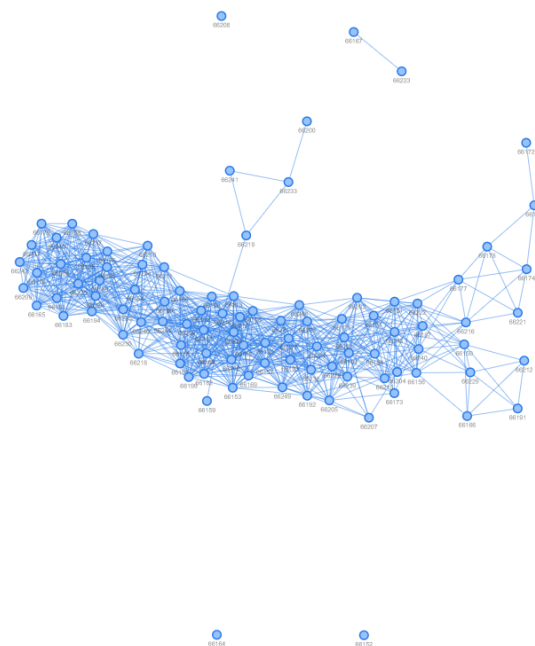
Après avoir récupéré le jeu de données, notre application commence par calculer les distances entre les livres (id1, id2), en définissant les voisins (avec un seuil de distance de 0.4) et en construisant le graphe de jaccard, comme illustré par les figures suivantes, où chaque nœud correspond à l'ID du livre dans la base de données Gutenberg.



```
class Home(TemplateView):
    template_name = 'accueil.html'

    def search(request):
        sortby='crank:desc'
        if (request.POST.get("scorebox")=="on"):
            sortby='_score'
            print("scorebox activated")
        search() if (request.POST.get("scorebox")...
```

```
d(66186,66153) = 1.0527 --- Progress ... 72.16 %
d(66186,66154) = 1.00835 --- Progress ... 72.2 %
d(66186,66155) = 1.02202 --- Progress ... 72.24 %
d(66186,66156) = 1.60408 --- Progress ... 72.28 %
d(66186,66157) = 1.07127 --- Progress ... 72.32 %
d(66186,66158) = 1.00587 --- Progress ... 72.36 %
d(66186,66159) = 1.23592 --- Progress ... 72.4 %
d(66186,66160) = 2.49554 --- Progress ... 72.44 %
d(66186,66161) = 1.39293 --- Progress ... 72.48 %
d(66186,66167) = 1.31732 --- Progress ... 72.72 %
d(66186,66168) = 1.02846 --- Progress ... 72.76 %
d(66186,66169) = 1.07554 --- Progress ... 72.8 %
d(66186,66170) = 1.00947 --- Progress ... 72.84 %
d(66186,66171) = 1.20033 --- Progress ... 72.88 %
```



Après la construction, le crank est calculé pour chaque livre et inclus dans un dictionnaire avec son titre, l'auteur et son identifiant. Ce dictionnaire est alors indexé dans le nœud Elasticsearch, mais seulement si l'index n'est pas encore existant, donc l'indexation ne sera pas effectuée à chaque lancement de l'application web car l'index est toujours présent, sauf si on le supprime.

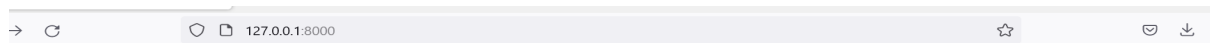
Une fois les données indexées, notre application web peut être utilisée en se connectant sur la 127.0.0.1:8000.

### ***VIII. Conclusion***

La réalisation de ce projet nous a permis de mettre en pratique les différents enseignements reçus durant le cours de **DAAR**. Ces derniers nous ont permis de répondre aux contraintes et aux différentes fonctionnalités demandées.

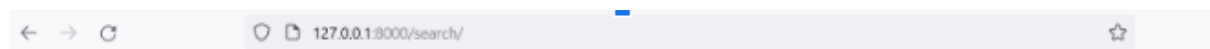
L'efficacité en termes de temps d'exécution et de traitement au moyen devra être réalisée. En outre, nous travaillons à modifier la façon dont nous codons afin de réduire la complexité des calculs nécessaires pour parcourir les fichiers textuels présents dans notre base de données.

Pour conclure, cette expérience nous a non seulement permis de monter en compétence en algorithmique mais aussi d'appréhender ce module et de mettre en pratique les connaissances acquises sur un projet réel.



☐ recherche avec RegEx  
☐ Recherche par nombre de match  
☐ Search par pertinence

Rechercher



Suggestion

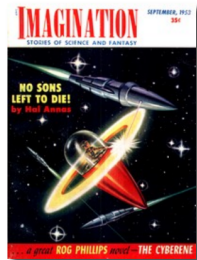
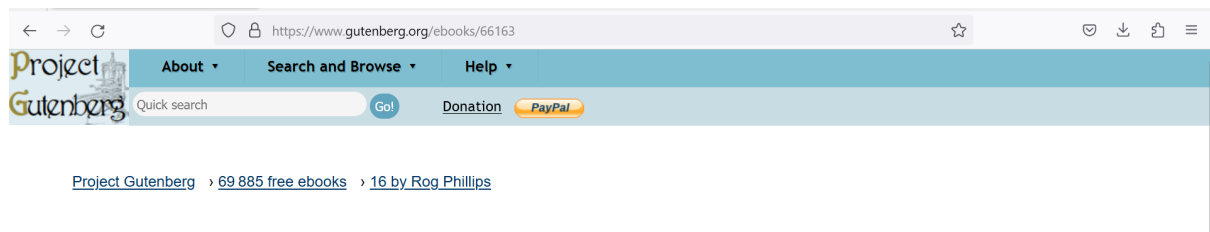
## Le mot "m[a-zA-Z0-9\_]ts" recherché se trouve dans :

- [None BY VariousEditor: George Bell](#)
- [None BY Melati van Java](#)
- [None BY Mark TwainTranslator: Jenő RákosiIllustrator: Tibor Pólya](#)
- [None BY Basil Thomaon](#)
- [None BY John Harington Gubbins](#)
- [None BY Dezső Kosztolányi](#)
- [None BY Aarne Mustasalo](#)
- [None BY Dorothy Kathleen Broster](#)
- [None BY Hilda Huntuvuori](#)

## Voir aussi >>>

- [None BY Winifred Eaton Reeve](#)
- [None BY Edward Stratemeyer](#)
- [None BY John T. McIntyre](#)
- [None BY Mack Reynolds](#)

La section voir plus correspond à la fonctionnalité de suggestion des 3 premiers livres du résultat qui sont classés par ordre de pertinence.



#### Download This eBook

Format	Size			
<a href="#">Read this book online: HTML5</a>	168 kB			
<a href="#">Read this book online: HTML (as submitted)</a>	169 kB			
<a href="#">EPUB3 (E-readers incl. Send-to-Kindle)</a>	501 kB			
<a href="#">EPUB (older E-readers)</a>	504 kB			
<a href="#">EPUB (no images)</a>	169 kB			

Lorsqu'on clique sur un des résultats retournés, celui ci nous redirige vers l'emplacement du livre dans le site de la bibliothèque de gutenberg.