**RAPPORT PROJET FINAL DAAR**

# CHOIX A : MOTEUR DE RECHERCHE D'UNE BIBLIOTHÈQUE

Auteurs :
Ariane ZHANG
Ngoc Anh NGUYEN

Enseignants :
Binh-Minh BUI-XUAN

2024-2025

SCIENCES SORBONNE UNIVERSITÉ

# Table des matières

# 1. Introduction

## 1. Contexte

Avec la croissance des bibliothèques numériques, accéder rapidement à des documents dans de vastes bases de données, comme le projet Gutenberg, est devenu un défi majeur. La recherche manuelle est inefficace, nécessitant des moteurs de recherche performants pour simplifier l'accès à l'information. Ces outils doivent non seulement permettre des recherches rapides, mais également proposer des fonctionnalités avancées telles que le classement et la suggestion des résultats.

## 2. Objectif

Ce projet vise à concevoir et développer un moteur de recherche pour une bibliothèque numérique contenant au moins 1 664 livres, chacun composé d'au moins 10 000 mots. L'objectif principal est de fournir une application web dotée de fonctionnalités de recherche et de classement intuitives et efficaces. Les principales fonctionnalités incluent :

- ➢ Une recherche simple par mots-clés
- ➢ Une recherche avancée par expressions régulières
- ➢ Un système de classement des résultats basé sur des critères comme les occurrences du mot clé ou la centralité dans un graphe.

Ce rapport détaillera les étapes de développement, les choix technologiques, les algorithmes utilisés et les résultats obtenus, tout en proposant des pistes d'amélioration pour des évolutions futures.

# 2. Technologie utilisée

Ce projet utilise Spring Boot pour la gestion du backend et Vue.js pour la construction du frontend. Voici une description détaillée de chaque technologie et son rôle dans l'application.

# 3. Données

La gestion des données constitue le socle du moteur de recherche. Cette section décrit la source des données utilisées, leur préparation et leur méthode de stockage.

## 1. Source des données

La bibliothèque exploitée dans ce projet s'appuie sur des bases de données textuelles volumineuses, comme le projet Gutenberg, connu pour héberger des milliers de livres numériques en libre accès. Pour faciliter l'accès à ces données et leurs métadonnées, nous utilisons l'API externe **Gutendex**, qui permet de récupérer facilement des informations essentielles sur les livres. Ces données incluent notamment le titre des ouvrages, les auteurs, les liens vers les fichiers texte et les images associées.

### 2. Préparation des données

Les données récupérées via l'API Gutendex (https://gutendex.com/books/) nécessitent un traitement préalable pour répondre aux exigences du projet. Dans un premier temps, nous extrayons les informations principales, à savoir le titre du livre, le nom de l'auteur, un lien vers une image associée, ainsi qu'une URL permettant d'accéder au contenu textuel complet du livre. Ces champs correspondent respectivement à `"title"`, `"authors/name"`, `"formats/image/jpeg"`, et `"formats/text/plain; charset=utf-8"` dans la structure de données fournie par l'API.

Ensuite, les données sont filtrées afin de respecter les critères imposés par le projet. Chaque livre doit contenir au minimum **10 000 mots** et la bibliothèque doit inclure au moins **1 664 livres**. Les ouvrages ne répondant pas à ces exigences sont exclus de la base.

Pour charger les données dans la bibliothèque, nous utilisons une requête POST intitulée `"http://localhost:8080/api/livres/charge"`, qui permet de récupérer les livres depuis l'API et de les stocker dans notre système. Cette requête, appelée une seule fois, nous a permis de charger un total de **1664 livres**, respectant ainsi les exigences minimales du projet.

### 3. Stockage des données

Une fois les données extraites et filtrées, elles sont organisées dans un fichier JSON pour simplifier leur gestion. Ce format a été choisi pour sa flexibilité et sa compatibilité avec les outils backend utilisés, comme Spring Boot.

Chaque entrée JSON contient les champs suivants : un identifiant unique (`id`), le titre du livre (`title`), le nom de l'auteur (`author`), un lien vers l'image associée (`image`), et un lien vers le texte intégral du livre (`txt`). Cette structure hiérarchique permet une manipulation efficace des données, notamment pour leur indexation et leur recherche.

L'organisation des données dans un fichier JSON garantit une gestion centralisée, tout en assurant une compatibilité avec les différentes étapes de traitement du moteur de recherche.

## 4. Recherche simple

Présentation de la recherche par mot-clé.

### 1. Explication de l'algorithme KMP

L'algorithme Knuth-Morris-Pratt (KMP) avec Carry Over est une variante améliorée du KMP classique. En plus de la construction du tableau des préfixes (LPS - Longest Prefix Suffix),

cette version introduit un mécanisme supplémentaire appelé "Carry Over" qui optimise davantage la gestion des correspondances partielles dans le motif.

**Étapes principales :**

1. **Création du tableau LPS** :

   Comme dans l'algorithme KMP classique, le tableau LPS est construit pour chaque position du motif. Il indique la longueur du plus long préfixe qui est également un suffixe du motif jusqu'à cette position.

2. **Génération du tableau Carry Over** :

   Le tableau Carry Over est dérivé du tableau LPS. Il est conçu pour gérer les cas où le motif contient des répétitions imbriquées complexes.

   En fonction des caractères du motif et des préfixes/suffixes identifiés, des ajustements sont effectués pour réduire encore les comparaisons inutiles. Par exemple :

   - Si un caractère dans le motif correspond à un préfixe mais ne possède pas de préfixe imbriqué valide, il est marqué comme “-1” dans le tableau Carry Over
   - Sinon, le tableau Carry Over exploite les relations imbriquées pour ajuster les index de manière plus efficace.

3. **Recherche dans le texte** :

   L'algorithme parcourt le texte et le motif en parallèle, mais utilise à la fois les tableaux LPS et Carry Over pour ajuster les positions du motif sans retour en arrière dans le texte.

## 2. Justification de l'utilisation du Carry Over

L'extension Carry Over renforce l'algorithme KMP en améliorant ses performances dans les cas où le motif contient des répétitions complexes. Cette approche est particulièrement avantageuse dans les bibliothèques numériques volumineuses comme celle utilisée dans ce projet. Les principaux bénéfices sont :

- **Optimisation des transitions** : Réduction du nombre d'étapes nécessaires pour ajuster les correspondances partielles.
- **Robustesse accrue** : Meilleure gestion des motifs répétitifs ou imbriqués.
- **Efficacité maintenue** : La complexité reste linéaire, soit .

## 3. Recherche de motif

Pseudo code

```
Fonction kmpRecherche(motif, texte):
    Si taille du motif == 0 ou taille du motif > taille du texte alors
        return false

    LTS = creerLTS(motif)
    carryover = creerCarryOver(motif, LTS)

    i = 0  // Index dans le texte
    j = 0  // Index dans le motif

    Tant que i < taille du texte faire :
        Si texte[i] == motif[j] alors
            i = i + 1
            j = j + 1

        Si j == taille du motif alors   // motif trouvé
            return true

        Sinon si i < taille du texte et texte[i] ≠  motif[j] alors // si les caractère ne correspond pas
            Si j > 0 alors
                j = carryover[j - 1] + 1 // on ajuste j en fonction du tableau de retenu
            Sinon
                i = i + 1  // Avancer dans le texte si j == 0
    return false
```

**Complexité**
La complexité est O(n), où n est la longueur du texte.

## 4. Mise en œuvre dans le backend

Pour la mise en œuvre du **backend**, plusieurs étapes ont été définies afin de garantir une recherche performante et structurée dans un grand volume de fichiers texte :

1. **Lecture des livres** : Les informations sur les livres, notamment leur titre et l'URL des fichiers texte associés, sont extraites d'un fichier JSON nommé **data.json**.
2. **Recherche du motif** : Pour chaque livre, la méthode `searchMotifInURLKMP` lit le contenu du fichier ligne par ligne. Elle filtre les caractères non ASCII ou spéciaux et applique l'algorithme **KMP (Knuth-Morris-Pratt)** afin de détecter efficacement les occurrences du motif dans le texte.
3. **Agrégation des résultats** : Les livres contenant le motif recherché sont organisés dans une structure de données (`HashMap` ou `List`), classés selon le nombre d'occurrences du motif détectées.
4. **Tri des résultats** : Une méthode dédiée permet de trier les livres par ordre décroissant du nombre d'occurrences du motif, facilitant ainsi un classement pertinent des résultats.
5. **Résultat final** : Deux méthodes principales exposent les résultats de la recherche :
    - `searchMotifInAllURLKMP` retourne les livres contenant le motif sans tri particulier.

- `searchMotifInAllURLKMPSortByOcurrences` retourne les résultats non triés et est utilisée ultérieurement pour intégrer un classement par **centralité de proximité**.

Deux **points d'entrée API REST** sont exposés pour interagir avec ce système :

- ➢ `GET http://localhost:8080/api/livres/search/{motif}` : Cette requête permet de rechercher un motif dans les fichiers associés aux livres, en les classant par occurrences décroissantes.
- ➢ `GET http://localhost:8080/api/livres/searchSortByClosenessCentrality/{motif}` : Cette requête offre une recherche avancée, basée sur un classement par **centralité de proximité**.

# 5. Recherche avancée

Description de la recherche par expressions régulières (RegEx).

## 1. Description de la recherche par expressions régulières (RegEx) avec automate

Les expressions régulières sont une méthode puissante pour rechercher et manipuler des motifs dans des textes. Elles permettent de définir des motifs complexes incluant des opérateurs comme la concaténation, l'alternative (|), l'étoile (*) ou encore des caractères universels (.). Ces motifs sont traduits en structures exploitables par un automate, qui assure une recherche efficace et systématique dans le texte.

Dans ce projet, la recherche avancée repose sur la conversion des motifs en arbres syntaxiques, puis en automates finis non déterministes (NFA) avant leur transformation en automates finis déterministes (DFA). Une étape supplémentaire permet de minimiser le DFA pour améliorer les performances.

## 2. Construction de l'automate pour traiter les RegEx

**Étape 1 : Construction de l'arbre syntaxique**

1. **Analyse du motif** :

   Chaque caractère et opérateur du motif est traduit en un nœud d'arbre syntaxique ("RegExTree"). Les opérateurs évalués en premier sont les parenthèses, suivis de l'étoile (*), de la concaténation (.) et enfin de l'alternative (|).

2. **Arbre hiérarchique** :

Par exemple, pour le motif “a|(bc)*”, l’arbre syntaxique représentera les relations entre les opérateurs et leurs opérandes.

**Étape 2 : Conversion de l’arbre syntaxique en NFA**

L'arbre syntaxique est transformé en un automate fini non déterministe (“NDFAutomaton”) selon les règles suivantes :

- **Feuilles de l’arbre** : Création d’un NFA simple avec une transition unique pour chaque caractère ou opérateur (ex. “a” ou “.”).
- **Concaténation** : L’état final du premier NFA est relié à l’état initial du second par une transition épsilon.
- **Alternative** : Deux chemins distincts sont créés depuis un nouvel état initial, chacun correspondant à une branche de l’arbre syntaxique.
- **Etoile** : Un état initial est créé avec des transitions épsilon pour permettre des itérations du motif.

**Étape 3 : Conversion du NFA en DFA**

L’algorithme de construction des sous-ensembles est utilisé pour convertir le NFA en DFA (“DFAutomaton”) :

- **Clôture épsilon** : Calcul des ensembles d’états accessibles par transitions épsilon.
- **Transitions explicites** : Chaque symbole du motif définit des transitions déterministes.
- **Unicité des états** : Chaque combinaison unique d’états NFA correspond à un état DFA.

**Étape 4 : Minimisation du DFA**

Une fois le DFA construit, il est minimisé pour optimiser son utilisation :

- **Partition des états** : Les états finaux et non finaux sont initialement regroupés.
- **Raffinement** : Les groupes sont subdivisés en fonction des transitions afin de conserver uniquement les états essentiels.

## 3. Recherche de motif

Pseudo code

```
Fonction search(motif, texte):
    Si l'état 0 est un état final alors return true

    Pour i = 0 à taille du texte - 1 faire :
        étatCourant = 0  // Initialisation de l'état courant à 0
        Pour j = i à taille du texte - 1 faire :
            étatCourant = tableTransition[étatCourant][texte[j]]
            Si étatCourant est un état final alors  return true
    return false
```

**Complexité**

- La première boucle itère sur chaque caractère du texte. Cette boucle s'exécute n  fois où est la taille du texte.
- La seconde boucle commence à partir de l'indice i jusqu'à n. Cela signifie que pour chaque valeur de i, le nombre d'itérations de i diminue linéairement.
- La somme des itérations pour les valeurs de i de 0 à n−1 est : $n(n+1)/2 = O(n^2)$

La complexité de l'algorithme de recherche est O(n2) en temps, ce qui le rend inefficace pour les textes longs.

## 4. Complexité et impact sur les performances

**Complexité** :

- **Construction du NFA** : La complexité dépend de la taille du motif, soit , où est la longueur du motif.
- **Conversion en DFA** : Cette étape peut avoir une complexité exponentielle dans le pire des cas, en fonction des états du NFA.
- **Minimisation du DFA** : L'algorithme de partitionnement a une complexité , où est le nombre d'états DFA.

**Impact sur les performances** :

- Bien que la construction soit coûteuse, l'utilisation du DFA offre une recherche en temps linéaire , où est la taille du texte.
- Une fois minimisé, le DFA consomme moins de mémoire et effectue moins de transitions, ce qui améliore l'efficacité.

## 5. Comparaison avec d'autres approches possibles

**Recherche naive** :

- Avantage : Implémentation simple.
- Inconvénient : Complexité , ce qui est inefficace pour des textes ou motifs longs.

**Algorithme Knuth-Morris-Pratt (KMP)** :

- Avantage : Recherche en pour des motifs simples.
- Inconvénient : Ne gère pas les motifs complexes comme ceux avec des alternatives ou des parenthèses imbriquées.

**Automates** :

- Avantage : Une fois construits, les DFA offrent des performances optimales pour des recherches répétées.
- Inconvénient : Coût initial élevé (conversion et minimisation).

## 6. Mise en œuvre dans le backend

Voici les principales étapes de l'implémentation :

1. **Lecture des livres** : Les informations sur les livres (tels que leur titre et l'URL de leurs fichiers texte) sont extraites d'un fichier JSON nommé `books.json`. Ces données servent de base pour accéder au contenu de chaque fichier.
2. **Traitement des expressions régulières** : La méthode `searchRegExpInFileAutomaton` commence par convertir l'expression régulière en un **arbre d'expression régulière** (RegExTree). Cet arbre est ensuite transformé en un automate non déterministe (NFA), qui est à son tour converti en un automate déterministe (DFA) et optimisé à l'aide de la minimisation des états du DFA.
3. **Recherche dans les fichiers** : Chaque fichier est lu ligne par ligne. Les caractères non-ASCII ou spéciaux sont remplacés par des points d'interrogation pour assurer la compatibilité avec l'automate. L'expression régulière est ensuite recherchée dans chaque ligne grâce à l'automate minimisé. Le nombre d'occurrences est enregistré.
4. **Agrégation des résultats** : Les livres contenant l'expression régulière sont organisés dans une structure de données (HashMap), classés par le nombre d'occurrences du motif.
5. **Tri des résultats** : Une méthode de tri (`sortByNumberOccurences`) permet d'organiser les livres par ordre décroissant du nombre d'occurrences, pour présenter les résultats les plus pertinents en priorité.
6. **Résultat final** :
    - `searchMotifInAllURLAutomaton` retourne les livres contenant l'expression régulière sans tri particulier et est utilisée ultérieurement pour intégrer un classement par **centralité de proximité**.
    - `searchMotifInAllURLAutomatonSortByOcurrences` retourne les livres triés par ordre décroissant du nombre d'occurrences.

Deux **points d'entrée API REST** sont exposés pour interagir avec ce système :

- ➢ `GET http://localhost:8080/api/livres/advancedSearch/{motif}` : Cette requête permet de rechercher une expression régulière dans les fichiers associés aux livres, en les classant par occurrences décroissantes.
- ➢ `GET http://localhost:8080/api/livres/advancedSearchSortByClosenessCentrality/{motif}` : Cette requête offre une recherche avancée, basée sur un classement par **centralité de proximité**.

# 6. Classement des recherches

Le classement des résultats de recherche est un élément clé pour garantir une expérience utilisateur satisfaisante. Ce projet intègre deux méthodes principales de classement : le classement par occurrence et le classement par centralité de proximité (closeness centrality). Ces approches permettent d'afficher les résultats de manière pertinente, en fonction des critères définis.

## 1. Classement par occurrence

Le classement par occurrence est une méthode simple et efficace pour organiser les résultats de recherche. Cette approche consiste à trier les livres en fonction du **nombre d'occurrences** du mot-clé recherché dans leur contenu dans l'ordre décroissant.

**Description du critère**

Chaque livre est évalué en comptant le nombre d'occurrences du mot-clé dans son texte. Les livres où le mot-clé apparaît fréquemment sont considérés comme plus pertinents et apparaissent en tête des résultats. Cette méthode repose sur une relation directe entre la fréquence du mot-clé et la pertinence du document.

**Mise en œuvre dans le backend**

- ➢ Lorsqu'une recherche simple ou avancée est effectuée, le backend utilise une structure de données de type **map** pour organiser les résultats :
  - **Clé :** le nombre d'occurrences du mot-clé dans un livre.
  - **Valeur :** la liste des livres ayant ce nombre d'occurrences.
- ➢ À partir de cette map, les résultats sont triés dans l'ordre décroissant en fonction de la clé. Une fois triés, ils sont envoyés au frontend pour affichage.

Cette méthode, particulièrement adaptée aux recherches simples, garantit des performances optimales tout en offrant une expérience utilisateur intuitive.

## 2. Classement par voisinage : closeness centrality

Le classement par **centralité de proximité** est une méthode avancée, basée sur l'analyse du **graphe de Jaccard**, qui représente les relations de similarité entre les livres en fonction des mots qu'ils partagent.

**Calcul dans le graphe de Jaccard**

Le graphe de Jaccard est construit en établissant des liens entre les livres ayant un chevauchement significatif dans leurs ensembles de mots-clés. La **similarité de Jaccard** entre deux livres D1 et D2 est calculée comme suit :

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

Une fois le graphe construit, chaque nœud est évalué pour calculer sa centralité de proximité.

**Explication de la centralité de proximité**

La **closeness centrality** mesure à quel point un nœud (ici, un livre) est proche des autres nœuds dans le graphe. Elle est définie comme l'inverse de la somme des distances les plus courtes entre un nœud donné et tous les autres. Un livre avec une centralité élevée est considéré comme bien connecté et central dans le réseau de similarités.

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

**Mise en œuvre dans le backend:**

- ➢ Dans un premier temps, tous les mots de chaque livre sont extraits et organisés dans une map où la clé est l'identifiant du livre et la valeur est la liste des mots correspondants. Les mots des livres sont filtrés en sorte qu'ils ne prennent pas en compte les mots dont la taille est inférieure ou égale à.
- ➢ À partir de cette map, un graphe de Jaccard est généré pour représenter les relations de similarité entre les livres. Ce graphe est ensuite sauvegardé dans des fichier txt, situé dans le répertoire graph, afin d'éviter de recalculer le graphe à chaque recherche utilisateur. Cette étape est effectuée une seule fois via une requête POST http://localhost:8080/api/livres/chargeGraph
- ➢ Lorsqu'une recherche avec classement par centralité de proximité est effectuée, le graphe sauvegardé est chargé depuis le répertoire graph avec seulement la liste des livres concernés.
- ➢ Les livres sont ensuite classés en fonction de leur centralité calculée, et les résultats sont organisés dans une map similaire à celle utilisée pour le classement par occurrence :
  - **Clé :** la valeur de la centralité.
  - **Valeur :** la liste des livres ayant cette valeur de centralité.
- ➢ Enfin, les résultats sont triés par ordre décroissant de centralité et renvoyés au frontend pour affichage.

Ces deux approches de classement, complémentaires, permettent de répondre à des besoins variés des utilisateurs : la simplicité et la rapidité pour les recherches basées sur les occurrences, et une analyse plus approfondie pour les recherches enrichies par les relations

sémantiques entre les livres. Ce système garantit ainsi des résultats pertinents et adaptés à chaque type de recherche.

# 7. API

Le backend dispose d'un Le contrôleur **`LivreController. Il`** expose plusieurs points d'entrée RESTful pour interagir avec les fonctionnalités de l'application, Voici les requêtes principales :

1. **Requête GET `/api/livres`** : Retourne une liste complète de tous les livres disponibles dans la bibliothèque.
2. **Requête GET `/api/livres/{id}`** : Permet de récupérer les détails d'un livre spécifique en fonction de son ID. Si le livre n'existe pas, une réponse "404 Not Found" est retournée.
3. **Requête GET `/api/livres/search/{motif}`** : Effectue une recherche dans les fichiers texte associés aux livres pour trouver un **motif textuel** spécifique en utilisant l'algorithme **KMP (Knuth-Morris-Pratt)**. Les résultats sont triés par nombre d'occurrences décroissant.
4. **Requête GET `/api/livres/advancedSearch/{regEx}`** : Recherche des livres contenant une **expression régulière** spécifique, en s'appuyant sur la conversion de l'expression en automates finis pour une recherche optimisée. Les résultats sont également triés par nombre d'occurrences décroissant.
5. **Requête GET `/api/livres/searchSortByClosenessCentrality/{motif}`** : Combine la recherche de motif avec le classement des résultats par **centralité de proximité**.
6. **Requête GET `/api/livres/advancedSearchSortByClosenessCentrality/{regEx}`** : Propose une recherche avancée par expression régulière, avec un classement des résultats basé sur la **centralité de proximité**.
7. **Requête POST `/api/livres/charge`** : Permet de charger un grand nombre de livres depuis une source externe.
8. **Requête POST `/api/livres/chargeGraph`** : Génère et construit un **graphe de similarité Jaccard** pour les livres de la bibliothèques.

Ces API offrent une large gamme de fonctionnalités, allant de la gestion de la bibliothèque à des recherches avancées basées sur des motifs textuels, des expressions régulières, ou encore des relations de proximité. Elles garantissent une flexibilité et une performance adaptées à des volumes importants de données textuelles.

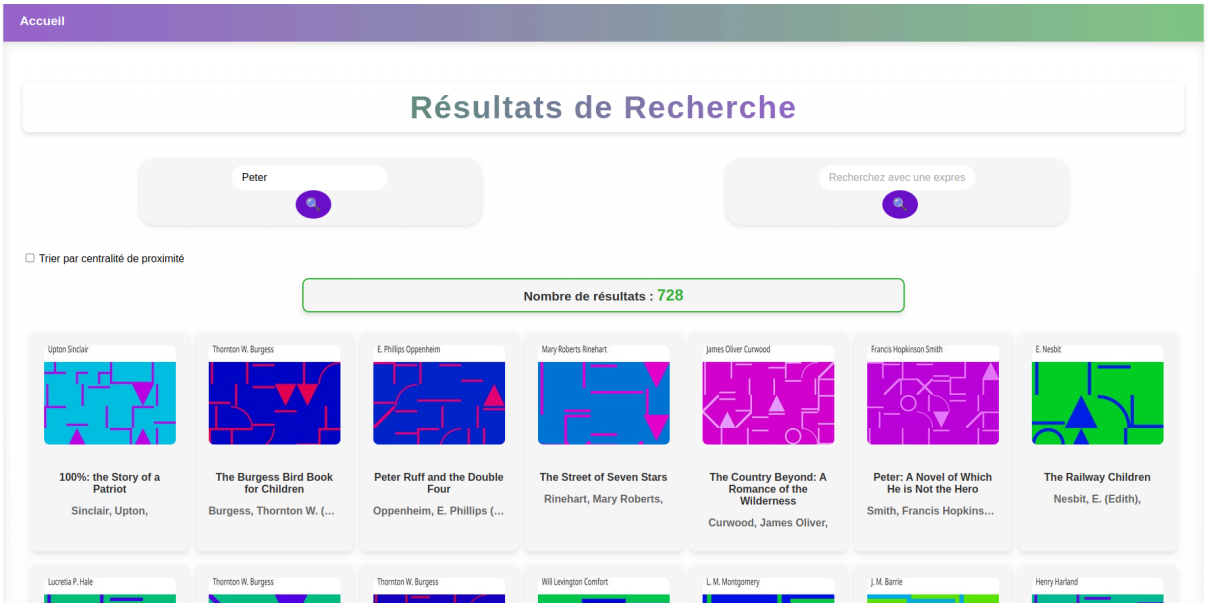
# 8. Frontend

La partie **frontend** du site web est conçue pour offrir une interface utilisateur **intuitive et fonctionnelle**, facilitant la recherche et la navigation dans la bibliothèque. Voici un aperçu des fonctionnalités et des caractéristiques principales de l'interface utilisateur :

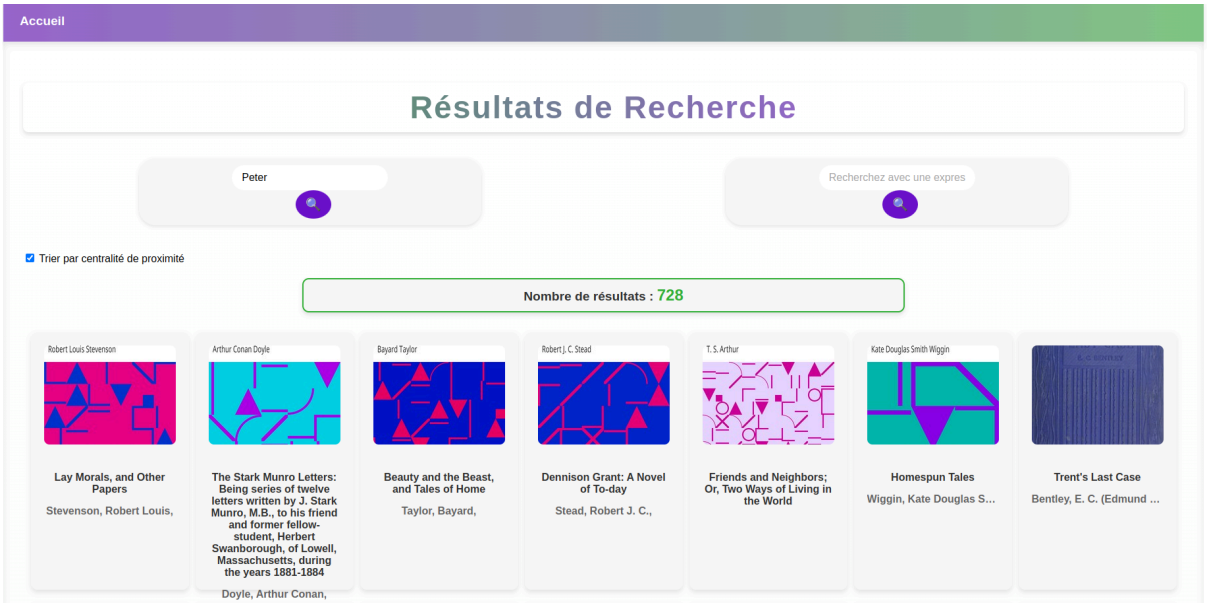
1. **Page d’accueil conviviale** :
   - ➢ La page d’accueil affiche **tous les livres disponibles dans la bibliothèque**, ce qui offre une vue d’ensemble complète dès le chargement du site.
   - ➢ Deux barres de recherche distinctes sont intégrées en haut de la page, chacune spécialisée dans un type de recherche :
     - **Barre de recherche gauche (KMP)** : Optimisée pour effectuer des recherches textuelles simples. L’utilisation de l’algorithme **Knuth-Morris-Pratt (KMP)** assure une recherche rapide et performante.
     - **Barre de recherche droite (Automate)** : Destinée aux recherches avancées, elle permet d'exploiter la flexibilité des **expressions régulières** grâce à l’utilisation d’automates.
2. **Classement des résultats** :
   - ➢ Les résultats sont, par défaut, triés par le **nombre d'occurrences** du mot-clé ou de l'expression régulière dans les livres.
   - ➢ Une option supplémentaire, intitulée **"Trier par centralité de proximité"**, permet de classer les livres en fonction de leur **score de centralité** dans le graphe de similarité (graphe de Jaccard).
3. **Retour visuel et interaction** :
   - ➢ Pendant le traitement d’une recherche, un message **"Recherche en cours"** est affiché pour signaler à l’utilisateur que la tâche est en cours d’exécution.
   - ➢ Une fois la recherche terminée, le **nombre total de livres trouvés** est affiché, offrant un retour immédiat sur les résultats obtenus.

# 9. Résultat

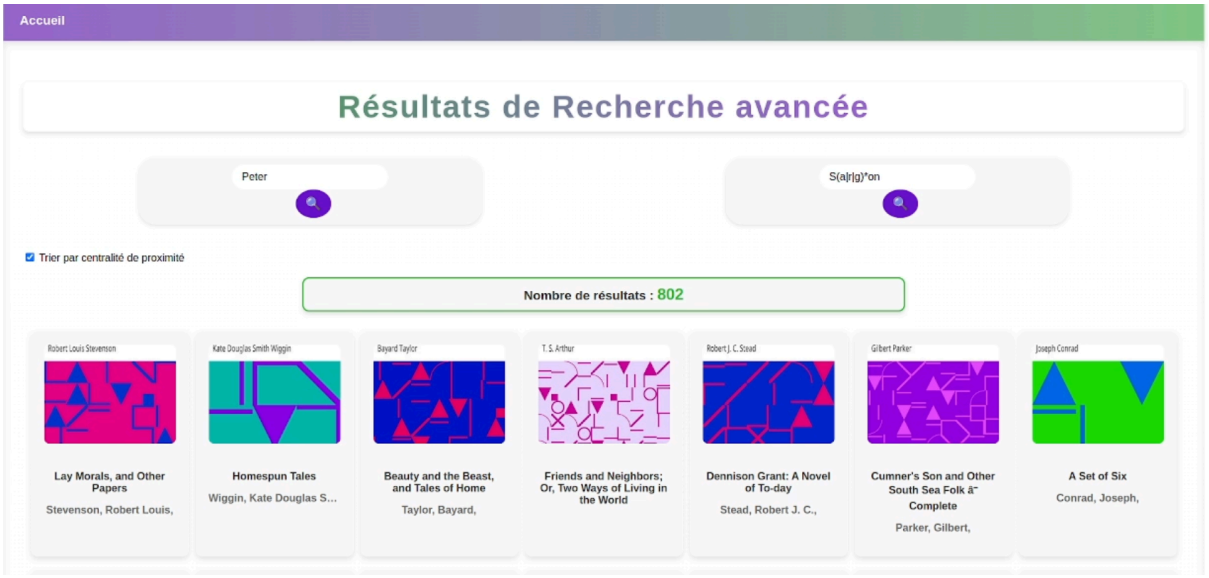

Page d’accueil




Recherche simple du mot clé “Peter” par classement décroissante de nombre d’occurence




Recherche simple du mot clé “Peter” par classement décroissante de centralité de proximité




Recherche avancé de l'expression "S(a|g|r)*on" par classement décroissante de nombre d'occurence

# 10. Conclusion

En conclusion, ce projet combine des algorithmes avancés et une interface intuitive pour offrir une solution efficace de recherche dans une bibliothèque numérique. L'utilisation de **KMP** pour les recherches simples, des automates pour les recherches avancées, et du classement par **centralité de proximité** garantit flexibilité et performance. L'interface conviviale, avec ses options de tri et de recherche, offre une expérience utilisateur fluide et optimisée pour explorer efficacement les données textuelles.