Sorbonne Université
Faculté des Sciences et Ingénierie
Département d'Informatique

Projet DAAR 2023

CHOIX A. Moteur de recherche d'une bibliothèque.

Réalisé Par :

ABED Nawres

CHOKOSSA HEMADEU Alphonse Daudet

SY Malick Laye

Professeur :

Bùi Xuân Bình Minh

I. Introduction et etat de l'art

Ce rapport présente le projet 4 "Moteur de recherche d'une bibliothèque réalisé" pour l'UE DAAR. Nous avons choisi d'utiliser le langage Kotlin pour coder ce projet, afin de continuer à exploiter les compétences que nous avions acquises lors de la réalisation du projet egrep. Notre objectif principal était de créer un moteur de recherche efficace pour une base de documents textuels, puis de le rendre accessible sur le web sous la forme d'une application (web /mobile).

Le moteur de recherche que nous avons développé est capable de retourner une liste de livres contenant un mot clé donné par l'utilisateur. En effet, il est en mesure de fournir une liste de suggestions de livres similaires aux meilleurs résultats. Nous avons utilisé la distance de Jaccard entre les différents livres pour classer les résultats et déterminer quels livres suggérer. Cette mesure de similarité nous a permis de placer les livres de notre bibliothèque dans un graphe géométrique. Les arêtes du graphe correspondent à la distance de Jaccard entre les livres. Nous pouvons ainsi identifier les livres les plus proches et les plus similaires à ceux que l'utilisateur recherche.

En utilisant ce graphe:

- Nous pouvons également définir un critère de centralité pour classer les livres en fonction de leur importance dans le graphe.
- Ce classement nous permet de trier la liste de résultats obtenue par l'utilisateur selon leur pertinence.
- Lorsque l'utilisateur recherche un livre, le moteur de recherche calcule la distance de Jaccard entre le livre recherché et tous les autres livres de la bibliothèque.
- Ensuite, il trouve les livres les plus proches et les plus similaires, en fonction des critères de distance et de closeness centrality.

  Le moteur de recherche retourne ensuite la liste des résultats triés par ordre de pertinence.

II.Structure de données

La structure de données que nous avons choisie pour stocker nos index est les HashMaps. Ces structures nous permettent un accès rapide aux données en O(1), même pour des volumes de données très importants.

Tous nos index sont stockés dans des objets sérialisables, qui sont conservés sur le système de fichiers du serveur. Nous avons opté pour cette solution plutôt qu'une base de données traditionnelle, car nous n'avons pas besoin de modifier ces données une fois qu'elles ont été construites. Les données sont pré-construites et chargées une seule fois au démarrage du programme. Une fois le serveur lancé, les données sont accessibles dans l'environnement d'exécution.

Les avantages de cette méthode sont multiples. Premièrement, la taille relativement faible des index nous permet de les charger très rapidement, ce qui est primordial pour un accès fluide aux données. De plus, le stockage dans des objets sérialisables permet une manipulation facile des données et leur transmission sur le réseau, sans que cela n'altère la structure des données.

Cependant, il y a aussi des inconvénients à cette méthode. En effet, les temps de chargement peuvent être influencés par la taille de l'index et le nombre d'entrées qu'il contient. Nous avons donc remarqué que certains identifiants ne correspondaient à aucun document ou que certains documents ne contenaient aucune métadonnée. Cela a pour conséquence que la taille de certains index, comme celui du champ "books", diffère de la taille attendue.

Malgré ces quelques inconvénients, nous sommes convaincus que nos index restent très performants et permettent une recherche rapide et précise dans notre bibliothèque numérique.

III. Construction de la couche Data

1. Bibliothèque et Api

Les documents que nous utilisons sont des livres hébergés sur le site Gutenberg, qui sont identifiés par des numéros allant de 1 à 1664.

Gutenberg est une initiative visant à numériser et à archiver des œuvres littéraires dans le domaine public, c'est-à-dire des livres dont les droits d'auteur ont expiré.

- Pour construire notre bibliothèque, nous avons utilisé Gutendex, une API web qui permet de récupérer les métadonnées des documents présents sur Gutenberg.
- Cette API nous évite d'avoir à envoyer des requêtes directement à Gutenberg et de parser les réponses qui sont des pages HTML.
- Nous utilisons la bibliothèque com.google.gson pour parser les réponses JSON de Gutendex et récupérer les informations dont nous avons besoin, notamment le lien permettant de récupérer le contenu du livre.

- Il convient de noter que les textes peuvent être disponibles sous différents formats, en fonction de l'encodage du document (UTF-8, ASCII).
- Pour récupérer les textes, nous devons donc chercher l'objet "format" dans la réponse JSON, puis les différents éléments "text/plain" possibles. Cela nous donne le lien vers le contenu brut du document hébergé sur Gutenberg, que nous pouvons ensuite récupérer avec une requête HTTP. Ci-dessous, un extrait de code illustrant la façon dont nous parse les réponses JSON.

2. Indexage

Nous avons utilisé des livres réels écrits en anglais pour tester et déployer notre projet. Nous nous sommes procuré plus de 1600 livres, mis à disposition par le projet Gutenberg. À chaque recherche, nous devrons parcourir ces livres, il est donc judicieux de chercher à optimiser les fichiers que nous allons lire.

Afin de calculer nos critères pour ordonner les livres et suggérer des livres à l'utilisateur,nous avons utilisé une structure de hashMap où chaque élément représente un livre, relié aux autres par sa distance de Jaccard avec eux, c'est donc un graphe ou les noeud c'est les éléments et les arcs, la distance qui les relie entre eux.

Pour l'indexage, puisque notre moteur de recherche doit uniquement renvoyer le nom des livres où le mot est détecté, et non pas les phrases où ce mot est détecté, nous

Nous n'avons pas besoin de faire des recherches dans le texte original. Nous avons donc utilisé des fichiers d'index, qui contiennent chaque mot d'un livre et son nombre d'occurrences. Un tel fichier peut avoir une taille O(log(n)) comparée à la taille n du texte original, même si ce cas n'arrivera jamais pour un texte écrit en anglais usuel, puisqu'il nécessite que chaque mot ait le même nombre d'occurrences dans le texte. Néanmoins, nous avons remarqué une nette amélioration du temps de calcul pour effectuer une recherche sur ces index.

Pour créer des index, nous avons d'abord transformé les textes en graph afin de compter les occurrences de tous les mots de manière efficace en une seule lecture du texte, même si la construction du graph est coûteuse. Nous ignorons plus de 80 mots les plus fréquents de la langue anglaise, car nous supposons qu'ils ne sont pas intéressants pour l'utilisateur (savoir que tous les 2000 livres contiennent le mot "is" n'est pas trop intéressant).

Afin de faciliter les recherches tout en évitant de perdre des informations importantes, nous avons mis en place plusieurs index qui permettent un accès rapide

aux données requises en fonction du type de recherche, sans devoir parcourir tous les documents de la bibliothèque.

Actuellement, nous utilisons cinq index distincts :

- L'index "keywords" , qui gère les mots-clés des documents,
- L'index "titles" , utilisé pour les recherches par titre,
- L'index "authors" , utilisé pour les recherches par auteur,
- L'index "books", utilisé pour renvoyer les données d'affichage au client.
- L'index en mémoire "recommandation" , c'est un graphe utilisé pour les suggestion
- L'index "closeness", qui permet d'accéder à la valeur de la centralité de proximité de chaque document.

IV. Création d'index : les méthodes techniques

Pour créer nos index, nous avons utilisé Gutendex, une bibliothèque pour la construction d'index de documents.

- Les métadonnées des ouvrages de la bibliothèque de Gutenberg ont été utilisées pour créer les index des titres, des auteurs, des livres et des mots clés. Les index des titres et des auteurs sont des structures de type HashMap<String, Integer> et HashMap<String, List<Integer>> respectivement. Les clés de la carte des titres sont les titres des livres, et les valeurs sont les identifiants associés à chaque titre. Les clés de la carte des auteurs sont les noms des auteurs, et les valeurs sont les identifiants des livres qu'ils ont écrits. Les index des titres et des auteurs sont enregistrés sous forme d'objets sérialisés : "titles.ser" et "authors.ser".
- L'index des livres est une structure de type HashMap<Integer, Book>. Il contient les métadonnées des ouvrages, telles que le titre ou la couverture, et est principalement utilisé pour l'affichage côté client.
- Pour construire l'index des mots clés, une approche plus complexe a été utilisée. L'index des mots clés est représenté sous forme de HashMap<String, List<Integer>>. Les clés de la map sont les mots-clés des documents, et les valeurs sont les identifiants des livres où ils apparaissent. Pour extraire les mots clés de chaque livre, nous avons utilisé Apache Lucene, une bibliothèque permettant de créer une liste de mots-clés pour un texte tout en conservant leur fréquence d'apparition. Nous avons conservé les 50 mots-clés les plus fréquents pour chaque texte, afin de ne garder que les plus informatifs sur le contenu.

- Les mots clés ont également été récupérés en conservant leur racine principale. Nous avons utilisé une liste de mots usuels qui ne doivent pas être pris en compte.
- Les mots ayant moins de 30 occurrences dans la bibliothèque ont été conservés dans l'index. L'index des mots clés est enregistré en tant qu'objet sérialisé sous le nom de "keywords.ser" dans le système de fichiers.

V. Algorithme principaux

— Algorithme du projet Daar 1 (Kmp et Aho-Ullman)

Pour rappeler ce qui à été réalisé:

- Lorsque le motif est une expression régulière, l'algorithme utilisé est celui d'Aho-Ullman. Dans un premier temps, l'expression régulière est transformée en un arbre de syntaxe abstraite (AST) qui est ensuite converti en automate fini non-déterministe avec epsilon-transition (NFA) selon la méthode d'Aho-Ullman. Ensuite, l'automate est transformé en un automate déterministe qui est minimisé. Enfin, cet automate est utilisé pour tester si un suffixe d'une ligne du texte en entrée est reconnaissable par l'automate. Si un match est trouvé, la ligne est ajoutée à la liste des lignes qui correspondent à l'expression régulière.
- Si le motif recherché est une suite de concaténations, l'algorithme utilisé est le Knuth Morris-Pratt (KMP). Ce dernier est plus rapide que l'algorithme d'Aho-Ullman, mais ne peut pas être utilisé pour des expressions régulières. L'algorithme KMP est utilisé dans le cadre de la recherche avancée si l'expression recherchée ne contient pas de regex.
- Lorsque le motif est une expression régulière, l'algorithme est divisé en cinq étapes : parsing de la RegEx, création de l'automate fini non-déterministe avec epsilon-transition, conversion en automate déterministe, minimisation de l'automate et utilisation de l'automate pour tester si un suffixe d'une ligne du texte en entrée est reconnaissable par cet automate.
- Pour trouver les matchs de l'expression régulière dans un texte, chaque lettre de chaque mot de chaque ligne du texte doit être parcourue. Si un match est trouvé, la ligne correspondante est ajoutée à la liste des lignes qui correspondent à l'expression régulière. Pour gagner en performance, dès qu'un match est trouvé, la ligne est ajoutée à la liste résultante sans la parcourir en entier.

— **Jaccard**

Le coefficient de Jaccard est une méthode pour évaluer la similitude entre deux textes en se basant sur leur contenu.

Supposons que A et B soient deux textes distincts, le coefficient peut être calculé à l'aide de la formule suivante:

$$J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}.$$

Le nombre de mots en commun dans les deux textes est divisé par le nombre total de mots dans les deux textes. Pour stocker les mots des textes, nous utilisons des Set<String> car ils ne contiennent pas de doublons, ce qui est important pour le calcul du coefficient de Jaccard, qui nécessite le nombre de mots distincts. Nous utilisons la méthode intersection du package com.google.common.collect.Sets dans kotlin pour créer un Set contenant les éléments en commun de deux Sets donnés en paramètre. Cela nous permet d'obtenir le nombre de mots en commun dans les deux textes. Enfin, nous cherchons à obtenir la distance de Jaccard, qui est simplement calculée selon la formule 1 - J(A, B).

Nous cherchons ensuite à construire un graphe de Jaccard. Ce graphe contient les distances de Jaccard entre chaque paire de livres. C'est donc une matrice symétrique de taille 1664 x 1664, ce qui nécessite 2 768 896 calculs de distance. Bien que le calcul soit efficace pour deux textes simples, même de taille importante, la création de la matrice devient extrêmement chronophage. Nous notons également que la taille du texte influence considérablement le temps d'exécution du calcul de similarité. Ce temps peut varier de 20 secondes pour une ligne de la matrice (1664 appels) pour un fichier de 200Ko à 20 minutes pour un fichier de 44 Mo. Il est donc crucial de tenir compte de la taille des textes lors de l'utilisation du coefficient de Jaccard pour évaluer la similitude.

— **Closeness centrality**

Lorsqu'on étudie un graphe, on peut utiliser un outil appelé "indice de Closeness Centrality" pour voir à quelle distance se trouve chaque nœud par rapport aux autres. En utilisant ce calcul sur un graphe de Jaccard, on peut savoir si un nœud contient beaucoup d'informations importantes sur les autres nœuds. Plus un nœud est proche des autres, plus son indice de Closeness Centrality est élevé. Pour le calculer, on utilise une formule qui prend en compte la distance entre les nœuds dans le graphe de Jaccard (qu'on appelle graphe de suggestion). Une fois que l'on a fait la recherche et obtenu les résultats, on peut ranger les livres dans un ordre particulier en utilisant une "HashMap" qui conserve les indices de Closeness Centrality de chaque livre. La Closeness est calculer avec la formule suivant:

$$Closeness(x) = \frac{1}{\sum_{i=1}^{1664} d(y,x)}$$

oú d(y,x) représente la distance dans le graphe de Jaccard entre les livres x et y. Plus deux livres se ressemblent, plus leur distance est petite. Par conséquent, plus la somme des distances est petite, plus l'indice de closeness est élevé, donc pertinent.

Cependant, bien que le calcul soit simple, il y a certains problèmes avec cet outil. Par exemple, la liste des résultats est toujours la même et ne tient pas compte du contexte de la recherche. De plus, si les livres sont écrits dans des langues différentes, cela peut affecter les résultats de l'algorithme. Dans notre bibliothèque, les livres en anglais sont beaucoup plus nombreux que les livres dans d'autres langues, donc ils auront des distances plus courtes entre eux. Cela signifie que ces livres seront toujours mis en avant, même s'ils ne sont pas pertinents pour la recherche en cours. Par exemple, si vous recherchez des livres de Jules Verne en français, les résultats en anglais apparaîtront avant les résultats en français.

VI. Différentes fonctionnalités

- Recherche par mots-clés basique et multiple :

Nous utilisons un seul service pour la recherche simple et la recherche multiple. Quand

Il s'agit de cette dernière, on le gère côté backend en splittant par un espace.

Pour vérifier le bon fonctionnement de la recherche par mots-clés, nous utilisons directement les ressources de notre index.

Lorsque nous effectuons une recherche avec ce mot-clé via la recherche basique de notre application, nous obtenons bien ces deux documents.

- Recherche par auteur :

La recherche par auteur est testée de manière intuitive. Si nous saisissons la chaîne "spinoza", le moteur de recherche nous renvoie tous les livres de spinoza présents dans

notre index.

Serveur distant

`http://bookle.voltevieira.fr:8088/searchbyauthor?author=spinoza`

- Recherche par regex

Nous pouvons également effectuer cette même recherche via la recherche avancée, qui nous renvoie également les deux ouvrages. Enfin, en utilisant une expression régulière pour former la chaîne de recherche "`tuba*`", la recherche avancée renvoie toujours les mêmes deux livres qui ont été renvoyés par recherche par mot clé tuba et d'autres commençant par ce mot.

Endpoint serveur distant

`http://localhost:8088/searchbyregex?regex=tuba*`

Endpoint serveur

`http://bookle.voltevieira.fr:8088/searchbyregex?regex=tuba*`

- Recherche par titre :

  La recherche par titre fonctionne de manière similaire à la recherche par mots-clés. Par exemple, lorsque nous saisissons la chaîne “herland”, le moteur de recherche nous renvoie le livre en question.

  EndPoint serveur distant:

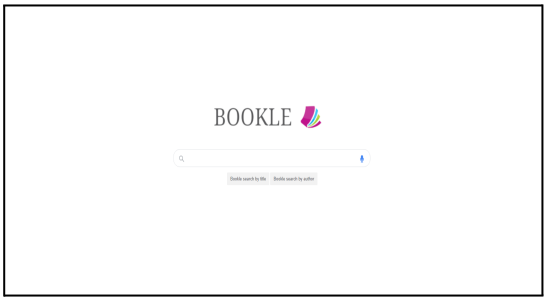
  ```
  http://bookle.voltevieira.fr:8088/searchbytitle?title=Herland
  ```

  EndPoint serveur local

  ```
  http://localhost:8088/searchbytitle/?word=herland
  ```

VII. Choix des technologies

- Kotlin : langage de programmation qu’on avait utilisé pour le projet DAAR 1 et qui est beaucoup plus performant que java.
- Apache Lucene : est une bibliothèque offrant de puissantes fonctionnalités d'indexation et de recherche, ainsi que des fonctions de vérification orthographique, de surbrillance des résultats et d'analyse/tokenisation avancées
- fichier extension ser permettant de faciliter la sérialisation des données.
- React js : Utilisé côté front end, en important le reactDom. On a produit exactement comme Google.



VIII. Mesure de performances

Pour les performances on exprime ici que pour la recherche basique et la recherche avancé puisque qu’on avait déjà comparé dans egrep KMP ET recherche par regex.

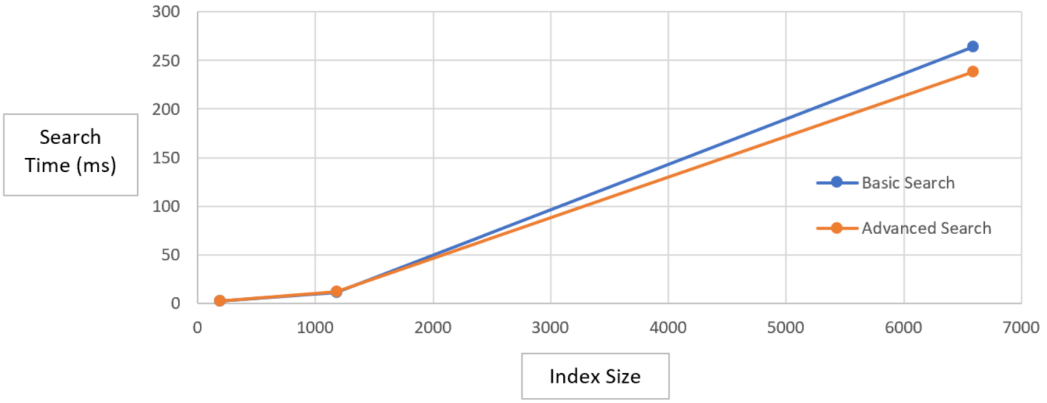
Les tests ont été réalisés en comparant deux méthodes de recherche différentes:

la méthode de recherche basique qui n'utilise pas d'expressions régulières et

la méthode de recherche avancée qui utilise la méthode Knuth-Morris-Pratt (KMP) pour traiter les expressions régulières.

La méthode de recherche basique est basée sur la méthode contains() de Kotlin qui compare simplement la chaîne entrée par l'utilisateur avec l'index de la bibliothèque de mots clés. Cette méthode est moins optimale que la méthode KMP, qui est utilisée dans la recherche avancée.

Les tests ont été réalisés en utilisant la chaîne de caractères "spinoza" comme mot clé car elle apparaît le plus souvent dans l'index. On constate que plus la chaîne de caractères a d'occurrences dans l'index, plus la recherche est longue. Par exemple, pour la chaîne "spinoza", qui apparaît 1560 fois dans l'index, la recherche a pris environ 264 millisecondes, alors que pour la chaîne "spinoza*", qui n'apparaît que dans deux documents, la recherche a pris seulement 2 millisecondes.
Les résultats des tests montrent que pour des petits ensembles de mots clés, la différence de performance entre la méthode de recherche basique et la méthode de recherche avancée est négligeable, avec des temps de recherche pratiquement instantanés. Par exemple, pour 193 mots clés, la recherche a pris en moyenne 2 millisecondes pour les deux méthodes. Pour 1179 mots clés, la différence de temps entre les deux méthodes était encore très faible, avec environ 12 millisecondes pour les deux.



Cependant, pour des ensembles de données plus volumineux, la différence de performance entre les deux méthodes est plus importante. Par exemple, pour un index contenant 6587 mots clés, la recherche basique a pris environ 160 millisecondes, alors que la recherche avancée avec KMP a pris environ 180 millisecondes, soit une différence de 8%. Ces résultats indiquent que la méthode de recherche avancée est plus efficace pour les ensembles de données plus volumineux et contenant un grand nombre d'occurrences de mots clés.

IX. Conclusion et perspective

- Ce projet a été une opportunité passionnante de découvrir le fonctionnement d'un moteur de recherche et de comprendre les différents critères d'ordonnancement qui permettent d'organiser les résultats de recherche. Nous avons également travaillé avec une couche de données importante composée de différents index, dont la création a été une partie chronophage mais essentielle pour garantir des recherches rapides et efficaces.
- Dans une optique d'amélioration de notre application, nous pourrions envisager plusieurs perspectives. Tout d'abord, nous pourrions développer un système de suggestion plus sophistiqué, basé sur le graphe de Jaccard et prenant en compte les recherches précédentes de l'utilisateur. Cela permettrait de proposer des résultats encore plus pertinents et de guider l'utilisateur dans sa recherche.

- Par ailleurs, il serait intéressant d'implémenter différents critères d'ordonnancement tels que Betweenness ou Pagerank, qui offrent des résultats plus pertinents que Closeness. Cela permettrait d'augmenter la précision des résultats de recherche et de mieux répondre aux besoins des utilisateurs.
- Il serait également possible d'explorer d'autres méthodes de recherche et de traitement des données, telles que l'apprentissage automatique ou l'analyse sémantique, afin d'élargir les capacités de notre moteur de recherche et d'offrir une expérience encore plus personnalisée et adaptée aux besoins des utilisateurs.

En somme, ce projet a été une expérience riche en enseignements et en perspectives d'amélioration, et nous sommes convaincus que les améliorations proposées permettront d'offrir une expérience de recherche encore plus satisfaisante aux utilisateurs de notre application.