M2 Informatique - STL

Projet Final Cloud A & B

Kelly SENG : Choix B
Ning GUO : Choix A

9 février 2020

Table des matières

1 Introduction

Dans ce projet nous avons réalisé une application web avec un moteur de recherche dans une base de données qui permet à l'utilisateur d'accéder plus rapidement à un document textuel par recherche de mot-clef. Nous nous intéressons à la caractérisation de grandes collections de documents (en utilisant les liens entre ces derniers) afin de faciliter leur manipulation et leur exploitation.

Dans un premier temps, étant donné un ensemble de documents textuels et un seuil appelé seuil de similarité θ nous définissons le graphe de Jaccard de ces documents comme le graphe géométrique de seuil θ et de fonction de distance Jaccard qui permet de calculer les différentes distances entre chaque document, décrit dans la section 3.1. Nous abordons la problématique du calcul de centralité dans les graphes de documents. Nous décrivons les principaux algorithmes de calcul de centralité demandés *closeness centrality* (section 4.1), *betweenness centrality* (section 4.2).

2 Recherche & Recherche avancée

2.1 Radix Tree (Recherche Simple)

Pour réaliser une recherche, nous avons implémenté un radix tree commun pour l'ensemble des livres comprenant ainsi tous les mots présents dans le contenu des livres, avec leurs occurrences.

Un arbre radix est une structure de données compacte permettant de représenter un ensemble de mots adaptée pour la recherche. Il est obtenu à partir d'un arbre préfixe en fusionnant chaque nœud n'ayant qu'un seul fils avec celui-ci.

Pour construire l'arbre radix, on appliquera les deux étapes suivantes :

1. Construction de l'index

   L'index est représenté par une Map<String,List<Coordonnees> >. Une clé est un mot composé de caractères alphabétiques, et les valeurs correspondent à tous les occurrences de ce mot dans un livre. Une occurrence est représentée par un triplet comprenant l'id du livre, la ligne et la colonne où se trouve le mot. L'index se trouve en mémoire.

Pour le construire, nous parcourons le contenu de chaque livre mot par mot. Nous vérifions ensuite si le mot est présent. S'il n'est pas présent, alors nous ajoutons ce mot en tant que clé dans l'index avec sa coordonnée contenue dans une liste comme valeur. Sinon, on ajoute la coordonnée dans la liste déjà existante.

2. Construction de l'arbre radix à partir de l'index obtenu à l'étape 1

   On insère chaque paire <String,List<Coordonnees>> de l'index dans l'arbre radix. L'insertion dans l'arbre radix tree débute à la racine de celui-ci et se poursuit par la recherche du plus grand préfixe commun existant parmi ces fils jusqu'à atteindre une feuille de l'arbre. Si ce préfixe existe et est identique au mot, alors nous continuons le parcours avec le suffixe du mot de manière récursive. Sinon, s'il n'existe pas alors nous créons, un nouveau fils avec pour préfixe de navigation l'intégralité du mot. Dans le cas où le préfixe existe mais le mot à être inséré ne correspond pas à tout le préfixe, alors nous créons un fils ayant le préfixe en commun remplaçant ainsi le fils précédent. Puis nous ajoutons pour le nouveau noeud créé, des fils ayant pour nom le suffixe du mot et le suffixe des fils précédents.

   A la racine de chaque feuille se trouve une liste contenant les coordonnées des mots. Lorsque nous insérons un même mot pour différents livres, nous ajoutons à la liste les nouvelles valeurs.

On suppose que les mots sont de longueur k et que le nombre de mot est n. Les opérations de recherche et d'insertion d'un mot ont des complexités en O(k). Par conséquent, la complexité temporelle de ces opérations ne dépend pas du nombre de données contenues par l'arbre radix.

2.2 Expression régulière (Recherche avancée)

On peut réaliser une recherche "avancée" de livre par RegEx sur notre moteur de recherche. A la suite d'une entrée texte RegEx de l'utilisateur, l'application retourne une liste de tous les documents textuels dont la table de l'indexage contient une chaîne de caractères S qui vérifie l'expression régulière RegEx. Pour réaliser cette fonctionnalité, nous allons d'abord transformer la RegEx en automate.

Théorème : Tout langage regulier est accepte par un automate fini deterministe. –> Possibilité de transformer une expression régulière en automate.

Dans cette partie, nous détaillerons la construction d'un automate fini déterministe minimal à partir d'expressions régulières.

On appliquera les 5 étapes suivants :

1. Transformation du motif en un arbre de syntaxe (voir automate.RegEx.java)
2. Transformation de l'arbre de syntaxe en automate fini non déterministe avec ϵ-transitions selon la méthode Aho-Ullman
3. Transformation de l'automate fini non déterministe en un automate fini déterministe avec la méthode des sous-ensembles
4. Minimisation d'un automate fini

   L'objectif de cette étape est de trouver une representation minimale, unique, (canonique) de l'automate reconnaissant un langage regulier.

   L'idée de l'algorithme va être de partitionner l'ensemble des états d'un automate fini déterministe. Pour trouver cette partition, nous allons procéder par raffinements successifs. Au départ, il n'y aura que 2 classes, les états appartenant à F (les états finaux) et les états appartenant à Q-F (les états non finaux). À chaque étape, on vérifie que pour tout couple d'états $q, q\prime$ appartenant à la même classe, on a la propriété $\forall a \in \Sigma, (q, a), (q\prime, a)$ sont dans la même classe, si ce n'est pas le cas $q, q\prime$ ne sont pas dans la même classe et on rejette $(q, q\prime)$.
   Le processus s'arrête quand il n'y a plus aucun retrait.

5. Utilisation de l'automate minimal obtenu précédemment pour tester si un suffixe d'une ligne du fichier textuel est reconnaissable par cet automate.

3 Graphe de Jaccard

Afin de caractériser ces grandes collections de documents, on s'intéresse à la pertinence que l'on accordera à un document pour répondre à une requête donnée. La solution la plus naturelle pour estimer cette pertinence est de mesurer la proximité entre le contenu d'un document et le contenu de la requête (d'autres documents).

La solution que l'on étudie dans ce projet consiste à caractériser l'importance des différents documents. Pour estimer cette importance, on considère qu'un document est plus important s'il existe une grande similarité avec les autres documents. Étant donné un ensemble de documents textuels et un seuil appelé seuil de similarité θ, nous définissons le graphe de Jaccard de ces documents comme le graphe géométrique de seuil θ et de fonction de distance Jaccard qui permet de calculer les différentes distances entre chaque document.
Note : Deux documents ont un lien si leur distance de Jaccard est inférieure ou égale au seuil.

Un graphe de documents est un graphe où les sommets correspondent à des livres et les liens sont calculés à partir de la distance de Jaccard 3.1 entre deux documents. Des exemples de graphes similaire incluent les graphes du web (i.e. de pages web reliées par des hyperliens) et les graphes de citations (i.e. d'articles scientifiques reliés par des références bibliographiques).

3.1 Distance de Jaccard

Étant donné un ensemble de livres L et un seuil θ, nous définissons le graphe de Jaccard de ces documents comme le graphe G géométrique de seuil θ et de fonction distance de Jaccard. La distance de Jaccard est utilisée particulièrement en statistiques pour comparer la similarité et la diversité entre des échantillons. L'indice de Jaccard est le rapport entre la cardinalité (la taille) de l'intersection des ensembles considérés et la cardinalité de l'union des ensembles. Il permet d'évaluer la similarité entre les ensembles. Dans ce projet la distance de Jaccard sera utilisée pour calculer la similarité entre deux livres. Soit deux documents textuels $D1$ et $D2$, nous définissons la distance entre $D1$ et $D2$ comme :

$$d(D1, D2) = \frac{\sum_{(m,k1)\in index(D1)\wedge(m,k2)\in index(D2)} max(k1, k2) - min(k1, k2)}{\sum_{(m,k1)\in index(D1)\wedge(m,k2)\in index(D2)} max(k1, k2)}$$

De-là, si la distance entre deux documents est inférieure ou égale au seuil θ défini alors on autorise une arête entre ces deux documents.

Analyse de complexité

Soit n la taille du graphe (le nombre de documents) et t_i le nombre de mots pour chaque documents textuels D_i.
La complexité pour calculer la distance entre deux document D_1 et D_2 est O($|D_2 \cup D_1|$).
Soit m le nombre de mots total pour tous les documents (m = $|D_1 \cup D_2 \cup ....D_i|$) et n le

nombre de documents. Pour calculer la matrice de distance de Jaccard, on a une complexité de $O(n^2 \times m)$.

3.2 Fonctionnalité de suggestion

A la suite d'une réponse à la fonctionnalité recherche, l'application retourne une liste de livres qui contiennent ce mot-clef. Si on clique sur une des réponse, on se dirigera vers une page de détail qui s'affichera tous les contenu de ce livre ainsi que une suggestion de 3 livres similaires.
Pour réaliser cette fonctionnalité, nous nous servons du graphe Jaccard calculé précédemment et on cherche dans le graphe de Jaccard 3 sommets qui sont les plus proches voisins avec le sommet du document actuel. Pour cela, nous avons d'abord trié la liste de voisin de ce sommet par la distance croissante et puis on retourne les 3 premiers sommets qui sont donc les plus proches voisins.

4 Classification par indice de centralité

A la suite d'une réponse à la fonctionnalité recherche, l'application retourne la liste des documents triée par un certain critère de pertinence : Par défaut, les résultats sont affichés par le nombre d'occurrences du mot-clef dans le livre, l'utilisateur peut aussi choisir d'autres façons de trier par indice de centralité décroissant dans le graphe de Jaccard. Dans notre application, nous avons implémenté ***closeness*** et ***betweenness*** pour les indices de centralité.

Dans la représentation des documents sous un graphe, les documents importants sont alors ceux qui occupent des positions centrales dans un tel graphe. Dans cette section, nous présentons les différentes méthodes vues en cours permettant de classifier les livres (documents textuels) grâce à leur indice de centralité.

La classification par indice de centralité se décompose en deux étapes : la construction du graphe avec la matrice de la distance de Jaccard vu dans la section 3.1, puis le calcul de l'indice de centralité avec la méthode différente.

4.1 Classification par Closeness

Dans un graphe donné, la centralité de proximité d'un noeud est une mesure de la centralité dans un réseau (dans notre cas un ensemble de livres) calculée comme étant l'inverse

de la somme de la longueur des chemins les plus courts entre un sommet (livre) et tous les autres sommets (livres) du graphe. Ainsi, plus un livre est central, plus il est proche de tous les autres livres (ce qui est calculé à partir de la distance de Jaccard 3.1).

Ainsi on peut décomposer le calcul en deux étapes :

1. Le calcul du plus court chemin avec **Floyd-Warshall**
   Soit G un graphe contenant les sommets $\{1, 2, 3, 4, \ldots, n\}$, on note $\mathscr{W}^k$ la matrice $\mathscr{W}^k_{ij}$. Pour k = 0, $\mathscr{W}^0$ est la matrice de distance de Jaccard de G et $\mathscr{W}^k_{ij}$ est le poids minimal d'un chemin du sommet i au sommet j n'empruntant que des sommets intermédiaires dans $\{1, 2, 3, \ldots, k\}$ s'il en existe un. La relation de récurrence pour calculer le plus court chemin est la suivante : $\mathscr{W}^k_{ij} = \min(\mathscr{W}^{k-1}_{ij}, \mathscr{W}^{k-1}_{ik} + \mathscr{W}^{k-1}_{kj})$

Algorithm 1 Algorithme de Floyd-Warshall

Input: $\mathscr{J}^n$: matrice de distance de Jaccard (matrice n × n)

Output: $\mathscr{W}^n$: matrice contenant le poids minimal parmi tous les chemins entre ces deux sommets (matrice n × n)

$\mathscr{W}^n := \mathscr{J}^n$
for *k :=0* ***to*** *n* **do**
  for *i :=0* ***to*** *n* **do**
    for *j :=0* ***to*** *n* **do**
      $\mathscr{W}^k_{ij} = \min(\mathscr{W}^{k-1}_{ij}, \mathscr{W}^{k-1}_{ik} + \mathscr{W}^{k-1}_{kj})$
    end
  end
end
return $\mathscr{W}^n$;

2. Le calcul de la proximité qui est défini comme suit :

$$C(x) = \frac{1}{\sum_y d(x, y)}$$

où $d(x, y)$ est la distance de Jaccard 3.1 entre les sommets x et y. Cependant, dans le cadre du projet on se réfère à sa forme normalisée qui représente la longueur moyenne des chemins les plus courts au lieu de leur somme, où N est le nombre de sommets, on obtient la formule suivante :

$$C(x) = \frac{N}{\sum_y d(x,y)}$$

4.2 Classification par Betweenness

La centralité intermédiaire est une mesure de centralité d'un sommet d'un graphe. Elle est égale au nombre de fois que ce sommet est sur le chemin le plus court entre deux autres nœuds quelconques du graphe.
Un nœud possède une grande intermédiarité s'il a une grande influence sur les transferts de données dans le réseau.
La centralité d'intermédiarité d'un sommet v est donnée par l'expression :

$$g(x) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

avec

- σ_{st} : le nombre de plus courts chemins de s à t
- $\sigma_{st}(v)$: le nombre de plus courts chemins entre s et t passant par v

4.2.1 Algorithmes

Pour le comptage de plus courts chemins passant par un point, nous avons adapté l'algorithme Floyd-Warshall. On crée une matrice $Path[i][j]$ qui stocke les prochains sommets dans les chemins plus courts entre i et j .

Algorithm 2 Floyd-Warshall

Input W := matrice d'adjacence de G (matrice $n \times n$)
Output $Path$:= matrice qui stocke les sommets suivants dans les plus courts chemins
//Initialisation $Path$
for all i,j **do**
  $Path[i][j] = \{j\}$
end for
for for k := 0 to n-1 **do**
  for for i := 0 to n-1 **do**
    for for j := 0 to n-1 **do**
      if $W_{ik} + W_{kj} < W_{ij}$ **then**
        $W_{ij} := W_{ik} + W_{kj}$
        $Path_{ij} := Path_{ik}$
      else if $W_{ik} + W_{kj} == W_{ij}$ **then**
        $Path_{ij} := Path_{ik} \cup Path_{ij}$
      end if
    end for
  end for
end for

W_{ij} est la distance minimale d'un chemin du sommet i au sommet j. on considère un chemin p entre i et j de distance minimale dont les sommets intermédiaires sont dans $1, 2, 3, \ldots, k$. Soit p emprunte le sommet k et $W_{ik} + W_{kj} < W_{ij}$ alors on affecte la valeur de $W_{ik} + W_{kj}$ à W_{ij} et on fait la même chose pour $Path_{ij}$. Si on trouve que $W_{ik} + W_{kj} == W_{ij}$, alors on ajoute des sommets de $Path_{ik}$ dans $Path_{ij}$.

La complexité en temps de cet algorithme est $O(n^3)$ et sa complexité en espace $O(n2)$.

Ensuite, on effectue un parcours DFS en convertissant la matrice de $Path$ en matrice $Chemins$ qui stocke une liste des plus courts chemins sous forme de liste de liste des sommets entre i et j, pour chaque sommet i,j, $Chemins[i][j] = \{\{k_1, k_2...\}, \{r_1, r_2...\} | k \neq i, j \cap r \neq i, j\}$, ici k,r sont les sommets intermédiaires dans les chemins plus courts entre i et j.

Analyse de complexité

L'analyse de complexité pour cette méthode se décompose en deux parties, la méthode du calcul des plus courts chemins avec **Floyd-Warshall** qui est en O(n^3) et la complexité du calcul de la Betweennes qui est négligeable face à cette complexité. Ainsi la complexité est O(n^3).

4.3 Classification par PageRank

Le PageRank est une mesure de centralité sur le réseau du web. C'est un algorithme d'analyse des liens concourant au système de classement des pages Web utilisé par le moteur de recherche Google. Il mesure quantitativement la popularité d'une page web. Le PageRank n'est qu'un indicateur parmi d'autres dans l'algorithme qui permet de classer les pages du Web dans les résultats de recherche de Google.

4.3.1 Implémentation de PageRank

Pour implémenter le PageRank dans notre projet. Nous avons le pseudo-code présenté dans le cours de CPA 2018 (4), nous attribuons à chaque sommet dans graphe G une valeur (ou score). Plus le PageRank d'un livre sera important et plus le noeud le représentant comportera d'arêtes.
La méthode se décompose en deux méthodes :

- Produit matrice/vecteur :
  où $\forall$ i $\in$ (livres), $\mathscr{B}_i = \sum_{j=1}^{n} \mathscr{M}_{ij} \times \mathscr{A}_j$
- Le calcule de Pagerank avec t itération.

Analyse de complexité

Le produit matrice/vecteur est en O(n^2) et pour le calcule du score pour tous les documents est en O($n \times t$) où t est le nombre d'itération.
Ainsi la complexité est de O(tn^2)

5 Base de données

Dans cette partie nous présentant le choix de construire de nos base de données se basant sur des documents textuels du site Gutenberg. La taille minimum de chaque livre est 10000 mots. Nous construisons la bibliothèque de 1664 livres sur la base Gutenberg (1).

Pour réaliser toutes les fonctionnalités du moteur de recherche, nous avons crées trois tables différentes : Livre, MatriceDistance et Classement pour stocker tous les informations nécessaires au développement de l'application.
Avec Django, on peut créer des tables par la création de modèle, chaque modèle correspond à une seule table de base de données, et chaque attribut du modèle représente un champ de base de données.

5.1 Table Livre

Pour stocker les livres dans une base de données, nous avons définit un model Livre avec des champs : titre, auteur, language et contenu. Nous avons créé une commande Django pour automatiser le téléchargement de livres et l'insérer dans la table de Livre. Le script est dans *management/commands/downloadbooks.py*

Tous les eBooks sur Gutenberg commencent par #10000, chaque eBook est posté dans différents formats de fichiers dans un seul répertoire. Par exemple, pour accéder à l'eBook #12345, on doit chercher dans l'adresse :

www.gutenberg.org/files/12345/12345.txt

Pour notre base de données de 1664 livres, nous cherchons les livres de id #10000 à #11664,

Ainsi pour un documents [ID] on doit faire la lecture du lien suivant :

www.gutenberg.org/files/[ID]/[ID].txt

Au cas où le livre de id $\#k$ a un nombre de mots inférieur à 10000, on l'abandonne et on continue à chercher sur le livre de id +1. Si, le code de requête n'est pas OK (différent de 200) quand on cherche le livre k, on passe au livre suivant de id $\#k + 1$ dans l'adresse **www.gutenberg.org/files/k+1/k+1.txt**. Pour chaque lecture de livre réussi, on extrait les informations et les insérer dans la table Livre.
Ce processus nous permet de faire la lecture directement sur le site **Gutenberg** afin de les insérer dans la base de données au lieu de télécharger un nombre volumineux de documents localement à la main.

5.2 Table MatriceDistance

Nous avons créé un modèle MatriceDistance avec des champs : id_ligne, id_colonne et value pour stocker la matrice de distance. id_ligne et id_colonne représente les id de livre(unique), et l'attribut value est pour stocker la distance calculée entre deux document avec son id égale respectivement à id_ligne et id_colonne. Donc chaque ligne dans la table représente une distance entre deux livres.

Le script *management/commands/calcul_matrice_jaccard.py* est fait pour le calcul de la matrice de distance Jaccard et puis le stocker dans la base de données dans la table MatriceDistance.

5.3 Table Classement

Nous avons créé un modèle Classement avec des champs : closeness, betweenness et pagerank pour stocker les classements de livres. Chaque attribut est pour stocker une liste d'id de livres triés par son indice. Donc il y n'a que une seule ligne dans la table de Classement. Quand on veut trier les résultats de recherche par closeness, on fait une fusion sur la liste d'id des résulats et la liste d'id stockée dans closeness en gardant l'ordre de closeness, et on produira une liste d'id qui est triée par closeness.

Le remplissage de table est aussi automatisé par le script *calcul_matrice_jaccard.py* en utilisant la matrice de distance calculée.

6 Déploiement

6.1 Choix du fournisseur Cloud

La solution implémentée utilise un index en mémoire sous forme de radix-tree pour accélérer la recherche de mots. Le stockage de cette structure en mémoire avec un contenu de plus de 800 livres devient trop coûteuse (plusieurs Go) pour les services gratuits offerts par les différents fournisseurs (Heroku, AWS Free Tier, etc).

On passe donc par une solution à mesure sur le service IaaS EC2 de AWS, avec une instance de type *r5a.large*, possédant 16Go de mémoire.

6.2 Déploiement des services

L'application est divisée en deux services. Un premier service développé en Python prend en charge l'initialisation de la base de données de livres à partir du site de la bibliothèque Gutenberg, le calcul du graphe de Jaccard, la classification par indice de centralité (Closeness, Betweenness, PageRank) et le frontend de l'application. Ce service est développé avec le framework Django.

Un deuxième service développé en Java sert de support pour l'indexation et la recherche des mots dans les livres. Lors de son initialisation, il construit le radix tree en mémoire à partir des données de livres chargées par le service python. Une fois l'index créé, on expose une API REST grâce à un servlet Java qui permet au service Python de réaliser des recherches de mots dans les livres. On déploie ce servlet dans un serveur tomcat.

Sur le disque, on retrouve les livres et les indices de centralité, qui sont stockés dans une base de données Postgres. Nos deux serveurs Django et Tomcat sont exposés au public à travers un proxy inverse Nginx, configuré avec terminaison SSL sous le domaine *daar-search.me*.

On peut consulter sur l'adresse daar-search.me l'application web doté d'une barre de recherche pour les livres, mais on peut aussi accéder directement aux données pour un mot, en format JSON : [(idlivre,[(colonne,ligne)])], sur l'adresse `daar-search.me/search/<mot>`

7 Conclusion

Nous avons répondu aux objectifs demandés, qui consistent à calculer la matrice de distance de Jaccard entre toute paire de documents textuels et le calcul de l'indice de centralité avec les trois méthodes vues en cours : *Betweenness*, *Closeness* et *Pagerank*. L'index permet d'effectuer des recherches simples quasi-instantanément et les vues de l'application sont responsives (voir annexe). L'interface graphique permet de faire des recherche et d'afficher le contenu complet des livres, en plus de recommander et d'ordonner les livres par indice de centralité.

Néanmoins, nous avons rencontré des problèmes par rapport au passage à l'échelle de nos solutions. L'indexation en mémoire sans recherche d'expressions régulières fonctionne pour environ 2000 livres, mais cette solution n'est plus viable lors de l'ajout des données pour permettre la recherche de RegEx. Il faudrait donc trouver un compromis entre consommation mémoire et performance en stockant par exemple une partie des données sur le disque. De plus, il resterait des optimisations mémoires à faire dans le stockage de l'arbre en soi.

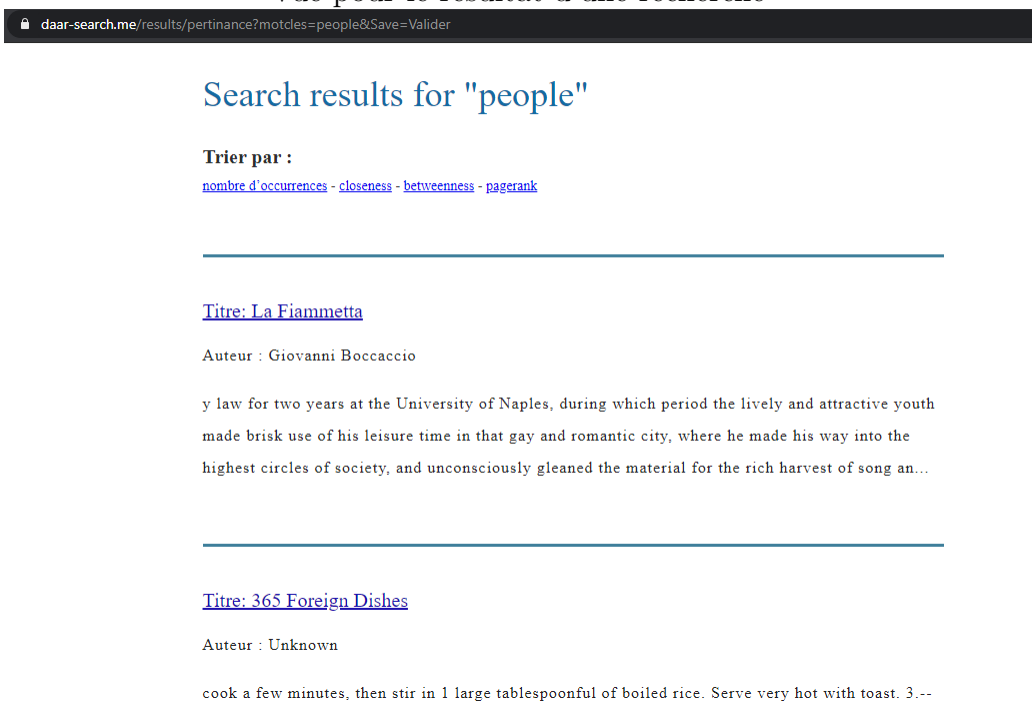
Les améliorations que nous pourrions ajouter sont :

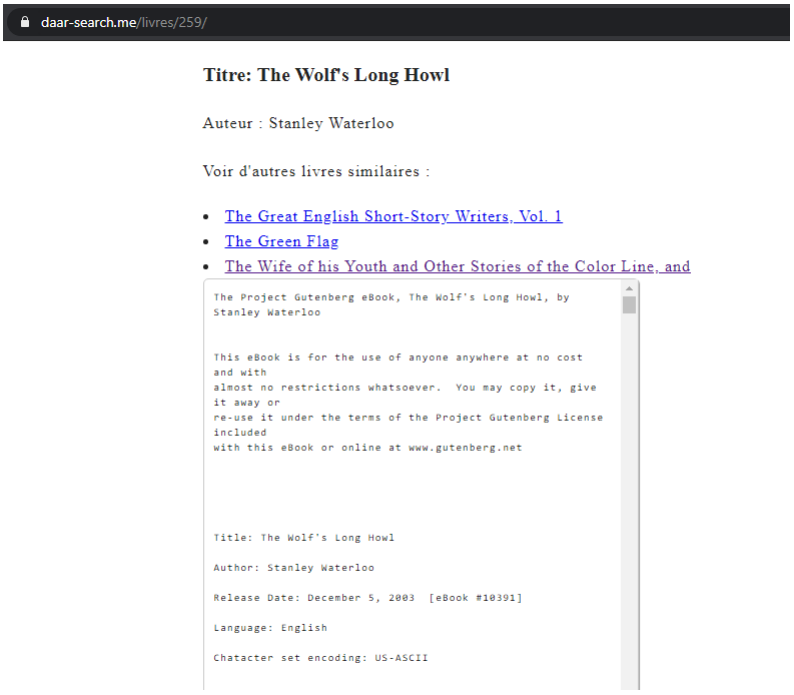
1. Une meilleure gestion du référencement des objets afin de prévenir des fuites de mémoire lorsque la taille de l'index augmente
2. Une optimisation des algorithmes des calculs d'indices de centralité et du graphe de Jaccard pour permettre une insertion plus rapide dans la base de données. A l'heure actuelle, le calcul de 1600 livres prend plus de 5 heures.

8 Annexes

Vue pour la page d'accueil

Vue pour le résultat d'une recherche





Vue pour afficher le contenu

Références

[1] Les livres sont récupérés sur la base de Gutenberg
`https://www.gutenberg.org/`

[2] Closeness centrality, wikipedia.
`https://en.wikipedia.org/wiki/Closeness_centrality`

[3] Betweenness centrality, wikipedia.
`https://en.wikipedia.org/wiki/Betweenness_centrality`

[4] Pagerank, cours CPA master 1 STL .
`https://www-apr.lip6.fr/~buixuan/files/cpa2018/slidesCM_cours5.pdf`