Projet UE DAAR

Rapport de projet :
Moteur de recherche d'une bibliothèque

VIGNE André
FOUQUET Justin
DUTRA Enzo

S1-2020

I - Introduction

Dans ce projet, nous allons proposer une solution pour un problème très général qui est celui de la recherche d'informations dans une vaste base de données. Nous allons ici nous concentrer sur la recherche de livres mais la méthode peut être applicable dans d'autres domaines comme la recherche de sites web, de mails, ou autres.

Depuis que les livres existent et avec leur nombre grandissant, le problème de la recherche de document se pose: comment trouver, dans une bibliothèque proposant des milliers voir des dizaines de millier d'ouvrages, le document recherché ?
Des solutions ont été apportées, comme classer les livres par auteur, les indexer afin de faciliter les recherches.

Les documents dématérialisés qui ont fait leur entrée avec l'arrivée de l'informatique n'échappent pas à ce problème. En réalité, grâce aux progrès technologiques qui permettent désormais de stocker des millions voire des milliards de documents, il a fallu trouver de nouveaux algorithmes performants pour rendre la recherche d'un document possible dans cet immense ensemble.

Étant donné le caractère massif de la base de données à gérer, il n'est pas possible de scanner tous les livres qu'elle contient à chaque recherche d'un utilisateur si l'on souhaite obtenir un résultat dans un temps raisonnable. La recherche s'effectue donc généralement sur une base de données plus légère composée d'informations pré-traitées.

Dans la suite de ce rapport, nous verrons donc un certain nombre d'algorithmes vus en cours, nous présenterons la solution pour laquelle nous avons opté puis nous étudierons les performances de celle-ci.

II - Présentation du problème et des algorithmes connus

Jaccard

La distance de Jaccard est une mesure de la similarité entre deux ensembles, qui évolue entre 0 (deux ensembles identiques) et 1 (aucun élément en commun). Le cas d'utilisation d'une bibliothèque est particulièrement approprié car deux livres traitant du même sujet ont généralement de nombreux mots en commun et donc une distance faible. Cette propriété permet d'avoir des résultats pertinents pour les suggestions faites lors d'une recherche.

Notre implémentation se base sur la définition de la distance de Jaccard ci-dessous, pour laquelle un attribut valant 1 est un mot présent dans un des deux livres et un attribut valant 0 est un mot non présent dans un des livres dont on cherche à calculer la distance.

M_{11} représente le nombre d'attributs qui valent 1 dans A et 1 dans B
M_{01} représente le nombre d'attributs qui valent 0 dans A et 1 dans B
M_{10} représente le nombre d'attributs qui valent 1 dans A et 0 dans B

$$J_\delta = \frac{M_{01} + M_{10}}{M_{01} + M_{10} + M_{11}}.$$

Betweenness / Closeness centrality

La centralité de proximité et d'intermédiarité sont des indicateurs qui permettent d'identifier les sommets les plus importants dans un graphe.

La centralité de proximité (Closeness Centrality) est définie comme l'inverse de l'excentricité d'un sommet, c'est à dire par l'inverse de la somme des distances à tous les autres sommets [2].

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

La centralité d'intermédiarité (Betweenness Centrality) compte le nombre de fois où un sommet agit comme un point de passage le long du plus court chemin entre deux autres sommets [3].

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

Ces deux mesures peuvent être utilisées pour déterminer quels documents peuvent être rapprochés de la recherche effectuée, pour un système de suggestion par exemple, mais dans notre application, nous n'utilisons que la distance de Jaccard.

Page rank

L'algorithme de pagerank [1] est très connu pour son utilisation dans le moteur de recherche Google qui a permis à la marque de se démarquer de la concurrence avec un classement des résultats bien plus pertinents que ce qui était fait à l'époque par leurs concurrents.

Dans cet algorithme, on voit le web comme un graphe dirigé où les nœuds correspondent à des pages web et les arêtes des liens hypertextes entre ces pages web (dirigé de la page source vers la page destination).

La valeur d'une page sera calculée en faisant la somme des liens qu'elle reçoit avec leur valeur respective.
Pour décider de la valeur d'un lien, on va partir de 2 suppositions :

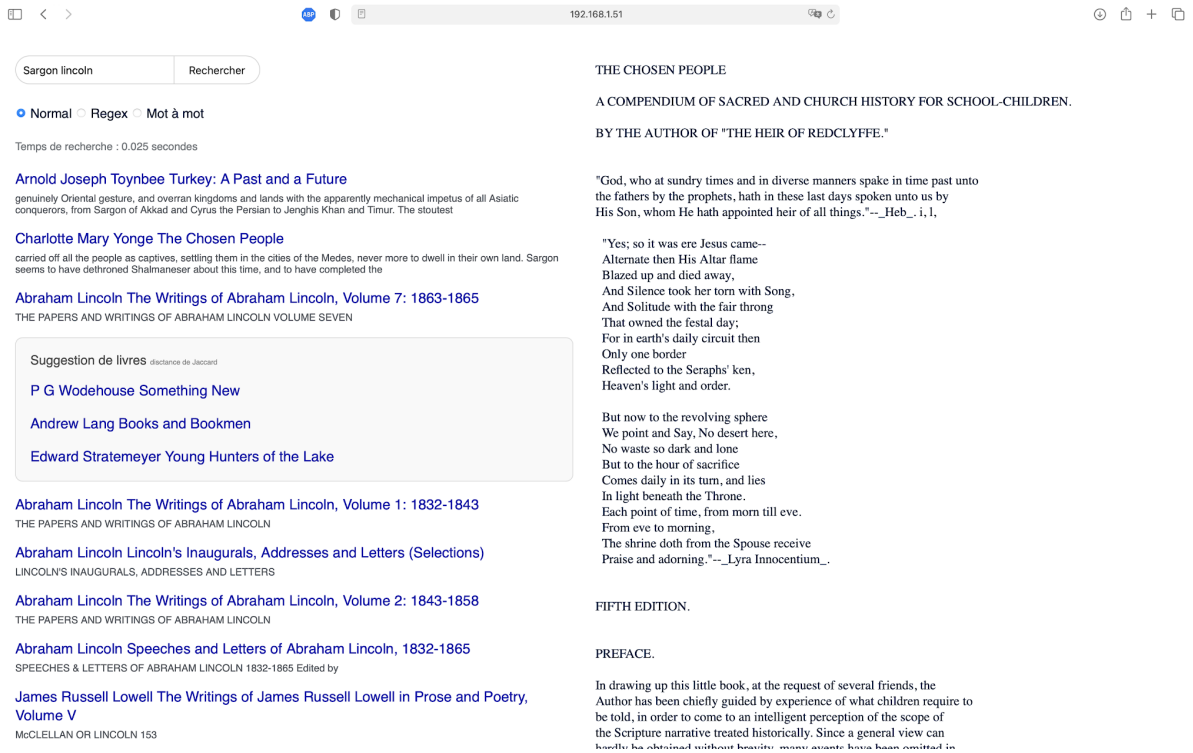
- Un lien venant d'une page importante a plus de valeur
- Si une page contient beaucoup de liens sortants, leur valeur sera moindre.

Le problème qui se pose désormais est que pour calculer la valeur d'un lien, nous avons besoin de la valeur des pages mais pour calculer la valeur des pages, nous avons besoin de la valeur des liens.

La solution qui est proposée dans cet algorithme utilise les chaînes de Markov. Le principe général va être de partir d'une situation neutre où toutes les valeurs des pages sont à 1 puis prendre la valeur des pages que l'on obtient à partir de la valeur des nœuds pour la remettre en entrée de la fonction.
On peut prouver que la valeur des pages va converger en temps raisonnable. Lorsque la valeur converge, on trie la liste des pages dans l'ordre décroissant et on la retourne.

Cet algorithme ne semble cependant pas tout à fait correspondre à notre situation car les livres ne possèdent pas de liens hypertextes entre eux.

III - Présentation de notre implémentation



L'algorithme que nous allons présenter dans la suite du rapport nécessite donc une étape préliminaire avant de pouvoir être utilisé. En effet, étant donné le caractère massif de la base de données à gérer, nous ne pouvons pas nous permettre de scanner tout le contenu des livres qu'elle contient à chaque recherche d'utilisateur. Nous allons donc effectuer la recherche et calculer le classement des résultats sur une base de données plus légère composée d'informations pré-traitées.

Prétraitements

Pour constituer notre bibliothèque, nous avions le choix de la manière de récupérer des livres. Nous avons choisi d'utiliser le corpus disponible sur le site de S. Lahiri [4] composé d'environ 3000 livres.

L'application que nous présentons comporte les 4 points demandés : recherche, recherche avancée, classement implicite et suggestions. Afin que toutes ces fonctionnalités donnent rapidement des résultats lors d'une requête utilisateur, nous effectuons 5 étapes de pré-traitement à l'aide de scripts bash, appelés successivement dans le script init.sh.

- Filtre : La première étape est de retirer de la bibliothèque les livres ne correspondant pas au critère des 10000 mots minimum. Afin de diminuer la taille totale du dictionnaire de la bibliothèque, nous retirons aussi tous les livres contenant des caractères non latins.
- Renommage : Pour faciliter les opérations sur les fichiers, nous avons pris la décision de créer une table d'indexage associant un livre à un identifiant et de renommer les fichiers par l'identifiant. Ceci permet de supprimer les espaces et de pouvoir considérer les titres comme des entiers dans le code Java.

- Indexage : Pour chaque livre, cette étape crée un fichier contenant des lignes "*nombre mot*" pour lesquelles *nombre* est le nombre d'occurrences du mot dans le livre. Cette étape permet de s'affranchir de tous les séparateurs de mots étranges lorsqu'on aura besoin de l'information en Java. (liste non exhaustive : \r, --, :)
- Compte Mots : Pour notre fonction de calcul de score, il est utile de connaître le nombre total de mots de chaque livre. Nous avons décidé de calculer ces valeurs en bash et les regrouper dans un fichier wordCount.txt
- bigIndexing : Pour assurer le fonctionnement de notre implémentation de la recherche avancée, ce script calcule le dictionnaire complet des mots présents dans la bibliothèque.

Fonctionnalités attendues

Classement

Lorsque l'utilisateur entre une requête dans la barre de recherche, il s'attend à ce que l'application lui renvoie des livres contenant les mots qu'il a entrés. Afin de sélectionner les livres les plus pertinents, nous avons attribué un score à chaque mot dans chaque livre. Ce score est défini comme suit :

$$\frac{nombre\ d'occurrences\ du\ mot\ dans\ le\ livre}{nombre\ d'occurrences\ total\ du\ mot * nombre\ de\ livres\ dans\ lesquels\ le\ mot\ apparaît\ au\ moins\ 1\ fois}$$

Ces paramètres permettent que lors d'une recherche contenant plusieurs mots, l'importance des mots présents dans une majorité des livres (tels que the, a, I) soit considérablement réduite au profit des mots "importants" de la recherche.

Recherche

La recherche est la fonctionnalité centrale du projet, à l'aide du système de score elle permet de renvoyer les livres jugés les plus pertinents. La barre de recherche permet d'entrer un ou plusieurs mots séparés par des espaces. Dans le cas où plusieurs mots sont entrés, le score de chaque livre est la somme des scores de chaque mot présent dans ce dernier.
Nous avons également implémenté une fonction de recherche "stricte" qui ne renvoie que les livres dans lesquels tous les mots de la requête sont présents. Cette fonctionnalité est appelée "mot à mot" dans l'application web.

Recherche avancée

La recherche avancée correspond au support de regex dans la recherche. Pour cela, nous avons fait le choix d'interpréter la regex fournie par l'utilisateur et agir comme s'il avait entré la liste de tous les mots retournés par un appel à egrep sur le dictionnaire total de la bibliothèque. Cet élargissement rend difficilement compatible le mode avancé et le mode strict, ainsi nous avons choisi d'interdire l'utilisation simultanée de ces deux options.

Suggestions

Nous avons choisi d'interpréter le sujet en sélectionnant les N premiers livres qui sont voisins des 3 meilleurs résultats. Pour cela nous calculons la matrice des distances de jaccard entre chaque livre en amont des recherches, puis la matrice est lue dans un fichier au moment du démarrage de l'application pour la charger en mémoire.

Notre implémentation

Classement

Pour stocker les scores associés à un mot, nous avons utilisé une Map prenant comme clé le mot et comme valeur une liste de couples. Les Couples sont des objets simples avec 2 champs : un entier représentant l'id du livre dans lequel on peut retrouver le mot et un flottant (double) qui est la valeur du score.

En parcourant les listes de mots fournies par le script indexing, qui a généré des fichiers avec des lignes de la forme : quantité mot, on peut remplir facilement la map avec comme valeur de départ pour le score le nombre d'occurrence du mot dans le livre. Lors de ce premier passage, on construit également une seconde Map contenant pour chaque mot son nombre total d'occurrence, ce qui évite de devoir parcourir chaque liste de couple lors du deuxième passage.

Un deuxième passage une fois la map construite permet de remplacer les scores par leurs valeurs finales car les deux autres valeurs nécessaires à leur évaluation sont disponibles (total nb occurrence du mot et nb de livres où il est présent).

Recherche

Nous avons choisi de stocker les scores dans une map contenant plusieurs listes courtes pour pouvoir récupérer les scores pour chaque livre rapidement. Les résultats sont stockés dans une map <livre,score>, par un parcours de toutes les listes associées aux mots de la requête qui ajoute le score au score total actuel du livre. Cette approche permet une recherche très rapide même lorsque le nombre de mots dans la requête augmente.

Recherche avancée

Cette fonctionnalité est construite comme la précédente, avec un appel à egrep pour interpréter la requête. Dans les cas où cet appel renvoie trop de lignes, la fonction java utilisée prend un temps considérable. C'est pourquoi nous avons implémenté une version de findBestBooks avec un timeout (qu'on a mis à 1 seconde), qui renvoie un message à l'utilisateur quand le temps est dépassé.

Suggestions

Pour le calcul de la matrice de jaccard, nous utilisons la formule énoncée dans la section II.A. Pour effectuer ce calcul, on cherche d'abord le dictionnaire complet des mots présents dans ces deux livres. Pour s'assurer que deux livres du même sujet mais n'ayant pas la même longueur aient une distance faible, le nombre d'occurrences est pondéré afin de simuler que les deux livres ont le même nombre de mots.

Une fois le dictionnaire obtenu, pour chaque mot on compte son nombre d'occurrences dans i et j (occi et occj). On incrémente ensuite m11 de min(occi,occj), m10 de max(0, occi - occj) et m01 de max(0, occj - occi).

En pratique, on constate que les livres considérés comme proches semblent cohérents.

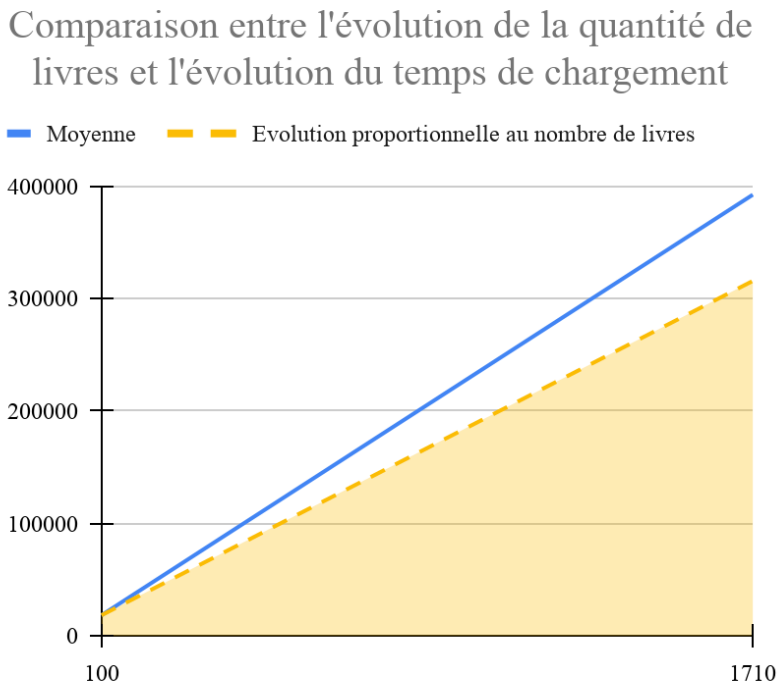
IV - Tests de performances

Pour mettre à l'épreuve notre approche, nous avons effectué plusieurs tests de performances que nous montrerons ici et pour lesquels nous allons proposer une interprétation.

Temps de démarrage de l'application

Le temps de démarrage de l'application est ici non négligeable, mais ceci n'est pas nécessairement rédhibitoire car dans un contexte de production il est probable que l'application reste active pendant relativement longtemps.

Dans le graphe suivant, chaque point est la moyenne de 10 mesures :



On peut remarquer que l'évolution du temps de chargement de l'application évolue plus rapidement que l'évolution du nombre de livres. C'est à dire que si le chargement d'une bibliothèque composée de 1 livre prends x secondes, le chargement d'une bibliothèque composée de n livres prendra plus de n * x secondes.

La mesure n'a été effectuée que sur 2 tailles de bibliothèques différentes. Il aurait en effet pu être intéressant d'effectuer des mesures sur d'autres tailles de bibliothèque.

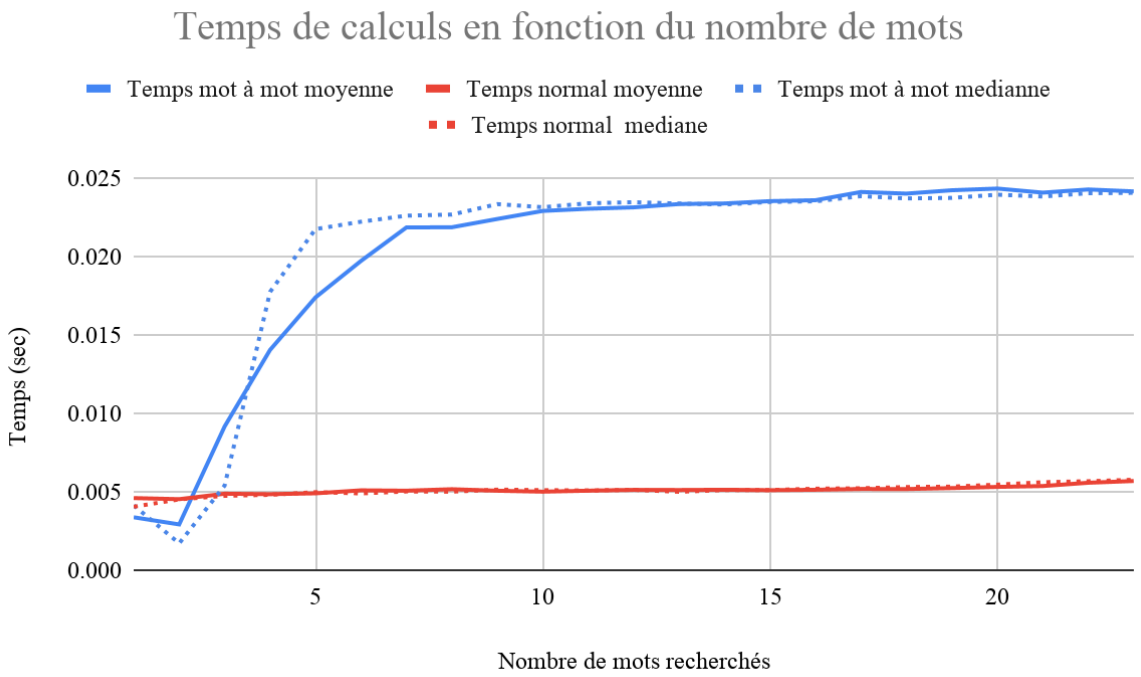
On peut cependant noter que le différentiel entre la courbe jaune et bleue est raisonnable. Ce chargement pourrait en revanche difficilement tenir la charge face à une bibliothèque de taille bien plus conséquente (100.000 livres ou 1 milliards de livres).

Temps de calculs des résultats

En fonction du nombre de mots

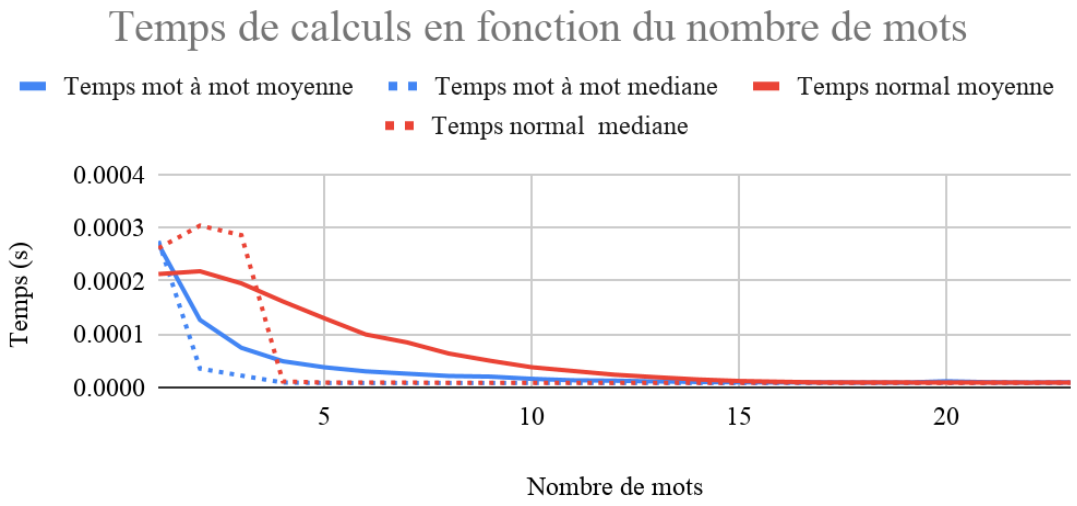
Le graphique suivant nous permet de connaître le temps moyen d'une recherche en fonction des mots. Chaque mot pouvant prendre un temps de recherche différent (en fonction du nombre de livres dans lesquels il se trouve), il nous a semblé intéressant d'y ajouter une courbe correspondant au temps médian du temps de recherche en fonction du nombre de mots. Cette deuxième mesure étant bien moins sensible à la variance, elle permet d'apporter des informations supplémentaires.

Dans le graphe suivant, chaque point est la moyenne de 500 mesures :

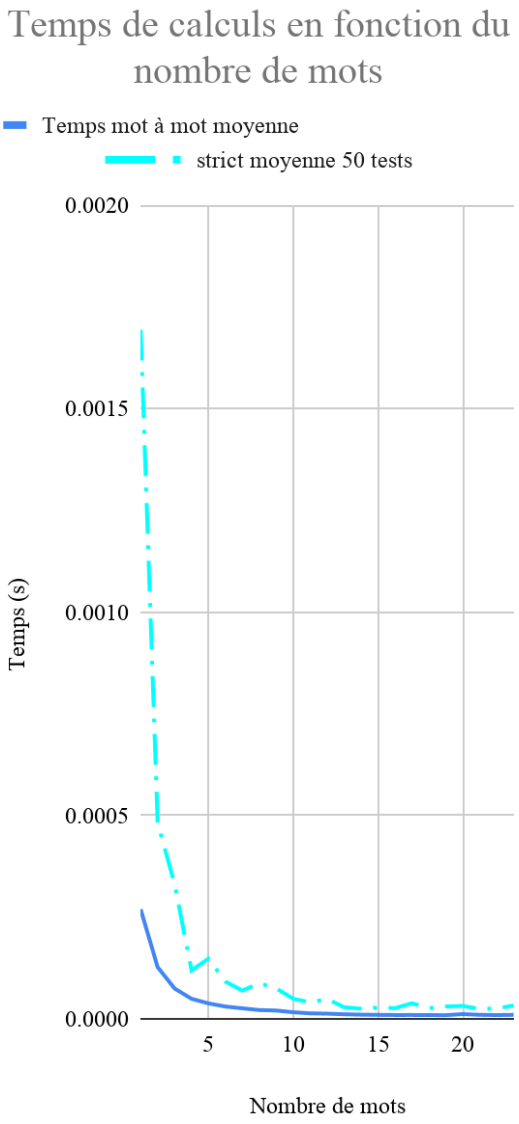


Le temps de calcul des résultats pour une recherche normale (chaque résultat ne contient pas forcément la totalité des mots de la recherche) semble être constant pour un nombre de mots raisonnable. (Il nous semble en effet improbable qu'un utilisateur recherche plus de 20 mots d'un coup).
En ce qui concerne le temps de calcul des résultats pour une recherche mot à mot (les résultats doivent inclure tous les mots de la recherche), il semblerait qu'il évolue de manière logarithmique en fonction du nombre de mots recherchés.
On peut également noter la différence entre la courbe du temps moyen et la courbe du temps médian qui pourrait s'expliquer par le fait qu'entre 2 et 10 mots la variance du temps de calcul est plus élevée.

Il peut également être intéressant de voir ces mêmes mesures appliquées à une bibliothèque de taille plus petite (100 livres au lieu de 1710). Dans le graphique suivant, chaque point correspond à 5000 mesures.



On remarque une grande différence dans l'évolution de la courbe. Nous n'avons pas réussi à expliquer ce comportement. Il est possible que cela soit dû à l'adaptation du processeur à la charge de travail.



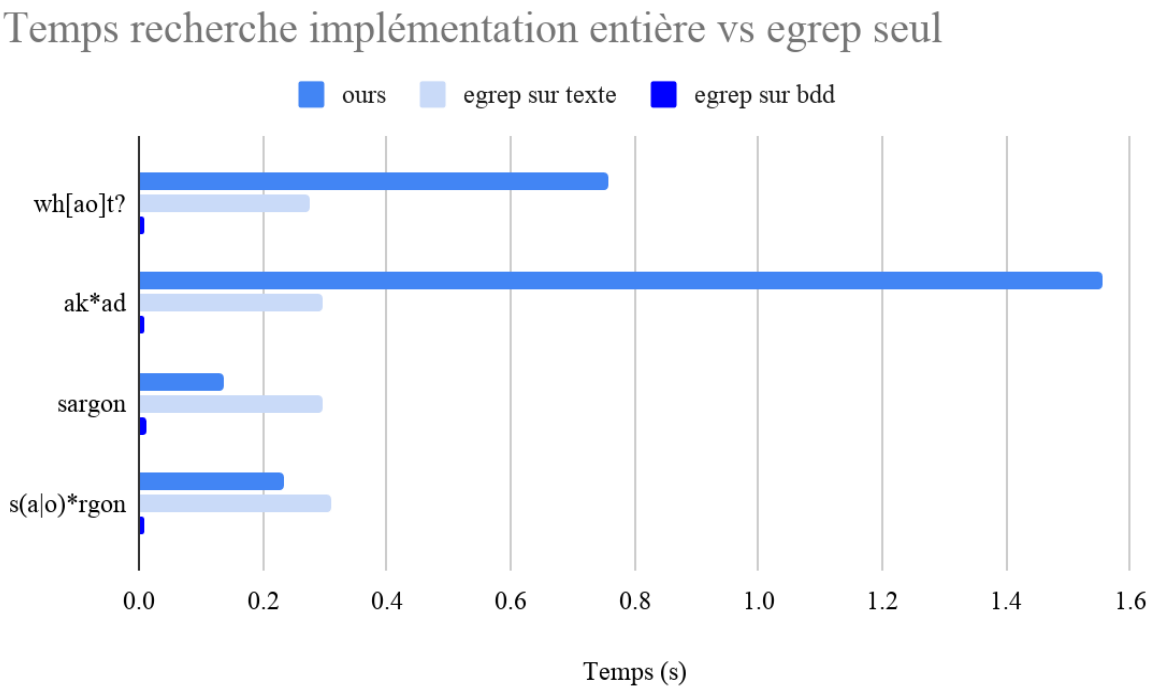
Cette hypothèse semble se confirmer en effectuant moins de mesures par points (50 au lieu de 5000 sur la courbe cyan). On observe que lorsque le processeur n'a pas le temps de s'adapter à la charge de travail et donc de lisser la moyenne de la première mesure, le temps de calcul que l'on semble mesurer explose. Cela expliquerait également la chute de du temps médian plus abrupte que celle du temps moyen. Ce phénomène pourrait être aussi visible grâce à la taille très réduite de la bibliothèque qui rend le temps de calcul des résultats presque négligeable face à d'éventuels phénomènes parasites.

Si cette hypothèse s'avérait vraie, il pourrait être intéressant de le savoir en lisant les mesures faites sur la bibliothèque de 1710 livres.

En fonction de la regex

La recherche par regex étant possible, il peut également être intéressant de comparer le temps de calculs incluant tout le processus de classement des résultats ainsi que l'utilisation de grep avec l'utilisation de egrep seul.

La graphique suivant montre cette comparaison (chaque point est une moyenne de 10 mesures) :



On remarque que le temps de calcul induit par notre implémentation est très variable en fonction des termes de recherche. Ces variations sont dues aux méthodes appelées par *getPossibleWordsFromRegex* dont l'évolution du temps d'exécution en fonction du nombre de lignes retourné par egrep n'est pas linéaire.

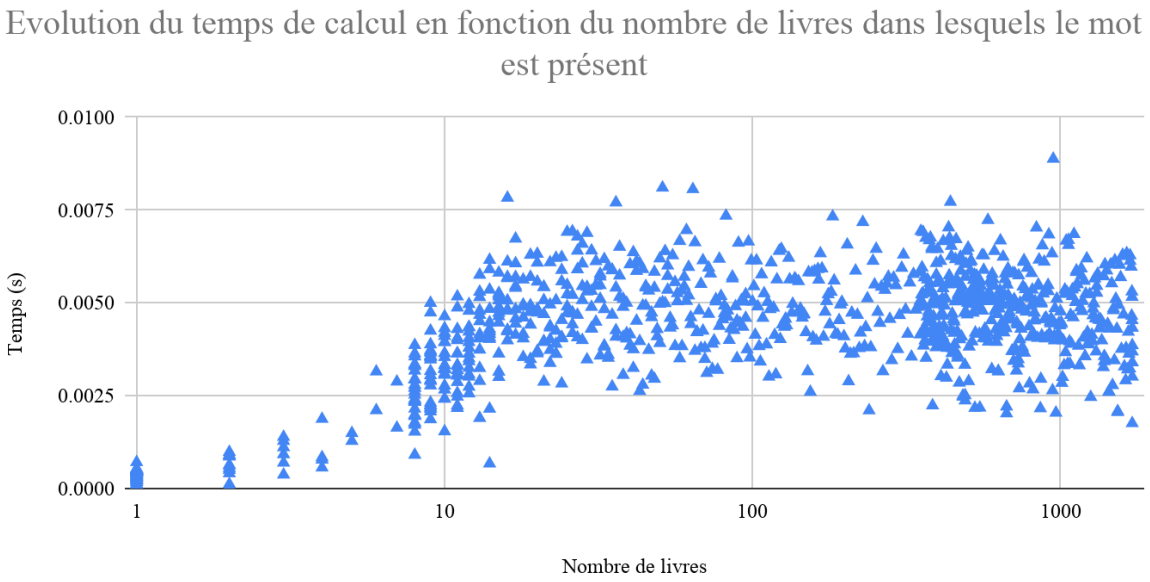
Pertinence des résultats

Notre formule de calcul des scores donne une importance plus élevée aux mots qui sont présents dans un nombre réduit de livres. En effet, le score cumulé d'un mot dans l'ensemble des livres est lié à l'inverse du nombre de livres dans lequel il est présent.

Voici quelques exemples de valeurs de score :

	gratitude	sargon	carrot	exotic	king	the
médiane	1000	250	0,32	0,005	0,00014	0,00034
moyenne	1000	250	0,54	0,073	0,0063	0,00034

On peut constater que la médiane des scores est souvent bien inférieure à la moyenne (et jamais supérieure), ce qui signifie que les livres contenant de nombreuses fois un mot lui associent un score très élevé comparé à ceux dont le sujet n'a pas de rapport avec le mot en question. Cette observation renforce l'idée que les résultats obtenus sont pertinents.



On a choisi une échelle logarithmique pour mieux voir l'évolution du temps de calcul pour les mots présents dans peu de livres. On peut voir que pour les mots présents dans 1 à 20 livres, ce nombre de livres a une grande influence sur le temps de calcul. A partir du moment où le mot est présent dans au moins ~20 livres, ce chiffre ne semble plus avoir d'impact significatif sur le temps de calcul. Nous n'avons cependant pas trouvé d'explication particulièrement précise sur ce changement d'évolution à partir d'un certain nombre de livres donnés.
Le temps de recherche constant constaté sur le graphe à partir de ~20 livres pourrait éventuellement être dû à la manière dont sont implémentés les objets Map en java, et dans lesquels nous effectuons nos recherches ou encore diverses optimisations dans les algorithmes de tri de la librairie standard.

Discussion

Les différents tests produits dans le cadre de ce projet ont été effectués sur 2 machines :

Machine 1 :

Tests	Figure 2, 3, 4
Configuration	
Nom du modèle	XPS-15-9500
Système d'exploitation	Ubuntu 20.04.1 LTS
Version de Java	15.0.2 2021-01-19
Version du JRE	Java(TM) SE Runtime Environment (build 15.0.2+7-27)
Version de la JVM	HotSpot 64-Bit Server 15.0.2+7-27 mixed mode sharing
Nom du processeur	i7-10750H
Nombre de cœurs	6
Nombre de threads	12
Mémoire (RAM)	16 Go, 3200 MHz

Machine 2 :

Tests	Figure 1
Configuration	
Nom du modèle	Aspire A715-71G
Système d'exploitation	Windows 10
Version de Java	15.0.1 2020-10-20
Version du JRE	OpenJDK Runtime Environment (build 15.0.1+9-18)
Version de la JVM	OpenJDK 64-Bit Server VM (build 15.0.1+9-18, mixed
Nom du processeur	i5 7300HQ
Nombre de cœurs	4
Nombre de threads	4
Mémoire (RAM)	16 Go, 2400 MHz

Il aurait peut être été intéressant d'avoir un autre système de moteur de recherche de référence nous permettant de juger de la pertinence des choix d'implémentation que nous avons effectué et ainsi enrichir l'interprétation des différentes mesures effectuées.

Les résultats de recherches se basant sur les mots entrés par l'utilisateur (hors suggestions) nous semblent tout à fait corrects et il n'a pas été nécessaire de créer une liste noire de mots trop communs à ne pas analyser. L'ajout de mots communs tels que "the", "a" ou "for" a un effet négligeable sur le résultat (éventuels changement d'ordres à partir de la position ~5).

Les suggestions semblent également proposer des livres en rapport aux 3 premiers livres de la liste des résultats même s'il peut être difficile de juger de leur qualité ne connaissant pas ces livres.

Références

[1] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd, 1998, The PageRank Citation Ranking: Bringing Order to the Web.

[2] Alex Bavelas, « Communication patterns in task-oriented groups », *J. Acoust. Soc. Am.*, vol. 22, nº 6, 1950, p. 725–730.

[3] Ulrik Brandes, « A faster algorithm for betweenness centrality », *Journal of Mathematical Sociology*, vol. 25, 2001, p. 163–177.

[4] https://web.eecs.umich.edu/~lahiri/gutenberg_dataset.html