Algorithmes et Structures de Données pour l'Indexation de Grands Volumes de Données Textuelles

Siying Jiang, Alex Huang, Adrien Moll, Adrien Lepic, Ziggy Vergne

30 novembre 2018

1 Introduction

Le but de ce projet est de mettre en oeuvre un système d'indexation de données textuelles pour le requêtage d'articles du journal LaTimes. Nous allons vous exposer nos fonctionnalités implémentées, nos résultats et les problématiques rencontrées.

2 Présentation de la structure du projet

Nous avons choisi de diviser le projet Wordtraveller en deux programmes.

— Wordmapper, chargé de l'indexation (création de l'Inverted File)

— Wordexplorer, chargé de l'exécution des requêtes utilisateur sur l'Inverted File créé par Wordmapper.

Les deux programmes ont pour point commun l'utilisation de l'Inverted File, constitué d'un fichier .voc, d'un fichier .pl et d'un fichier .score.pl.

— Le fichier .voc, au format textuel, contenant l'ensemble des mots ainsi qu'un offset indiquant l'emplacement de la Posting List du mot dans le fichier .pl

— Le fichier binaire .pl contenant les Posting Lists de tous les mots, groupées par mot et triées par document Id croissant

— Le fichier binaire score.pl contenant les Posting Lists de tous les mots, groupées par mot et triées par score croissant

En plus des fichiers décrits précédemment, l'utilisateur peut également générer et utiliser des fichiers .ri et .vori, dédiés au Random Indexing.

3 Fonctionnalités implémentées

En ce qui concerne la création de l'Inverted File, nous avons implémenté les fonctionnalités suivantes :

— Indexation en mettant tout le contenu de l'Inverted File avant de le stocker sur le disque.
— Indexation partiellement en mémoire avec stockage sur disque périodique sous la forme de fichiers temporaires avant de les fusionner
— Compression des fichiers temporaires, en utilisant les algorithmes Vbyte ainsi que Gzip (via une bibliothèque annexe pour ce dernier).

En ce qui concerne les requêtes, nous avons implémenté les fonctionnalités suivantes :

— Requête sur un ou plusieurs termes en utilisant les algorithmes suivants :
  — Naïve
  — FaginsTopK
  — FaginsTA
  — FaginsTA avec Epsilon paramétrable
— Recherches de synonymes via les algorithmes dits de Random Indexing ainsi que ceux de la bibliothèque Word2Vec.
— Décompression des fichiers compressés lors de la phase d'indexation.

4 Utilisation du programme

Dans le programme Wordmapper, nous avons les options ci-dessous :

— -d : dossier avec les documents
— -f : nom de fichier pour enregistrer les fichiers Posting List et vocabulaire
— -o : dossier pour enregistrer les fichiers après l'indexation
— –zip : compression Vbyte et Zip à la fin
— –partial : créer les fichiers par réunion de plusieurs fichiers avec une granularité de documents choisie. Valeur conseillée : 2000.
— –stemmer : activer le Stemmer
— –randomindexing : activer le Random Indexing

Dans Wordexplorer :

— -d : dossier avec les fichiers Posting List et vocabulaire
— -f : nom de fichier Posting List et vocabulaire
— -q : requête contenant les termes séparés par une virgule
— -n : nombre de documents souhaité en résultat
— –stemmer : activer Stemmer sur les termes de la requête
— –algo : algorithme de requêtage souhaité. Options possibles : naive, fagins et faginsTA.
— –view : type de visualisation du résultat. Options possibles : simple et fullText.
— –vpath : dossier avec les fichiers sources LaTimes pour l'option –view fullText
— –improvedquery : activer la recherche de synonymes pour l'amélioration de la requête

5 Architecture des modules

Nous avons structuré notre projet en plusieurs modules :
- analysis.py : module qui sert à parser et indexer tous les journaux du LaTimes.
- compressor.py : module qui sert à faire la compression et décompression VByte et Zip des fichiers .vo et .pl
- faginstavf.py : module qui, à partir d'un ensemble de mots, applique l'algorithme Fagins TA pour avoir les journaux plus pertinents.
- faginstopkvf.py : module qui, à partir d'un ensemble de mots, applique l'algorithme Fagins pour avoir les journaux plus pertinents
- filemanager.py : module qui sert à faire le lien entre la RAM et le disque. Il lit ou écrit les différents fichiers .vo et .pl sur le disque.
- findsynonym.py : module qui applique et utilise Word2Vec pour apprendre et prédire des mots similaires (synonymes).
- naivetopk.py : module qui, à partir d'un ensemble de mots, applique l'approche naïve pour chercher les journaux plus pertinents.
- preprocessing.py : module qui fait un prétraitement sur les données textuelles afin qu'ils soient tous homogènes.
- randomIndexing.py : module utilisé pour indexer les différents termes par l'approche vectorielle.
- randomIndexingFindSynonym.py : module stand-alone pour faire des requêtes de synonymes. Il faut indexer les termes avant de faire des requêtes.
- view.py : module qui sert à définir l'affichage sur le terminal.
- wordexplorer.py : module d'entrée pour faire des requêtes sur l'indexation.
- wordmapper.py : module d'entrée pour indexer les journaux LaTimes.

6 Comparaison des performances : Création de l'indexe

6.1 Environnement de test

Pour les tests de performance de la création de l'index, nous avons utilisé un ordinateur équipé d'un processeur i5-2520M (2 cœurs, 4 threads), de 8Go de mémoire vive de type DDR3 1333 Mhz, avec un SSD SanDisk SSD PLUS 240GB, branché sur le secteur, avec aucun programme tournant en parallèle. La version de Python est 3.6.6 et le système d'exploitation est Fedora 28.

6.2 Tokenisation

Le Tokeniseur utilisé est celui de NLTK, et nous avons utilisé une expression régulière afin de séparer les mots du texte, tout en ne supprimant pas les tirets.

Cette expression régulière est exécutée avant une autre qui remplace les quantités de monnaie (signe de la monnaie suivi d'un nombre et éventuellement d'un mot tel que « million » ou « billion ») par <money>, suivie par une expression régulière similaire, mais sans le symbole monétaire, afin de détecter les nombres et les remplacer par <number>.

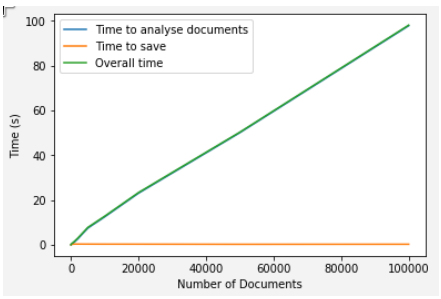
6.3 Stemming

Nous avons donné la possibilité à l'utilisateur de choisir ou non d'activer le Stemming. Si l'option est activée, un Stemming sera opéré sur les mots des documents. Le Stemming utilisé est celui de NLTK. Nous avons choisi de désactiver par défaut le Stemming, car celui-ci prend beaucoup de temps et ralentit massivement le temps d'exécution de l'indexation.

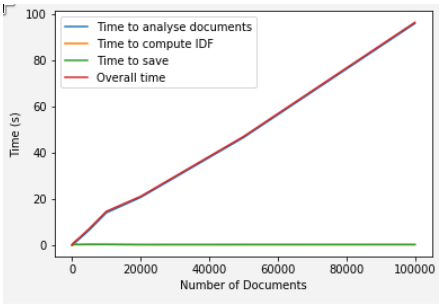
6.4 Construction de l'index en mémoire

Dans un premier temps, nous avons exploré la voie de l'indexation tout en mémoire. Celle-ci se déroule en trois parties : l'analyse des documents, le calcul du score IDF, et la sauvegarde sur disque des fichiers de vocabulaire, Posting List triée par Id et par score. Pour la suite des tests, nous avons désactivé le Stemmer afin de pouvoir nous concentrer sur les temps d'exécutions liés à l'analyse des documents en entrée ainsi qu'à leur sauvegarde.

Afin de lisser nos résultats, nous avons effectué les benchmarks trois fois et fait une moyenne des résultats, qu'il est possible de voir sous la forme d'un graphe en figure 1. Les tests ont été effectués avec 1, 10, 100, 1000, 2000, 5000, 10000, 20000, 50000 et 100000 documents.



(a) Graphe de l'évolution des temps d'exécution, sans calcul de l'IDF.



(b) Graphe de l'évolution des temps d'exécution, avec calcul de l'IDF.

FIGURE 1 – Graphes de l'évolution des temps d'exécution de l'indexation, sans et avec calcul de l'IDF.

On peut donc voir que l'immense majorité du temps est passé à analyser les documents d'origine, et que le temps passé à sauvegarder le fichier final et à calculer l'IDF (si on le fait) est négligeable. En effet, la ligne du temps d'analyse

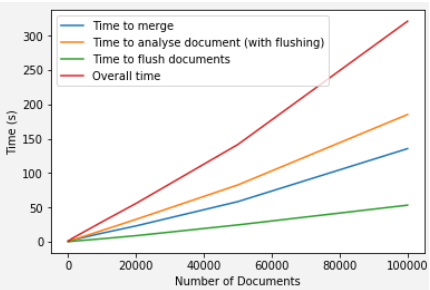
est presque confondue avec celle du temps total, et la ligne de calcul de l'IDF et de la sauvegarde sur disque est proche de zéro.

De plus, l'évolution du temps d'exécution de l'indexation évolue linéairement en fonction du volume de document analysé, ce qui est un indicateur fort de la mise à l'échelle de notre programme.

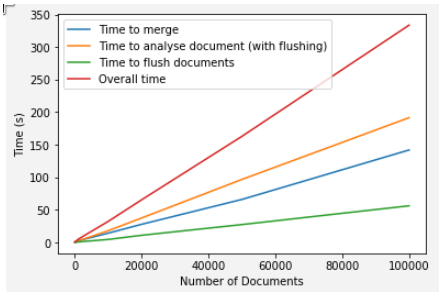
6.5 Construction de l'index par fusion

Nous avons également exploré la voie de l'indexation par partie. Ainsi, nous analysons une sous-partie des documents, créons un Inverted File partiel et le stockons en mémoire. Une fois tous les documents analysés, nous pouvons alors parcourir tous les Inverted File partiels en parallèle et fusionner les fichiers de vocabulaires partiels, les Posting Lists partiels et créons aussi le fichier de Posting List trié par score. Nous donnons la possibilité à l'utilisateur de choisir la granularité avec laquelle les Inverted Files partiels sont créés.

Nous avons donc effectué des constructions d'index par fusion sur 1000, 10000, 20000 et 50000 documents en faisant varier la granularité sur 1000, 2000, 5000 et 10000. Les graphes détaillant l'évolution et la répartition des temps d'exécution pour une granularité de 1000 et de 10000 sont visibles respectivement à la figure 2a et figure 2b. Nous n'avons pas pu faire descendre la granularité plus bas, car nous sommes limités dans le nombre de fichiers que nous pouvons ouvrir simultanément. Pour pouvoir faire une granularité plus basse, il nous aurait fallu réécrire l'algorithme afin qu'il ferme et rouvre de manière régulière les fichiers partiels, ce qui aurait dramatiquement fait chuter les performances.



(a) Sauvegarde de fichiers partiels tous les 1000 documents



(b) Sauvegarde de fichiers partiels tous les 10000 documents

FIGURE 2 – Graphe de la répartition du temps d'exécution en fonction du nombre de documents.

Les graphes n'étant pas suffisamment précis pour que nous puissions discerner des différences de performances, nous exposons les résultats de plusieurs tableaux (voir Tableau 5).

Les résultats montrent qu'à volume de document égal, les performances variaient en fonction de la granularité de sauvegarde de fichiers temporaires. Nous

Nombre documents	1000	10000	20000	50000
Fusion fichiers temporaires (s)	3,45	13,71	23,90	58,95
Analyse documents d'origine (et sauvegarde temporaire) (s)	2,58	20,46	37,05	83,72
Sauvegarde sur disque (temporaire) (s)	0,91	4,97	11,77	24,38
Total (s)	6,07	34,2	60,97	142,81

TABLE 1 – Sauvegarde de fichiers partiels tous les 1000 documents

Nombre documents	1000	10000	20000	50000
Fusion fichiers temporaires (s)	2,19	12,36	23,65	61,06
Analyse documents d'origine (et sauvegarde temporaire) (s)	2,33	16,54	32,62	83,77
Sauvegarde sur disque (temporaire) (s)	0,73	4,37	8,78	24,47
Total (s)	4,59	28,91	56,32	144,93

TABLE 2 – Sauvegarde de fichiers partiels tous les 2000 documents

Nombre documents	1000	10000	20000	50000
Fusion fichiers temporaires (s)	2,28	12,8	25,02	62,96
Analyse documents d'origine (et sauvegarde temporaire) (s)	2,59	17,81	34,39	85,21
Sauvegarde sur disque (temporaire) (s)	0,76	4,48	9,10	24,83
Total (s)	4,93	30,61	59,47	148,30

TABLE 3 – Sauvegarde de fichiers partiels tous les 5000 documents

Nombre documents	1000	10000	20000	50000
Fusion fichiers temporaires (s)	2,47	12,66	26,05	62,63
Analyse documents d'origine (et sauvegarde temporaire) (s)	2,41	17,99	33,88	86,48
Sauvegarde sur disque (temporaire) (s)	0,74	4,61	8,91	25,82
Total (s)	4,95	30,69	59,96	149,19

TABLE 4 – Sauvegarde de fichiers partiels tous les 10000 documents

TABLE 5 – Résultats benchmarks sur 1000, 10000, 20000 et 50000 documents analysés en faisant varier la granularité de sauvegarde dans un fichier temporaire.

remarquons que les temps de sauvegarde de fichiers temporaires restent relativement constants à partir d'une granularité de 1000 documents, et qu'ils ne sont que peu dépendants du nombre de fichiers qui sont effectivement sauvegardés sur le disque, mais plus du volume total qui doit être sauvegardé. De plus, nous pouvons voir que le temps d'analyse augmente avec la granularité de sauvegarde des fichiers temporaires. Ceci peut s'expliquer par le fait que si la granularité est grande, la quantité d'information contenue dans la structure de donnée en mémoire vive aussi, et que la performance de la structure de donnée souffre de la quantité d'information massive qu'elle contient. Ainsi, en diminuant la quantité d'information dans cette structure de données, c'est-à-dire en vidant son contenu sur le disque, nous augmentons donc la performance de notre programme.

En regardant ces résultats, nous remarquons que les performances du programme diminuent avec l'augmentation de la granularité pour de grands volumes de documents (50000), mais que pour des volumes de documents plus petits, de l'ordre de 20000 ou 10000, la granularité offrant de meilleures performances est de **2000 documents**.

En effet, cela est dû en partie au fait que, pour une granularité de 2000 documents, nous atteignons, pour 20000 documents à indexer, un optimum du temps de fusion, c'est-à-dire que nous avons un nombre relativement petit de fichiers d'une taille raisonnable. De plus, les temps de sauvegarde de fichiers temporaires sont meilleurs que ceux produits avec une granularité de 1000 documents, car deux fois moins nombreux. Cependant, cette différence de temps est moins marquée si l'on augmente la granularité.

6.6 Compression

Comme nous l'avons précisé précédemment, nous avons développé, pour notre projet, une fonctionnalité de compression pour les Posting Lists triées par Id de document et groupées par mots.

Le contenu du vocabulaire a dû être modifié afin de pouvoir tenir compte du fait que la taille d'un élément d'une Posting List n'était pas constante, et que donc les offsets contenus dans le fichier vocabulaire devaient être exprimés en fonction du nombre d'octets, et non plus en fonction du nombre d'éléments de la Posting List correspondante.

Nous avons effectué nos tests sur les Posting Lists générées après analyse de la totalité de la base de données LaTimes. Les résultats sont visibles sur les tableaux 6 et 7.

Le taux de compression a été calculé via la formule :

$$\tau = Volume\ final / Volume\ initial$$

Veuillez noter que par « compression Vbyte sur le vocabulaire », il est décrit la place occupée, en octets sur le disque, par le fichier de vocabulaire adapté pour la Posting List compressée. Le vocabulaire adapté est donc un fichier texte contenant les mots du vocabulaire suivis d'un offset en octets de la Posting List compressée par rapport à celui de la Posting List du mot précédent, contrairement au fichier de vocabulaire classique qui contient un offset incrémental en

	Vocabulaire (o)	Posting List (o)	PL par score (o)
Sans compression	6155213	287494212	287494212
Gzip	2008659	69077352	63447213
Vbyte	4431088	225448345	–
Vbyte et Gzip	1496370	48525151	–

TABLE 6 – Comparaison des tailles des vocabulaires et des Posting Lists avant et après compression

	Vocabulaire	Posting List	PL par score	Total
Gzip	0.3263	0.2403	0.2207	0.2315
Vbyte	0.7199	0.7842	0.2207	0.5047
Vbyte et Gzip	0.2431	0.1688	0.2207	0.1953

TABLE 7 – Comparaison des taux de compression des vocabulaires et des Posting Lists.

nombre d'éléments de la Posting List. De plus, nous n'avons pas pu effectuer sur une compression Vbyte sur les Posting Lists triées par score, nous nous sommes alors contentés d'une compression Gzip.

Il est intéressant de noter que la compression Vbyte diminue de manière non négligeable le volume disque utilisé par les Posting Lists. Cependant, la compression offerte par la bibliothèque Gzip offre néanmoins de bien meilleures performances en termes de gain d'espace disque (taux de compression de 0.23). L'utilisation conjointe des deux méthodes permet d'obtenir des résultats encore meilleurs (taux de compression de 0.20), mais avec comme inconvénient d'engendrer des surcoûts en termes de temps d'exécution. En effet, la conjonction d'une compression Vbyte et d'une compression Gzip sur les Posting Lists triées par score et par Id ainsi que sur le vocabulaire nécessite 238 secondes.

6.7 Tentative d'optimisation sur l'indexation : Multithreading

Au cours de l'écriture du programme, nous nous sommes aperçus que nous pouvions diminuer le temps d'indexation en utilisant le Multithreading. Nous nous sommes donc concentrés sur le Multithreading de la partie analyse de documents, avec un Multithreading à l'échelle d'un journal (un fichier de LaTimes). Ainsi, chaque thread se verrait attribuer un groupe de fichier à analyser et les résultats de l'analyse seraient à la fin fusionnés.

Nous avons donc développé cette fonctionnalité, mais avons été déçus des résultats. Dans certains cas, la version multithreadée est systématiquement plus lente que la version classique, et ce, avec 2 ou 4 threads ! Ceci peut s'expliquer par la perte de performance induite par la fusion des résultats générés par chaque thread (voir table 8).

Ainsi, nous avons abandonné l'idée de multithreader notre indexation, et

Threads \Journaux	1	10	100	200	300	400
Séquentiel	0,16	1,51	13,85	27,28	41,42	55.63
Deux Threads	0,31	2,46	20,19	39,97	59,59	83,56
Quatre Threads	0,14	1,99	19,57	39,21	59,61	78,31

TABLE 8 – Temps d'exécutions, en secondes, des analyses de journaux (étape de l'indexation). Les nombres de journaux indiqués correspondent respectivement à 167, 1643, 18424, 36596, 54742, 72901 documents.

nous nous sommes ainsi focalisés sur les autres sources d'optimisations.

6.8 Optimisations sur l'indexation

La première version de notre programme ne passait pas à l'échelle, et nous avons décidé d'œuvrer afin de diminuer les temps d'indexations.

La première optimisation que nous avons faite consista en la diminution de l'utilisation de SortedDict. En effet, les SortedDict ne passent pas très bien à l'échelle pour peu qu'on les édite fréquemment. Nous avons donc identifié les utilisations abusives de ces objets et les avons remplacés par de simples dictionnaires ou par des listes.

La seconde optimisation consista en la suppression de fermetures et ouvertures de fichiers intempestives. Lors de sessions de profilage, via l'outil intégré de PyCharm, nous avons vu que la majorité du temps d'exécution du programme se déroulait dans la fonction système de fermeture de fichier et dans la fonction d'ouverture de fichier. Nous avons donc optimisé notre programme afin de fermer et de n'ouvrir un fichier qu'en cas d'absolue nécessité, tout en nous assurant que tous les fichiers soient bien fermés à la fin de l'exécution de notre programme.

6.9 Conclusion sur l'indexation

Nous avons étudié les différences de performances entre l'indexation en mémoire et l'indexation dite « partielle ». Nous pouvons donc dire que, en dépit de tous nos efforts d'optimisation, la solution d'indexation en mémoire est toujours, de loin, la plus performante (100 secondes environ contre 150 secondes pour l'indexation de 100000 documents).

Cependant, il faut également prendre en compte le fait que l'indexation en mémoire, bien que plus rapide, consomme plus de mémoire que l'indexation partielle, et qu'elle n'est, quand la quantité de mémoire du système est limitée (utilisation du swap et chute des performances), pas recommandable. Dans ce cas-là, l'indexation partielle est donc préférable, et l'étude de l'évolution du temps d'exécution nous montre qu'une granularité de 2000 documents sera optimale dans ce cas là.

De plus, si l'on souhaite optimiser le stockage de l'Inverted File sur le disque, notre programme offre la possibilité de compresser les fichiers générés avec l'algo-

rithme Vbyte (sauf pour les Posting Lists triées par score) puis avec l'algorithme Gzip, issu de la bibliothèque Python éponyme. Les résultats sont satisfaisants en termes de gain d'espace, mais au prix d'un temps de décompression/compression long. C'est la raison pour laquelle l'option n'est pas activée par défaut.

7 Comparaison des performances : Requêtes

Nous avons décidé d'implémenter l'approche disjonctive des mots et sans la possibilité de faire des «non» sur certains termes. Cela a permis de simplifier le partage de travail ainsi qu'avoir un résultat comparable entre les trois algorithmes implémentés.

le notebook contenantes les expériences réaliser dans cette partie et la suivantes est disponible a cette adresse : https ://github.com/Zed314/WordTraveller/blob/master/benchmark/Ben

7.1 Évolution des temps d'exécution en fonction du nombre document souhaité

7.1.1 Expérience réalisée

La requête est effectuée sur les mots représentés 9

Mot	'place'	'later'	'according'	'although'	'former'
Nombre de documents dans la Posting List	18312	18116	18060	18000	17996

Table 9 – Mot utilisé pour la requête de l'expérience 1

Nous avons volontairement choisi des thèmes dont les Posting List ont une taille différente afin de s'assurer que le seul critère de l'analyse soit le paramètre "k", le nombre de documents souhaité.

Pour chaque expérience, on teste chaque algorithme 10 fois pour k variant de 0 à 9. L'expérience est calculée 150 fois afin de minimiser l'intervalle de confiance. Sur le graphique 3, nous affichons la moyenne des temps d'exécutions et l'intervalle de confiance à 95%.

7.1.2 Analyse de l'expérience

On constate, dans un premier temps, que l'algorithme naïf s'exécute en un temps constant en fonction du nombre de documents attendu, ce qui correspond a nos attentes : en effet, l'algorithme naïf trie tous les documents.

On constate par ailleurs, pour cette expérience, que l'algorithme FaginsTA est plus rapide que les deux autres. Il est important de préciser que, selon les mots utilisés pour faire l'expérience, l'algorithme le plus performant peut changer. Cela dépend, entre autres, de la similitude des Posting Lists.

requete/f(k)=t.png

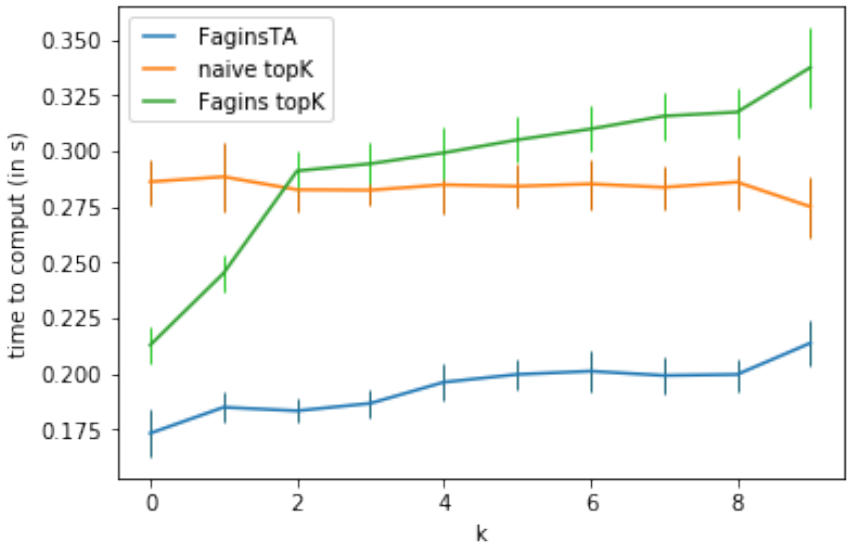


FIGURE 3 – Évolution des temps d'exécution en fonction du nombre de résultats obtenue

Finalement, le temps d'exécution nécessaire aux algorithmes non naïfs est croissant en fonction de "k". Pour une valeur de "k" élevée, on constate que l'algorithme naïf est meilleur.

7.2 Évolution des temps d'exécution en fonction de mots dans la requête

7.2.1 Expérience réalisée

Analyse faite sur 361 mots, dont les Posting Lists contiennent entre 311 et 295 documents.

Pour chaque expérience, on teste chaque algorithme 20 fois avec les "n" n^2 ("n" varie entre 0 et 20) premiers mots, issus des 361 autres. L'expérience est calculée 15 fois afin de minimiser l'intervalle de confiance. Sur ce graphique, nous affichons la moyenne des temps d'exécutions et l'intervalle de confiance à 95%.

7.2.2 Analyse de l'expérience

On constate (Fig. 4) que le temps de calcul semble évoluer de manière quadratique en fonction du nombre de mots dans la requête. Cela ne semble pas être un problème pour une utilisation classique de moteur de recherche, en revanche si pour une utilisation spécifique le nombre de la requête est amené à

requete/f(nbMots)=t.png

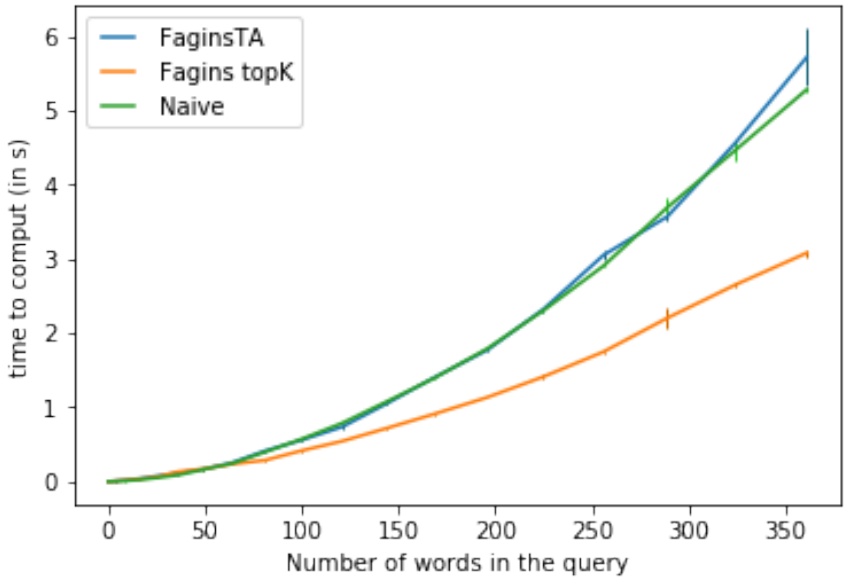


FIGURE 4 – Évolution des temps d'exécution en fonction de mots dans la requête.

augmenter, l'algorithme Fagins Top K sera plus efficace que ces concurrents.

7.3 Evolution des temps d'exécution en fonction de la taille de la Posting List

7.3.1 Expérience réalisée

Les requêtes sont réalisées en utilisant des paires de mots comme décrits dans le tableau 10 (à noter : les tailles des Posting Lists des paires sont croissantes).

Requête	Mot 1 (taille de Posting List, 'mot')	Mot 2 (taille de Posting List, 'mot')
1	(110, 'zimbabwe'),	(110, 'whalers')
2	(440, 'volunteered')	(440, 'retaliation')
3	(990, 'suicide')	(990, 'earning')
4	(1760, 'operated')	(1760, 'israel')
5	(2750, 'supported'),	(2744, 'collection')
6	(3960, 'peter')	(3959, 'separate')
7	(5389, 'sign')	(5389, 'secretary')
8	(7030, 'western')	(7030, 'situation')
9	(8904, 'car')	(8861, 'toward')
10	(10943, 'different')	(10939, 'political')
11	(13288, 'thing')	(13216, 'family')
12	(15724, 'far')	(15651, 'country')
13	(18568, 'might')	(18516, 'took')
14	(21536, 'however')	(21513, 'ago')
15	(24712, 'work')	(24596, 'say')

TABLE 10 – Requêtes Expérience 3

Pour chaque expérience, on teste chaque algorithme sur les 15 requêtes du tableau 10. L'expérience est répétée 150 fois afin de minimiser l'intervalle de confiance. K : le nombre de documents à retourner, il est fixé à 15. Sur ce graphique, nous affichons la moyenne des temps d'exécutions et l'intervalle de confiance à 95%.

requete/f(nbPL)=t.png

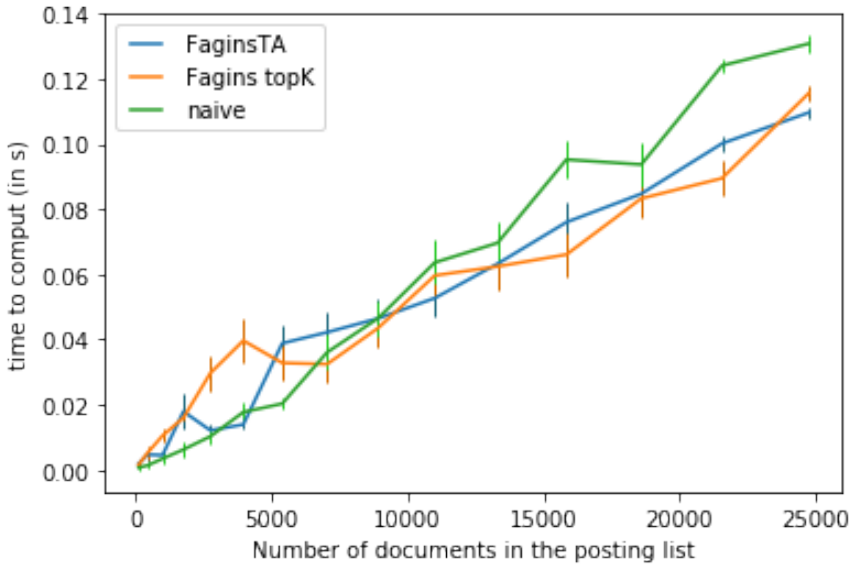


FIGURE 5 – Évolution des temps d'exécution en fonction de la taille de la Posting List

7.3.2 Analyse de l'expérience

On constate que le temps de calcul semble évoluer de manière On constate (Fig. 5) que le nombre de mot que le temps d'exécution semble linéaire par rapport a la taille des posting list. Ce pendant on constat que les courbe ne sont pas lisse. En considérant les intervalles d'incertitude, il ne semble pas possible d'approximer les point par une fonction affine. Cela s'explique par la manière dont sont calculé les temps d'exécution. En effet les pairs mots choisis peuvent être plus ou moins similaire. Or on sait que le nombre de document pressant dans une et pas dans l'autre posting lists a une influence sur les temps d'exécution. On notera que le caractère Linéaire de l'évolution du temps d'exécution par rapport a la taille de la posting list est une propriété intéressante du point de vue du passage à l'échelle, étant donné que dans l'utilisation de moteur de recherches le nombre de document indexer et par conséquent la taille des postings lists est amener à croître.

8 Évaluation de l'approximation

Nous avons implémenter une version de Fagin's TA qui permet de faire une approximation des document allant le meilleur score dans l'objectif de réduire les temps d'exécution. Cette algorithme garantie que tout les document ayant un score $1 + \varepsilon$ meilleur que le prie document renvoyer sont renvoyer. nous dans

Mots	'party'	'areas'	'health'	'player'	'sales'
Taille PL	7501	7487	7481	7470	7464
Mots	'perhaps'	'club'	'lives'	'c'	'proposed'
Taille PL	7443	7442	7429	7408	7396

TABLE 11 – Mots Expérience 4

cette partie analyser les résultat de cet Algorithme.

8.1 Expérience réalisée

Chaque requête est effectuer sur l'ensemble de mots 11

Pour chaque expérience on test chaque algorithmes 20 fois avec ε variant de 0 à 0 .3. L'expérience est calculée 200 fois afin de minimiser l'intervalle de confiance. K : le nombre de documents a retournée est fixer à 20. Sur ce graphique nous affichons la moyenne des temps d'exécutions et l'intervalle de confiance à 95%.

requete/f(e)=t.png

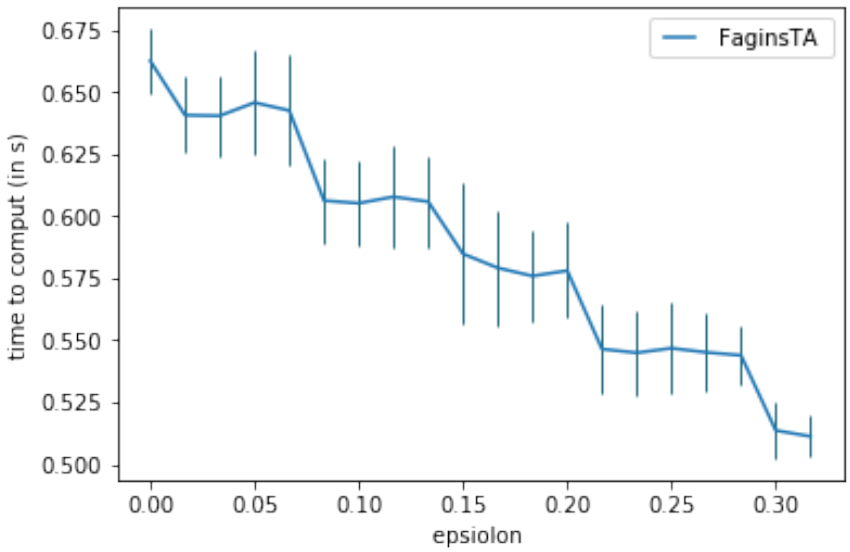


FIGURE 6 – Evolution des temps d'exécution en fonction de ε

8.2 Analyse de l'expérience

On constate fig. 6 que le temps d'exécution est décroissant par rapport à epsilon. L'utilisation de cet algorithme permet de faire une économe de 25%

du temps avec epsilon de 30% sur cet exemple. Lors d'une utilisation réelle, un juste équilibre entre vitesse et précision des résulta.

9 Recherche de synonymes

Nous avons aussi exploré la recherche de synonymes par l'approche vectorielle des mots et par la bibliothèque Word2Vec. Le but de cette recherche était d'améliorer les requêtes de mots par des mots supplémentaires extraites par l'approche vectorielle ou le module Word2Vec.

Pour l'approche vectorielle, nous avons représenté les mots par des vecteurs de 1000 dimensions. Les documents avaient aussi une taille de 1000 dimensions avec 3 à 10 non-zéros placés de manière aléatoire. Nous nous sommes rendu compte que, si nous essayions de tout indexer, les résultats de cette approche étaient assez mauvais. Alors que, si l'indexation portait plutôt sur un nombre petit de fichiers, les résultats étaient améliorés de manière non négligeable.

Pour avoir des résultats plus optimaux, nous avons implémenté une autre approche en utilisant la librairie Word2Vec, qui permet de reconstruire le contexte linguistique des mots. Nous avons pu stocker dans des fichiers un modèle entraîné à partir de tous les documents, ainsi, à chaque appel, nous pouvons lire le modèle et trouver les mots les plus similaires de notre requête.

9.1 Requêtage de synonymes

Nous avons implémenté aussi un petit module pour tester le requêtage de synonymes à partir de l'indexation par l'approche vectorielle (random indexing). Le module est appelé randomIndexingFindSynonym.py et permet à partir d'un dossier avec les indexations et un mot, avoir les n synonymes de ce mot. Paramètres :

- -d : dossier avec les fichiers .ri et .vori après l'indexation
- -f : nom de fichier .ri et .vori
- -t : requête contenant le terme (juste un terme)
- -n : nombre de termes souhaité en résultat
- –stemmer : activer Stemmer sur les termes de la requête

10 Conclusion

Nous avons donc, au cours de la réalisation de notre programme, comparé les performances d'indexation et de requêtes sur les différents algorithmes de Fagins.

Cependant, nous avons réalisé notre algorithme en Python, il pourrait être intéressant de comparer notre programme à une implémentation sur un langage de plus bas niveau, tel que le C++, afin de comparer les performances et pouvoir faire des optimisations plus fines.