Rapport Indexing and Querying Text

Dorian Lefeuvre - Lucas Marie - Fatima-Ezzahra Mezidi - Jiaye Pu
https://github.com/DoLuFaJi/Index

Général

Notre programme traite des requêtes conjonctives. Nous créons l'index de la façon suivante: pour chaque document dans un fichier nous le tokenizons en supprimant les stop-words (et stemming si souhaité). Nous comptons la fréquence de chaque terme dans un document. On associe ensuite un terme_id au terme puis nous enregistrons le tuple suivant (terme_id, doc_id, fréquence) dans une liste.
Après avoir traité X documents nous trions la liste par terme_id et doc_id, ensuite nous l'écrivons dans un fichier temporaire avant de créer une nouvelle liste. Une fois tous les documents traités nous fusionnons les fichiers temporaires dans un seul fichier binaire. Lorsque la fusion est finie nous parcourons chaque ligne du fichier binaire pour créer l'inverted file. Chaque terme est associé à un dictionnaire contenant l'index de la posting list dans le fichier binaire, la taille de posting list, et l'idf. L'idf est calculé une fois que l'inverted file est construit. Quand une posting list est chargée en mémoire, nous lisons dans le fichier binaire de l'index de la posting list jusqu'à avoir chargé l'ensemble de la posting list pour ce terme. Chaque posting list est enregistrée dans un simple mécanisme de cache.
Nous avons implémenté quatre algorithmes: Naif, Fagins, Fagins Threshold, Fagins Threshold with Epsilon.
Les algorithmes reçoivent une posting list triée par doc_id, et chacun, sauf naif, font une copie partielle de la posting list pour la trier par score. Le random accessing se fait avec une recherche binaire.

Test

Pour les tests, nous voulions trouver l'effet que les variables ont sur le temps d'exécution.

- Lors de la création de l'inverted file
  Après avoir mis en place le merge base il fallait tester l'impact du nombre de fichiers traiter à la fois. Nous avons donc créé un test qui crée des inverted file avec des tailles de batch différentes (200, 500, 1000, 10000, 25000, 50000, 75000, 100000), à la fin il nous donne un graphique qui montre le temps mit pour chaque taille.
- Pour le temps d'exécution des algorithmes
  Nous voulions tester l'efficacité de nos algorithmes.

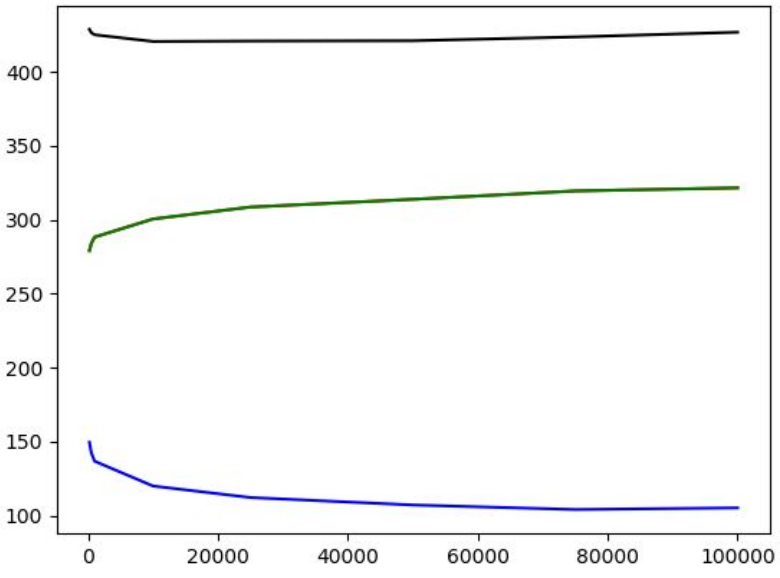
Pour avoir des valeurs représentatives, nous avons fait le choix de prendre le temps d'exécution du naïf en référence et de calculer le temps d'exécution des autres algorithmes par rapport à ce temps de référence.
Nous avons testé avec une génération de termes aléatoires: les réponses étant très souvent vides, nous avons décidé de prendre des termes plus significatifs pour avoir des résultats variés (et des résultats qui auront du sens).
Lors des tests on peut faire varier le nombre de termes, le k pour les Fagins, epsilon pour Fagins Threshold. On vérifie aussi que les résultats de chaque exécution sont juste (les scores des documents sortis sont bien égaux à ceux du naïf)
Nos tests donnent à la fin des statistiques de rapidité (algo x y% plus rapide que algo z) et des graphiques (points).

Test sur le nombre de fichiers testés à la fois

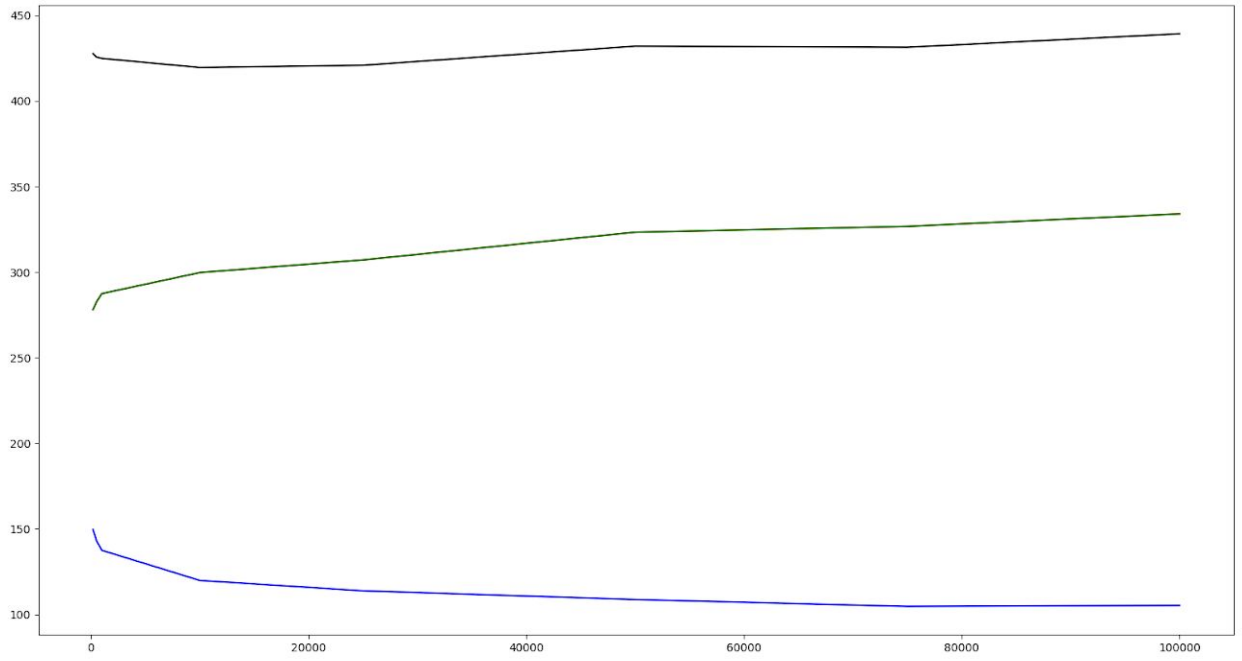
Abscisse nombre de fichiers, ordonnée temps

- En vert le temps mis lors du traitement des fichiers
- En bleu le temps mis lors du merge
- En noir le temps total

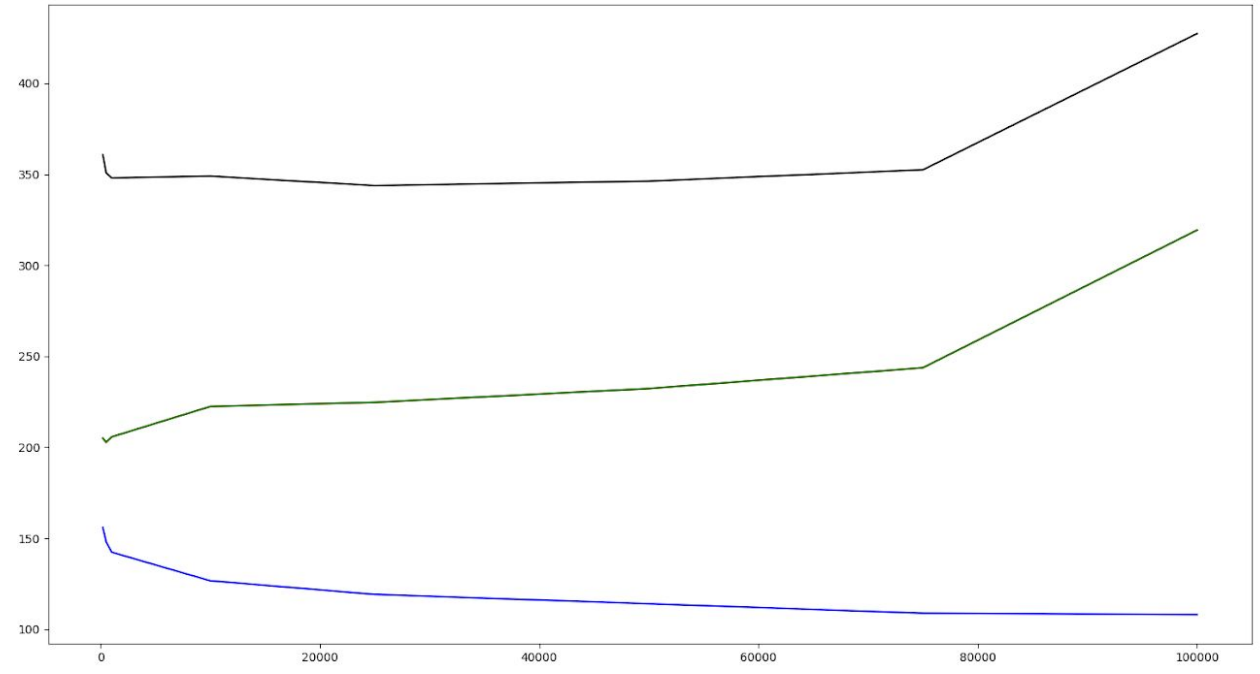
Test réalisé sur un ordinateur sans ssd



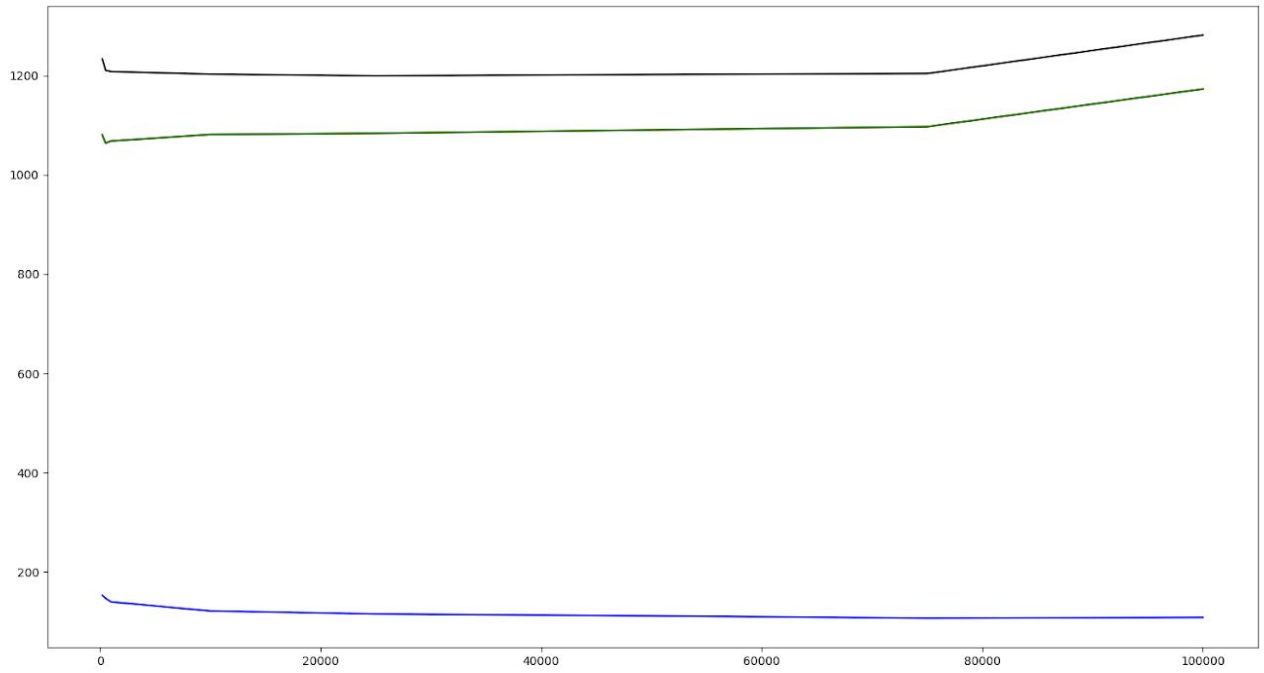
Avec stemming:



Tests réalisés sur un ordinateur avec SSD



Avec stemming



On voit bien que le temps perdu dans le traitement se récupère lors du merge
Mais si on choisi un nombre de fichiers trop élevé pour les ordinateurs avec ssd le temps d'exécution augmente très rapidement.

Test sur les algorithmes

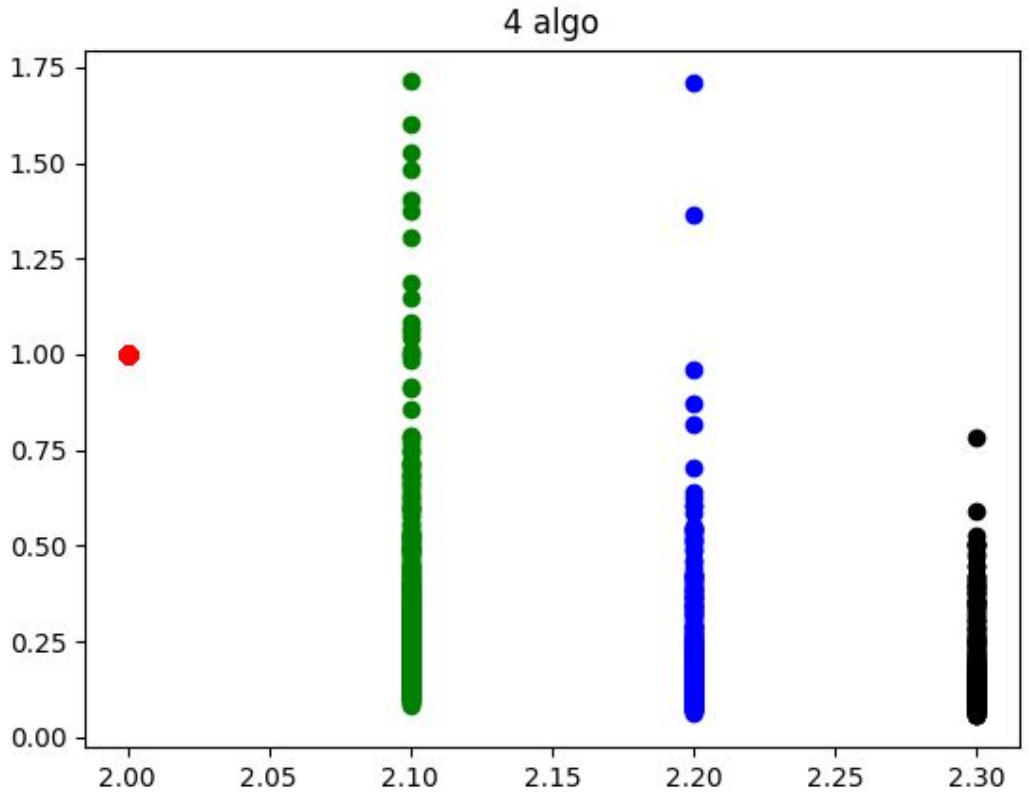
Chaque algorithme à une couleur propre sur les graphiques:

- Rouge pour naif
- Vert pour Fagins
- Rouge pour Fagins Threshold
- Noir pour Fagins Threshold avec Epsilon

Pour un soucis de lisibilité, nous décalons de 0.1 chaque trace de résultats d'algorithmes.

Comparaison des algorithmes

TEST 1 (taille de la réponse de l'algorithme naïf >50, posting list de chaque termes >500 k=1, nombre de termes=2, e0.5)



Fagins :

66% acceleration compared with naive algo.

Fagins Threshold :

81% acceleration compared with naive algo.

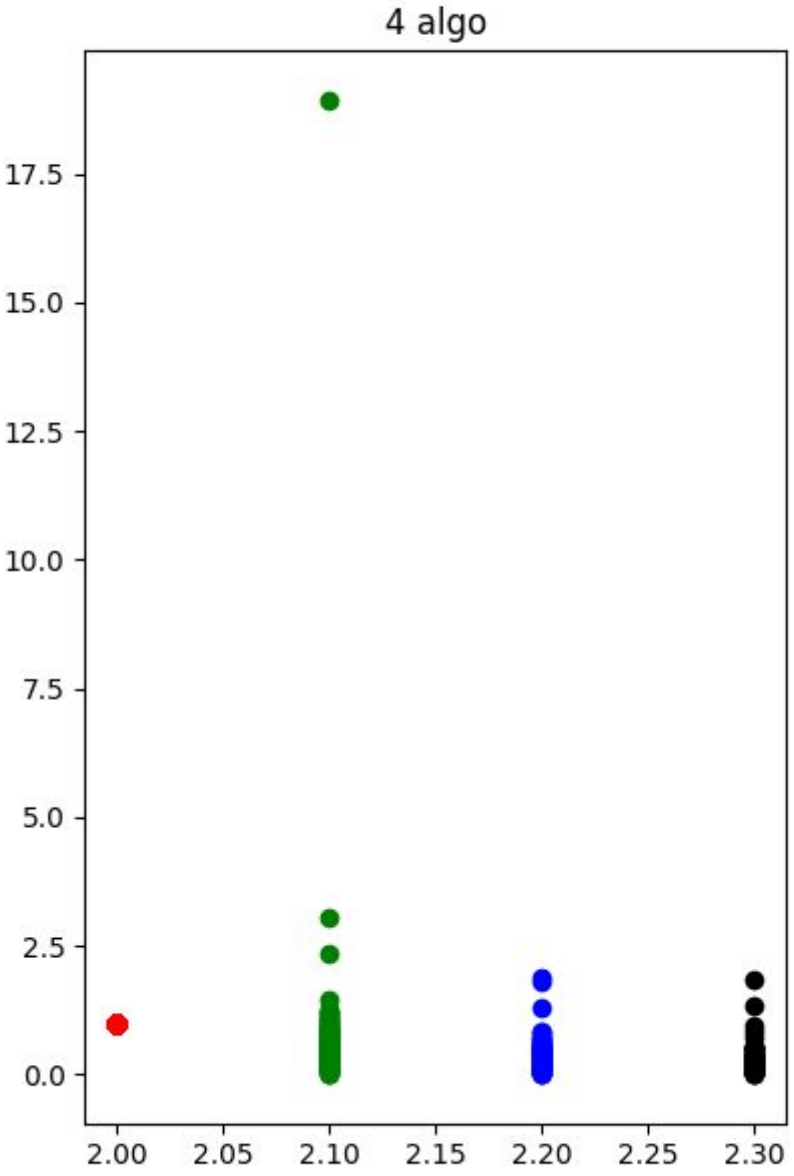
44% acceleration compared with Fagins.

Fagins Threshold With Epsilon:

85% acceleration compared with naive algo.

56% acceleration compared with Fagins.

22% acceleration compared with Fagins Threshold.



Fagins :

63% acceleration compared with naive algo.

Fagins Threshold :

82% acceleration compared with naive algo.

50% acceleration compared with Fagins.

Fagins Threshold With Epsilon:

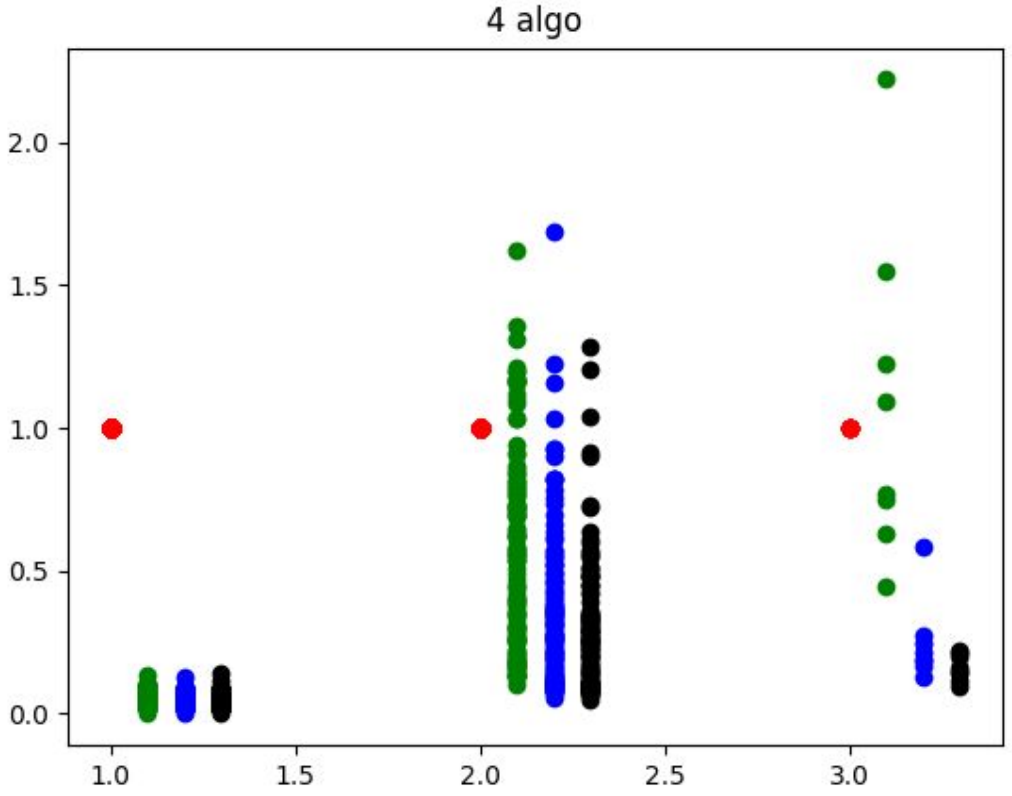
85% acceleration compared with naive algo.

60% acceleration compared with Fagins.

19% acceleration compared with Fagins Threshold.

Nos algorithmes nous font gagner beaucoup de temps comparé à l'algorithme naïf. Pour le pic >5 c'est dû au système - le garbage collector de python. Nous avons désactivé le garbage collector ensuite dans le graphe juste avant.

TEST 2 (k = 1, epsilon = 0.5, on fait varier le nombre de termes)



Fagins :

79% acceleration compared with naive algo.

Fagins Threshold :

86% acceleration compared with naive algo.

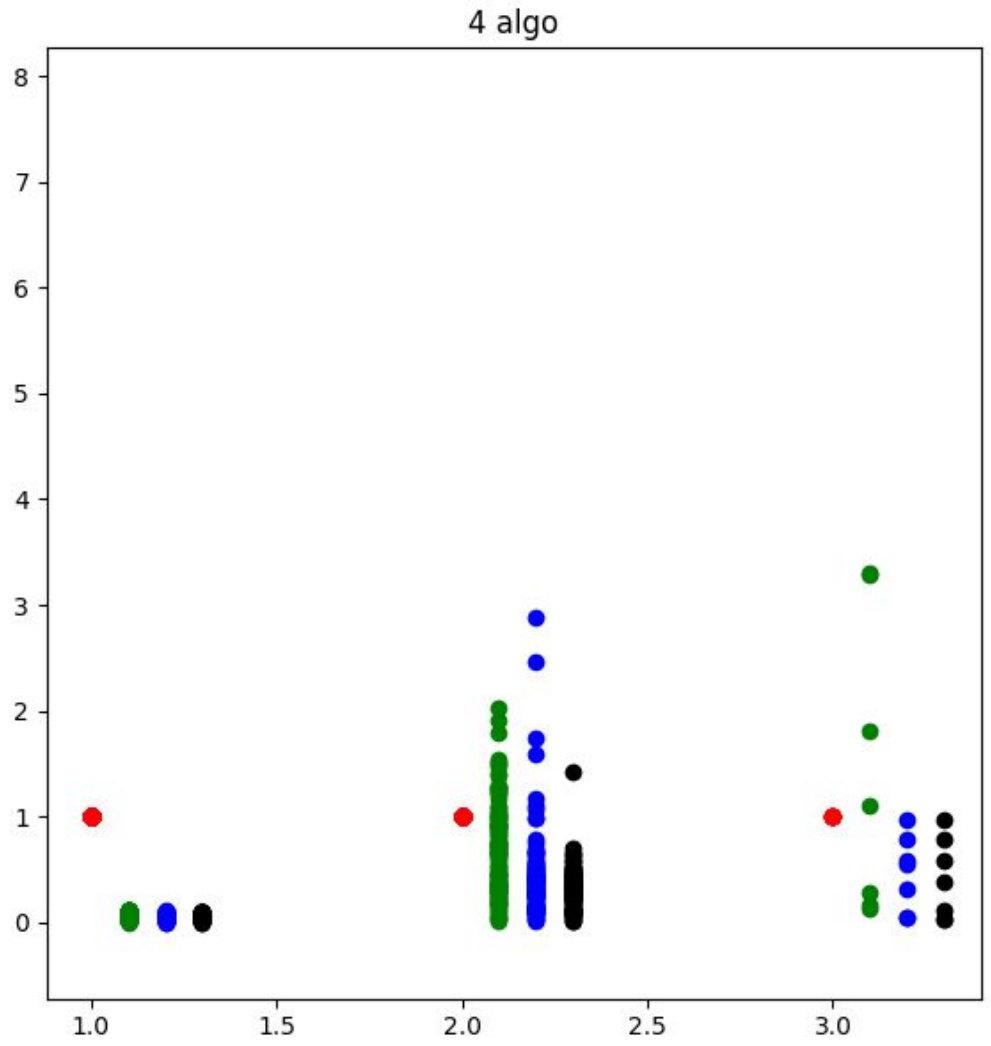
33% acceleration compared with Fagins.

Fagins Threshold With Epsilon:

88% acceleration compared with naive algo.

43% acceleration compared with Fagins.

15% acceleration compared with Fagins Threshold.



Fagins :

28% acceleration compared with naive algo.

Fagins Threshold :

84% acceleration compared with naive algo.

78% acceleration compared with Fagins.

Fagins Threshold With Epsilon:

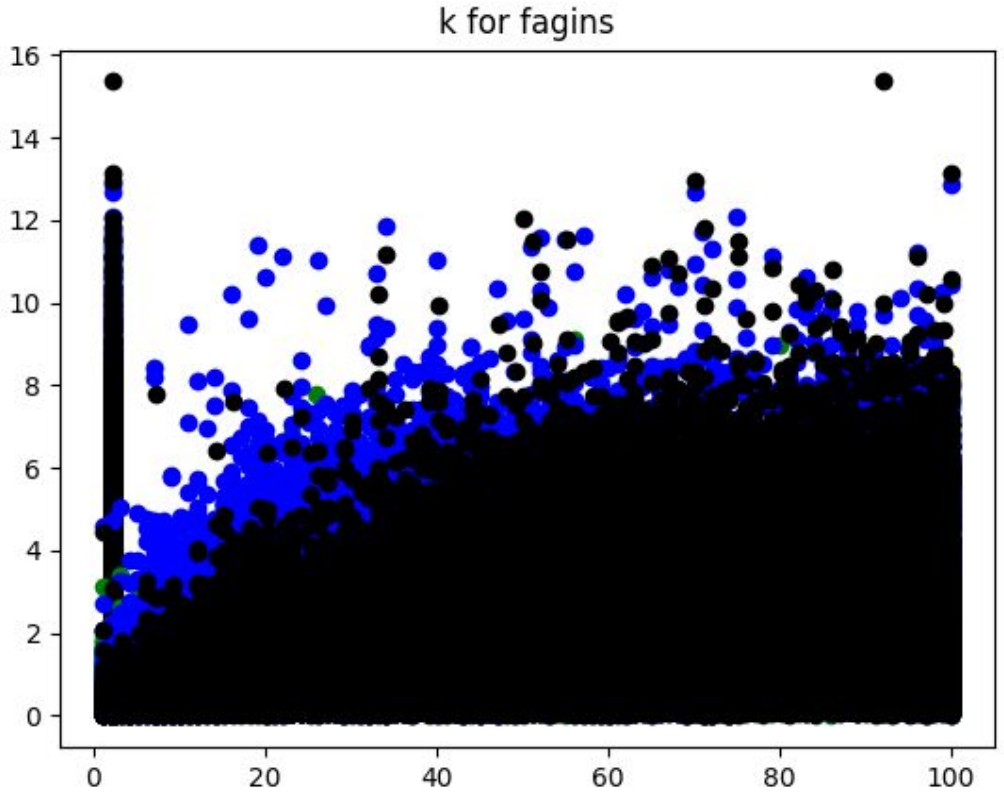
87% acceleration compared with naive algo.

83% acceleration compared with Fagins.

19% acceleration compared with Fagins Threshold.

Plus il y a de termes moins nous avons de points sur le graphe car nos requêtes sont conjonctives et nous avons mis une condition lors de l'enregistrement de points dans le graphe (taille de la réponse de naïf > 50).

TEST 3 (nombre de termes = 2, epsilon = 0.5, on fait varier k)



Naturellement quand k=0, il est inutile. Mais on remarque que lorsque k augmente, les algorithmes sont de plus en plus lents.

TEST 4 (nombre de termes = 10, k=1, on fait varier epsilon)

Epsilon à 0 pour 100 tests

Fagins :

71% acceleration compared with naive algo.

Fagins Threshold :

84% acceleration compared with naive algo.

44% acceleration compared with Fagins.

Fagins Threshold With Epsilon:

84% acceleration compared with naive algo.

43% acceleration compared with Fagins.

-1% acceleration compared with Fagins Threshold.

Epsilon à 0.25 pour 100 tests

Fagins :
- 75% acceleration compared with naive algo.

Fagins Threshold :
- 84% acceleration compared with naive algo.
- 37% acceleration compared with Fagins.

Fagins Threshold With Epsilon:
- 87% acceleration compared with naive algo.
- 49% acceleration compared with Fagins.
- 18% acceleration compared with Fagins Threshold.

Epsilon à 0.5 pour 100 tests

Fagins :
- 72% acceleration compared with naive algo.

Fagins Threshold :
- 84% acceleration compared with naive algo.
- 41% acceleration compared with Fagins.

Fagins Threshold With Epsilon:
- 86% acceleration compared with naive algo.
- 51% acceleration compared with Fagins.
- 17% acceleration compared with Fagins Threshold.

Epsilon à 0.75 pour 100 tests

Fagins :
- 76% acceleration compared with naive algo.

Fagins Threshold :
- 84% acceleration compared with naive algo.
- 37% acceleration compared with Fagins.

Fagins Threshold With Epsilon:
- 88% acceleration compared with naive algo.
- 51% acceleration compared with Fagins.
- 22% acceleration compared with Fagins Threshold.

Epsilon à 1 pour 100 tests

Fagins :

73% acceleration compared with naive algo.

Fagins Threshold :

85% acceleration compared with naive algo.

43% acceleration compared with Fagins.

Fagins Threshold With Epsilon:

88% acceleration compared with naive algo.

57% acceleration compared with Fagins.

23% acceleration compared with Fagins Threshold.

On constate que plus epsilon est grand plus l'accélération par rapport à Fagins Threshold est grande mais reste limité, 25% maximum.