Discussions et Conclusion CSI2510

Introduction :

Ce projet a pour objectif d'explorer le problème crucial de la recherche du voisin le plus proche dans un ensemble de données de grande dimension. Cette tâche revêt une importance fondamentale dans de nombreuses applications informatiques, notamment la recherche d'images similaires, la recommandation musicale, et bien d'autres domaines encore.

Dans ce projet, nous sommes amenés à concevoir et implémenter différentes versions de l'algorithme kNN (k Plus Proches Voisins) en utilisant diverses structures de données pour gérer la liste des k voisins les plus proches. Ces structures incluent une file à priorité réalisée avec un tableau trié, une file à priorité basée sur un tableau structuré en monceau, et enfin, l'utilisation de la classe standard java.PriorityQueue.

L'objectif ultime est de comparer les performances de ces différentes versions en termes de temps d'exécution sur une base de données contenant un million de vecteurs, en utilisant des requêtes spécifiées par une liste de 100 points.

Ce projet offre une occasion unique d'appliquer les concepts théoriques appris en cours, tout en mettant l'accent sur des compétences essentielles telles que la conception de structures de données, la programmation efficace et l'analyse des performances.

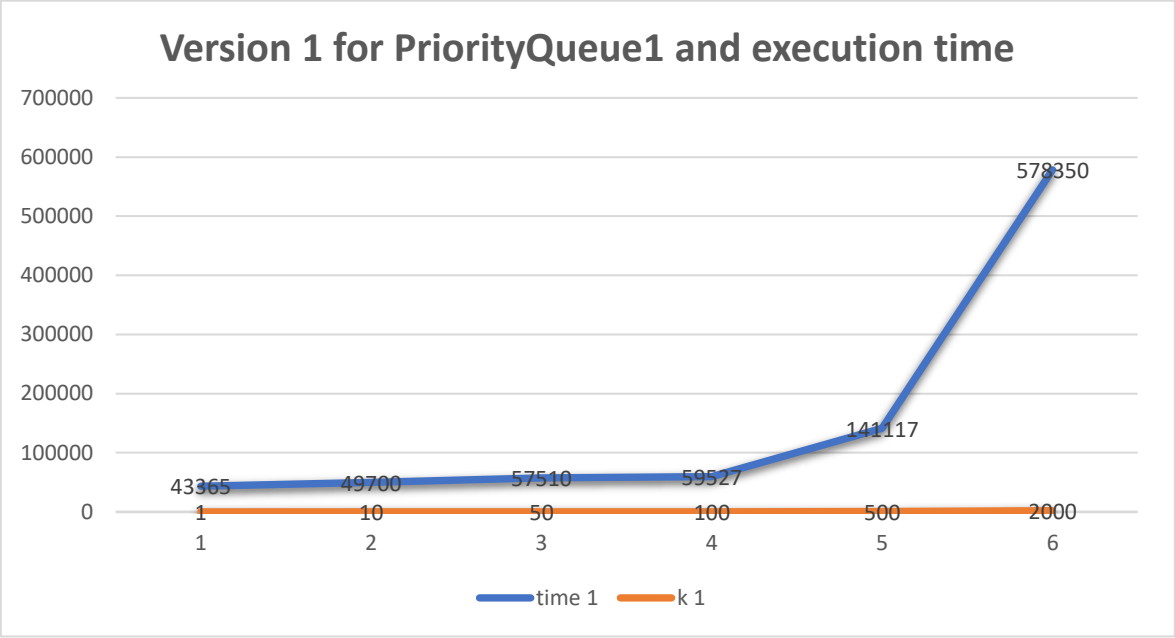
Nous allons maintenant détailler la conception et l'implémentation de chaque version de l'algorithme, ainsi que les expérimentations réalisées pour évaluer leurs performances. Enfin, nous conclurons avec une analyse des résultats obtenus et une discussion sur les implications et les applications pratiques de ces travaux.

En utilisant une base de données de 1 000 000 de points, nous avons mesuré le temps d'exécution nécessaire pour trouver les k voisins les plus proches parmi une liste de 100 points de requête. Cette expérience a été menée pour les valeurs de k suivantes : 1, 10, 50, 100, 500 et 2000, pour trois versions différentes.

Pour Run mon algorithme, j’ai utilisé un Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99 GHz et de mémoire 8 GB.

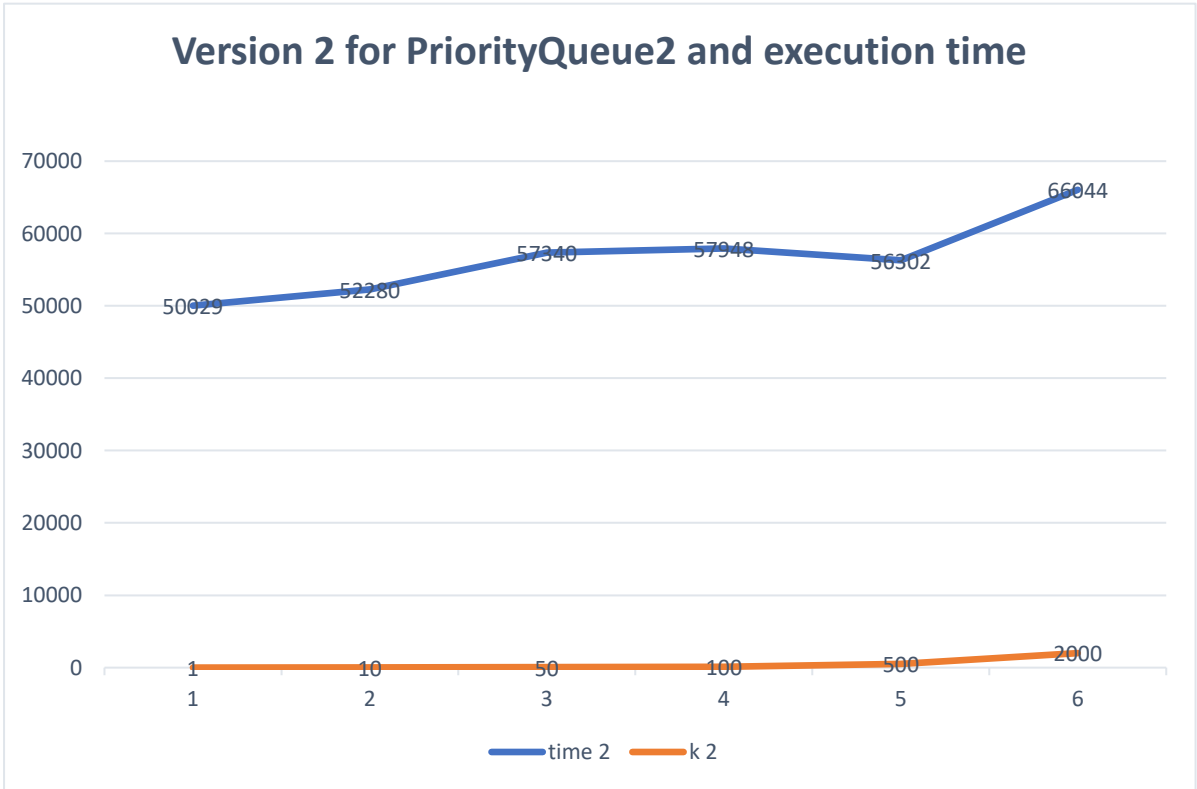
PriorityQueue1 :

La PriorityQueue1 est une implémentation de file à priorité basée sur un tableau trié. Bien qu'efficace pour de petites tailles d'entrée, sa complexité linéaire devient un inconvénient pour des ensembles de données plus importants. L'insertion, notamment, nécessitant le décalage des éléments, la rend moins optimale pour des tailles d'entrée conséquentes.



PriorityQueue2 :

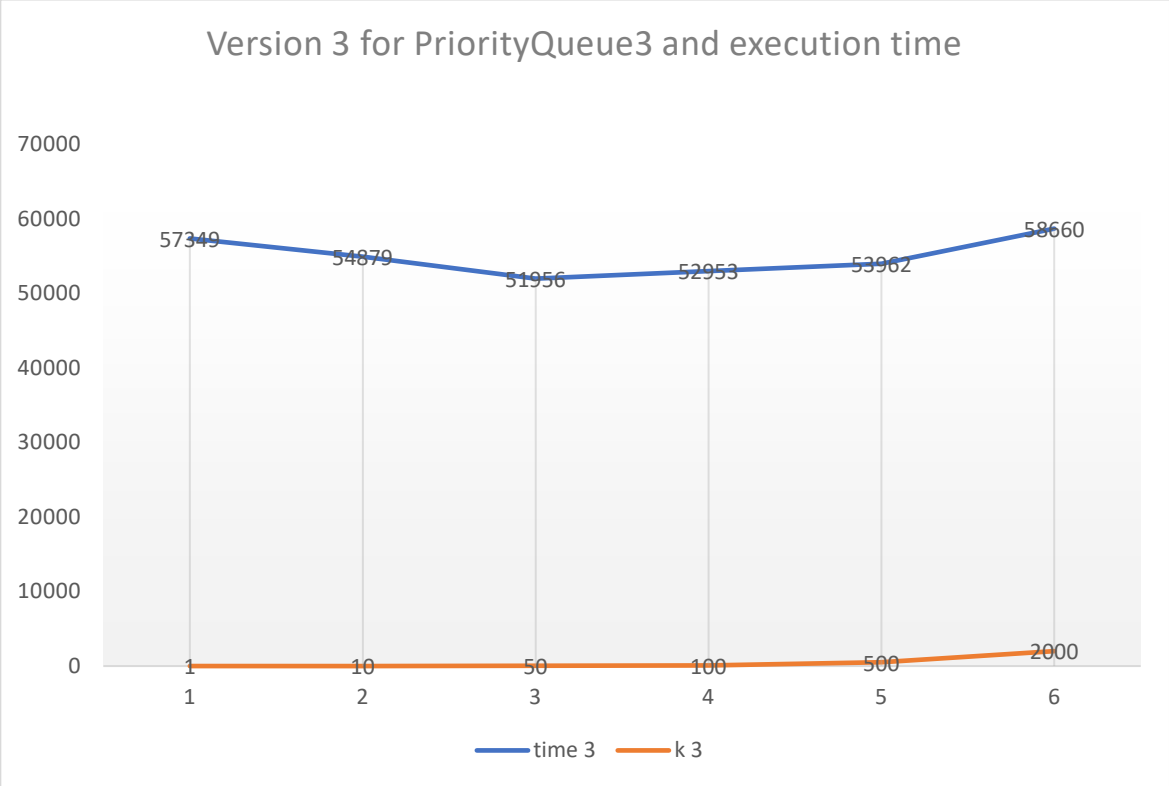
La version associée à PriorityQueue2 est une implémentation de file à priorité fondée sur un monceau utilisant un tableau. Le monceau garantit que l'élément de priorité maximale est toujours accessible rapidement, à la racine. Bien que cette méthode puisse sembler moins performante que le tableau organisé pour de petites tailles d'entrée, son avantage en termes de



complexité logarithmique devient évident avec des tailles d'entrée plus grandes, faisant de cette structure un choix optimal pour de larges ensembles de données.

PriorityQueue3 :

La version 3, correspondant à PriorityQueue3, est basée sur la classe standard java.util.PriorityQueue. Pour des tailles d'entrée plus petites, la queue prioritaire et le tableau trié peuvent présenter des performances similaires. Néanmoins, à mesure que la taille de l'entrée s'accroît, la queue prioritaire, avec sa complexité logarithmique, tend à être nettement plus performante que le tableau trié. Cette observation est en accord avec les caractéristiques intrinsèques des structures de données sous-jacentes.



Conclusion :

L'utilisation d'un tas binaire est probablement la meilleure option pour la plupart des scénarios, en particulier lorsque la taille de l'entrée est grande ou inconnue. Ces méthodes offrent une meilleure complexité algorithmique par rapport à un tableau trié et sont donc plus adaptées à des entrées plus importantes. Cependant, pour des entrées très petites, la différence de performance pourrait être négligeable, et la simplicité d'un tableau trié pourrait être privilégiée.