R-Trees

Akajou Ilias, Noé Bourgeois

March 2023

Table des matières

1 R-Trees

1.1 Contexte

Dans certains domaines, il est parfois nécessaire de connaître à quel polygon un point appartient. Pour résoudre ce problème, un algorithme du nom de Point in polygon (PIP) existe.

Cependant, cet algorithme devient moins performant dès lors que que le nombre et la complexité des polygones deviennent importants.

Pour rendre, cet algorithme plus performant, il y a plusieurs optimisations qui sont possibles :

— La première optimisation consiste en la recherche du MBR (minimum bounding rectangle), le reactangle qui englobe totalement le polygone.
— On peut encore aller plus loin, en regroupant les MBR en des ensembles proches. Pour ce faire, on a utilisé un algorithme connu sous le nom de R-Tree.

Dans ce projet, on a réalisé deux différentes variantes de cet algorithme : **quadratique et linéaire.** Dans ce projet, on réalisera plusieurs tests sur ces algorithme afin de voir leurs comportement face à la variation de différents paramètres.

Voici une explication en bref d'un R-Tree : https://en.wikipedia.org/wiki/R-tree

1.1.1 Point In Polygon

L'algorithme PIP est un algorithme qui permet de vérifier si un point se trouve ou non dans un polygone. Voici un aperçu du fonctionnement de cet algorithme :

— Pour chaque côté du polygone, vérifier s'il intersecte une ligne verticale passant par le point. Si c'est le cas, on peut l'ajouter à une liste d'intersection.
— Si le nombre d'intersections est impair, le point se trouve à l'intérieur du polygon, sinon à l'extérieur.

Cet algorithme nous est utile dans le R-tree. En effet, on utilise cet algorithme en recherchant tout les polygones qui ont un MBR qui contient ce point. On applique ensuite PIP à chacun des polygones trouvés pour déterminer si le point est réellement contenu.

1.1.2 Minimum Bounding Rectangle

Un MBR est un rectangle de taille minimum qui englobe un polygone. Les MBR, dans un R-Tree sont utilisés pour faire des recherche de point dans une zone, en éliminant plus efficacement les zones qui ne peuvent pas contenir le point.

Cela permet notamment de gagner du temps d'exécution et de préserver les performances de sa machine.

2 Explication de l'algorithme R-tree développé dans ce projet

2.1 Split Linear

Le Split Linear est un algorithme de partitionnement sur les arbres RTree.

L'algorithme consiste à répartir les noeuds du R-tree en deux sous-ensembles. Voici les étapes du fonctionnement de cet algorithme :

- On commence par calculer l'indice du milieu de la liste d'enfants RNode.
- On créé une liste (newChildren) de RNode qui contiendra les nouveaux noeuds.
- On ajoute à cette liste newChildren, la liste des enfants de RNode depuis l'index du milieu définit plus haut.
- On supprime les noeuds déjà ajouté de la liste de RNode.
- On parcours la liste de RNode qui en résulte et on étend son MBR pour inclure chaque noeud.
- On retourne finalement le nouveau RNode résultant.

Voici un pseudo-code montrant l'algorithme de Split Linear :

Algorithm 1 Split Linear

```
1: function splitLinear(rnode : Node) : RNode
2:     numChildren ← taille de rnode.getChildren()
3:     midIndex ← numChildren / 2
4:     newChildren ← une nouvelle liste de Nodes
5:     for i domidIndexnumChildren-1
6:         ajouter rnode.getChildren().get(i) à newChildren
7:     end for
8:     supprimer les éléments de rnode.getChildren() de l'indice midIndex à numChildren-1 inclus
9:     newNode ← un nouvel objet RNode créé avec newChildren et une nouvelle valeur de quantité obtenue
   à l'aide de getRNodeQuantity()
10:    rNodeQuantity ← rNodeQuantity + 1
11:    for i do0taille de rnode.getChildren() - 1
12:        rnode.getMBR().expandToInclude(rnode.getChildren().get(i).getMBR())
13:    end for
14:    for i do0taille de newNode.getChildren() - 1
15:        newNode.getMBR().expandToInclude(newNode.getChildren().get(i).getMBR())
16:    end for
17:    retourner newNode
18: end function
```

2.2 Split Quadratic

Cette algorithme prend un noeud en entré et sépare ses enfants en deux groupes de manière à ce que l'espace que deux noeuds ont en commun soit le plus petit possible. On utilise ensuite la méthode pickseeds pour choisir les noeuds initiaux qui seront placés dans sous-listes distinctes.

PickNext, est ensuite appelé pour choisir le prochain noeud à placer. Cette méthode va comparer l'overlap entre chaque noeud et choisir celui dont l'overlap est le plus faible.

Voici un pseudo code illustrant cet algorithme :

Algorithm 2 Split quadratic

```
1:  function splitQuadratic(rnodeToSplit)
2:      childrenToSplit ← rnodeToSplit.getChildren()
3:      childrenToSplitQuantity ← childrenToSplit.size()
4:      if childrenToSplitQuantity ≤ 1 then
5:          log "splitQuadratic() called with less than 2 children"
6:          return null
7:      end if
8:      seeds ← pickSeeds(childrenToSplit)
9:      node1 ← createNewRNode()
10:     node2 ← createNewRNode()
11:     node1.addChild(seeds[0])
12:     node2.addChild(seeds[1])
13:     remove seeds[0] and seeds[1] from childrenToSplit
14:     while childrenToSplit is not empty do
15:         pickNext(childrenToSplit, node1, node2)
16:     end while
17:     parent ← rnodeToSplit.getParent()
18:     if parent is null then
19:         create new root node
20:         set parent to the new root node
21:         add node1 and node2 to parent
22:         return parent
23:     else
24:         remove rnodeToSplit from parent
25:         add node1 and node2 to parent
26:         if parent.getChildren().size() ≥ maxChildren then
27:             recursively split parent node
28:         end if
29:         return null
30:     end if
31: end function
```

3 Test

3.1 Tests de variation de profondeur d'une R-tree : LINEAR

Pour ces tests, nous avons pris le Shape File de la carte du monde et nous avons recherché le point qui correspondait à la Belgique. On a donc créé un R-Tree dont on a modifié la profondeur. Les résultats ont été les suivants :

maxDepth	minDepth	Time (ms)
100	80	188
1000	800	238
10000	8000	273
100000	800	312

TABLE 1 – Variation du temps de recherche en fonction de la profondeur de la R-tree.

Voici un graphique montrant l'évolution de temps de recherche en fonction de la profondeur maximal :

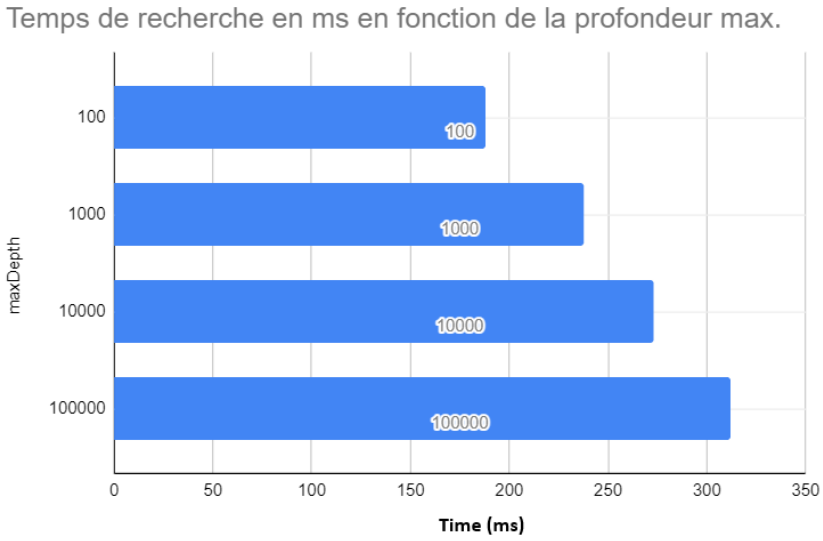

FIGURE 1 – Graphique sur le temps de recherche en fonction de la profondeur maximale

On constate, grâce à ces données générées par le code que l'on a développé, qu'une augmentation de la profondeur de l'arbre fait augmenter le temps de recherche, ce qui s'explique par le nombre de noeuds à parcourir pour procéder à la recherche. Et le temps diminue si la profondeur diminue aussi.

PS : Ces tests ont été réalisé sur un fichier Shapefile représentant la carte du monde. On a ensuite demandé à cherché le point correpondant à la Belgique (Fichier SHapeFile utilisé pour les tests.

3.2 Test d'exécution de ces algorithme sur des quantités de données importantes

Lorsque nous avons exécuter notre algorithme de rcherche dans uen R-Tree avec le split linear et quadratic, nous avons constaté plusieurs choses.

Nous supposons que la recherche utilisant le split Linear sera plus rapide pour des petites quantités de données, alors que le split quadratic lui le sera plus rapide pour de grande quantités de données. En effet, le split linear fait une division linéaire à chaque étape de la recherche. Donc pour des jeux de donneés petites, elle aura tendance à atteindre plus rapidement la valeur cible. Mais si la quantité de données augmente, le nombre d'opération linéaire augmentera lui aussi considérablement.

Pour ce qui est du split quadratic, celui-ci fait une division quadratique ce qui signifie qu'elle mettra plus de temps à atteindre la cible (qui dans ce projet et la recherche d'un point). Pour des quantités de données plus importantes, le split quadratic est le mieux adapté, car le nombre d'opération sera moins important.

3.3 Utilisation de la mémoire

Le split linear utilise moins de mémoire que le split quadratic. En effet, le split quadratic stocke plus de noeuds que le split linear lors de la recherche ce qui demande plus de ressources à la machine.

Cependant, en fonction des paramètres mis à la Rtree, les algorithmes consomment moins ou plus de mémoire.

Voici un graphique illustrant l'augmentation de la consommation de la mémoire pour un split quadratique :

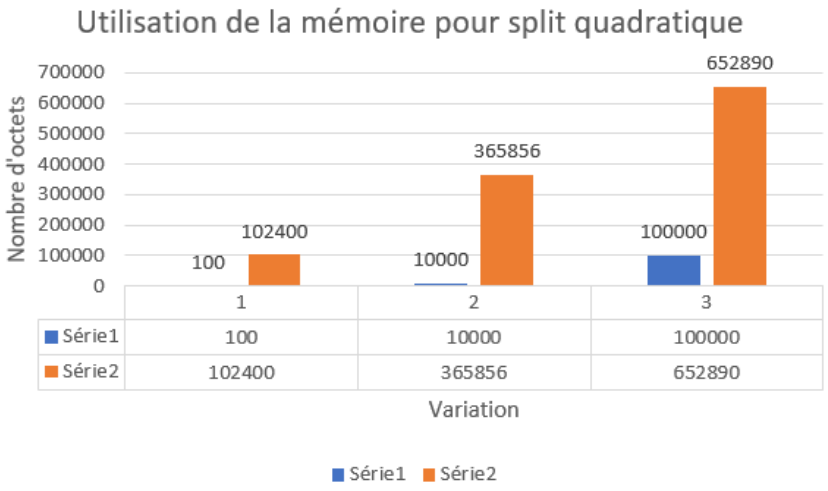

FIGURE 2 – Graphique sur l'utilisation de la mémoire en octet d'un split quadratic

On suppose aussi que si la profondeur dépasse les 10000 il est inimaginable d'estimer l'utilisation de la mémoire. En effet, cela nécessiterait des milliards d'octets que peu d'ordinateur voir aucun ne possède. De plus le temps, de recherche dans ce genre de RTree prendrait un temps très longs. *Ici on parle de mémoire secondaire.*

4 Complexité

4.1 Split Linear

La complexité d'un split linear est liée au nombre de noeuds que nous devons diviser. En effet, en toute logique si le nombre de noeuds augmente la complexité l'est aussi.

La complexité de l'algorithme développé dans ce projet est de O(n) où n est le nombre d'enfants de RNode. En effet, la boucle 'for' qui itère sur la moitié des enfants pour les ajouter à un nouvel objet 'newNodeChildren' va s'exécuter avec une complexité de O(n/2).

Ensuite, la suppression de la deuxième moitié des enfant 'rnode' se fait aussi en O(n/2).

Nous avons donc : O(n/2) + O(n/2) = O(n)

4.2 Split Quadratic

La complexité du split quadratic est de O(n$\hat{2}$).

En effet, les deux boucles qui parcourent les enfants du noeuds à scinder ont une complexité de O(n$\hat{2}$). n ici, est le nombre d'enfants du noeuds à "split"/ scinder.

Si on regarde le pire des cas, il s'agirait de tester toutes les paires de noeuds ce qui donnerait une complexité de O((n*(n - 1))/2).

4.3 Organisation de notre projet JAVA

Le projet a été effectué en utilisant Maven pour tout ce qui est gestion de dépendance.

Le dossier "src" contient tout les fichiers java à exécuter (RTree, RNode, RLeaf, ...).

Le dossier "ressources" contient tout les fichiers de tests. Il s'agit de fichier shp à exécuter avec le programme JAVA.

5 Expériences sur des données réelles

Pour ce qui est des

5.1 La librairie Geotools

5.1.1 Découpe de la Belgique en secteurs statistiques

https://statbel.fgov.be/fr/open-data?category=191

5.1.2 Pays du monde

https://datacatalog.worldbank.org/search/dataset/0038272/World-Bank-Official-Boundaries

5.1.3 Communes françaises

https://www.data.gouv.fr/fr/datasets/contours-des-regions-francaises-sur-openstreetmap/

5.2 Conclusions

Dans ce rapport scientifique nous avons effectuer plusieurs tests sur des algorithmes de RTree avec un split linear et quadratic.

Nous avons remarqué que le split quadratic est beaucoup plus performant pour de grandes quantités de données et vice-versa pour ce qui est du Split linear.

Nous avons aussi effectué plusieurs tests de mémoire, de temps, et de variation de profondeur.

6 Ressources

http://informatique.umons.ac.be/algo/redacSci.pdf

https://algs4.cs.princeton.edu/code/

https://statbel.fgov.be/fr/open-data?category=191

https://datacatalog.worldbank.org/search/dataset/ 0038272/World-Bank-Official-Boundaries

L'écriture du code a été accélérée à l'aide du plugin "Github Copilot" https://www.data.gouv.fr/fr/datasets/contours-des-regions-francaises-sur-openstreetmap/