1.5em 0pt

R-Trees

Akajou Ilias, Noé Bourgeois

March 2023

Table des matières

1 R-Trees 3
1.1 Contexte . . . 3
1.1.1 Point In Polygon . . . 3
1.1.2 Minimum Bounding Rectangle . . . 3
1.2 Structure . . . 3
1.2.1 algorithme . . . 3
1.2.2 Description . . . 3
1.2.3 Complexité . . . 3
1.2.4 Mise en oeuvre en Java . . . 3
1.2.5 Minimum Bounding Rectangle . . . 4
1.3 Création . . . 4
1.3.1 Split quadratique . . . 4
1.3.2 Split linéaire . . . 5
2 Expériences sur des données réelles 6
2.1 La librairie Geotools . . . 6
2.1.1 Découpe de la Belgique en secteurs statistiques . . . 6
2.1.2 Pays du monde . . . 7
2.1.3 Communes françaises . . . 7
2.2 Conclusions . . . 8
3 Ressources 9

1 R-Trees

1.1 Contexte

Dans certains domaines, il est parfois nécéssaire de connaître à quel polygone 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 poly- gones deviennent complexes. 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 rectangle 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 une structure de donnée connu sous le nom de R-Tree. Dans ce projet, on a réalisé deux 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.

https://en.wikipedia.org/wiki/R-tree

La recherche est définie de la façon suivante : — Pour une feuille : — Si le point appartient au MBR du nœud et le point appartient au polygone, retourner "this", — Sinon, retourner "null" ; — Pour un nœud : — Si le point appartient au MBR, tester récursivement l'appartenance pour chacun des sous-nœuds. Dès qu'un appel récursif renvoie autre chose que null, retourner ce résultat, — Sinon, retourner null ; — Pour un arbre : appeler la fonction de recherche sur la racine.

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 tous 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 ma- chine.

1.2 Structure

1.2.1 algorithme

1.2.2 Description

1.2.3 Complexité

Temps :

— Borne inférieure : $\Omega(|V| + |E|)$

— Pire cas : $\mathcal{O}(|V| + |E|)$

1.2.4 Mise en oeuvre en Java

1.2.5 Minimum Bounding Rectangle

Introduction

Description

Complexité

1.3 Création

1.3.1 Split quadratique

Cette algorithme prend un noeud en entrée 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. On choisit le nœud qui, ajouté à l'un des deux nœuds déjà choisis, maximise la différence entre les superficies des deux nœuds. On ajoute ce nœud au nœud qui maximise cette différence. On répète ce processus jusqu'à ce qu'il ne reste plus de nœuds à traiter.

Algorithme :

Voici un pseudo code illustrant cet algorithme :

```
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
```

Description

Complexité

Mise en oeuvre en Java

1.3.2 Split linéaire

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.

Algorithme :

Voici un pseudo-code montrant l'algorithme de 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
```

Description

Complexité

Mise en oeuvre en Java

2 Expériences sur des données réelles

2.1 La librairie Geotools

2.1.1 Découpe de la Belgique en secteurs statistiques

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

Tests de variation de profondeur d'une R-tree

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 :

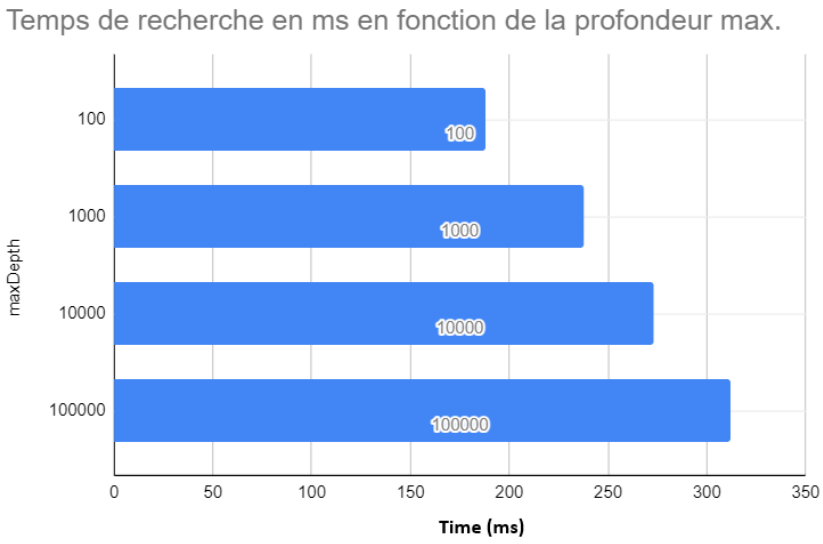

FIGURE 1 – Graphique sur le temps de recherche en fonction de la profondeur maximale

On constate, grâce à ces données générées par le code que l'on a développé, qu'une augmentation de la profondeur de l'arbre faut 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.

2.1.2 Pays du monde

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

2.1.3 Communes françaises

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

2.2 Conclusions

3 Ressources

(cf. énoncé)

Rédaction scientifique :

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

Ressources bibliographiques :

https://www.bibtex.com/.

Classes de la bibliothèque Java algs4.jar, disponible à l'adresse suivante :

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

L'écriture du code a été accélérée à l'aide du plugin https ://copilot.github.com/ ainsi que de l'IDE IntelliJ IDEA Ultimate Edition ainsi que https://chat.openai.com/?model=text-davinci-002-render-sha