Projet d'Algorithmique: R-trees

Andrius Ezerskis & Moïra Vanderslagmolen

Avril 2023

Contents

1 Introduction

Ce rapport se divise en plusieurs parties. Tout d'abord, nous expliquons la structure de notre code en expliquant le rôle de chaque classe et leurs méthodes associées. Ensuite, nous parlerons des optimisations effectuées afin d'améliorer la vitesse de nos algorithmes. Enfin, nous analyserons les tests réalisés dans le cadre de ce projet et les influences des différentes optimisations sur le temps d'exécution.

2 Structure du code

2.1 MBRNode

MBRNode représente les noeuds des R-Tree. Elle contient une "feature", un Minimum Bounding Rectangle (MBR), un polygone, des enfants (sauf si c'est une feuille) et un parent (sauf si c'est la racine de l'arbre). Si le noeud n'est pas une feuille, alors il n'aura pas de feature, et son MBR contiendra l'union de tous les MBR de ses enfants.

2.2 RTree

RTree est une classe abstraite. Elle permet de regrouper ensemble les méthodes communes à RTreeLinear et RTreeQuadratic, par exemple la recherche d'un noeud (searchNode), l'initialisation de la classe, les méthodes split, pickNext, chooseNode et addLeaf.

2.2.1 addLeaf

Lorsque nous voulons ajouter un nouveau node, la méthode addLeaf est appelée avec la racine de l'arbre en paramètre. Si la racine de l'arbre n'a pas d'enfants ou que son premier enfant n'a pas d'enfant, alors le nouveau node est rajouté à la racine de l'arbre. Ensuite, nous augmentons le MBR du parent, ainsi que celui de ses parents grâce à la méthode expandMBR. Sinon, nous choisissons un noeud grâce à la méthode chooseNode et puis la méthode addLeaf est de nouveau appelée, avec cette fois-ci le node choisi par la méthode chooseNode.

Si le nombre d'enfants du node passé en paramètre dépasse un nombre N, alors un split est effectué.

2.2.2 Split

L'algorithme de split commence par appeler le pickSeeds quadratique ou linéaire afin d'obtenir deux seeds.
Ensuite, la méthode pickNext s'assure de choisir, pour chaque sous-noeud, la seed dont le MBR augmente le moins avec ce sous-noeud, et le sous-noeud devient alors l'enfant de cette seed.
Pour finir, les deux splitSeeds obtenus via pickSeeds reçoivent comme parent le node (celui

que l'on doit split) et, réciproquement, le node reçoit les deux splitSeeds comme ses enfants. Le noeud a été divisé.

2.2.3 chooseNode

La méthode chooseNode prend en paramètre le bestNode et le noeud à ajouter. Si le bestNode n'a pas d'enfants ou si son premier enfant n'a pas d'enfants, alors le bestNode est renvoyé. Sinon, nous itérons à travers les enfants de bestNode. Nous choisissons le noeud qui minimise l'aire de l'union de ce noeud avec le noeud à ajouter. Nous appelons ensuite récursivement la méthode chooseNode.

2.2.4 Search

La fonction search commence à partir de la racine. Si la racine n'a pas d'enfants, si le MBR de la racine et si le polygone de la racine contiennent le point, alors la racine est retournée. Sinon, si la racine a des enfants et si son MBR contient le point cherché, alors nous itérons dans les enfants et appelons la méthode search sur chaque enfant.

2.3 RTreeLinear et RTreeQuadratic

RTreeLinear représente l'implémentation du R-Tree avec l'algorithme de split linéaire et RTreeQuadratic représente le R-Tree avec l'algorithme de split quadratique.

2.3.1 pickSeeds quadratique

Le pickSeeds quadratique cherche les deux seeds qui maximisent l'aire de l'enveloppe qui les entoure. Pour ce faire, nous itérons à travers tous les sous-noeuds du noeud donné en paramètre, et nous calculons la différence d'aire entre l'union de deux sous-noeuds consécutifs et l'aire de ces derniers séparement. Le pickSeeds retourne donc un vecteur contenant les deux seeds choisies.

La complexité de cette méthode est $\mathcal{O}(n)$.

2.3.2 pickSeeds linéaire

Le pickSeeds linéaire commence par itérer dans chaque sous-noeud du noeud que nous voulons split. Ensuite, nous choisissons le noeud dont le côté droit du MBR associé a le plus petit x, le noeud dont le côté gauche du MBR a le plus grand x. Nous faisons de même avec la dimension y. Par après, nous testons si les nodes sont tous les mêmes, ou si les noeuds d'une dimension sont les mêmes. Dans le premier cas, nous renvoyons null et le split ne se produit pas. En effet, si les seeds trouvées sont les mêmes, alors le split ne se produira pas correctement car tous les noeuds iront donc dans la première seed. Dans le deuxième cas, nous choisissons l'autre dimension dont les seeds ne sont pas les mêmes.

Si les noeuds sont tous différents, nous calculons la normalisation, qui correspond au ratio entre les côtés internes et les côtés externes pour chaque dimension (dans ce projet-ci, la dimension x et y). Nous prenons ensuite la normalisation la plus élevée.

La complexité de cette méthode est $\mathcal{O}(n)$.

2.4 FileLoader

Cette classe permet de charger le shapefile en mémoire. Nous l'avons implémentée afin de facilement changer la façon dont les fichiers sont chargés.

3 Optimisations

3.1 Multi-polygones

Notre première optimisation se base sur les multi-polygones. En effet, comme expliqué dans les consignes du projet d'algorithmique, nous avons remarqué que des polygones très étendu comme la France (voir Before optimization) prenaient énormément de place. Nous avons donc décidé de séparer les multi-polygones en polygones (voir After optimization). Cela nous a permis de gagner quelques secondes lors des tests dont nous parlerons au chapitre Analyse des tests.

3.2 pickSeeds quadratique

Dans le pickSeeds quadratique, nous itérons à travers le vecteur d'enfants du node à split et nous prenons les deux noeuds les plus éloignés. Nous avons vite remarqué qu'il n'était pas nécéssaire de faire un double for en itérant à travers chaque élément dans le vecteur. En effet, si nous calculons l'aire pour le noeud A et pour le noeud B, il ne faut pas recalculer pour le noeud B et le noeud A, de même qu'il ne faut pas calculer l'aire pour le noeud A et le noeud A. Nous faisons donc deux boucles for, l'une itérant dans le vecteur entier de sous-noeuds, l'autre commençant à l'indice de la précédente boucle + 1. De cette manière, nous commençons avec le noeud A et calculons avec le noeud B et tous les autres noeuds restants, puis le noeud B avec le noeud C et ainsi de suite. Cela nous permet de réduire le temps d'exécution.

3.3 Nombre d'enfants maximum

Au début, nous avions N (le nombre d'enfants maximum pour les RTree) en attribut de la classe RTree. Afin d'optimiser notre programme, nous avons décidé de le passer en paramètre. Grâce à cette amélioration, les 8 tests que nous avons fait tourner ont pris 4,429 secondes au total au lieu de 5,865 secondes.

4 Expérience sur données réelles

4.1 Présentation des tests

1. Carte de la belgique - Algorithme linéaire - Campus universitaire
2. Carte de la belgique - Algorithme linéaire - Point pas dans le polygone
3. Carte du monde - Algorithme linéaire - Kazakhstan
4. Carte du monde - Algorithme linéaire - Canada
5. Carte de la france - Algorithme linéaire - Auvergne
6. Carte de la france - Algorithme linéaire - Guyane
7. Carte de la belgique - Algorithme quadratique - Campus universitaire
8. Carte de la belgique - Algorithme quadratique - Point pas dans le polygone
9. Carte du monde - Algorithme quadratique - Kazakhstan
10. Carte du monde - Algorithme quadratique - Canada
11. Carte de la france - Algorithme quadratique - Auvergne
12. Carte de la france - Algorithme quadratique - Guyane
13. Carte du japon - Algorithme linéaire - JP21042
14. Carte du japon - Algorithme quadratique - JP21042

4.2 Présentation des cartes

1. Carte de la belgique: 19795 polygones
2. Carte du monde: 251 polygones
3. Carte de la france: 18 polygones
4. Carte du japon: 1892 polygones

Ces tests ont été effectués sur une machine Lenovo Yoga 7. La distribution linux est KDE Neon 5.27 et la release ubuntu est la 22.04.

4.3 Résultats des tests

Temps en millisecondes de la fonction search en fonction des différents tests et valeurs.

Temps en millisecondes de la fonction search			
Test	N=4	N = 500	N = 10000
Test1	15	15	17
Test2	1	1	1
Test3	76	8	11
Test4	123	214	219
Test5	23	68	-
Test6	3	3	-
Test7	2	0	2
Test8	1	1	4
Test9	3	2	2
Test10	29	76	73
Test11	29	37	-
Test12	2	2	-
Test13	2	3	2
Test14	1	1	1

Temps en millisecondes de la fonction search avec l'optimisation			
Test	N=4	N = 500	N = 10000
Test1	12	14	19
Test2	1	1	6
Test3	11	12	9
Test4	20	28	23
Test5	99	95	-
Test6	3	2	-
Test7	3	0	2
Test8	0	3	2
Test9	1	2	2
Test10	5	5	6
Test11	29	19	-
Test12	2	2	-
Test13	0	1	1
Test14	1	0	0

4.4 Analyse des tests

4.4.1 Optimisation

Nous avons donc remarqué que le test 4 prenait beaucoup de temps, et plus généralement les tests concernant la carte du monde (Tests 3, 4, 9, 10) et les tests concernant la carte de la france (Tests 5,6,11,12). Ces tests devraient être les plus rapides, car ils contiennent le moins de polygones, et les algorithmes sont quadratique et linéaire. Nous avons compris que le problème venait du fait que les multi-polygones prenait énormément de place et nous avons refait les tests.

4.4.2 Influence du N

Nous remarquons aussi que le nombre N (nombre maximum d'enfants) n'a pas d'influence notable sur la rapidité de notre algorithme.

4.4.3 Algorithme linéaire vs Algorithme quadratique

Nous avons aussi remarqué que l'algorithme linéaire prend beaucoup plus de temps à chercher un point dans un polygone. En moyenne, la recherche prend 39,9 millisecondes pour l'algorithme linéaire et 13,25 millisecondes pour l'algorithme quadratique.

Nous avons donc comparé les temps d'exécution. En moyenne, le temps d'exécution sur 6 tests pour l'algorithme quadratique est de 6,357 secondes. Le temps d'exécution sur les mêmes tests pour l'algorithme linéaire est de 3,087 secondes. La création de l'arbre est donc beaucoup plus rapide pour l'algorithme linéaire que pour le quadratique, ce qui est attendu vu la complexité de chaque algorithme. Cependant, il est intéressant de noter que la fonction de recherche prend beaucoup plus de temps dans l'algorithme linéaire.

5 Conclusion

En conclusion, nous avons remarqué une différence notable de la fonction search entre l'algorithme linéaire et quadratique, l'algorithme quadratique étant beaucoup plus rapide. Cependant pour la création du R-Tree, l'algorithme linéaire est bien plus rapide. Nous avons également optimisé notre programme grâce à la décomposition de multi-polygones en polygones.

6 Annexes

References

[Al-Badarneh et al., 2010] Al-Badarneh, A. F., Yaseen, Q., and Hmeidi, I. (2010). A new enhancement to the r-tree node splitting. *Journal of Information Science*, 36(1):3–18.

[Guttman, 1984] Guttman, A. (1984). R trees: A dynamic index structure for spatial searching. volume 14, pages 47–57.

Figure 1: Before optimization

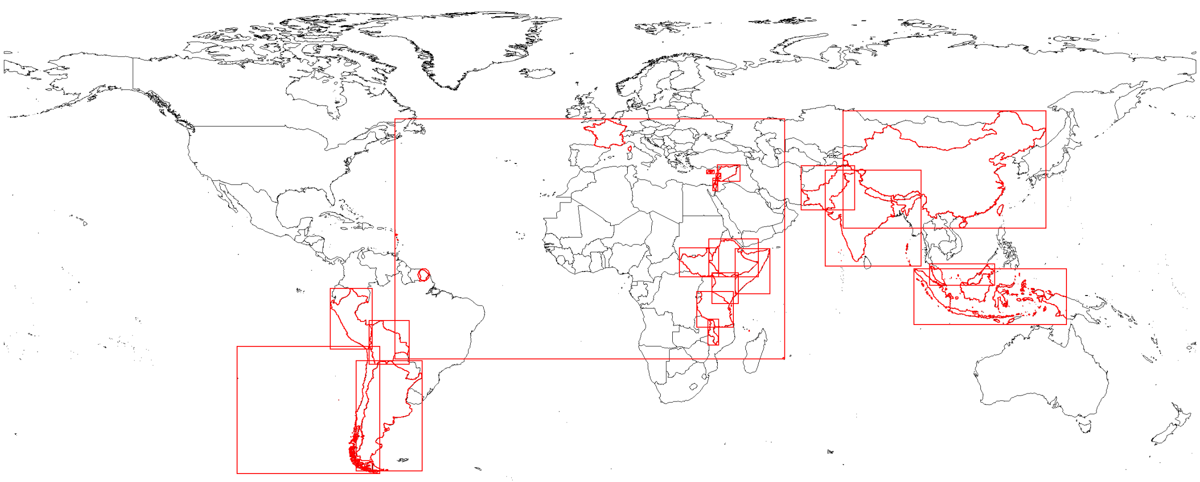

Figure 2: After optimization