Université Libre de Bruxelles

INFO-F203

Algorithmique 2

Projet : R-trees

Nieto Navarrete Matias , 502920

Laakel Oussama , 516197

B-INFO

Avril 2023

Table des matières

1 Introduction

1.1 Contexte et enjeu

De nos jours, avec le développement rapide des technologies géographiques et l'augmentation des données spatiales, il est important de pouvoir stocker, organiser et manipuler ces données efficacement. Les R-Trees, introduits par A. Guttman en 1984 [1], sont des structures de données qui nous aident à résoudre ce problème en facilitant les opérations de recherche, d'insertion de données géographiques. Ils sont souvent utilisés dans divers domaines comme la cartographie, la navigation et la géolocalisation.

Cependant, les différentes variantes de R-Trees, en particulier les variantes quadratiques et linéaires, montrent des différences notables en matière de performances lors de la création et de la recherche de points.

1.2 Objectifs du projet

L'objectif principal de ce projet est de développer une implémentation des R-Trees adaptée à la gestion de polygones géographiques et d'étudier les performances des différentes variantes de R-Trees, en s'appuyant sur les travaux d'A. Guttman [1]. Les objectifs spécifiques de ce projet sont les suivants :

— Comprendre les notions clés des R-Trees, leurs variantes et les opérations liées.
— Effectuer une série d'expériences pour évaluer les performances de l'implémentation développée en termes de temps de construction, de recherche de points.
— Comparer les performances des variantes quadratiques et linéaires de R-Trees à travers diverses expériences.
— Analyser les résultats obtenus.

Dans ce rapport, nous abordons les points suivants : la section 1 introduit le contexte et les objectifs du projet. La section 2 expose les concepts clés et les différentes variantes des R-Trees. La section 3 traite de l'implémentation des R-Trees. La section 4 explore en détail les aspects de l'implémentation, tels que l'insertion, les splits linéaires et quadratiques, ainsi que les méthodes de recherche. La section 5 évalue les performances de l'implémentation, et enfin, la section 6 offre une conclusion et présente les perspectives d'évolution future.

2 R-Trees : Concepts et variantes

2.1 Concepts de base

Les R-Trees sont des structures de données en arbre, destinées à faciliter les recherches spatiales sur des points et des polygones. Ils sont conçus pour permettre une recherche efficace dans des espaces à plusieurs dimensions et pour gérer des recherches de points. Voici quelques concepts clés des R-Trees :

— **Nœud** : Chaque nœud représente un rectangle englobant minimal (MBR - Minimum Bounding Rectangle) qui inclut des objets géographiques ou d'autres MBR. Un nœud contient une liste de descendants, qui peuvent être d'autres nœuds ou des feuilles, ainsi qu'un MBR représentant l'union de tous les MBR de ses descendants.
— **Rectangle englobant minimal (MBR)** : Un MBR est un rectangle qui englobe complètement un ensemble d'objets géographiques avec la plus petite surface possible. Les MBR sont utilisés pour représenter les limites des objets spatiaux dans un R-Tree.
— **Recherche** : La recherche dans un R-Tree consiste à parcourir l'arbre en suivant les MBR qui intersectent la région de recherche jusqu'à ce que les objets géographiques appropriés soient trouvés.

— **Insertion** : L'insertion d'un nouvel objet géographique dans un R-Tree se fait en parcourant l'arbre pour trouver le nœud le plus approprié pour accueillir le nouvel objet, puis en mettant à jour les MBR et en ajustant la structure de l'arbre si nécessaire.

2.2 Variantes des R-Trees

Au fil des années, plusieurs variantes des R-Trees ont été proposées pour améliorer leur performance et leur adaptabilité. Les deux variantes les plus courantes sont les R-Trees quadratiques et linéaires, qui diffèrent principalement par les heuristiques utilisées pour répartir les objets entre les nœuds lors des opérations d'insertion :

— **R-Tree quadratique (Quadratic R-Tree)** : Cette heuristique met en œuvre un algorithme quadratique pour déterminer la répartition optimale des objets entre les nœuds lors des opérations d'insertion et de suppression. L'algorithme cherche à minimiser le chevauchement et l'expansion totale des MBR des nœuds, ce qui peut entraîner une amélioration des performances lors des recherches.

— **R-Tree linéaire (Linear R-Tree)** : L'heuristique choisit deux éléments les plus éloignés pour les répartir dans deux nœuds distincts lors de l'insertion. Par la suite, chaque nouvel objet est inséré dans le nœud dont le MBR aurait la plus petite augmentation de surface en cas d'insertion. Cette approche linéaire vise à minimiser le chevauchement et la taille des MBR, mais peut entraîner un déséquilibre de la hauteur de l'arbre et une moins bonne performance globale par rapport à l'heuristique quadratique.

Mis à part les R-Trees quadratiques et linéaires, d'autres variantes de R-Trees sont décrites dans le livre R-Trees : Theory and Applications [2], telles que les R*-Trees, les Hilbert R-Trees et les Packed R-Trees. Néanmoins, pour ce projet, nous nous concentrons sur les R-Trees quadratiques et linéaires.

3 Implémentation des R-Trees

Dans cette implémentation des R-Trees, plusieurs classes ont été créées pour construire et manipuler un R-Tree. Les classes principales sont Node, Leaf, RectangleTree, QuadraticRectangleTree et LinearRectangleTree.

3.1 La classe Node

représente un nœud interne dans un R-Tree. Chaque nœud interne stocke un objet Envelope, qui représente le Minimum Bounding Rectangle (MBR), ainsi qu'une liste de sous-nœuds et un parent. Des méthodes pour ajouter un sous-nœud et mettre à jour le MBR sont également fournies.

3.2 La classe Leaf

hérite de la classe Node et représente un nœud feuille dans un R-Tree. Chaque nœud feuille stocke un label et un objet SimpleFeature qui représente un polygone. Des getters sont mis en place pour accéder aux attributs tels que le label et le polygone associé. Cette classe permet de gérer les nœuds feuilles spécifiques dans la structure de l'arbre.

3.3 La classe RectangleTree

est une classe abstraite qui représente un R-Tree générique. Cette classe fournit des méthodes pour insérer des éléments dans l'arbre, ajuster la structure de l'arbre après l'insertion, choisir le nœud approprié pour l'insertion, diviser un nœud en deux et rechercher un point dans l'arbre. De plus, elle définit des méthodes abstraites pour sélectionner les graines pour la division et calculer le coût d'expansion

3.4 La classe QuadraticRectangleTree

étend la classe RectangleTree et implémente un R-Tree avec l'algorithme de division quadratique. Cette classe fournit des implémentations pour les méthodes abstraites de la classe RectangleTree, telles que pickSeeds() et calculateCost().

3.5 La classe LinearRectangleTree

étend également la classe RectangleTree et implémente un R-Tree avec l'algorithme de division linéaire. Tout comme QuadraticRectangleTree, cette classe fournit des implémentations pour les méthodes abstraites de la classe RectangleTree.

Ces classes travaillent ensemble pour construire et manipuler un R-Tree efficacement.
Les classes QuadraticRectangleTree et LinearRectangleTree sont les deux heuristiques des R-Tree.

4 Description de l'implémentation

L'implémentation du R-Tree présentée ici s'appuie sur les concepts décrits dans l'article d'A. Guttman [1] et les classes mentionnées précédemment. Dans cette section, nous expliquons les détails de l'implémentation des méthodes d'insertion, de division et de recherche pour les R-Trees linéaires et quadratiques.

4.1 Insertion

L'insertion d'un nouvel élément dans un R-Tree suit les étapes qui sont dans l'algorithme décrit par Gutmann en 1984 [1] :

Recherche du nœud d'insertion (chooseNode) : À partir de la racine de l'arbre, la méthode chooseNode() parcourt les nœuds en descendant l'arbre pour trouver le nœud approprié où insérer le nouvel élément. Cette méthode cherche à minimiser l'expansion des MBR en choisissant le sous-nœud dont l'élargissement pour le polygone est minimal. La complexité de cette méthode est O(log n), où n est le nombre de polygones dans l'arbre.

Ajout de l'élément (addLeaf) : Une fois le nœud d'insertion trouvé, l'élément est ajouté à ce nœud via la méthode addLeaf(). Cette méthode n'est pas récursive car elle ajoute simplement un élément dans le nœud donné, sans avoir à parcourir l'arbre. La complexité de cette méthode est O(1), car elle insère simplement un élément dans le nœud.

Vérification de la capacité et ajustement de l'arbre (adjustTree) : Si l'ajout de l'élément fait dépasser la capacité maximale du nœud, la méthode split() est appelée pour diviser le nœud. La méthode adjustTree() est utilisée pour mettre à jour les MBR des nœuds et propager les modifications à travers l'arbre. Cette méthode est récursive et permet d'ajuster l'arbre en cas de division d'un nœud, en remontant jusqu'à la racine si nécessaire. La complexité de cette méthode est également O(log n), car elle parcourt l'arbre en remontant de manière récursive.

Algorithm 1 Insertion d'une caractéristique dans l'arbre

```
1: function INSERT(label, feature)
2:     if root.isEmpty() then
3:         Leaf newLeaf = new Leaf(label, feature)
4:         root.addLeaf(newLeaf)
5:     else
6:         Node chosenNode = chooseNode(root, feature)
7:         Node newNode = addLeaf(chosenNode, label, feature)
8:         adjustTree(chosenNode, newNode)
9:     end if
10: end function
```

Ainsi, la complexité totale de l'insertion ci-dessus 1 dans un R-Tree est O(log n).

4.2 Split linéaire et quadratique

Les algorithmes de division linéaire et quadratique sont implémentés respectivement dans les classes LinearRectangleTree et QuadraticRectangleTree. Leur but est de répartir les éléments d'un nœud surchargé entre deux nouveaux nœuds.

1. **Split linéaire** : L'algorithme de division linéaire comme le décrit Guttman[1], commence par la méthode pickSeeds(), qui choisit deux éléments les plus éloignés pour les répartir dans deux nœuds distincts. Dans cette implémentation, la distance entre les éléments est mesurée en utilisant la surface des MBR combinés. Une fois les graines sélectionnées, la méthode pickNext() est utilisée pour attribuer chaque élément restant à l'un des deux nœuds. Le coût d'insertion, calculé par La méthode calculateCost(), la méthode calculateCost(), est basé sur l'augmentation de la surface du MBR du nœud en cas d'insertion. L'élément est inséré dans le nœud avec le coût le plus faible. La complexité de l'algorithme split linéaire est $O(n)$, où n est le nombre d'éléments dans le nœud.

Algorithm 2 Pickseed pour l'heuristique linéaire

```
1: function PICKSEEDS(subnodes)
2:     Initialiser maxDiffX, maxDiffY, indexX1, indexX2, indexY1, indexY2
3:     for i ← 0 to taille(subnodes) − 1 do
4:         Calculer diffX et diffY pour subnodes[i]
5:         if diffX > maxDiffX then
6:             Mettre à jour maxDiffX, indexX1, indexX2
7:         end if
8:         if diffY > maxDiffY then
9:             Mettre à jour maxDiffY, indexY1, indexY2
10:        end if
11:    end for
12:    if maxDiffX > maxDiffY then
13:        Retourner [indexX1, indexX2]
14:    else
15:        Retourner [indexY1, indexY2]
16:    end if
17: end function
```

Le pseudo-code ci dessus 2 montre comment sélectionner les deux éléments les plus éloignés en termes de surface des MBR combinés. Il met ensuite à jour les valeurs maxDiffX et maxDiffY en fonction de la différence maximale trouvée pour les deux axes. Enfin, il retourne les indices des deux éléments sélectionnés pour constituer les graines des nouveaux nœuds.

2. **Split quadratique** : L'algorithme de division quadratique commence également par la méthode pickSeeds(). Cependant, dans ce cas, la méthode choisit la paire d'éléments qui, si répartis dans deux nœuds distincts, entraînerait le plus grand gaspillage. Le chevauchement est mesuré en comparant la surface des MBR combinés des éléments avec la somme des surfaces des MBR individuels. La méthode pickNext() est ensuite utilisée pour attribuer chaque élément restant à l'un des deux nœuds, en minimisant l'augmentation totale de la surface des MBR des deux nœuds. Le coût d'insertion est également calculé par la méthode calculateCost(), mais il prend en compte à la fois l'augmentation de la surface et l'augmentation du gaspillage des MBR. La complexité de l'algorithme split quadratique est $O(n^2)$, où n est le nombre d'éléments dans le nœud.

Algorithm 3 Pickseed pour l'heuristique quadratique

```
1:  function PickSeeds(subnodes)
2:      Initialiser maxWaste, seeds
3:      for i ← 0 to taille(subnodes) − 1 do
4:          Calculer e1 pour subnodes[i]
5:          for j ← i + 1 to taille(subnodes) − 1 do
6:              Calculer waste avec e1 et subnodes[j].MBR
7:              if waste > maxWaste then
8:                  Mettre à jour maxWaste, seeds[0], seeds[1]
9:              end if
10:         end for
11:     end for
12:     Retourner seeds
13: end function
```

Le pseudo-code ci-dessus 3 montre comment sélectionner les deux éléments deux nœuds distincts qui entraînent le plus grand gaspillage en termes de surface des MBR combinés. Il utilise des boucles imbriquées pour parcourir tous les éléments possibles de la liste des sous-nœuds. Ensuite, il calcule le gaspillage entre les MBR de chaque paire d'éléments et met à jour les valeurs maxWaste et seeds en fonction du gaspillage maximal trouvé. Enfin, il retourne les indices des deux éléments sélectionnés pour constituer les graines (seeds) des nouveaux nœuds.

les méthodes pickSeeds() et pickNext() sont implémentées différemment pour les R-Trees linéaires et quadratiques, afin de prendre en compte les heuristiques spécifiques de chaque algorithme de division. La méthode calculateCost() est également adaptée en fonction de l'implémentation pour évaluer correctement les coûts d'insertion. Les complexités des algorithmes sont $O(n)$ pour le split linéaire et $O(n^2)$ pour le split quadratique.

4.3 Recherche

La recherche dans un R-Tree est implémentée dans la classe RectangleTree. La méthode principale pour effectuer la recherche est search(Point). Cette méthode prend en entrée un objet Point représentant les coordonnées du point à rechercher et renvoie un objet Leaf correspondant au polygone contenant le point recherché.

La recherche est effectuée en parcourant récursivement l'arbre, en commençant par la racine. La méthode searchPoint(Node, Point) est utilisée pour la recherche récursive. Elle prend en entrée le nœud courant et le point recherché. Si le nœud courant est une feuille, la méthode vérifie si le point est contenu dans le polygone associé à la feuille et renvoie l'objet Leaf correspondant si nécessaire. Si le nœud courant est un nœud interne, la méthode vérifie si le point est contenu dans les MBR des sous-nœuds. Si c'est le cas, la méthode searchPoint() est appelée récursivement pour chaque sous-nœud qui contient le point.

Dans cette implémentation, les objets Envelope sont utilisés pour représenter les MBR des nœuds, et la méthode contains() de la classe Envelope est utilisée pour vérifier si le point est contenu dans un MBR. De plus, la méthode contains() de la classe Geometry est utilisée pour vérifier si un point est contenu dans un polygone.

La recherche se termine lorsque tous les nœuds contenant le point recherché ont été visités et que le polygone correspondant a été trouvé. L'objet Leaf correspondant est renvoyé par la méthode search(Point) à la fin de la recherche.

5 Performances

Dans cette section, nous analysons les performances de l'implémentation du R-Tree présentée et la rapidité des opérations d'insertion et de recherche. Nous comparons les résultats obtenus pour les R-Trees linéaires et quadratiques en utilisant un fichier donné, en faisant varier le nombre d'éléments (N) dans l'arbre et en mesurant le temps nécessaire pour construire l'arbre et effectuer une recherche sur 100 points [1] pour chacune des cartes.

1. La génération des 100 points utilisée dans ces résultats sont aléatoires et ne comprend pas de point null, ce qui peut prendre du temps.

TABLE 1 – Temps de construction et de recherche pour différentes cartes et tailles d'arbre (N)

Carte	N	Temps de construction (ms)		Temps de recherche (ms)	
		Linéaire	Quadratique	Linéaire	Quadratique
Monde	5	87	130	864	1150
	10	86	129	894	977
	50	87	134	977	1110
	100	90	133	1019	1003
	1000	80	116	884	884
Belgique	5	1313	2948	64	58
	10	1108	2081	46	30
	50	672	1242	22	28
	100	787	1411	22	25
	1000	893	1841	55	52
France	5	9151	27463	87	89
	10	5125	10972	93	85
	50	1944	3468	64	65
	100	1474	3084	47	40
	1000	1729	4858	75	72
Turquie	5	86	124	9	12
	10	83	111	9	16
	50	66	106	14	13
	100	70	106	13	10
	1000	67	107	11	8

5.1 Analyse des performances

Les performances d'un R-Tree dépendent en grande partie des choix d'heuristiques et de l'implémentation des algorithmes de division. Les R-Trees linéaires et quadratiques offrent des compromis différents en termes de temps de construction de l'arbre et de temps de recherche.

En analysant les résultats du tableau [2] on constate que les R-Trees linéaires ont généralement un temps de construction plus rapide, quelle que soit la carte ou la taille de l'arbre (N). Cela est dû au fait que l'algorithme de division linéaire est moins complexe. Cependant, cette rapidité peut entraîner un chevauchement plus important entre les MBR, ce qui pourrait potentiellement réduire l'efficacité des recherches dans certaines situations.

Les R-Trees quadratiques, en revanche, prennent généralement plus de temps à construire en raison de la complexité de l'algorithme de division quadratique. Néanmoins, cet algorithme minimise davantage les chevauchements entre les MBR, ce qui peut conduire à des recherches plus rapides et efficaces dans certains cas.

En ce qui concerne les temps de recherche, les résultats sont plus mitigés. Pour certaines cartes et tailles d'arbre, le temps de recherche est plus rapide pour les R-Trees linéaires, tandis que pour d'autres, les R-Trees quadratiques sont plus rapides. Ceci montre que la performance de recherche peut varier en fonction des données spécifiques et des paramètres de l'arbre.

2. Le tableau présente des données moyennes, obtenues à partir de trois essais distincts pour chaque scénario testé. Ces résultats ont été réalisées sur un processeur Apple Silicon M2.

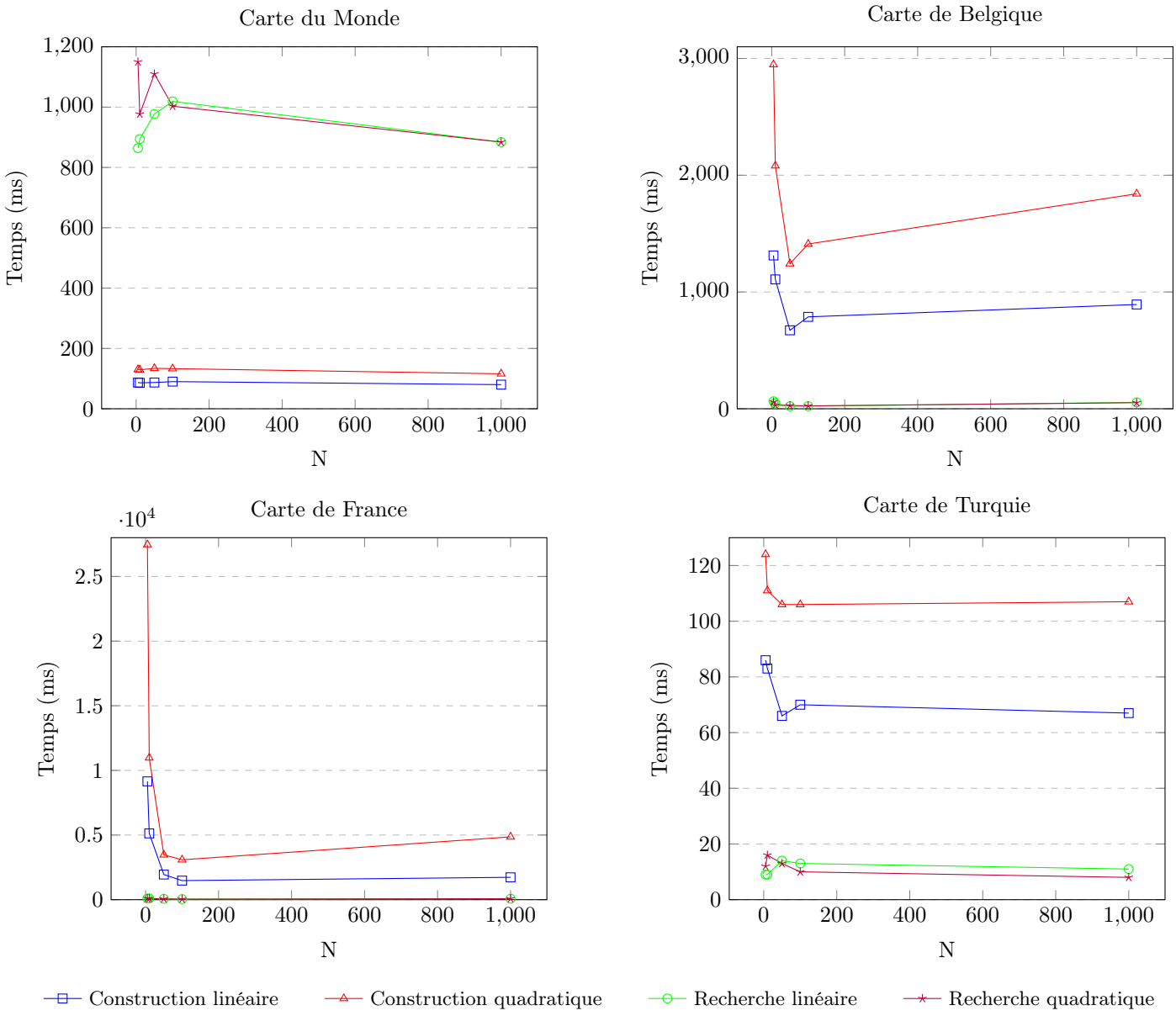

Figure 1 - Graphique du temps de construction et de recherche de points en fonction de N

Nous observons sur les graphiques présentés ci-dessus montrent que les performances des R-Trees linéaires et quadratiques varient en fonction des données et des paramètres de l'arbre. Ils mettent en évidence les différences entre les temps de construction et de recherche pour les deux types de R-Trees sur différentes cartes et tailles d'arbre (N).

6 Conclusion

Au cours de ce travail, nous avons présenté une implémentation des R-Trees linéaires et quadratiques basée sur les concepts décrits dans l'article d'Antonin Guttman [1]. Nous avons expliqué en détail les méthodes d'insertion, de division et de recherche pour les deux types de R-Trees. Les résultats de notre analyse des performances montrent que les R-Trees linéaires offrent généralement des temps de construction plus rapides, tandis que les R-Trees quadratiques peuvent offrir des temps de recherche plus rapides et efficaces dans certaines situations, en raison de la minimisation des chevauchements entre les MBR.

il est important de dire que les résultats de recherche peuvent changer selon les données et les options de l'arbre.

Plus tard, on pourrait améliorer ce qui est déjà fait, chercher d'autres façons de diviser et voir comment les R-Trees marchent dans d'autres situations.

7 Bibliographie

Références

[1] A. Guttman, "R-trees : A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data - SIGMOD '84*. Dallas, Texas : ACM, 1984, p. 47. [Online]. Available : https://dl.acm.org/doi/epdf/10.1145/971697.602266

[2] Y. Manolopoulos, A. Nanopoulos, and Y. Theodoridis, *R-Trees : Theory and Applications*. Springer, 2006. [Online]. Available : https://books.google.be/books?id=1mu099DN9UwC&pg=PR5&redir_esc=y#v=onepage&q&f=false