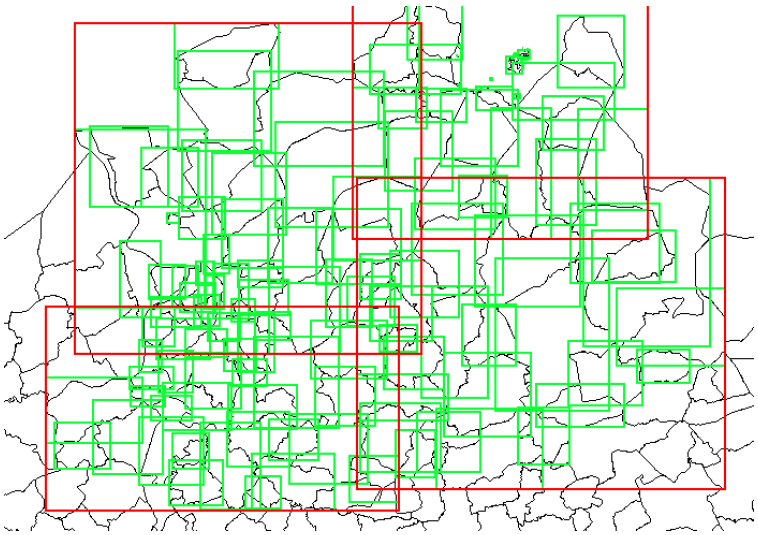
Algorithmique 2 (INFO-F203)
R-trees

Vandy Berten et Jean Cardinal

Université libre de Bruxelles (ULB)
Mars 2023



1 R-trees

1.1 Contexte

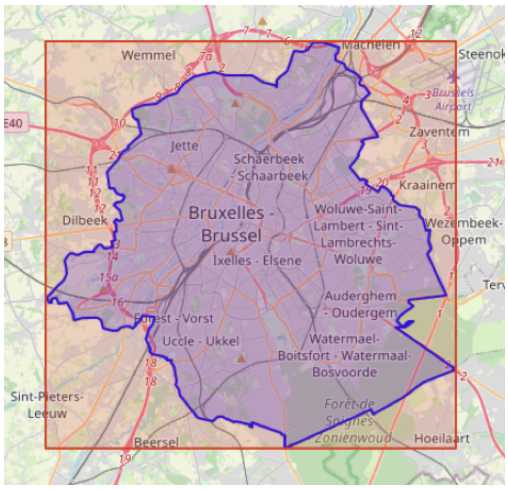
De nombreuses applications nécessitent de manipuler des polygones. Ils peuvent représenter des régions administratives sur une carte (pays, ville, commune, quartier...), correspondre à des informations géologiques (nature de sol, type de couverture...), mais aussi les différentes zones d'un dessin vectoriel ou d'une imagerie médicale.

Quand on manipule ce genre de données, une question vient souvent à l'esprit : à quel polygone appartient un point donné (clic d'une souris, position d'un capteur GPS...). Cette question nécessite de répondre au problème « Point in Polygon » (PIP), dont l'algorithme est connu depuis 1962 [2], et consiste à compter le nombre de fois qu'une demi-droite partant du point traverse une arête du polygone. Si ce nombre est pair, le point est en

dehors, sinon, il est contenu dans le polygone. Cet algorithme a une complexité en $O(m)$, où m est le nombre d'arêtes du polygone.

Supposons maintenant que l'on ait des milliers de polygones complexes, et que l'on veuille y tester un grand nombre N de points. Une méthode naïve consistera à tester l'algorithme PIP pour chaque point et chaque polygone, avec une complexité en $O(N \times M)$, où M est le nombre total de points dans l'ensemble des polygones.

Une première optimisation consiste à calculer au préalable pour chaque polygone l'enveloppe, où le « minimum bounding rectangle » (MBR), c'est-à-dire le plus petit rectangle horizontal englobant totalement le dit polygone. L'illustration ci-contre représente un polygone (en bleu, la région de Bruxelles-Capitale), et en rouge le MBR correspondant.



Avant d'exécuter l'algorithme PIP pour chaque polygone, on testera d'abord l'inclusion dans le MBR, en $O(1)$; puis, seulement si le point est dans le MBR, on invoque PIP. En effet, si le point ne fait pas partie du MBR, il ne fera nécessairement pas partie du polygone. Cette optimisation permet de drastiquement diminuer le nombre d'appels à PIP. Dans une situation idéale, PIP sera appelé un tout petit nombre de fois par point.

Mais si le nombre de polygones, et donc de MBR, est très élevé, il est possible d'éviter de considérer la totalité des MBR pour chaque point, grâce à une structure hiérarchique ou arborescente. Imaginons que nous recherchons à quel pays appartient un point. On pourrait dans un premier temps calculer le MBR de chaque continent. Dans la plupart des cas, seuls un ou deux continent(s) seront sélectionnés. On vérifiera alors l'inclusion du point dans l'ensemble de (sous-)MBR des pays appartenant à ce(s) continent(s), mais pas les autres.

Cependant, le regroupement (de pays en continent, de communes en provinces...) n'est pas toujours disponible, ni optimal.

L'algorithme R-Tree[1] tente de regrouper des MBR en ensembles proches, les plus disjoints possible des autres regroupements.

Dans ce projet, nous vous demanderons d'implémenter deux variantes de cette arborescence (quadratique et linéaire). Vous effectuerez ensuite un certain nombre de tests, pour en évaluer les performances en fonction des paramètres.

1.2 Structure

Un R-Tree (pour Rectangle-Tree) est un arbre, contenant deux types de composants : des nœuds (dont une racine), et des feuilles, toutes au même niveau.

Notons que nous utilisons ici une terminologie légèrement différente de celle de Guttman [1], qui considère des nœuds et des feuilles (correspondant tous les deux à nos

nœuds) et des *data objects* (correspondant à nos feuilles).

Une feuille contiendra un (multi-)polygone, le MBR associé, ainsi qu'un label. Un nœud contiendra une liste de descendants (nœuds ou feuilles), ainsi qu'un MBR correspondant à l'union de l'ensemble des MBR de ses descendants.

Pour des raisons de simplicité, nous ferons l'hypothèse que l'ensemble des polygones est disjoint, et qu'un point appartiendra donc à un ou zéro polygone.

Un point peut par contre appartenir à plusieurs MBR. Par exemple, dans l'illustration ci-dessus, un point situé au milieu du mot « Dilbeek » fera à la fois partie du MBR de Bruxelles-Capitale et du MBR correspondant au polygone voisin.

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.

1.3 Création

Si la recherche au sein d'un R-Tree est relativement simple, la construction d'une telle structure l'est nettement moins. Nous allons considérer deux méthodes, que nous vous demanderons de comparer.

Dans l'algorithme décrit par Gutmann en 1984[1], on suppose que l'on dispose d'un ensemble de polygones. L'insertion d'un polygone se passe globalement de la façon suivante :

- recherche du nœud d'insertion (`chooseNode`) : on commence par identifier le nœud pour lequel l'insertion du nouveau polygone minimisera l'augmentation du MBR ;
- insertion (`addLeaf`) : si, après ajout d'une nouvelle feuille, le nombre de feuilles composant un nœud atteint un seuil N fixé, ce nœud sera « coupé en deux » (split), selon une méthode décrite plus bas. Cette division augmentera dès lors de 1 le nombre de sous-nœuds du niveau supérieur. Si besoin, la division sera également réalisée à ce niveau-là, en remontant ainsi jusqu'à la racine.

Décrivons maintenant plus en détails ces algorithmes.

On définit par l'élargissement d'un MBR par un polygone `P` comme étant l'accroissement de superficie nécessaire pour inclure `P` dans le MBR, c'est-à-dire :
`area(MBR` $\cup$ `P) - area(MBR)`, où `MBR` $\cup$ `P` dénote l'enveloppe qui englobe MBR et `P`. Cet élargissement sera nul si le polygone est situé entièrement à l'intérieur du MBR.

La fonction `chooseNode(node, polygon)` renverra le sous-nœud de `node` minimisant

l'élargissement pour le polygone `polygon`. La recherche du nœud d'insertion se fera en utilisant la fonction `chooseNode` depuis la racine jusqu'aux nœuds dont les descendants sont des feuilles.

Nous définirons la méthode `addLeaf` de façon récursive. Si un « split » a été nécessaire, le nouveau nœud à rajouter est retourné, sinon « null » est renvoyé.

```
addLeaf(node, label, polygon):
    if size(n.subnodes)==0 or n.subnodes[0] is a leaf:
    # bottom level is reached -> create leaf
        n.subnodes.add(new Leaf(name, polygon))

    else:# still need to go deeper
        n = chooseNode(node, polygon)
        new_node = addLeaf(n, label, polygon)
        if new_node !=null:
            # a split occurred in addLeaf,
            # a new node is added at this level
            subnodes.add(new_node)
    expand node.mbr to include polygon
    if size(node.subnodes)>=N:
        return split(node)
    else:
        return null
```

Nous arrivons maintenant à la partie la plus complexe : la division (split). Il s'agira de répartir l'ensemble des branches d'un nœud en deux ensembles, de façon à minimiser la superposition des deux MBR (de sorte à minimiser la probabilité, lors du search, de devoir explorer les deux branches), tout en divisant au mieux entre deux ensembles équilibrés (pour que le principe du « diviser pour régner » soit maximal). Dans notre algorithme, `split(node)` va :

— extraire de node.subnodes une série de sous-composants (nœuds ou feuilles) ;

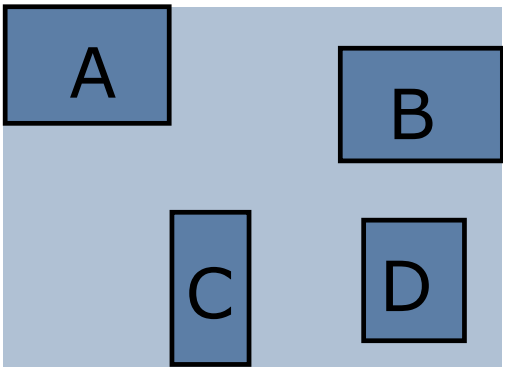
— créer un nouveau nœud `new_node`, de sorte que : $\text{node}_{\text{presplit}}.\text{subnodes} = \text{node}_{\text{postsplit}}.\text{subnodes} \cup \text{new\_node.subnodes}$

Une méthode « optimale » consiste à comparer l'ensemble des partitions possibles de N nœuds. Il y en a $2^N - 2$ (car on exclut les deux cas « tout dans A » et « tout dans B »). Ce n'est, dans la pratique, pas réaliste. Nous allons dès lors considérer deux heuristiques.

1.3.1 Split quadratique

Dans cette version, on commence par choisir deux « seeds » (`pickSeeds`), soit deux nœuds les plus éloignés possible. Pour ce faire, on considère chaque paire de nœuds. Pour chacune de ces paires, on calcule la superficie de l'enveloppe englobant les deux enveloppes, moins celles des enveloppes elles-mêmes. On sélectionnera la paire qui maximise cette superficie. Dans l'exemple ci-contre, la paire (A, D) est celle maximisant cette surface.

Ensuite, on parcourra chaque autre polygone, pour choisir celui qui maximisera sa « préférence » à rejoindre un groupe plutôt que l'autre (`pickNext`). Concrètement, on calcule, pour chaque polygone restant, sa « préférence » de la façon suivante : soit a_k ($k \in \{1, 2\}$) la superficie de l'enveloppe e_k du groupe k, et A_k^i la superficie de l'enveloppe qui contient à la fois e_k et le polygone p_i. $A_k^i - a_k$ dénote le coût d'ajouter p_i au groupe k. On choisira comme prochain polygone celui maximisant la différence de coût entre les deux groupes, à savoir $|(A_1^i - a_1) - (A_2^i - a_2)|$.



1.3.2 Split linéaire

Pour la version linéaire, nous vous invitons à lire la description de Guttman[1].

2 Expériences sur des données réelles

Après avoir implémenté les deux variantes du R-Tree (quadratique et linéaire), nous vous demandons de les évaluer, en calculant le temps nécessaire à l'exécution de la fonction `search`, en fonction du nombre de points et des différents paramètres. Vous ferez cette évaluation sur les 3 jeux de données proposés ci-dessous, ainsi que sur un autre de votre choix, qui devra contenir au minimum 100 (multi)-polygones disjoints. Dans le jeux de données mondial, certains pays, comme la France, vous poseront problème. Il s'agit d'un multi-polygone extrêmement étendu (France métropolitaine, Île de la Réunion, Martinique...), pour très peu de territoire effectif, permettant difficilement de découper efficacement l'espace. Un problème similaire se posera pour les territoires près des pôles. Il vous faudra néanmoins trouver une solution pour les identifier et les traiter correctement pour garder un index efficace.

2.1 La librairie Geotools

De façon à se concentrer uniquement sur les aspects « arbres » et non géométriques, nous utiliserons la libraires Java `geotools` (`https://www.geotools.org/`). Vous trouverez sur ce site web la démarche nécessaire pour intégrer la libraire dans votre IDE favori (Eclipse, Netbeans, IntelliJ). Celle-ci nous permettra de lire des fichiers au format « shapefile », comme ceux que l'on peut trouver ici :

— Découpe de la Belgique en secteurs statistiques : `https://statbel.fgov.be/fr/open-data?category=191`
— Pays du monde : `https://datacatalog.worldbank.org/search/dataset/0038272/World-Bank-Official-Boundaries`

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

Vous trouverez sans difficulté le « shapefile » correspondant à la découpe de votre pays favori. Notez que vous pouvez également importer un fichier au format GeoJSON.

En annexe à cet énoncé, vous trouverez un exemple de code Java (SinglePoint.java) qui lit un tel fichier, le parcourt à la recherche du polygone contenant un point donné, puis affiche une carte à l'écran, mettant en évidence le polygone trouvé (si présent). Vous y trouverez l'essentiel des fonctions nécessaires à l'élaboration de ce projet. En résumé (notons qu'il existe plusieurs façons d'utiliser la libraire, libre à vous de vous écarter des conseils ci-dessous) :

— On importe :
  — `import org.opengis.feature.simple.SimpleFeature`
  — `import org.locationtech.jts.geom.MultiPolygon`
  — `import org.locationtech.jts.geom.Point`
  — `import org.locationtech.jts.geom.Envelope`
  — `import org.geotools.geometry.jts.GeometryBuilder`
— On obtient une « `SimpleFeature feature` » en parcourant le shapefile
— On peut en extraire un « `MultiPolygon` » grâce à un appel à « `(MultiPolygon) feature.getDefaultGeometry();` »
— On peut en extraire l'attribut « NAME » grâce à « `feature.getAttribute("NAME");` »
— Pour créer un « `point (double x, double y)` », on utilisera un « `GeometryBuilder gb = new GeometryBuilder();` », puis « `Point pt = gb.point(x,y)` »;
— Pour savoir si `polygon` contient `pt` : « `polygon.contains(pt)` »
— Pour connaître le MBR de polygon : « `Envelope mbr = polygon.getEnvelopeInternal()` »
— Pour savoir si `mbr` contient `pt` : « `mbr.contains(pt.getCoordinate())` »

Consignes

— Votre travail sera composé d'un rapport scientifique, d'un programme en Java, et d'un fichier `README`.
— Le fichier `README` est un fichier texte devant contenir les instructions permettant au correcteur de compiler, exécuter et tester le code.
— Le rapport sera composé de parties correspondant aux sections précédentes, chacune faisant l'objet d'une évaluation.
— On demande les réponses aux questions posées, une description détaillée des algorithmes mis en œuvre, et une description synthétique de l'organisation de vos programmes Java, avec un commentaire détaillé des parties les plus importantes.
— Le rapport devra être au format `pdf`, rédigé en français correct, et mis en page avec le logiciel LaTeX.
— Le rapport comprendra également la liste des références bibliographiques utilisées.

— On s'attend pour ce travail à un rapport dont la longueur est d'environ une dizaine de pages.
— Les fichiers sont à remettre sur l'Université Virtuelle (UV).

Recommandations et ressources externes

Rédaction scientifique. Des conseils généraux sur la rédaction d'un rapport scientifique se trouvent dans le document « Éléments de rédaction scientifique en informatique », rédigé par Hadrien Mélot (UMons) et disponible à l'adresse suivante :
`http://informatique.umons.ac.be/algo/redacSci.pdf`.

Ressources bibliographiques. Vous êtes encouragé·e·s, pour répondre aux questions, à consulter la littérature scientifique. Les documents utilisés doivent être clairement mentionnés dans le texte et figurer dans la liste de références bibliographiques. Nous vous encourageons à utiliser le logiciel BibTeX :
`https://www.bibtex.com/`.

Code. Les programmes doivent être dûment commentés et structurés, et respecter les bonnes pratiques enseignées dans les cours de programmation orientée objet. Vous êtes autorisé·e·s à réutiliser, en le mentionnant clairement, les classes de la bibliothèque Java `algs4.jar`, disponible à l'adresse suivante :
`https://algs4.cs.princeton.edu/code/`.

Travail en binôme, échanges et plagiat

Vous êtes autorisé·e·s à rendre le travail en collaboration avec un·e autre étudiant·e, auquel cas la même note sera attribuée au deux membres du binôme. Vous pouvez également rendre le travail à titre individuel.

Vous êtes autorisé·e·s à discuter des problèmes et de vos solutions éventuelles avec d'autres étudiant·e·s et d'autres binômes. Si un point de votre rapport est le fruit de ces discussions, vous êtes invité·e·s à le mentionner explicitement. La rédaction du rapport et des programmes doit cependant être propre à chaque étudiant·e / binôme. Tout emprunt non mentionné explicitement est un plagiat, et constitue une fraude. Les étudiant·e·s reconnu·e·s coupables de fraude ou de tentative de fraude s'exposent à des sanctions disciplinaires comprenant l'annulation de la session d'examens et l'interdiction de s'inscrire à la session d'examens suivante.

`https://bib.ulb.be/fr/support/boite-a-outils/evitez-le-plagiat`

Date de remise

Vendredi 28 mai 2023, avant minuit.

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*, page 47, Dallas, Texas, 1984.

[2] M. Shimrat. Algorithm 112 : Position of point relative to polygon. *Communications of the ACM*, 5(8) :97–111, 1962.