

Compte rendu de projet Initiation à la Recherche

MARGUERITE Alain
RINCÉ Romain

Université de Nantes
2 rue de la Houssinière, BP92208, F-44322 Nantes cedex 03, FRANCE



Table des matières

1	Étude des structures de données	2
1	Introduction	2
2	Définition de la boîte	2
2.1	Généricité des caractéristiques	3
3	Pavage	3
3.1	Problématique	3
3.2	Étude de structures	4
2	Stockage des boîtes et visualisation	6
1	Étude d'une solution possible : le QuadTree	6
2	Étude d'une seconde solution : le R-tree	9

1 Étude des structures de données

1 Introduction

L'objectif de ce document est de mener une étude sur les différentes structures de données nécessaires aux futurs algorithmes de visualisation. Nous nous intéresserons en particulier à l'opération d'accès à une caractéristique ou d'une donnée d'une boîte ainsi qu'à l'occupation mémoire de cette dernière. Il s'appuie sur le document de spécification (cf : annexe). Ce document sur les calculs de complexité seront réalisés selon le paramètre n représentant le nombre de boîtes, et d est le nombre de dimensions du problème. Le nombre de boîtes réellement utilisées dans le pavage est représenté par N .

2 Définition de la boîte

C'est l'entité atomique du pavage. Les accès à ses attributs sont donc cruciaux. On rappelle qu'une boîte est définie de la manière suivante :

- Un identifiant : soit des chaînes de caractères (IDStr) respectant un format précis (c.f 1.1 du document de spécifications), soit des entiers positifs (IDInt).
- Une liste de coordonnées dans l'ordre des variables définies en entête. Une liste de type **Interval**.
- Une liste des caractéristiques, dans l'ordre et selon les types définis en entête.

Ces données seront régulièrement requises lors de la mise en œuvre des algorithmes nécessaires à la visualisation. Il est donc important que leurs accès soient rapides, voire directs. Pour le cas de l'identifiant, s'agissant d'une simple **String** le problème de la structure à utiliser ne se pose pas. Pour la liste des coordonnées en revanche, il s'agit d'une séquence finie de données. Plusieurs possibilités sont alors envisageables :

- Un tableau : L'accès à une coordonnée est direct. Les opérations d'ajout et de suppression sont en revanche coûteuses pour les tableaux dynamiques ($O(n)$).
- Une liste : Si l'accès à une coordonnée n'est pas direct ($O(n)$), les opérations d'ajout et de suppression sont en temps constant.
- Une table de hachage : Coûteuse si la fonction de hachage n'est pas appropriée, une table de hachage propose cependant un accès en $O(1)$. Cependant le phénomène de

collisions (mauvaise répartition des clefs entraînant un conflit entre deux valeurs) est à prendre en considération :

Implémentation avec un tableau dynamique Une telle structure permet de garantir un accès en temps constant. Cependant chaque collision va doubler l'occupation mémoire du tableau. Or même si la fonction de hachage est bonne, il est possible d'avoir au moins une collision, ce qui aurait pour conséquence une perte de l'espace mémoire qui se répercuterait sur chaque boîte.

Gestion des collisions avec chaînage Contrairement à la structure précédente, cette méthode permet de ne pas occuper trop d'espace en chaînant les éléments entrants en collision. Malheureusement la complexité en pire cas des opérations d'accès passe en $O(n)$. En revanche, grâce à une bonne fonction de hachage, on accèdera généralement en temps constant sans contre coût mémoire.

Le passage en revue de ces différentes structures écarte la liste et la table de hachage implémentée par un tableau dynamique. En effet les performances des opérations d'accès de la liste ne sont pas raisonnables. De même l'implémentation d'une table de hachage par un tableau dynamique risque d'entraîner une perte de mémoire trop importante.

2.1 Généricité des caractéristiques

Un problème majeur apparaît pour l'instanciation des boîtes. On rappelle qu'une caractéristique à plusieurs types (`String`, `Number` ou `Interval`). Plusieurs solutions sont envisageables :

- Une Map unique contenant des `String` stocke les différentes caractéristiques de la boîte. On aura casté les attributs `Number` ou `Interval` en `String`. Il sera alors nécessaire, pour chaque futurs accès, d'effectuer un cast dynamique. On rajoute alors une constante supplémentaire à la complexité de cette opération.
- Trois tableaux (un pour chaque type) au sein d'une boîte. Trois Maps (une pour chaque type) «générales» au niveau du pavage. La clef d'une map est l'id de la caractéristique et la valeur de la map son indice dans le tableau concret. La boîte peut alors retrouver la valeur de la caractéristique au sein de son tableau.
- Chaque boîte possède trois Maps pour ces trois types de caractéristiques.

3 Pavage

3.1 Problématique

Le cahier de spécification exige de l'outil la capacité à charger un pavage de taille non déterminée. Si le nombre de boîtes sera en pratique nécessairement borné (limite mémoire de la machine), il faut cependant répondre à cette attente en proposant une structure de stockage capable de supporter un très grand nombre de boîtes. De plus

l'outil doit être en mesure de fournir régulièrement des listing spécifiques de boîtes. Par exemple pour afficher la liste de celles concernées par un filtre. La structure du pavage doit être en mesure de répondre de manière efficace à des requêtes de séquences de boîtes selon un ordre particulier. La structure du pavage doit être aussi en mesure de répondre efficacement à la structure de visualisation graphique (fournir rapidement les nouvelles boîtes dans le champs de visualisation, lors d'une rotation de caméra par exemple). Quelles structures de données et quelles stratégies choisir face à de telles exigences ?

3.2 Étude de structures

Vector : Collection de données à accès direct par indice. Le nombre de boîte étant donné dans l'entête du fichier d'entrée, une implémentation par un tableau statique proposerait une complexité en $O(n)$ pour l'opération de stockage du pavage. Si l'accès à une boîte à partir de son indice serait direct, l'opération de recherche en revanche aurait une complexité en $O(n)$.

Dictionnaire : Collection de données à accès direct par clef. Dans l'hypothèse de posséder une fonction de hachage ne provoquant pas de collisions, une implémentation par une fonction de hachage propose une complexité en $O(n)$ (à nouveau grâce à la connaissance du nombre de boîtes dans l'entête du fichier d'entrée).

Arbre binaire Par exemple l'utilisation d'un arbre binaire de recherche pour la création de n boîtes aurait une complexité de n^2 en pire cas. En effet il s'agit du cas où les boîtes arriveraient triées selon l'ordre inverse de celui que l'on souhaite. Il faudrait alors effectuer $(p - 1)$ comparaisons, pour chaque boîte : $\sum_{p=2}^n (p - 1)$ Soit $O(n) = \frac{1}{2}n^2 - \frac{1}{2}n$. Cependant dans le meilleur des cas cette opération a une complexité en $O(n \log n)$. La création d'un pavage composé de n boîtes à d dimensions aurait alors une complexité égale à : $O(d \times n \log(n))$

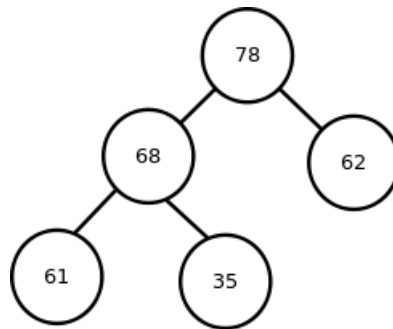


FIGURE 1.1 – arbre binaire

L'arbre binaire était équilibré par définition, la complexité de son opération de recherche est en $O(\log_2 n)$ (hauteur de l'arbre).

Arbres a-b Il s'agit d'un arbre de recherche avec les propriétés suivantes :

- $a \leq 2$ et $b \leq 2a - 1$ deux entiers.
- La racine a au moins 2 fils (sauf si l'arbre ne possède qu'un nœud) et au plus b fils. Les feuilles sont de même profondeur.
- Les autres nœuds internes ont au moins a et au plus b fils.

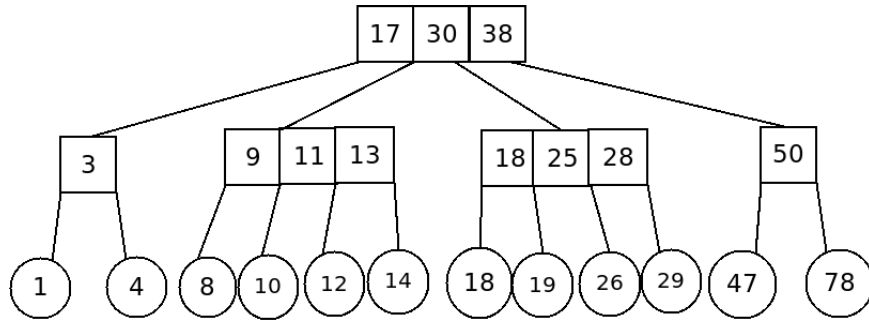


FIGURE 1.2 – a-b arbre

L'avantage des arbres a-b est que leurs hauteurs sont comprises entre les valeurs suivantes : $\frac{\log n}{\log b} \leq h < 1 + \frac{\log n/2}{\log a}$. Ainsi les opérations d'insertion ne seraient plus en $O(n \log n)$ mais en $O(\log n)$. La création d'un pavage composé de n boîtes à d dimensions aurait alors une complexité en $O((n \times d) \log(n))$. La recherche d'une boîte quant à elle aurait une complexité en $O(\log n)$.

2 Stockage des boîtes et visualisation

Trois difficultés majeures apparaissent dans la réalisation du logiciel de visualisation. La première est bien entendu la gestion d'une très grande quantité de boîtes lors de l'affichage. Il est en effet nécessaire d'offrir un accès rapide aux informations des boîtes dans la fenêtre. La seconde est la gestion des filtres sur ces mêmes boîtes. Et le troisième apparaît lors du changement des variables étudiées (changement des dimensions visualisées). Dans cette section, nous chercherons d'avantage à apporter une solution pour les problèmes de temps d'accès et de changement de variables. Il est en effet probable que la gestion des filtres soit effectuée par une structure différente.

1 Étude d'une solution possible : le QuadTree

Une des solutions qui pourrait permettre une visualisation fluide du pavage tout en répondant au document de spécifications serait de représenter le pavages sous une forme de QuadTree pour deux dimensions ou OcTree pour trois dimensions.

Le QuadTree consiste à découper récursivement un espace fini en deux dimensions en quatre parties égales. Chacune de ces parties sont stockées dans un nœud. On itère ce mécanisme sur chacun de ces nœuds jusqu'à isoler spatialement les éléments recherchés.

Cette structure pourrait être utilisée pour déterminer la position des boîtes dans l'espace.

L'algorithme permettant de partitionner est récursif. Dans le cas récursif, l'algorithme divise l'espace en quatre et itère sur chaque sous-espace. Dans le cas d'arrêt on ne subdivise plus. Nous décrivons par la suite l'ensemble des cas de récursions et d'arrêts pour la division d'un espace donné. Les images jointes au texte sont des représentations des différents cas dans lesquels les cadres rouges sont des boîtes solutions et les cadres noirs un espace fourni à l'algorithme. Vous trouverez une classification visuelle dans la table 2.1.

- Cas d'arrêts :

- L'espace fourni est entièrement inclus dans une boîte .

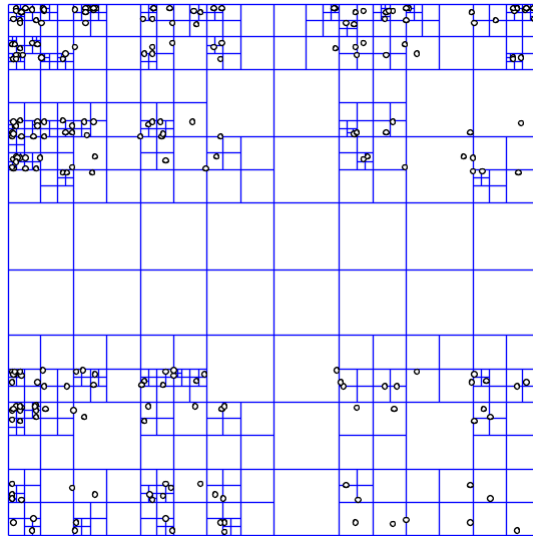







FIGURE 2.1 – Représentation d'un QuadTree où les données sont des points

- L'espace fourni contient entièrement une seule boîte .
- L'espace fourni contient en partie une seule boîte .
- L'espace fourni n'intersecte aucune boîte .
- L'espace fourni ne peut plus être subdivisé car on a fourni une taille minimale pour les espaces.
- cas de récursions :
 - L'espace fourni intersecte plusieurs boîtes  ou encore .







Cas de récursion	 
Cas d'arrêt	   

TABLE 2.1 – Classification visuelle des cas d'arrêts et de récursions

L'OcTree repose sur le même principe mais étendu à trois dimensions. L'espace est donc découpé en huit parties à chaque fois.

Avantage de cette méthode Cette structure est particulièrement intéressante pour la visualisation du pavage. En effet pour une fenêtre de visualisation donnée, il est très simple et rapide d'extraire la sous-arborescence correspondante à l'espace visualisé et permet aussi de ne pas afficher les objets trop petits.

De plus il serait possible de créer une structure reposant sur le même principe que le QuadTree mais étant k -dimensionnelle¹. Chaque espace peut alors être potentiellement subdivisé en 2^k sous-espaces. Cette structure permettrait d'éviter de recalculer l'arbre à chaque changement de variables de visualisation. On évite par ailleurs le cas de superposition de boîtes pour lequel l'algorithme n'est plus efficace (cf : figure 2.2).

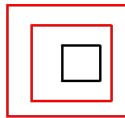


FIGURE 2.2 – Superposition de boîtes

Inconvénient de cette méthode Le problème majeur de cette méthode se présente lorsque des boîtes sont côte à côte (cf : figure 2.3). Dans une telle situation chaque

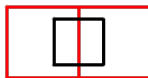


FIGURE 2.3 – Superposition de boîtes 1

division va entrainé la création d'un espace dans la même configuration. L'algorithme ne s'arrêtera donc pas avant d'avoir atteint la taille minimale d'un espace. Nous nous retrouvons donc avec un grand nombre de boîtes au niveau de ces « frontières » (cf : figure 2.4). Ainsi sachant que la précision maximale par défaut de *RealPaver* est de

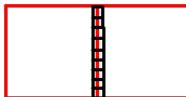


FIGURE 2.4 – Superposition de boîtes 2

10^{-16} , cela implique qu'il est nécessaire d'au moins égaler cette précision pour l'arbre de visualisation. Ainsi pour une sortie de *RealPaver* contenant un total l de longueurs de « frontières » cumulées et p la précision du modèle, on a un nombre d'espaces à créer supérieur à $\frac{l}{p}$ avec p très petit. Par exemple pour une sortie de *RealPaver* comportant une « frontière » de taille 1, il faudra au moins 10^{16} espaces pour la contenir.

Brève conclusion Le QuadTree est une structure intéressante pour la visualisation mais si un nœud de l'arbre a un coût en mémoire non nul, alors l'espace mémoire de la structure va exploser. Elle semble donc, pour le moment, inappropriée.

1. k étant la dimension du problème fourni à *RealPaver*.

2 Étude d'une seconde solution : le R-tree

Bien que le QuadTree soit une structure intéressante pour permettre l'indexation et la recherche de points dans un espace, celui-ci l'est beaucoup moins pour la gestion de données à dimensions non nulles. Le R-tree est une structure proposée en 1984 par Antonin GUTTMAN permettant l'indexation et la recherche d'éléments de dimension $d > 0$ dans k dimensions [Gut84].

Le R-tree est une variante équilibrée de l'arbre B, sa structure est la suivante :

- Un noeud de l'arbre correspond à une boîte non-solution du pavage (généralement appelé « page »).
- Chaque boîte peut contenir entre m et M sous-boîtes entièrement incluses. Avec $m \leq \frac{M}{2}$.
- Une feuille de l'arbre est une boîte ne contenant que des boîtes solution du pavage.

La figure 2.5 donne une bonne idée de l'organisation des R-trees :

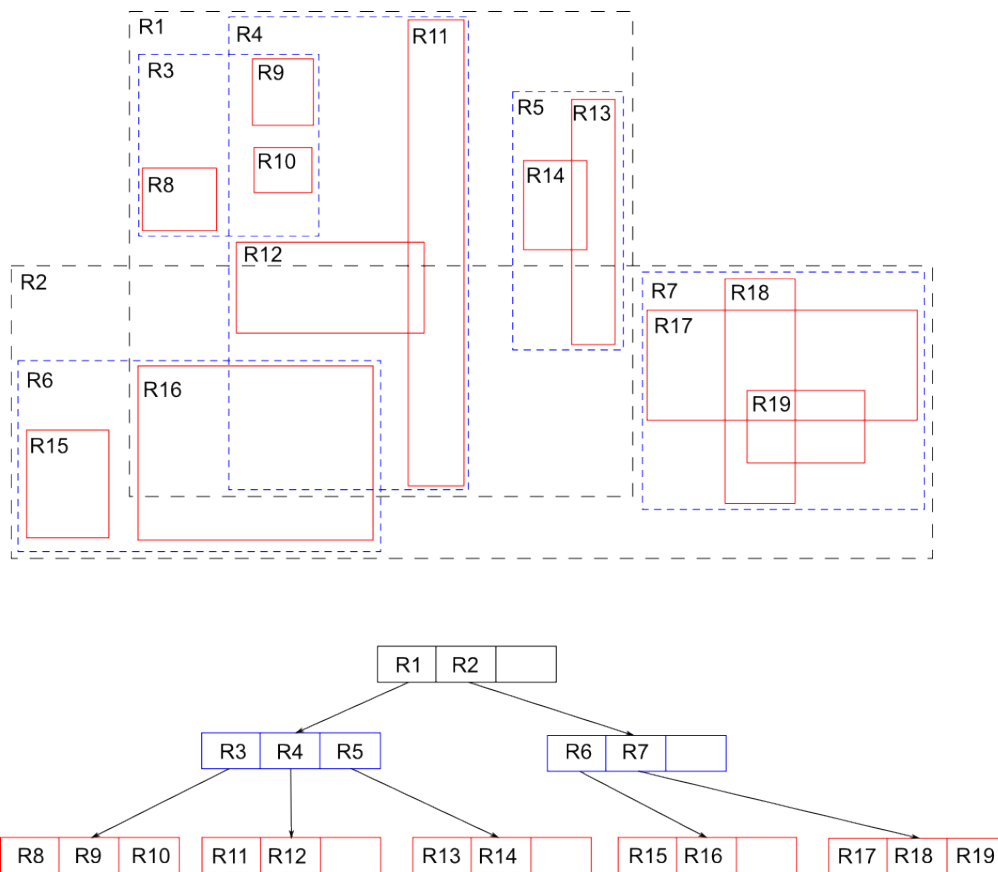


FIGURE 2.5 – Représentation d'un R-tree [Wik]

Les algorithmes permettant la recherche et la création du R-tree sont qu'en à eux décrivent dans l'article de A. GUTTMAN dans la section **3. Searching and Updating**[Gut84].

Avantages et inconvénients du R-tree Le R-tree est une structure de données spécialement conçue pour la recherche de données à dimensions d pour dans des espaces k -dimensionnels. L'arbre a une profondeur maximale h_{max} égale à :

$$h_{max} = \lceil \log_m n \rceil - 1 \quad (2.1)$$

et le nombre de nœuds est au pire égale à :

$$\sum_{i=1}^{h_{max}} \left\lceil \frac{n}{m^i} \right\rceil = \left\lceil \frac{n}{m} \right\rceil + \left\lceil \frac{n}{m^2} \right\rceil + \dots + 1 \quad (2.2)$$

Bien que ne pouvant garantir de bonnes performances en pire cas², le R-tree offre en pratique de bons résultats; on pourra d'ailleurs se reporter à l'analyse de performances effectuer par A. GUTTMAN³. Cependant il existe aujourd'hui de nombreuses variantes du R-tree (R*tree, Hilbert R-tree, etc. . .), on pourra donc, pour une analyse plus générale des différentes implémentations, préférer « *R-trees : Theory and Applications* »[MNPT06].

2. « *More than one subtree under a node may need to be searched, hence it is not possible to guarantee good worst-case performance.* »[Gut84], section **3.1 Searching**

3. section **4. Performance**[Gut84]

Table des figures

1.1	arbre binaire	4
1.2	a-b arbre	5
2.1	Représentation d'un quadtree où les données sont des points	7
2.2	Superposition de boîtes	8
2.3	Superposition de boîtes 1	8
2.4	Superposition de boîtes 2	8
2.5	Représentation d'un R-tree [Wik]	9

Bibliographie

- [Gut84] Antonin Guttman. R-trees : A dynamic index structure for spatial searching. <http://www-db.deis.unibo.it/courses/SI-LS/papers/Gut84.pdf>, 1984.
- [MNPT06] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Yannis Theodoridis. *R-Trees : Theory and Applications*. Springer, 2006.
- [Wik] Wikipedia. R-tree. <http://en.wikipedia.org/wiki/R-tree>.