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

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 données 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 revanches 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 entrainant 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 chainant 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'entrainer 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 retrouve l'indice au sein d'une caractéristique son tableau .
- Chaque boîte possède trois Maps pour ces trois types de caractéristiques. Au niveau de l'allocation mémoire, l'o au coût de la recherche dans la Map (si par exemple une boîte ne possède que des caractéristiques de type `String` et aucune de types `Number` ou `Interval`.

3 Pavage

L'outil de visualisation peut charger un fichier entrée de manière dynamique ou non. Nous nous plaçons ici dans le cadre où cette option de chargement dynamique n'est pas

activée.
L'outil va lire séquentiellement chaque ligne du fichier d'entrée. Le cahier de spécification exige fréquemment un listing de boîtes. Par exemple pour afficher la liste des boîtes concernées par un filtre. La structure du pavage doit être en mesure de répondre de manière efficace à des requêtes d'une séquence d'un nombre de boîtes selon un ordre particulier. La structure du pavage doit être aussi en mesure de répondre efficacement à la structure de visualisation graphique (cf : 1). On peut d'emblée envisager plusieurs stratégies pour le stockage du pavage :

- Le fichier d'entrée est lu une unique fois. Les méthodes de structures dans laquelle les boîtes sont stockées sont suffisamment performantes pour répondre à toutes les spécifications. On peut alors se poser la question s'il faut :
  - Insérer les boîtes dans la structure puis la trier plus tard.
  - Utiliser un tri par insertion.
- Il est possible que refaire des «passes» sur le fichier d'entrée soit une solution. Dans les cas où les opérations de tris et/ou de recherches seraient trop coûteuses pour la structure du pavage (par exemple lors d'une demande de listing de certaines boîtes). On aurait une complexité en pire cas en $O(n)$.

3.1 Stockage du pavage

Le nombre potentiellement très grand de boîtes élimine d'emblée la possibilité de choisir une HashMap. En effet même si la fonction de hashage est judicieusement choisie, l'occupation mémoire requise serait bien trop importante. Les listes ne sont pas appropriées ici. Une complexité en $O(n^2)$ pour un accès à une boîte n'est pas raisonnable. Les arbres ont l'atout de pouvoir stocker et manipuler un grand nombre de d'entités. Les arbres de recherches sont des arborescences ordonnées permettant un accès en $O(\log(n))$. Dans le cas où la structure serait triée au fur et à mesure de sa construction. Les arbres de recherche proposent de bonnes performances. Nous développerons pourquoi à travers des cas d'exemples dans les prochains paragraphes :

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 : $O(d \times n \log(n))$

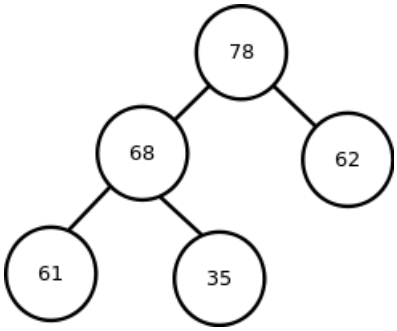

FIGURE 1.1 – a-b tree

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

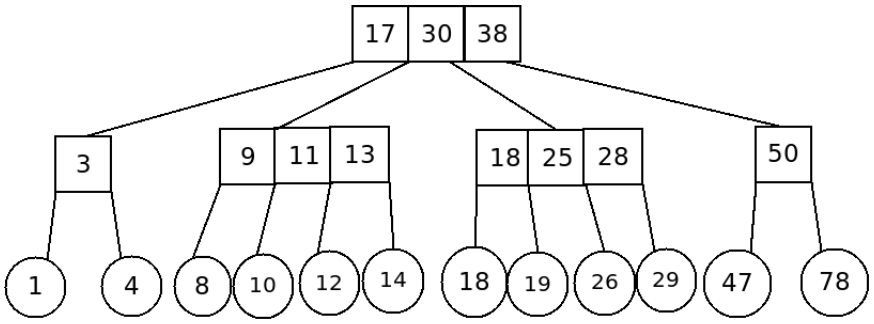

FIGURE 1.2 – a-b tree

L'avantage des arbres a-b est que leurs hauteurs sont comprises entres 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 $O((n \times d) \log(n))$.

2 Stockage des boîtes et visualisation

Trois problèmes majeurs 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 apparait lors du changement des variables étudiées (changement des dimensions visualisées).

1 Étude d'une solution possible : le QuadTree

Une des solutions qui permettrait d'offrir une visualisation fluide du pavage tout en répondant aisément 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 strucure pourrait être utilisée pour déterminer la position des boîtes dans l'espace selon la méthode suivante :

Si un des nœuds du quadtree ne contient aucune boîte ou qu'il est entièrement inclus dans un ensemble de boîtes alors il n'est plus nécessaire de le subdiviser. Ce nœud contiendra donc une référence sur chacune de ces boîtes.

- À contrario si un nœud du QuadTree contient une des bornes d'une des boîtes, alors il est nécessaire de subdiviser ce nœud.
- On arrête aussi de diviser les nœuds lorsque l'on arrive à une précision inférieur à la précision d'affichage (taille d'un pixel).

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

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.

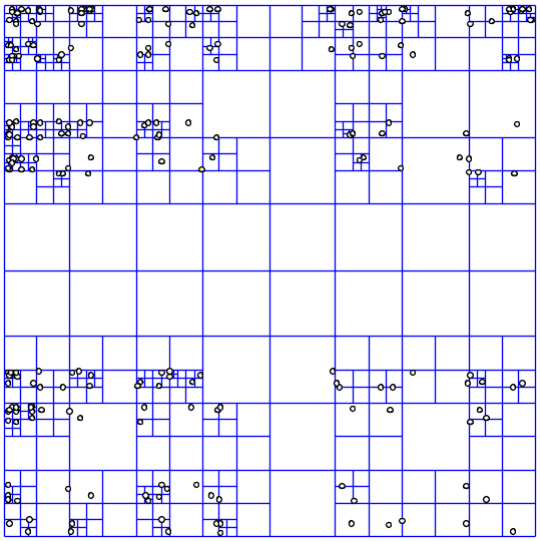

FIGURE 2.1 – Représentation d'un quadtree

Le problème de cette méthode est essentiellement du à la construction. La structure devra découper en la plus petite feuille possible au niveau des bornes des boîtes ce qui va nécessairement entrainer la construction d'un grand nombre de feuilles. Ce problème est particulièrement gênant puisque si l'on cherche à localiser une boîte de dimensions 1 sur 1 avec la précision donnée par défaut par Realpaver de 10^{-16}, on se retrouve avec au moins 4×10^{16} feuilles. On peut optimiser l'algorithme de construction en considérant que si un des nœuds contient l'intégralité ou une partie d'une seule boîte, il n'est plus nécessaire de subdiviser. La recherche d'une seule boîte étant immédiate.

Un autre problème apparait lorsque l'utilisateur souhaite changer les variables visualisées dans l'outil. Si l'on reste sur une structure du type QuadTree ou octree, il sera nécessaire de recalculer entièrement l'arbre.

On pourrait alors supposer une structure similaire k-dimensionnelle [1] où chaque nœud possède 2^k nœuds. L'avantage d'une telle structure est que lorsque l'on désire changer les variables de visualisation, le calcul est déjà effectué.

Malgré le fait que le QuadTree pouvait être une structure intéressante, le nombre de feuilles créées est bien trop important et ne peut donc pas être utilisé.

1. k étant la dimension du problème fourni à Realpaver.

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

Le R-tree est une structure de données utilisée pour stocker des informations k-dimensionnelles. Le principe est le suivant :

- Un noeud de l'arbre correspond à une boîte non-solution du pavage.
- 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.
- L'arbre est équilibré.

La figure 2.2 donne une bonne idée du principe des R-trees[Wik] :

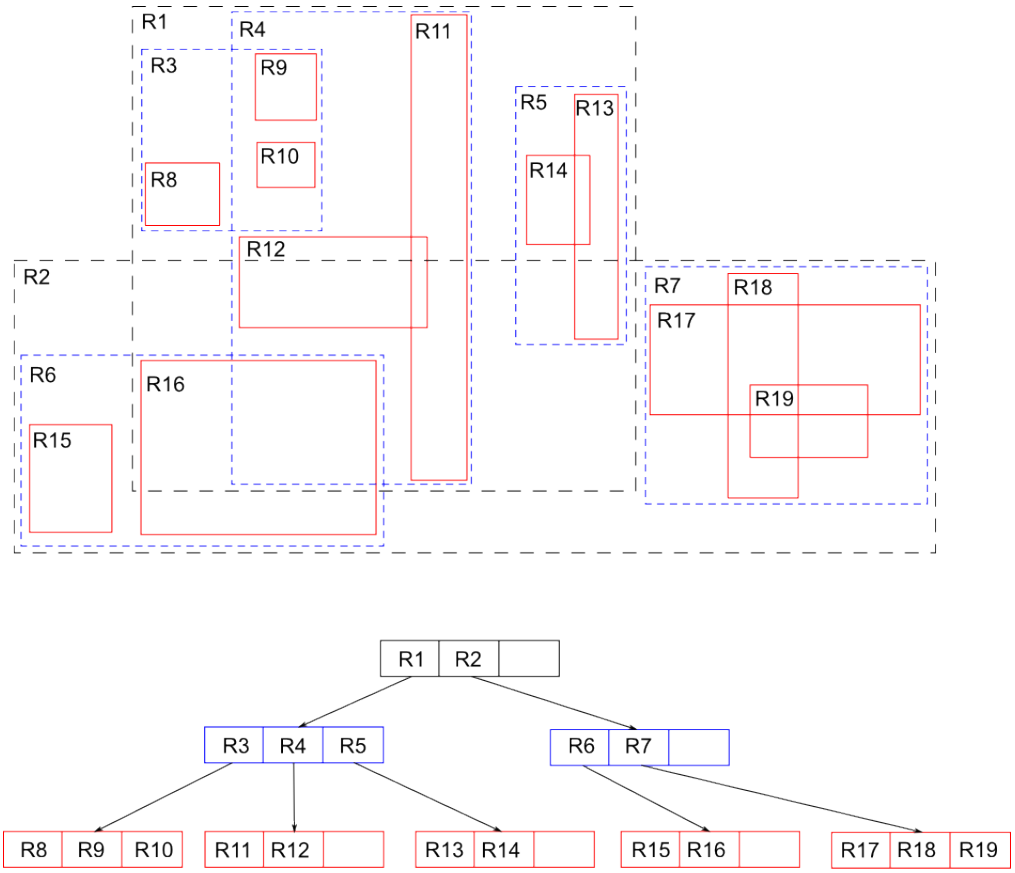

FIGURE 2.2 – Représentation d'un R-tree

Pour plus de détails sur le R-tree on pourra se référer à l'article de A. Guttman [Gut84]

Une telle structure semble bien plus intéréssante en terme d'occupation mémoire. En effet le nombre de feuilles de l'arbre est au pire égale à $\frac{N}{m}$. Cependant on peut se demander si rechercher une fenêtre de visualisation serait efficace. En effet il est

nécessaire de parcourir toutes les boîtes dont l'intersection est non nulle avec la fenêtre. De nombreuses analyses ont été effectuées sur le sujet. On pourra se reporter à une analyse de performances sur les Priority R-trees [AdBJY04].

Bibliographie

[AdBJY04] Lars Arge, Mark de Berg, Herman J.Haverkort, and Ke Yi. The priority r-tree : A practically efficient and worst-case optimal r-tree. http://daimi.au.dk/~large/Papers/prtreesigmod04.pdf, 2004.

[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.

[Wik] Wikipedia. R-tree. http://en.wikipedia.org/wiki/R-tree.