Indexation dans les espaces métriques Index arborescent et parallélisation

Zineddine Kouahla

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Zineddine KOUAHLA

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
sous le label de l'Université de Nantes Angers Le Mans

École doctorale : Sciences et technologies de l'information, et Mathématiques

Discipline : Informatique et applications
Spécialité : Informatique
Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 14 février 2013
Thèse n° : ED 503-183

Indexation dans les espaces métriques
Index arborescent et parallélisation

JURY

Rapporteurs :	**M^me^ Jenny BENOIS-PINEAU**, Professeur des universités, Université Bordeaux 1
	M. Olivier PIVERT, Professeur des universités, ENSSAT
Examinateurs :	**M. Laurent AMSALEG**, Chargé de recherche CNRS, IRISA
	M. Pascal MOLLI, Professeur des universités, Université de Nantes
Directeur de thèse :	**M. José MARTINEZ**, Professeur des universités, École polytechnique de l'université de Nantes

Remerciements

En trois ans, je suis passé de l'enthousiasme de commencer un nouveau projet, aux doutes et au découragement face aux nombreuses difficultés et à l'exigence de ce travail, à la satisfaction enfin de le voir progresser et évoluer. En octobre 2009, a débarqué à Nantes un jeune étudiant timide, il en repartira en février 2013 un jeune chercheur débutant beaucoup plus confiant en lui-même. Cette évolution est le fruit des nombreuses rencontres que je souhaite remercier ici pour toute l'aide, le soutien, les conseils qu'elles m'ont apportés ou tout simplement pour leur bonne humeur et leur joie de vivre qui ont fait de ces trois années, une aventure incroyable.

Je souhaite remercier mon directeur de thèse, M. José Martinez pour m'avoir accueilli au sein de son équipe. Je lui suis également reconnaissant pour le temps conséquent qu'il m'a accordé, ses qualités pédagogiques et scientifiques, sa franchise et sa sympathie. J'ai beaucoup appris à ses côtés et je lui adresse ma gratitude pour tout cela.

Je voudrais remercier mes rapporteurs de cette thèse, Mme Jenny Benois-Pineau, Professeur des universités à l'université de Bordeaux 1, et M. Olivier Pivert, Professeur des universités à l'ENSSAT, université de Rennes 1, pour l'intérêt qu'ils ont porté à mon travail.

J'associe à ces remerciements M. Laurent Amsaleg, Chargé de recherche CNRS à l'IRISA, et M. Pascal Molli, Professeur des universités à l'université de Nantes, pour avoir accepté d'examiner mon travail.

Je ne saurais que remercier Mmes Florence Sédes et Elisabeth Murisasco pour leurs implications durant mes trois ans de thèse dans le comité de suivi de thèse. Elles m'ont beaucoup appris, j'ai apprécié leurs enthousiasmes et leurs sympathies.

Je tiens à remercier tous les membres de l'équipe GRIM, pour leur bonne humeur. Nous avons partagé de bons moments.

Enfin, j'adresse de tendres remerciements à toute ma famille, mes parents, mes frères et ma sœur, et spécialement à *Gabriela* et le petit *Abdel-karim* d'avoir été présents dans mes moments de doutes et de « craquages », de m'avoir remonté le moral et encouragé.

Table des matières

1

Introduction

Comme le titre de ce mémoire l'indique, dans ce travail, nous nous sommes intéressés à l'indexation de données dans un espace métrique. Ce besoin est né de la nécessité d'indexer et de rechercher des données de plus en plus nombreuses et complexes.

Introduisons donc le contexte général dans lequel s'inscrit ce travail, ayant amené à une problématique à résoudre, des contraintes imposées et des idées préliminaires, et ayant finalement abouti à une contribution.

1.1 Contexte

Avec la révolution numérique de ces dernières décennies, la quantité de données numériques a explosé. La mise à la portée aux utilisateurs de certain outils comme les appareils photo numériques, les numériseurs (*scanners*), cyber-caméras (*webcams*) et téléphones portables, la montée en puissance des capacités de stockage des ordinateurs, ainsi que la démocratisation d'Internet, ont contribué à cette explosion. Notons que, de nos jours, l'information multimédia est l'information la plus répandue et la plus utilisée dans différents domaines y compris dans notre vie quotidienne [54].

De cette explosion est né le besoin de gérer d'immenses volumes de données pour profiter au mieux de cette masse d'information, et cela pour un public de plus en plus large. L'indexation et la recherche efficients dans de grandes collections de données est donc un problème d'actualité des plus pressants [78, 92, 54, 61, 16].

Traditionnellement, les bases de données ont géré des recherche exactes, notamment dans le cas des bases de données structurées contenant des données nu-

mériques et/ou alphabétiques. Les méthodes utilisées pour faire face à ces bases de données ont été efficaces. Mais, l'augmentation de la taille des bases de données et surtout l'apparition de nouveaux types de données ont changé la problématique.

Les bases de données modernes contiennent des types de données tels que du multimédia (images, audio et vidéo) mais aussi des séries chronologiques, des empreintes digitales, des séquences d'ADN, des documents, etc.

Un autre changement important est le type de requêtes posées évolue également. Face à des données parfois imprécises, les recherches exactes des bases de données traditionnelles n'ont pas toujours beaucoup de sens, et des recherches par similarité leur sont préférables (dès lors que l'on dispose d'un exemple à fournir).

Ces évolutions nous éloignent pour partie du modèle traditionnel des bases de données structurées pour nous rapprocher de celui de la recherche d'information, traditionnellement concentrée sur la recherche de documents textuels. Inversement, la plus grande richesse de description des données oblige à envisager des descriptions sinon structurées du moins semi-structurées. Ces descriptions complexes sont alors plus difficiles à indexer et à rechercher que de simples textes. L'évolution « naturelle » est donc de chercher à fournir des systèmes d'indexation et de recherche qui soient aussi bien utilisables dans le cadre des bases de données que de la recherche d'information.

1.2 Problématique

La problématique qui nous intéresse est alors celle de fournir des techniques d'indexation pour les recherches par similarité. Insistons que le fait que l'utilité de l'information dépend non seulement de sa qualité, mais aussi de la vitesse à laquelle elle est récupérée, ce qui, à son tour, dépend de la façon dont elle est indexée. La recherche par similarité *efficiente* dans une collection de données reste donc un problème fondamental en informatique.

Durant ces cinquante dernières années, plusieurs méthodes d'indexation ont été proposées. Il ne nous semble pas possible de les comparer toutes entre elles, d'autant plus que de nombreuses hypothèses différentes peuvent être à l'origine de leur proposition et que leur efficacité est liée à différents facteurs (type de données utilisé, qualité de la machine de calcul, etc.) [37, 90].

Les petites collections de données, qui contiennent de plus des objets simples, peuvent être manipulées facilement. Mais la gestion de grandes bases, comme la plupart des bases de données en usage aujourd'hui, nécessite des techniques plus sophistiquées, surtout quand elles contiennent des types de données complexes.

En effet, les objets à indexer sont souvent plus complexes que de simples vecteurs (homogènes – ex. : espaces vectoriels – ou hétérogènes – ex. : n-uplets dans

une base de données relationnelle). Par exemple, dans une base de données multimédia, on pourra retrouver pêle-mêle des descriptions « sémantiques » (comme des simples mots clés mais jusqu'à des graphes de description conceptuels associés à une ontologie) et des descriptions sur le contenu des médias (comme des histogrammes de couleurs).

Tout cela soulève des questions sur les méthodes d'indexation et de recherche. Par conséquent, la mise au point de l'indexation a été partiellement transférée des espaces multidimensionnels vers les espaces métriques, afin d'exploiter non pas la représentation des données elles-mêmes, devenues trop riches et complexes, mais de travailler « seulement » sur les similitudes qui peuvent être calculées entre les objets.

Intrinsèquement, les difficultés d'indexation dans les espaces multidimensionnels persistent dans une version généralisée, tandis que de nouvelles difficultés surgissent où en raison du manque d'informations sur les objets.

1.3 Contraintes préalables

Au delà des difficultés déjà posées par la généralisation depuis les espaces multidimensionnels vers les espaces métriques, des contraintes, parfois techniques, doivent être imposées au préalable sur les solutions possibles.

Le problème donc est de proposer une technique d'indexation, sachant que toute solution doit respecter certaines contraintes proposées par d'autres auteurs [64] :

- *(C1) passage à l'échelle* (*scalability*) : La structure d'index doit être capable d'indexer des volumes de données toujours croissants sans perte de performances (efficacité pour répondre aux requêtes).

- *(C2) efficience* : Contrairement aux structures d'accès physiques habituelles, qui ne tentent « que » de minimiser le nombre d'entrées-sorties sur disque, un index de recherche par similarité doit aussi prendre en compte les coûts de calcul car les calculs de similarité entre deux objets peuvent être très coûteux, voire plus coûteux que les temps de lecture sur disque.

- *(C3) dynamicité* : La structure d'index se doit d'être dynamique vis-à-vis des changements de la base de données, c'est-à-dire de ne pas nécessiter des réorganisations périodiques coûteuses mais d'assurer une indexation continue.

- *(C4) indépendance aux données* : Les structures d'accès devraient offrir de bonnes performances pour toutes distributions possibles des données, c'est-à-dire répondre efficacement aux problèmes liés à la « malédiction de la dimension » ou *inefficacité dans les grandes dimensions* (cf. section 2.6).

Répondre à tous ces critères est encore d'actualité :

1. En ce qui concerne (C1), nous devons rejeter les structures d'indexation qui travaillent sur des bases de données connues d'avance, c'est-à-dire qui ont besoin des informations sur l'ensemble des données avant de procéder à l'étape d'indexation. Comme contre-exemple, l'arbre-VP [18, 91] calcule des médianes sur l'ensemble des données afin de regrouper les objets.

   En pratique, cela se traduit par la limitation à des algorithmes *incrémentaux*.

   Par ailleurs, la complexité maximale en temps à retenir pour un algorithme d'indexation est en $O(n.\log n)$ dès lors que l'on travaille sur un grande nombre de données, *n*, et que les données sont de surcroît stockées sur disque [76].

   Bien évidemment, pour les algorithmes de recherche, le but est d'être aussi rapide que possible, idéalement en $O(\log n)$!

2. À l'égard de (C2), nous devons rejeter les structures d'indexation qui ne fonctionnent qu'en mémoire principale.

3. Ensuite, par rapport à (C3), nous rejetons les structures de données qui se basent sur le calcul de la matrice des distances entre tous les éléments, ce qui conduira immédiatement à des algorithmes en $O(n^2)$ où n est le nombre d'éléments dans la base, c'est-à-dire inutilisables en pratique pour les grandes collections de données.

4. Enfin, pour (C4), nous avons également éliminé les structures d'indexation qui travaillent dans des hypothèses restrictives, en particulier celles qui exigent des distances discrètes, par exemple l'arbre de Burkhard-Keller [21] (même si elles ont des applications intéressantes comme dans les correcteurs orthographiques).

Nous faisons donc nôtres ces contraintes, mais sommes obligés de fixer un cadre plus rigide encore pour une proposition de solution.

1.4 Idées préliminaires

Des expérimentations préalables dans un environnement parallèle, similaires à d'autres [3], nous ont permis de bien cerner les problèmes liés à l'indexation dans un espace métrique.

Nous avons alors confirmé le choix de certaines directions de travail, basées pour partie sur les contraintes préalables introduites ci-dessus et sur des idées préliminaires, voire des « intuitions », dont les expériences préalables nous faisait penser à leur fondement.

Ces idées préliminaires nous amènent à construire un index :

- arborescent ;
- de manière incrémentale ;
- de complexité en temps limitée en $O(n.\log n)$;
- paginé ;
- quasi équilibré ;
- parallélisable.

Index arborescent. Tout d'abord, nous avons opté pour l'indexation au travers de structures arborescentes. Il s'agit peut-être d'un parti pris mais, dans la littérature sur l'indexation dans les espaces multidimensionnels, cette approche domine largement et elle est quasi exclusive dans les espaces métriques.

Même si d'autres approches existent (grilles dans les espaces multidimensionnels, graphes dans les espaces métriques), nous n'avions pas le temps pour une recherche d'une telle ampleur.

De par l'absence de solution satisfaisante, à notre connaissance, et à l'intérieur de ce vaste champ de connaissances, technologies et techniques encore imparfaites, nous souhaitons nous attaquer sinon à la résolution du moins à l'amélioration de l'indexation dans les espaces de données métriques.

Incrémentalité et complexité en temps. Une construction incrémentale de l'index nous semble un passage obligé, de même que la limitation de la complexité de sa construction. En fait, ce travail se place dans la poursuite de la thèse de Jorge Manjarrez [56]. Il y est proposé d'appliquer un algorithme de classification (les *k-means*) dans une phase hors-ligne pour organiser les données. Le problème majeur de cette technique est qu'elle est très lente pendant la construction de l'index, avec une complexité en $O(n^2.\log n)$, sans même tenir compte des calculs de distances.

Version paginée. Nous faisons le choix d'un implémentation paginée de l'index. Le but est double. Il s'agit de permettre une implémentation sur disque. Il s'agit également de pouvoir prendre des décisions sur un « échantillon » des données plutôt que de devoir subir les effets d'insertions aléatoires dans une structure maintenue de manière incrémentale.

Équilibrage. Bien qu'il s'agisse d'une version paginée à exploiter sur disque, nous ne cherchons pas à imposer un arbre parfaitement équilibré, à la manière d'un arbre-B traditionnel. Il nous semble qu'un quasi équilibrage est suffisant. Il devrait permettre d'éviter des regroupements arbitraires d'éléments, c'est-à-dire de

s'approcher d'une classification hiérarchique qui, par nature, n'a pas à présenter un aspect parfaitement balancé. Toutefois, des déséquilibres importants ne sont pas souhaitables non plus et nous veillerons à ce que la solution retenue amène à des index à peu près équilibrés.

Parallélisme. Face aux difficultés liées à la dimension intrinsèque, il nous semble qu'aucune technique d'indexation ne peut permettre d'atteindre des temps de recherche logarithmiques. Manjarrez [57] obtient des temps de réponse en $O(\sqrt{n})$ mais avec une phase d'organisation préalable coûteuse et sur une grappe de $O(\log n)$ machines.

Or, nous savons qu'un temps de réponse logarithmique est atteignable. Les implémentations sont faciles sous un modèle d'exécution parallèle abstrait. En séquentiel, un tri général est en $O(n.\log n)$ où n est le nombre d'objets. Sur une machine parallèle, un tri peut être mis en œuvre en $O(\log n)$ en temps et en $O(n)$ en surface, à savoir le nombre de processeurs [89].

Le problème avec cette solution théorique porte clairement sur la surface. Il est impossible, et par ailleurs déraisonnable, d'utiliser $O(n)$ processeurs avec de grandes bases de données contenant des millions ou des milliards d'objets ! En outre, le calcul de la fonction de distance pourrait certainement bénéficier du parallélisme et l'on peut imaginer que les fonctions simples peuvent être mises en œuvre en $O(\log d)$ (où d le nombre de dimension) pour la complexité en temps et en $O(d)$ pour la surface. Dans la solution globale, la complexité en temps est alors uniquement en $O(\log k.\log d)$ (où k la taille de requête de plus proches voisins), mais avec une surface en $O(n.d)$...

En d'autres termes, le problème de l'indexation de la recherche de manière efficace dans les espaces métriques peut être considéré comme déjà résolu du point de vue *théorique*, mais il reste une tâche très complexe lors de la prise en compte des ressources limitées et communes des ordinateurs, ainsi que les inhérentes (mauvaise) propriétés des espaces métriques (la distance est la seul information). Cela s'applique à n'importe quelle architecture raisonnable, soit un super-calculateur, une grappe de machines ou encore un réseau pair-à-pair. Par conséquent, le parallélisme peut et doit certainement faire partie d'une solution, mais ne peut pas être la seule réponse [44, 57].

Le but est donc de permettre la parallélisation de l'algorithme de recherche sur un nombre raisonnable et extensible de machines.

1.5 Contributions

Nous pouvons déjà percevoir notre contribution comme une proposition d'une structure d'index et d'un algorithme de recherche, séquentiel dans un premier temps, qui soit le plus rapide possible. Nous verrons que la structure même de l'index place une borne inférieure à cette rapidité.

Pour aller au delà, voire espérer atteindre le cas idéal d'une recherche en temps logarithmique, le parallélisme est alors introduit.

Ce travail a donné lieu à trois propositions d'index successives.

La première [49, 48] peut être vue essentiellement comme une paramétrisation d'une proposition antérieure, l'arbre-MM [65]. L'espace est découpé récursivement au travers de deux boules (ou hyper-sphères) en distinguant (i) leur intersection, (ii) leurs différences réciproques et (iii) le reste de l'espace. Le but poursuivi dans ce travail est de mieux contrôler les volumes des régions de l'espace et la distribution des données dans ces régions.

La structure d'index qui fait l'objet essentiel de ce mémoire améliore les performances de la proposition précédente en cherchant à limiter les concavités dans les sous-espaces [50]. Pour cela, un hyper-plan est ajouté. Nous avons donc un découpage légèrement plus complexe. Outre le découpage induit par les hyper-sphères, décompose l'espace en deux sous-parties, « gauche » et « droite ».

Les résultats expérimentaux soutiennent les choix faits dans une version suivante qui fait davantage partie des travaux à venir [59] car elle n'a pas subi toutes les expérimentations présentées ici.

1.6 Plan de l'étude

Cette thèse est divisée en deux parties.

La première partie présente le contexte général. Une revue de la littérature connexe s'achève sur un chapitre de synthèse.

La deuxième partie est consacrée au développement de nos propositions ainsi qu'à leurs tests pour une validation expérimentale.

En reprenant chapitre par chapitre, le chapitre 2 présente les notions mathématiques utiles pour la présentation et la compréhension des différentes techniques d'indexation. Ces notions concernent essentiellement les espaces métriques dans leur généralité.

Toutefois, dans le chapitre 3, nous présentons les principales techniques d'indexation arborescentes dans les espaces multidimensionnels, ainsi que leurs algorithmes de construction et recherche de plus proches voisins.

Le chapitre 4 fait une étude similaire sur les principales techniques d'indexation arborescentes dans les espaces métriques.

À la fin de cette partie, le chapitre 5 effectue une synthèse et une comparaison entre toutes les techniques présentées précédemment. Nous nous intéressons ici à mettre en évidence les différences et similitudes entres ces approches.

Au début de la deuxième partie, dans le chapitre 6, nous proposons une méthode d'indexation que nous baptisons l'« arbre-IM ». La présentation est bâtie sur le même principe et les mêmes étapes que pour l'étude des index de l'état de l'art. Le but est de faciliter une comparaison de notre approche avec les propositions antérieures. Cette dernière démontre une certaine simplicité de conception et de mise en œuvre.

La proposition est validée sur différents types de données réels, les résultats expérimentaux étant présentés dans le chapitre 7.

Après, dans le chapitre 8, nous abordons l'amélioration de l'efficacité des recherches dans l'arbre-IM au travers de la parallélisation.

Cette voie est également validée expérimentalement par la suite, dans le chapitre 9.

Finalement, dans le chapitre 10, nous énonçons quelques conclusions au sujet de nos résultats et des orientations futures de recherche.

I

État de l’art

L'indexation est une étape d'organisation des données qui doit permettre d'accéder de manière efficiente à ces dernières lors de l'exécution de requêtes de similarité. Le principe est d'organiser les données similaires pour accélérer les recherches [15]. Le but de tout index est donc l'accès rapide aux objets d'une base, en réduisant l'espace de recherche, le coût des entrées-sorties et le nombre de calculs de distances entre les objets. En d'autres termes, l'index fournit la mise en œuvre efficace de la recherche associative [36].

La nature et la structure des index a fait, et fait toujours, l'objet de très nombreuses recherches, les contextes étant assez variables : bases de données relationnelles, bases textuelles, bases multimédias, etc. En ce qui nous concerne, nous nous intéressons à l'indexation de données *complexes*, c'est-à-dire pour lesquelles aucune propriété particulière liée à la nature des données ne peut être exploitée. (Par exemple, sur des données discrètes, on peut utiliser des tableaux de bits – *bitmaps*.)

Par ailleurs, nous nous intéressons à des requêtes de plus proches voisins (*k*-ppv ou kNN). On retrouve cette forme d'interrogation dans les bases de données traditionnelles *via* les requêtes dites « *top-n* », implémentées de différentes façons dans les systèmes de gestion de bases de données. On retrouve également ce type de requête dans les bases de données moins traditionnelles comme les systèmes de gestion de bases de données géographiques, textuelles et multimédias. Par exemple, dans les bases de données multimédias, le traitement des requêtes par le contenu fonctionne par l'exemple, c'est-à-dire en recherchant les objets les plus proches d'un objet multimédia fourni en exemple au travers d'une distance censée traduire au mieux la perception humaine.

L'indexation d'un espace multidimensionnel, et au delà d'un espace métrique, est donc une approche importante.

Le but de cette partie est donc de faire une analyse des méthodes d'indexation qui permettent de traiter les requêtes kNN – dans les espaces multidimensionnels puis dans les espaces métriques –, en comparant leurs structures, leurs algorithmes, leurs avantages et inconvénients, et les performances atteintes. Pour faciliter cette tâche de comparaison, les structures et les algorithmes seront traduits dans un formalisme aussi commun que possible au risque d'améliorer les versions originelles dans cet état de l'art ou... d'introduire des erreurs de détail !

2

Brève introduction aux espaces métriques

Les hypothèses que nous posons sur les données nous amènent à travailler dans un espace métrique. Il existe plusieurs avantages à effectuer des recherches dans un espace métrique. Le plus important est qu'un plus grand nombre de types de données peuvent être indexés, car l'approche est basée seulement sur le calcul de distances entre les objets et non sur leur contenu [93].

Nous introduisons toutefois quelques éléments sur les espaces multidimensionnels puisqu'il s'agit de cas particuliers d'espaces métriques. Dans un espace vectoriel, les objets sont représentés par des vecteurs et les propriétés géométriques de ces vecteurs sont exploitables pour la recherche. Bien sûr, ces propriétés ne peuvent pas être étendues aux espaces métriques [25]. Toutefois, ils sont sujets aux mêmes problèmes et nous verrons par la suite qu'ils amènent donc à des solutions sinon identiques du moins similaires.

Notons enfin que, si les espaces multidimensionnels sont *de facto* des espaces métriques puisqu'ils disposent d'une métrique – une norme –, il est possible de transformer arbitrairement un espace métrique en un espace vectoriel. Nous verrons rapidement cela dans les deux sections de ce chapitre.

Commençons donc par le cas général où nous introduisons également les définitions sur les plus proches voisins.

2.1 Espace métrique

En mathématiques, un espace métrique est un ensemble auquel on associe une notion de distance entre les éléments sous la forme d'une fonction également dénommée métrique.

Définition 1 (Métrique) *Soit $\mathcal{O}$ un ensemble d'éléments. Soit d une fonction de distance d'un couple d'éléments de $\mathcal{O}$ vers un réel positif ou nul :*

$$d : \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+ \tag{2.1}$$

c'est-à-dire vérifiant les quatre axiomes suivants :

1. *positivité (par définition) :*

$$\forall (x, y) \in \mathcal{O}^2, d(x, y) \geq 0 \; ; \tag{2.2}$$

2. *identité :*[1]

$$\forall x \in \mathcal{O}, d(x, x) = 0 \; ; \tag{2.3}$$

3. *symétrie :*

$$\forall (x, y) \in \mathcal{O}^2, d(x, y) = d(y, x) \; ; \tag{2.4}$$

4. *inégalité triangulaire :*

$$\forall (x, y, z) \in \mathcal{O}^3, d(x, z) \leq d(x, y) + d(y, z). \tag{2.5}$$

Alors $(\mathcal{O}, d)$ est un espace métrique.

2.2 Espace multidimensionnel

Un espace multidimensionnel est défini lorsque les éléments de l'ensemble considéré sont des vecteurs, homogènes ou hétérogènes, dont les composantes sont totalement ordonnées.

Le cas le plus fréquent est celui des sous-espaces orthonormés, c'est-à-dire définis sur $\mathbb{R}^n$ et munis d'une norme, c'est-à-dire des espaces vectoriels.

Nous ne nous attardons pas sur ces derniers. Nous soulignons seulement que les espaces vectoriels sont des sous-ensembles des espaces métriques. En effet, toute norme sur un espace vectoriel est également une métrique, mais l'inverse n'est pas nécessairement vrai. (Pour les espaces multidimensionnels quelconques,

[1] Par souci de généralisation, cet axiome est parfois écrit $\forall (x, y) \in \mathcal{O}^2, d(x, x) = d(y, y)$.

il faut fournir une distance et il deviennent alors naturellement des espaces métriques (cf. section suivante).) Cela peut devenir un avantage, une distance donnée révélant la dimension *intrinsèque* (cf. page 26)des objets plutôt que la dimension « réelle » des vecteurs (cf. section 2.6.1).

En sens inverse, il est possible de plonger arbitrairement un espace métrique dans un espace vectoriel. Pour cela, il suffit de choisir un nombre $n > 0$ d'éléments de l'espace vectoriel, distincts les uns des autres, et de les ordonner arbitrairement. Les distances de chacun des éléments de l'espace métrique à ces éléments-pivots, ordonnées de la même façon, fournissent des vecteurs dans $\mathbb{R}^n$. Une telle approche est utilisée par *FastMap* [35] pour réduire la dimension des données.

Il existe donc des nombreux points communs ou rapprochements possibles entre ces deux familles d'espaces de données.

2.3 Fonctions de distance

Les fonctions de distance peuvent être adaptées à une application spécifique ou à un domaine d'application donné. Elles sont alors spécifiées par un expert. Cependant, la définition d'une fonction de distance donnée n'est généralement pas limitée à un unique type de requêtes. Il existe de nombreuses d'un usage assez vaste.

Les fonctions de distance peuvent être divisées en deux groupes suivant les valeurs retournées :

- Les fonctions de distance *discrètes* ne renvoient qu'un petit ensemble de valeurs. Un exemple du cas discret est la distance d'édition entre deux chaînes de caractères (nombre de caractères à insérer, supprimer ou déplacer) ou encore la distance de HAMMING entre chaînes de bits (nombre de couples de bits différents deux à deux). Des techniques d'indexation spécifiques ont été proposées pour ces cas-là.

- Pour les fonctions de distance *continues*, le cardinal de l'ensemble des valeurs renvoyées par une fonction de distance est sinon véritablement infini du moins très grand. Un exemple du cas continu est la distance euclidienne (cf. ci-dessous, formule (2.8)).

Les fonctions de distance peuvent également être classées en fonction de leur coût de calcul, en retenant tout simplement leur complexité asymptotique. Les plus simples sont linéaires en la taille de l'objet, ce qui est optimal. Beaucoup sont déjà quadratiques, comme les distances entre ensembles ou la distance d'édition mentionnée ci-dessus. Mais d'autres peuvent présenter des complexités encore supérieures. Dans nos travaux nous ne nous intéressons pas directement aux dis-

tances utilisées mais tenons compte de cette contrainte et tentons de minimiser le nombre de calculs de distances.

À titre d'illustration, fournissons un exemple emblématique de métrique, la famille des distances de MINKOWSKI.

Distances de MINKOWSKI. Les distances de MINKOWSKI forment une famille de fonctions métriques, appelées « métriques L_p », qui peuvent être appliquées sur différents types de données à partir du moment où l'on peut les considérer, voire transformer, en vecteurs de nombres.

Définition 2 (Métriques L_p) *Soit $(x_1, \ldots, x_n) \in \mathbb{R}^n$ et $(y_1, \ldots, y_n) \in \mathbb{R}^n$ deux vecteurs. Soit $p \in \mathbb{R}$ avec $p \geq 1$.*

Alors, on définit :

$$L_p((x_1, \ldots, x_n), (y_1, \ldots, y_n)) = \sqrt[p]{\sum_{\forall i} |x_i - y_i|^p}. \tag{2.6}$$

Quelques valeurs particulières du paramètre p sont plus souvent utilisées. Il s'agit des métriques :

- L_1, communément appelée distance de Manhattan, qui se récrit plus simplement :
$$L_1((x_1, \ldots, x_n), (y_1, \ldots, y_n)) = \sum_{\forall i} |x_i - y_i| \; ; \tag{2.7}$$
- L_2, qui n'est autre que la distance euclidienne :
$$L_2((x_1, \ldots, x_n), (y_1, \ldots, y_n)) = \sqrt{\sum_{\forall i} (x_i - y_i)^2} \; ; \tag{2.8}$$
- L_∞, ou distance de CHEBYSHEV, communément appelée distance de l'échiquier, se ramène à :
$$L_\infty((x_1, \ldots, x_n), (y_1, \ldots, y_n)) = \max_{\forall i} |x_i - y_i|. \tag{2.9}$$

La figure 2.1 illustre ces membres-là de la famille L_p. Chaque courbe représente un ensemble de points, dans le plan, à une même distance du point central. On y reconnaît le cercle pour la métrique L_2. On voit que L_1 se traduit par un losange, tandis que L_∞ donne un carré. Les valeurs intermédiaires produisent un renflement progressif depuis le losange vers le carré en passant par le cercle.

Sur la figure 2.2, il est montré la distance à partir d'une requête q à un point p_i selon les trois normes citées avant.

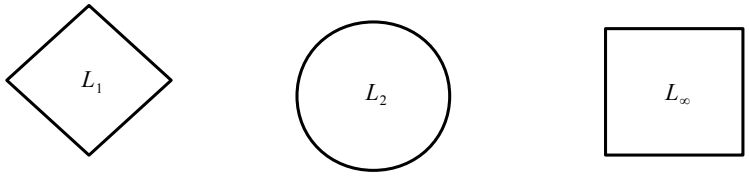


FIGURE 2.1 – Ensembles de points à la même distance du point central pour les métriques L_1, L_2 et L_∞ (repris de [56])

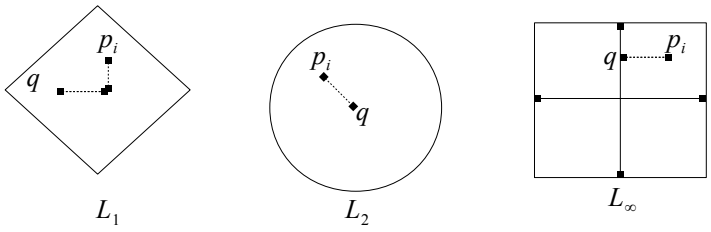


FIGURE 2.2 – Formes induites par différentes distances (repris de [56])

Une application pratique des métriques L_p est le calcul de distance entre les histogrammes de couleurs ou entre les vecteurs de caractéristiques extraites à partir d'images. Nous en ferons usage au chapitre 7.

2.4 Notions de boule et d'hyper-plan

La géométrie classique peut être bousculée par l'emploi d'autres métriques. De manière générale, la fonction utilisée pour évaluer la distance entre éléments détermine une géométrie, ou topologie, de l'espace. Dans les espaces métriques, les auteurs exploitent souvent deux concepts clés :

- la boule ;
- l'hyper-plan.

Nous les emploierons également dans nos propositions. Présentons-les.

2.4.1 Boule

Une boule est une notion topologique qui généralise celle de disque dans le plan euclidien et de sphère dans l'espace. On parle d'ailleurs souvent d'hyper-sphère,

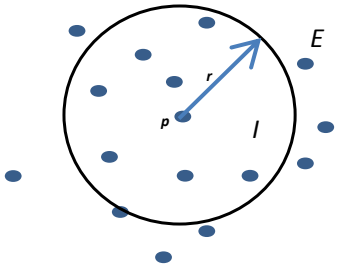


FIGURE 2.3 – Boule

même si le terme ne s'applique *stricto sensu* qu'à des espaces multidimensionnels. C'est un sous-ensemble d'un espace métrique définit par un objet central, ou « pivot » p, et un rayon r.

Définition 3 (Boule fermée) *Soit $(\mathcal{O}, d)$ un espace métrique. Soit $p \in \mathcal{O}$ un objet pivot et $r \in \mathbb{R}^+$ le rayon de couverture.*

Alors $B(\mathcal{O}, d, p, r)$ – ou seulement $B(p, r)$ lorsqu'il n'y a pas d'ambiguïté sur l'espace métrique – définit une boule qui partitionne l'espace en deux parties :

$$\begin{aligned} I(\mathcal{O}, d, p, r) &= \{o \in \mathcal{O} : d(p, o) \leq r\} \ ; \\ E(\mathcal{O}, d, p, r) &= \{o \in \mathcal{O} : d(p, o) > r\} \ ; \end{aligned} \tag{2.10}$$

respectivement l'intérieur – noté seulement $I(p, r)$ lorsqu'il n'y a pas d'ambiguïté – (au sens large, ici) et l'extérieur – noté respectivement $E(p, r)$ – (au sens strict) de la boule.

On distingue la boule fermée, définie ci-dessus, de la boule ouverte. Dans cette dernière, les éléments situées à la distance exacte r du pivot, ne font pas partie de la boule. Cette distinction ne nous sera pas utile dans la suite.

La figure 2.3 illustre la définition dans le cas euclidien. Notons que le losange et le carré de la figure 2.2 sont aussi des « boules » !

Il est possible de définir des opérations ensemblistes entre boules, en particulier l'intersection et l'inclusion.

Définition 4 (Intersection entre boules) *Soit $(\mathcal{O}, d)$ un espace métrique. Soit une boule (fermée) de centre p_1 et de rayon r_1, $B_1(\mathcal{O}, d, p_1, r_1)$. Soit $B_2(\mathcal{O}, d, p_2, r_2)$ une boule (fermée) de centre p_2 et de rayon r_2.*

Alors il existe une intersection non vide entre B_1 et B_2 si, et seulement si :

$$d(p_1, p_2) \leq r_1 + r_2. \tag{2.11}$$

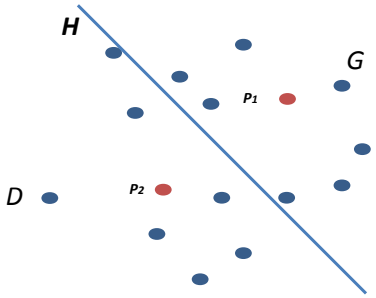


FIGURE 2.4 – Plan

Définition 5 (Inclusion entre boules) *Soit $(\mathcal{O}, d)$ un espace métrique. Soit une boule (fermée) de centre p_1 et de rayon r_1, $B_1(\mathcal{O}, d, p_1, r_1)$. Soit $B_2(\mathcal{O}, d, p_2, r_2)$ une boule (fermée) de centre p_2 et de rayon r_2.*

Alors la boule B_1 est incluse dans la boule B_2 si, et seulement si :

$$d(p_1, p_2) < r_2 - r_1. \tag{2.12}$$

2.4.2 Hyper-plan

Un deuxième concept important dans un espace métrique est celui d'hyper-plan. Un hyper-plan est *indirectement* déterminé par deux éléments *distincts*.

Définition 6 (Hyper-plan généralisé) *Soit $(\mathcal{O}, d)$ un espace métrique. Soit $(p_1, p_2) \in \mathcal{O}^2$ deux pivots, avec $d(p_1, p_2) > 0$.*

Alors $H(\mathcal{O}, d, p_1, p_2)$ – ou seulement $H(p_1, p_2)$ – définit un hyper-plan :

$$H(\mathcal{O}, d, p_1, p_2) = \{o \in \mathcal{O} : d(p_1, o) = d(p_2, o)\} ; \tag{2.13}$$

qui permet de partitionner l'espace en deux sous-espaces :

$$\begin{aligned} G(\mathcal{O}, d, p_1, p_2) &= \{o \in \mathcal{O} : d(p_1, o) \leq d(p_2, o)\} ; \\ D(\mathcal{O}, d, p_1, p_2) &= \{o \in \mathcal{O} : d(p_1, o) > d(p_2, o)\} ; \end{aligned} \tag{2.14}$$

soit respectivement le demi-plan « gauche » au sens large – $G(p_1, p_2)$ – et le demi-plan « droit » au sens strict – $D(p_1, p_2)$.

En d'autres termes, les objets de $G(p_1, p_2)$ sont plus proches de p_1 que de p_2 (ou à égale distance des deux), et les objets de $D(p_1, p_2)$ sont strictement plus proches de p_2 que de p_1.

La figure 2.4 illustre ce concept.

Il faut noter que, dans un espace métrique, un hyper-plan quelconque ne peut pas être défini. Il lui faut absolument s'appuyer sur la présence de deux élé-

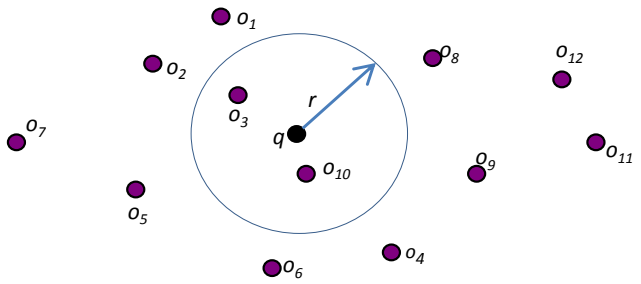


FIGURE 2.5 – Requête intervalle $R(q, r)$

ments de l'ensemble. Par ailleurs, dans un espace métrique général, des propriétés comme le parallélisme, l'orthogonalité, etc., ne sont pas toujours définissables.

2.5 Requête par similarité

Le concept de boule vu ci-dessus permet de définir le « voisinage » d'un élément. En terme de recherche, nous parlerons de « requête par similarité ».

Une requête par similarité est définie à partir d'un élément de référence. Cet élément n'appartient pas nécessairement à l'ensemble de référence considéré. À partir de là, il est possible de définir différents types de requêtes. Nous en présentons deux :

1. les requêtes par intervalles (traduction approximative de *range query*) ;
2. les requêtes de *k* plus proches voisins (*k*-ppv ou encore kNN pour *k-nearest-neighbours*).

2.5.1 Requête par intervalle

Une requête par intervalle est spécifiée comme le sous-ensemble de tous les éléments qui se trouvent à une distance inférieure à r d'un élément donné q. La figure 2.5 illustre ce type de requête. Cela peut être décrit formellement par la définition suivante.

Définition 7 (Requête par intervalle) *Soit $(\mathcal{O}, d)$ un espace métrique. Soit $q \in E$ un point requête. Soit $r \in \mathbb{R}^+$ la distance maximale autorisée à l'objet q.*

Alors $(\mathcal{O}, d, q, r)$ définit une requête d'intervalle, dont la valeur est :

$$R(\mathcal{O}, d, q, r) = \{e \in E : d(q, e) \leq r\}.$$

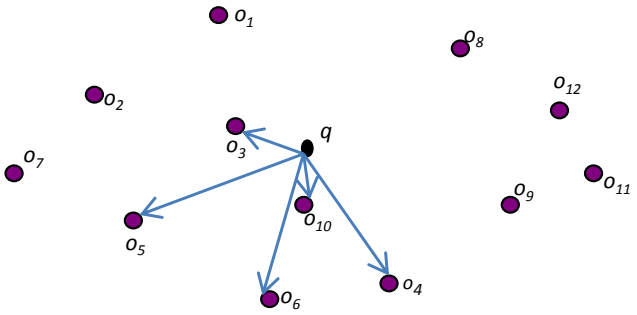


FIGURE 2.6 – Requête kNN

En fait, une requête par intervalle se ramène à déterminer le sous-ensemble des éléments d'un espace métrique appartenant à l'*extension* d'une boule.

2.5.2 Plus proches voisins

Dans une requête de plus proches voisins, nous recherchons les *k* objets les plus similaires à un élément donné. La requête n'est rien d'autre qu'un élément de référence q et le nombre d'éléments recherchés *k* (cf. figure 2.6).

L'ensemble des réponses n'est jamais vide, sauf dans des cas particuliers triviaux. En outre, sa taille est définie au préalable par l'utilisateur. Ce type de requêtes peut être formalisé par la définition suivante.

Définition 8 (Requête kNN) *Soit $(\mathcal{O}, d)$ un espace métrique. Soit $q \in E$ un élément requête. Soit $k \in \mathbb{N}$ le nombre recherché de réponses.*

Alors NN$(\mathcal{O}, d, q, k)$ définit une requête kNN, dont la valeur est $S \subseteq E$ de telle sorte que $|S| = k$ (sauf si $|E| < k$) et $\forall (s, e) \in S \times E, d(q, s) \leq d(q, e)$.

Dans le cas où plusieurs objets se trouvent à la même distance de la requête, le choix de fait d'une manière arbitraire [8, 14].

Ici aussi, il s'agit de déterminer un sous-ensemble en extension à partir d'une boule. La différence par rapport à une requête par intervalle vue précédemment et que le rayon r est initialement inconnu. Il sera déterminé *a posteriori* comme étant égal à la distance au k^e élément de la réponse. (Notons qu'il peut y avoir une toute petite différence entre les deux définitions lorsqu'il existe des éléments *ex æquo*. Autrement dit, lorsque la réponse à une requête kNN est connue, se servir de la k^e distance pour définir la requête par intervalle correspondante peut renvoyer un sur-ensemble.)

2.6 Dimension d'un espace métrique et problèmes liés

Nous avons déjà évoqué la difficulté de définir certaines notions classiques des espaces vectoriels dans les espaces métriques. La dimension est l'une d'entre elles.

2.6.1 Dimension intrinsèque

Les données multidimensionnelles présentent naturellement un nombre donné de dimensions. Dans les espaces métriques, cette notion disparaît, non seulement parce que l'on ne prend en compte l'objet que comme un tout et non pas comme un ensemble de composantes, mais aussi parce que certains objets sont naturellement sans dimension perceptible. C'est le cas d'une chaîne de caractères, d'un ensemble d'éléments quelconques, d'un graphe, etc.

Dans un espace métrique, la notion de « dimension » est alors traduite par des définitions autres. Une de ces définitions est celle de la dimension intrinsèque. Cette notion correspond à la notion de dimension « réelle » d'un ensemble de données, au sens où il serait possible de transformer toutes les données dans un espace vectoriel de cette dimension-là tout en conservant leurs distances respectives [25]. Une façon de définir une dimension intrinsèque est fournie ci-dessous.

Définition 9 (Dimension intrinsèque) *Soit $M = (\mathcal{O}, d)$ un espace métrique. Soit $E \subseteq \mathcal{O}$ un sous-ensemble d'éléments.*

Alors, on définit la dimension intrinsèque de E dans M par :

$$dim(E, M) = \frac{1}{2}\frac{\mu^2}{\sigma^2} \tag{2.15}$$

où :

- *μ est la moyenne des distances observées entre tout couple d'éléments :*

$$\mu = \frac{1}{|E|} \sum_{\forall (x,y) \in E^2} d(x, y) \; ; \tag{2.16}$$

- *σ^2 est sa variance :*

$$\mu = \frac{1}{|E|} \sum_{\forall (x,y) \in E^2} d(x, y)^2 - \mu^2. \tag{2.17}$$

Elle donne une mesure de la complexité de l'indexation d'un ensemble de données [75, 18, 85]. Elle a été établie de manière à retrouver approximativement, pour les distances L_1 et L_2, les dimensions d'un espace vectoriel uniformément peuplé. Soulignons que le paramètre principal de la formule est la variance. Plus

cette valeur est proche de zéro, plus il est difficile de distinguer les éléments les uns des autres, donc de les classer ou de les indexer. Cette constatation a également été faite préalablement pour les espaces vectoriels, le comportement statistique des données en grandes dimensions étant assez particulier.

En résumé, cette mesure de dimension intrinsèque permet de déterminer à quel point les éléments sont équidistants les uns des autres. Or, cela apparaît naturellement dans les espaces de grandes dimensions.

2.6.2 Quelques propriétés des espaces multidimensionnels

Revenons aux espaces multidimensionnels pour y relever des propriétés importantes liées à la notion de dimension.

Sous-espaces vides

Le problème principal dans les espaces de grandes dimensions est le phénomène des grands sous-espaces vides. Cela est dû au fait que les volumes des sous-espaces croissent de manière exponentielle avec le nombre de dimensions.

Illustrons cela au travers d'un découpage dichotomique de l'espace en forme de grille grossière. Pour un espace à deux dimensions, on divise chaque dimension en deux parties, ce qui donne $2^2 = 4$ sous-espaces. Un ensemble de quatre données pourrait être réparti uniformément et occuper donc chacune des quatre régions de l'espace. Toutefois, pour un espace étendu à trois dimensions, le même découpage dichotomique donne $2^3 = 8$ régions. C'est plus que le nombre d'éléments disponibles. Au moins la moitié de l'espace restera inoccupé. Avec une augmentation exponentielle du nombre de sous-espaces, 2^n pour n dimensions, l'espace est de plus en plus vide par rapport à un jeu de données considéré. Dit autrement, sous l'hypothèse d'une répartition des plus uniformes des données, il faut au strict minimum 2^n éléments pour avoir une (faible) chance de ne pas laisser de région vide. Par exemple, avec un espace à douze dimension, il ne faudrait pas moins de $2^{12} = 4\,096$ éléments, formant une grille quasi parfaite au demeurant. Encore, n'avons nous utilisé qu'un découpage dichotomique et non plus fin.

Voyons plus précisément le cas de l'hyper-sphère. Le volume de l'hyper-sphère est défini par la formule suivante [56] :

$$V_s(r) = \frac{r^d \pi^{d/2}}{\Gamma(d/2+1)} \tag{2.18}$$

où la fonction $\Gamma(x)$:

$$\Gamma(x) = \int_0^\infty e^{-x}.x^{n-1}dx, \tag{2.19}$$

est la généralisation de la factorielle à un réel.

d	1	2	3	4	5	6	7
$f_v(r)$	1	0,785	0,524	0,308	0,164	0,08	0,037

TABLE 2.1 – Variation du rapport du volume de l'hyper-sphère inscrite dans un hyper-cube

Après, en considère le volume de l'espace de données modélisé par $V_c(r) = 2r^d$ hyper-cube, puis pour un hyper-cube inscrit sous l'hyper-sphère, la fraction du volume contenu par l'hyper-sphère est :

$$f_v(r) = \frac{V_s(r)}{V_c(r)} = \frac{\frac{r^d \pi^{d/2}}{\Gamma(d/2+1)}}{2r^d} = \frac{\pi^{d/2}}{2^d(d/2+1)} \tag{2.20}$$

Lorsqu'on fait tendre le nombre de dimensions d vers l'infini, le volume de l'espace tend vers zéro :

$$\lim_{d\to\infty} f_v(r) = 0 \tag{2.21}$$

ce qui est illustré par les valeurs du tableau 2.1.

Cela signifie que pour les dimensions supérieures de l'espace retenu par l'inclusion hyper-sphère et hyper-cube sont presque vides et par conséquent toutes les structures d'indexation qui s'appuient là-dessus finiront par échouer.

En outre, le nombre de points inclus dans l'hyper-sphère tend vers 0 en tant que dimensions croît.

Une autre propriété, importante, est que, en grandes dimensions, le volume de l'hyper-sphère est très majoritairement concentré près de la surface [47]. Cela se traduit aussi par le fait que tous les points vont se retrouver à des distances proches les unes des autres. Ils vont donc être plus difficiles à distinguer.

2.6.3 Malédiction de la dimension

Si l'on s'intéresse maintenant aux métriques définies sur les espaces multidimensionnels, l'un des problèmes capitaux dans ce type des travaux est celui qui a donné lieu au terme « malédiction de la dimension ». Quand les distances sont basées sur un grand nombre de paramètres, et quand on les combine avec des sommes ou des moyennes (ce qui est le cas des métriques L_p vues ci-dessus), le théorème Central-Limite s'applique et la variance des distances observées tend vers zéro. Par conséquent, toutes les distances ont tendance à être proches les unes des autres ; les objets deviennent presque indiscernables et la réponse n'a guère de sens [12].

La malédiction de la dimension s'applique donc tout autant aux espaces vectoriels, à cause de ce théorème, qu'aux espaces métriques, au travers de la dimension intrinsèque, le facteur essentiel étant dans les deux cas la réduction de la variance observée sur les distances entre les éléments.

Cet effet complique, voire tend à rendre impossible, tout travail de classification ou d'indexation sur de telles données, c'est-à-dire de regroupements d'éléments proches et de séparation des groupes. Par contrecoup, pour la détermination de l'extension d'une requête (par intervalle ou kNN), il n'est plus possible de distinguer aisément entre candidats possibles et éléments à écarter d'emblée. *Les recherches ont tendance à dégénérer en analyse complète de l'ensemble des données. Les résultats ont tendance à basculer brutalement de l'ensemble vide à l'ensemble de tous les éléments.*

Heureusement, et comme cela est théoriquement démontré [12], il reste possible de distinguer les objets dans certains espaces de grandes dimensions, notamment dans le cas des données qui ne suivent pas une distribution uniforme, ce qui est effectivement le cas pour la plupart des données réelles.

3

Techniques d'indexation arborescentes dans un espace multidimensionnel

Dans ce chapitre, nous allons présenter quelques techniques d'indexation dans les espaces multidimensionnels. Nous ne recherchons absolument pas l'exhaustivité. D'autres auteurs ont proposé des ouvrages ou des synthèses de référence en la matière [36, 14, 73].

Nous ne traitons pas ici des approches basées sur un découpage à base de grille, comme les VA-files notamment [88, 87]. Comme notre travail concerne les espaces métriques généraux et non les espaces multidimensionnels, nous nous limitons aux apports et enseignements que l'on peut tirer de l'étude des principales propositions de la littérature, dans le cas des index arborescents.

Certains auteurs considèrent que les techniques d'indexation multidimensionnelles sont des méthodes de classification non supervisée [55]. Nous ne nous étendons pas sur ce sujet. Nous notons seulement, de nouveau, que, dans une classification, les classes ne sont pas de même cardinal et que, dans une classification hiérarchique, toutes les classes feuilles ne sont pas situées à la même profondeur.

On peut classer les techniques d'indexation dans les espaces multidimensionnels en deux grandes approches :

1. *le partitionnement de l'espace* qui utilise des cellules d'espace pour indexer les données ;

2. *le partitionnement des données* qui utilise des cellules d'objets similaires (fonc-

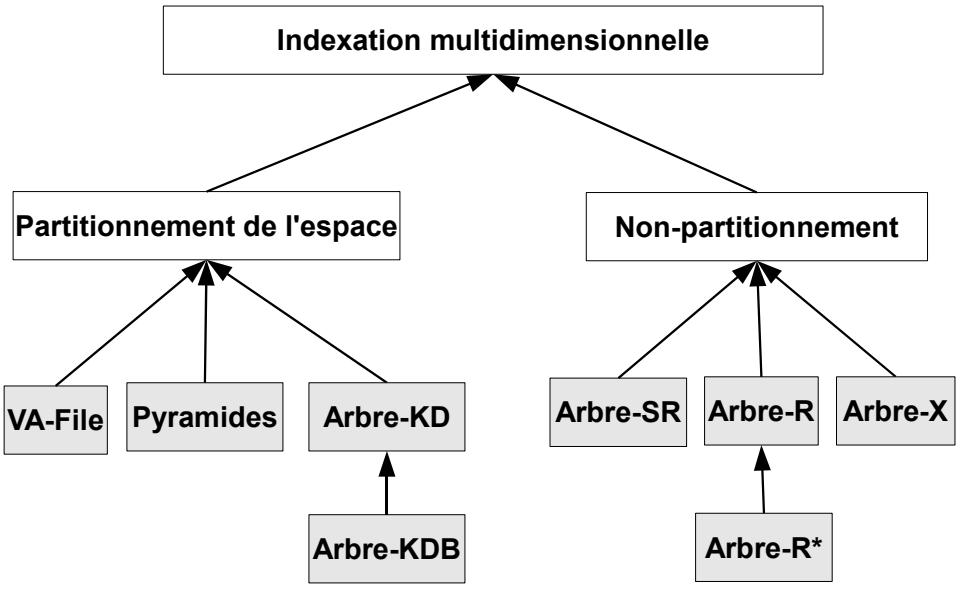


FIGURE 3.1 – Taxonomie simple de quelques techniques d'indexation multidimensionnelle

tion d'approximation) pour indexer les données. Qu'on va d'abord détailler cette approche.

3.1 Partitionnement des données

L'idée principale du partitionnement de données est la création de paquets regroupant des données, également appelés « formes englobantes ».

Nous allons voir comme principaux représentants de cette approche : l'arbre-R, une technique très ancienne, et d'autres techniques un peu plus récentes : l'arbre-SR et l'arbre-X.

3.1.1 Arbre-R

L'arbre-R a été proposé par Antonin Guttman [39, 58]. Il est parmi les premières propositions qui indexent des données dans les espaces multidimensionnels sous la forme d'un découpage hiérarchique équilibré en ensembles de rectangles. C'est originellement une méthode d'accès spatiale utilisée pour indexer des coordonnées géographiques.

Définition d'un arbre-R

La structure d'un arbre-R est basée sur la décomposition de l'espace autour de *rectangles englobants minimaux* (REM). Dans un espace multidimensionnel, disons $\mathbb{R}^n$ par souci de simplification, un rectangle englobant minimal est défini par un

couple de vecteurs tel que les composantes du premier vecteur sont deux à deux inférieures ou égales à celles du second vecteur. Ce couple de coordonnées définit le plus petit volume qui englobe un ensemble de points et/ou de formes géométriques donné.

Définition 10 (REM dans $\mathbb{R}^n$) *Soit $S \subseteq \mathbb{R}^n$ un ensemble de points.*

Alors le couple $(x, y) \in (\mathbb{R}^n)^2$ avec :

- $x = (x_1, x_2, \ldots, x_n)$*;*
- $y = (y_1, y_2, \ldots, y_n)$*;*

définit le rectangle englobant minimal (REM) de S si, et seulement si :

$$\begin{aligned} \forall 1 \leq i \leq n,\ x_i &= \min\{z_i, \forall z \in S\}, \\ y_i &= \max\{z_i, \forall z \in S\}. \end{aligned} \tag{3.1}$$

On notera cette opération sous la forme fonctionnelle :

$$(x, y) = \textit{REM}(S). \tag{3.2}$$

Notons qu'il faudrait parler d'hyper-rectangles (ou de rectangles de degré n) plutôt que de simples rectangles.

Cette définition s'étend à des ensembles d'hyper-rectangles.

Définition 11 (REM sur $\mathbb{R}^n$) *Soit $R \subseteq (\mathbb{R}^n)^2$ un ensemble d'hyper-rectangles.*

Alors, le couple $(x, y) \in (\mathbb{R}^n)^2$ définit le rectangle englobant minimal (REM) de R si, et seulement si :

$$\begin{aligned} \forall 1 \leq i \leq n,\ x_i &= \min\{z_i, \forall (z, t) \in R\}, \\ y_i &= \max\{t_i, \forall (z, t) \in R\}. \end{aligned} \tag{3.3}$$

On surcharge la notation fonctionnelle précédente à cette généralisation :

$$(x, y) = \textit{REM}(R). \tag{3.4}$$

L'arbre-R indexe aussi bien des points que des rectangles, les premiers pouvant être vus comme des cas dégénérés de rectangles.

Les données dans un arbre-R sont organisées en pages, qui peuvent avoir un nombre variable d'entrées :

- Les nœuds feuilles stockent les données. Le nombre de rectangles n'est pas fixé, car il dépend de la taille des données elles-mêmes et de la taille des pages qui les stockent sur le disque.

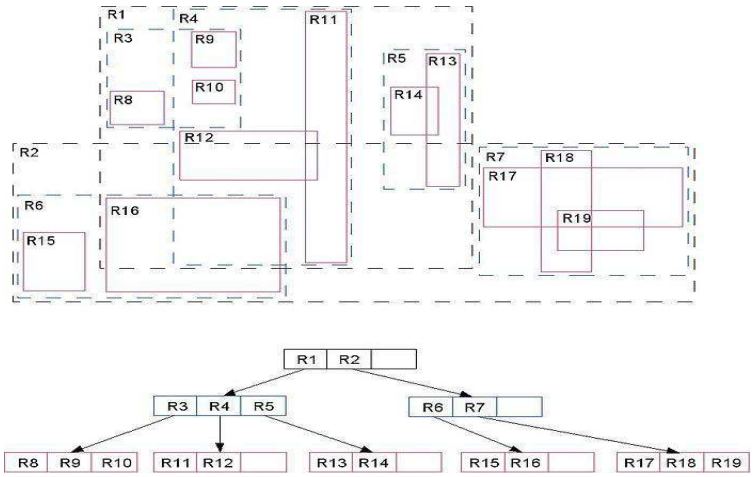


FIGURE 3.2 – Arbre-R [58]

- Les nœuds internes stockent un nombre variables d'« entrées ». Chaque entrée stocke deux éléments : un rectangle englobant minimal et un sous-arbre.

Cette configuration est illustrée en figure 3.2 [39].

Nous pouvons la décrire dans un formalisme et avec des notations que nous allons nous forcer à appliquer à tous les index arborescents à venir.

Définition 12 (Arbre-R) *Soit $\mathbb{R}^n$ un espace multidimensionnel. Soit $E \subseteq \mathbb{R}^n$ un sous-ensemble de vecteurs à indexer (extensible à des hyper-rectangles).*

Alors, on définit les nœuds $\mathcal{N}_R$ – ou seulement $\mathcal{N}$ s'il n'y a pas de risque d'ambiguïté – d'un arbre-R de la manière suivante, en distinguant les feuilles des nœuds internes :

1. *Tout d'abord, un nœud feuille L_R – ou seulement L – consiste simplement en un sous-ensemble des objets indexés :*

$$L_R \subseteq E. \tag{3.5}$$

Le contenu des feuilles partitionne E.

2. *Ensuite, un nœud interne $\mathcal{N}_R$ – ou seulement $\mathcal{N}$ – est un sous-ensemble de couples :*

$$\{((x_i, y_i), N_i)\} \tag{3.6}$$

où :

- *N_i est l'un des fils, c'est-à-dire un sous-arbre-R ;*
- *(x_i, y_i) est le rectangle englobant minimal de tous les éléments associés à ce fils.*

Autrement dit, $\mathcal{N}_R$ est un type récursif :

$$\mathcal{N}_R = \mathcal{P}(\mathcal{O}) \cup \mathcal{P}\left((\mathbb{R}^n)^2 \times \mathcal{N}_R\right). \tag{3.7}$$

Construction d'un arbre-R

La construction d'un arbre-R peut être vue, en première approximation, comme une adaptation de l'algorithme classique d'insertion dans un arbre-B [4]. Toutefois, contrairement à un arbre-B, différents ordres d'insertion des éléments sont susceptibles d'entraîner des structures finales distinctes de l'arbre. Autrement dit, la construction – incrémentale – de l'arbre-R n'est pas déterministe.

L'arbre-R est une structure dynamique. La construction incrémentale de ce type d'index se fait par un parcours récursif avec une phase descendante à partir du nœud racine suivie par une phase ascendante :

- Durant la descente, la contrainte de « minimum de couverture » [58, 14] est utilisée pour propager l'insertion d'un nouvel objet dans le sous-arbre dont le REM sera le moins modifié en volume.

  L'algorithme d'insertion peut aussi choisir un nœud en utilisant une méthode heuristique telle que le choix du rectangle qui exige le moins d'augmentation de ses côtés [58], mais d'autres méthodes ont été proposées dans des variantes de l'arbre-R.

- Lorsqu'une feuille est atteinte, l'insertion dans ce nœud est effectuée si la feuille n'est pas pleine.

- Dans le cas où la feuille est pleine, elle doit être divisée en deux nouvelles feuilles portant chacune approximativement la moitié des données d'origine et offrant chacune la meilleure compacité possible, tout en minimisant le volume total des deux nouveaux rectangles englobants.

  L'algorithme de division exhaustive a un coût quadratique – *quadratic split* – et on peut lui préférer une version moins efficace mais seulement linéaire – *linear split* – [4].

- Durant la phase ascendante, un nœud interne qui reçoit en retour deux sous-arbres suite à l'éclatement d'une feuille plus bas insère le nouveau sous-arbre dans sa description. Cela peut entraîner l'éclatement du nœud interne lui-même.

- La phase ascendante se poursuit jusqu'à la racine qui peut elle-même finir par donner naissance à une nouvelle racine, l'arbre grandissant donc par le haut, ce qui conserve la propriété d'arbre parfaitement équilibré.

Algorithme 1 Insertion dans un arbre-R

Insérer-R $\begin{pmatrix} o \in \mathcal{O}, \\ N \in \mathcal{N}, \\ c_{\max} \in \mathbb{N}^* \end{pmatrix} \in \mathcal{N}^2$

avec :

- $L' = L \cup \{o\}$, nouvelle valeur d'une feuille ;
- $(L'_1, L'_2) =$ choix_aléatoire $\Big(\text{argmin}_{\forall (L'_1, L'_2), L' = L'_1 \cup L'_2} \{V(REM(L'_1)) + V(REM(L'_2))\}\Big)$, un découpage minimisant les volumes, $V(\cdot)$, des deux REM issus d'une bipartition des éléments d'une feuille et du nouvel objet inséré ;
- $((x_I, y_I), N_I) =$ choix_aléatoire $\big(\text{argmin}_{\forall ((x_i, y_i), N_i) \in N} \{V(\{x_i, y_i, o\})\}\big)$, sélection d'un fils minimisant l'accroissement du volume de son REM ;
- $(R_1, R_2) =$ Insérer-R$(o, N_I, c_{\max})$, résultat d'une insertion récursive dans le fils sélectionné N_I ;
- $REM(N) = REM(\{x, y : ((x, y), \cdot) \in N\})$ quand N est un nœud interne (ex. : R_2) ;
- $N' = N \setminus \{((x_I, y_I), N_I)\} \cup \{(REM(\{x_I, y_I), o\}), R_1)\} \cup \{(REM(R_2), R_2)\}$, nouvelle valeur d'un nœud interne quand l'insertion dans le fils sélectionné a entraîné un éclatement de ce dernier ;
- $(N'_1, N'_2) =$ choix_aléatoire $\Big(\text{argmin}_{\forall (N'_1, N'_2), N' = N'_1 \cup N'_2} \{V(REM(N'_1)) + V(REM(N'_2))\}\Big)$, un découpage minimisant les volumes des deux REM issus des éléments d'un nœud interne et d'un nouveau fils à insérer en débordement ;

$$
\triangleq \begin{cases}
(L', \perp) & \text{si } N = L \wedge |L| < c_{\max} \\
(\{(\text{REM}(L'_1), L'_1)\}, \{(\text{REM}(L'_2), L'_2)\}) & \text{si } N = L \wedge |L| = c_{\max} \\
\begin{pmatrix} N \setminus \{((x_I, y_I), N_I)\} \\ \quad \cup \{(REM(\{x_I, y_I, o\}), R_1)\} \end{pmatrix}, \perp & \text{si } N = \{((x_i, y_i), N_i) : 1 \le i \le m\} \wedge R_2 = \perp \\
\begin{pmatrix} N \setminus \{((x_I, y_I), N_I)\} \\ \quad \cup \{(REM(\{x_I, y_I), o\}), R_1)\} \\ \quad \cup \{(REM(R_2), R_2)\} \end{pmatrix}, \perp & \text{si } N = \{((x_i, y_i), N_i) : 1 \le i \le m\} \wedge R_2 \neq \perp \wedge |N| < c_{\max} \\
(\{(\text{REM}(N'_1), N'_1)\}, \{(\text{REM}(N'_2), N'_2)\}) & \text{si } N = \{((x_i, y_i), N_i) : 1 \le i \le m\} \wedge R_2 \neq \perp \wedge |N| = c_{\max}
\end{cases}
$$

Algorithme 2 Recherche kNN dans un arbre-R

$$\text{kNN-R}\left(\begin{array}{l} N \in \mathcal{N}, \\ q \in \mathcal{O}, \\ k \in \mathbb{N}^*, \\ A \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}} = \emptyset \end{array}\right) \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}}$$

avec :

- $A = ((d_1, o_1), (d_2, o_2), \ldots, (d_{k'}, o_{k'}))$;
- $C = ((o_i > x_i) \ \wedge \ (o_i < y_i))$;
- Élaguer$_1$ $=$ (REALDIST(q, o) $>$ MINMAXDIST(q, R)) ;[l'élagage se fait en fonction de la distance $d_{k'}$ du $o_{k'}$]
- Élaguer$_2$ $=$ (MINDIST(q, R) $>$ MINMAXDIST(q, R')) ;
- Élaguer$_3$ $=$ (MINDIST(q, R) $>$ ACTUALDIST(q, o)).
- une limite supérieure spécifique pour chaque branche :
  - $A_0 = A$;
  - $C =$ true ;
  - $A_1 =$ kNN-R(N_i, q, k, A) si C sinon A_0 ;

$$\begin{cases} k\text{-tri}(A \cup \{(d(x,q), x) : x \in L\}) & \text{si } N = L \\ k\text{-fusion}\{A\} & \text{si } N = \{((x_i, y_i), I)\} \wedge \\ & \neg\text{Élaguer}_1 \wedge \\ & \neg\text{Élaguer}_2 \wedge \\ & \neg\text{Élaguer}_3 \end{cases}$$

L'algorithme 1 formalise pour partie la construction incrémentale de l'arbre-R. (Il manque la version particularisée, et triviale au demeurant, de la racine). Le paramètre $c_{\max}$ traduit approximativement la taille limitée des pages sur le disque. L'expression formelle est assez complexe, de nombreux cas devant être pris en compte aussi bien au niveau feuilles (deux cas : sans et avec éclatement) que des nœuds internes (trois cas : insertion simple, « absorption » d'un éclatement du fils, propagation d'un éclatement). Le découpage d'une feuille en débordement (respectivement d'un nœud interne) utilise une expression de minimisation pouvant amener à un code excessivement coûteux (en $O({c_{\max}}^4)$!), mais son but est seulement de fournir une définition générale. En pratique, les versions quadratiques et linéaires du découpage remplaceront cette expression.

Recherche kNN dans un arbre-R

L'algorithme 2 décrit formellement la recherche des *k* plus proches voisins dans un arbre-R.

La recherche commence à partir de la racine. La recherche se fait d'une manière

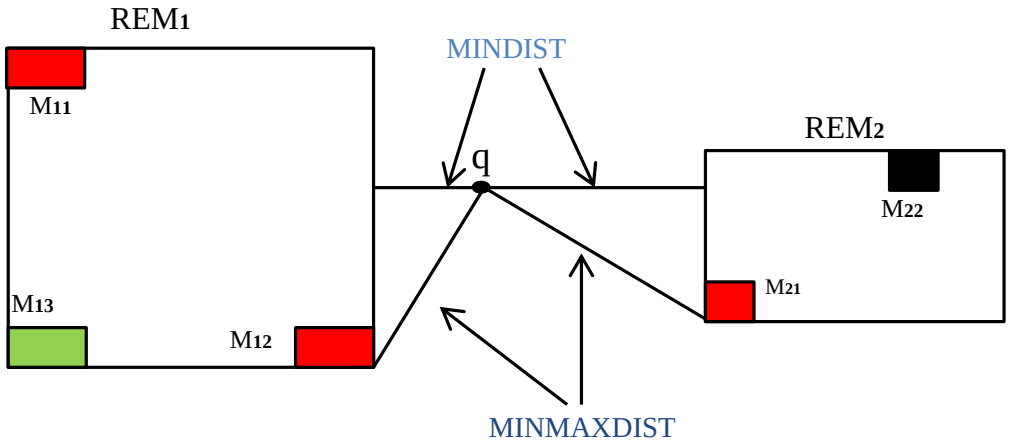


FIGURE 3.3 – MINDIST et MINMAXDIST (inspiré de [36])

récursive. Pour chaque rectangle dans un nœud, il doit être décidé s'il recouvre la requête q ou non. Si oui, le nœud fils correspondant doit être recherché aussi. Dans le cas où l'objet-requête appartient à plusieurs rectangles, on choisit le rectangle le plus petit, jusqu'à ce que tous les nœuds qui se chevauchent soient parcourus.

Pour calculer les distances, qui correspondent au rayon d'une boule-requête, dans un index où l'espace est décomposé en hyper-rectangles, on s'appuie sur deux estimations : MINDIST et MINMAXDIST (cf. figure 3.3).

Définition 13 (MINDIST) *Soit $q \in \mathbb{R}^n$ un point. Soit $R \in (\mathbb{R}^n)^2$ un REM.*

Alors on définit la distance minimale de q à R, MINDIST$(q, R) \in \mathbb{R}^+$, comme la distance minimale entre q et n'importe quel point p appartenant au rectangle R :

$$\textit{MINDIST}(q, R) = \min\{d(q, p) : \forall p \in R\} \tag{3.8}$$

où la d est distance utilisée.

Noter que si le point q est à l'intérieur du rectangle R, alors MINDIST$(q, R) = 0$. Autrement, seuls les points sur le périmètre du rectangle nécessitent d'être envisagés, comme cela est visible sur la figure 3.3.

Définition 14 (MINMAXDIST) *Soit $q \in \mathbb{R}^n$ un point. Soit $R \in (\mathbb{R}^n)^2$ un REM. Soit $n \in \mathbb{N}$ la dimension des rectangles.*

Alors on définit la distance MINMAXDIST comme une limite supérieure de q :

$$\textit{MINMAXDIST}(q, R) = \min_{0<k<n} \{\max\{d(q, o_k) : \forall o_k \in R_k\}\} \tag{3.9}$$

En d'autres termes, MINMAXDIST correspond au minimum de l'ensemble des distances maximales possibles du point q à une face de sommet (dimension) du rectangle R.

Muni de ces définitions, nous pouvons mieux résumer l'algorithme de recherche par les étapes suivantes :

1. Un parcours descendant de la racine vers les feuilles ; À chaque nouvelle visite d'un nœud interne, l'algorithme trie tous ses REM dans une liste (A), tout en suivant les commandes métriques définies dans les (MINDIST(cf. définition 13) ou MINMAXDIST(cf. définition 14)) (cf. figure 3.3) [36].

   Dans l'algorithme 2 nous utilisons les deux métriques, donc les trois types d'élagage (voir ci-dessous).

2. Selon ces deux métriques nous pouvons déjà définir deux types d'ordre de parcours de l'index : pour chaque rectangle en face du point requête nous définissons les deux distances MINDIST (optimiste) et MINMAXDIST (pessimiste).

3. Après avoir choisi un type d'ordre, nous appliquons des stratégies d'élagage pour enlever les branches inutiles :

   (a) Élagage vers le bas : Un REM R est rejeté s'il existe un autre REM R' tel que :

   $$\text{MINDIST}(q, R) > \text{MINMAXDIST}(q, R') \; ; \tag{3.10}$$

   (b) Élagage vers le bas : Un objet o est éliminé s'il existe un R tel que :

   $$\text{REALDIST}(q, o) > \text{MINMAXDIST}(q, R) \; ; \tag{3.11}$$

   (c) Élagage vers le haut : Un R est mis au rebut si un objet o se trouve de telle sorte que :

   $$\text{MINDIST}(q, R) > \text{ACTUALDIST}(q, o). \tag{3.12}$$

   Comme nous cherchons les k les plus proches voisins, nous tenons en compte du nombre d'objets dans les rectangles (feuilles) dans la procédure d'élagage.

4. À un nœud feuille, calculer la distance de chaque objet à partir du point de requête, et comparer avec la distance de l'objet le plus proche jusqu'à présent, et mettre à jour la liste si nécessaire.

5. Au retour de récursivité, appliquer les trois stratégies d'élagage pour enlever les branches inutiles.

Extensions de l'arbre-R

La création des formes emboîtées provoque souvent le problème de chevauchements d'un objet o avec deux ou même plusieurs REM [45]. Les structures d'arbre-R+ [74] et d'arbre-R* [5, 1] ont été proposées pour optimiser la recherche en minimisant le chevauchement des rectangles englobants.

La structure d'arbre-R+ évite le recouvrement par la division de chaque rectangle qui recouvre un autre en plus petits rectangles, jusqu'à ce qu'il n'y ait plus de recouvrement. Cela peut augmenter la hauteur de l'arbre, mais permet de gagner du temps de recherche tout en réduisant le nombre de sous-arbres à visiter.

La structure d'arbre-R* minimise le recouvrement entre les rectangles et minimise aussi le volume des rectangles, en appliquant, pendant l'insertion d'un nouvel objet à un nœud plein, un mécanisme de réinsertion de quelques nœuds fils du nœud plein avant de le diviser en deux avec l'espoir de trouver des meilleures positions pour les objets réinsérés. L'arbre-R* est la structure ayant le plus de succès dans la famille arbre-R.

Analyse. En résumé, l'arbre-R est une structure dynamique basée sur la création de cellules de filtrage (REM) organisées de façon hiérarchique. La procédure de recherche parcourt les branches en se basant sur la contrainte de minimum de couverture [39]. La technique a été définie d'emblée pour une implémentation paginée, donc sans limitation sur la mémoire centrale.

L'arbre-R souffre du problème du recouvrement entre les REM qui diminue l'efficacité des procédures de recherche dans les grandes dimensions [5].

Une variante a été proposée qui a pour objectif la réduction du taux de chevauchement, il s'agit de l'arbre-R*. Elle est basée sur le principe de la réinsertion des objets. Le problème majeur de cette variante est celui de la complexité de l'algorithme de réinsertion surtout dans les grandes dimensions [5].

3.1.2 Arbre-SR

En 1997, il a été proposé une nouvelle structure d'index, l'arbre-SR (pour Sphère et Rectangle) [45], afin de limiter le problème du chevauchement entre cellules. L'arbre-SR indexe des régions constituées de *l'intersection entre une hyper-sphère et un hyper-rectangle*. Pourquoi combiner ces deux formes ? Rappelons que le problème des chevauchements apparaît surtout dans les grandes dimension. Sachant que dans ce type d'espaces les hyper-sphères occupent un grand volume (pour un petit diamètre) et que les hyper-cubes ont une grande diagonale (pour un plus petit volume), alors l'intersection entre ces deux formes donne un petit volume pour un petit diamètre, donc moins de recouvrement entre les nœuds [45].

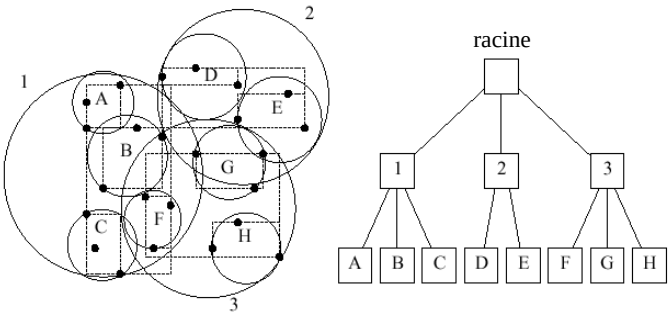


FIGURE 3.4 – Arbre-SR [45]

Construction de l'arbre-SR

Partant du même principe de construction basé sur les rectangles dans l'arbre-R, l'arbre-SR [45] propose d'utiliser des régions englobantes issues de l'intersection d'un hyper-rectangle et d'une hyper-sphère (cf. figure 3.4).

L'insertion et la mise à jour dans un arbre-SR se fait de la même façon que pour l'arbre-R, sachant que l'arbre-SR utilise un algorithme de calcul de centroïde pour choisir le centre de la sphère dans la construction de l'index. Les étapes sont les suivantes :

1. L'algorithme descend dans l'arbre à partir de la racine. À chaque nœud de l'arbre, l'entrée la plus proche est choisie.

2. Chaque entrée traversée est éventuellement modifiée. La région englobante est ajustée en recalculant les coordonnées de l'hyper-rectangle – à la manière de l'arbre-R – ainsi que la centre et le rayon de l'hyper-sphère si nécessaire – avec une nouvelle évaluation du centroïde.

Recherche dans l'arbre-SR

L'algorithme de recherche des *k* plus proches voisins dans l'arbre-SR d'un point q est similaire à celui de l'arbre-R. Il y a seulement une étape d'initialisation aléatoire de la recherche. Grossièrement, les deux étapes sont donc les suivantes :

1. Sélectionner *k* points comme *k* plus proches voisins d'une façon aléatoire.

2. Lancer l'algorithme de recherche dans l'index avec la distance au pseudo k^e objet pour diriger la recherche vers les nœuds fils qui ont les régions superposées avec l'intervalle des *k* points. Toute les étapes de recherche sont similaires à celle de l'arbre-R [45]. Nous signalons que les auteurs de ce type d'index ont définit la métrique MINDIST qui correspond a son homologue dans les arbres-R, c'est la distance entre le point requête q et la région R :

$$\text{MINDIST}(q, R) = \max(\text{MINDIST}(q, R_{REM}), \text{MINDIST}(q, R_{Sphere})) \quad (3.13)$$

Analyse. De manière similaire à l'arbre-R, la construction suit un découpage hiérarchique équilibré d'un ensemble d'intersections entre des hyper-rectangles et des hyper-sphères. Le volume des régions devient plus petit, ce qui entraîne automatiquement un moindre taux de chevauchement, qui va à son tour améliorer la performance des algorithmes de recherche [45]. Expérimentalement, l'arbre-SR accroît la performance des requêtes par plus proches voisins pour les données de haute dimension [45].

L'inconvénient majeur de cette technique est la complexité de la construction des formes englobantes, qui accroît le coût des opérations d'insertion et de mise à jour dans la mesure de calcul de l'intersection entre les deux formes surtout dans les grandes dimension [36].

3.1.3 Arbre-X

L'arbre-X (*eXtended node-tree*) [9] est un autre index multidimensionnel similaire à l'arbre-R dont le but est de limiter le problème du chevauchement des formes englobantes. Pour cela, la proposition adopte une toute autre stratégie de partitionnement des nœuds.

L'arbre-X utilise des nœuds étendus de tailles variables, dénommés super-nœuds (*extended nodes*). L'arbre-X peut être considéré comme un index hybride :

- pour partie hiérarchique – de type arborescent – ;
- pour partie linéaire – de type liste.

Il a été démontré, avec les structures d'indexation précédentes, que les faibles dimensions sont bien supportées par l'organisation hiérarchique.

L'idée donc est d'organiser l'arbre de telle sorte que les régions de données qui produisent des chevauchements extrêmes soient organisées de manière linéaire. Dans ces cas-là, un parcours séquentiel est finalement plus efficace qu'un parcours arborescent complexe.

La structure générale de l'arbre-X est présenté sur la figure 3.5.

L'arbre-X se compose de trois types de nœuds :

- les nœuds de données (feuilles) ;
- les nœuds de répertoire normaux (internes) ;
- les super-nœuds.

Les super-nœuds sont des nœuds internes de taille variable. Plus exactement, il s'agit d'un multiple de la taille d'un bloc sur disque. Les super-nœuds ne sont créés, lors des insertions, que s'il n'y a pas de possibilité intéressante pour éviter ou diminuer suffisamment les chevauchements.

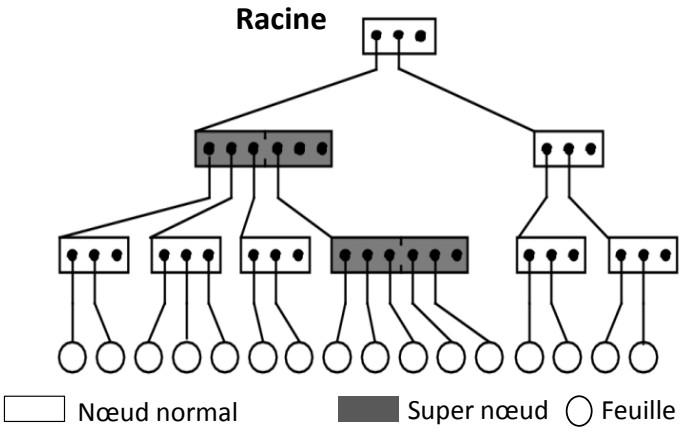


FIGURE 3.5 – Arbre-X [9]

Dans les cas extrêmes, l'arbre serait entièrement sérialisé...

La nouvelle stratégie est associée à la notion de super-nœud. L'objectif est d'empêcher les chevauchements, tout en évitant la division des nœuds par la création des super-nœuds.

L'algorithme d'insertion est similaire à celui de l'arbre-R. Il détermine récursivement le REM correspondant par rapport aux cordonnées de l'objet et des REM englobants rencontrés. Si aucune division ne se produit durant le processus d'insertion récursive, seule la taille des REM rencontrés doit être mise à jour.

Dans le cas d'une division d'un nœud feuille, un REM supplémentaire doit être ajouté au père de ce nœud. Dans ce cas, l'algorithme exécute un partage similaire à celui de l'algorithme de partage dans l'arbre-R.

Si le nombre de REM est en dessous d'un seuil donné, l'algorithme de division se termine sans fournir aucun partage. Dans ce cas, le nœud actuel sera agrandi pour devenir un super-nœud de deux fois la taille d'un bloc standard. Si le même cas se produit pour un super-nœud déjà existant, ce super-nœud est étendu d'un bloc supplémentaire [9, 45].

Si un super-nœud est créé ou étendu, il peut ne pas y avoir suffisamment d'espace contigu sur le disque pour stocker séquentiellement le super-nœud. Dans ce cas, le gestionnaire de disque doit effectuer une réorganisation locale.

L'insertion est faite si le partitionnement obtenu est équilibré, ou au moins un seuil garantit l'existence d'un minimum d'entrée dans les pages créées. Sinon le fractionnement du nœud est annulé et une nouvelle page de disque est allouée et concaténée au nœud saturé. Le rejet du fractionnement et l'ajout d'une nouvelle page forment un super-nœud.

Le chevauchement croît avec l'augmentation de dimension, donc il est évident

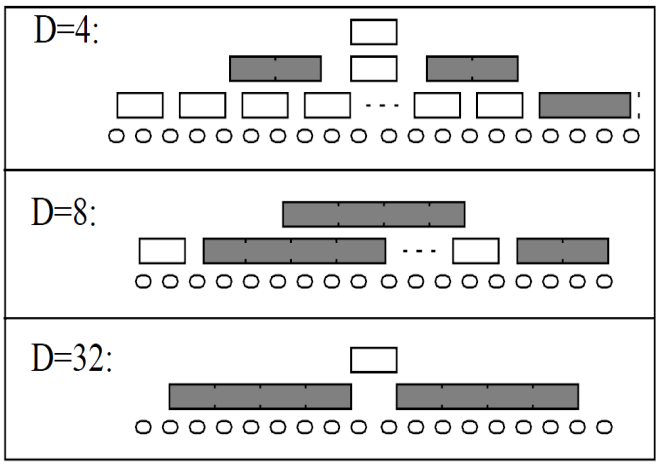


FIGURE 3.6 – Différentes formes de l'arbre-X avec différentes dimensions [9]

que la structure interne de l'arbre X change en augmentant la dimension. Dans la figure 3.6[9], les auteurs présentent trois exemples d'arbre-X contenant des données avec des dimensions différentes. Comme prévu, le nombre et de la taille des super-nœuds augmentent avec la dimension. En raison de l'augmentation du nombre et la taille des super-nœuds, la hauteur de l'arbre-X aussi va diminuer [9].

On peut noter aussi que cette technique gère bien les recouvrements, mais aussi que l'arbre-X est variable : ni la taille de l'arbre ni sa complexité ne peuvent être calculées.

3.2 Partitionnement de l'espace

Suite au principal problème associé aux techniques d'indexation basée sur le partitionnement des données, c'est-à-dire les recouvrements entre formes englobantes, il existe une autre approche où les intersections de régions sont nulles. Cette approche est basée sur le partitionnement de l'espace.

Plusieurs techniques s'appuient sur ce principe : arbre-LSD, fichier-grille, etc. Nous introduisons l'arbre-kD.

3.2.1 Arbre-*k*D

L'arbre-*k*D [7] est un type d'index basé sur le partitionnement d'un espace de dimension *k* suivant chacun de ses axes. Contrairement à une « simple » grille, tous les axes ne sont pas pris en compte simultanément. Les données sont structurées sous la forme d'un arbre binaire. À chaque niveau de l'index, on partitionne le sous-espace en cours en deux sous-sous-espaces selon une dimension différente (cf. figure 3.7). Les choix (i) de l'axe et (ii) de la valeur selon laquelle on bi-partitionne les données sur cet axe sont des paramètres importants.

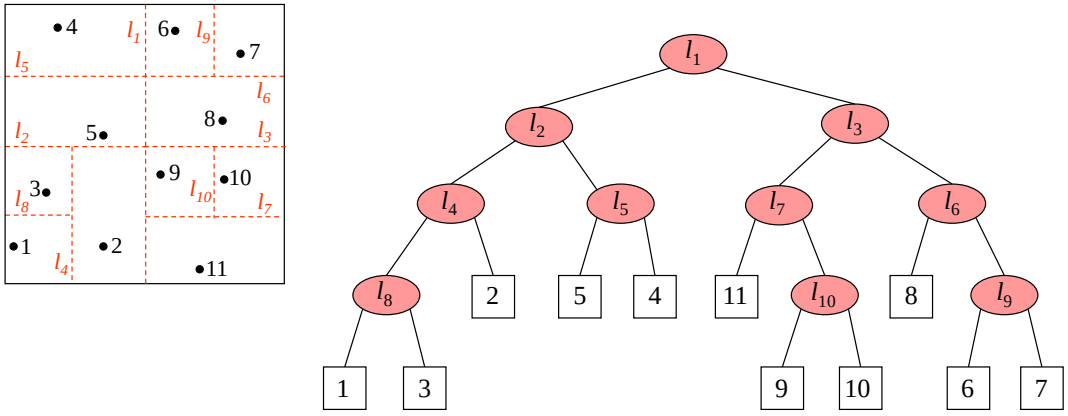


FIGURE 3.7 – Arbre-*k*D [7]

Définition de l'arbre-*k*D

En fait, l'arbre-*k*D est une généralisation à k dimensions d'un arbre binaire de recherche. (Ce dernier serait donc un arbre-1D, c'est-à-dire monodimensionnel.)

Définissons formellement l'arbre-*k*D.

Définition 15 (Arbre-*k*D) *Soit $\mathbb{R}^n$ un espace multidimensionnel. Soit $E \subseteq \mathbb{R}^n$ un sous-ensemble de vecteurs à indexer.*

*Alors, on définit les nœuds $\mathcal{N}_{kD}$ – ou seulement $\mathcal{N}$ s'il n'y a pas de risque d'ambiguïté – d'un arbre-*k*D d'une manière habituelle :*

1. *Tout d'abord, un nœud N est un triplet :*

$$(v, G, D) \in E \times \mathcal{N}_{kD} \times \mathcal{N}_{kD} \tag{3.14}$$

   où :

   - *v est la valeur d'un vecteur ;*
   - *G et D sont respectivement les sous-arbres gauche et droit.*

2. *Ensuite, un arbre peut être vide, ce que l'on dénote par $\perp$.*

Le type $\mathcal{N}_{kD}$ est donc un type récursif :

$$\mathcal{N}_{kD} = (\mathbb{R}^n \times \mathcal{N}_{kD} \times \mathcal{N}_{kD}) \cup \{\perp\}. \tag{3.15}$$

Notons que nous nous restreignons à $\mathbb{R}^n$ par souci de simplification. Il est possible de s'appuyer sur n'importe quel produit cartésien. Il suffit que les données, sur chaque dimension, soient d'une manière ou d'une autre totalement ordonnables.

Algorithme 3 Construction en lot d'un arbre-*k*D

Construire-*k*D $\begin{pmatrix} S \subseteq \mathbb{R}^n, \\ p \in \mathbb{N} = 0 \end{pmatrix} \in \mathcal{N}$
avec :

- $p < n$;
- $m =$ médiane$(S, \lambda x.x_p)$, sélection d'un vecteur dont la composante selon la dimension en cours p est de valeur médiane.

$$\triangleq \begin{cases} \bot & \text{si } S = \emptyset \\ \begin{pmatrix} m, \\ \text{Construire-}k\text{D}(\{v \in S : v_p < m_p\}, (p+1) \bmod n), \\ \text{Construire-}k\text{D}(\{v \in S : v_p \geq m_p\} \setminus \{m\}, (p+1) \bmod n) \end{pmatrix} & \text{sinon} \end{cases}$$

Algorithme 4 Insertion dans un arbre-*k*D

Insérer-*k*D $\begin{pmatrix} N \in \mathcal{N}, \\ v \in \mathbb{R}^n, \\ p \in \mathbb{N} = 0 \end{pmatrix} \in \mathcal{N}$

$$\triangleq \begin{cases} (v, \bot, \bot) & \text{si } N = \bot \\ (m, \text{Insérer}(v, G, (p+1) \bmod n), D) & \text{si } N = (m, G, D) \wedge \\ & \quad v_p < m_p \\ (m, G, \text{Insérer}(v, D, (p+1) \bmod n)) & \text{si } N = (m, G, D) \wedge \\ & \quad v_p \geq m_p \end{cases}$$

Construction d'un arbre-*k*D

Construction en lot. L'algorithme 3 formalise la construction de cet index dans une version en lot. Le paramètre p, initialisé par défaut à zéro permet de déterminer la *profondeur modulo* n du nœud courant, donc de déterminer quelle composante des vecteurs est utilisée afin de partitionner l'espace en deux demi-espaces. En d'autres termes, le choix de l'axe à prendre en considération est arbitraire et cyclique. En revanche, on choisit la médiane sur l'axe courant ce qui assure que l'on divise le sous-ensemble d'éléments en deux sous-sous-ensembles de même taille (à un élément près). Tous les vecteurs de coordonnée x_p strictement inférieure à la coordonnée du point associé au nœud, soit m_p, sont stockés dans la branche gauche du nœud. De manière symétrique, les points de coordonnée x_p supérieure à son homologue dans le vecteur médian sont stockés dans la branche droite du nœud [7]. L'arbre vide est associé à l'ensemble vide.

Ce mode de construction amène à un arbre « parfaitement » équilibré. De plus, son temps de construction est efficient, en $O(n.\log n)$, sous l'hypothèse que la fonction « médiane » a été implémentée en temps linéaire.

Construction incrémentale. En revanche, la construction incrémentale (cf. algorithme 4), peut entraîner la dégénérescence de l'arbre en une liste (par exemple,

si les éléments à insérer apparaissent ordonnés). En fait, l'arbre-*k*D est sujet dans sa généralité au même problème que leur plus simple variante, l'arbre-1D qui n'est rien d'autre que l'arbre binaire de recherche [32].

Recherche kNN dans un arbre-*k*D

Construit sur un produit cartésien quelconque, l'arbre-*k*D permet facilement d'effectuer des recherches sur une conjonction d'intervalles, c'est-à-dire des hyper-rectangles généralisés.

Pour des recherches kNN, nous nous restreignons à des vecteurs réels. Les algorithmes proposés dans la littérature utilisent la distance euclidienne. Nous généralisons cela au moins aux distances de MINKOWSKI.

L'algorithme 5 est *a priori* assez complexe. Comme pour l'arbre-R, les recherches se font à partir de boules alors que l'espace a été partitionné grâce à des hyper-plans perpendiculaires entre eux. Hormis le cas de l'arbre vide, trivial, on trouve quatre cas, duaux deux à deux. Soit la recherche ne se poursuit que dans un seul sous-arbre (*a priori* en fonction de r ou *a posteriori* en fonction de r^G ou r^D), soit il faut parcourir les deux branches en commençant alors par le demi-plan qui contient la requête.

La difficulté principale vient de la détermination d'une intersection entre la boule-requête $B(q, r_q)$ et l'hyper-plan. Dans le cas général, la condition est d'application très restrictive. Ici, grâce au système de coordonnées nous pouvons aligner la boule-requête sur la normale à l'hyper-plan au point m. L'intersection se déduit alors simplement à partir du rayon de la requête et de la distance entre m et le centre déplacé q' de la requête. En résumant, si la boule-requête actuelle $B(q, r_q)$ coupe l'hyper-plan, l'algorithme doit parcourir l'autre branche du nœud courant à la recherche de meilleurs candidats potentiels.

Nous pouvons résumer l'algorithme de recherche (cf. algorithme 5) comme suit [53, 7, 86] :

1. Dans la phase descendante, l'algorithme parcourt l'arbre de manière récursive. Il vérifie à chaque niveau l'intersection entre l'hyper-plan et la boule-requête sachant que le rayon de la boule est initialisé au début à l'infini.

2. Au niveau des feuilles, l'algorithme vérifie si un objet courant est plus proche que les meilleures solutions actuelles. Il devient alors l'un des meilleurs candidats. La liste des meilleures solutions est enregistrée au fur et à mesure dans une liste ordonnée d'au plus *k* objets [7]. Cette dernière est initialisée à une liste vide.

3. dans la phase montante, l'algorithme vérifie les nœuds internes précédents. Il pourrait y avoir des points de l'autre côté du plan qui sont plus proches

Algorithme 5 Recherche kNN dans un arbre-*k*D

$$\text{kNN-}k\text{D}\left(\begin{array}{l} N \in \mathcal{N}, \\ q \in \mathbb{R}^n, \\ k \in \mathbb{N}^*, \\ d \in \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+, \\ r_q \in \mathbb{R}^+ = +\infty, \\ A \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}} = \emptyset, \\ p \in \mathbb{N} = 0 \end{array}\right) \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}}$$

avec :

- $p < n$;
- $q' = (x_i : x_i = m_i$ si $i \neq p$ sinon $q_p, \forall 1 \leq i \leq n)$, le vecteur associé à q dont toutes les composantes autres que la p^e sont identiques à m ;
- $A^G =$ kNN-*k*D$(G, q, k, d, r_q, A, (p+1)$ mod $n)$, résultat de la recherche dans le sous-arbre gauche ;
- $A^D =$ kNN-*k*D$(D, q, k, d, r_q, A, (p+1)$ mod $n)$, *idem* à droite ;
- $r^G = \max\{d : (d, o) \in A^G\}$ si $|A^G| = k$ sinon r, rayon réduit après une recherche dans le sous-arbre gauche ;
- $r^D = \max\{d : (d, o) \in A^D\}$ si $|A^D| = k$ sinon r, *idem* à droite ;
- $A^{GD} =$ kNN-*k*D$(D, q, k, d, r^G, k$-insertion$(A^G, ((d(m, q), m)), (p+1)$ mod $n)$, résultat de la recherche dans le sous-arbre gauche poursuivie dans le sous-arbre droit ;
- A^{DG}, *idem* en sens opposé ;

$$\triangleq \left\{\begin{array}{ll} A & \text{si } N = \perp \\ A^G & \text{si } N = (m, G, D) \wedge \\ & \quad q_p < m_p \wedge \\ & \quad (r < d(q', m) \ \vee \ r^G < d(q^{(p)}, m)) \\ A^D & \text{si } N = (m, G, D) \wedge \\ & \quad q_p \geq m_p \wedge \\ & \quad (r < d(q', m) \ \vee \ r^D < d(q^{(p)}, m)) \\ A^{GD} & \text{si } N = (m, G, D) \wedge \\ & \quad q_p < m_p \wedge \\ & \quad r_q \geq d(q', m) \\ A^{DG} & \text{si } N = (m, G, D) \wedge \\ & \quad q_p \geq m_p \wedge \\ & \quad r_q \geq d(q', m) \end{array}\right.$$

que les points trouvés dans la branche qui a été déjà parcourue [7].

4. Quand l'algorithme achève cette procédure sur le nœud racine, la recherche est terminée et les meilleurs candidats actuels sont bien la réponse à la requête.

En règle générale, l'algorithme utilise, pour les comparaisons de distances, les distances élevées au carré afin d'éviter le calcul des racines carrées.

Analyse. L'arbre-*k*D partitionne l'espace à l'aide d'hyper-plans. Toutefois, dans l'algorithme de recherche et dans les cas où le point requête est proche de la frontière entre deux régions sœurs, il est nécessaire de visiter les deux régions voisines, ce qui influe négativement sur le temps de réponse surtout dans les espaces à grande dimension [7, 69].

L'arbre-*k*D n'est pas adapté à une recherche efficace de plus proches voisins dans des espaces de grande dimension. En règle générale, si la dimension est k et le nombre de points est n, alors on devrait avoir $n \gg 2k$ pour espérer des recherches efficientes. Sinon, lorsque l'arbre-*k*D est utilisé avec des données de grande dimension, la plupart des points dans l'arbre seront parcourus et l'efficacité ne sera donc pas meilleure qu'une recherche exhaustive [11, 70].

4

Techniques d'indexation arborescentes dans les espaces métriques

Dans ce chapitre, nous étendons notre étude de techniques d'indexation aux espaces métriques. Les techniques d'indexation dans les espaces métriques disposent de moins d'information que celles présentées au chapitre précédent sur les espaces multidimensionnels ; la notion d'axe a disparu. Néanmoins, nous allons constater que ces nouvelles techniques présentent beaucoup de similitudes avec les précédentes dans les principes de mise en œuvre.

Une taxinomie des techniques d'indexation dans les espaces métriques est présentée sur la figure 4.1. On y retrouve clairement les deux grandes approches déjà vues en figure 3.1 :

1. une approche qui ne partitionne pas l'espace (*non-space-partitioning*), comme les arbres M, slim, etc. ;

2. une deuxième approche qui partitionne l'espace et dans laquelle on trouve deux sous-approches :

   (a) une qui utilise des boules (*ball partitioning*), comme les arbres VP, mVP, etc. ;

   (b) une autre utilisant des hyper-plans (*hyper-plane partitioning*), tels les arbres GH, GNAT, EGNAT, etc.

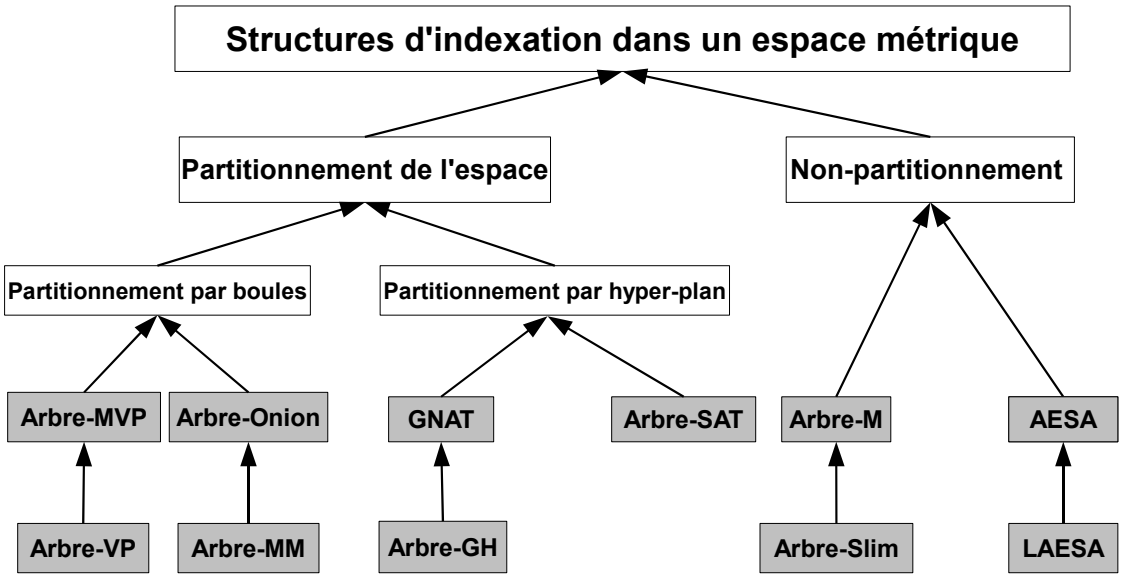


FIGURE 4.1 – Taxonomie des méthodes d'indexation dans les espaces métriques

Il existe aussi une approche basée sur le calcul de la matrice de distance : AESA (*Approximating and Eliminating Search Algorithm*) [83, 84] qui utilise une matrice $n \times n$ de distances entre les n objets. D'après les auteurs, l'AESA est applicable uniquement pour les petits ensembles de données d'au plus quelques milliers d'objets [83]. Ce problème étant lié directement à la complexité quadratique de cette technique, une amélioration a été proposée. Cette méthode atténue cet inconvénient en choisissant un nombre k fixe de pivots dont les distances à d'autres objets sont conservées. La procédure de recherche est presque la même que dans AESA [67].

On peut citer aussi qu'il existe des approches basées sur le *mapping*. Ces méthodes sont basées sur la transformation de l'espace original en un espace vectoriel multidimensionnel.

Dans la suite, nous présentons ces trois grandes approches et nous analysons le comportement de quelques-unes des techniques particulières par rapport à notre application et aux contraintes (C1) à (C4).

4.1 Techniques basées sur le non-partitionnement de l'espace

Une partie de notre taxinomie a comme principe le regroupement des données selon leurs proximités, on utilisant une fonction de distance. La technique la plus connue est probablement l'arbre-M. Dans cette branche il n'y a pas uniquement l'arbre-M mais aussi plusieurs variantes comme l'arbre-slim, AESA, etc.

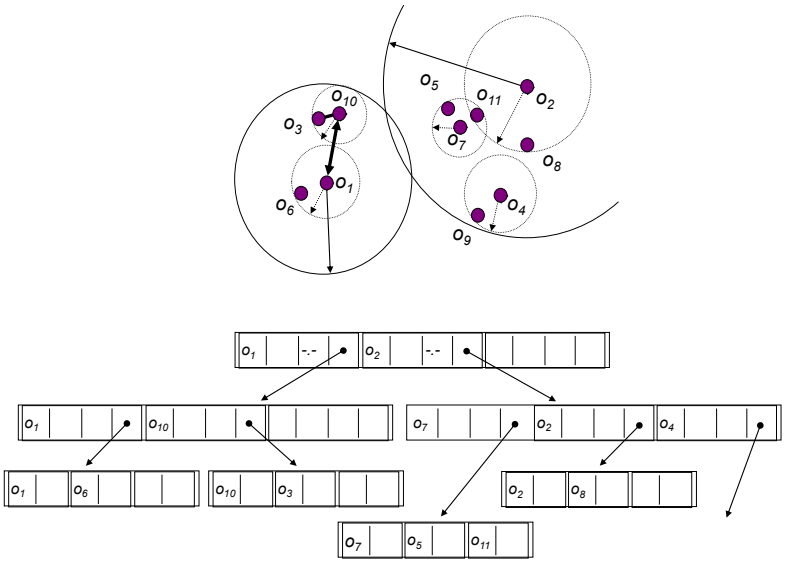


FIGURE 4.2 – Description de l'arbre-M [94]

4.1.1 Arbre-M

L'arbre-M, ou arbre métrique, est un exemple d'index à base de non-partitionnement de l'espace métrique [29, 95, 28]. L'arbre-M est considéré par ses auteurs comme l'une des premières méthodes d'indexation qui vise à réduire le nombre d'entrées-sorties *et* le nombre de calculs de distances en utilisant la propriété de l'inégalité triangulaire [26].

Définition de l'arbre-M

L'arbre-M est une structure basée sur le regroupement des données dans des boules (ou hyper-sphères) (cf. figure 4.2). Elle s'appuie donc sur la définition 3 vue dans le chapitre 2 en page 24.

Elle est basée sur deux principes :

- l'utilisation de la fonction du distance d pour regrouper des données ;
- le rejet des sous-arbres qui ne satisfont pas la contrainte de proximité lors de la recherche, issue du critère de l'inégalité triangulaire.

Définition 16 (Arbre-M) *Soit $(\mathcal{O}, d)$ un espace métrique. Soit $E \subseteq \mathcal{O}$ un sous-ensemble d'éléments à indexer.*

Alors, on définit les nœuds $\mathcal{N}_M$ – ou seulement $\mathcal{N}$ s'il n'y a pas de risque d'ambiguïté – d'un arbre-M de la manière habituelle :

1. *Tout d'abord, un nœud feuille L_M – ou seulement L – consiste simplement en un sous-ensemble* non vide *des objets indexés (sauf lorsqu'il s'agit de la racine d'un*

arbre vide) accompagnés de leur distance au centre de leur boule de référence (stockée dans le nœud père, cf. ci-dessous) :

$$L_M \subseteq E \times \mathbb{R}^+. \tag{4.1}$$

Le contenu des feuilles partitionne E.

2. *Ensuite, un nœud interne* N *est un sous-ensemble de « triplets » :*

$$\{(B(p,r), d_p, I)\} \subseteq \mathcal{P}((\mathcal{O} \times \mathbb{R}^+) \times \mathbb{R}^+ \times \mathcal{N}_M) \tag{4.2}$$

où :

- $B(p,r)$ *est un boule de centre* p *et de rayon* r *englobant de manière* minimale *tous les objets associés à ce nœud fils :*

$$\begin{array}{l} \forall o \in I, d(p,o) \leq r \wedge \\ \exists o \in I, d(p,o) = r \ ; \end{array} \tag{4.3}$$

- d_p *est la distance de* p *au centre de sa boule englobante, stockée dans le nœud père, si elle existe ;*
- I *est un « identifiant » vers le nœud fils, c'est-à-dire un sous-arbre-M.*

Un arbre-M est donc un type récursif sans arbre vide :

$$\mathcal{N}_M = \mathcal{P}(\mathcal{O} \times \mathbb{R}^+) \cup \mathcal{P}((\mathcal{O} \times \mathbb{R}^+) \times \mathbb{R}^+ \times \mathcal{N}_M). \tag{4.4}$$

On notera immédiatement la très forte similitude avec l'arbre-R (cf. définition 12 en page 12), et plus précisément les définitions de type (3.7) et (4.4). La boule (ou hyper-sphère), $\mathcal{O} \times \mathbb{R}^+$, remplace l'hyper-rectangle, $(\mathbb{R}^n)^2$. L'ajout d'une information sur la distance au centre de la boule des éléments d'une feuille ainsi que des pivots dans les nœuds internes est une optimisation qui permet – éventuellement – d'éviter quelques calculs de distance lors des recherches. Elle n'est pas présente dans l'arbre-R où l'on s'appuie directement sur les axes, bien entendu inconnus de l'arbre-M. Les hyper-rectangles d'un arbre-R sont des MBR, de la même façon que les boules de l'arbre-M sont des boules englobantes minimales. Enfin, les chevauchements entre hyper-rectangles de l'arbre-R se traduisent également par l'existence de chevauchements dans un arbre-M.

Construction d'un arbre-M

Face à toutes ces similitudes, les algorithmes associés à un arbre-M vont être très comparables à ceux de l'arbre-R (avec l'ajout de l'optimisation liée à la présence

Algorithme 6 Insertion dans un arbre-M

$$\text{Insérer-M} \left(\begin{array}{l} o \in \mathcal{O}, \\ N \in \mathcal{N}, \\ d \in \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+, \\ c_{\max} \in \mathbb{N}^*, \\ p_p \in \mathcal{O} \cup \{\perp\} = \perp \end{array} \right) \in \mathcal{N}^2$$

avec :

- $L' = L \cup \{(o, d(p_p, o))\}$ avec $d(o, \perp) = +\infty$;
- $r' = \max\{d(p, o), r\}$;
- $p_1 = \text{argmin}_{e \in E}\{d(o, e)\}$ [élément le plus proche de o] ;
- $p_2 = (\text{argmin}_{e \in E \setminus \{p_1\}}\{d(o, e)\}$ [élément le plus éloigné de o, stratégie L-LB-DIST] ;
- m-RAD $(r_1, r_2) = \{\forall p \in L', \exists (r_1, r_2) \in \mathbb{R}^2 : (d(o, p) < r_1 \vee d(o, p) < r_2) \wedge (r_1 + r_2)$ est le minimum $\}$;
- $L'_1 = \{(o, d_p) \in L' : d(p_1, o) \leq r_1\}$;
- $L'_2 = \{(o, d_p) \in L' : d(p_2, o) \leq r_2\}$ [choix aléatoire si $d(p_1, o) = r_1 \wedge d(p_2, o) = r_2$
- choix aléatoire si $d(p_1, o) \leq r_1 \vee d(p_2, o) \leq r_2$.

$$\triangleq \left\{ \begin{array}{ll} L' & \text{si } N = L \wedge |L| < c_{\max} \\ \left(\begin{array}{l} \{(B(p_1, r'_1), d(p_1, p_p), L'_1)\} \\ \{(B(p_2, r'_2), d(p_2, p_p), L'_2)\} \end{array} \right) & \text{si } N = L \wedge |L| = c_{\max} \\ \left(\begin{array}{l} \{(B(p, r'), d_p, \\ \text{Insérer-M}(o, N, d, c_{\max}, p))\} \end{array} \right) & \text{si } N = \{(B(p, r), d_p, N)\} \wedge \\ & d(p, o) \leq r \end{array} \right.$$

de distances pré-calculées dans l'arbre-M).

L'algorithme de construction de l'arbre-M est basé sur des insertions successives d'objets à partir de la racine, en choisissant à chaque niveau le nœud fils le plus approprié selon la proximité au centre de la sphère p [27]. Cela peut provoquer un dépassement de la capacité d'une feuille, qui devra donc être divisée de façon appropriée. Comme pour l'arbre-R, le dépassement de capacité peut se propager aussi jusqu'à la racine.

Plus précisément, l'algorithme 6 décrit formellement l'insertion dans les arbres-M, qui se fait progressivement et de manière récursive. L'algorithme donc est basé sur le choix des centres des boules, qui peut se faire selon la stratégie L-LB-DIST, c'est-à-dire choisir deux objets le premier est le plus proche de l'objet o, le deuxième est le plus éloigné d'un objet o. Aussi, le choix des rayons de couverture pour les deux nœuds est basé sur la technique m-RAD : c'est-à-dire choisir un couple de rayons r_1,r_2 qui ont le minimume valeur de leurs somme. À chaque

insertion d'un nouvel objet o, durant la descente l'objet est inséré dans la boule correspondante. Lorsqu'une feuille est atteinte, l'insertion dans ce nœud est effectuée si la feuille n'est pas pleine. Dans le cas où la feuille est pleine, elle doit être divisée en deux nouvelles feuilles, tout se en basant sur le choix des centres des boules et les rayons de couverture avec les techniques citées ci-dessus. Dans l'algorithme d'insertion :

- Il faut sélectionner la boule la plus proche parmi celles qui ne nécessitent pas une augmentation du rayon lors de l'insertion d'un nouvel élément.
- La modification de la structure de l'arbre se fait de manière ascendante, c'est-à-dire que l'insertion se propage depuis les feuilles vers la racine, comme dans les algorithmes dans la famille de l'arbre-B.
- Quand une feuille est pleine, elle est divisée en deux nouvelles feuilles et leurs deux boules couvrantes sont insérées dans leur nœud parent. Parmi les critères de division d'un nœud N :
  - Le choix des deux objets de routage en utilisant un critère qui a été proposé par les auteurs de l'index, il s'agit de L-LB-DIST, il permet de choisir deux objet p_1 l'objet le plus proche et p_2 l'objet le plus éloigné.

    $$\forall p \in L', \exists p_1 \in L' : d(o,p_1) < d(o,p). \forall p \in L', \exists p_2 \in L' : d(o,p_2) > d(o,p). \tag{4.5}$$

  - Si un élément est situé à l'intersection de deux ou plusieurs boules, il doit obligatoirement être attribué à un seul des nœuds fils. Ensuite ce processus remonte récursivement aux nœuds parents si le débordement persiste. En arrivant à la racine, une division de la racine est effectué en deux nœuds et une nouvelle racine est créée, l'arbre gagne donc un niveau de plus.
  - Une mise à jour des rayons de couverture r_1, r_2 est nécessaire après l'insertion de chaque nouvel objet.
  - Pour minimiser les rayons de couverture des deux nœuds résultants, on utilise m-RAD (*minimum (sum of) radii*) : Le minimum RAD est un algorithme de rayons très complexe en termes de calcul de distance.

    $$\{\forall p \in L', \exists (r_1, r_2) \in \mathbb{R}^2 : (d(o,p) < r_1 \vee d(o,p) < r_2) \wedge (r_1 + r_2) \text{espace est le minimum}\}; \tag{4.6}$$

    Cette algorithme favorise le couple d'éléments pour lesquels la somme des rayons couvrants, $r_1 + r_2$, est minimale. (Une autre stratégie, mM-RAD, minimise le maximum des deux rayons.)

Recherche kNN dans un arbre-M

L'algorithme 7 décrit formellement l'algorithme de recherche kNN, qui récupère les k plus proches voisins d'une requête q. Il est supposé qu'au moins k objets sont indexés par l'arbre-M. L'algorithme de recherche utilise une technique par « séparation-et-évaluation », tout à fait semblable à celui conçu pour l'arbre-R, qui utilise deux structures :

1. une « file à priorités » (*liste des candidats* A) : c'est une file d'attente de sous-arbres des réponses qui pourraient être trouvées ;

2. une liste k-objets kNN, qui contient le résultat à la fin de l'exécution.

L'algorithme « recherche dans un nœud » (cf. algorithme 8) est une composante de l'algorithme de recherche. Il implémente la plupart de la logique de la recherche. Sur un nœud interne, il détermine d'abord les sous-arbres candidats et les insère dans la file d'attente. Puis, si nécessaire, il appelle la fonction ajouter-(A) (non indiquée ici) pour effectuer une insertion ordonnée dans la liste A.

Des actions similaires sont effectuées dans les nœuds feuilles.

Dans les deux cas, une optimisation consiste à réduire le nombre de calculs de distance en utilisant les distances pré-calculées d'un élément vis-à-vis de son pivot dans le nœud père. Elles sont stockées dans chaque feuille ainsi que dans les nœuds internes pour les pivots eux-mêmes.

Analyse. Cette technique d'indexation présente plusieurs avantages :

- Elle est d'emblée prévue pour une version paginée et non seulement en mémoire centrale.

- Elle construit des arbres équilibrés.

- Elle s'appuie sur des mises à jour dynamiques (incrémentales).

- Elle se comporte raisonnablement bien en haute dimension [29].

L'arbre-M souffre cependant du problème des chevauchements. Ce problème diminue l'efficacité de l'index. Il a été évalué expérimentalement sur la technique de l'arbre-slim (cf. la section suivante).

4.1.2 Arbre-slim

À la suite de l'arbre-M, les chercheurs ont développé un ensemble de variantes et d'extensions, dont l'arbre-slim [81]. Comme le degré de chevauchement influe

Algorithme 7 Recherche kNN dans un arbre-M

$$\text{kNN-M}\left(\begin{array}{l} N \in \mathcal{N}, \\ q \in \mathcal{O}, \\ d \in \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+, \\ k \in \mathbb{N}^* \\ A \in (\mathcal{N})^{\mathbb{N}} = \emptyset \\ \text{kNN} \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}} = \emptyset \end{array}\right) \in \mathcal{N}$$

avec :

- A : liste des nœuds candidats ordonnée selon la distance entre le centre de boule p et la requête q ;
  - $\max\{0, d(q,p) - r\}$;
- kNN : liste est triée en ordre croissant de $d(q, o_i)$; [la liste des couple (objet, distance) ordonnés] ;
- r_q : rayon de recherche correspond au d'_k ;
- o_h : centre de la boule associé à la tête de la liste A.
- r_h : distance entre la requête q et le centre de la boule o_h associé à la tête de la liste A.

$\triangleq$

$A \leftarrow N$
kNN $\leftarrow$ choix-aléatoire$(k - objets)$; /* Choisir *k* objets aléatoirement */
$r_q \leftarrow \max\{d(q, o_i) : o_i \in \text{kNN}\}$
tant que $A \neq \emptyset$ **et** $r_q > \max\{0, d(q, o_h) - r_h\}$ **faire**
    Enlever(tête(A))
    Recherche-N(tête$(A), q, d, r_q$, kNN)
fin tant que

Algorithme 8 recherche dans un nœud d'un arbre-M

recherche-N $\left(\begin{array}{l} N \in \mathcal{N}, \\ q \in \mathcal{O}, \\ d \in \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+, \\ r_q \in \mathbb{R}^+ \text{kNN} \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}} \end{array} \right) \in (\mathcal{N}, (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}})$

si $N \in \mathcal{P}((\mathcal{O} \times \mathbb{R}^+) \times \mathbb{R}^+ \times \mathcal{N}_M)$ [est un nœud interne] **alors**
  pour chaque $(o_e) \in N$ **faire**
    /* o_e les centres des boules du niveau inférieur dans l'index */
    /* o_n le centres du boule du niveau supérieur dans l'index */
    si $|d(q, o_n) - d(o_n, o_e)| \leq r_e + r_q$ **alors**
      si $d(q, o_e) \leq r_e + r_q$ **alors**
        ajouter-A(N_{o_e}) /*le nœud inférieur devient un candidat*/
      fin si
    fin si
  fin pour
  renvoyer (kNN, A) ;
sinon
  pour chaque $o \in N_{o_n}$ **faire**
    si $|d(q, o_n) - d(o, o_n)| \leq r_q$ **alors**
      si $d(q, o) \leq r_q$ **alors**
        ajouter-KNN(o) /*l'objet o devient un candidat*/
        $r_q \leftarrow \max_{o_i \in \text{kNN}}\{d(q, o_i)\}$
      fin si
    fin si
  fin pour
fin si
renvoyer (kNN, A) ;

directement sur la performance de l'index, l'arbre-slim tente de réduire les chevauchements entre les sphères en réinsérant les objets. On notera ici la similitude avec l'arbre-R* (cf. section 3.1.1 en page 42).

L'insertion dans cet index se fait à partir de la racine. Dans chaque étape, l'algorithme se comporte comme suit :

- Sélectionner le nœud dont le pivot est la plus proche par rapport au nouvel objet.

  Dans l'arbre-M, il s'agit de sélectionner le nœud dont le rayon de couverture nécessite la plus petite extension, c'est-à-dire qui nécessite un minimum d'agrandissement.

  Dans l'arbre-slim, si plusieurs nœuds sont qualifiés, il sélectionne celui qui occupe l'espace minimum par rapport au reste des nœuds. La différence entre les deux techniques par rapport à cette propriété est qu'elle est applicable dans l'algorithme d'insertion dans l'arbre-M seulement dans le cas où l'objet peut être inséré dans plusieurs boules à la fois. Par contre, dans l'arbre-slim, cette propriété est applicable dans l'algorithme *slim-down* qui permet la réinsertion des objets, donc qui peut affecter tous les objets dans un nœud.

- La procédure d'insertion dans l'arbre-slim remplit d'abord les nœuds insuffisamment occupés.

- Dans le cas où un nœud est rempli, une étape de fractionnement apparaît.

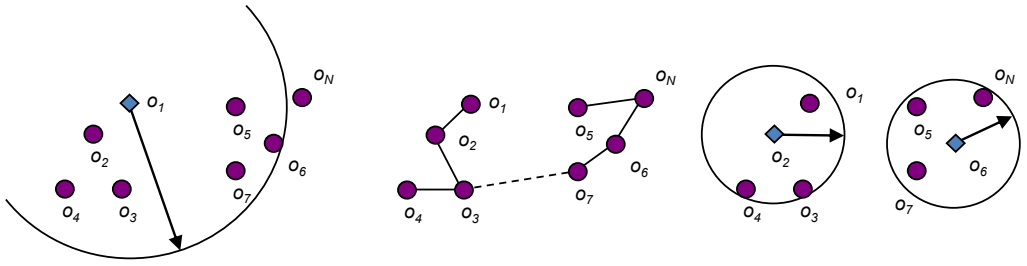
  Dans l'arbre-M, l'algorithme de partage (mM-RAD) a un coût élevé même si cette stratégie est considérée comme la meilleure à ce jour avec une complexité en $O(n^3)$ et en utilisant des calculs de distance en $O(n^2)$. Le partage dans l'arbre-slim se fait en se basant sur l'arbre couvrant minimal (*minimum spanning tree*) [68] avec une complexité en $O(n^2.\log n)$ et en utilisant des calculs de distance en $O(n^2)$.

Détaillons l'algorithme de partitionnement basé sur le calcul d'un arbre couvrant minimal.

Nous avons n objets d'où $n \times (n-1)$ arêtes portant les indications de distances entre leurs deux extrémités (cf. figure 4.3). C'est donc un graphe complet. Le partage des nœuds dans l'arbre-slim est basé sur un arbre couvrant minimal. Il se fait comme suit :

1. Construire l'arbre couvrant minimal sur le graphe complet.

2. Supprimer l'arête associée à la distance la plus longue.



(a) (b) (c)

FIGURE 4.3 – Fonctionnement de l'algorithme « *slim-down* » dans l'arbre-slim [81]

3. Les sommets des deux sous-graphes forment les nouveaux nœuds (cf. figure 4.3).

4. Choisir un pivot pour chaque nœud résultant parmi les objets dont la distance aux autres membres du groupe est la plus faible.

Cet algorithme ne garantit pas une répartition équilibrée. Une variante, choisir l'arête la plus « appropriée » parmi les plus longues, amène à une division plus équilibrée. Si aucune arête n'est trouvée (par exemple, pour un ensemble de données en forme d'étoile), accepter le partage d'origine qui est déséquilibrée [81].

Algorithme *slim-down*

L'algorithme *slim-down* peut être considéré comme une procédure de post-traitement. Mais, il est à la base de la construction d'un arbre-slim, par rapport à un arbre-M. Il a pour objectif de réduire les chevauchements et, par contrecoup, de minimiser le nombre de nœuds visités par la procédure de recherche.

L'algorithme *slim-down* est appliqué uniquement sur les nœuds internes les plus bas, c'est-à-dire situés juste au-dessus de feuilles. L'algorithme *slim-down* peut être exécuté durant la construction de l'arbre.

Pour chaque nœud N au niveau des feuilles, il faut :

1. trouver l'objet o le plus éloigné du pivot de N ;

2. rechercher un nœud frère M qui couvre également l'objet o ;

3. si un tel nœud non entièrement occupé existe, déplacer o de N à M et mettre à jour le rayon de couverture de N.

Ces étapes sont appliquées à tous les nœuds à chaque niveau. Si, après un parcours complet, au moins un objet a été déplacé, alors l'algorithme est réexécuté.

L'algorithme de recherche kNN dans l'arbre-slim est identique à celui de l'arbre-M.

Analyse

L'arbre-slim est une structure d'indexation métrique basée sur le regroupement des données dans des formes englobantes qui permettent d'affiner le filtrage des candidats lors la recherche. Elle est basée sur la technique de l'arbre-M. Comme l'arbre-M souffre du problème de chevauchement entre les formes, l'arbre-slim s'est donné pour objectif la réduction de ce taux de chevauchement en utilisant l'algorithme *slim-down*.

La nouvelle méthode de fractionnement dans l'arbre-slim est considérée plus rapide que la méthode de fractionnement qui a été considérée comme la meilleure pour l'arbre-M.

Cette structure a prouvé de bonnes performances par rapport aux formes englobantes vues auparavant pour l'arbre-M. Des essais ont démontré la réduction des coûts d'entrées-sorties d'au moins 10 % sur l'algorithme de recherche [81].

Le grand défaut de cet algorithme est la possibilité de création de nœuds contenant peu d'objets, voire de nœuds vides, ce qui réduit fortement les performances de l'index, surtout dans les espaces à grande dimension [79, 94, 77].

4.2 Techniques basées sur le partitionnement de l'espace

Pour les techniques à base de partitionnement de l'espace, deux types de partitionnement ont été développés : le premier s'appuie sur les hyper-plans (arbres GH, GNAT, EGNAT, etc.) tandis que le second utilise les boules (arbres VP, mVP, MM, oignon, etc.).

4.2.1 Arbre-VP

L'arbre-VP [91] (pour *vantage point* ou « point de vue avantageux ») est un arbre binaire où le partitionnement se fait en utilisant une boule.

Définition de l'arbre-VP

Dans l'arbre-VP (cf. figure 4.4), on choisit un objet p aléatoirement et on le considère comme le pivot. Puis on calcule la médiane des distances d_m de l'ensemble des objets au pivot et on le considère comme le rayon de la boule. Le pivot p et la médiane d_m sont utilisés pour définir une boule $B(p, d_m)$ qui va partager l'espace en deux sous-espaces disjoints.

Définition 17 (Arbre-VP) *Soit $(\mathcal{O}, d)$ un espace métrique. Soit $E \subseteq \mathcal{O}$ un sous-ensemble d'éléments à indexer.*

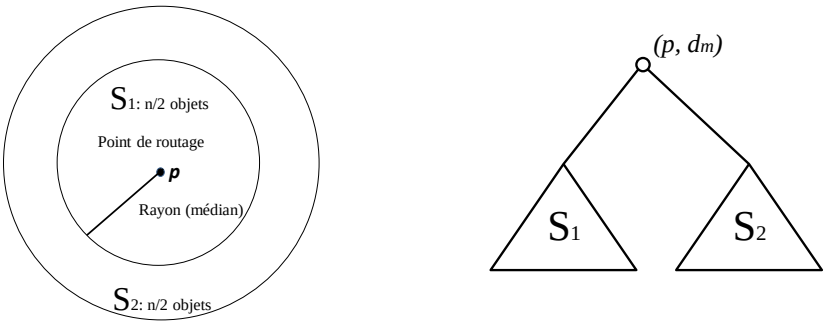


FIGURE 4.4 – Principe de construction de l'arbre-VP [91]

Alors, on définit les nœuds $\mathcal{N}_{VP}$ – ou seulement $\mathcal{N}$ s'il n'y a pas de risque d'ambigüité – d'un arbre-VP de la manière habituelle :

1. *Tout d'abord, un nœud L_{VP} – ou seulement L – consiste en un élément, centre d'une boule de rayon indiqué, et de deux fils :*

$$(p, r, I, O) \in E \times \mathbb{R}^+ \times L_{VP} \times L_{VP}. \tag{4.7}$$

   où :

   - *p est un pivot, choisi aléatoirement ;*
   - *r est le distance associée à l'objet « médian », c'est-à-dire à un élément de E situé à une distance médiane de p ;*
   - *I et O sont respectivement les fils associés aux deux parties d'espace : l'intérieur $I(\mathcal{O}, d, p, r)$ et l'extérieur $O(\mathcal{O}, d, p, r)$ de la boule $B(p, r)$ conformément à la définition 3 en page 24.*

2. *Un (sous) arbre peut être vide, ce que l'on dénote par $\perp$.*

Un arbre-VP est donc un type récursif :

$$\mathcal{N}_{VP} = \mathcal{P}(\mathcal{O} \times \mathbb{R}^+ \times L_{VP} \times L_{VP}) \cup \{\perp\}. \tag{4.8}$$

On notera la similitude entre l'arbre-VP et l'arbre-*k*D (cf. définition 15 en page 47). La notion d'axe étant absente, cette dernière est remplacée par le choix d'un élément quelconque. En revanche, la notion de partage médian est bien conservée. Dans un arbre-*k*D, on bi-partitionne de manière équitable suivant un axe. Dans l'arbre-VP, on bi-partitionne de manière équitable par rapport aux distances à l'élément pivot.

Algorithme 9 Construction en lot d'un arbre-VP

Construire-VP $(S \in \mathcal{P}(\mathcal{O})) \in \mathcal{N}$
avec :

- $p =$ choix_aléatoire(S), un élément quelconque ;
- $r =$ médiane$(\{d(e,p) : e \in S\})$, sélection de la distance médiane de p aux autres éléments de S ;

$$\triangleq \begin{cases} \bot & \text{si } S = \emptyset \\ (e, \bot, \bot, \bot) & \text{si } S = \{e\} \\ \begin{pmatrix} p, r, \\ \text{Construire-VP}(\{e \in S : d(e,p) \leq r\} \setminus \{p\}), \\ \text{Construire-VP}(\{e \in S : d(e,p) > r\}) \end{pmatrix} & \text{sinon} \end{cases}$$

Algorithme 10 Insertion dans un arbre-VP

$$\text{Insérer-VP} \begin{pmatrix} o \in \mathcal{O}, \\ N \in \mathcal{N} \end{pmatrix} \in \mathcal{N}$$

$$\triangleq \begin{cases} (o, \bot, \bot, \bot) & \text{si } N = \bot \\ (p, d(p,o), (o, 0, \bot, \bot), \bot) & \text{si } N = (p, \bot, \bot, \bot) \\ (p, r, \text{Insérer}(o, I), O) & \text{si } N = (p, r, I, O) \wedge \\ & \quad d(p,o) \leq r \\ (p, r, I, \text{Insérer}(o, O)) & \text{si } N = (p, r, I, O) \wedge \\ & \quad d(p,o) > r \end{cases}$$

Construction d'un arbre-VP

Compte tenu de toutes les similitudes entre les deux approches, les algorithmes vont également être semblables.

Construction en lot. L'algorithme (cf. algorithme 9) formalise la construction de cet index dans une version en lot. Il est une transposition très directe de l'algorithme 3 en page 48. Bien entendu, le paramètre p (profondeur) a disparu, remplacé par le choix arbitraire de p (pivot). Le calcul d'une médiane porte sur un rayon de boule plutôt qu'un point sur un intervalle de valeurs. Le bi-partitionnement se fait alors sur l'intérieur et l'extérieur de la boule $B(p,r)$ plutôt que sur les intervalles $]-\infty, m]$ et $]m, +\infty[$. Ce mode de construction amène donc également à un arbre « parfaitement » équilibré (cf. figure 4.4). Son temps de construction est également en $O(n.\log n)$ (sous l'hypothèse d'une fonction « médiane » linéaire). Il faut noter l'apparition d'un cas particulier quand le sous-ensemble à indexer se ramène à un singleton. Il faut alors construire un nœud partiellement instancié pour contenir cet élément singulier.

Construction incrémentale. La mise à jour incrémentale d'un arbre-VP est possible, même si le principe de construction en lot, implicite par l'usage de la mé-

diane, sous-entend qu'il n'a pas été prévu pour cela. Là aussi, l'algorithme 10 peut être vu comme une adaptation de l'algorithme 4 en page 48. Une petite difficulté supplémentaire est que les deux paramètres, p et r (pendants de p et m dans l'arbre-*k*D) ne peuvent pas être déterminés en une seule insertion. Le premier élément inséré dans une feuille donne une boule de rayon indéterminé. Ce n'est que lorsqu'un second élément arrive dans une telle feuille qu'il peut servir à déterminer un rayon, des plus arbitraires. Les mêmes réserves peuvent alors être faites : la construction incrémentale ne construit pas nécessairement le même arbre que la construction par lot. En particulier, l'équilibrage de l'arbre n'est plus garanti, pouvant aller jusqu'à une dégénérescence en liste.

Recherche kNN dans un arbre-VP

L'algorithme 11, bien qu'étant une transcription de l'algorithme 5 en page 50, s'est simplifié. En effet, il n'y a plus d'hyper-plan à prendre en compte, seulement des boules. Les intersections et inclusions entre boules sont simples à définir, ainsi que l'appartenance d'un élément à une boule. Dans le cas du parcours des deux sous-arbres, l'ordre envisagé est ici aussi de commencer par le sous-arbre qui contiendrait l'objet-requête q. Soulignons qu'il s'agit seulement d'une heuristique, q pouvant être situé très près de la frontière entre l'intérieur et l'extérieur.

Arbre-MVP. L'arbre-MVP (*multi-way VP*), est une généralisation de l'arbre-VP [18]. Il en est une version *n*-aire. Les nœuds sont divisés en plusieurs « sections » (anneaux concentriques) de même centre et de cardinaux égaux au lieu d'un unique rayon. En fait, on s'appuie sur des quantiles au lieu de la médiane.

Le fonctionnement de ce type d'index est donc très similaire à celui de l'arbre-VP. Le temps de construction est en $O(n.\log n)$.

Il se comporte souvent un peu mieux, mais il ne semble pas y avoir suffisamment de différence pour justifier une étude plus approfondie. Bozkaya et Ozsoyoglu [17] démontrent expérimentalement que l'idée de l'arbre améliore un peu plus l'arbre-VP (et non pas dans tous les cas), tandis qu'une plus grande amélioration est obtenue en utilisant de nombreux pivots par nœud [25].

4.2.2 Arbre-GH

L'arbre-GH (*Generalised Hyper-plane*) [82] utilise le principe du partitionnement dans les espaces métriques à l'aide d'hyper-plans (cf. définition 6). Rappelons que ce plan est défini par deux points p_1 et p_2. La figure 4.5 illustre le concept.

Algorithme 11 Recherche kNN dans un arbre-VP

$$\text{kNN-VP}\left(\begin{array}{l} N \in \mathcal{N}, \\ q \in \mathbb{R}^n, \\ k \in \mathbb{N}^*, \\ d \in \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+, \\ r_q \in \mathbb{R}^+ = +\infty, \\ A \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}} = \emptyset \end{array}\right) \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}}$$

avec :

- A^I = kNN-VP$(I, q, k, d, r_q, k\text{-insertion}(A, ((d(p,q), p)))$, résultat de la recherche dans le sous-arbre intérieur (dont p fait « partie ») ;
- A^O = kNN-VP(O, q, k, d, r_q, A), *idem* à l'extérieur ;
- $r_q^I = \max\{d : (d, o) \in A^I\}$ si $|A^I| = k$ sinon r_q, rayon réduit après une recherche dans le sous-arbre intérieur ;
- $r_q^O = \max\{d : (d, o) \in A^O\}$ si $|A^O| = k$ sinon r_q, *idem* à l'extérieur ;
- A^{IO} = kNN-VP(O, q, k, d, r_q^I, A^I), résultat de la recherche dans le sous-arbre intérieur poursuivie dans le sous-arbre extérieur ;
- A^{OI}, *idem* en sens opposé ;

$$\triangleq \begin{cases} A & \text{si } N = \perp \\ A^I & \text{si } N = (p, r, I, O) \wedge \\ & \quad (B(q, r_q) \subseteq B(p, r) \ \vee \ B(q, r_q^I) \subseteq B(p, r)) \\ A^O & \text{si } N = (p, r, I, O) \wedge \\ & \quad (B(q, r_q) \cap B(p, r) = \emptyset \ \vee \ B(q, r_q^I) \cap B(p, r) = \emptyset) \\ A^{IO} & \text{si } N = (p, r, I, O) \wedge \\ & \quad (B(q, r_q) \cap B(p, r) \neq \emptyset \ \wedge \ B(q, r_q^I) \cap B(p, r) \neq \emptyset) \wedge \\ & \quad q \in B(p, r) \\ A^{OI} & \text{si } N = (p, r, I, O) \wedge \\ & \quad (B(q, r_q) \cap B(p, r) \neq \emptyset \ \wedge \ B(q, r_q^I) \cap B(p, r) \neq \emptyset) \wedge \\ & \quad q \notin B(p, r) \end{cases}$$

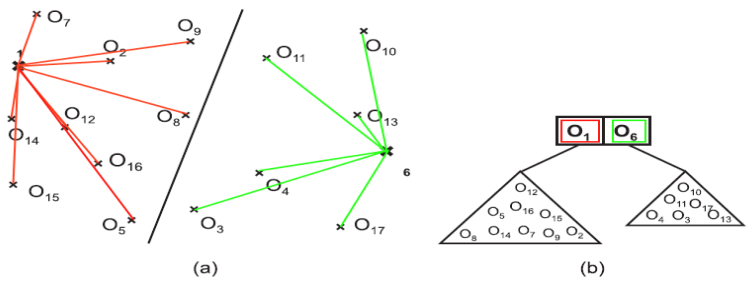


FIGURE 4.5 – Structure de l'arbre-GH [82]

Définition de l'arbre-GH

Définition 18 (Arbre-GH) *Soit $(\mathcal{O}, d)$ un espace métrique. Soit $E \subseteq \mathcal{O}$ un sous-ensemble d'éléments à indexer.*

Alors, on définit les nœuds $\mathcal{N}_{GH}$ – ou seulement $\mathcal{N}$ s'il n'y a pas de risque d'ambiguïté – d'un arbre-GH de la manière habituelle :

1. *Tout d'abord, un nœud $\mathcal{N}_{GH}$ – ou seulement $\mathcal{N}$ – consiste en deux éléments et deux fils :*

$$(p_1, p_2, G, D) \in E \times E \times \mathcal{N}_{GH} \times \mathcal{N}_{GH}. \tag{4.9}$$

   où :

   - *p_1, p_2 sont deux éléments non confondus, $d(p_1, p_2) > 0$, dénommés « pivots », ils définissent ainsi un hyper-plan ;*
   - *G et D sont les sous-arbres associés aux éléments se plaçant respectivement dans les parties « gauche », $G(\mathcal{O}, d, p_1, p_2)$, et « droite », $D(\mathcal{O}, d, p_1, p_2)$ de l'hyper-plan $H(\mathcal{O}, d, p_1, p_2)$ tels que décrits dans la définition 6 en page 25.*

2. *Un (sous) arbre peut être vide, ce que l'on dénote par $\perp$.*

Un arbre-GH est donc un type récursif :

$$\mathcal{N}_{GH} = \mathcal{P}(\mathcal{O} \times \mathcal{O} \times \mathcal{N}_{GH} \times \mathcal{N}_{GH}) \cup \{\perp\}. \tag{4.10}$$

On notera que cette définition est extrêmement proche de celle de l'arbre-VP (cf. définition 17 en page 64), la différence portant sur l'usage d'un hyper-plan au lieu d'une boule (cf. définition 3 en page 24) pour partitionner l'espace des objets.

Construction d'un arbre-GH

Nous allons donc avoir affaire à des algorithmes très proches. De manière générale, tous les algorithmes de construction d'un index arborescent sont similaires. Tant que la condition d'arrêt n'est pas atteinte, les éléments permettant de décomposer l'espace sont choisis et le traitement est répété récursivement sur les partitions ainsi définies à ce niveau de l'arbre.

Construction en lot. L'algorithme 12 est une adaptation immédiate de l'algorithme 9 en page 66 pour l'arbre-VP (lui-même transcription de celui de l'arbre-*k*D). Le couple (p_1, p_2) y remplace (p, r). En revanche, on note que rien n'est prévu pour équilibrer l'arbre à l'issue de la construction. Les deux pivots sont choisis au hasard. Bien sûr, on peut envisager de mettre en place des stratégies pour tenter d'équilibrer l'arbre, comme choisir deux éléments les plus éloignés l'un de l'autre.

Algorithme 12 Construction en lot d'un arbre-GH

Construire-GH $(S \in \mathcal{P}(\mathcal{O})) \in \mathcal{N}$
avec :

- $(p_1, p_2) =$ choix_aléatoire$(\{(o_1, o_2) \in S^2 : d(o_1, o_2) > 0\})$, sélection de deux pivots non confondus ;

$$\triangleq \begin{cases} \bot & \text{si } S = \emptyset \\ (e, \bot, \bot, \bot) & \text{si } S = \{e\} \\ \begin{pmatrix} p_1, p_2, \\ \text{Construire-GH}(\{e \in S : d(p_1, e) \leq d(p_2, e)\} \setminus \{p_1\}), \\ \text{Construire-GH}(\{e \in S : d(p_1, e) > d(p_2, e)\} \setminus \{p_2\}) \end{pmatrix} & \text{sinon} \end{cases}$$

Algorithme 13 Insertion dans un arbre-GH

$$\text{Insérer-GH} \begin{pmatrix} o \in \mathcal{O}, \\ N \in \mathcal{N} \end{pmatrix} \in \mathcal{N}$$

$$\triangleq \begin{cases} (o, \bot, \bot, \bot) & \text{si } N = \bot \\ (p_1, o, \bot, \bot) & \text{si } N = (p_1, \bot, \bot, \bot) \\ (p_1, p_2, \text{Insérer}(o, G), D) & \text{si } N = (p_1, p_2, G, D) \wedge \\ & \quad d(p_1, o) \leq d(p_2, o) \\ (p_1, p_2, G, \text{Insérer}(o, D)) & \text{si } N = (p_1, p_2, G, D) \wedge \\ & \quad d(p_1, o) > d(p_2, o) \end{cases}$$

Il faut toutefois veiller à ne pas utiliser une fonction de complexité plus que linéaire sous peine de voir l'algorithme dépasser une complexité en $O(n. \log n)$ qui est celle qu'il possède dans la version actuelle.

Notons que le fait d'utiliser un découpage de l'espace par hyper-plans est plus proche de l'arbre-*k*D que de l'arbre-VP. Cela est davantage vrai dans les espaces à grande dimension où le choix aléatoire d'un hyper-plan a davantage de chances d'être quasiment perpendiculaire aux précédents.

Construction incrémentale. L'algorithme de construction incrémentale s'appuie sur des répétitions d'insertions *via* l'algorithme 13, tout comme l'algorithme 10 en page 70. Une fois les substitutions appliquées, on note une difficulté supplémentaire. En cas d'insertion de doublons successifs, la contrainte $d(p_1, p_2) > 0$ n'est plus assurée. L'algorithme peut toutefois fonctionner en créant des arbres dégradés. Sur un tel nœud, le fils « droit » restera toujours vide. (Dans l'arbre-VP, cela se traduit par l'apparition d'une boule de rayon nul, pouvant donc contenir uniquement ces doublons ainsi que ceux à venir. Ce n'est donc pas une violation des contraintes sur l'arbre-VP.) On peut aussi prévoir un cas supplémentaire où le second pivot est remplacé dès qu'un objet distinct apparaît.

Algorithme 14 Recherche kNN dans un arbre-GH

$$\text{kNN-GH} \left(\begin{array}{l} N \in \mathcal{N}, \\ q \in \mathbb{R}^n, \\ k \in \mathbb{N}^*, \\ d \in \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+, \\ r_q \in \mathbb{R}^+ = +\infty, \\ A \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}} = \emptyset \end{array} \right) \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}}$$

avec :

- $A^G = \text{kNN-GH}(G, q, k, d, r_q, k\text{-insertion}(A, ((d(p_1, q), p))))$, résultat de la recherche dans le sous-arbre gauche (dont p_1 fait « partie », en l'absence de suppressions) ;
- $A^D = \text{kNN-GH}(D, q, k, d, r_q, A)$, *idem* pour la partie droite ;
- $r_q^G = \max\{d : (d, o) \in A^G\}$ si $|A^G| = k$ sinon r_q, rayon réduit après une recherche dans le sous-arbre gauche ;
- $r_q^D = \max\{d : (d, o) \in A^D\}$ si $|A^D| = k$ sinon r_q, *idem* pour la partie droite ;
- $A^{GD} = \text{kNN-GH}(O, q, k, d, r_q^G, A^G)$, résultat de la recherche dans le sous-arbre gauche poursuivie dans le sous-arbre droite ;
- A^{DG}, *idem* en sens opposé ;

$$\triangleq \left\{ \begin{array}{ll} A & \text{si } N = \perp \\ A^G & \text{si } N = (p, r, G, D) \wedge \\ & \quad d(q, p_1) - r_q^G < d(q, p_2) \wedge \\ & \quad d(q, p_1) < d(q, p_2) \\ A^D & \text{si } N = (p, r, G, D) \wedge \\ & \quad d(q, p_1) - r_q^D > d(q, p_2) \wedge \\ & \quad d(q, p_1) > d(q, p_2) \\ A^{GD} & \text{si } N = (p, r, G, D) \wedge \\ & \quad d(q, p_1) - r_q^G \leq d(q, p_2) + r_q^G) \wedge \\ & \quad d(q, p_1) < d(q, p_2 \\ A^{DG} & \text{si } N = (p, r, G, D) \wedge \\ & \quad d(q, p_1) + r_q^D \geq d(q, p_2) - r_q^D) \wedge \\ & \quad d(q, p_1) > d(q, p_2) \end{array} \right.$$

Recherche kNN dans un arbre-GH

L'algorithme 14, qui décrit formellement la recherche kNN dans un arbre-GH, est aussi assez complexe. Les recherches se font à partir de boules alors que l'espace a été partitionné grâce à des hyper-plans.

La recherche se fait en calculant la distance entre le point requête et les deux pivots, tout en descendant dans l'arbre.

Sans compter le cas de l'arbre vide, on peut rencontrer quatre cas lors du passage dans un nœud de l'arbre :

- le premier cas est celui où le résultat de la recherche est entièrement situé dans le sous-arbre gauche. En d'autres termes, la boule de recherche se situe entièrement dans le demi-plan gauche.

  Similairement, le deuxième cas est celui où le résultat de la recherche est entièrement présent dans le sous-arbre droit.

- Les troisième et quatrième cas sont ceux où la recherche doit *a priori* être poursuivie dans les deux sous-arbres car la boule de recherche empiète sur les deux demi-espaces.

  Ce qui distingue le troisième du quatrième cas est la position du centre. Si le centre se situe dans l'hyper-plan gauche, alors la recherche se poursuivra tout d'abord dans le fils gauche. Ce n'est que si la recherche n'a pas permis de réduire suffisamment le rayon de la boule de recherche que la poursuite aura lieu dans le fils droit. La recherche peut donc être modifiée *a posteriori* pour finalement se ramener au premier cas.

  Évidemment, le quatrième cas est celui où la recherche dans les deux fils est inversée.

Notons qu'à chaque passage dans les nœuds de l'index, c'est la valeur de rayon de requête r_q qui diminue et qui correspond en réalité à la distance au k^e objet dans la liste ordonné A.

Extension de l'arbre-GH

GNAT. L'idée essentielle de l'arbre-GH est étendue pour obtenir le GNAT (*General Near-neighbour Access Tree*). Le GNAT [20] est un arbre m-aire. C'est une généralisation de l'arbre-GH qui utilise m pivots dans chaque nœud interne au lieu de deux. Nous prenons un ensemble de pivots $P = \{p_1, \ldots, p_m\}$ et divisons les données en sous-arbres S_i tel que $S = \{S_1, \ldots, S_m\}$ en fonction de la distance la plus proche par rapport à un pivot de P. Dans chaque nœud interne une matrice de distances de taille m^2 est stockée. Plus formellement, pour tout objet $o \in S \setminus \{P\}$,

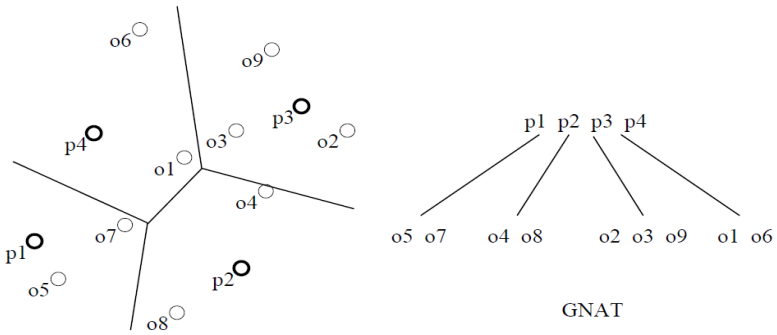


FIGURE 4.6 – Un exemple de partitionnement géométrique utilisé par GNAT : (à droite) L'accès a l'arbre-GNAT, (à gauche) l'arbre correspondant (inspiré de [20])

o est un élément de l'ensemble S_i si $d(p_i, o) \leq d(p_j, o)$ avec $j \in \{1, \ldots, m\}$, c'est-à-dire que la distance par rapport à p_i est inférieure à toutes les distances aux autres pivots. (En cas d'égalité un choix aléatoire est effectué.)

Ainsi, en appliquant cette procédure de manière récursive nous construisons un arbre d'arité m. La figure 4.6 montre un exemple simple du premier niveau d'un GNAT.

La complexité spatiale de l'index GNAT est importante car des tableaux de m^2 éléments sont stockés dans chaque nœud interne. Un GNAT est construit en $O(n.m.\log_m n)$ [2]. La complexité de la recherche n'a pas été analysée par les auteurs, mais les expériences [20] révèlent le fait que GNAT surpasse les arbres GH et VP [42]. On note aussi que cette version de GNAT est statique. Mais il en existe une extension dynamique : l'EGNAT [63].

4.2.3 Arbre-MM

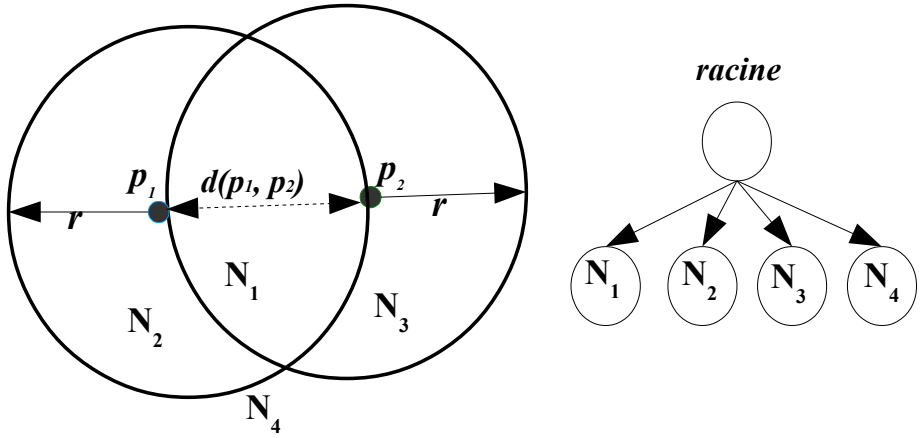
L'arbre-MM (*Memory-based Metric tree*) est un arbre qui partitionne l'espace récursivement en quatre régions non chevauchantes. Il se base sur un partitionnement en quatre régions à l'aide de deux boules à la fois en distinguant : l'intersection des boules, les différences de chaque boule par rapport à l'autre, et le reste de l'espace. Pour construire les boules, on choisit deux objets et on les considère comme deux pivots. La distance entre ces deux pivots est aussi le rayon des deux boules.

Définition 19 (Arbre-MM) *Soit $(\mathcal{O}, d)$ un espace métrique. Soit $E \subseteq \mathcal{O}$ un sous-ensemble d'éléments à indexer.*

Alors, on définit les nœuds $\mathcal{N}_{MM}$ – ou seulement $\mathcal{N}$ s'il n'y a pas de risque d'ambiguïté – d'un arbre-MM de la manière habituelle :

1. *Tout d'abord, un nœud $\mathcal{N}_{MM}$ – ou seulement $\mathcal{N}$ – consiste en deux pivots et quatre fils :*

$$(p_1, p_2, N_1, N_2, N_3, N_4) \in E \times E \times \mathcal{N}_{MM} \times \mathcal{N}_{MM} \times \mathcal{N}_{MM} \times \mathcal{N}_{MM} \quad (4.11)$$



(a)

FIGURE 4.7 – Description de l'arbre-MM

où :

- p_1, p_2 *sont les deux pivots, distincts, c'est-à-dire tels que* $d(p_1, p_2) > 0$*;*
- *le rayon des* deux *boules est la distance entre les deux pivots :* $r = d(p_1, p_2)$*;*
- N_1 *et* N_2 *et* N_3 *et* N_4 *sont respectivement les sous-arbres :*

$$
\begin{aligned}
N_1(\mathcal{O}, r, p_1, p_2) &= \{o \in \mathcal{O}, d(o, p_1) < d(p_1, p_2) \ \wedge \ d(o, p_2) < d(p_1, p_2)\} \ ; \\
N_2(\mathcal{O}, r, p_1, p_2) &= \{o \in \mathcal{O}, d(o, p_1) < d(p_1, p_2) \ \wedge \ d(o, p_2) \geq d(p_1, p_2)\} \ ; \\
N_3(\mathcal{O}, r, p_1, p_2) &= \{o \in \mathcal{O}, d(o, p_1) \geq d(p_1, p_2) \ \wedge \ (d(o, p_2) < d(p_1, p_2)\} \ ; \\
N_4(\mathcal{O}, r, p_1, p_2) &= \{o \in \mathcal{O}, d(o, p_1) \geq d(p_1, p_2) \ \wedge \ (d(o, p_2) \geq d(p_1, p_2)\}.
\end{aligned}
\tag{4.12}
$$

2. *Un (sous) arbre peut être vide, ce que l'on dénote par* $\perp$*.*

Un arbre-MM est donc un type récursif :

$$\mathcal{N}_{MM} = \mathcal{P}(\mathcal{O}^2 \times \mathcal{N}_{MM}^4) \cup \{\perp\}. \tag{4.13}$$

Dans l'arbre-VP le partage de l'espace se fait autour d'une unique boule. Dans l'arbre-MM, l'idée générale est de choisir deux objets (pivots), p_1 et p_2, et, à partir de ces deux objets, de construire deux boules ayant le même rayon r égal à la distance entre les deux pivots. De la sorte, les deux boules ont une intersection stricte. Cela permet de partitionner l'espace en *quatre* régions (cf. figure 4.7) au lieu de seulement deux.

Construction d'un arbre-MM

Construction en lot. L'algorithme de construction en lot n'est pas proposé par les auteurs. Il est toutefois facile à calquer sur ceux des arbres VP et GH. L'algorithme 15 présente cette adaptation directe.

Algorithme 15 Construction en lot d'un arbre-MM

Construire-MM $\left(\begin{array}{l} S \in \mathcal{P}(\mathcal{O}), \\ d : \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+ \end{array} \right) \in \mathcal{N}$

avec :

- $(p_1, p_2) =$ choix_aléatoire($\{(o_1, o_2) \in S^2 : d(o_1, o_2) > 0\}$), sélection de deux pivots non confondus ;
- $r = d(p_1, p_2)$;

$$\triangleq \begin{cases} \perp & \text{si } S = \emptyset \\ (e, \perp, \perp, \perp, \perp, \perp) & \text{si } S = \{e\} \\ \left(\begin{array}{l} p_1, p_2, \\ \text{Construire-MM}(\{e \in S : d(e,p_1) < r \ \wedge \ d(e,p_2) < r\}), \\ \text{Construire-MM}(\{e \in S : d(e,p_1) < r \ \wedge \ d(e,p_2) \geq r\} \setminus \{p_1\}), \\ \text{Construire-MM}(\{e \in S : d(e,p_1) \geq r \ \wedge \ d(e,p_2) < r\} \setminus \{p_2\}), \\ \text{Construire-MM}(\{e \in S : d(e,p_1) \geq r \ \wedge \ d(e,p_2) \geq r\}) \end{array} \right) & \text{sinon} \end{cases}$$

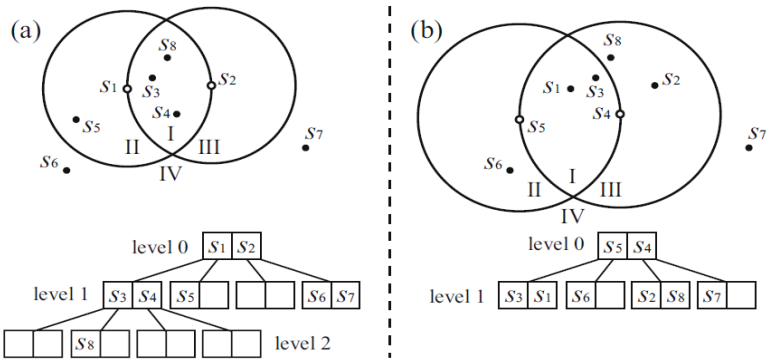


FIGURE 4.8 – Exemple de technique de semi-équilibrage [65]

Plus précisément, elle est quasi directe. En effet, l'algorithme dit de « semi-équilibrage » – présenté plus loin – est quadratique et ne peut pas être appliqué sur l'ensemble de données sans changer la classe de complexité de l'algorithme. Néanmoins, le choix aléatoire des pivots dans un grand ensemble d'éléments devrait amener à une structure quasi équilibrée.

Construction incrémentale. L'arbre-MM est construit de manière récursive, et nécessite seulement deux calculs de distances par nœud pour déterminer la région correspondant à l'objet à insérer (cf. algorithme 16 pour l'insertion d'un nouvel élément et l'algorithme 17 pour la version surchargée permettant d'insérer plusieurs objets successivement).

D'après les auteurs, on constate que la quatrième région pose un problème avec son volume. C'est la plus grande région des quatre. Cela amène à insérer la majorité des objets dans cette région au lieu de les distribuer sur l'ensemble des régions. À la fin, on peut ainsi générer des structures très déséquilibrées.

Algorithme 16 Insertion dans un arbre-MM

$$\text{Insérer-MM}\left(\begin{array}{l} o \in \mathcal{O}, \\ N \in \mathcal{N}, \\ d \in \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+ \end{array}\right) \in \mathcal{N}$$

$$\triangleq \begin{cases} (o, \bot, \bot, \bot) & \text{si } N = \bot \\ (p_1, o, \bot, \bot) & \text{si } N = (p_1, \bot, \bot, \bot) \\ \left(\begin{array}{l} p_1, p_2, r, \\ \text{Insérer-MM}(o, N_1, d), N_2, N_3, N_4 \end{array}\right) & \text{si } \begin{array}{l} N = (p_1, p_2, r, N_1, N_2, N_3, N_4) \wedge \\ d(p_1, o) < r \wedge \\ d(p_2, o) < r \end{array} \\ \left(\begin{array}{l} p_1, p_2, r, \\ N_1, \text{Insérer-MM}(o, N_2, d), N_3, N_4 \end{array}\right) & \text{si } \begin{array}{l} N = (\ldots) \wedge \\ d(p_1, o) < r \wedge \\ d(p_2, o) \geq r \end{array} \\ \left(\begin{array}{l} p_1, p_2, r, \\ N_1, N_2, \text{Insérer-MM}(o, N_3, d, c_{\max}), N_4 \end{array}\right) & \text{si } \begin{array}{l} N = (\ldots) \wedge \\ d(p_1, o) \geq r \ \wedge d(p_2, o) < r \end{array} \\ \left(\begin{array}{l} p_1, p_2, r, \\ N_1, N_2, N_3, \text{Insérer-MM}(o, N_4, d)) \end{array}\right) & \text{si } \begin{array}{l} N = (\ldots) \wedge \\ d(p_1, o) \geq r \ \wedge d(p_2, o) \geq r \wedge \\ d(p_2, o) \leq d(p_1, o) \end{array} \end{cases}$$

Algorithme 17 Insertion dans un arbre-MM, version « ensembliste »

$$\text{Insérer-MM}\left(\begin{array}{l} O \in \mathcal{P}(\mathcal{O}), \\ N \in \mathcal{N}, \\ d \in \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+ \end{array}\right) \in \mathcal{N}$$

avec $O = \{o\} \cup O'$ si $O \neq \emptyset$

$$\triangleq \begin{cases} N & \text{si } O = \emptyset \\ \text{Insérer-MM}(O', \text{Insérer-MM}(o, N, d), d) & \text{sinon} \end{cases}$$

Algorithme 18 Semi-équilibrage dans un arbre-MM

1: **fonction** Semi-équilibrage $(N \in \mathcal{N}, o_n \in \mathcal{O}) : \mathcal{N}$
2: **pour chaque** $(o, o') \in \{(o, o') \in N^2 : o \neq o'\}$ **faire**
3: **soit** $N' =$ Insérer-MM$((o, o', \bot, \bot, \bot, \bot), N \setminus \{o, o'\})$
4: **si** profondeur$(N') = 1$ **alors**
5: **renvoyer** Insérer-MM(N', o_n)
6: **fin si**
7: **fin pour**
8: **renvoyer** Insérer-MM(N, o_n)

Pour surmonter ce problème, les auteurs de l'arbre-MM ont proposé une technique de semi-équilibrage (cf. algorithme 18) (*semi-balancing*). C'est une technique qui s'applique près des feuilles de l'index à chaque nouvelle insertion [65]. Son but est d'essayer de réduire la hauteur de l'arbre, plus précisément d'obtenir un sous-arbre de hauteur un pour huit à dix éléments répartis sur quatre feuilles et leur père.

Exemple. La figure 4.8-(a) montre la structure d'un arbre-MM qui indexe les éléments de $S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$ en utilisant $\{s_1, s_2\}$ comme pivots. La distance r entre les pivots, c'est-à-dire $d(s_1, s_2)$, définit le rayon de chaque boule. Cela crée quatre régions disjointes : I, II, III et IV. Chaque élément de $S \setminus \{s_1, s_2\}$ est attribué à une région spécifique selon les conditions données en définition (19). Par exemple, l'élément s_5 est affecté à la région II, puisque $d(s_1, s_5) < r$ et $d(s_2, s_5) \geq r$.

La figure 4.8-(b) illustre le résultat de la technique de semi-équilibrage. Ici, on remplace les pivots initiaux du nœud parent, (s_1, s_2), par d'autres pivots, (s_4, s_5), en insérant les autres objets suivant les mêmes règles. On se rend compte, dans ce cas-là, que la hauteur de l'arbre est réduite et que l'arbre devient ainsi balancé. Cette structure remplace donc la précédente dans l'arbre complet.

Soulignons que cette méthode n'est appliquée que près des feuilles, plus précisément quand le nombre d'éléments indexés atteint huit et que l'index est déséquilibré comme sur la figure 4.8-(a). En effet, le coût de la reconstruction est quadratique donc prohibitif sur un trop grand nombre d'éléments et *a fortiori* sur l'ensemble de l'arbre.

Recherche kNN dans un arbre-MM

L'algorithme de recherche dans un arbre-MM [65] est basé sur un parcours simple de l'arbre, en s'appuyant toujours sur le calcul de distance par rapport aux deux pivots. La recherche des *k* plus proches voisins dans un arbre-MM tient à jour une liste de réponses (les éléments les plus proches jusqu'à ce point), ordonnées selon leur éloignement. Au départ, la valeur du rayon de requête r_q est initialisée à l'infini et la liste des résultats A est vide. Les nœuds candidats sont les nœuds pour lesquels la requête qui est présentée, la boule $B(q, r_q)$, « intersecte » la région correspondante. Cette intersection est définie par les auteurs de l'index dans le tableau 4.1. Dans le cas du parcours de plusieurs sous-arbres, l'ordre envisagé est défini *a priori* par les auteurs (cf. tableau 4.2). Il a été déduit à partir de plusieurs expérimentations réalisées par les auteurs de l'arbre-MM afin de réduire le plus rapidement possible le rayon de recherche. La liste A et le rayon de requête r_q sont progressivement mis à jour.

Algorithme 19 Recherche kNN dans un arbre-MM

kNN-MM $\left(\begin{array}{l} N \in \mathcal{N}, \\ q \in \mathcal{O}, \\ k \in \mathbb{N}^*, \\ d : \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+, \\ r_q \in \mathbb{R}^+ = +\infty, \\ A \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}} = \emptyset \end{array} \right) \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}}$

1: **si** $N = \perp$ **alors**
2: **renvoyer** A
3: **sinon si** $N = (p_1, p_2, r, N_1, N_2, N_3, N_4)$ **alors**
4: **soit** $d_1 = d(p_1, q)$
5: **soit** $d_2 = d(p_2, q)$
6: **si** $|A| < k$ **alors**
7: $r_q \leftarrow \infty$
8: **sinon**
9: $r_q \leftarrow d_k$
10: **fin si**
11: **pour** $i \in \{1, 2\}$ **faire**
12: **si** $d_i < r_q$ **alors**
13: $A \leftarrow A \cup (d(q, p_i), p_i)$
14: **soit** P la région contenant $q : B(q, r_q)$
15: kNN-MM(P, q, k, d, r_q, A)
16: **si** $p_1 \in P \wedge p_2 \in P$ **alors**
17: **pour chaque** $R_j \in \{N_1, N_2, N_3, N_4\} \wedge R_j \neq P$ **faire**
18: **si** $B(q, r_q) \cap R_j$ **alors**
19: kNN-MM(R_j, q, k, d, r_q, A)
20: **fin si**
21: **fin pour**
22: **fin si**
23: **fin si**
24: **fin pour**
25: **renvoyer** A
26: **fin si**

région	condition
I	$d(q, p_2) < r_q + r \wedge d(q, p_1) < r_q + r$
II	$d(q, p_2) + r_q \geq r \wedge d(q, p_1) < r_q + r$
III	$d(q, p_2) < r_q + r \wedge d(q, p_1) + r_q \geq r$
IV	$d(q, p_2) + r_q \geq r \wedge d(q, p_1) + r_q \geq r$

TABLE 4.1 – Condition d'intersection non vide entre les régions d'un nœud interne $N = (p_1, p_2, r, N_1, N_2, N_3, N_4)$ et une boule-requête $B(q, r_q)$

Algorithme 20 Recherche kNN dans un arbre-MM

$$\text{kNN-MM}\left(\begin{array}{l} N \in \mathcal{N}, \\ q \in \mathcal{O}, \\ k \in \mathbb{N}^*, \\ d : \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+, \\ r_q \in \mathbb{R}^+ = +\infty, \\ A \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}} = \emptyset \end{array}\right) \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}}$$

avec :

- $A = ((d_1, o_1), (d_2, o_2), \ldots, (d_{k'}, o_{k'}))$;
- $d_1 = d(p_1, q)$;
- $d_2 = d(p_2, q)$;
- $C_1 = B(q, r_q) \cap B(p_1, r) \neq \emptyset$, pour l'intersection ;
- $C_2 = B(q, r_q) \cap B(p_1, r) \neq \emptyset \wedge B(q, r_q) \cap B(p_2, r) \neq \emptyset$, la boule partielle centrée sur p_1 ;
- $C_3 = B(q, r_q) \cap B(p_1, r) \neq \emptyset \wedge B(q, r_q) \cap B(p_2, r) \neq \emptyset$, la boule partielle centrée sur p_2 ;
- $C_4 = B(q, r_q) \cap B(p_1, r_1) \neq \emptyset \vee B(q, r_q) \cap B(p_2, r_2) \neq \emptyset$, pour le reste de l'espace ;
- une limite supérieure spécifique pour chaque branche :
  - $A_0 = A$;
  - $C_0 =$ true ;
  - $r_{q_0} = \min\{r_q, d_{k'}\}$ si $k' = k$ sinon r_q ;
  - $A_i =$ kNN-MM$(N_i, q, k, r_{q_{i-1}}, A_{i-1})$ si C_i sinon A_{i-1} ;
  - $r_{q_i} = \min\{r_{q_{i-1}}, d_k\}$ si $|A_{i-1}| = k \wedge A_{i-1} = ((d_1, o_1), \ldots, (d_k, o_k))$ sinon $r_{q_{i-1}}$.

$$\triangleq \begin{cases} A, k\text{-tri}(A \cup \{(d(o, q), o) : o \in L\}) & \text{si } N = L \\ A_4 & \text{si } N = (p_1, p_2, r, N_1, N_2, N_3, N_4) \end{cases}$$

		ordre de visite	
région couvrant q	condition C	C vraie	C fausse
I	$d_1 < d_2$	I → II → (III, IV)	I → III → (II, IV)
II	$d_2 - r_q \geq r_q - d_1$	II → I → IV → III	II → IV → I → III
III	$d_1 - r_q \geq r_q - d_2$	III → I → IV → II	III → IV → I → II
IV	$d_1 \geq d_2$	IV → II → I → III	IV → III → I → II

TABLE 4.2 – Tableau de détermination de l'ordre de visite dans l'algorithme de recherche kNN [65]

Analyse. L'arbre-MM est parmi les index les plus rapides à ce jour. D'après les expérimentations faites par les auteurs de l'arbre-MM, avec un même jeu de données, le nombre de calculs de distances effectués pour une recherche du plus proche voisin est de 58 % inférieur à celui de l'arbre-VP.

Toutefois, le partitionnement de l'arbre-MM peut générer des sous-espaces de tailles très différentes, donc produire des structures fortement déséquilibrées [24]. En effet, les données expérimentales suggèrent qu'un des inconvénients de l'arbre-MM est la taille de la région IV, qui est *a priori* bien plus grande que la celle des régions I, II et III. Cela génère des structures asymétriques, car de nombreux éléments sont alors affectés à la région IV.

La politique de semi-équilibrage minimise cet inconvénient, mais il dégrade le coût de la construction de l'index en raison d'une transformation supplémentaire qui prend un temps quadratique pour déterminer les meilleurs pivots locaux [24].

En outre, les expériences préliminaires ont montré que l'arbre-MM n'est pas approprié pour des données de haute dimension. Tous ces inconvénients appellent à des améliorations sur l'arbre-MM. Par suite, certains des mêmes auteurs [24] ont proposé une nouvelle structure qui étend l'arbre-MM, appelée arbre-oignon.

4.2.4 Arbre-oignon

L'arbre-oignon est une extension de l'arbre-MM. Il partitionne récursivement l'espace en régions non chevauchantes à l'aide d'hyper-sphères. La différence est qu'au lieu d'utiliser seulement deux boules, il s'appuie sur deux ensembles de boules emboîtées.

Rappelons-nous que, dans l'arbre-MM, l'espace est partagé en quatre régions. Ce type de partage pose des problèmes au niveau de la distribution des objets dans l'espace, sachant que la quatrième région a le plus grand volume, ce qui entraîne un risque de dégénérescence de l'index. Pour résoudre ce problème, les auteurs ont proposé l'algorithme de semi-équilibrage, mais pour des raisons de complexité de cet algorithme, les mêmes auteurs proposent une nouvelle idée : l'emboîtement de boules.

Le principe de l'arbre-oignon est de créer des expansions de boules, partageant le même pivot, pour couvrir et décomposer progressivement la quatrième région de l'arbre-MM. Chacune des nouvelles intersections créées devient une nouvelle région donc un nouveau fils dans l'arbre-oignon. L'arbre-oignon est donc un arbre d'arité variable.

Définition de l'arbre-oignon

Définition 20 (Arbre-oignon) *Soit $(\mathcal{O}, d)$ un espace métrique. Soit $E \subseteq \mathcal{O}$ un sous-ensemble d'éléments à indexer.*

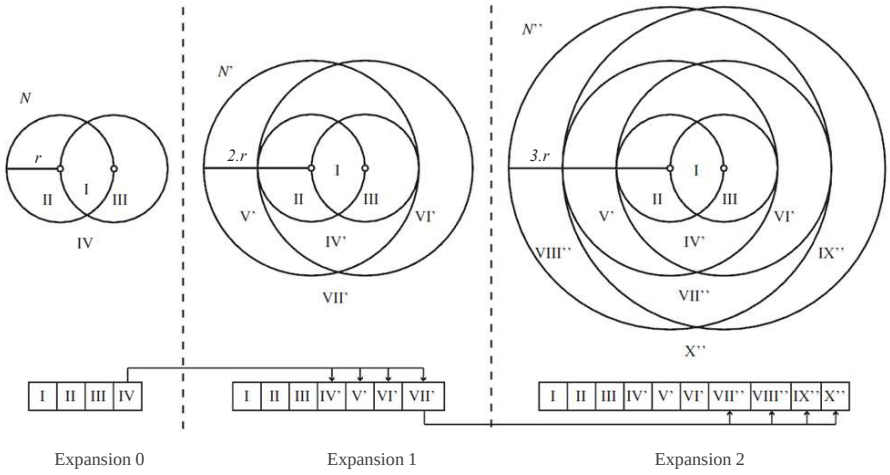


FIGURE 4.9 – Exemple de deux procédures d'expansion appliquée à un nœud N [24]

Alors, on définit les nœuds $\mathcal{N}_O$ – ou seulement $\mathcal{N}$ s'il n'y a pas de risque d'ambiguïté – d'un arbre-oignon de la manière habituelle :

1. *Tout d'abord, un nœud interne $\mathcal{N}_O$ – ou seulement $\mathcal{N}$ – consiste en un septuplet :*

$$(p_1, p_2, r, N_P, n_e, n_r, N_f) \in \mathcal{O} \times \mathcal{O} \times \mathbb{R}^+ \times \mathcal{N} \times \mathbb{N} \times \mathbb{N} \times \mathcal{N}^{\mathbb{N}} \tag{4.14}$$

où :

- p_1, p_2 *sont les deux pivots ;*
- $r = d(p_1, p_2)$ *est la distance entre les deux pivots ;*
- N_p *est le lien vers le nœud père (ce qui n'est guère mathématique comme notation) ;*
- n_e *est le nombre d'expansions, initialement à zéro ;*
- $n_r = 1 + 3 \times (n_e + 1)$ *est le nombre de régions ;*
- $N_f = (N_1, \ldots, N_{n_r})$ *sont les sous-arbres associées aux différentes régions.*

2. *Un (sous) arbre peut être vide, ce que l'on dénote par $\perp$.*

Un arbre-oignon est donc un type récursif :

$$\mathcal{N}_O = \mathcal{O}^2 \times \mathbb{R}^+ \times \mathcal{N} \times \mathbb{N}^2 \times \mathcal{N}^{\mathbb{N}} \cup \{\perp\}. \tag{4.15}$$

La figure 4.9 montre l'effet de deux procédures d'expansion appliquées à un nœud [24].

Les expansions successives font immanquablement penser à une grille. En effet, les deux pivots constituent la base d'un espace vectoriel à deux dimensions (cf. section 2.2). Par ailleurs, la création des expansions de la quatrième région

peut être quelque peu rapprochée des super-nœuds de l'arbre-X (cf. section 3.1.3), dans le double sens où la taille de ces nœuds-là est variable et vise à éviter – éventuellement – un déséquilibre réel (arbre-MM) ou structurel (arbre-X) dans les fils.

Construction de l'arbre-oignon

L'algorithme de construction de l'arbre-oignon utilise deux techniques particulières :

1. les expansions successives déjà décrites ;
2. la technique du remplacement des pivots ou *keep-small*.

Procédure d'expansion. Pour éviter le problème principal de l'arbre-M, à savoir la trop grande taille *a priori* de la région IV par rapport aux autres, la politique adoptée par l'arbre-oignon est celle d'expansions successives afin de créer de nouvelles régions qui décomposent la région IV initiale [24]. La figure 4.9 [24] montre la forme de ces expansions successives. Le but poursuivi est d'éviter une dégénérescence de l'arbre lors d'insertions successives. Notons que le déséquilibrage subi par l'arbre-M est en fait traduit par une arité croissante des nœuds de l'arbre-oignon. À notre sens, le rapport entre les deux n'a pas été discuté par les auteurs.

Il a été proposé deux méthodes d'expansion :

- Dans la méthode des expansions fixes, chaque nœud possède exactement la même structure, c'est-à-dire celle du nœud ayant subi le plus de mises à jour.
- Dans la méthode des expansions variables, l'agrandissement est différent pour chaque nœud : un nœud peut donc contenir dix régions soit deux agrandissements réalisés, alors qu'un autre peut encore n'en avoir que quatre si aucun agrandissement n'a encore eu lieu.

L'algorithme Créer-Régions (cf. algorithme 21) détermine le nombre d'expansions qui devraient être appliquées à un nœud N en fonction d'une entrée entière E. Dans le cas où $E > 0$, la politique fixe le nombre d'expansions. Dans le cas contraire, l'algorithme utilise la technique dite *keep-small*. À la fin, l'algorithme détermine le nombre de régions de N.

Précisons que l'algorithme de création des expansions utilise les deux mêmes pivots et s'appuie sur le même rayon. Les deux pivots sont donc partagés en tant que centres des nouvelles boules. Pour ces dernières, seul le rayon change, étant égal à $2 \times r$ pour le premier agrandissement, où r est le rayon des deux boules initiales, puis $3 \times r$ pour un deuxième agrandissement, etc.

Algorithme 21 Création de régions dans un arbre-oignon

Créer-Région $\begin{pmatrix} N \in \mathcal{N}, \\ E \in \mathbb{N}^* \end{pmatrix} \in \mathbb{N}$

si $E > 0$ **alors**
  $n_e \leftarrow E$
sinon si $r \geq \frac{r_p}{2}$ **alors**
  /* r le rayon du nœud courant r_p le rayon du nœud père*/
  $n_e \leftarrow 0$
sinon
  $n_e \leftarrow \frac{r_p}{r}$
fin si
$n_r \leftarrow (n_e \times 3) + 4$

Algorithme 22 Choix de régions dans un arbre-oignon

Choix-Région $\begin{pmatrix} N \in \mathcal{N}, \\ d_1 \in \mathbb{R}^+, \\ d_2 \in \mathbb{R}^+ \end{pmatrix} \in \mathbb{N}$

$Expansion \leftarrow 0$
$Region \leftarrow 0$
$R \leftarrow 0$
tant que $Expansion \leq n_e$ **et** $Region = 0$ **faire**
  /*n_e est le nombre d'expansion */
  $R \leftarrow R + r$
  /*r est la distance entre les deux pivots */
  si $d_1 < R$ **et** $d_2 < R$ **alors**
    $Region \leftarrow 1$
  sinon si $d_1 < R$ **et** $d_2 \geq R$ **alors**
    $Region \leftarrow 2$
  sinon si $d_1 \geq R$ **et** $d_2 < R$ **alors**
    $Region \leftarrow 3$
  fin si
  $Expansion \leftarrow Expansion + 1$
fin tant que
$Expansion \leftarrow Expansion - 1$
si $Region = 0$ **alors**
  $Region \leftarrow 4$
fin si
renvoyer $3 \times Expansion + Region$

Algorithme 23 Technique de remplacement dans un arbre-oignon

keep-small $\begin{pmatrix} N \in \mathcal{N}, \\ E \in \mathbb{N}^*, \\ o \in \mathcal{O}, \\ d : \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+ \end{pmatrix} \in \mathcal{O} \times \mathcal{O} \times \mathbb{R}^+$

soit $(p_1, p_2, r, \cdot, N_p, \cdot, \cdot) = N$
soit $d_1 = d(o, p_1)$
soit $d_2 = d(o, p_2)$
soit $\alpha = \frac{r_p}{2}$
soit $\lambda_r = |r - \alpha|$
soit $\lambda_1 = |d_1 - \alpha|$
soit $\lambda_2 = |d_2 - \alpha|$
si $\lambda_1 < \lambda_2$ **alors**
  si $\lambda_1 < \lambda_r$ **alors**
    renvoyer (p_1, o, d_1)
  sinon
    renvoyer (p_1, p_2, r)
  fin si
sinon si $\lambda_2 < \lambda_r$ **alors**
  renvoyer (o, p_2, d_2)
fin si

L'algorithme Choix-Région (cf. algorithme 22) est appliqué afin de distribuer les nœuds de la zone extérieure dans les nouvelles régions créées.

La stratégie adoptée par la procédure d'expansion définit que les expansions ne sont nécessaires que lorsque le rayon du nœud courant est inférieur à la moitié du rayon du nœud parent. Dans le cas contraire, elle n'est pas nécessaire. Les auteurs appellent cette approche le « *keep-small strategy* » [24].

Technique de remplacement. Soulignons deux constatations antagonistes sur l'arbre-oignon [24] :

1. Dans l'arbre-oignon, si les pivots sont trop proches, de nombreuses procédures d'expansion seront appliquées.

2. Inversement, si les pivots sont trop éloignés, aucune procédure d'extension ne sera faite et la plupart des éléments iront grossir la région I.

De même qu'un algorithme de semi-équilibrage avait été apporté à la construction de l'arbre-MM pour éviter des dégénérescences, une politique de remplacement des pivots est appliquée sur l'arbre-oignon de manière à minimiser l'apparition de ces deux cas.

L'algorithme *keep-small* (cf. algorithme 23) formalise cette politique de remplacement des pivots lors d'une insertion. Cette technique prend un temps constant et ne nécessite pas de calculs de distance supplémentaires.

Algorithme 24 Insertion dans un arbre-oignon

Insérer-oignon $\begin{pmatrix} N \in \mathcal{N}, \\ o \in \mathcal{O}, \\ d : \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+ \end{pmatrix} \in \mathcal{N}$

1: **si** $N = \perp$ **alors**
2: **renvoyer** $(o, \perp, \perp, \perp, 0, 0, \perp)$
3: **sinon si** $N = (p_1, \perp, \perp, \perp, 0, 0, \perp)$ **alors**
4: **renvoyer** $(p_1, o, d(p_1, o), \perp, 0, 4, (\perp, \perp, \perp, \perp))$
5: **sinon**
6: **soit** $d_1 = d(o, p_1)$
7: **soit** $d_2 = d(o, p_2)$
8: **si** $N = L$ **alors**
9: $r \leftarrow d(p_1, p_2)$
10: keep-small(N, E, o, d)
11: Créer-Region (N)
12: $Region \leftarrow$ Choix-Region (N, d_1, d_2)
13: Insérer-oignon $(Region, o, d)$
14: **fin si**
15: **fin si**

Pour résumer, la procédure d'insertion d'un nœud est une procédure récursive. L'algorithme Insérer-oignon (cf. algorithme 24) va tout d'abord vérifier si l'insertion nécessite ou non le remplacement d'un nœud en fonction des distances de ce nœud à chaque pivot. Si le remplacement est nécessaire, l'algorithme fait appel à la procédure Créer-Régions (cf. algorithme 21) pour déterminer le nombre d'expansions qui devraient être appliquées au nœud N. Après l'algorithme d'insertion fait appel à la procédure Choix-Région (cf. algorithme 22) afin de distribuer les nœuds de la zone extérieure dans les nouvelles régions créées. Finalement, l'insertion continue récursivement dans les nouveaux nœuds.

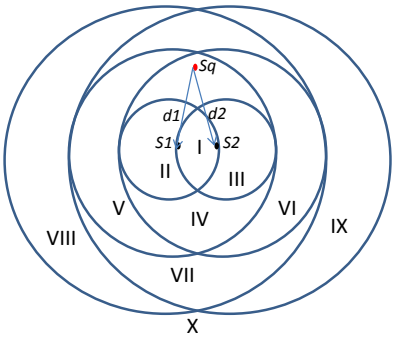
Recherche dans un arbre-oignon

L'algorithme de recherche kNN dans un arbre-MM traite seulement quatre régions par nœud. Une extension de l'algorithme kNN est proposée pour exploiter au mieux les régions supplémentaires qui ont été créées par la procédure d'expansion. À cet effet, l'arbre-oignon développe la stratégie de détermination d'un ordre de visite (cf. figure 4.10) [24].

Tout d'abord, l'algorithme visite la région d'un nœud N, où se trouve l'objet-requête s_q, c'est-à-dire q dans l'algorithme de recherche. Ensuite, il visite les autres régions de N en fonction de leur proximité à l'objet s_q. La politique détermine à la fois l'ordre de visite des expansions de N et l'ordre de visite des régions correspondantes :

1. Les expansions sont visitées dans l'ordre suivant :

Région de Sq mod 3	Condition	Ordre de Visite			
		1st	2nd	3rd	4th
1	$d_1 \leq d_2$	3E +1	3E + 2	3E + 3	3E + 4
1	$d_1 > d_2$	3E +1	3E + 3	3E + 2	3E + 4
2	$d_2 - R \leq R - d_1$	3E +2	3E +1	3E +4	3E +3
2	$d2 - R > R - d1$	3E +2	3E + 4	3E + 1	3E + 3
3	$d_1 - R \leq R - d_2$	3E +3	3E +4	3E +1	3E + 2
3	$d_1 - R > R - d_2$	3E +3	3E +1	3E +4	3E + 2
4	$d_1 \leq d_2$	3E +4	3E + 2	3E + 1	3E + 3
4	$d_1 > d_2$	3E +4	3E + 3	3E + 1	3E + 2



Ordre de visite
Expansion 1
Région IV, VI, V

Expansion 0
Régions I, III, II

Expansion 2
Régions VII, IX, VIII, X

FIGURE 4.10 – Ordre de visite des expansions et de leurs régions pour s_q [24]

(a) l'expansion e sur laquelle la requête est effectuée ;

(b) les expansions $e-1$ et $e+1$;

(c) les expansions $e-2$ et $e+2$;

(d) etc.

Visiter une expansion interne (par exemple, $e-1$) avant son homologue externe (par exemple, $e+1$) conduit à une réduction moins lente du rayon r_q. En effet, les expansions internes sont plus petites, il y a donc une grande probabilité pour qu'elles contiennent des objets plus proches. Cet ordre améliore donc l'élagage.

2. En ce qui concerne les régions des expansions, elles sont visitées dans l'ordre spécifié dans le tableau 4.10. Dans ce tableau, les régions de l'expansion e sont définies en utilisant l'opérateur modulo 3, étant donné que chaque procédure d'expansion ajoute trois régions au nœud, où $r_e = (e+1) \times r$ est le rayon de couverture utilisé dans l'expansion e.

   Par exemple, la figure 4.10 représente l'ordre de visite d'une requête s_q associée à la région IV d'un nœud possédant deux expansions. Selon la politique proposée, l'ordre de visite des expansions est de 1, 0 et 2. En outre, puisque s_q est affecté à la région IV et qu'elle est proche de s_2 (c'est-à-dire que $4 \bmod 3 = 1$ et $d_1 > d_2$), alors l'ordre de visite des régions est IV, VI et V, c'est-à-dire ce qui correspond à la deuxième ligne du tableau.

Algorithme 25 Recherche kNN dans un arbre-oignon

kNN-oignon $\left(\begin{array}{l} N \in \mathcal{N}, \\ q \in \mathcal{O}, \\ k \in \mathbb{N}^*, \\ d : \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+, \\ r_q \in \mathbb{R}^+ = +\infty, \\ A \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}} = \emptyset \end{array} \right) \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}}$

si $N = \perp$ **alors**
  renvoyer A
sinon
  soit $(p_1, p_2, r, N_p, n_e, n_r, N_f) = N$
  soit $d_1 = d(p_1, q)$
  soit $d_2 = d(p_2, q)$
  si $|A| < k$ **alors**
    $r_q \leftarrow \infty$
  sinon
    $(r_q, \cdot) \leftarrow A_k$
  fin si
  si $d_1 < r_q$ **alors**
    $A \leftarrow$ k-Insérer$(k, A, (d_1, p_1))$
  fin si
  si $d_2 < r_q$ **alors**
    $A \leftarrow$ k-Insérer$(k, A, (d_2, p_2))$
  fin si
  pour $i \in$ Ordre-visite(q, N) **faire**
    si $B(q, r_q) \cap N_i \neq \emptyset$ **alors**
      $A \leftarrow$ kNN-oignon(N_i, q, k, d, r_q, A)
    fin si
  fin pour
  renvoyer A
fin si

Rappellons que l'ordre de visite a été proposé est validé expérimentalement par les auteurs de cette technique.

Analyse

L'arbre-oignon est une méthode solide, dynamique, elle est basée sur la division de l'espace en sous-espaces disjoints. Elle est basée sur une procédure d'expansion qui contrôle le nombre de sous-espaces disjoints générés dans la structure de l'index. Elle applique également une technique de remplacement qui empêche la création de sous-espaces trop petits ou trop grands.

En outre, elle propose un algorithme de recherche kNN introduisant un ordre de visite des nœuds adapté à la méthode de partitionnements successifs par expansions.

Elle a été validée expérimentalement.

La comparaison de l'arbre-oignon avec l'arbre-slim et l'arbre-MM indique que le premier exige le plus petit temps pour construire l'index et pour traiter les requêtes.

On note aussi que l'ordre de visite optimal dépend de la distribution des données. Les auteurs supposent que les données présentent des regroupements naturels (des classes) et que les éléments sont insérés dans un ordre aléatoire. Cela reflète le scénario le plus courant en ce qui concerne les ensembles de données du monde réel.

Cette structure permet de réduire de 1 à 11 % le nombre de calculs de distances effectués pour les requêtes d'intervalle, et de 16 à 64 % pour les requêtes de k plus proches voisins [24] par rapport à l'arbre-MM. Les auteurs ont utilisé les collections « *Brazilian Cities* », « *Color Histograms* » et « *KDD Cup 2008* », que nous reprendrons pour partie aux chapitres 7 et 9.

C'est une structure dynamique, elle supporte donc les insertions grâce à la procédure de remplacement de nœuds. De plus, cette structure permet une construction mixte puisqu'elle autorise la création d'un « super nœud » lors de l'expansion de la zone IV.

5

Synthèse sur l'état de l'art

Notre propos initial concerne les espaces métriques. Nous avons toutefois étudié au chapitre 3 un certain nombre d'index arborescents portant sur le sous-ensemble des espaces multidimensionnels. La raison en est que nous apercevons des similitudes, mais aussi des dissemblances, entre ces derniers et les index dédiés aux espaces métriques. Nous allons les mettre en évidence.

5.1 Introduction

Rechercher ou répondre à une requête a toujours été l'une des plus importantes opérations de traitement automatisé. Toutefois, si la récupération de correspondances exactes est typique sur les bases de données traditionnelles, elle n'est ni suffisante ni faisable pour les types de données numériques actuels, par exemple les données multimédias.

À titre d'illustration, examinons les données textuelles comme le type le plus commun de donnée utilisée dans la recherche d'information. Selon Karen Kukich [52], le texte contient généralement environ 2 % de fautes de frappe et d'orthographe. Cela donne une motivation pour une recherche tolérante aux erreurs, ou une recherche de similarité.

Grosso modo, les objets qui sont à proximité d'un objet-requête donné constituent l'ensemble de réponses à la requête. La similitude est souvent mesurée par une fonction de distance. La notion d'espace métrique mathématique fournit une abstraction utile [43, p. 224].

Dans certaines applications, l'espace métrique se révèle être d'un type particulier appelé espace vectoriel, où les éléments sont constitués de vecteurs de n co-

ordonnées (souvent appelés vecteurs de caractéristiques). Par exemple, les images sont souvent représentées par des histogrammes de couleurs, de texture et/ou de forme [34, 40], généralement composés de 32, 64 ou 256 valeurs. Beaucoup de travaux [31, 6, 7, 39, 72, 9] ont été faits sur les espaces vectoriels en exploitant leurs propriétés géométriques.

Cependant, les techniques d'index et de recherche pour les espaces vectoriels ne fonctionnent bien que dans les espaces ayant une faible dimension. Même les méthodes d'indexation spécialement conçues pour les dimensions supérieures (arbre-X [9], arbre-LSDh [41], arbre-hybride [46]) cessent généralement d'être efficaces. Il faut accéder à presque tous les objets lorsque la dimension est supérieure à vingt.

En outre, les approches des espaces vectoriels ne peuvent pas être étendues aux espaces métriques généraux où la seule information disponible est la distance entre les objets. Enfin, dans ce cas général, la fonction de distance est souvent très coûteuse à évaluer (par exemple, la distance d'édition). Par conséquent, l'un des objectifs est de réduire le nombre de calculs de distances. Au contraire, les opérations effectuées dans les espaces vectoriels ont tendance à être simples, d'où l'objectif principal qui est de diminuer les entrées-sorties [35, 36, 33, 66, 62].

Malheureusement, certaines applications ont des difficultés à adapter leurs données aux espaces vectoriels. Pour cette raison, plusieurs auteurs recourent à des espaces métriques généraux.

En général, il y a deux approches d'indexation : une qui est basée sur les espaces multidimensionnels et une autre qui est basée sur les espaces métriques.

Le but de ce chapitre est de clarifier les apports des techniques présentées dans les deux précédents chapitres de la littérature et d'effectuer une comparaison structurelle entre elles afin de lisser la variété des approches et d'en extraire les similitudes et idées phares.

5.2 Études comparatives de méthodes d'accès multidimensionnelles

La plupart des algorithmes présentés dans le chapitre 3 ont été initialement proposés pour les structures d'indexation spatiale. Il est assez fréquent que les applications de recherche de similarité utilisent des vecteurs pour décrire les données, où les vecteurs peuvent être obtenus en utilisant l'extraction de caractéristiques (spécifiques au domaine). Ces vecteurs de caractéristiques sont indexés à l'aide de structures d'indexation multidimensionnelles et, comme elles appliquent une forme d'indexation spatiale, un arbre de recherche peut être défini pour l'exécution des requêtes de similarité (ou de proximité) sur l'ensemble des vecteurs.

Proposition	Avantages	Inconvénients
Arbre-R	Structure dynamique Création des cellules de filtrage REM MBR permet d'affiner la recherche Découpage hiérarchique équilibré Contrainte de minimum de couverture	Chevauchement des REMs Dégradation au passage à l'échelle
Arbre-R*	Variante plus efficace que l'arbre-R Réduction du taux de chevauchement Exploitation efficace de l'espace	Complexité de l'algorithme de réinsertion et de l'éclatement des nœuds Dégradation au passage à l'échelle
Arbre-SR	Construction simple Affinement : (intersection $S \wedge R$) Réduction du taux de chevauchement	Complexité des formes Algorithme de recherche coûteux Dégradation au passage à l'échelle
Arbre-X	Gestion du recouvrement (*overlap-free*) Évite la dégénérescence de l'index Réduction du taux de chevauchement	Complexité du seuil max Dégradation au passage à l'échelle
Arbre-*k*D	Découpage hiérarchique équilibré Implémentation simple	Coûteux et arbitraires Faible utilisation de l'espace alloué Dégradation au passage à l'échelle

TABLE 5.1 – Tableau comparatif des approches multidimensionnelles

Avantages	Arbre-R	Arbre-SR	Arbre-R*	Arbre-X	Arbre-*k*D
Dimension	≈ 5	≈ 10	≈ 8	≈ 20	≈ 15
Volume	★	★	★★	★★★	★★
Requête favorisée	Intervalle	Intervalle	Intervalle	Intervalle	kNN
Région	Rectangle	Rectangle/ sphère	Rectangle	Rectangle	Rectangle
Recouvrement	√	√	√	√	
Réinsertion			√	√	
Implémentation disque	√		√	√	

TABLE 5.2 – Tableau comparatif des avantages des propositions étudiées (★ signifie une unité de taille de données, √ signifie l'existance de propriétés)

Dans cette section, nous comparons leurs performances pour faire quelques observations. Nous avons donné au lecteur un aperçu général sur différentes méthodes d'accès multidimensionnelles développées au cours des deux dernières décennies. De la variété des structures de données et de leurs performances expérimentales, nous pouvons en tirer une idée assez précise sur leurs avantages et leurs inconvénients.

La performance d'une structure de données particulière dépend de nombreux facteurs comme le matériel utilisé, les paramètres du système d'exploitation, la taille des tampons, la taille des pages et les ensembles de données. En outre, la performance est généralement mesurée en termes de nombre d'accès au disque, de temps de recherche, etc.

Certains chercheurs notent qu'aucune méthode d'accès n'a fait ses preuves pour être bien supérieure à toutes les autres [13]. Si un résultat expérimental déclare une structure comme vainqueur définitif, un autre résultat expérimental peut prouver la même comme inférieure. Ce qui rend ces comparaisons si difficiles est le nombre de critères différents utilisés pour définir l'optimalité.

À l'issue de ce travail bibliographique, un résumé des éléments intéressants et des éléments manquants de chaque proposition est présenté. Les éléments sont présentés de manière synthétique dans les tableaux comparatifs 5.1 et 5.2.

On y a présenté, d'une part, une synthèse de l'ensemble de propriétés des techniques d'indexation basées sur le non-partitionnement de l'espace multidimensionnel et, d'autre part, l'utilisation des formes géométriques englobantes qui permettent d'affiner le filtrage des régions dans la phase de recherche.

La fameuse technique de l'arbre-R vient débuter l'application du principe d'imbrication de formes englobantes avec la création d'hyper-rectangles hiérarchiquement emboîtés. Malheureusement cette méthode souffre du problème de la malédiction de la dimension : inefficacité dans les grandes dimensions.

Dans le même contexte, vient alors l'arbre-R* qui introduit le principe de la réinsertion des objets. Cela a pour but la minimisation du taux de recouvrement entre les formes. Donc, l'arbre-R* propose la réinsertion dans le même niveau de l'arbre d'une page (nœud) saturée avant de la fractionner. Cette réinsertion permet dans la plupart des cas d'éviter le fractionnement. Elle assure donc une réorganisation continuelle de l'arbre.

Ensuite, la technique de l'arbre-X est apparue, avec l'idée de la création des super-nœuds, c'est-à-dire de le refus de créer une hiérarchie avec trop de recouvrements, c'est-à-dire en adoptant une stratégie locale de stockage *in extenso*. Elle gère bien mieux les recouvrements. Mais, malheureusement, si la dimension augmente, cette technique perd de son intérêt.

Une autre approche, l'arbre-SR, utilise une forme d'intersection entre rectangles et sphères. Le problème que cette technique pose réside dans la complexité des formes englobantes. Cela accroît le coût des opérations d'insertion et les recherches.

Nous avons aussi présenté une synthèse des propriétés d'une autre approche des techniques d'indexation, basées sur le partitionnement de l'espace multidimensionnel, comme l'arbre-*k*D. Le principe du partitionnement de l'espace élimine *de facto* le problème des recouvrements entre les formes. Toutefois, dans ce type d'approches, un problème se pose dans les cas où un point requête est proche de la frontière entre deux régions ; il est nécessaire de visiter toutes les régions voisines. D'autre part, la procédure de fractionnement d'une page saturée ne dépend pas de la distribution des données dans l'espace.

5.3 Étude comparative de méthodes d'accès métriques

De nombreuses méthodes sont discutées dans le chapitre 4 de notre état de l'art. Sur la base de deux techniques d'indexation, partitionnement et non-partitionnement, nous pouvons introduire une courte taxinomie de certaines techniques

d'indexation dans les espaces métriques [23]. Il y a deux cas principaux :

1. La première classe n'applique pas le de partitionnement de l'espace. Là, nous trouvons essentiellement la famille de l'arbre-M. L'arbre-M [81] génère un index équilibré, permet la mise à jour incrémentale. Malheureusement, il souffre du problème des chevauchement, qui augmente le nombre de calculs de distance pour répondre à une requête, car l'augmentation rapide de la superposition entre régions associées aux nœuds d'un même niveau diminue rapidement l'efficacité de l'index.

   Il existe une version optimisée de celui-ci : l'arbre-slim [81]. Il réorganise principalement l'arbre-M afin de réduire les chevauchements [30]. L'algorithme utilisé a montré de bonnes performances sur l'algorithme de recherche et a réduit son temps de traitement. Son défaut est la nécessité de réinsérer les objets, et le coût de cet algorithme dit *slim-down*.

2. La deuxième catégorie est basée sur le partitionnement de l'espace, et est plus riche. Deux sous-approches sont incluses : la première utilise le partitionnement par boule, comme l'arbre-VP [91], l'arbre-MVP [18], etc. ; l'autre approche utilise le partitionnement par hyper-plan tels que l'arbre-GH [82], GNAT [20], etc.

   L'arbre-VP est un type d'indice basé sur un partitionnement au traver d'une boule. Son processus de construction est basé sur la recherche de l'élément médian d'un ensemble d'objets.

   L'arbre-MVP est une généralisation de l'arbre-VP. Les nœuds dans l'arbre-MVP sont divisés en quantiles. En fait, les opérations d'insertion et de recherche sur ce type d'indice sont très similaires à l'arbre-VP. Souvent, il se comporte mieux, mais il n'y a pas suffisamment de différences pour mériter un étude plus approfondie.

   L'arbre-GH est un type d'index basé sur le partitionnement par hyper-plans. Il a prouvé son efficacité pour un nombre limité de dimensions, mais reste inefficace en grandes dimensions. Le principe de cette technique est le partitionnement récursif de l'espace en deux régions. Nous choisissons à chaque fois deux pivots, et chacun est associé aux objets les plus proches. Le problème de cette technique est que les formes géométriques des régions posent beaucoup de problèmes dans l'algorithme de recherche.

   Ces dernières années, une nouvelle technique est apparue dans cette seconde classe : l'arbre-MM [65], qui utilise aussi le principe de partitionnement par boules, mais elle est basée sur l'exploitation des régions obtenue à partir de *l'intersection* entre les boules.

Proposition	Avantages	Inconvénients
Arbre-M	Structure dynamique Réduction des calculs de distances	Non adapté aux données fortement groupées Problème des chevauchements Dégradation au passage à l'échelle
Arbre-slim	Performante par rapport arbre-M Réduction du taux de chevauchement	La complexité globale de calcul Dégradation au passage à l'échelle
Arbre-GNAT	Affiner la recherche Pas de chevauchement	Forme compliquée à manipuler Structure statique
Arbre-GH	Partitionnement simple Structure dynamique Réduction du taux de chevauchement	Forme compliquée à manipuler Dégradation au passage à l'échelle
Arbre-mVP	Peu affecté à grande échelle	Structure statique
Arbre-VP	Implémentation simple	Dégradation au passage à l'échelle
Arbre-MM	Meilleur partitionnement de l'espace	Dégradation au passage à l'échelle Dégénérative de l'index (quatrième région)
Arbre-oignon	Meilleur partitionnement de l'espace	« Réinsertion » des objets (semi-équilibrage) Dégradation au passage à l'échelle

TABLE 5.3 – Comparaison des approches métriques

Avantages	M	slim	MM	oignon	GH	GNAT	mVP	VP
Dimension	40	50	55	55	60	60	40	60
Volume	★	★	★	★	★	★	★	★★
Requêtes	kNN	kNN	kNN	kNN	kNN	kNN	kNN	kNN
Recouvrement	√	√	√	√				
Réinsertion		√	√	√				
Mémoire	√	√	√	√	√	√	√	√

TABLE 5.4 – Tableau comparatif des avantages des propositions étudiées (★ signifie une unité de taille de données)

Une extension de cette technique a été développée : l'arbre-oignon [24]. Son objectif est de diviser la quatrième région de l'arbre-MM afin de créer des agrandissements successifs. Cela améliore l'algorithme de recherche parce que la dernière région peut devenir particulièrement vaste si elle n'est pas subdivisée. À notre avis, le problème n'est pas totalement résolu, parce que la phase de construction reste lente en raison du coût de l'algorithme de semi-équilibrage.

Ce tableau 5.3, bien sûr, ne donne qu'un vue d'ensemble des avantages et inconvénients de ces méthodes.

5.4 Analyses

Après ce bref passage en revue et l'étude comparative faite des méthodes d'indexation dans les espaces métriques généraux et les espaces multidimensionnels, nous pouvons avancer quelques idées qui vont être par la suite à la base de nos choix dans la partie proposition.

Premièrement, nous constatons, comme plusieurs auteurs l'ont déjà signalé, que les espaces métriques sont devenus un modèle populaire pour contourner les limitation des espaces vectoriels dans différentes applications.

Nous avons étudié les algorithmes qui indexent les espaces métriques pour répondre aux requêtes de proximité. Nous n'avons pas seulement énuméré les approches existantes pour discuter de leurs bons et mauvais points. Il s'avère que la plupart des algorithmes existants sont des variations sur quelques idées communes. Au premier lieu, nous trouvons des méthodes basées sur le regroupement des objets [27, 81, 79] et d'autres basées sur le partage de l'espace. Parmi ces dernières, une partie des méthodes partage l'espace avec des sphères [91, 18, 65, 24] et d'autres avec des hyper-plans [20, 42, 63]. Donc nous remarquons qu'il y a beaucoup d'idées communes entre ces méthodes.

Nous pouvons également analyser les principaux facteurs qui influent sur l'efficacité des algorithmes de recherche dans les espaces métriques [19].

Parmi ces facteurs, dans les méthodes basées sur le regroupement, nous trouvons le facteur de chevauchement entre les régions qui va influencer par la suite les performances des algorithmes de recherche. En face, on se trouve avec les méthodes basées sur le partitionnement de l'espace et, dans ce cas, nous trouvons le facteur de topologie des régions. Généralement les méthodes ont deux choix. Dans un partage avec des boules, la taille de la boule est très influente car son volume grossit très vite par rapport à son rayon lors de l'insertion de nouveaux objets. En ce qui concerne les méthodes qui partagent avec des hyper-plans, on se retrouve avec des topologies très difficiles à gérer par le manque de compacité. Le problème reste encore ouvert.

Les principales conclusions de notre travail sont résumées comme suit :

- Nous avons présenté quelques principales techniques d'indexation basées sur des structures arborescentes car cette approche est largement utilisée dans ce domaine, même s'il existe d'autres approches.

- Une large classe de structures repose sur l'utilisation d'un certain nombre de pivots. Une autre classe, aussi importante, utilise le regroupement compact des objets dans des formes englobantes.

- Nous avons pu redécouvrir que les principaux facteurs qui influent sur l'efficacité des algorithmes de recherche sont la dimension intrinsèque de l'espace et le rayon de recherche.

- Plusieurs facteurs (types de données, performances des machines, etc.) influent sur ces techniques, d'où la difficulté de les comparer.

- Sachant que dans les dimensions élevées, le rayon de recherche nécessaire

pour récupérer un pourcentage fixe de la base de données est très grand, cela entraîne des recherches intrinsèquement inefficientes.

- Or, le problème de l'indexation et de la recherche de manière efficiente dans les espaces métriques peut être considéré comme déjà résolu en utilisant le parallélisme. Toutefois, cela reste théorique. Un but est donc de permettre la parallélisation de l'algorithme de recherche sur un nombre raisonnable et extensible de machines.

L'étude des principaux représentants des index multidimensionnels et métriques, vis-à-vis des taxinomies illustrées en figures 3.1 et 4.1 nous a permis, au travers des définitions et des algorithmes de construction des index et de recherche kNN dans ces derniers, de mettre en lumière les différences et surtout les ressemblances.

Un élément clé de l'étude est que l'utilisation d'index multidimensionnels pour des recherches *via* des distances, c'est-à-dire dans un espace métrique, se heurte à des difficultés sur les formes topologiques. En d'autres termes, on effectue des recherches autour d'une boule alors que l'espace est organisé en hyper-rectangles. En combinant cela au fait que déterminer une intersection entre un hyper-plan et un boule n'est pas une opération très précise, on se retrouve finalement avec des algorithmes délicats à préciser dans les détails et avec des possibilités d'élagages amoindries durant une recherche.

Un autre élément clé de l'étude est que les versions parfaitement balancées (arbre-R et variantes dans les espaces multidimensionnels, arbre-M et variantes dans les espaces métriques) amènent également à des algorithmes plus difficiles. Cette contrainte, d'origine « technique » puisqu'il s'agit d'écrire des versions paginées c'est-à-dire destinées à être stockées sur disque, n'est pas une nécessité absolue. En effet, si l'on voit l'indexation comme une forme de classification non supervisée, il n'y a aucune raison pour aboutir à un partitionnement parfaitement équilibré. Bien au contraire, la structure de l'arbre devrait suivre au plus près celle de classes hiérarchiques. D'ailleurs, sur ces mêmes structures d'index, nous avons vu qu'une meilleure qualité de l'indexation est obtenue grâce aux réinsertions (arbre-R+ et arbre-slim vis-à-vis respectivement de l'arbre-R et de l'arbre-M).

Sur les structures d'index restantes, essentiellement métriques, nous avons vu que les définitions et algorithmes associés sont proches les uns des autres. Dans tous les cas, il s'agit de *partitionner* l'espace, *via* des hyper-plans (arbre-*k*D et arbre-GH), des boules (arbre-VP) ou encore des intersections de boules (arbre-MM et arbre-oignon). Si les deux dernières structures donnent de meilleurs résultats expérimentaux, il nous semble que cela est dû à la notion d'intersection qui permet de réduire les volumes des partitions. On notera que l'arbre-SR utilise également cette technique (intersection entre une hyper-sphère et un hyper-rectangle pour améliorer les performances de l'arbre-R, son prédécesseur). L'amélioration appor-

tées par l'arbre-oignon peut, à notre avis, être rapprochée de celle de l'arbre-X ainsi que des index en grille (que nous n'avons toutefois pas pris le temps de développer). En effet, les élargissements successifs s'apparentent à la notion de super-nœud tandis que les emboîtements de boules créent un maillage qui est à la boule ce que la grille est à l'hyper-plan.

II

Proposition

Dans cette partie, nous introduisons notre proposition. Nous séparons son étude entre présentation et expérimentation, d'une part, et entre version séquentielle et version parallèle, d'autre part.

Notre proposition, l'arbre-IM, est fortement inspirée de l'arbre-MM qu'il tente d'améliorer d'une manière moins complexe que l'arbre-oignon. Nous proposons également de supprimer l'algorithme de semi-équilibrage, trop *ad hoc* à notre sens. Le but, dans un premier temps, est donc d'obtenir une structure d'index relativement simple à comprendre et à implémenter, et néanmoins compétitive avec les propositions antérieures vues dans l'état de l'art.

Dans un second temps, l'objectif est de proposer des recherches parallèles sur cette nouvelle structure d'index. En effet, il nous semble que la « malédiction de la dimension » laisse peu d'espoir d'arriver à des améliorations extraordinaires des performances en séquentiel. Après avoir trouvé une approche plus performante de par sa structure même, il nous faut donc la paralléliser pour tendre vers des temps de recherche logarithmiques, c'est-à-dire optimaux.

6

Arbre-IM

L'état de l'art a permis de mettre en avant plusieurs structures, dont certaines semblent être plus performantes que d'autres. Dans ce chapitre, nous proposons une structure d'index basée sur le partitionnement de l'espace métrique en utilisant boules et hyper-plans.

Comme nous indiquons en préliminaire, elle est principalement inspirée de l'arbre-MM. Nous la formalisons ensuite et présentons les algorithmes associés.

6.1 Idées préliminaires

Après quelques expériences avec les index de l'état de l'art, il nous est apparu que le partitionnement de l'espace est une technique qui amène à des structures de données – donc des algorithmes – plus simples.

Par ailleurs, le problème des volumes croissant de manière exponentielle dans les espaces de grande dimension plaide en faveur de techniques permettant sinon de réduire à l'envi du moins de limiter les volumes, voire de contrôler leur occupation. Avec des boules, cela peut s'obtenir *via* des intersections – idée de base de l'arbre-M – et *via* des boules emboîtées – idée complémentaire de l'arbre-oignon. Il ne faut toutefois pas abuser des emboîtements, les volumes des boules extérieures étant extrêmement plus grands que celui des boules internes. L'usage des hyper-plans est plus délicat (intersection entre une boule-requêtes et des hyper-plans). Nous pouvons toutefois les combiner avec les boules pour limiter les volumes et limiter les « trous » qui se produisent dans l'espace extérieur d'une boule.

Dans la famille des index métriques, seul l'arbre-M est proposé d'emblée en version paginée. Il existe néanmoins des versions paginées des arbres *k*D, GH et

VP. Les performances sur disque ne sont peut-être pas toujours à la hauteur concrètement [1, 85, 38], mais ces versions offrent davantage de latitude. Nous avons donc décidé de développer un index paginé.

Une raison complémentaire à ce dernier choix est de pouvoir disposer d'un sous-ensemble « significatif » d'éléments avant de procéder à un découpage hiérarchique. En effet, nous avons vu que les arbres MM et oignon sont obligés d'introduire des algorithmes de rééquilibrage de l'index pour faire face aux insertions incrémentales et dans un ordre aléatoire des éléments.

Enfin, même si nous n'exigeons pas de construire un arbre parfaitement balancé, nous pensons que le choix d'une version paginée devrait amener à un arbre quasi-équilibré. Ce choix est lié à la nature même d'une classification, si tant est que l'index que nous cherchons à construire s'apparente suffisamment à une classification hiérarchique.

Explicitons ci-dessous où nous amènent ces grandes orientations.

Rappelons que l'idée générale de l'arbre-MM est de partager l'espace *via* deux boules. On sélectionne deux pivots, p_1 et p_2, d'une manière aléatoire. Il donnent naissance à deux boules, $B_1(p_1, r)$ et $B_2(p_2, r)$ où le rayon r est la distance inter-pivots, $r = d(p_1, p_2)$. Cette structure permet de mettre en évidence quatre régions disjointes : (I) leur intersection, (II) et (III) leurs différences respectives, et (IV) les objets externes. Le partitionnement se poursuit de manière récursive.

Pour l'arbre-MM, les auteurs sont amenés à faire un choix quasi aléatoire des pivots (cf. section 4.2.3). En revanche, d'autres auteurs ont montré [71, 51] que le choix des pivots est un facteur important dans les algorithmes de recherche. Donc nous proposons par la suite de mettre en œuvre un choix des pivots qui peut améliorer en premier lieu la qualité de notre index ainsi que l'efficience des recherches.

Tout comme nous, les auteurs de l'index constatent que la quatrième région et sa taille peuvent poser beaucoup de problèmes [65]. Nous proposons de découper cette région en deux parties, grâce à un hyper-plan, afin de d'offrir une meilleure connexité des régions.

6.2 Formalisation

Nous introduisons donc un arbre quinquénaire, appelé arbre-IM (*intersection metric-tree*), une technique d'indexation dans les espaces métriques qui divise de manière récursive l'ensemble des données en cinq régions *disjointes* en sélectionnant deux pivots à chaque itération.

La figure 6.1 illustre de façon informelle la façon dont l'arbre est conçu à partir d'un nœud. La figure 6.2 illustre le développement d'un arbre.

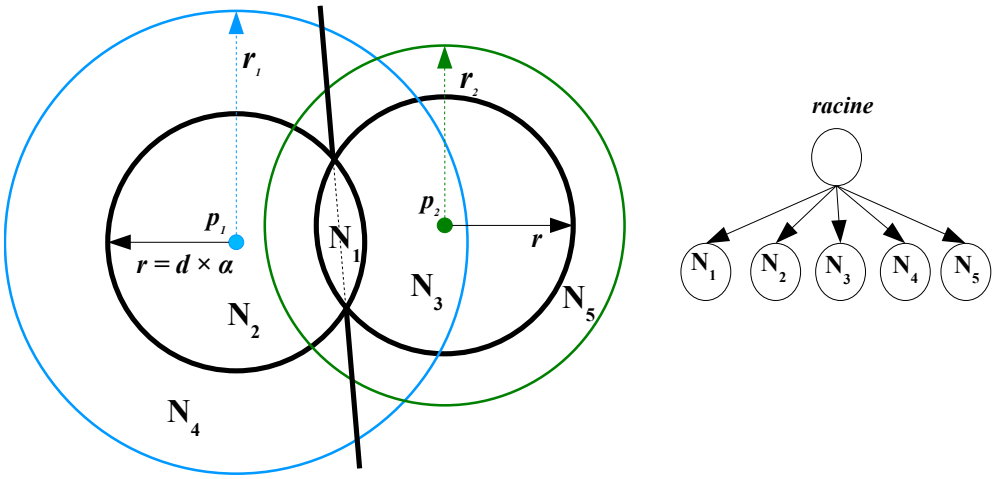


FIGURE 6.1 – Partitionnement de l'espace avec un arbre-IM

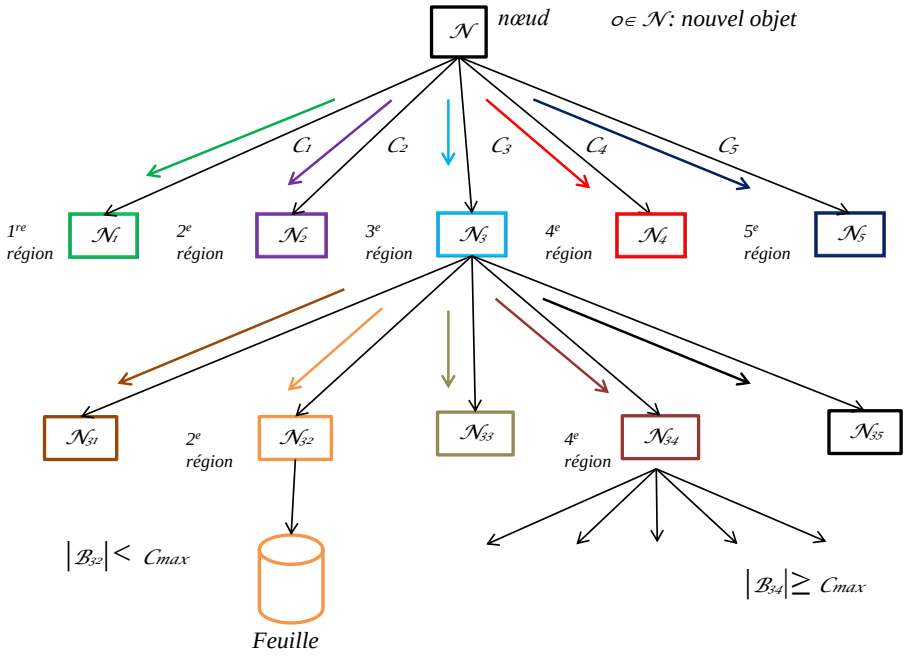


FIGURE 6.2 – Développment de l'arbre-IM

À chaque étape du processus récursif de construction de l'arbre, deux pivots sont choisis. À partir d'un sous-ensemble d'éléments, ils sont choisis comme étant les deux objets les plus éloignés l'un de l'autre.

Un rayon est calculé de manière à créer une intersection entre les deux boules centrées sur ces deux pivots. Par conséquent, le rayon varie entre la distance entre les deux pivots et leur milieu. La valeur exacte dépend d'un paramètre α présenté et discuté dans les sections suivantes. Ce paramètre est nécessaire sinon, du fait du choix de pivots, tous les objets se trouveraient dans l'intersection entre les deux boules.

Grâce à ces deux boules, nous pouvons diviser l'espace en cinq régions disjointes, à savoir (I) leur intersection (les boules), (II) et (III) leurs différences respectives, et (IV) et (V) les compléments pour chaque pivot respectivement.

Notez que nous ajoutons deux boules « extérieures » définies par les rayons r_1 et r_2, données par les points les plus éloignés de p_1 et p_2 respectivement et deux sous-ensembles partagés par un plan. Parfois ces points sont situés à l'intérieur des boules, parfois à l'extérieur. Ils aident à réduire plus rapidement le rayon r_q de la requête pendant le processus de recherche.

6.2.1 Définition

Présentons formellement l'arbre-IM.

Définition 21 (Arbre-IM) *Soit $M = (\mathcal{O}, d)$ un espace métrique. Soit $E \subseteq \mathcal{O}$ un sous-ensemble des objets qui vont être indexés. Soit $\frac{1}{2} < \alpha < 1$ un paramètre d'optimisation à étudier. Soit $c_{\max}$, avec $1 \leq c_{\max} \leq |E|$, le cardinal maximal que l'on associe à un nœud feuille.*

On définit les nœuds $\mathcal{N}$ d'un arbre-IM de la manière habituelle :

1. *Tout d'abord, un nœud feuille L_{IM} – ou simplement L – consiste en un sous-ensemble des objets indexés :*

$$E_L \subseteq E \tag{6.1}$$

   avec $|E_L| \leq c_{\max}$.

   Le contenu des feuilles partitionne E.

2. *Deuxièmement, un nœud interne N est un décuplet :*

$$(p_1, p_2, r, r_1, r_2, N_1, N_2, N_3, N_4, N_5) \in \mathcal{O}^2 \times \mathbb{R}^+ \times \mathbb{R}^{+2} \times \mathcal{N}^5 \tag{6.2}$$

où :

- (p_1, p_2) *sont deux objets* distincts*, c'est-à-dire avec $d(p_1, p_2) > 0$, appelés* pivots *;*

- $r = d(p_1, p_2) \times \alpha$ *aide à définir deux boules,* $B_1(p_1, r)$ *et* $B_2(p_2, r)$*, centrées sur* p_1 *et* p_2 *respectivement et ayant une valeur de rayon commune, assez grande pour que les deux boules aient une intersection non vide ;*

- (r_1, r_2) *sont les distances à l'objet le plus éloigné dans le sous-arbre enraciné en ce nœud* N *en ce qui concerne* p_1 *et* p_2 *respectivement, c'est-à-dire* $r_i = \max\{d(p_i, o), \forall o \in N\}$ *pour* $i \in \{1, 2\}$ *où la notation ensembliste* $o \in N$ *est abusivement utilisée pour l'union des extensions de feuilles qui sont enracinées en* N *;*

- $(N_1, N_2, N_3, N_4, N_5)$ *sont cinq sous-arbres, de telle sorte que :*

  - $N_1 = \{o \in N : d(p_1, o) \leq r \ \wedge \ d(p_2, o) \leq r\}$ *pour l'intersection ;*
  - $N_2 = \{o \in N : d(p_1, o) \leq r \ \wedge \ d(p_2, o) > r\}$ *pour la boule partielle centrée sur* p_1 *;*
  - $N_3 = \{o \in N : d(p_1, o) > r \ \wedge \ d(p_2, o) \leq r\}$ *pour la boule partielle centrée sur* p_2 *;*
  - $N_4 = \{o \in N : d(p_1, o) > r \ \wedge \ d(p_1, o) \leq d(p_2, o)\}$ *pour l'espace restant proche de* p_1 *;*
  - $N_5 = \{o \in N : d(p_2, o) > r \ \wedge \ d(p_1, o) > d(p_2, o)\}$ *pour l'espace restant proche de* p_2 *;*

  avec la même notation pour l'extension d'un nœud interne (cf. tableau 6.1).

6.2.2 Choix de α

Le rôle du paramètre α est d'assurer qu'il y a une intersection stricte et surtout de contrôler son volume. En imposant une valeur strictement inférieure à un, les pivots sont exclus de l'intersection. En imposant une valeur strictement supérieure à un demi, elle ne peut pas être absolument vide. Les expérimentations du chapitre suivant devront nous permettre de fixer une valeur moins empirique que deux tiers ou trois quarts.

6.2.3 Choix des pivots

Les pivots sont des objets qui sont utilisés pour éliminer au maximum des objets dans le parcours de recherche. Le choix des pivots est une étape très importante [10, 60, 91]. Bien que de nombreux algorithmes se basent sur un choix aléatoire ou arbitraire – les premiers arrivants –, ce dernier peut affecter la performance des algorithmes de recherche. Ce choix influe en premier lieu sur la répartition des objets dans l'ensemble de l'index, et du coup il influe sur l'efficacité des algorithmes

notation	description
α	coefficient pour équilibrer l'index
p_1	premier pivot
p_2	deuxième pivot
r	rayon des deux boules
r_1	distance à l'objet le plus éloigné dans le demi-plan associe à p_1
r_2	distance à l'objet le plus éloigné dans le demi-plan associe à p_2
N_1, N_2, N_3, N_4, N_5	cinq nœuds fils
$\|L\|$	cardinal de la feuille L (avec $\|L\| \leq c_{\max}$
$c_{\max}$	cardinal maximal d'une feuille

TABLE 6.1 – Différents paramètres de l'arbre-IM

$d(p_1, o)\ \theta\ d(p_2, o)$	$d(p_1, o)\ \theta\ r$	$d(p_2, o)\ \theta\ r$	$o \in \cdot$
	$\leq$	$\leq$	I
	$\leq$	$>$	II
	$>$	$\leq$	III
$\leq$	$>$		IV
$>$		$>$	V

TABLE 6.2 – Conditions d'appartenances d'un objet o à l'une des cinq régions

de recherche par la suite. Mais une telle distribution ou densité est difficile à trouver. C'est l'une des principales raisons pour lesquelles le choix aléatoire des pivots est très fréquent dans de nombreux algorithmes. Néanmoins, dans de nombreux cas, les résultats obtenus sont tout à fait acceptables [93]. Si cette étape doit être améliorée, elle doit donc l'être à un coût raisonnable.

Plusieurs heuristiques (comme le choix des pivots qui sont éloignés) ont été proposées pour améliorer la puissance d'élagage. Récemment, le problème a été étudié de façon systématique [22] et plusieurs stratégies de sélection des pivots ont été proposées et testées [91]. Selon ces auteurs, les bons pivots peuvent se résumer comme suit :

- un bon pivot est situé le plus loin possible du reste des objets dans l'espace métrique [22, 25, 93] ;
- de bons pivots sont situés le plus loin possible les uns des autres (cf. définition 22) [34, 10, 26].

Définition 22 (Diamètre) *Soit $(\mathcal{O}, d)$ un espace métrique. Soit $E \subseteq \mathcal{O}$ un sous-ensemble d'éléments.*

Alors $(p_1, p_2) = \text{argmax}_{(o_1,o_2)\in E^2}\{d(o_1, o_2) : d(o_1, o_2) > 0\}$ constitue un couple d'objets les plus éloignés l'un de l'autre.

Leur distance, $d(p_1, p_2)$, maximale, est le diamètre de E.

Nous nous appuyons sur cette définition pour choisir nos pivots, p_1 et p_2, lors du processus de construction de l'index.

Notons que le coût de l'algorithme correspondant est quadratique, c'est-à-dire bien trop important pour être considéré comme acceptable. Toutefois, tant que l'ensemble des données considérées ne dépasse pas la racine carrée de la taille de la collection complète, la détermination des deux éléments les plus éloignés l'un de l'autre dans ce sous-ensemble reste linéaire vis-à-vis de la collection dans son ensemble, c'est-à-dire acceptable. Par ailleurs, il est possible d'utiliser une version approximative et linéaire pour les cas où la taille du sous-ensemble est trop importante. Il s'agit d'une recherche itérée de l'élément le plus éloigné d'un objet donné, le tout premier étant choisi de manière aléatoire. C'est l'algorithme heuristique également utilisé par *FastMap* [35].

6.3 Construction de l'index

Pour la construction de l'index, nous proposons deux versions :

1. Une version non incrémentale, ou en lot, qui construit l'index à partir de toute la collection.

2. Une autre version qui insère les objets dans l'index au fur et à mesure de leur arrivée, c'est la version incrémentale.

6.3.1 Construction en lot

Commençons par la version en lot (*batch*) qui est plus simple que la version incrémentale. L'algorithme 26 décrit formellement cette version.

Au début, puis à chaque appel récursif, nous disposons de l'ensemble des objets, respectivement du sous-ensemble associé à un (futur) sous-arbre. Cela nous permet de choisir les deux objets, ou « pivots », les plus éloignés, ce qui va influencer la structure de l'index.

Prendre les deux objets les plus éloignés comme pivots amène à regrouper tous les objets à l'intérieur des deux boules et même dans leur seule intersection s'il n'y avait pas le paramètre α. Avec ce choix de pivots, le paramètre α permet finalement de répartir les objets sur les régions I à IV en évitant qu'il y en ait dans la région V.

Rappelons que l'opérateur argmax est remplacé par une *approximation linéaire en temps* afin d'éviter un calcul excessif en $O(|E|^2)$. (L'algorithme 32 décrit cette version dans l'annexe B.) La complexité globale de la construction de l'index est alors contenue en $O(n.\log n)$ avec $n = |E|$ ce qui en fait un algorithme suffisamment efficient pour être utilisé sur de grandes quantités de données, notamment stockées en mémoire secondaire. (Cette complexité ne tient pas compte de celle du calcul de la distance. Il faut les additionner.)

Algorithme 26 Construction en lot d'un arbre-IM

$$\text{Construire-IM}\left(\begin{array}{l} E \subseteq \mathcal{O}, \\ d \in \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+, \\ c_{\max} \in \mathbb{N}^*, \\ \alpha \in \left]\frac{1}{2}, 1\right[\end{array}\right) \in \mathcal{N}$$

avec :

- $(p_1, p_2) = \text{argmax}_{(o_1,o_2)\in E^2}\{d(o_1, o_2) : d(o_1, o_2) > 0\}$ [approximation linéaire] ;
- $r = d(p_1, p_2) \times \alpha$;
- $r_1 = \max\{d(p_1, e) : \forall e \in E\}$;
- $r_2 = \max\{d(p_2, e) : \forall e \in E\}$; [se limiter au demi-plan, c'est-à-dire E_4 et E_5]
- $E_1 = \{e \in E : d(p_1, e) \leq r \ \wedge \ d(p_2, e) \leq r\}$;
- $E_2 = \{e \in E : d(p_1, e) \leq r \ \wedge \ d(p_2, e) > r\}$
- $E_3 = \{e \in E : d(p_1, e) > r \ \wedge \ d(p_2, e) \leq r\}$
- $E_4 = \{e \in E : d(p_1, e) > r \ \wedge \ d(p_1, e) \leq d(p_2, e)\}$
- $E_5 = \{e \in E : d(p_2, e) > r \ \wedge \ d(p_1, e) > d(p_2, e)\}$

$$\triangleq \begin{cases} E & \text{si } |E| \leq c_{\max} \\ \left(\begin{array}{l} p_1, p_2, r, r_1, r_2, \\ \text{Construire-IM}(E_1, d, c_{\max}, \alpha), \\ \text{Construire-IM}(E_2, d, c_{\max}, \alpha), \\ \text{Construire-IM}(E_3, d, c_{\max}, \alpha), \\ \text{Construire-IM}(E_4, d, c_{\max}, \alpha), \\ \text{Construire-IM}(E_5, d, c_{\max}, \alpha) \end{array}\right) & \text{sinon} \end{cases}$$

Une fois les pivots déterminés (cf. définition 22), le principe du partage est celui donné par la définition 21. Ce partage se poursuit récursivement jusqu'à obtenir des nœuds d'au plus $c_{\max}$ éléments dans les feuilles de l'arbre.

6.3.2 Construction incrémentale

L'algorithme 27 décrit formellement le processus d'insertion progressive des objets dans l'index.

L'insertion se fait de manière descendante (*top-down*). Initialement, l'arbre est vide, c'est-à-dire qu'il s'agit d'une feuille avec un ensemble vide d'objets.

Les premières insertions dans une feuille ne font qu'augmenter sa taille jusqu'à un nombre maximum d'éléments $c_{\max}$.

En raison de considérations sur la complexité en temps, sa valeur ne peut pas être plus grande que $\sqrt{n}$ où $n = |E|$ est le cardinal de l'ensemble de la population d'objets à insérer dans l'arbre (*a priori* inconnu dans la version incrémentale, mais que l'on peut borner, voire modifier dynamiquement). En respectant cette condition, la complexité de la construction de l'arbre reste contenue en $O(n.\log n)$ (de nouveau sans tenir compte de la complexité du calcul d'une distance).

Lorsque le cardinal limite $c_{\max}$ est atteint, une feuille est remplacée par un nœud interne et cinq nouvelles feuilles sont obtenues en divisant le premier ensemble d'objets en cinq sous-ensembles en fonction des conditions indiquées dans la définition 21.

La fonction de partage de l'ensemble des objets est basée sur le choix de deux pivots *distincts*. Le choix de ces deux pivots joue un rôle important dans notre proposition avec le paramètre $c_{\max}$. Le but est d'équilibrer le plus possible l'arbre. Dans l'algorithme 27, nous implémentons donc le choix de deux objets les plus lointains l'un de l'autre. Malgré le fait que seulement un nombre restreint d'éléments sont pris en compte, des valeurs représentatives de l'ensemble de la collection des objets devraient néanmoins être choisies dès lors que la valeur de $c_{\max}$ n'est pas trop petite.

L'insertion d'un nouvel objet n'entraîne qu'un parcours descendant sélectionnant un unique sous-arbre jusqu'à la feuille qui contiendra l'objet. Par ailleurs, l'arbre a tendance à être plutôt équilibré (comme nous le verrons au chapitre 7), donc l'insertion d'un nouvel objet est une opération logarithmique, en coût amorti. Notons qu'à chaque nœud interne, seules deux distances sont calculées. La complexité logarithmique de l'insertion est donc à multiplier par deux fois celle de la distance utilisée. Comme effet secondaire, les rayons r_1 et/ou r_2 peuvent être modifiés. (Plus de détails sur le rôle de r_1 et r_2 seront donnés dans la fonction de recherche).

Algorithme 27 Insertion dans un arbre-IM

$$\text{Insérer-IM}\left(\begin{array}{l} o \in \mathcal{O}, \\ N \in \mathcal{N}, \\ d \in \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+, \\ c_{\max} \in \mathbb{N}^*, \\ \alpha \in \left]\frac{1}{2}, 1\right[\end{array}\right) \in \mathcal{N}$$

avec :

- $r'_1 = \max\{d(p_1, o), r_1\}$;
- $r'_2 = \max\{d(p_2, o), r_2\}$.

$$\triangleq \begin{cases} E \cup \{o\} & \text{si } N = E \ \wedge |E| < c_{\max} \\ \text{Construire-IM}(E \cup \{o\}, d, c_{\max}, \alpha) & \text{si } N = E \ \wedge |E| = c_{\max} \\ \left(\begin{array}{l} p_1, p_2, r, r_1, r_2, \\ \text{Insérer-IM}(o, N_1, d, c_{\max}, \alpha), N_2, N_3, N_4, N_5 \end{array}\right) & \text{si } \begin{array}{l} N = (p_1, p_2, r, r_1, r_2, \\ N_1, N_2, N_3, N_4, N_5) \wedge \\ d(p_1, o) \leq r \ \wedge \ d(p_2, o) \leq r \end{array} \\ \left(\begin{array}{l} p_1, p_2, r, r_1, r'_2, \\ N_1, \text{Insérer-IM}(o, N_2, d, c_{\max}, \alpha), N_3, N_4, N_5 \end{array}\right) & \text{si } \begin{array}{l} N = (\ldots) \wedge \\ d(p_1, o) \leq r \ \wedge \ d(p_2, o) > r \end{array} \\ \left(\begin{array}{l} p_1, p_2, r, r'_1, r_2, \\ N_1, N_2, \text{Insérer-IM}(o, N_3, d, c_{\max}, \alpha), N_4, N_5 \end{array}\right) & \text{si } \begin{array}{l} N = (\ldots) \wedge \\ d(p_1, o) > r \ \wedge \ d(p_2, o) \leq r \end{array} \\ \left(\begin{array}{l} p_1, p_2, r, r'_1, r_2, \\ N_1, N_2, N_3, \text{Insérer-IM}(o, N_4, d, c_{\max}, \alpha), N_5) \end{array}\right) & \text{si } \begin{array}{l} N = (\ldots) \wedge \\ d(p_1, o) > r \ \wedge \ d(p_1, o) \leq d(p_2, o) \end{array} \\ \left(\begin{array}{l} p_1, p_2, r, r_1, r'_2, \\ N_1, N_2, N_3, N_4, \text{Insérer-IM}(o, N_5, d, c_{\max}, \alpha)) \end{array}\right) & \text{si } \begin{array}{l} N = (\ldots) \wedge \\ d(p_2, o) > r \ \wedge \ d(p_1, o) > d(p_2, o) \end{array} \end{cases}$$

Algorithme 28 Recherche kNN dans un arbre-IM

$$\text{kNN-IM}\left(\begin{array}{l} N \in \mathcal{N}, \\ q \in \mathcal{O}, \\ k \in \mathbb{N}^*, \\ r_q \in \mathbb{R}^+ = +\infty, \\ A \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}} = \emptyset \end{array}\right) \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}}$$

avec :

- $A = ((d_1, o_1), (d_2, o_2), \ldots, (d_{k'}, o_{k'}))$;
- $C_1 = B(q, r_q) \cap B(p_1, r) \neq \emptyset \ \wedge \ B(q, r_q) \cap B(p_2, r) \neq \emptyset$, pour l'intersection ;
- $C_2 = B(q, r_q) \cap B(p_1, r) \neq \emptyset \ \wedge \ B(q, r_q) \cap B(p_2, r) \neq \emptyset$, la boule partielle centrée sur p_1 ;
- $C_3 = B(q, r_q) \cap B(p_1, r) \neq \emptyset \ \wedge \ B(q, r_q) \cap B(p_2, r) \neq \emptyset$, la boule partielle centrée sur p_2 ;
- $C_4 = B(q, r_q) \cap B(p_1, r_1) \neq \emptyset \vee B(q, r_q) \cap B(p_2, r_2) \neq \emptyset \wedge d(p_1, q) > d(p_2, q)$, pour le reste de l'espace situé dans le demi-plan associé à p_1 ;
- $C_5 = B(q, r_q) \cap B(p_1, r_1) \neq \emptyset \vee B(q, r_q) \cap B(p_2, r_2) \neq \emptyset \wedge d(p_1, q) \leq d(p_2, q)$, pour le reste de l'espace situé dans le demi-plan associé à p_2 ;
- une limite supérieure spécifique pour chaque branche :
  - $A_0 = A$;
  - $C_0 =$ vrai ;
  - $r_{q0} = \min\{r_q, d_{k'}\}$ si $k' = k$ sinon r_q ;
  - $A_i =$ kNN-IM$(N_i, q, k, r_{q_{i-1}}, A_{i-1})$ si C_i sinon A_{i-1} ;
  - $r_{q_i} = \min\{r_{q_{i-1}}, d_k\}$ si $|A_{i-1}| = k \ \wedge \ A_{i-1} = ((d_1, o_1), \ldots, (d_k, o_k))$ sinon $r_{q_{i-1}}$.

$$\triangleq \begin{cases} A, k\text{-tri}(A \cup \{(d(o, q), o) : o \in E_L\}) & \text{si } N = E_L \\ A_5 & \text{si } N = (p_1, p_2, r, r_1, r_2, N_1, N_2, N_3, N_4, N_5) \end{cases}$$

6.4 Requêtes de similarité

Dans cette partie, nous décrivons un algorithme de recherche pour répondre aux requêtes de type kNN avec l'arbre-IM. Nous introduisons un algorithme standard séquentiel.

Nous proposons une procédure de recherche avec l'algorithme 28 et son homologue davantage informatique, l'algorithme 29. C'est une version exacte. L'algorithme répond à une requête q pour rendre à la fin les k objets les plus proches de q, donc la réponse est un ensemble de k couples (o_i, d_i).

Cet algorithme est une version de recherche séquentielle, en profondeur d'abord et du type « séparation-et-évaluation ». L'arbre est donc parcouru en pré-ordre. Au

Algorithme 29 Recherche kNN dans un arbre-IM, version itérative

$$\text{kNN-IM}\left(\begin{array}{l} N \in \mathcal{N}, \\ q \in \mathcal{O}, \\ k \in \mathbb{N}^*, \\ r_q \in \mathbb{R}^+ = +\infty, \\ A \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}} = \emptyset \end{array}\right) \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}}$$

1: **si** $N \in \mathcal{P}(\mathcal{O})$ **alors**
2: **renvoyer** k-tri$(A \cup \{(d(o,q), o) : o \in N\})$
3: **sinon**
4: **soit** $(p_1, p_2, r, r_1, r_2, N_1, N_2, N_3, N_4, N_5) = N$
5: **si** $|A| = k$ **alors**
6: $r_q = \min\{r_q, \max\{d_i\} : (d_i, o_i) \in A\}\}$
7: **fin si**
8: **pour** $i \in \{1, \dots 5\}$ **faire**
9: **si** C_i **alors**
10: $A \leftarrow$ kNN-IM(N_i, q, k, r_q, A)
11: **si** $|A| = k$ **alors**
12: $r_q = \min\{r_q, \max\{d_i\} : (d_i, o_i) \in A\}\}$
13: **fin si**
14: **fin si**
15: **fin pour**
16: **renvoyer** A
17: **fin si**

avec :

- $C_1 = B(q, r_q) \cap B(p_1, r) \neq \emptyset \ \wedge \ B(q, r_q) \cap B(p_2, r) \neq \emptyset$, pour l'intersection ;
- $C_2 = B(q, r_q) \cap B(p_1, r) \neq \emptyset \ \wedge \ B(q, r_q) \cap B(p_2, r) \neq \emptyset$, la boule partielle centrée sur p_1 ;
- $C_3 = B(q, r_q) \cap B(p_1, r) \neq \emptyset \ \wedge \ B(q, r_q) \cap B(p_2, r) \neq \emptyset$, la boule partielle centrée sur p_2 ;
- $C_4 = B(q, r_q) \cap B(p_1, r_1) \neq \emptyset \vee B(q, r_q) \cap B(p_2, r_2) \neq \emptyset \wedge d(p_1, q) > d(p_2, q)$, pour le reste de l'espace situé dans le demi-plan associé à p_1 ;
- $C_5 = B(q, r_q) \cap B(p_1, r_1) \neq \emptyset \vee B(q, r_q) \cap B(p_2, r_2) \neq \emptyset \wedge d(p_1, q) \leq d(p_2, q)$, pour le reste de l'espace situé dans le demi-plan associé à p_2.

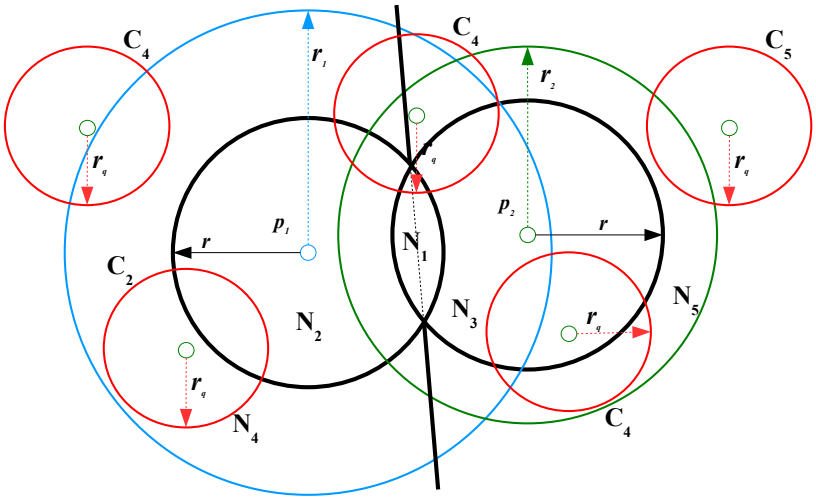


FIGURE 6.3 – Les différentes conditions d'intersections entre la boule-requête et les cinq régions de l'arbre-IM

début le rayon de requête r_q est initialisé à l'infini. La recherche s'effectue dans les nœuds internes, en profondeur d'abord. Le choix des nœuds candidats se fait par l'appartenance probable des réponses à ceux-ci s'il y a une intersection entre la boule-requête et la topologie de la région (cf. figure 6.3).

En arrivant à un nœud feuille, la solution calculée localement est fusionnée avec la pré-solution locale. Notons que le coût de calcul sur une feuille peut être contrôlé, donc rendu relativement faible, puisqu'il est effectué sur un ensemble d'objets dont le cardinal est limité par le paramètre $c_{\max}$.

Au fur et à mesure de la recherche, le rayon de requête r_q décroît de manière monotone pour aboutir à la distance à la k^e réponse. Une approximation de la réponse est ainsi construite et transmise d'appel en appel, aussi bien récursivement que d'un fils à un autre. Dans le second cas, la nouvelle valeur de r_q doit être prise en compte pour élaguer continuellement un maximum de branches. Dans cette proposition nous avons mis la priorité de la première visite sur la région qui couvre la requête q. Après, nous passons aux autres régions sans aucune priorité. Ce non-choix est contestable puisque nous avons vu que les arbre MM et oignon proposent des stratégies de parcours. En particulier, l'adaptation de la stratégie de l'arbre-MM est assez immédiate. (Une alternative, présentée dans l'annexe (cf. algorithme 43, s'appuie sur un tri des régions par rapport à leur distance maximale à la requête. Bien entendu, c'est la proposition la plus coûteuse en temps.)

6.5 Conclusion

Nous avons ici présenté une nouvelle méthode d'indexation dans les espaces métriques, bâtie sur le principe de base de l'arbre-MM vu dans l'état de l'art mais en

y apportant un grand nombre de modifications. Ces dernières sont inspirées de la synthèse critique faite sur un ensemble de techniques d'indexation dans les espaces multidimensionnels et métriques. Toutefois, toutes les variations *a priori* intéressantes n'ont pas été introduites dans notre proposition. Nous avons déjà introduit quelques paramètres supplémentaires et prendre en compte de nombreuses variations devrait en toute logique impliquer de faire des dizaines, des centaines, voire des milliers, de tests.

Nous avons ensuite décrit les algorithmes à mettre en œuvre pour construire puis effectuer des recherches kNN dans notre arbre-IM. L'annexe B présente une implémentation de référence rapide, en Python, des algorithmes décrits dans ce chapitre. La version finalement développée est une version véritablement paginée où les feuilles sont stockées sur disque tandis que l'on essaie de maintenir en mémoire centrale les nœuds internes.

À partir de ces codes, nous allons pouvoir tester, dans le chapitre suivant, l'efficience de notre proposition.

7

Expérimentations et résultats : approche séquentielle

L'objectif de l'arbre-IM étant de construire un index et de répondre *efficacement* aux requêtes kNN, dans ce chapitre nous présentons une évaluation expérimentale de cette structure.

Nous avons mis au point un prototype entièrement fonctionnel de l'arbre-IM. Notons qu'il ne s'agit pas de la version donnée en exemple dans l'annexe B mais d'une version stockant l'index de manière paginée et persistante sur disque ayant permis des expérimentations jusqu'à deux millions de données.

Nous comparons également notre structure avec d'autres méthodes d'indexation dans les espaces métriques, à savoir les arbres MM, oignon et slim.

7.1 Collections indexées et requêtes

Afin de montrer l'efficacité de notre approche, nous avons exécuté toutes les expériences sur des données réelles avec des distributions de données significativement différentes (cf. figure 7.1) pour démontrer le large éventail d'applicabilité de notre index.

Nous avons utilisé trois ensembles de données (cf. tableau 7.1) :

1. les coordonnées de « Villes de France » ;
2. les histogrammes de « Couleurs dominantes » issues de la base CoPHIR (descripteurs MPEG-7 SCD) ;[1].

[1]La collection COPhIR est disponible à l'adresse http://cophir.isti.cnr.it

Données	Distance	#Éléments	Dimensions Réelle	Dimensions Intrinsèque	Description
Villes de France	L_2	35 183	2	1,9	Cordonnées des villes de France
Couleurs dominantes	L_1	10 000	64	8,3	Ensemble de descripteurs de la base CoPHIR (couleurs dominantes)
Histogrammes de couleurs	L_1	68 025	32	9,8	Histogrammes de couleurs de la base KDD de l'université de Californie à Irvine

TABLE 7.1 – Caractéristiques des bases de données utilisées

3. les histogrammes de couleurs de la base « KDD 2008 ».[2]

Les données réelles choisies présentent des difficultés différentes. En particulier, il est nécessaire de déterminer la dimension intrinsèque et d'évaluer son influence sur les performances [25]. Le tableau 7.1 résume les caractéristiques de ces ensembles de données et de leur utilisation. Détaillons-les :

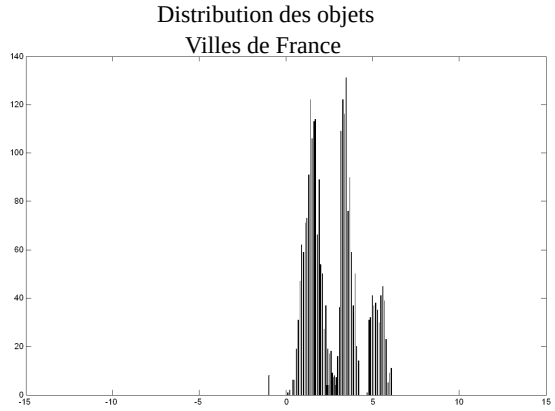
1. Nous commençons par les « Villes de France » qui présentent une faible dimension, tout à la fois visible et intrinsèque (cf. définition 9 en page 28). Elles sont donc *a priori* faciles à indexer au vu des résultats de la littérature.

2. Les descripteurs multimédias sont un bon exemple d'objets « complexes », en l'occurrence des histogrammes. Les descripteurs multimédias de la base « Couleurs dominantes » présentent une dimension réelle beaucoup plus importante : 64, la dimension intrinsèque étant également assez élevée : 8,3. Certains auteurs [25] estiment qu'au delà d'une dimension intrinsèque égale à 10, les données deviennent difficilement gérables et que cela est même impossible au delà d'une vingtaine.

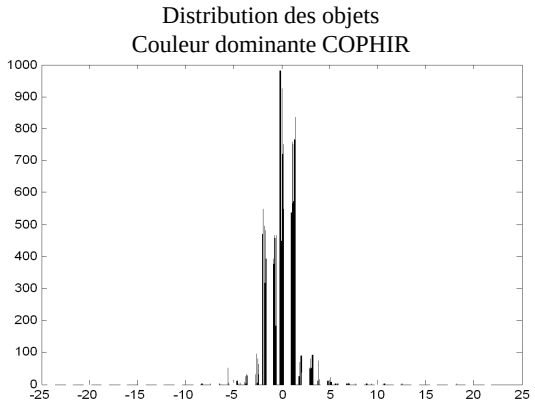
   Par ailleurs, la dimension réelle va jouer un rôle important dans le coût de construction et surtout dans les recherches puisque cela influe directement sur la complexité. La métrique utilisée, L_1, étant linéaire à évaluer, chaque calcul de distance va nécessiter $64 \times 3 = 192$ opérations élémentaires : différences, valeurs absolues et additions de la somme (cf. définition 2 en page 22).

3. Les histogrammes de couleurs de la collection « KDD 2008 » présentent une dimension intrinsèque quasiment égale à la limite empirique évoquée ci-dessus : 9,8. Cette différence avec la collection précédente, pour des descripteurs somme toute analogues, est certainement due au fait que les descripteurs SCD précédents sont des histogrammes issus d'une transformée de Haar qui a compressé l'information et donc réduit sa redondance, d'où sa dimension intrinsèque. Les différences sont visibles sur les distributions des distances observées (cf. figure 7.1) où nous avons utilisé deux échantillons de 1 000 vecteurs pour chaque collection.

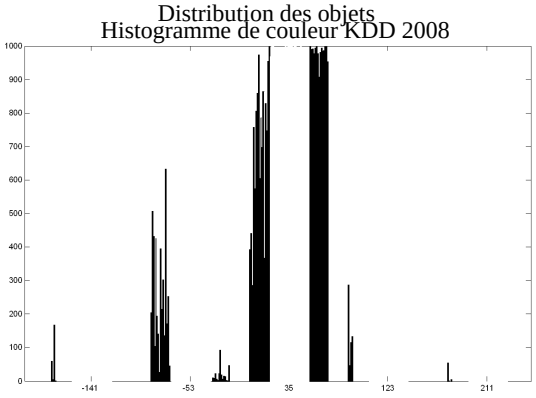
[2] Ce jeu de données peut être trouvé à l'adresse `http://kdd.ics.uci.edu`.



(a) « Villes de France »



(b) « Couleurs dominantes »



(c) « KDD 2008 »

FIGURE 7.1 – Distribution des distances sur 1 000 éléments de chaque collection

Nous présenterons des résultats sur ces trois ensembles de données.

7.2 Mesures relatives à la structure d'index

La « qualité » de l'index est très importante pour l'algorithme de recherche [73, 16, 94]. On peut la déterminer à travers différentes mesures, comme l'équilibrage de l'index, le taux de remplissage des nœuds, la distribution des objets dans l'index.

Nous commençons les études sur l'arbre-IM par des statistiques sur la qualité de la structure. Rappelons que dans les constructions incrémentales (en particulier pour l'arbre-*k*D ou l'arbre-MM), le risque de dégénérescence de l'arbre est important. (Les auteurs de l'arbre-MM ont bien rencontré ce problème et proposé une solution *ad hoc* au travers de l'algorithme de semi-équilibrage.) Notons qu'une structure d'index parfaitement équilibrée n'assure pas en elle-même des recherches performantes. Inversement, un index particulièrement déséquilibré présente des risques importants de baisses de performances sur certaines requêtes, bien que pouvant se comporter mieux en moyenne.

Notre but est donc de vérifier que l'arbre construit est relativement bien équilibré. Contrairement à l'arbre-MM, notre proposition est un arbre « paginé » (*bucketed tree*), c'est-à-dire dont les feuilles sont constituées d'un ensemble d'éléments au lieu d'un seul. Par conséquent, l'algorithme de semi-équilibrage proposé pour l'arbre-MM est remplacé par notre stratégie de fractionnement de la feuille.

7.2.1 Expérimentations

Nous arrivons à la phase d'expérimentation de l'algorithme de recherche kNN, dans une version séquentielle. Avant de commencer les expérimentations, nous décrivons tout d'abord le protocole suivi. Ensuite, les résultats bruts sont présentés *via* des tableaux et graphiques. Enfin, nous en tirons des conclusions.

Protocole des expérimentations

Dans cette partie nous avons lancé plusieurs expérimentations sur un prototype de l'arbre-IM sous les conditions suivantes :

1. Nous utilisons successivement les trois collections introduites ci-dessus, de difficultés intrinsèques différentes.

2. Pour chaque collection, nous utilisons les deux types de construction de l'index : en lot et incrémental.

3. Nous faisons varier le cardinal des feuilles, le paramètre $c_{\max}$, avec les valeurs $\sqrt{n}$ et $\log_2 n$, où n est la taille de l'ensemble de données.

α	niveau							Total
	1	2	3	4	5	6	7	
0,526	1	5	23	90	300	266	233	918
0,550	1	5	23	85	299	240	210	863
0,600	1	5	25	90	310	210	187	828
0,690	1	5	25	90	300	190	154	765
« parfait »	1	5	25	125	625	...	...	...

TABLE 7.2 – Nombre de nœuds par niveau de l'index sur la « Ville de France » avec $n = 35\,183$ et $c_{\max} = 187$

α	niveau						Total
	1	2	3	4	5	6	
0,526	1	5	12	70	210	170	468
0,550	1	5	12	75	219	166	478
0,600	1	5	16	82	224	88	416
0,690	1	5	16	130	230	35	417
« parfait »	1	5	25	125	625	...	...

TABLE 7.3 – Nombre de nœuds par niveau de l'index sur la « Couleurs dominantes » $n = 10\,000$ et $c_{\max} = 100$

4. Enfin, nous faisons varier le paramètre α dans l'intervalle $[1/2, 1]$ en choisissant les valeurs 0,52, 0,55, 0,6, et 0,69. Cela fait varier le rayon des deux boules, donc les volumes des régions I à V. Les quatre premières deviennent plus petites tandis que la région V est la seule à s'agrandir au détriment des autres.

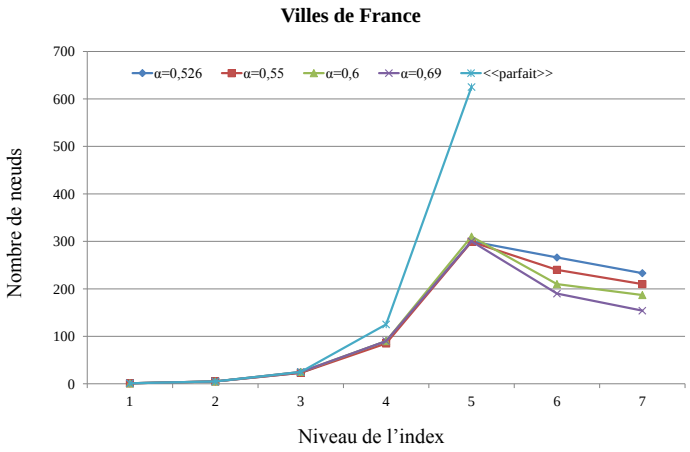
Dans chacun des scénarios, nous calculons :

- le nombre d'objets rencontrés pour répondre à une requête ;
- le nombre de distances calculées pour répondre à une requête ;
- le nombre de comparaisons nécessaires pour répondre à une requête ;

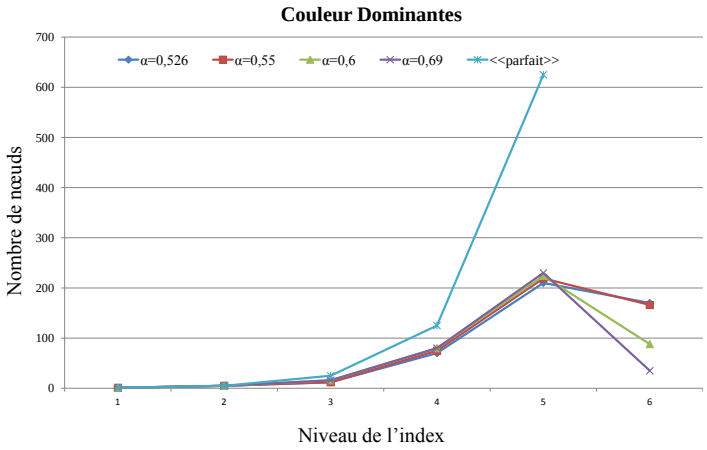
pour la construction d'un arbre-IM.

Résultats et analyse des expérimentations

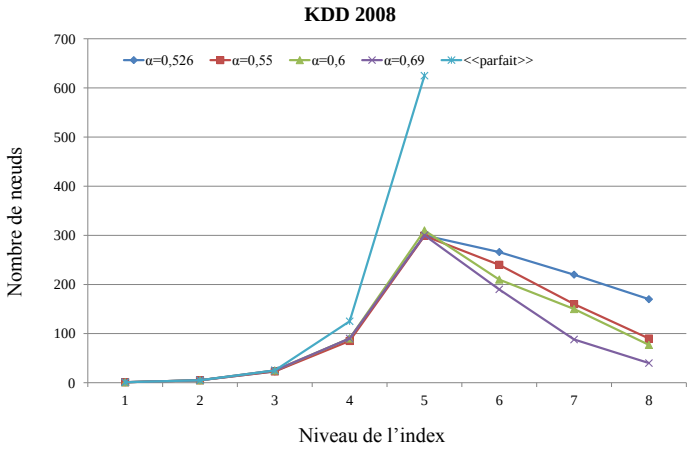
La figure 7.2 résume la structure d'index de l'arbre-IM sur les trois collections avec les différentes valeurs de α. Elles montrent le nombre de nœuds présents à chaque niveau de l'arbre ainsi que la profondeur maximale atteinte. Les détails numériques de ces histogrammes sont reportés dans les tableaux 7.2, 7.3 et 7.4. En général, la structure assure un quasi-équilibrage des index, donc le problème de dégénérescence ne se pose pas dans ce cas.



(a) « Villes de France »



(b) « Couleurs dominantes »



(c) « KDD 2008 »

FIGURE 7.2 – Nombre de nœuds niveaux par niveaux

α	niveau								Total
	1	2	3	4	5	6	7	8	
0,526	1	5	23	90	300	266	220	170	1 075
0,550	1	5	23	85	299	240	160	90	903
0,600	1	5	25	90	310	210	150	77	868
0,690	1	5	25	90	300	190	88	40	739
« parfait »	1	5	25	125	625	3 125	...	...	...

TABLE 7.4 – Nombre de nœuds par niveau de l'index sur l'histogramme de couleur « KDD 2008 » avec $n = 68\,025$ et $c_{\max} = 280$

Mode de construction	Nombre de	$c_{\max}$	α			
			0,52	0,55	0,6	0,69
Incrémental	distances calculées	log	220 111	244 014	258 985	250 658
Incrémental	distances calculées	sqrt	201 544	204 854	212 147	234 000
En lot	distances calculées	log	334 123	336 658	340 854	349 985
En lot	distances calculées	sqrt	310 654	312 456	318 547	320 014
Incrémental	comparaisons	log	134 541	137 554	144 547	151 414
Incrémental	comparaisons	sqrt	100 214	107 104	111 954	117 333
En lot	comparaisons	log	169 145	170 100	179 147	207 144
En lot	comparaisons	sqrt	155 002	160 178	168 147	180 942

TABLE 7.5 – Mesures de la construction des arbres « Villes de France »

Mode de construction	Mesure	$c_{\max}$	α			
			0,52	0,55	0,6	0,69
Incrémental	Distances calculées	log	104 375	115 658	120 951	126 789
Incrémental	Distances calculées	sqrt	95 123	100 489	101 002	120 958
En lot	Distances calculées	log	225 658	234 985	239 012	348 658
En lot	Distances calculées	sqrt	200 963	210 005	219 741	229 985
Incrémental	Nombre de comparaison	log	55 595	69 592	75 514	80 009
Incrémental	Nombre de comparaison	sqrt	49 655	55 658	56 987	60 954
En lot	Nombre de comparaison	log	114 325	120 658	129 654	140 956
En lot	Nombre de comparaison	sqrt	100 958	110 958	118 957	130 958

TABLE 7.6 – Mesures de la construction des arbres « Couleurs dominantes »

Mode de construction	Mesure	$c_{\max}$	α			
			0,52	0,55	0,6	0,69
Incrémental	Distances calculées	log	1 000 958	1 058 287	1 097 009	1 120 254
Incrémental	Distances calculées	sqrt	854 547	945 478	1 024 657	1 100 147
En lot	Distances calculées	log	1 225 654	1 240 159	1 250 753	1 281 620
En lot	Distances calculées	sqrt	1 117 367	1 125 931	1 140 954	1 148 754
Incrémental	Nombre de comparaison	log	574 990	580 987	566 555	570 310
Incrémental	Nombre de comparaison	sqrt	404 084	418 887	420 954	450 987
En lot	Nombre de comparaison	log	622 987	640 592	649 984	668 984
En lot	Nombre de comparaison	sqrt	600 554	650 789	680 199	690 214

TABLE 7.7 – Mesures de la construction des arbres « KDD 2008 »

Les tableaux 7.5, 7.6 et 7.7 montrent les détails des nombres de calculs de distances et des nombres de comparaisons pour les deux versions des algorithmes de construction, incrémentale et en lot, tout en faisant varier aussi les deux paramètres α et $c_{\max}$ pour chacune des trois collections.

Nous faisons les constatations suivantes :

1. À deux rares exceptions près, on constate que sur chaque ligne le coût augmente avec la valeur croissante de α.

2. On note, en comparant les lignes successives deux à deux, que les regroupements avec une valeur de $c_{\max} = \sqrt{n}$ sont toujours meilleurs qu'avec $c_{\max} = \log_2 n$, toutes choses étant égales par ailleurs.

3. En comparant maintenant la version en lot avec la version incrémentale, sous les mêmes conditions par ailleurs, on se rend compte que la version incrémentale, *a priori* moins informée, se comporte finalement mieux que la version en lot.

Finalement, il semble que la plus petite valeur de α est à préférer. En fait, on peut même se poser la question de savoir s'il ne faut pas la rendre égale à 0,5, ce qui revient à faire disparaître la région I, c'est-à-dire l'intersection entre les deux boules ! C'est presque le choix fait dans une proposition récente et non présentée dans ce mémoire [59].

On choisit une taille importante pour $c_{\max}$, soit $\sqrt{n}$, ce qui permet d'aboutir à une version paginée sur disque qui peut mieux exploiter les entrées-sorties.

On retient la version incrémentale de l'algorithme de création d'un arbre-IM, ce qui est une bonne nouvelle pour l'applicabilité de la proposition dans un contexte réel.

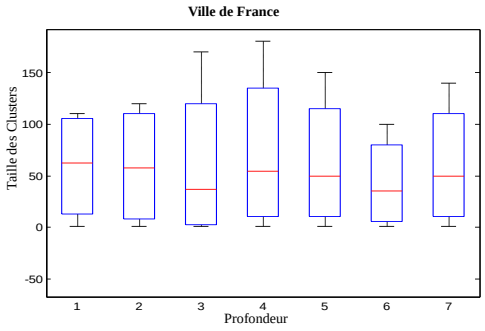
Nous passons maintenant à la distribution des objets dans l'index et au taux de remplissage des nœuds feuilles.

Bien entendu, nous travaillons sur les mêmes collections et nous fixons les paramètres α et $c_{\max}$ selon les résultats trouvés précédemment. Nous calculons, d'une part, la taille moyenne des feuilles durant le processus de construction et, d'autre part, la taille moyenne des clusters dans tous les niveaux de l'index.

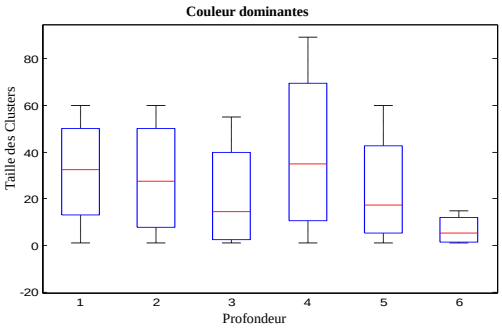
La figure 7.3 montre les répartitions des objets dans l'ensemble des nœuds de l'index (nœuds interne et feuilles) dans chaque niveau pour chaque collection.

Après, nous avons calculé la taille des clusters, mais cette fois-ci uniquement dans les feuilles et par rapport à chaque région, toujours en utilisant l'algorithme de construction incrémentale. La figure 7.4 montre le nombre d'objets pour chaque région de l'index pour chaque collection.

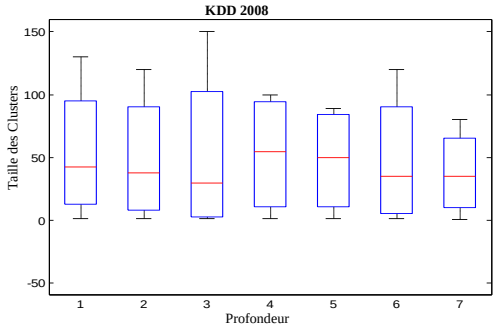
Les détails numériques de ces histogrammes sont reportés dans les tableaux 7.8, 7.9 et 7.10.



(a) « Villes de France »

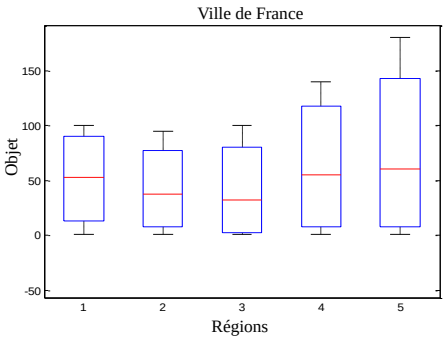


(b) « Couleurs dominantes »

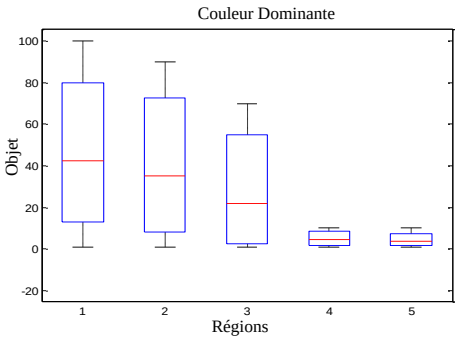


(c) « KDD 2008 »

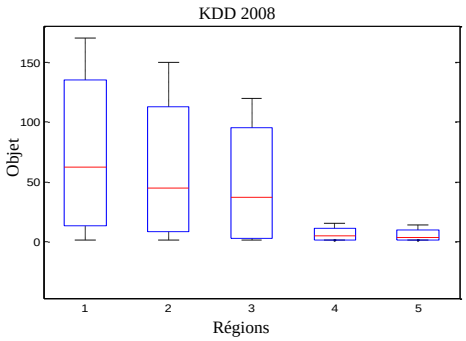
FIGURE 7.3 – Distribution des données dans l'arbre-IM



(a) « Villes de France »



(b) « Couleurs dominantes »



(c) « KDD 2008 »

FIGURE 7.4 – Nombre d'éléments par région dans la structure de l'index

Niveau	1	2	3	4	5	6	7
Minimum	25	15	4	20	20	10	20
Moyenne	99,7	89,9	70,1	88,8	80,1	60,4	80,2
Variance	15,2	18,2	22,7	26,8	25,5	20,1	28,6
Maximum	110	120	170	180	150	100	140

TABLE 7.8 – Distribution des données dans l'arbre-IM « Villes de France »

Niveau	1	2	3	4	5	6
Minimum	25	15	4	20	10	2
Moyenne	40,4	40,1	25,4	49,9	25,2	15,1
Variance	10,5	20,5	16,3	18,4	14,7	20,3
Maximum	60	60	55	89	60	90

TABLE 7.9 – Distribution des données dans l'arbre-IM « Couleurs dominantes »

Nous faisons principalement les deux constatations suivantes :

1. En fixant la taille maximale des feuilles de l'index à $c_{\max} = \sqrt{n}$ objets, et en imaginant que l'algorithme par lot amène à un index parfaitement équilibré, alors l'arbre quinquénaire parfaitement équilibré aurait une hauteur égale à $\log_5 \sqrt{n}$.

   Puisque nous employons l'algorithme incrémental, nous ne pouvons raisonnablement pas nous attendre à ce que chaque feuille contienne exactement $\sqrt{n}$ objets ; nous ne pouvons pas imaginer que l'équilibre sera parfait, même pas à ce qu'aucun nœud fils ne soit vide. Dans ce cadre, la distribution observée de la figure 7.2 est alors très satisfaisante.

   Finalement, la profondeur observée dans les trois collections n'est pas plus de deux fois celle correspondant au cas parfait. Par ailleurs, nous avons observé que le nombre moyen d'objets par feuille est de 54 dans les villes de France et 26 dans les couleurs dominantes et 60 dans les KDD 2008.

2. La figure 7.4 montre le nombre d'objets pour chaque région de l'index pour chaque collection.

   Il semble qu'une distribution parfaite d'un tiers de la population pour chacune des régions N_1 à N_3 est le meilleur choix (cf. figure 7.4). Autrement dit, les régions N_4 et N_5 devraient rester aussi vides que possible. Ainsi, l'objectif

Niveau	1	2	3	4	5	6	7
Minimum	25	15	4	20	20	10	20
Moyenne	60,2	59,7	54,8	89,2	89,4	60,5	79,9
Variance	20,2	20,1	25,0	20,4	19,7	18,3	20,2
Maximum	130	120	150	100	80	120	150

TABLE 7.10 – Distribution des données dans l'arbre-IM « KDD 2008 »

semble être d'obtenir un arbre ternaire, les régions N_4 et N_5 étant surtout utilisées pour les « valeurs aberrantes » comme dans la collection « Villes de France» où la distribution est presque équilibrée, au vu d'une distribution des donnée dans l'espace bien moins gaussienne que pour les deux autres collections (cf. figure 7.1).

7.2.2 Comparaison avec des techniques récentes

Dans cette partie, nous comparons les performances de notre approche à celles de trois méthodes précédentes. Évidemment, nous nous comparons à l'arbre-MM [80] dont notre approche est inspirée, à son extension, l'arbre-oignon, ainsi qu'à l'arbre-slim [80], une version améliorée de l'arbre-M.

Nous avons utilisé la bibliothèque C++ « GBDI Arboretum » qui implémente ces méthodes. [3]

Protocole des expérimentations

Dans cette partie nous avons lancé les expérimentions sur les quatre prototypes sous les conditions suivantes :

1. Nous utilisons successivement les trois collections introduites précédemment.
2. Pour chaque collection, nous utilisons la construction incrémentale de l'index.
3. Enfin, nous faisons varier le cardinal des feuilles, le paramètre $c_{\max}$, avec les valeurs $\sqrt{n}$ et $\log_2 n$, où n est la taille de l'ensemble de données uniquement pour le prototype de l'arbre-IM.

Dans chacun des scénarios, nous calculons :

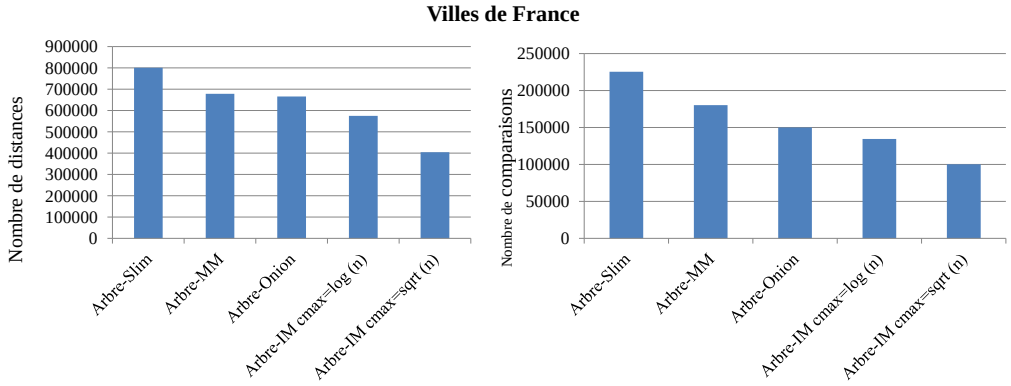
- le nombre de distances calculées ;
- le nombre de comparaisons nécessaires ;

pour la construction des index.

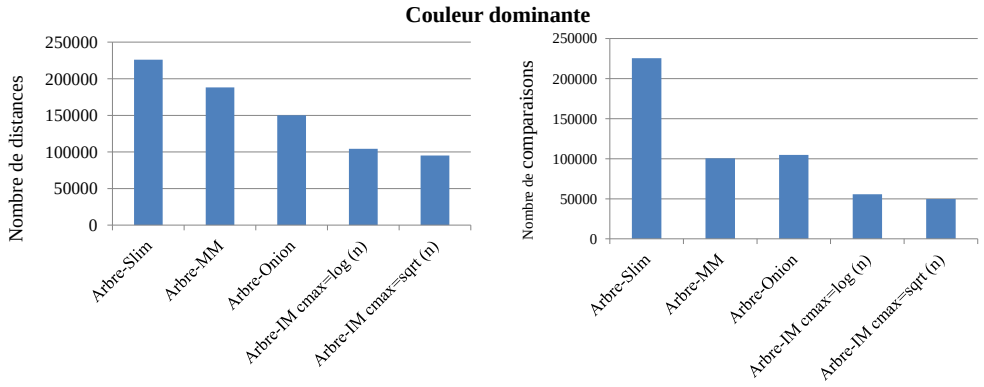
Dans la figure 7.5, nous voyons clairement que notre proposition est la plus performante par rapport aux autres avec une différence de plus de 10 % avec l'arbre-oignon dans les trois collections (en moyenne et avec les deux valeurs de $c_{\max}$), et avec plus que 20 % par rapport à l'arbre-slim.

La différence par rapport aux arbres MM et oignon est facilement explicable. La raison en est l'absence respective des algorithme de « semi-équilibrage » et

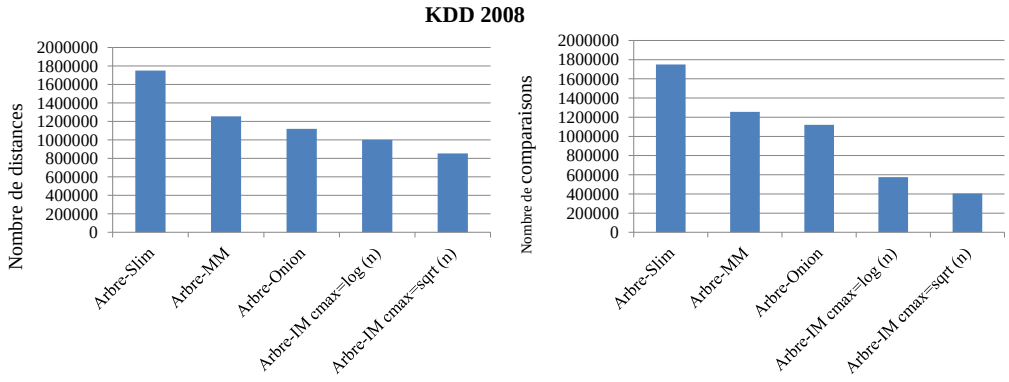
[3] GBDI Arboretum est une bibliothèque C++ qui implémente différentes méthodes d'accès métrique (MAM) (cf. http://http://www.gbdi.icmc.usp.br/old/arboretum).



(a) « Villes de France »



(b) « Couleurs Dominantes »



(c) « KDD 2008 »

FIGURE 7.5 – Nombre de distances calculées et nombre de comparaisons effectuées dans la construction de l'arbre-IM (avec $c_{\max}$ égal à $\log n$ ou à $\sqrt{n}$) en comparaison avec les arbres slim, MM et oignon

		Arbre				
		IM		MM	oignon	slim
Collection	Nombre de	$c_{\max} = \log n$	$c_{\max} = \sqrt{n}$			
Villes de France	distances	574 990	404 084	678 254	666 025	800 457
	comparaisons	134 541	100 214	180 325	150 144	225 458
Couleurs dominantes	distances	104 375	95 123	188 254	150 222	225 987
	comparaisons	55 595	49 655	100 555	104 852	225 411
KDD 2008	distances	1 000 958	854 547	1 255 360	1 120 480	1 750 000
	comparaisons	574 990	404 084	1 180 147	1 190 987	1 651 147

TABLE 7.11 – Détail du nombre de distances calculées et du nombre de comparaisons effectuées dans la construction de l'arbre-IM (avec $c_{\max}$ égal à $\log n$ ou à $\sqrt{n}$) en comparaison avec les arbres MM, oignon et slim

de « *keep-small* » qui nécessitent un nombre important de calculs de distances et de comparaisons. Dans le cas de l'arbre-slim, rappelons-nous que l'algorithme « *slim-down* » a également un coût important qui a été noté par ses propres auteurs [81, 77].

Cela montre déjà que notre approche, bien qu'assez simple dans le principe d'insertion des nouveaux éléments (ce qui était l'un des objectifs initiaux vis-à-vis de la complexité en temps) permet d'obtenir une structure d'index compétitive. Rappelons notamment que l'un des objectifs initiaux était de se restreindre à une classe de complexité en temps en $O(n.\log n)$, afin de permettre des ajouts incrémentaux à un coût raisonnable.

7.3 Performances des recherches kNN

Nous arrivons à la phase d'expérimentation de l'algorithme de recherche kNN. Avant de commencer les expérimentations, nous décrivons tout d'abord le protocole. Ensuite, les résultats bruts sont présentés *via* tableaux et graphiques. Enfin, nous en tirons nos conclusions.

7.3.1 Protocole des expérimentations

Les éléments du protocole concernent pour partie les éléments introduits précédemment – collections et paramètres de l'arbre – et pour partie des éléments liés à la recherche kNN – *k* et algorithmes. De plus, nous précisons quelles mesures sont effectuées.

Collections. Nous retrouvons ici les mêmes paramètres que dans le protocole d'expérimentation des algorithmes de construction des arbres (cf. section 7.2.1 en page 120). Pour mémoire, nous utilisons trois collections de complexités intrinsèques différentes : « Villes de France », « Couleurs dominantes » et « KDD 2008

».

Paramètres de construction de l'arbre-IM. Par ailleurs, nous effectuons les recherches sur les index construits à partir des versions incrémentales et en lot, en faisant varier les paramètres α (0,52, 0,55, 0,6 et 0,69) et $c_{\max}$ ($\sqrt{n}$ et $\log n$).

Taille du résultat. Un premier paramètre supplémentaire est *k*. Nous effectuons des recherches avec $k \in \{5, 10, 15, 20, 50, 100\}$.

Algorithmes de recherche. De plus, nous utilisons trois variantes algorithmiques :

1. *Naïf.* L'algorithme naïf est essentiellement un tri amélioré (de notre point de vue) dans la mesure où il ne trie que les *k* premiers éléments. Par conséquent, sa complexité en temps est seulement en :

$$O(n.d) + O(n. \log_2 k) \tag{7.1}$$

   où d indique ici la complexité du calcul d'une distance.

   Il sert d'étalon de comparaison, dans ce qui devrait être le *pire* des cas.

2. *Séquentiel.* L'algorithme dit séquentiel est celui présenté dans l'algorithme 28. Sa complexité en temps est en :

$$O(n_1) + O(n_2.d) + O(n_3. \log k). \tag{7.2}$$

   où :

   - n_1 est le nombre d'objets considérés lors du processus de recherche ;
   - n_2 est le nombre de distances calculées lors du processus de recherche ;
   - n_3 est le nombre d'objets qui ont été retenus comme candidats à un moment ou à un autre durant le processus de recherche.

   Ces valeurs sont ordonnées :

$$n \geq n_1 \geq n_2 \geq n_3. \tag{7.3}$$

   En effet, un certain nombre d'éléments, n_1, va être parcouru durant la recherche, d'autres étant élagués. Pour ceux, n_2, que l'on ne peut pas exclure comme candidats en se basant sur l'inégalité triangulaire, il faudra calculer explicitement la distance à l'objet-requête. Enfin, il n'y a que ceux, n_3, qui sont encore des candidats potentiels qui seront insérés dans la pré-solution.

La comparaison avec la complexité d'un « *k*-tri » montre immédiatement que cet algorithme ne devrait pas être moins bon que l'algorithme naïf. Il sera d'autant meilleur que les valeurs successives n_1 à n_3 seront rapidement décroissantes. (En pratique, dans des cas défavorables, la surcharge de travail pourrait amener à légèrement dépasser cette limite haute. En particulier, les éléments stockés dans les nœuds internes comme pivots apparaissent aussi dans les feuilles comme éléments indexés. Le nombre effectif d'objets présents dans l'arbre-IM est donc légèrement supérieur à celui de la collection en raison de ces doublons.)

3. *Parfait.* Un algorithme « parfait » est introduit. Il s'agit d'une recherche basée sur la version précédente mais lancée avec un rayon r_q initialisé par avance à la distance exacte au k^e objet ! Il s'agit donc d'un cas idéal qui permet une comparaison de base avec les deux autres approches. *A priori*, il doit s'agir du *meilleur* cas, c'est-à-dire qu'il doit fournir un minorant aux performances que l'on peut espérer atteindre avec l'arbre-IM. En d'autres termes, il amène, vis-à-vis d'une instance d'arbre-IM donnée, aux valeurs minimales de n_1, n_2 et n_3.

Mesures. Pour une meilleure interprétation des résultats de nos expériences, nous rappelons que nous avons exécuté une centaine de requêtes kNN différentes sur les collections, et avons calculé la moyenne des résultats.

Nous mesurons :

(i) le pourcentage moyen de feuilles visitées ;

(ii) le pourcentage moyen de nœuds internes visités ;

(iii) le nombre moyen de distances calculées ;

(iv) le nombre moyen de comparaisons ;

(v) le nombre moyen d'objets visités.

7.3.2 Résultats et analyse des expérimentations

Nous analysons maintenant les performances de l'algorithme de recherche kNN. Dans les trois tableaux 7.12, 7.13 et 7.14, nous montrons les performances de l'algorithme de recherche kNN (nombre de feuilles et de nœuds internes visités, nombre de calculs de distances et de comparaisons effectués et nombre d'objets candidats) dans l'arbre-IM pour chaque collection avec une valeur de $c_{\max}$ correspond à la racine carrée de n. Nous faisons varier à chaque fois la valeur du paramètre α. Nous calculons à chaque fois les $k = 5$ plus proches voisins.

	nœuds				
α	feuilles (%)	internes (%)	#comparaison (n_3)	#distances (n_2)	#objets (n_1)
0,526	6,75	8,16	9 111	8 011	3 888
0,530	7,25	8,80	9 210	8 423	3 930
0,550	7,80	9,14	9 268	8 735	4 172
0,590	8,03	9,63	9 514	8 946	4 220
0,600	8,23	10,24	9 570	9 258	4 345
0,650	8,88	10,83	10 147	9 664	4 559
0,690	9,01	11,34	10 249	9 879	4 720

TABLE 7.12 – Statistiques sur les performances de l'arbre-IM sur la collection « Villes de France » avec $n = 35\,183$, $c_{\max} = \sqrt{n}$ et $k = 5$

	nœuds				
α	feuilles (%)	internes (%)	#comparaison (n_3)	#distances (n_2)	#objets (n_1)
0,526	7,65	9,36	3 854	3 420	2 951
0,530	8,37	9,71	4 200	3 488	2 951
0,550	8,75	10,21	4 277	3 700	3 001
0,590	9,03	10,90	4 529	3 985	3 254
0,600	9,05	11,01	4 564	4 200	3 625
0,650	10,02	11,63	5 035	4 666	3 748
0,690	10,11	11,69	5 474	4 820	3 812

TABLE 7.13 – Statistiques sur les performances de l'arbre-IM sur la collection « Couleurs dominantes » avec $n = 10\,000$, $c_{\max} = \sqrt{n}$ et $k = 5$

Nous avons vu dans la première partie expérimentale (cf. section 7.2) que le paramètre α a joué un rôle sur la distribution des objets dans les cinq régions. Ici nous remarquons que, pour les trois collections, il existe un gain sur les performances de 2 à 3 % tout en diminuant la valeur de α.

Deuxièmement, dans les tableaux 7.15, 7.16 et 7.17, nous faisons varier les types d'algorithmes de recherche (naïf, séquentiel et parfait) tout en gardant à chaque fois la meilleure valeur de α et en affectant au paramètre $c_{\max}$ la valeur $\sqrt{n}$ et $k = 5$ pour toutes les collections. Sans surprise, la version naïve est la plus coûteuse et la version parfaite est la plus efficace. Mais nous pouvons voir que la version standard séquentielle fonctionne efficacement, et reste concurrente

	nœuds				
α	feuilles (%)	internes (%)	#comparaison (n_3)	#distances (n_2)	#objets (n_1)
0,526	8,00	10,14	18 654	15 111	5 900
0,530	8,33	10,70	19 001	17 654	6 120
0,550	9,05	11,55	19 120	17 799	6 222
0,590	10,03	11,70	19 500	18 787	6 454
0,600	11,05	12,25	20 004	19 330	6 600
0,650	12,02	15,14	20 145	19 787	7 001
0,690	13,11	15,65	20 379	20 020	7 221

TABLE 7.14 – Statistiques sur les performances de l'arbre-IM sur la collection « KDD 2008 » avec $n = 68\,025$, $c_{\max} = \sqrt{n}$ et $k = 5$

	nœuds				
algorithme	feuilles (%)	internes (%)	#comparaison (n_3)	#distances (n_2)	#objets (n_1)
naïf	N/A	N/A	35 183	35 183	35 183
séquentiel	6,75	8,16	9 111	8 011	3 888
parfait	2,22	4,16	2 958	2 322	1 500

TABLE 7.15 – Statistiques de performance de quatre algorithmes de recherche kNN dans un arbre-IM sur la collection « Villes de France » avec $n = 35\,183$, $c_{\max} = \sqrt{n}$ et $k = 5$

	nœuds				
algorithme	feuilles (%)	internes (%)	#comparaison (n_3)	#distance s(n_2)	#objets (n_1)
naïf	N/A	N/A	10 000	10 000	10 000
séquentiel	7,65	9,36	3 854	3 420	2 951
parfait	2,42	5,22	1 298	960	1 100

TABLE 7.16 – Statistiques de performance de quatre algorithmes de recherche kNN dans un arbre-IM sur la collection « Couleurs dominantes » avec $n = 10\,000$, $c_{\max} = \sqrt{n}$ et $k = 5$

avec la version parfaite, bien qu'elle ne puisse pas recueillir autant d'informations concernant le rayon de requête r_q.

Du point de vue de la complexité, pour chaque requête nous comptons :

(i) le nombre de distances calculées (n_1) ;

(ii) le nombre de comparaisons (n_2) ;

(iii) le nombre d'objets qui ont été candidats pendant tout le parcours de la recherche (n_3).

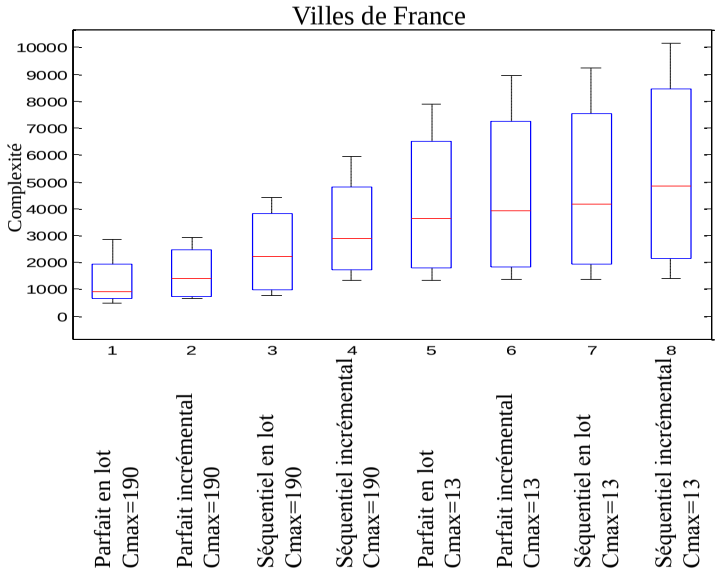
Leur somme détermine la complexité exacte de chaque requête. Nous utilisons cette complexité effective et déterminons les minimums, maximums, moyennes et variances sur les cent requêtes. Les figures 7.6 et 7.7 illustrent graphiquement ces mesures sur le comportement de nos algorithmes de recherche avec toutes leurs variations avec $k \in \{5, 50\}$.

Pour plus de détails, les tableaux 7.18, 7.19 et 7.20 montrent les chiffres exacts des performances des algorithmes de recherche.

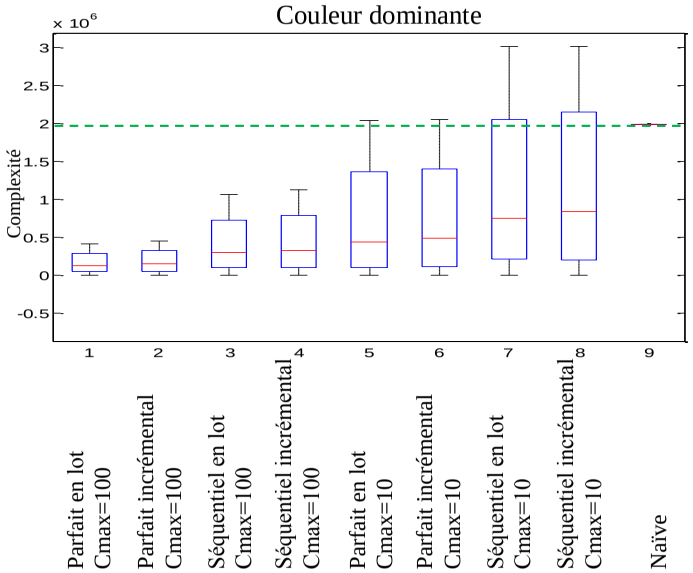
En premier lieu, nous constatons que la version naïve est la plus coûteuse. Après vient la version parfaite qui est la plus efficace. En même temps nous remar-

	nœuds				
algorithme	feuilles (%)	internes (%)	#comparaison (n_3)	#distances (n_2)	#objets (n_1)
naïf	N/A	N/A	68 025	68 025	68 025
séquentiel	8,00	10,14	18 654	15 111	5 900
parfait	2,42	5,22	10 369	8 987	2 388

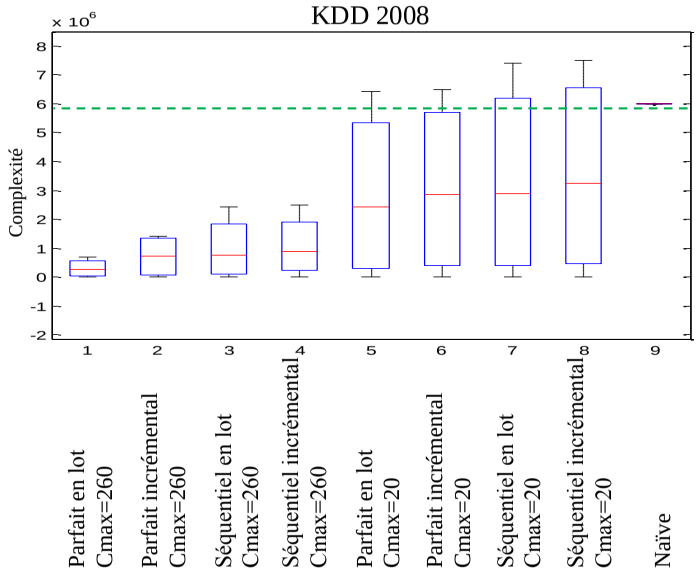
TABLE 7.17 – Statistiques de performance de quatre algorithmes de recherche kNN dans un arbre-IM sur la collection « KDD 2008 » avec $n = 68\,025$, $c_{\max} = \sqrt{n}$ et $k = 5$



(a) « Villes de France »

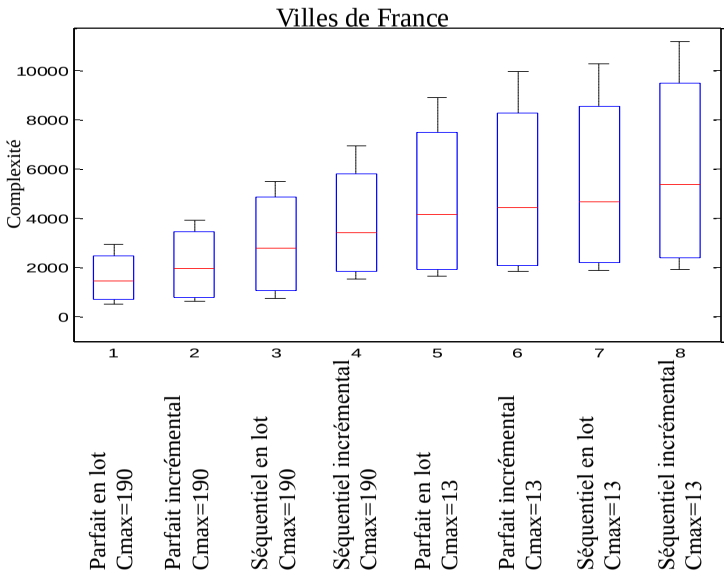


(b) « Couleurs dominantes »

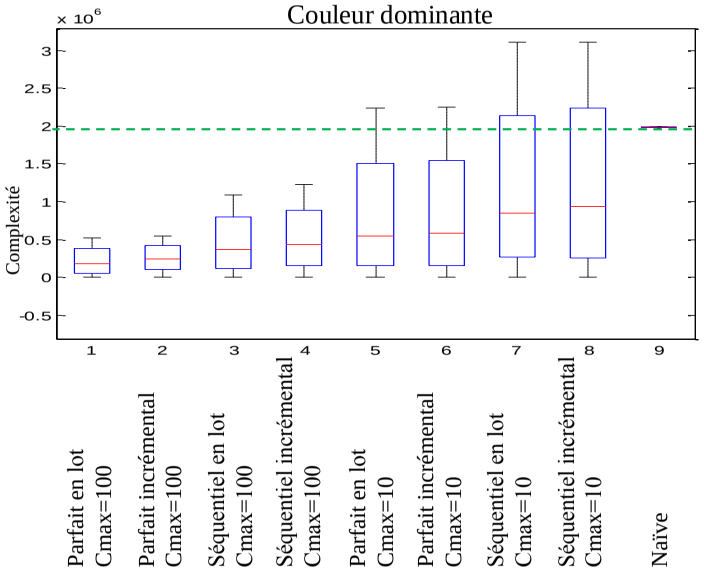


(c) « KDD 2008 »

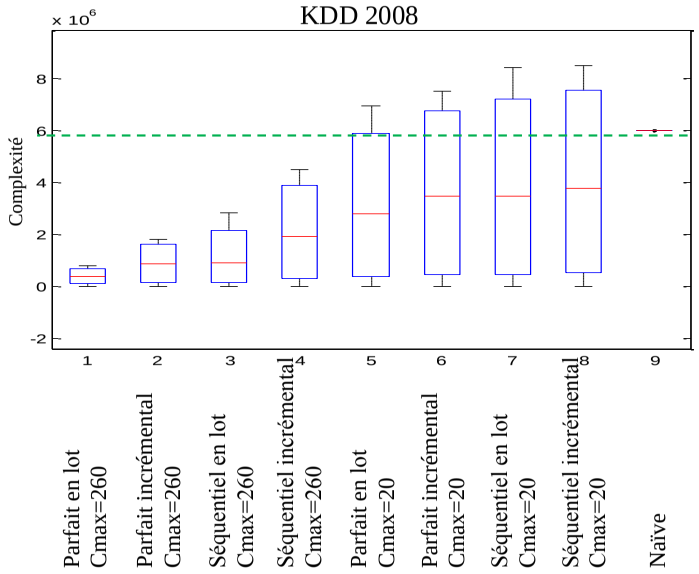
FIGURE 7.6 – Complexité par type de requête avec $k = 5$



(a) « Villes de France »



(b) « Couleurs dominantes »



(c) « KDD 2008 »

FIGURE 7.7 – Complexité par type de requête pour $k = 50$

algorithme	construction	$c_{\max}$	minimum	maximum	moyenne	variance
parfait	en lot	190	820	2 855	1 002	500
parfait	incrémental	190	820	2 920	1 985	654
séquentiel	en lot	190	1 244	4 400	3 211	1 450
séquentiel	incrémental	190	1 326	5 951	3 654	2 645
parfait	en lot	13	1 357	7 899	5 087	2 214
parfait	incrémental	13	1 369	8 950	5 547	2 310
séquentiel	en lot	13	1 380	9 240	5 858	2 498
séquentiel	incrémental	13	1 412	10 147	6 785	2 920
naïf	N/A	N/A	320 000	320 000	320 000	0

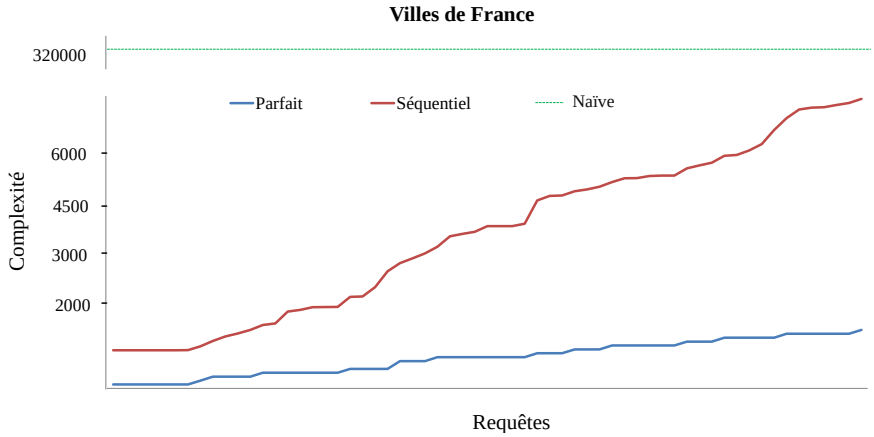
TABLE 7.18 – Complexité des requêtes sur la collection « Villes de France » avec $k = 5$

algorithme	construction	$c_{\max}$	minimum	maximum	moyenne	variance
parfait	en lot	100	500	416 598	151 200	105 000
parfait	incrémental	100	655	450 595	194 500	103 250
séquentiel	en lot	100	720	1 054 132	401 265	206 550
séquentiel	incrémental	100	880	1 123 654	452 536	205 450
parfait	en lot	10	1085	2 046 777	543 565	220 000
parfait	incrémental	10	1295	2 136 511	673 532	206 100
séquentiel	en lot	10	1805	3 004 156	1 276 126	400 755
séquentiel	incrémental	10	2145	3 004 996	1 388 215	423 712
naïf	N/A	N/A	1 960 000	1 960 000	1 960 000	0

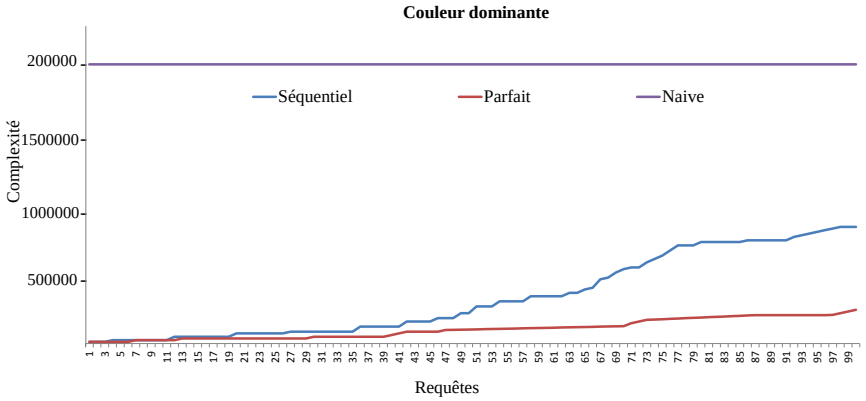
TABLE 7.19 – Complexité des requêtes sur la collection « Couleurs dominantes » avec $k = 5$

algorithme	construction	$c_{\max}$	minimum	maximum	moyenne	variance
parfait	en lot	260	655	680 000	450 785	100 100
parfait	incrémental	260	1 400	1 416 598	1 151 200	170 500
séquentiel	en lot	260	2 900	2 420 560	1 185 455	201 450
séquentiel	incrémental	260	3 680	2 500 560	1 189 822	455 400
parfait	en lot	20	3 995	6 436 511	4 052 532	601 610
parfait	incrémental	20	3 995	6 500 699	4 599 978	799 654
séquentiel	en lot	20	4 145	7 404 106	4 876 215	800 712
séquentiel	incrémental	20	4 245	7 504 106	5 276 215	900 712
naïf	N/A	N/A	6 000 000	6 000 000	6 000 000	0

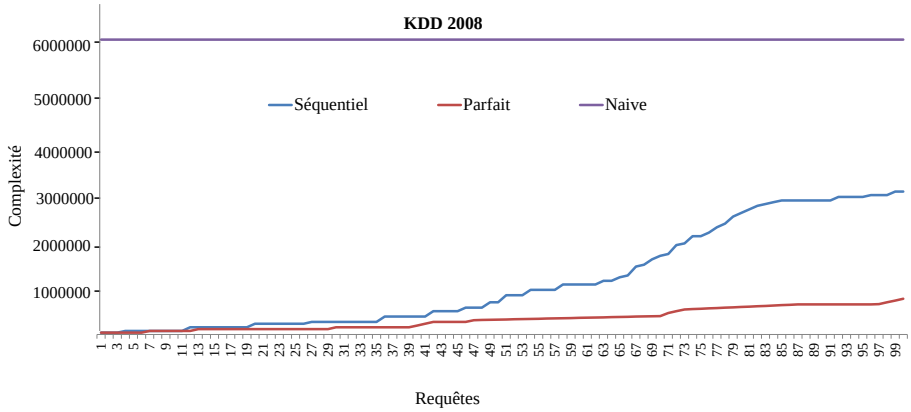
TABLE 7.20 – Complexité des requêtes sur la collection « KDD 2008 » avec $k = 5$



(a) « Villes de France »



(b) « Couleurs dominantes »



(c) « KDD 2008 »

FIGURE 7.8 – Complexité des requêtes

quons qu'un index qui a été construit en mode incrémental est moins performant qu'un index qui a été construit en lot. Cela explique l'importance de connaître à l'avance sinon la distribution de la totalité de nos objets du moins d'un sous-ensemble significatif afin de permette un bon choix des pivots (les plus éloignés) malgré le coût de construction induit.

Dans le même contexte, pour le seuil maximal du cardinal des feuilles (paramètre $c_{\max}$), nous voyons que l'algorithme de recherche séquentiel donne de meilleurs résultats avec un valeur de $c_{\max}$ égale à $\sqrt{n}$ plutôt que $\log n$. Cela semble indiquer que le choix de la paire de pivots parmi un nombre plus importants de candidats donne de meilleurs résultats.

Nous constatons aussi que, si on augmente la dimension intrinsèque, les performances des algorithmes de recherche diminuent. Par exemple, avec la collection « Villes de France », l'écart entre la version séquentielle et naïve est tellement important que nous ne pouvons pas le montrer sur les graphiques. En revanche, dans les deux autres collections (« Couleurs dominantes » et « KDD 2008 »), l'écart est minimal, ce qui traduit l'effet de la « malédiction de la dimension ».

7.4 Comparaison avec des techniques récentes

Dans cette partie, nous comparons les performances de notre approche avec les mêmes techniques d'indexation métrique utilisées en section 7.2.2 en page 127 : l'arbre-MM [80] dont notre approche est inspirée ; son extension, l'arbre-oignon ; ainsi que l'arbre-slim [80], une version améliorée de l'arbre-M.

7.4.1 Protocole des expérimentations

Nous lançons 100 requêtes différentes pour chaque collection. Pour la construction, nous choisissons les meilleures valeurs pour les deux paramètres, à savoir $\alpha = 0,526$ et $c_{\max} = \sqrt{n}$, sachant que la construction se fait en mode incrémental.

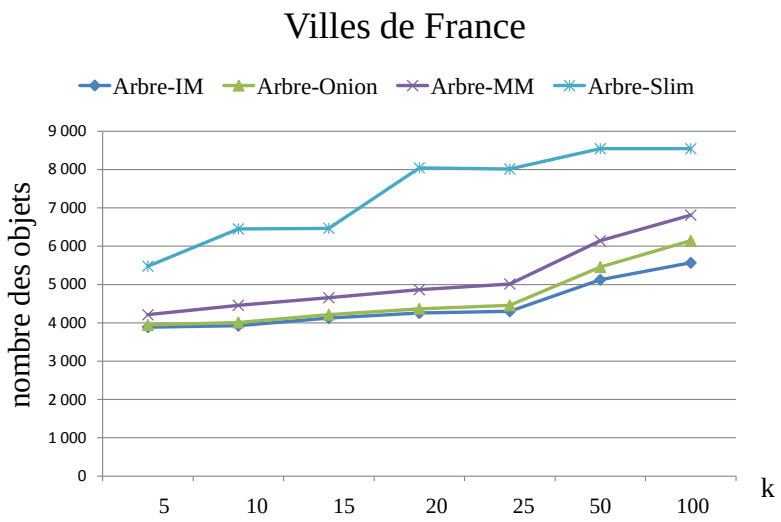
Nous faisons varier le paramètre *k* avec les valeurs 5, 10, 20, 25, 50 et 100.

Dans chacun des scénarios, nous calculons :

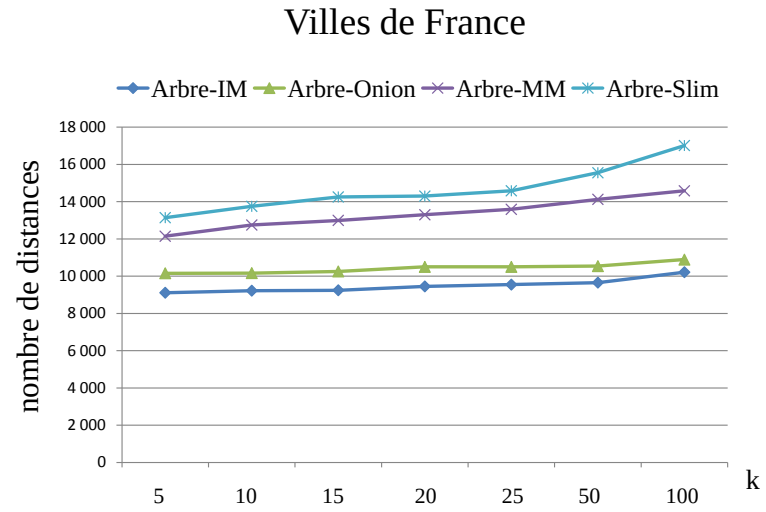
- le nombre d'objets rencontrés (n_1) ;
- le nombre de distances calculées (n_2) ;
- le nombre de comparaisons nécessaires (n_3) ;

pour la construction d'un arbre-IM.

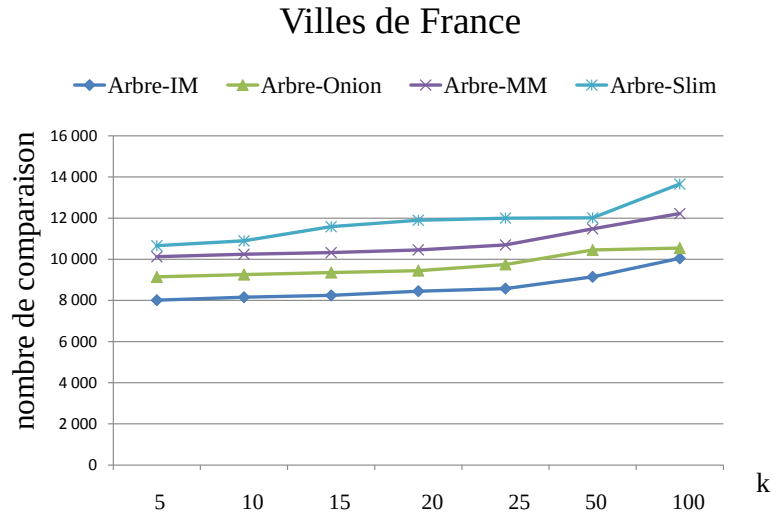
Les figures 7.9, 7.10 et 7.11 montrent le nombre d'objets, n_1, le nombre de distances, n_2, et le nombre de comparaison, n_3, effectuées lors la recherche kNN pour chaque collection.



(a) n_1

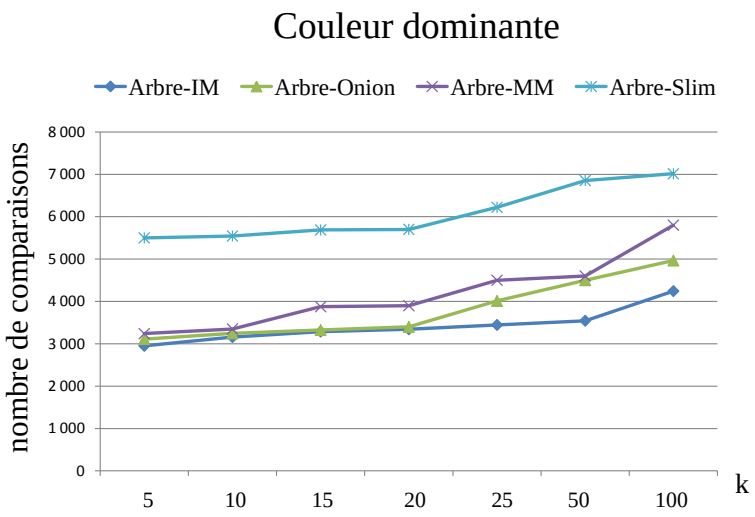


(b) n_2

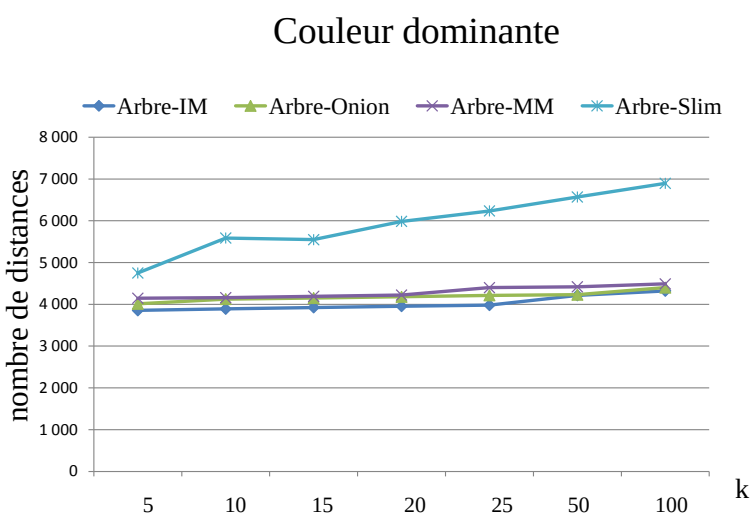


(c) n_3

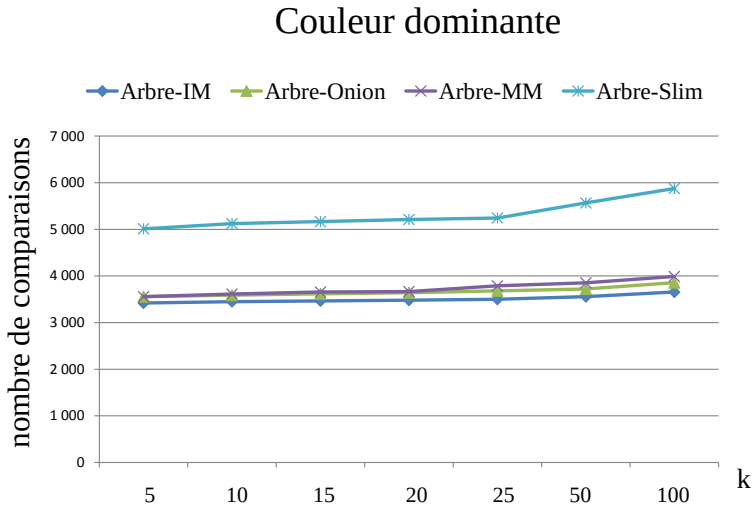
FIGURE 7.9 – Nombre d'objets parcourus, de distances calculées et de comparaisons effectuées dans une recherche kNN sur la collection « Villes de France » avec les arbres IM, oignon, MM et slim



(a) n_1

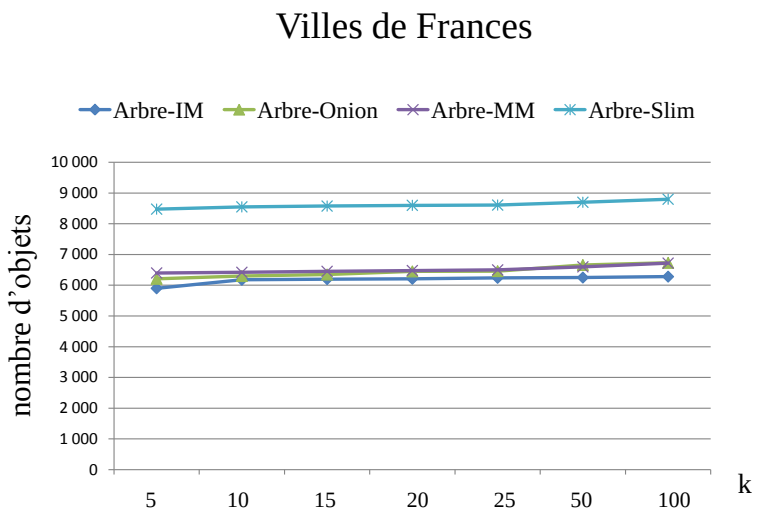


(b) n_2

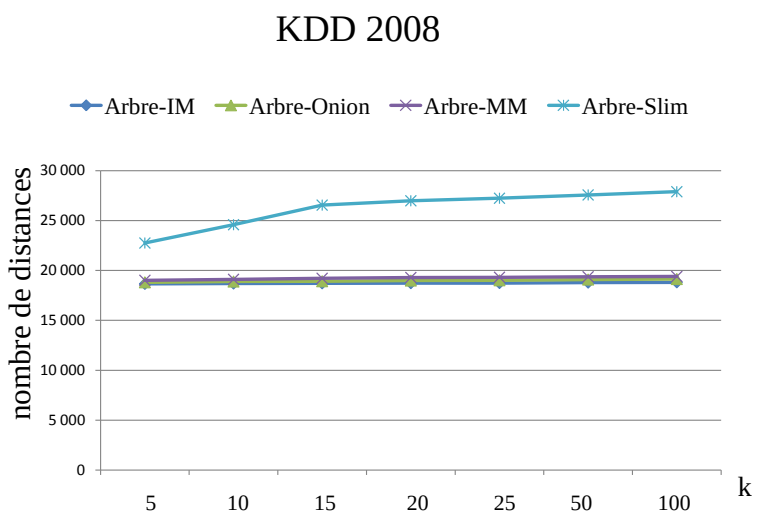


(c) n_3

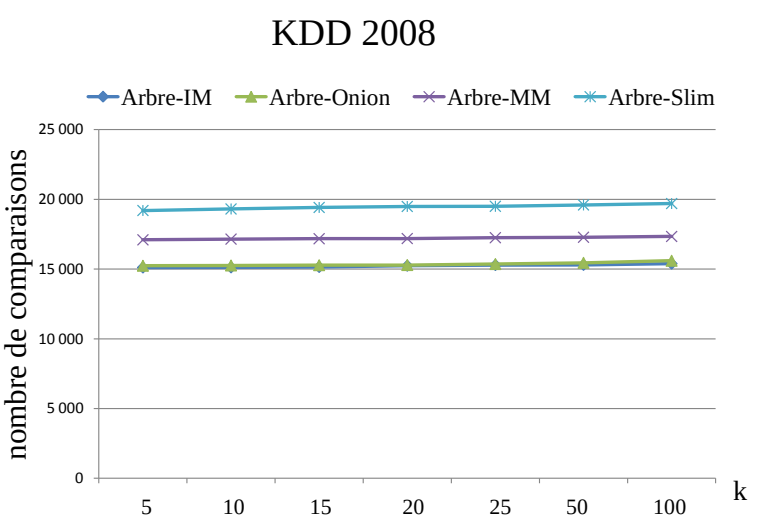
FIGURE 7.10 – Nombre d'objets parcourus, de distances calculées et de comparaisons effectuées dans une recherche kNN sur la collection « Couleurs dominantes » avec les arbres IM, oignon, MM et slim



(a) n_1



(b) n_2



(c) n_3

FIGURE 7.11 – Nombre d'objets parcourus, de distances calculées et de comparaisons effectuées dans une recherche kNN sur la collection « KDD 2008 » avec les arbres IM, oignon, MM et slim

Nous faisons les constatations suivantes :

1. Sans exception, l'augmentation de la valeur de k augmente le coût des algorithmes séquentiels.

2. Notons aussi, en comparant les méthodes entre elles, que l'arbre-slim est le plus coûteux.

3. En comparant maintenant l'arbre IM avec les arbres MM et oignon, nous constatons qu'il y a un avantage en moyenne de notre approche séquentielle par rapport à l'arbre-MM et aussi, bien qu'un peu moins, par rapport à l'arbre-oignon.

7.5 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle approche d'indexation dans les espaces métriques, une méthode dénommée « arbre-IM ».

Notre avons proposé deux algorithmes de construction et un algorithme de recherche des k plus proches voisins. Nous avons démontré avec les expérimentations que, dans la phase de construction, l'arbre-IM est plus performant par rapport à d'autres techniques récentes : arbre-MM, arbre-oignon et arbre-slim. Dans un deuxième temps, nous avons montré, aussi avec des expérimentations, que l'algorithme de recherche des k plus proches voisins dans sa version séquentielle est plus performant que les trois autres types d'index sur trois collections différentes.

Malgré cela, ce que nous pouvons retenir – et à notre avis c'est le plus important – est la détérioration des performances pour les collections présentant une dimension intrinsèque importante. C'est la célèbre « malédiction de la dimension » qui a été citée plusieurs fois par les chercheurs du domaine.

Nous avons mentionné, dès l'introduction (cf. section 1.4), qu'une solution théorique existe grâce au parallélisme. Après avoir essayé d'obtenir les meilleures performances possibles en séquentiel dans ce chapitre, nous allons paralléliser l'arbre-IM dans le chapitre suivant.

8

Parallélisation de l'arbre-IM

8.1 Introduction

Nous avons vu dans le chapitre précédent que la technique de l'arbre-IM est concurrentielle par rapport aux techniques actuelles. Mais nous avons aussi remarqué qu'en augmentant la dimension intrinsèque, l'écart avec la version naïve diminue (cf. figure 7.6). Cela nous replace sur le problème rencontré par toutes les techniques d'indexation de l'état de l'art, à savoir la « malédiction de la dimension » (cf. section 2.6.3 en page 30).

Dans ce chapitre, nous poursuivons une direction de recherche à notre avis importante pour résoudre ce problème. Il s'agit de fournir un algorithme de recherche *parallèle*.

La figure 8.1 montre une hiérarchie des différentes méthodes d'indexation. En première position, on trouve le simple balayage des objets. Après, au deuxième niveau, on se retrouve avec les techniques d'indexation séquentielles, comme les arbres GH, VP, etc. Au troisième et dernier niveau, on trouve les algorithmes parallèles, comme les arbres GH*, VP*, etc.

Nous allons donc proposer une version parallèle de l'algorithme de recherche kNN dans l'arbre-IM du chapitre précédent.

8.1.1 Objectif de la version parallèle

Répondre à une requête q dans une recherche du type « plus proches voisins » se traduit par le parcours d'une boule-requête $B(q, r_q)$ sur la structure de l'index, en calculant son intersection avec les régions rencontrées, en retenant les éléments

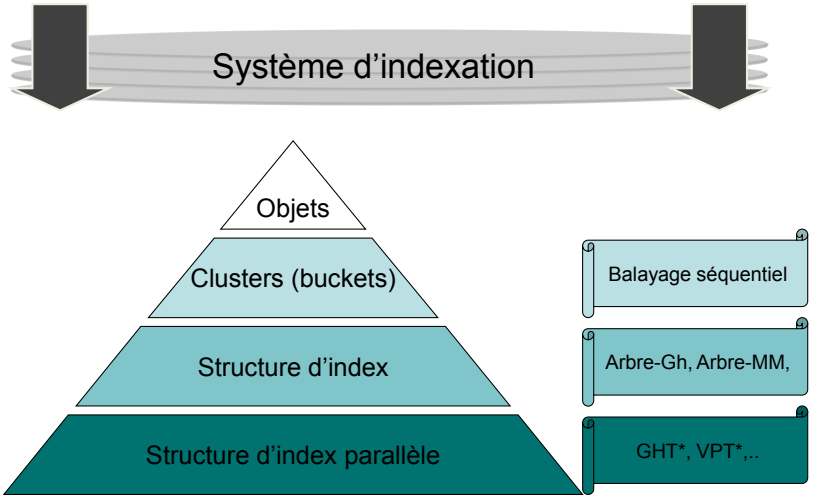


FIGURE 8.1 – Structures d'index et parallélisme (inspiré de [94])

faisant partie de la réponse sur la partie parcourue de l'index et en réduisant de manière monotone r_q de manière à élaguer au maximum le parcours de l'index. Dans la version séquentielle, l'efficience de l'algorithme de recherche est fortement liée à la diminution rapide du volume de cette boule-requête, c'est-à-dire le rayon requête r_q. Or, ce dernier est initialisé à l'infini. Mais la version séquentielle profite du transfert d'information entre branches sœurs d'un nœud afin de réduire rapidement l'espace à parcourir (réduire la taille de la boule-requête).

Cela ne pourra plus être le cas dans la version parallèle. Nous nous fixons donc comme objectif intermédiaire de trouver rapidement et grossièrement une valeur initiale à r_q, qui reste toutefois un majorant de la distance au futur k^e élément de la réponse.

En l'absence d'échange d'information entre les branches d'un même nœud, une initialisation de la valeur du rayon de requête à l'infini rend la recherche parallèle équivalente à un parcours (parallèle) – quasi – *total* de l'index (nœuds internes et feuilles). Du point de vue du travail engendré, les performances sont alors équivalentes à celle de la recherche naïve. La version parallèle va simplement plus vite, car elle dispose de plusieurs processeurs.

Ce que nous souhaitons obtenir est une version parallélo-séquentielle qui puisse bénéficier des avantages constatés sur une indexation, notamment celle de notre proposition, avec les gains attendus d'une parallélisation. Comme solution préliminaire pour résoudre ce problème, nous proposons un algorithme d'estimation de la valeur de r_q, qui va injecter cette valeur au début du processus parallèle. Cet algorithme est détaillé dans la section suivante.

Algorithme 30 Recherche kNN dans un arbre-IM, version parallèle

$$\text{kNN-IM}\left(\begin{array}{l} N \in \mathcal{N}, \\ q \in \mathcal{O}, \\ k \in \mathbb{N}^*, \\ r_q \in \mathbb{R}^+ = +\infty \end{array}\right) \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}}$$

avec :

- $d_1 = d(q, p_1)$;
- $d_2 = d(q, p_2)$;
- $C_1 = B(q, r_q) \cap B(p_1, r) \neq \emptyset \ \wedge \ B(q, r_q) \cap B(p_2, r) \neq \emptyset$;
- $C_2 = B(q, r_q) \cap B(p_1, r) \neq \emptyset \ \wedge \ B(q, r_q) \not\subseteq B(p_2, r)$;
- $C_3 = B(q, r_q) \not\subseteq B(p_1, r) \ \wedge \ B(q, r_q) \cap B(p_2, r) \neq \emptyset$;
- $C_4 = B(q, r_q) \not\subseteq B(p_1, r_1) \ \wedge \ d(p_2, q) \leq d(p_1, q)$;
- $C_5 = B(q, r_q) \not\subseteq B(p_2, r_2) \ \wedge \ d(p_1, q) > d(p_2, q)$;
- $r_{q1} = \min\{r_q, d_1 + r_1, d_2 + r_2\}$;
- $r_{q2} = \min\{r_q, d_1 + r_1\}$;
- $r_{q3} = \min\{r_q, d_2 + r_2\}$;
- $r_{q4} = \min\{r_q, d_1 + r_1\}$;
- $r_{q5} = \min\{r_q, d_2 + r_2\}$.

$$\triangleq \begin{cases} k\text{-sort}(\{(d(x,q), x) : x \in E_L\}) & \text{si } N = E_L \\ k\text{-merge}(\{\text{kNN-IM}(N_i, q, k, r_{qi}) : C_i, \forall 1 \leq i \leq 5\}) & \text{si } N = (p_1, p_2, r, r_1, r_2, N_1, N_2, N_3, N_4, N_5) \end{cases}$$

8.2 Algorithme parallèle pour une recherche kNN

Dans cette section, nous introduisons formellement une version parallèle de l'algorithme de recherche kNN dans un arbre-IM. Le parallélisme en lui-même apporte une solution satisfaisante en temps de calcul sinon en travail. Nous essayons alors d'améliorer les performances en introduisant un algorithme de pré-traitement visant à déterminer une limite supérieure du rayon de requête aussi près que possible de la valeur réelle.

8.2.1 Recherche kNN parallèle dans un arbre-IM

L'algorithme 30 formalise une recherche kNN parallèle dans le cas de l'arbre-IM que nous proposons. (Elle serait rapidement adaptable aux autres types d'arbres que nous avons vus dans l'état de l'art.) Cet algorithme fonctionne comme suit.

Les nœuds feuilles contiennent un sous-ensemble des données indexées avec

un cardinal maximal égal à $c_{\max}$. Au niveau des feuilles la procédure est assez simple. Afin de trouver les *k* plus proches voisins d'une feuille, il suffit de les trier selon leurs distances croissantes à l'objet-requête q. Puis, nous retournons *au plus* les *k* premiers éléments triés. Noter qu'un tri réel n'est pas nécessaire ; il existe une variante, appelée « *k*-tri », qui est seulement en :

$$O(c_{\max}.\log_2 k) \tag{8.1}$$

plutôt qu'en :

$$O(c_{\max}.\log_2 c_{\max}). \tag{8.2}$$

Noter que $c_{\max}$ étant soit une constante, soit un logarithme de la taille de la collection, soit sa racine carrée, la complexité de l'opération sur une feuille est très rapide, voire constante.

Dans les nœuds internes, le principe consiste à exécuter en parallèle une recherche kNN sur tous les nœuds *candidats*. Être un fils candidat dépend de l'intersection entre la boule-requête $B(q, r_q)$ et la topologie du fils. Les cinq régions sont distinctes dans leur forme (cf. figure 6.1 en page 105). Par conséquent, nous devons fournir cinq conditions distinctes, C_1 à C_5 dans l'algorithme 30 :

1. $C_1 = B(q, r_q) \cap B(p_1, r) \neq \emptyset \ \wedge \ B(q, r_q) \cap B(p_2, r) \neq \emptyset$, pour l'intersection ;
2. $C_2 = B(q, r_q) \cap B(p_1, r) \neq \emptyset \ \wedge \ B(q, r_q) \not\subseteq B(p_2, r)$, pour la boule partielle centrée sur p_1 ;
3. $C_3 = B(q, r_q) \not\subseteq B(p_1, r) \ \wedge \ B(q, r_q) \cap B(p_2, r) \neq \emptyset$, pour la boule partielle centrée sur p_2 ;
4. $C_4 = B(q, r_q) \not\subseteq B(p_1, r_1) \ \wedge \ d(p_2, q) \leq d(p_1, q)$, pour le reste de l'espace situé dans le demi-plan associé à p_1 ;
5. $C_5 = B(q, r_q) \not\subseteq B(p_2, r_2) \ \wedge \ d(p_1, q) > d(p_2, q)$, pour le reste de l'espace situé dans le demi-plan associé à p_2.

Le rayon de requête r_q joue le rôle essentiel pour l'optimisation de recherche (le minimum possible vaut un maximum d'élagage). Il est initialement fixé à $+\infty$ par défaut, mais nous *espérons* le voir diminuer à chaque passage sur un nœud interne. Cela, à nouveau, dépend du nœud fils considéré, plus précisément de sa topologie. Comme les cinq conditions précédentes, nous avons donc cinq variantes de la réduction de la valeur de r_q :

1. $r_{q1} = \min\{r_q, d_1 + r_1, d_2 + r_2\}$;
2. $r_{q2} = \min\{r_q, d_1 + r_1\}$;

3. $r_{q3} = \min\{r_q, d_2 + r_2\}$;
4. $r_{q4} = \min\{r_q, d_1 + r_1\}$;
5. $r_{q5} = \min\{r_q, d_2 + r_2\}$;

où $d_1 = d(q, p_1)$ et $d_2 = d(q, p_2)$. Cette étape pourrait être encore améliorée en tenant compte des cardinaux de chaque fils. Notons toutefois que plus l'estimation initiale de r_q sera proche de sa valeur finale, moins cette étape sera utile.

Les résultats de zéro à cinq fils sont fusionnés et au plus *k* d'entre eux forment la réponse.

Notons, encore une fois, que cette étape ne nécessite pas vraiment un tri, mais seulement une séquence de fusions. La complexité est « constante », c'est-à-dire en :

$$O(5.k) \tag{8.3}$$

plutôt qu'en :

$$O(5.k.\log_2 k). \tag{8.4}$$

8.2.2 Complexité parallèle

Déterminons la complexité de cet algorithme :

- Dans le pire des cas, le calcul global s'effectue de toutes les feuilles vers la racine de l'arbre ; c'est une recherche parallèle *complète*.
- Dans le meilleur des cas, seules les feuilles qui contiennent au moins une réponse participent au calcul ascendant ; c'est la recherche parallèle *parfaite*.

Plaçons-nous dans le cas où $c_{\max} = \sqrt{n}$. La complexité moyenne globale en temps peut être estimée en :

$$O\left(\frac{1}{2}\sqrt{n}.\log_2 k + \left(\log_5 \frac{n}{\frac{1}{2}\sqrt{n}}\right).5.k\right) \tag{8.5}$$

avec $n = |E|$ où :

- le premier terme correspond aux calculs effectués sur une unique feuille, tous les calculs sur les feuilles étant réalisés en parallèle, où le premier facteur – $\frac{1}{2}\sqrt{n}$ – estime que les feuilles sont à moitié remplies en moyenne ;
- le deuxième terme correspond à la séquence de calculs effectués en montant dans l'index, c'est-à-dire en multipliant la hauteur de l'arbre – $\log_5 \frac{n}{\frac{1}{2}\sqrt{n}}$ – par le temps de fusion – $5.k$ – dans les nœuds internes le long d'un unique chemin.

Si l'on considère que *k* est une « constante », dans le sens où sa valeur est indépendante de la taille de la collection et sans doute bien plus petite que $\sqrt{n}$ dans la pratique, alors la complexité se ramène à :

$$O(\sqrt{n}) \tag{8.6}$$

c'est-à-dire le temps de calcul séquentiel sur une unique feuille, totalement remplie.

Il s'agit bien évidemment d'une amélioration par rapport à un « *k*-tri » séquentiel, en $O(n.\log_2 k)$. Elle reste moins efficace qu'un *k*-tri parallélisé, donc logarithmique, mais ne nécessite « que » $O(\sqrt{n})$ processeurs, dans le pire des cas, pour être exécutée à pleine vitesse, au lieu de $O(n)$. Il s'agit donc d'un compromis conservant l'optimalité de la parallélisation.

Dans le meilleur des cas, la complexité en temps ne change pas ; c'est seulement le nombre de processeurs qui est réduit, pouvant se limiter à un seul dans le cas idéal où une unique feuille contient toutes les réponses.

Nous pouvons refaire l'exercice en posant maintenant $c_{\max} = \log n$. La complexité en temps peut alors être estimée en :

$$O\left(\frac{1}{2}\log n.\log_2 k + \left(\log_5 \frac{n}{\frac{1}{2}\log n}\right).5.k\right). \tag{8.7}$$

Face à une fonction logarithmique, il est plus difficile de considérer *k* comme une « constante » dans tous les cas pratiques (ex. : $\log_5 10^6 \simeq 9$ alors que $\sqrt{10^6} = 1\,000$). Une simplification moins drastique donne donc une complexité en :

$$O(k.\log n) \tag{8.8}$$

plus efficiente encore mais sous réserve de disposer de jusqu'à $O\left(\frac{n}{\log n}\right)$ processeurs...

Ici aussi, la complexité en temps en change pas dans le meilleur des cas, seul le nombre de processeurs se voit réduit jusqu'à éventuellement n'en nécessiter qu'un seul.

Toutefois, ces évaluations ne tiennent que sous les hypothèses que :

1. l'arbre est (quasi) équilibré ;
2. la distribution des éléments dans les feuilles est uniforme ;
3. la répartition des éléments entre les différents enfants est également uniforme (pas de grandes différences entre les cardinaux des feuilles).

Ces dernières ont été vérifiées expérimentalement. Il ne reste donc plus qu'à chercher à diminuer le nombre de processeurs nécessaires pour mener au mieux

Algorithme 31 Estimation du rayon de la requête pour la version parallèle

limite $\begin{pmatrix} N \in \mathcal{N}, \\ q \in \mathcal{O}, \\ \beta \in \mathbb{N}, \\ r_q \in \mathbb{R}^+ = +\infty \end{pmatrix} \in \mathbb{R}^+ \times \mathbb{N}$

avec les mêmes conditions C_i et majorants r_{qi} que dans l'algorithme 30.

si $\beta = 0$ **or** $N = E_L$ **alors**
  renvoyer (r_q, β)
sinon
  $\beta \leftarrow \beta - 1$
  pour $i \in \{1, \ldots, 5\}$ **faire**
    si C_i **alors**
      $(r_q, \beta) \leftarrow$ limite$(N_i, q, \beta, \min\{r_q, r_{qi}\})$
    fin si
  fin pour
  renvoyer (r_q, β)
fin si

les calculs. C'est ici que l'estimation la plus fine possible du rayon initial de la requête intervient.

8.2.3 Estimation du rayon de requête initial

S'appuyer uniquement sur les diminutions successives du rayon de la requête exprimées par les calculs de r_{q1} à r_{q5} dans l'algorithme 30 s'avère insuffisant.

Il est nécessaire d'estimer un majorant à la future k^e distance. De cette manière, le premier appel sur le nœud racine pourra être initialisé avec une valeur beaucoup plus satisfaisante que l'infini, la meilleure estimation conduisant à une recherche « *parfaite* ».

C'est le rôle d'un algorithme compagnon, l'algorithme 31. Il recherche une limite supérieure « minimale » par balayage d'un nombre limité, β, de nœuds internes. Les nœuds feuilles sont considérés comme trop coûteux pour être balayés.

Cette proposition heuristique devrait être mise en concurrence avec une version aléatoire, c'est-à-dire une initialisation de la recherche après avoir choisi *k* objets différents au hasard dans l'index. C'est cette approche-là qui est utilisée par l'arbre-SR (cf. section 3.1.2 en page 43).

8.3 Conclusion

La proposition d'un algorithme parallélisé pour les recherches kNN dans un arbre-IM se révèle assez directe. Elle offre un potentiel de performances optimales mais

se heurte à la limite en nombre de processeurs réellement disponibles et, de manière plus générale, que l'on souhaite affecter à cette tâche. Par conséquent, nous avons introduit un algorithme, heuristique mais non aléatoire, permettant de fournir une estimation, la moins grossière possible de préférence, de la distance à la future k^e réponse. Sans garantie d'aucune sorte à apporter ici, il nous faut encore une fois en recourir aux expérimentations pour juger du bien fondé de la proposition.

9

Expérimentations et résultats : approche parallèle

Rappelons que l'objectif de l'algorithme de recherche en parallèle est de répondre *efficacement* aux requêtes kNN.

Nous arrivons à la phase d'expérimentation de l'algorithme de recherche kNN dans sa version parallèle. Comme dans la version séquentielle, avant de commencer les expérimentations, nous décrivons tout d'abord le protocole. Ensuite, les résultats bruts sont présentés sous forme de tableaux et graphiques. Enfin, nous en tirons des conclusions.

9.1 Protocole des expérimentations

Les éléments du protocole concernent pour partie les éléments introduits dans le chapitre précédent – collections et paramètres de l'arbre – et pour partie des éléments liés à la recherche kNN – k et algorithmes. De plus, nous précisons quelles mesures sont effectuées.

Collections et paramètres de construction de l'arbre-IM. Nous retrouvons ici les mêmes paramètres que dans le protocole d'expérimentation des algorithmes de construction des arbres (cf. section 7.2.1). Pour mémoire, nous utilisons trois collections de complexités intrinsèques différentes : « Villes de France », « Couleurs dominantes » et « KDD 2008 ».

Par ailleurs, nous effectuons les recherches sur les index construits à partir des versions incrémentales et en lot, en faisant varier les paramètres α (0,52, 0,55, 0,6

et 0,69) et $c_{\max}$ ($\sqrt{n}$ et $\log n$).

Taille du résultat. Un premier paramètre supplémentaire est *k*. Nous effectuons des recherches avec $k \in \{5, 10, 15, 20, 50, 100\}$.

Estimation de r_q. Un paramètre supplémentaire, β, est utilisé par l'algorithme d'estimation du rayon de requête initial. Il varie entre 10, 20 et 30 %. Cela représente le pourcentage de nœuds internes qui vont être parcourus. Il est ensuite traduit en nombre de nœuds internes effectifs.

Variante des algorithmes de recherche kNN. Pour les algorithmes de recherche, nous utilisons les cinq variantes algorithmiques suivantes :

- *Parallèle-complet.* C'est la recherche kNN parallèle de l'algorithme 30 où r_q est initialisé à $+\infty$.

  Cela conduit à un balayage – quasiment – complet de l'ensemble des données, mais en parallèle. Il sert donc comme étalon pour la comparaison entre les différents algorithmes. Il est supposé être un majorant, même si une recherche *via* un arbre pourrait présenter quelques cas aberrants au-dessus de cette limite.

- *Séquentiel.* C'est la version séquentielle kNN du chapitre 7 (cf. algorithme 28 en page 113).

- *Parallèle-borné.* La dernière variante est l'algorithme de recherche parallèle où r_q est initialisé avec une *estimation* de la distance au k^e plus proche voisin, celle déterminée par l'algorithme 31.

  En outre, à la fin des expériences, nous montrerons l'effet du paramètre β sur la version parallèle bornée, plus précisément sur l'algorithme compagnon, avec $\beta = 10, 20, 30$ % du nombre total des nœuds internes.

- *Parfait.* C'est un algorithme de recherche très particulier, car nous lançons une recherche parallèle avec r_q initialisé avec la distance exacte au (futur) k^e objet !

  Il s'agit donc du cas idéal qui permet une comparaison avec les autres approches. C'est le minorant pour la structure d'index.

- *Naïf.* L'algorithme parallèle naïf est un tri parallèle, dont on ne retiendra que les *k* premières valeurs. Cela donne un algorithme dont le temps d'exécution est optimal, en :

$$O(\log d \times \log n) \tag{9.1}$$

	nœuds				
algorithme	feuilles (%)	internes (%)	#comparaison (n_3)	#distances (n_2)	#objets (n_1)
naïf	N/A	N/A	35 183	35 183	35 183
parallèle borné	7,23	10,80	10 645	10 214	4 890
séquentiel	6,75	8,16	9 111	8 011	3 888
parfait	2,22	4,16	2 958	2 322	1 500

TABLE 9.1 – Statistiques de performance de quatre algorithmes de recherche kNN dans un arbre-IM « Villes de France » avec $n = 35\,183$, $c_{\max} = \sqrt{n}$ et $k = 5$

où d représente ici la complexité séquentielle du calcul d'une distance. Mais il nécessite :

$$O(d.n) \tag{9.2}$$

processeurs !

Tout comme la version parallèle « parfaite » ci-dessous, il sert comme étalon pour la comparaison en fournissant une limite inférieure.

Mesures. Pour une meilleure interprétation des résultats pour nos expériences, nous rappelons que nous avons exécuté une centaine de requêtes kNN différentes sur les collections, et avons calculé la moyenne des résultats.

Comme dans le chapitre 7, nous mesurons d'une part :

(i) le pourcentage moyen de feuilles visitées ;

(ii) le pourcentage moyen de nœuds internes visités ;

(iii) le nombre moyen de distances calculées ;

(iv) le nombre moyen de comparaisons ;

(v) le nombre moyen d'objets visités ;

et, d'autre part, nous calculons la complexité des algorithmes de recherche dans chacun des cas.

9.2 Résultats et analyse des expérimentations

Premièrement, dans les tableaux 9.1, 9.2 et 9.3, nous faisons varier les types d'algorithmes de recherche naïf, parallèle borné, séquentiel, et le parfait, tout en gardant à chaque fois la meilleure valeur de α (trouvée dans la phase de construction) et en affectant au paramètre $c_{\max}$ la valeur $\sqrt{n}$ et $k = 5$ pour toutes les collections.

Sans surprise, la version naïve est la plus chère, et la version parfaite est la plus efficace. Mais nous pouvons voir que la version parallèle bornée est tout à fait compétitive par rapport à l'approche séquentielle. Bien qu'elle ne puisse pas

	nœuds				
algorithme	feuilles (%)	internes (%)	#comparaison (n_3)	#distances (n_2)	#objets (n_1)
naïf	N/A	N/A	10 000	10 000	10 000
parallèle borné	8,40	11,56	5 904	4 985	3 861
séquentiel	7,65	9,36	3 854	3 420	2 951
parfait	2,42	5,22	1 298	960	1 100

TABLE 9.2 – Statistiques de performance de quatre algorithmes de recherche kNN dans un arbre-IM « Couleurs dominantes » avec $n = 10\,000$, $c_{\max} = \sqrt{n}$ et $k = 5$

	nœuds				
algorithme	feuilles (%)	internes (%)	#comparaison (n_3)	#distances (n_2)	#objets (n_1)
naïf	N/A	N/A	68 025	68 025	68 025
parallèle borné	10,20	13,64	19 963	20 951	12 945
séquentiel	8,00	10,14	18 654	15 111	5 900
parfait	2,42	5,22	10 369	8 987	2 388

TABLE 9.3 – Statistiques de performance de quatre algorithmes de recherche kNN dans un arbre-IM « KDD 2008 » avec $n = 68\,025$, $c_{\max} = \sqrt{n}$ et $k = 5$

recueillir autant d'informations que son homologue séquentielle, nous voyons que la version parallèle peut être compétitive grâce à l'algorithme compagnon 31 qui estime une limite supérieure pour r_q au début de la recherche.

Toutefois, cela doit être pris avec précaution ; il s'agit de la *meilleure* estimation de la borne supérieure par l'algorithme compagnon (cf. algorithme 31).

Deuxièmement, nous faisons varier les algorithmes de recherche, mais cette fois-ci nous utilisons les deux modes de construction ainsi que les deux paramètres : α et $c_{\max}$.

Du point de vue de la complexité, elle sera calculée de la même façon que dans le protocole expérimental de la version séquentielle.

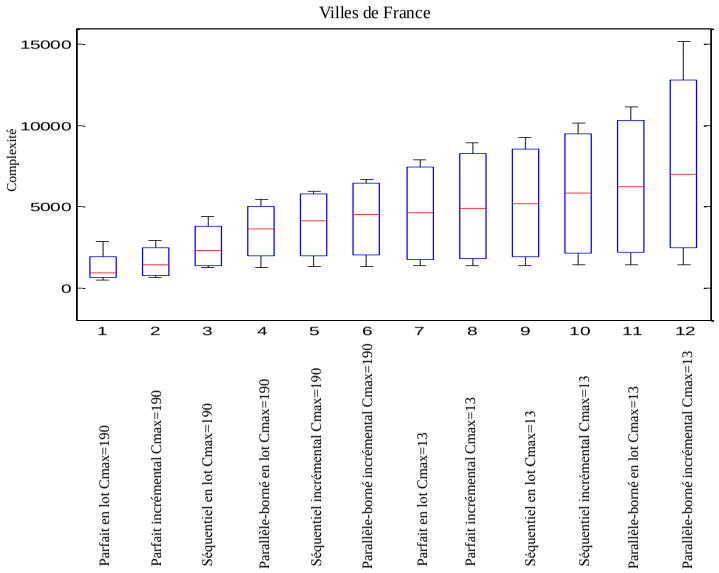
Nous traçons après des « boîtes à moustaches » (afin de mieux visualiser les courbes traditionnelles) pour chaque type de variation. Nous présentons dans les figures 9.1, 9.2 les statistiques détaillées du comportement des algorithmes de recherche, en utilisant les trois collections.

Ces résultats sont aussi représentés en détail par les tableaux 9.4, 9.5 et 9.6 pour les trois collections « Villes de France », « Couleurs dominantes » et « KDD 2008 ».

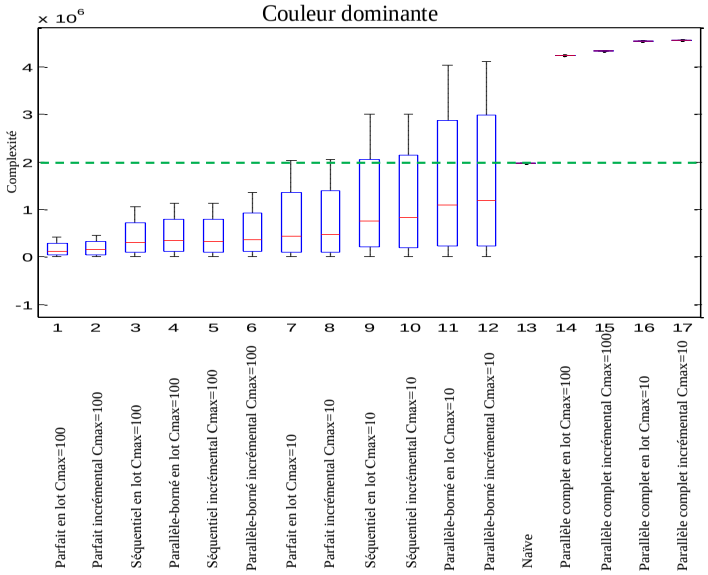
Pour l'algorithme naïf, c'est presque une constante ; effectivement, sa courbe apparaît comme une ligne horizontale sur les différentes figures.

Dans la figure 9.3, nous présentons l'évolution du rayon de requêtes.

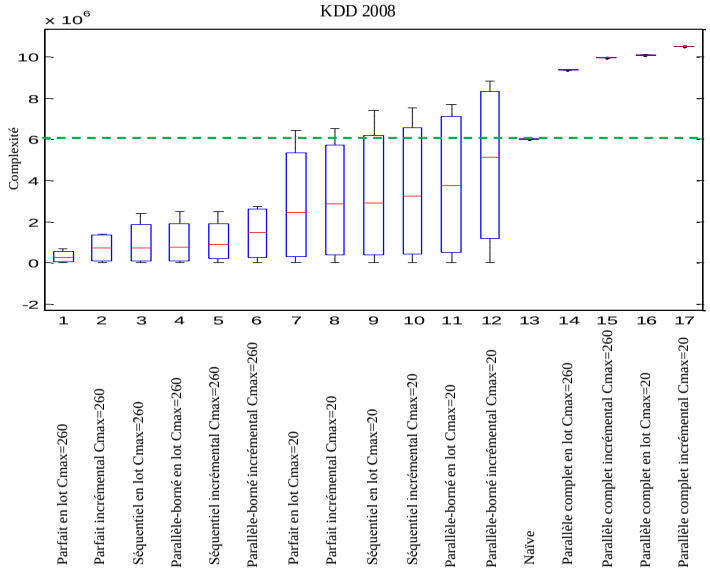
- Dans la section précédente, au début de l'analyse des résultats, nous avons montré que la structure de l'index joue un rôle important sur les performances de recherche. Cela est vraiment le cas dans ces expérimentations. Nous voyons, avec les boîtes à moustache, qu'à chaque fois, et avec toutes les variations, la construction en lot est toujours meilleure que la version in-



(a) « Villes de France »

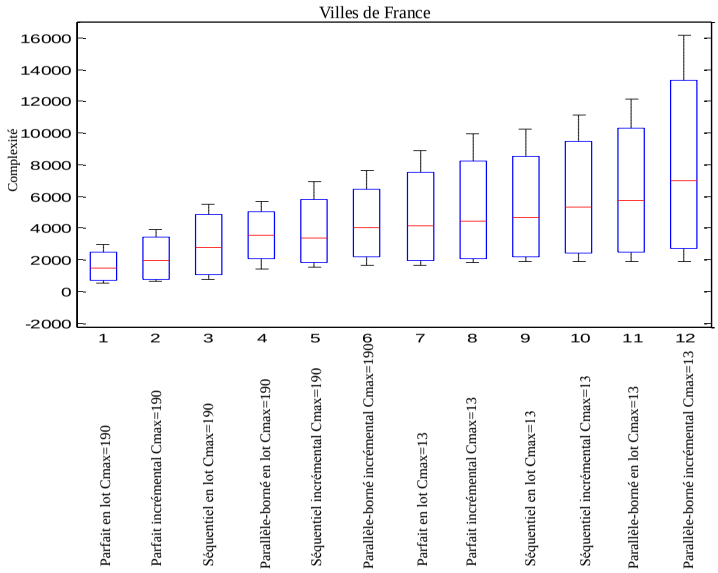


(b) « Couleurs dominantes »

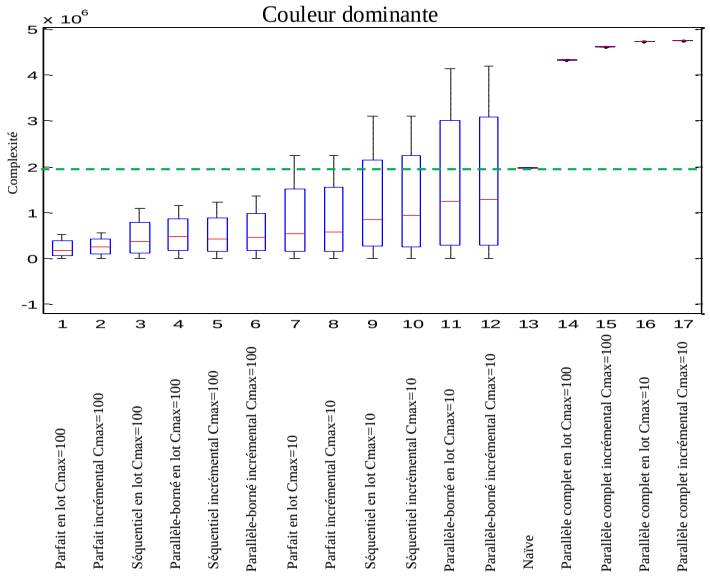


(c) « KDD 2008 »

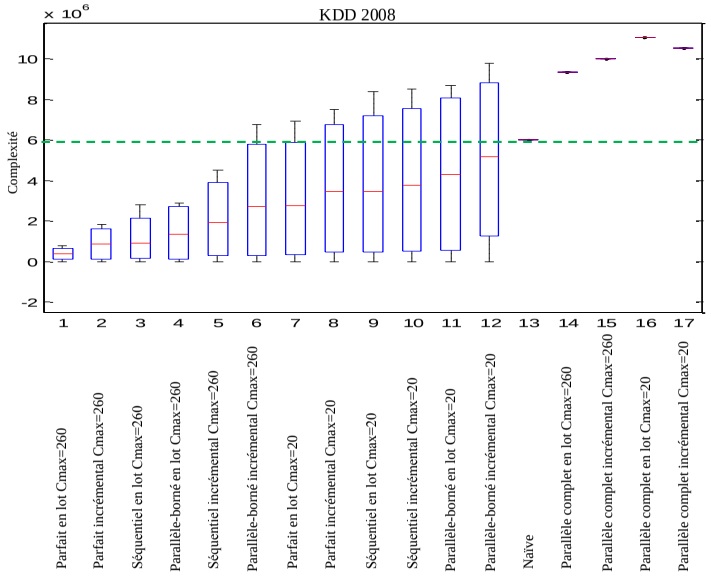
FIGURE 9.1 – Complexité par type de requête avec $k = 5$



(a) « Villes de France »



(b) « Couleurs dominantes »



(c) « KDD 2008 »

FIGURE 9.2 – Complexité par type de requête avec $k = 50$

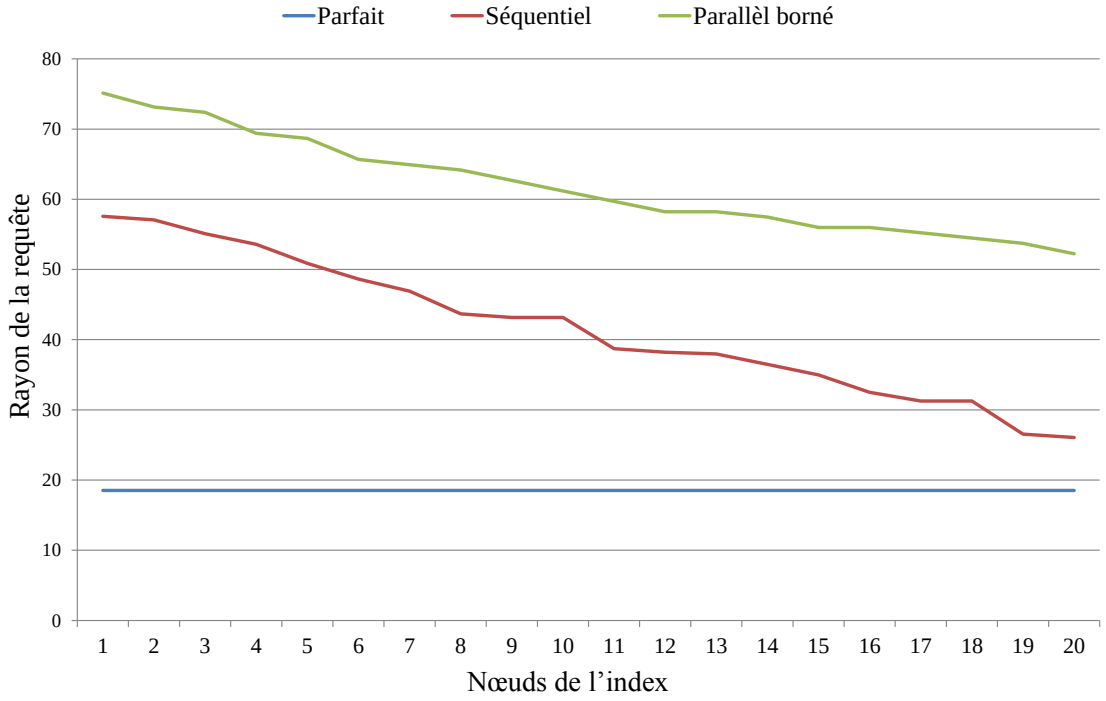


FIGURE 9.3 – Évolution de la valeur du rayon de requêtes

crémentale. Cela illustre encore une fois que la structure de l'index et la façon de le construire influence directement les performances des algorithmes de recherche.

- Sans aucune surprise, la version parallèle complète est la plus coûteuse sous toutes les conditions. Dans ce sens-là, la version naïve devient utile. Une version complète n'est qu'un parcours total des objets avec un parcours total des nœuds internes ce qui explique son coût important.

- Remarquons aussi que l'écart entre la version parfaite, qui est la plus efficace, et les versions séquentielles n'est pas négligeable. Cela nous permet de dire qu'il y a une possibilité de minimiser cet écart en utilisant des techniques pour trouver le plus vite possible le plus proche voisin avec un minimum d'énergie.

- Après on trouve en concurrence les versions séquentielles et la version parallèle bornée. Cette dernière est notre objectif pour résoudre notre problème initial. Cette concurrence apparaît surtout sous certaines conditions. Par exemple, une valeur importante de $c_{\max}$, égale à 100, est meilleure que 10 sur la collection « Couleurs dominantes » : le seuil maximal des feuilles. On constate qu'à chaque fois que cette valeur de $c_{\max}$ augmente, le coût l'algorithme de recherche séquentiel devient plus important. Cela prouve que le

algorithme	construction	$c_{\max}$	minimum	maximum	moyenne	variance
parfait	en lot	190	820	2 855	1 002	500
parfait	incrémental	190	820	2 920	1 985	654
séquentiel	en lot	190	1 244	4 400	3 211	1 450
parallèle borné	en lot	190	1 248	4 560	3 455	2 660
séquentiel	incrémental	190	1 326	5 951	3 654	2 645
parallèle borné	incrémental	190	1 348	6 665	4 263	2 740
parfait	en lot	13	1 357	7 899	5 087	2 214
parfait	incrémental	13	1 369	8 950	5 547	2 310
séquentiel	en lot	13	1 380	9 240	5 858	2 498
séquentiel	incrémental	13	1 412	10 147	6 785	2 920
parallèle borné	en lot	13	1 419	11 152	7 463	3 000
parallèle borné	incrémental	13	1 422	15 151	9 457	3 490
naïf	N/A	N/A	320 000	320 000	320 000	000 000
parallèle complet	en lot	190	1 954 851	1 954 851	1 954 851	000 000
parallèle complet	incrémental	190	2 365 198	2 365 198	2 365 198	000 000
parallèle complet	en lot	13	2 578 545	2 578 545	2 578 545	000 000
parallèle complet	incrémental	13	2 578 911	9 578 911	2 578 911	000 000

TABLE 9.4 – Complexité des requêtes sur la collection « Villes de France » avec $k = 5$

choix d'une bonne combinaison des pivots influence directement la structure de l'index et après, bien évidemment, les performances des recherches.

- Enfin, concernant le facteur *k*, nous avons traité deux cas. Dans ces deux cas nous avons remarqué que, si nous augmentons la valeur de k, les performances diminuent mais avec un écart compris entre moins de 1 % et jusqu'à moins de 2 % quand $k = 50$. Donc la valeur de k n'a pas d'influence majeure sur les performance des algorithme de recherche.

Pour plus de détails, les tableaux 9.4 et 9.5 et 9.6 montrent les chiffres exacts des performances des algorithmes de recherche.

Nous rappelons que nous avons exécuté une centaine de requêtes différentes, à la recherche à chaque fois de 5 ou 50 plus proches voisins (cf. figure 9.4).

9.2.1 Effet de l'algorithme d'estimation du rayon de requête

L'algorithme d'estimation du rayon de couverture (cf. algorithme 31) est un algorithme qui a prouvé son efficacité au vu des résultats précédents.

Pour mieux maîtriser ce paramètre, nous avons fait varier la valeur de β, qui est le pourcentage des nœuds internes visités par l'algorithme de l'estimation, entre 10, 20 et 30 %. Nous avons mis comme valeur maximale, le tiers du nombre total des nœuds internes (en raison de la complexité totale de l'algorithme de recherche). Le tableau 9.7 présente les résultats détaillés. On note une amélioration importante en augmentant le nombre des nœuds visités par l'algorithme 31 d'estimation avant le lancement de l'algorithme de recherche kNN.

algorithme	construction	$c_{\max}$	minimum	maximum	moyenne	variance
parfait	en lot	100	500	416 598	151 200	105 000
parfait	incrémental	100	655	450 595	194 500	103 250
séquentiel	en lot	100	720	1 054 132	401 265	206 550
parallèle borné	en lot	100	830	1 123 200	450 000	250 330
séquentiel	incrémental	100	880	1 123 654	452 536	205 450
parallèle borné	incrémental	100	888	1 350 000	510 000	224 320
parfait	en lot	10	1085	2 046 777	543 565	220 000
parfait	incrémental	10	1295	2 136 511	673 532	206 100
séquentiel	en lot	10	1805	3 004 156	1 276 126	400 755
séquentiel	incrémental	10	2145	3 004 996	1 388 215	423 712
parallèle borné	en lot	10	2255	4 036 500	1 696 700	475 654
parallèle borné	incrémental	10	2344	4 100 000	1 881 180	480 010
naïf	N/A	N/A	1 960 000	1 960 000	1 960 000	0
parallèle complet	en lot	100	4 236 545	4 236 545	4 236 545	0
parallèle complet	incrémental	100	4 325 985	4 325 985	4 325 985	0
parallèle complet	en lot	10	4 532 569	4 532 569	4 532 569	0
parallèle complet	incrémental	10	4 549 321	4 549 321	4 549 321	0

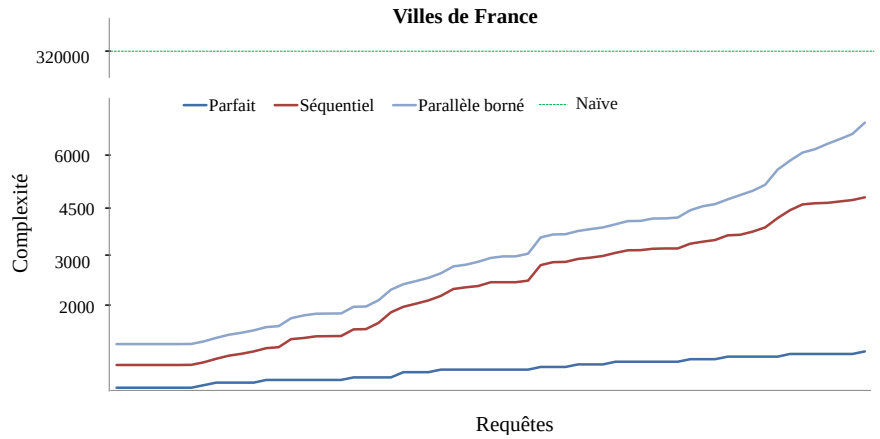
TABLE 9.5 – Complexité des requêtes sur la collection « Couleurs dominantes » avec $k = 5$

algorithme	construction	$c_{\max}$	minimum	maximum	moyenne	variance
parfait	en lot	260	655	680 000	450 785	100 100
parfait	incrémental	260	1 400	1 416 598	1 151 200	170 500
séquentiel	en lot	260	2 900	2 420 560	1 185 455	201 450
parallèle borné	en lot	260	3 200	2 500 000	1 155 263	201 450
séquentiel	incrémental	260	3 680	2 500 560	1 189 822	455 400
parallèle borné	incrémental	260	3 720	2 754 132	1 901 265	501 450
parfait	en lot	20	1 995	6 436 511	4 052 532	601 610
parfait	incrémental	20	1 995	6 500 699	4 599 978	799 654
séquentiel	en lot	20	4 145	7 404 106	4 876 215	800 712
séquentiel	incrémental	20	4 245	7 504 106	5 276 215	900 712
parallèle borné	en lot	20	4 345	7 704 106	6 276 215	1 000 712
parallèle borné	incrémental	20	3 445	8 804 106	7 276 215	2 400 712
naïf	N/A	N/A	6 000 000	6 000 000	6 000 000	0
parallèle complet	en lot	260	9 365 198	9 365 198	9 365 198	0
parallèle complet	incrémental	260	9 954 851	9 954 851	9 954 851	0
parallèle complet	en lot	20	10 078 911	10 078 911	10 078 911	0
parallèle complet	incrémental	20	10 521 444	10 521 444	10 521 444	0

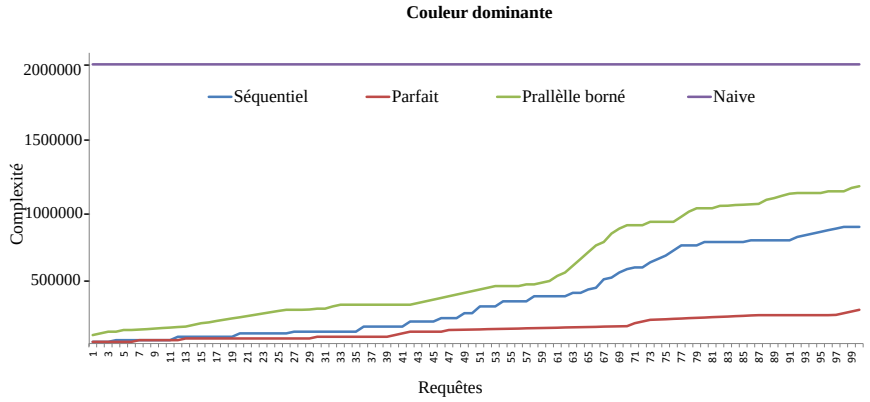
TABLE 9.6 – Complexité des requêtes pour Histogramme de couleur KDD, avec $k = 5$

β (%)	Feuilles (%)	Nœuds internes (%)	#distances	#comparaison	#objets
10	18,00	40,11	35 987	6 866	8 411
20	10,25	30,25	32 698	6 236	6 123
30	7,65	9,36	3 854	3 420	2 951

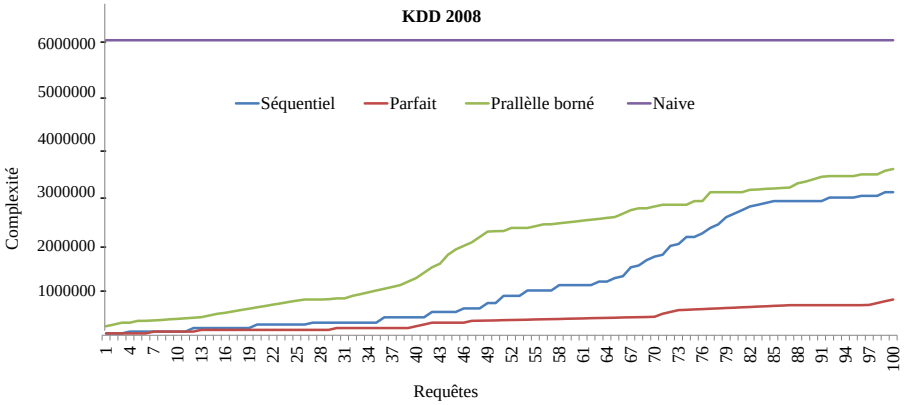
TABLE 9.7 – Statistiques de performance de la variante parallèle bornée pour différentes valeurs de β



(a) « Villes de France »



(b) « Couleurs dominantes »



(c) « KDD 2008 »

FIGURE 9.4 – Complexité des requêtes

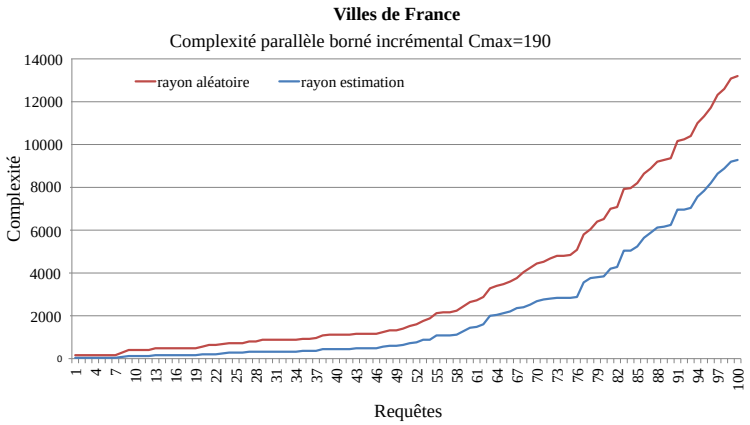


FIGURE 9.5 – Effet de l'algorithme de l'estimation du rq dans la collection « Villes de France » avec la version incrémentale et $c_{\max} = 190$, $\beta = 80$

Une autre étude a été faite dans le même sens. Nous avons lancé notre algorithme parallèle borné avec *k* objets aléatoires sans utiliser l'algorithme de l'estimation habituelle, et nous avons comparé les résultats (complexité) sur la collection « Villes de France » avec une construction incrémentale et une taille des feuille $c_{\max} = 190$. La figure 9.5 montre la différence entre un lancement avec une estimation réelle et une estimation aléatoire. Dans la première version, avec une estimation de r_q, nous avons utilisé une limite calculée à partir des premiers 30 % des nœuds internes de l'index. Dans la deuxième version, nous avons lancé l'algorithme de recherche avec un choix aléatoire des k plus proche voisins.

Nous remarquons que la version avec une estimation réelle de r_q a un effet bénéfique sur les performances.

9.3 Comparaison avec des techniques récentes

Dans cette partie, nous comparons les performances des deux versions, séquentielle et parallèle bornée, de l'approche « arbre-IM » avec quelques techniques d'indexation métrique récentes. Comme dans la partie séquentielle, nous avons retenu les même trois méthodes : arbres-MM, arbre-oignon et arbre-slim.

9.3.1 Protocole des expérimentations

Nous lançons 100 requêtes différentes sur les collection utilisées durant la construction. Mais, cette fois ci, nous ne faisons varier qu'un seul paramètre, *k*, avec les valeurs 5, 10, 20, 25, 50 et 100.

Pour notre approche nous choisissons les meilleures valeurs pour les deux paramètres α et $c_{\max}$. La construction se fait en mode incrémental.

Dans chacun des scénarios, nous calculons :

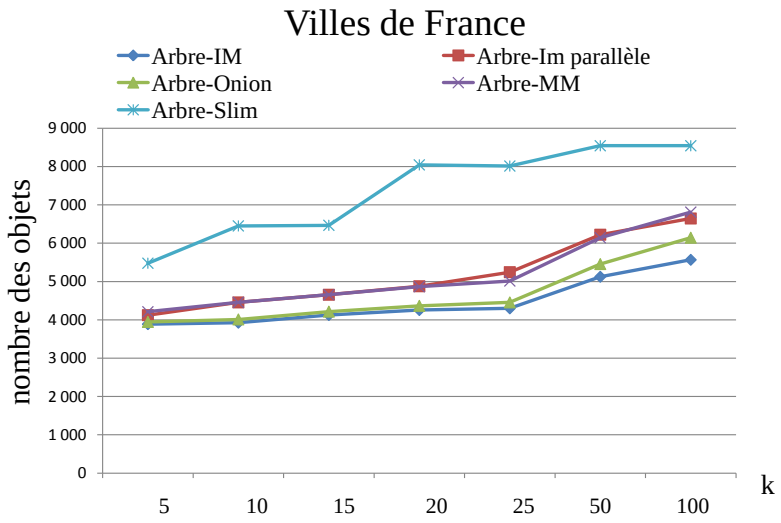
- le nombre d'objet rencontrés (n_1) ;
- le nombre de distances calculées (n_2) ;
- le nombre de comparaisons nécessaires (n_3) ;

pour la construction d'un arbre-IM.

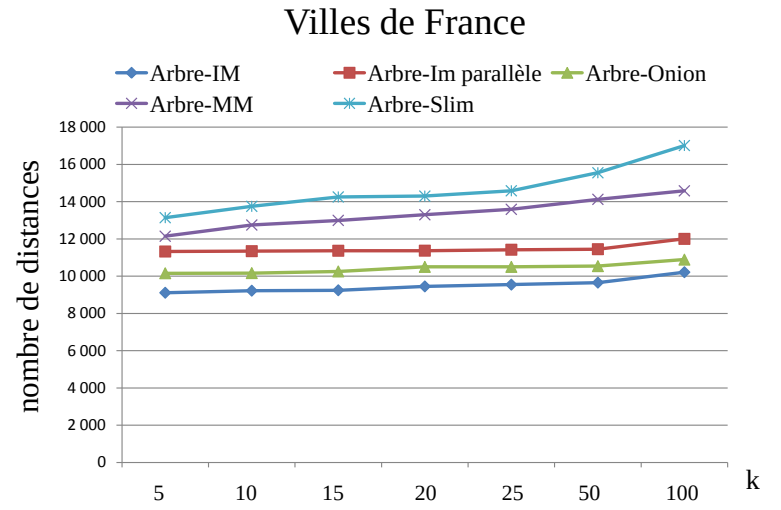
Les figures 9.6 9.7 9.8 montrent le nombre d'objet (n_1) , le nombre de distances (n_2) et le nombre de comparaison (n_3) effectuées lors la recherche kNN pour chaque collection.

Nous faisons les constatations suivantes :

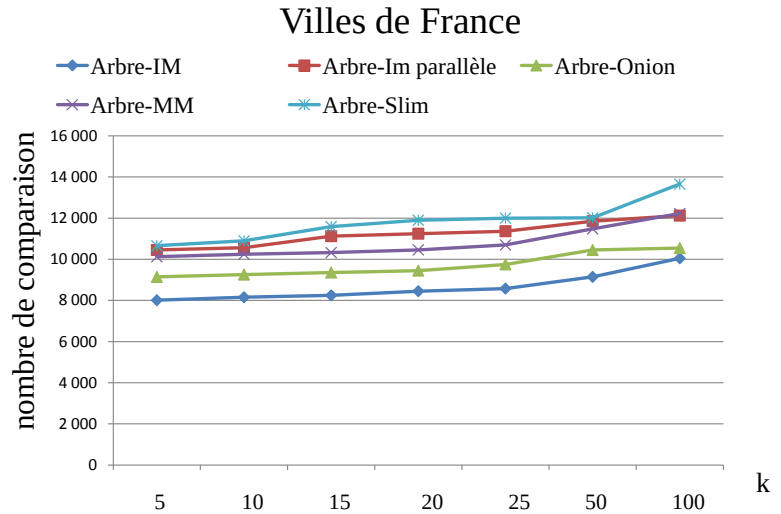
1. L'approche parallèle bornée proposée offre un potentiel important même si elle dépend du nombre de processeurs disponibles. Elle est encore plus intéressante avec un algorithme heuristique non aléatoire permettant de fournir une estimation du rayon de recherche.
2. Sans exceptions, et comme nous l'avons aussi remarqué dans le chapitre précédent, l'augmentation de la valeur de *k* augmente le coût des algorithmes de recherche sur les trois collections, sur chaque scénario et avec toutes les méthodes d'indexation.
3. On note aussi, en comparant les méthodes entre elles, que l'arbre-slim est le plus coûteux.



(a) n_1

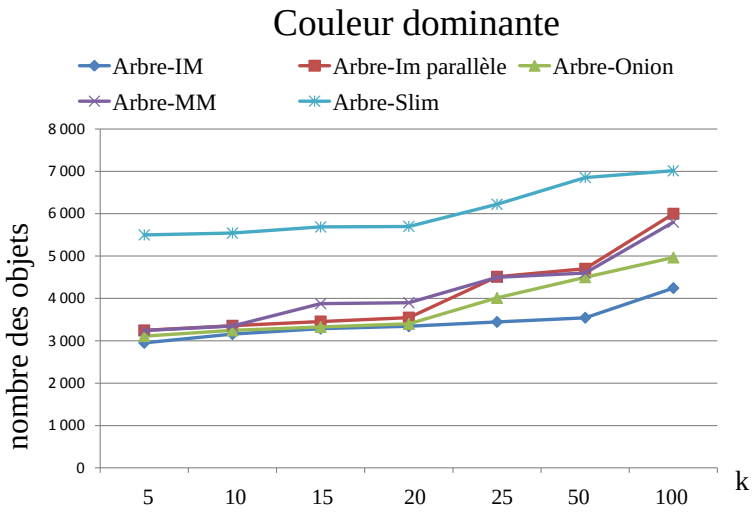


(b) n_2

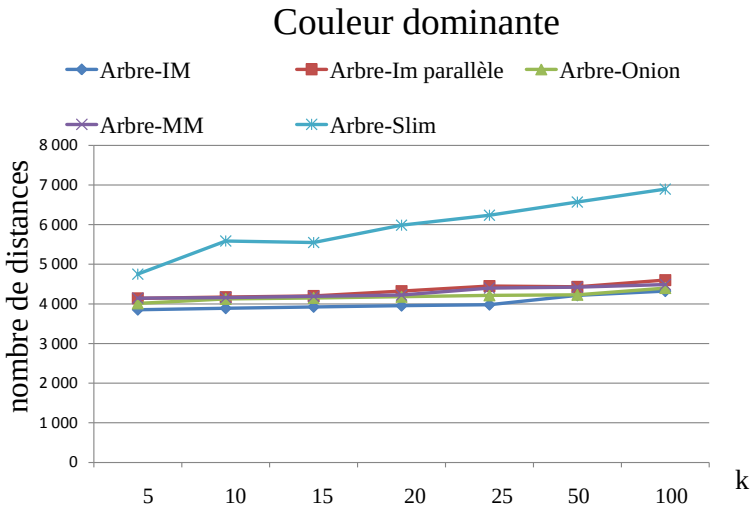


(c) n_3

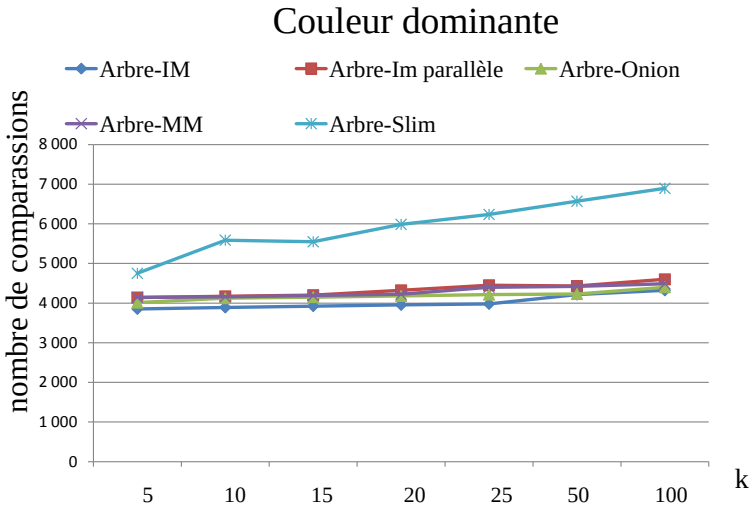
FIGURE 9.6 – Nombre d'objets parcourus, de distances calculées et de comparaisons effectuées dans une recherche kNN dans la collection « Villes de France » des arbres IM, oignon, MM et slim



(a) n_1

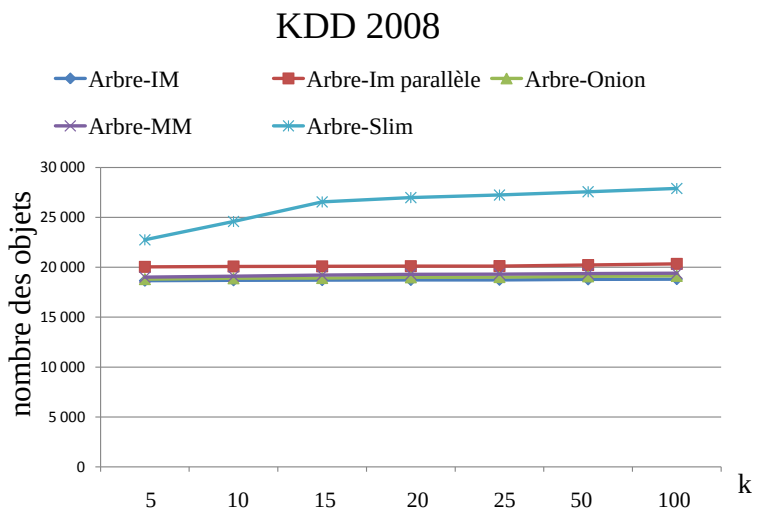


(b) n_2

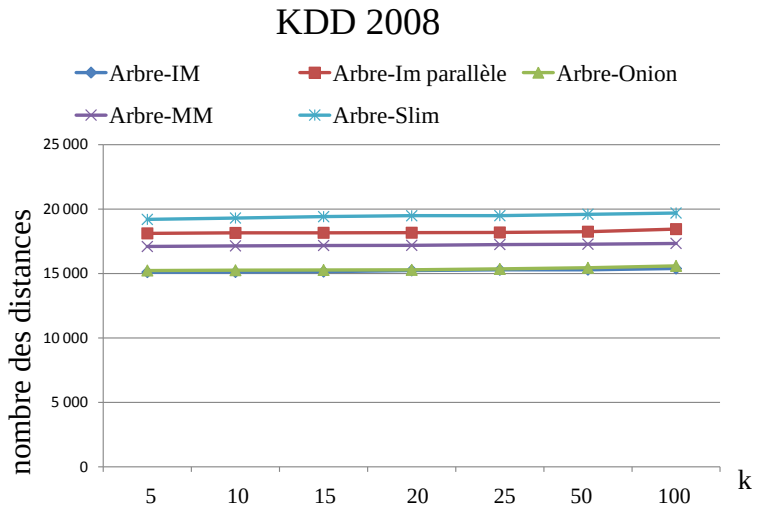


(c) n_3

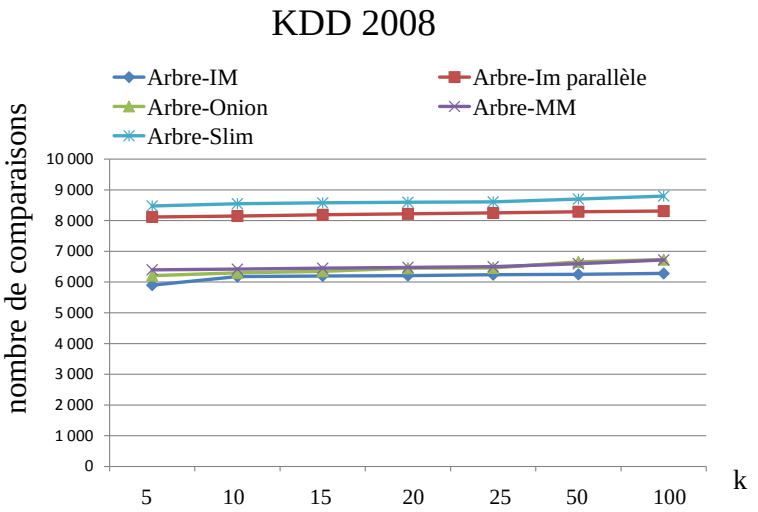
FIGURE 9.7 – Nombre d'objets parcourus, de distances calculées et de comparaisons effectuées une recherche kNN dans la collection « Couleurs dominantes » des arbres IM, oignon, MM et slim



(a) Nombre de distances



(b) Nombre de comparaisons



(c) Nombre d'éléments

FIGURE 9.8 – Comparaisons du nombre d'objets candidats, du nombre de distances calculées et du nombre de comparaisons effectuées dans une recherche kNN sur la collection « KDD 2008 » avec les arbres IM, oignon, MM et slim

10

Conclusion et perspectives

La quantité d'information numérisée a été multipliée dans des proportions considérables durant les dernières années (par exemple, plusieurs milliards d'images sur la Toile, plusieurs millions pour une agence photographique, plusieurs dizaines ou centaines de milliers chez les particuliers). Par conséquent, disposer d'un système de recherche *efficient* dans ces masses de données a été l'objectif principal de notre travail.

Nous avons étudié un ensemble de techniques d'indexation dans les espaces métriques ainsi que dans les espaces multidimensionnels, un sous-ensemble strict. Nous n'avions ni le temps ni la place d'effectuer un tour d'horizon exhaustif. Mais nous avons proposé une comparaison structurelle entre les techniques en listant les point communs et les différences, aussi bien sur la construction des index que sur les algorithmes de recherche. D'autres auteurs ont proposé un survol bien plus complet de cette littérature.

Ce que nous avons cherché à mettre en évidence sont les différences et similitudes entres un certain nombre d'approches. Nous avons effectivement pu constater qu'au delà du nombre de variantes, quelques idées clés étaient présentes dans les index. En particulier, nous avons pu mettre en évidence que les index sur les espaces métriques, malgré la disparition de la notion d'axe et de dimension, utilisent des approches similaires aux index sur les espaces multidimensionnels.

En nous appuyant sur ce qui semble être les meilleures approches récentes, nous avons proposé une nouvelle approche pour indexer les données d'un espace métrique, inspiré de l'arbre-MM essentiellement. Succinctement, notre proposition peut être vue comme une version paginée et paramétrée de ce dernier.

Dans un premier temps, nous avons pu montrer un gain vis-à-vis de son prédé-

cesseur dans une version séquentielle.

Néanmoins, le problème rencontré à chaque fois par ces techniques est celui de la « malédiction de la dimension ». Dans un second temps, nous avons développé une version parallèle afin d'améliorer des performances qui nous semblent devoir sinon stagner du moins peu progresser dans cette famille d'index.

Les perspectives que nous tirons de ce travail se déclinent en travail à court terme et en idées de portée plus lointaine.

Tout d'abord, il nous semble utile de poursuivre le travail, sans doute long et fastidieux, d'introduire des alternatives *a priori* viables tenant compte des idées clés ainsi que de faire varier quelques paramètres sur ces versions. En effet, tout travail d'optimisation s'appuyant sur des heuristiques doit trouver un équilibre entre le coût d'appliquer l'heuristique elle-même et le gain qu'elle apporte à l'algorithme d'origine. Minimiser leur somme est souvent délicat surtout si l'on combine plusieurs heuristiques, ce que nous avons fait.

À plus long terme, nous pouvons introduire plusieurs voies de recherche.

Une approche extrêmement utile lorsque l'on manipule des données complexes en grande quantité est de combiner plusieurs index, utilisant des métriques différentes, plutôt que de devoir créer « à la volée » un nouvel index. Par exemple, dans une base de données multimédia, on pourrait disposer d'index indépendants portant, d'une part, sur des descripteurs extraits à partir du contenu des médias comme des histogrammes de couleurs, et, d'autre part, sur des descriptions sémantiques, au minimum de simples mots clés.

Une autre extension vise à améliorer la parallélisation en la portant sur un environnement réel, aussi bien une grappe de machines (c'est-à-dire un environnement bien contrôlé) qu'un environnement pair à pair. Au delà des difficultés techniques, un index arborescent présente un goulot d'étranglement qui est sa racine. Ainsi donc, mieux paralléliser nécessite un travail important qui peut aller jusqu'à remettre en cause la notion même d'arborescence.

Table des figures

Liste des tableaux

Liste des algorithmes

Bibliographie

[1] C. H. ANG AND T. TAN, *Advances in spatial databases*, in Proceedings of the International Multi-Conference on Systems, Signals and Devices (SSD), 1997. 42, 104

[2] F. AURENHAMMER, *Voronoi diagrams - a survey of a fundamental geometric data structure*, ACM Computing Surveys, 23 (1991), pp. 345–405. 73

[3] M. BATKO, D. NOVAK, F. FALCHI, AND P. ZEZULA, *On scalability of the similarity search in the world of peers*, in Proceedings of the 1st international conference on Scalable information systems (InfoScale), Hong Kong, China, May 2006, ACM Press, pp. 20–31. 10

[4] R. BAYER AND E. MCCREIGHT, *Organization and maintenance of large ordered indexes*, Acta Informatica, 1, Fasc. 3 (1972), pp. 173–189. 37

[5] N. BECKMANN, H. KRIEGEL, R. SCHNEIDER, AND B. SEEGER, *The R*-tree : an efficient and robust access method for points and rectangles.*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, vol. 19, 1990, pp. 322–331. 42

[6] J. L. BENTLEY, *Multidimensional binary search trees used for associative searching*, Communications of the ACM, 18 (1975), pp. 509–517. 90

[7] ——, *Multidimensional binary search in database application*, IEEE Transactions on Software Engineering, 5 (1979), pp. 333–340. 46, 47, 48, 49, 51, 90, 169

[8] S. BERCHTOLD, C. BÖHM, AND H.-P. KRIEGEL, *The pyramid technique : Towards breaking the curse of dimensionality*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, vol. 27, 1998, pp. 142–153. 27

[9] S. BERCHTOLD, D. A. KEIM, AND H.-P. KRIEGEL, *The X-tree : An index structure for high-dimensional data*, in Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB), Mumbai (Bombay), India, 1996, pp. 28–39. 44, 45, 46, 90, 169

[10] S. BERCHTOLD, D. A. KEIM, AND H.-P. KRIEGEL, *A cost model for nearest neighbor search in high-dimensional data space*, in Proceedings of the International Symposium on Principles of Database Systems, 1997, pp. 12–14. 107, 108

[11] M. D. BERG, O. CHEONG, M. V. KREVELD, AND M. OVERMARS, *Computational Geometry : Algorithms and Applications, 3rd Ed.*, Springer-Verlag, Mar. 2008. 51

[12] K. BEYER, J. GOLDSTEIN, R. RAMAKRISHNAN, AND U. SHAFT, *When is "nearest neighbor" meaningful ?*, in Proceedings of the International Conference on Database Theory (ICDT), C. Beeri and P. Buneman, eds., vol. 1540 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 1999, pp. 217–235. 30, 31

[13] H. BLOK, A. DE VRIES, AND H. BLANKEN, *Top N MM query optimization - the best of both IR and DB worlds*, in Conferentie Informatiewetenschap (Information Science Conference), Amsterdam, The Netherlands, Dec. 1999. 91

[14] C. BÖHM, S. BERCHTOLD, AND D. A. KEIM, *Searching in high-dimensional spaces : Index structures for improving the performance of multimedia databases*, ACM Computing Surveys, 33 (2001), pp. 322–373. 27, 33, 37

[15] C. BÖHM, S. BERCHTOLD, H.-P. KRIEGEL, AND U. MICHEL, *Multidimensional index structures in relational databases*, Journal Intelligent Information Systems, 15 (2000), pp. 51–70. 17

[16] P. BOLETTIERI, F. FALCHI, C. LUCCHESE, Y. MASS, R. PEREGO, F. RABITTI, AND M. SHMUELI-SCHEUER, *Searching 100m images by content similarity*, in Post-proceedings of the 5th Italian Research Conference on Digital Library Systems (IRCD), Padova, Italy, Jan. 2009, pp. 88–99. 7, 120

[17] T. BOZKAYA AND M. ÖZSOYOGLU, *Distance-based indexing for high-dimensional metric spaces*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, 1997, pp. 357–368. 67

[18] ——, *Indexing large metric spaces for similarity search queries*, ACM Transactions on Database Systems, 24 (1999), pp. 361–404. 10, 28, 67, 93, 95

[19] T. BOZKAYA AND M. OZSOYOGLU, *Indexing large metric spaces for similarity search queries*, ACM Transactions on Database Systems, 24 (2002), pp. 361–404. 95

[20] S. BRIN, *Near neighbor search in large metric spaces*, in Proceedings of the 21st International Conference on Very Large Data Bases (VLDB), 1995. 72, 73, 93, 95, 169

[21] W. BURKHARD AND R. KELLER, *Some approaches to best-match file searching*, Communications of the ACM, 16 (1973), pp. 230–236. 10

[22] B. BUSTOS, G. NAVARRO, AND E. CHAVEZ, *Pivot selection techniques for proximity searching in metric spaces*, in Pattern Recognittion Letters, 2003. 108

[23] B. BUSTOS AND T. SKOPAL, *Dynamic similarity search in multimetric spaces*, in Proceedings of the 8th ACM international workshop on Multimedia information retrieval, 2006. 93

[24] C. C. M. CARÉLO, I. R. V. POLA, R. R. CIFERRI, A. J. M. TRAINA, C. T. JR., AND C. D. DE AGUIAR CIFERRI, *Slicing the metric space to provide quick indexing of complex data in the main memory*, Information Systems, 36 (2011), pp. 79–98. 80, 81, 82, 84, 85, 86, 88, 94, 95, 169

[25] E. CHAVEZ, G. NAVARRO, J. L. MARROQUIN, AND R. BAEZA-YATES, *Searching in metric spaces*, ACM Computing Surveys, 33 (2001), pp. 273–321. 19, 28, 67, 108, 118

[26] P. CIACCIA AND M. PATELLA, *Bulk loading the M-tree*, in Proceedings of the 9th Australian Database Conference (ADC), 1998. 55, 108

[27] P. CIACCIA AND M. PATELLA, *Using the distance distribution for searching in high-dimensional metric spaces*, in Atti del Settimo Convegno Nazionale su Sistemi Evoluti per Basi di Dati (SEBD), 1999. 57, 95

[28] ——, *Searching in metric spaces with user-defined and approximate distances*, ACM Transactions on Database Systems (TODS), 27 (2002), pp. 398–437. 55

[29] P. CIACCIA, M. PATELLA, AND P. ZEZULA, *M-tree : An efficient access method for similarity search in metric spaces*, in Proceedings of the International Conference on Very Large Data Bases (VLDB), 1997. 55, 59

[30] P. CIACCIA, M. PATELLA, AND P. ZEZULA, *Processing complex similarity queries with distance-based access methods*, in Proceedings of the 6th International conference on Extending Database Technology (EDBT), 1998. 93

[31] D. COMER, *Ubiquitous B-tree*, ACM Computing Surveys, 11 (1979), pp. 121–137. 90

[32] T. CORMEN, C. LEISERSON, AND R. RIVEST, *Algorithmique*, Dunod, Sciences Sup, 2010. 49

[33] T. F. COX AND M. A. COX, *Multidimensional Scaling*, vol. 1, Chapman and Hall, 1994. 90

[34] C. FALOUTSOS, R. BARBER, M. FLICKNER, J. HAFNER, W. NIBLACK, D. PETKOVIC, AND W. EQUITZ, *Efficient and effective querying by image content*, Journal of Intelligent Information Systems (JIIS), 3 (1994), pp. 231–262. 90, 108

[35] C. FALOUTSOS AND K.-I. LIN, *Fastmap : A fast algorithm for indexing, data-mining and visualization of traditional and multimedia data*, in Proceedings of ACM SIGMOD International Conference on Management of Data, San Jose, CA, 1995, pp. 163–174. 21, 90, 109, 194

[36] V. GAEDE AND O. GÜNTHER, *Multidimensional access methods*, ACM Computing Surveys, 30 (1998), pp. 170–231. 17, 33, 40, 41, 44, 90, 169

[37] V. GUDIVADA AND V. RAGHAVAN, *Content-based image retrieval systems*, Computer, 28 (1995), pp. 18–22. 8

[38] W. GUOREN, X. ZHO, W. BIN, B. QIAO, AND D. HAN, *A hyperplane-based indexing technique for high data*, Information Sciences, 177 (2007), pp. 2255–2268. 104

[39] A. GUTTMAN, *R-trees : A dynamic index structure for spatial searching*, in AProceedings of the 1984 ACM SIGMOD international conference on Management of data, 1984, pp. 47–57. 34, 36, 42, 90

[40] J. L. HAFNER, H. S. SAWHNEY, W. EQUITZ, M. FLICKNER, AND W. NIBLACK, *Efficient color histogram indexing for quadratic form distance functions*, IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), 17 (1995), pp. 729–736. 90

[41] A. HENRICH, *The LSDh-tree : An access structure for feature vectors*, in Proceedings of the 14th International Conference on Data Engineering (ICDE), 1998. 90

[42] G. R. HJALTASON AND H. SAMET, *Index-driven similarity search in metric spaces*, ACM Transactions on Database Systems, 28 (2003), pp. 517–580. 73, 95

[43] K. JOHN, *General Topology*, Van Nostrand, 1955. 89

[44] K. KANTH, D. AGRAWAL, A. EL ABBADI, A. SINGH, AND T. SMITH, *Parallelizing multidimensional index structures*, in Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing, New Orleans, LA, Oct. 1996, pp. 376–383. 12

[45] N. KATAYAMA AND S. SATOH, *The SR-tree : An index structure for high-dimensional nearest neighbor queries*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, 1997. 42, 43, 44, 45, 169

[46] S. M. KAUSHIK CHAKRABARTI, *The hybrid tree : An index structure for high dimensional feature spaces*, in Proceedings of the 15th International Conference on Data Engineering (ICDE), 1999. 90

[47] F. KORN, B.-U. PAGEL, AND C. FALOUTSOS, *On the 'dimensionality curse' and the 'self-similarity blessing'*, IEEE Transactions on Knowledge and Data Engineering, 13 (2001), pp. 96–111. 30

[48] Z. KOUAHLA, *Exploring intersection trees for indexing metric spaces*, in 3e conférence internationale sur l'informatique et ses applications (CIIA), Saida, Algérie, Dec. 2011, p. 6. 13

[49] Z. KOUAHLA AND J. MARTINEZ, *Indexing in metric spaces*, in First Sino-French Workshop on Education and Research collaborations in Information and Communication Technologies (SIFWICT), Nantes, France, May 2011, p. 2. 13

[50] ——, *A new intersection tree for content-based image retrieval*, in 10th International Workshop on Content-Based Multimedia Indexing (CBMI), vol. 2, Annecy, France, June 2012, pp. 1–6. 13

[51] L. KOVÁCS, *Reduction of distance computations in selection of pivot elements for balanced GHT structure*, Proceedings of the 8th international conference on Machine Learning and Data Mining in Pattern Recognition, 13 (2012), pp. 50–62. 104

[52] K. KUKICH, *Technique for automatically correcting words in text*, ACM Computing Surveys, 24 (1992), pp. 377–439. 89

[53] D. LEE AND C. WONG, *Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees*, in Acta Informatica, 1977. 49

[54] D. LOWE, *Distinctive image features from scale-invariant keypoints*, International Journal of Computer Vision, 60 (2004), pp. 91–110. 7

[55] J. B. MACQUEEN, *Some methods for classification and analysis of multivariate observations*, Fifth Berkeley Symposium, 1 (1966), pp. 281–297. 33

[56] J. R. MANJARREZ SANCHEZ, *Efficient Content-Based Retrieval in Parallel Databases of Images*, PhD thesis, Université de Nantes, 2009. 11, 23, 29, 169

[57] J. R. MANJARREZ SANCHEZ, J. MARTINEZ, AND P. VALDURIEZ, *Efficient processing of nearest neighbor queries in parallel multimedia databases*, in Proceedings of the 19th International Conference on Database and Expert Systems Applications (DEXA), 2008. 12

[58] Y. MANOLOPOULOS, A. NANOPOULOS, A. N. PAPADOPOULOS, AND T. Y, eds., *R-Trees : Theory and Applications (Advanced Information and Knowledge Processing)*, Springer-Verlag New York, Secaucus, NJ, USA, 2005. 34, 36, 37, 169

[59] J. MARTINEZ AND Z. KOUAHLA, *Indexing metric spaces with nested forests*, in Proceedings of the 23rd International Conference on Database and Expert Systems Applications (DEXA), vol. 2 of LNCS 7447, Vienna, Austria, Sept. 2012, pp. 458–465. 13, 124

[60] M. L. MICO, J. ONCINA, AND E. VIDAL, *A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements*, in Pattern Recognition Letters, 1994. 107

[61] K. MIKOLAJCZYK AND SCHMID, *A performance evaluation of local descriptors*, IEEE Transactions on Pattern Analysis, 10 (2005), pp. 1615–1630. 7

[62] P. O'NEIL AND D. QUASS, *Improved query performance with variant indexes.*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, 1997. 90

[63] R. U. PAREDES AND G. NAVARRO, *EGNAT : A fully dynamic metric access method for secondary memory*, in Proceedings of the Second International Workshop on Similarity Search and Applications, 2009. 73, 95

[64] M. PATELLA, *Similarity Search in Multimedia Databases*, PhD thesis, Università degli Studi di Bologna, 1999. No. 1448. 9

[65] I. R. POLA, C. TRAINA, JR, AND A. J. TRAINA, *The MM-tree : A memory-based metric tree without overlap between nodes*, in Proceedings of the East-European Conference on Advances in Databases and Information Systems (ADBIS), vol. LNCS 4690, 2007, pp. 157–171. 13, 75, 77, 79, 93, 95, 104, 169, 173

[66] K. RAVI KANTH, D. AGRAWAL, AND A. SINGH, *Dimensionality reduction for similarity searching in dynamic databases*, in Proceedings of the 1998 ACM SIGMOD international conference on Management of data, 1998. 90

[67] J. R. RICO-JUAN AND L. MICO, *Comparison of AESA and LAESA search algorithms using string and tree-editdistances.*, Pattern Recognition Letters, 24 (2003), pp. 1417–1426. 54

[68] P. A. H. ROBERT G. GALLAGER, *A distributed algorithm for minimum-weight spanning trees*, ACM Transactions on Programming Languages and Systems, 5 (1983), pp. 66–77. 62

[69] J. T. ROBINSON, *The K-D-B-tree : A search structure for large multidimensional dynamic indexes*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, 1981. 51

[70] J. B. ROSENBERG, *Geographical data structures compared : A study of data structures supporting region queries*, IEEE Transactions on CAD of Integrated Circuits and Systems, 4 (1985), pp. 53–67. 51

[71] M. SALVETTI, C. DECO, N. REYES, AND C. BENDER, *Adaptive and dynamic pivot selection for similarity search*, Journal of Information and Data Management (JIDM), 2 (2011), pp. 27–34. 104

[72] H. SAMET, *The quadtree and related hierarchical data structures*, ACM Computing Surveys, 16 (1984), pp. 187–260. 90

[73] ——, *Foundations of Multidimensional and Metric Data Structures*, Morgan-Kaufmann, Sept. 2006. 993 p. 33, 120

[74] T. K. SELLIS, N. ROUSSOPOULOS, AND C. FALOUTSOS, *The R+-tree : A dynamic index for multi-dimensional objects*, in Proceedings of the 13th International Conference on Very Large Data Bases (VLDB), 1987. 42

[75] B. SILVERMAN, *Density estimation for statistics and data analysis*, Monographs on statistics and applied probability, Chapman and Hall, 1986. 28

[76] S. SKIENA, *The Algorithm Design Manual*, ISBN 0-387-94860-0, Springer & Verlag, 1997. 10

[77] T. SKOPAL, J. POKORNÝ, M. KRÁTKÝ, AND V. SNÁSEL, *Revisiting M-tree building principles*, 7th East European Conference Advances in Databases and Information Systems, 2798 (2003), pp. 3–6. 64, 129

[78] M. SWAIN AND D. BALLARD, *Color indexing*, International Journal of Computer Vision, 7 (1991), pp. 11–32. 7

[79] C. TRAINA, JR., A. TRAINA, C. FALOUTSOS, AND B. SEEGER, *Fast indexing and visualization of metric data sets using slim-trees*, IEEE Transactions on Knowledge and Data Engineering, 14 (2002), pp. 244–260. 64, 95

[80] C. TRAINA JR, A. J. TRAINA, AND ET AL., *A portable C++ library which implements various metric access method*. Gbdi arboretum, 2005. 127, 138

[81] C. TRAINA, JR, A. J. TRAINA, B. SEEGER, AND C. FALOUTSOS, *Slim-trees : High performance metric trees minimizing overlap between nodes*, in Proceedings of the International conference on Extending Database Technology (EDBT), 2000. 59, 63, 64, 93, 95, 129, 169

[82] J. K. ULHMANN, *Satisfying general proximity/similarity queries with metric trees*, Information Processing Letters, 40 (1991), pp. 175–179. 67, 68, 93, 169

[83] E. VIDAL, *New formulation and improvements of the nearest-neighbour approximating and eliminating search algorithm (aesa)*, Pattern Recognition Letters, 15 (1994), pp. 1–7. 54

[84] E. VIDAL RUIZ, *An algorithm for finding nearest neighbors in (approximately) constant average time*, Pattern Recognition Letters, 4 (1986), pp. 145–157. 54

[85] P. VLADIMIR, *Intrinsic dimensionality*, SIGSPATIAL Special, 2 (2010), pp. 8–11. 28, 104

[86] I. WALD AND V. HAVRAN, *On building fast kd-trees for ray tracing, and on doing that in o(nlogn) on building fast kd-trees for ray tracing, and on doing that in o(nlogn)*, in IEEE Symposium on Interactive Ray Tracing, 2006. 49

[87] R. WEBER AND C. BÖHM, *Trading quality for time with nearest neighbor search*, Proceedings of the 7th international Conference on Extending Database Technology, 1777 (2000), pp. 21–35. 33

[88] R. WEBER, H.-J. SCHEK, AND S. BLOTT, *A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces*, in Proceedings of the 24th International Conference on Very Large Data Bases (VLDB), New York, NY, 1998, pp. 194–205. 33

[89] B. WILKINSON AND M. ALLEN, *Parallel Programming : Techniques and Applications Using Networked Workstations and Parallel Computers, 2nd Edition*, Prentice-Hall, Mar. 2004. 431 p. 12

[90] J.-K. WU, A. NARASIMHALU, B. MEHTRE, C.-P. LAM, AND Y. GAO, *Core : A content-based retrieval engine for multimedia information systems*, Multimedia Systems, 3 (1995), pp. 25–41. 8

[91] P. N. YIANILOS, *Data structures and algorithms for nearest neighbor search in general metric spaces*, in Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms, 1993. 10, 64, 65, 93, 95, 107, 108, 169

[92] A. YOSHITAKA AND T. ICHIKAWA, *A survey on content-based retrieval for multimedia databases*, IEEE Transactions on Knowledge and Data Engineering, 11 (1999), pp. 81–93. 7

[93] P. ZEZULA, G. AMATO, V. DOHNAL, AND M. BATKO, *Similarity Search - The Metric Space Approach*, vol. 32 of Advances in Database Systems, Springer, 2006. 19, 108

[94] ——, *Similarity Search : The Metric Space Approach*, Springer, Nov. 2010. 220 p. 55, 64, 120, 144, 169, 170

[95] P. ZEZULA, P. SAVINO, P. CIACCIA, AND F. RABITTI, *On the region proximity in metric spaces*, in Proceedings of the 1st International Workshop on Similarity Search (IWOSS'99), 1999. 55

A

Notations utilisées

Ensembles et notations « standards » :

- $\mathbb{N}$: ensemble des entiers naturels ;
- $\mathbb{N}^*$: ensemble des entiers naturels strictement positifs ;
- $\mathbb{R}$: ensemble des réels ;
- $\mathbb{R}^+$: ensemble des réels positifs ou nuls ;
- $\mathbb{R}^{+*}$: ensemble des réels strictement positifs.
- $\lfloor x \rfloor$: plus grand entier inférieur ou égal à x (partie entière ou quotient de la division euclidienne) ;
- x mod y : reste de la division euclidienne de x par y avec $y \in \mathbb{N}^*$;
- x div y : quotient de la division euclidienne de x par y avec $y \in \mathbb{N}^*$;
- $\lambda x, y, \ldots z.exp(x, y, \ldots, z)$: définition de fonction anonyme (λ-fonction, fonction « en ligne ») ;
- $\{o \in \mathcal{O} : P(o)\}$ ou $\{o \in \mathcal{O} | P(o)\}$: définition en intention, sous-ensemble des éléments d'un ensemble $\mathcal{O}$ vérifiant le prédicat P.

Ensembles d'objets :

- $\mathcal{O}$: ensemble d'objets de référence ;
- $S \in \mathcal{P}(\mathcal{O})$ ou $S \subseteq \mathcal{O}$: sous-ensemble d'objets ;

- $\mathcal{O}^{\mathbb{N}}$ ou $\mathcal{O}^n$: séquence d'objets, de longueur variable ou égale à n respectivement ;
- $\mathbb{R}^n$: vecteur dans un espace réel de dimension n, cas particulier de $\mathcal{O}$;
- $o \in \mathcal{O}$: un objet ;
- $q \in \mathcal{O}$: un objet-requête ;
- $p, p_1, p_2, \ldots \in \mathcal{O}$: des objets « pivots » ;
- $m \in \mathcal{O}$: un objet « médian » ;
- $n = |\mathcal{O}| \in \mathbb{N}$: nombre total d'objets à indexer ;
- $d : \mathcal{O} \times \mathcal{O} \to \mathbb{R}^+$: une fonction de distance, ou métrique, sur les objets ;
- $k \in \mathbb{N}^*$: nombre d'éléments à renvoyer dans une recherche de plus proches voisins (kNN pour *k-nearest-neighbours*).

Arbres :

- $\mathcal{N} \subseteq T_1 \times T_2 \times \cdots \times T_n$: ensemble des nœuds dans un index arborescent, issus d'un produit cartésien des composantes le décrivant ;
- $N \in \mathcal{N}$: nœud (interne) dans un index arborescent ;
- $F \in \mathcal{N}$: nœud feuille dans un index arborescent, si la distinction est possible.

B

Une implémentation de l'arbre-IM

À titre d'illustration « pédagogique », nous fournissons ici une implémentation quasi directe, dans le langage d'assez haut niveau qu'est Python, de notre proposition d'arbre-IM. Il s'agit de traduire les pseudo codes mathématiques décrits dans le chapitre 6 ainsi que d'y ajouter quelques codes de test et d'« instrumentation » (*profiling*) manuelle.

Tout cela fournit au lecteur la base à une implémentation plus efficiente et de plus large ampleur. En effet, le langage Python n'est pas assez rapide pour traiter des volumes importants de données, notamment lorsque les distances utilisées sont coûteuses (ex : la distance d'édition sur les chaînes). Par ailleurs, cette version ne fonctionne qu'en mémoire centrale. Enfin, elle ne tient pas compte de la présence de doublons (*a priori* rares mais devant toutefois être gérés de manière adéquate).

B.1 Construction d'un arbre-IM

Rappelons que notre proposition permet de construire aussi bien un arbre en lot, lorsque toutes les données sont connues à l'avance, que de manière incrémentale.

B.1.1 Construction en lot

L'algorithme 32 est la traduction directe de l'algorithme 26 en page 110. Quelques assertions ont été ajoutées pour clarifier le code. Noter que la syntaxe des listes en intention de Python permet de produire des expressions très proches des notations ensemblistes classiques : `[e for e in E if P(e)]` pour $\{e \in E : P(e)\}$.

Algorithme 32 Construction d'un arbre-IM en Python

```
def ConstruireIM (E, d, cmax, alpha):
    assert type(E) is list
    assert type(cmax) is int
    assert cmax > 0
    assert type(alpha) is float
    assert 0.5 < alpha < 1
    if len(E) <= cmax:
        return E
    else:
        ((p1, p2), d12) = PlusEloignesApprox(E, d)
        assert d12 == d(p1, p2)
        assert d12 > 0
        r = d12 * alpha
        E1 = [e for e in E
                if d(p1, e) <= r and d(p2, e) <= r]
        E2 = [e for e in E
                if d(p1, e) <= r and d(p2, e) >  r]
        E3 = [e for e in E
                if d(p1, e) >  r and d(p2, e) <= r]
        E4 = [e for e in E
                if r < d(p1, e) and d(p1, e) <= d(p2, e)]
        E5 = [e for e in E
                if r < d(p2, e) and d(p1, e) >  d(p2, e)]
        assert p1 in E2
        assert p2 in E3
        r1 = max([d(p1, e) for e in E4]) if E4 != [] else r
        r2 = max([d(p2, e) for e in E5]) if E5 != [] else r
        return (p1, p2, r, r1, r2, \
                ConstruireIM(E1, d, cmax, alpha), \
                ConstruireIM(E2, d, cmax, alpha), \
                ConstruireIM(E3, d, cmax, alpha), \
                ConstruireIM(E4, d, cmax, alpha), \
                ConstruireIM(E5, d, cmax, alpha))
```

Algorithme 33 Couple d'éléments les plus éloignés en Python, version récursive

```
def PlusEloigneDe (e0, E, d):
    assert type(E) is list
    assert len(E) >= 1
    ED = [(e, d(e0, e)) for e in E]
    dm = max([d for (e, d) in ED])
    em = [e for (e, d) in ED
            if d == dm][0]
    assert dm == d(e0, em)
    return ((e0, em), dm)

def PlusEloignesApprox (E, d, i = 5):
    assert type(E) is list
    assert len(E) >= 2
    assert type(i) is int
    assert i >= 0
    if i == 0:
        return ((E[0], E[1]), d(E[0], E[1]))
    else:
        ((e0, e1), d01) = PlusEloignesApprox(E, d, i - 1)
        assert d01 == d(e0, e1)
        return PlusEloigneDe(e1, E, d)
```

Algorithme 34 Couple d'éléments les plus éloignés en Python, version itérative

```
def PlusEloignesApproximatif (E, d,
                              i = 5):
    assert type(E) is list
    assert len(E) >= 2
    assert type(i) is int
    assert i >= 0
    e1 = E[0]
    for j in range(0, i):
        e0 = e1
        dmax = 0
        for e in E:
            d = d(e0, e)
            if d > dmax:
                e1 = e
                dmax = d
    return ((e0, e1), dmax)
```

Algorithme 35 Insertion dans un arbre-IM en Python

```
def InsérerIM (e, N, d, cmax, alpha):
    assert type(cmax) is int
    assert cmax > 0
    assert type(alpha) is float
    assert 0.5 < alpha < 1
    if type(N) is list and len(N) < cmax:
        return N + [e]
    elif type(N) is list and len(N) == cmax:
        return ConstruireIM(N + [e], d, cmax, alpha)
    else:
        (p1, p2, r, r1, r2, N1, N2, N3, N4, N5) = N
        assert r == d(p1, p2) * alpha
        if d(p1, e) <= r and d(p2, e) <= r:
            return (p1, p2, r, r1, r2, \
                    InsérerIM(e, N1, d, cmax, alpha), N2, N3, N4, N5)
        elif d(p1, e) <= r and d(p2, e) > r:
            return (p1, p2, r, r1, r2, \
                    N1, InsérerIM(e, N2, d, cmax, alpha), N3, N4, N5)
        elif d(p1, e) >  r and d(p2, e) <= r:
            return (p1, p2, r, r1, r2, \
                    N1, N2, InsérerIM(e, N3, d, cmax, alpha), N4, N5)
        elif r < d(p1, e) and d(p1, e) <= d(p2, e):
            return (p1, p2, r, max(r1, d(p1, e)), r2, \
                    N1, N2, N3, InsérerIM(e, N4, d, cmax, alpha), N5)
        elif r < d(p2, e) and d(p1, e) >  d(p2, e):
            return (p1, p2, r, r1, max(r2, d(p2, e)), \
                    N1, N2, N3, N4, InsérerIM(e, N5, d, cmax, alpha))
        else:
            assert False # cannot occur
```

L'algorithme 32 se sert de la fonction « argmax », et plus exactement d'une version approximative – mais linéaire au lieu de quadratique – afin de déterminer les deux éléments « les plus éloignés ». L'algorithme 33 (et la version itérative 34) permet de calculer un couple d'éléments *a priori* assez éloignés l'un de l'autre en itérant – par défaut cinq fois – la recherche de l'élément le plus éloigné de l'élément précédent en en choisissant un quelconque initialement. Noter qu'il s'agit de l'heuristique également utilisée par l'algorithme *FastMap* [35].

B.1.2 Construction incrémentale

L'algorithme 35 est la traduction directe de l'algorithme 27 en page 112. Noter que, sous réserve de borner $c_{\max}$ par la racine carrée de la population, on pourrait utiliser une version exacte du calcul des deux éléments les plus éloignés plutôt que la version approximative présente dans l'algorithme précédent, et réutilisé dans la seconde branche de cet algorithme.

Algorithme 36 Construction incrémentale d'un arbre-IM en Python (version itérative pour éviter un débordement de pile)

```
def ConstruireIMIncrémentalement (E, d, cmax, alpha):
    N = []
    for e in E:
        N = InsérerIM(e, N, d, cmax, alpha)
    return N
```

Algorithme 37 Vérification de la structure d'un arbre-IM en Python

```
def VérifierIM (N, d):
    if type(N) is list:
        pass
    else:
        (p1, p2, r, r1, r2, N1, N2, N3, N4, N5) = N
        assert all([d(p1, e) <= r and d(p2, e) <= r
                    for e in ElementsIM(N1)])
        assert all([d(p1, e) <= r and d(p2, e) >  r
                    for e in ElementsIM(N2)])
        assert all([d(p1, e) >  r and d(p2, e) <= r
                    for e in ElementsIM(N3)])
        assert all([r < d(p1, e) <= r1 and d(p1, e) <= d(p2, e)
                    for e in ElementsIM(N4)])
        assert all([r < d(p2, e) <= r2 and d(p1, e) >  d(p2, e)
                    for e in ElementsIM(N5)])
```

L'algorithme 36 ajoute le code trivial de construction incrémentale d'un arbre à partir d'insertions itérées.

Noter que grâce à la proximité entre la description mathématique et le code Python, les algorithmes de construction 32 et 36 sont corrects sans véritable démonstration ! Néanmoins, suite à la construction d'un arbre-IM, il est possible de les vérifier *a posteriori* grâce au code de l'algorithme 37 au cas où une erreur d'inattention, voire du langage, serait présente.

Pour la vérification, l'algorithme 37 fait appel à une fonction récursive élémen-

Algorithme 38 Éléments d'un (sous) arbre-IM en Python

```
def ElementsIM (N):
    if type(N) is list:
        return N
    else:
        (p1, p2, r, r1, r2, N1, N2, N3, N4, N5) = N
        return ElementsIM(N1) + \
               ElementsIM(N2) + \
               ElementsIM(N3) + \
               ElementsIM(N4) + \
               ElementsIM(N5)
```

Algorithme 39 Visualisation arborescente d'un arbre-IM en Python

```
def IndenterIM (N,
                i = 0):
    assert type(i) is int
    assert i >= 0
    if type(N) is list:
        print(' | ' * i, N)
    else:
        (p1, p2, r, r1, r2, N1, N2, N3, N4, N5) = N
        print(' | ' * i, 'p1 : ', p1)
        print(' | ' * i, 'p2 : ', p2)
        print(' | ' * i, 'r  : ', r)
        print(' | ' * i, 'r1 : ', r1)
        print(' | ' * i, 'r2 : ', r2)
        print(' | ' * i, 'N1')
        IndenterIM(N1, i + 1)
        print(' | ' * i, 'N2')
        IndenterIM(N2, i + 1)
        print(' | ' * i, 'N3')
        IndenterIM(N3, i + 1)
        print(' | ' * i, 'N4')
        IndenterIM(N4, i + 1)
        print(' | ' * i, 'N5')
        IndenterIM(N5, i + 1)
```

taire, décrite par l'algorithme 38, qui renvoie l'ensemble des éléments associés à un nœud de l'arbre. Rappelons que, par définition (cf. définition 21 en page 106), il s'agit d'une partition.

Enfin, l'algorithme 39 est un algorithme récursif élémentaire de visualisation d'un arbre sous forme textuelle.

Algorithme 40 Nombre de feuilles d'un arbre-IM en Python

```
def NombreFeuilles (N):
    if type(N) is list:
        return 1
    else:
        (p1, p2, r, r1, r2, N1, N2, N3, N4, N5) = N
        return NombreFeuilles(N1) + \
               NombreFeuilles(N2) + \
               NombreFeuilles(N3) + \
               NombreFeuilles(N4) + \
               NombreFeuilles(N5)
```

Algorithme 41 Nombre de nœuds internes d'un arbre-IM en Python

```
def NombreNoeudsInternes (N):
    if type(N) is list:
        return 0
    else:
        (p1, p2, r, r1, r2, N1, N2, N3, N4, N5) = N
        return 1 + \
               NombreNoeudsInternes(N1) + \
               NombreNoeudsInternes(N2) + \
               NombreNoeudsInternes(N3) + \
               NombreNoeudsInternes(N4) + \
               NombreNoeudsInternes(N5)
```

Algorithme 42 « Noms » et cardinaux des feuilles d'un arbre-IM en Python

```
def NomCardinalFeuilles (N, name = ''):
    if type(N) is list:
        return [(name, len(N))]
    else:
        (p1, p2, r, r1, r2, N1, N2, N3, N4, N5) = N
        return NomCardinalFeuilles(N1, name + '1') + \
               NomCardinalFeuilles(N2, name + '2') + \
               NomCardinalFeuilles(N3, name + '3') + \
               NomCardinalFeuilles(N4, name + '4') + \
               NomCardinalFeuilles(N5, name + '5')
```

B.2 Statistiques sur un arbre-IM

On peut facilement extraire quelques statistiques sur un arbre-IM grâce à un ensemble de fonctions récursives élémentaires comme le nombre de feuilles et de nœuds internes ainsi que la liste des feuilles et de leurs cardinaux, fournis par les algorithmes 40 à 42. Noter que le « nom » des feuilles est issu de la concaténation des numéros des régions parcourues depuis la racine. La longueur de cette chaîne donne également la profondeur de la feuille dans l'arbre.

B.3 Recherche kNN dans un arbre-IM

Une fois un arbre-IM construit, il va nous servir à effectuer les recherches de k plus proches voisins.

Grâce à la syntaxe sur les listes et à l'existence des fonctions lambda, l'algorithme 43 est sans doute plus clair que son homologue mathématique en page 113.

Noter que l'on se sert dans ce code de la fonction de base `sorted` plutôt que d'un « k-tri ». Pourtant, même développé en Python, ce dernier se révèle plus rapide que la fonction de tri de base écrite directement en C !

Noter que l'assertion finale dans la branche `else` (quantificateur universel)

Algorithme 43 Recherche kNN dans un arbre-IM en Python

```
def kNNIM (N, q, k, d,
           rq = float('Infinity'), A = []):
    if type(N) is list:
        return sorted(A + [(d(e, q), e) for e in N],
                      key=lambda c: c[0])[0:k]
    else:
        (p1, p2, r, r1, r2, N1, N2, N3, N4, N5) = N
        assert r < d(p1, p2)
        d1 = d(p1, q)
        d2 = d(p2, q)
        rq = min(rq, A[-1][0]) if len(A) == k else rq
        for (Ni, Ci, rmini, rmaxi) in \
            sorted([(N1, lambda rq: d1 <= r + rq and
                                    d2 <= r + rq,
                         max(d1 - r, d2 - r),
                         min(d1 + r, d2 + r)),
                    (N2, lambda rq: d1 <= r + rq and
                                    not d2 + rq <= r,
                         d1 - r,
                         d1 + r),
                    (N3, lambda rq: d2 <= r + rq and
                                    not d1 + rq <= r,
                         d2 - r,
                         d2 + r),
                    (N4, lambda rq: d1 <= r1 + rq and
                                    not d1 + rq <= r and
                                    not d2 + rq <= r,
                         d1 - r1,
                         d1 + r1),
                    (N5, lambda rq: d2 <= r2 + rq and
                                    not d1 + rq <= r and
                                    not d2 + rq <= r,
                         d2 - r2,
                         d2 + r2)],
                    key=lambda t: t[3]):
            if Ci(rq):
                A = kNNIM(Ni, q, k, d, min(rq, rmaxi), A)
                rq = min(rq, A[-1][0]) if len(A) == k else rq
            else:
                assert all([d(e, q) > rq for e in ElementsIM(Ni)])
        return A
```

Algorithme 44 Distance euclidienne en Python

```
from math import sqrt

def L2 (v1, v2):
    return sqrt(sum([(a - b)**2 for (a, b) in zip(v1, v2)]))
```

Algorithme 45 Génération de vecteurs aléatoires en Python

```
from random import randint

def GénérerVecteurs (n, t, inf, sup):
    assert type(n) is int
    assert n >= 0
    assert type(t) is int
    assert t >= 0
    assert inf <= sup
    return [[randint(inf, sup)
             for j in range(0, t)]
             for i in range(0, n)]
```

doit être enlevée lors de tests en grandeur réelle, car elle augmente considérablement le temps d'exécution de l'algorithme. (Elle le rend quadratique.)

B.4 Quelques tests

Nous testons l'arbre-IM sur des données vectorielles.

Pour cela nous définissons tout d'abord comme distance classique la distance euclidienne *via* l'algorithme 44. Noter que la fonction `zip` permet de construire une liste de couples à partir d'un couple de listes (de mêmes tailles).

Nous générons aussi quelques vecteurs de manière aléatoire grâce à la fonction de l'algorithme 45.

À partir de là, tout est prêt pour quelques tests.

Le code de l'algorithme 46 génère et affiche deux arbres à partir des mêmes données aléatoires initiales.

Une exécution particulière donne les vecteurs aléatoires en deux dimensions reportés dans le « tableau » B.1.

Une partie de l'arbre construit par l'algorithme de construction en lot est illustré en figure B.1. Noter que la plupart des nœuds IV et V sont vides. S'ils ne le sont pas tous, c'est que la fonction de détermination des deux éléments les plus éloignés n'est qu'approximative.

De manière similaire, la figure B.2 montre l'arbre obtenu par la version incrémentale de l'algorithme de construction. Noter que le nombre de nœuds IV et V vides a fortement diminué puisque la population entière n'était pas disponible au

Algorithme 46 Génération d'arbre-IM

```
cmax = 5
d = L2

V = GénérerVecteurs(50, 2, 10, 99)
print('Vecteurs aléatoires : ', V)
print('Arbre-IM construit en lot :')
IML = ConstruireIM(V, d, cmax, 0.6)
print("Vérification de l'arbre construit en lot")
VérifierIM(IML, d)
IndenterIM(IML)
print('Arbre-IM construit incrémentalement :')
IMI = ConstruireIMIncrémentalement(V, d, cmax, 0.6)
print("Vérification de l'arbre construit incrémentalement")
VérifierIM(IMI, d)
IndenterIM(IMI)
```

```
[[49, 53], [93, 64], [99, 53], [42, 66], [43, 15],
 [53, 81], [55, 53], [59, 15], [60, 78], [10, 61],
 [11, 75], [10, 61], [54, 67], [80, 77], [19, 41],
 [24, 23], [86, 82], [31, 33], [99, 63], [32, 79],
 [97, 26], [11, 36], [68, 46], [25, 65], [59, 15],
 [90, 37], [52, 63], [50, 81], [30, 51], [14, 65],
 [23, 36], [91, 30], [88, 45], [69, 76], [80, 63],
 [62, 22], [62, 99], [46, 21], [77, 89], [94, 84],
 [21, 75], [96, 40], [63, 64], [72, 78], [53, 64],
 [75, 69], [58, 45], [16, 75], [30, 35], [32, 17]]
```

TABLE B.1 – Cinquante vecteurs bidimensionnels aléatoires

```
p1 :  [94, 84]
p2 :  [11, 36]
r  :  57.52808010006939
r1 :  58.077534382926416
r2 :  57.52808010006939
N1
|  p1 :  [42, 66]
|  p2 :  [58, 45]
|  r  :  15.840454538932901
|  r1 :  15.840454538932901
|  r2 :  15.840454538932901
|  N1
|  |  [[49, 53]]
|  N2
|  |  [[42, 66], [54, 67], [52, 63],
       [53, 64]]
|  N3
|  |  [[55, 53], [58, 45]]
|  N4
|  |  []
|  N5
|  |  []
N2
|  p1 :  [91, 30]
|  p2 :  [62, 99]
|  r  :  44.90790576279415
|  r1 :  44.90790576279415
|  r2 :  44.90790576279415
|  N1
|  |  [[80, 63], [63, 64], [75, 69]]
|  N2
|  |  p1 :  [99, 63]
|  |  p2 :  [68, 46]
|  |  r  :  21.213203435596427
|  |  r1 :  23.194827009486403
|  |  r2 :  28.0178514522438
|  |  N1
|  |  |  [[88, 45]]
|  |  N2
|  |  |  [[93, 64], [99, 53], [99, 63]]
|  |  N3
|  |  |  [[68, 46]]
|  |  N4
|  |  |  [[96, 40]]
|  |  N5
|  |  |  [[90, 37], [91, 30]]
|  N3
|  |  p1 :  [50, 81]
|  |  p2 :  [94, 84]
|  |  r  :  26.461292485439934
|  |  r1 :  26.461292485439934
|  |  r2 :  26.461292485439934
|  |  N1
|  |  |  [[69, 76], [72, 78]]
|  |  N2
|  |  |  [[53, 81], [60, 78], [50, 81],
          [62, 99]]
|  |  N3
|  |  |  [[80, 77], [86, 82], [77, 89],
          [94, 84]]
|  |  N4
|  |  |  []
|  |  N5
|  |  |  []
|  N4
|  |  []
|  N5
|  |  []
N3
|  p1 :  [59, 15]
|  p2 :  [11, 75]
|  r  :  46.10249450951651
|  r1 :  46.10249450951651
|  r2 :  46.10249450951651
|  N1
|  |  [[23, 36], [30, 35]]
|  N2
|  |  p1 :  [62, 22]
|  |  p2 :  [24, 23]
|  |  r  :  22.807893370497855
|  |  r1 :  22.807893370497855
|  |  r2 :  22.807893370497855
|  |  N1
|  |  |  [[43, 15], [46, 21]]
|  |  N2
|  |  |  [[59, 15], [59, 15], [62, 22]]
|  |  N3
|  |  |  [[24, 23], [31, 33], [32, 17]]
|  |  N4
|  |  |  []
|  |  N5
|  |  |  []
|  N3
|  |  p1 :  [32, 79]
|  |  p2 :  [11, 36]
|  |  r  :  28.712366673612955
|  |  r1 :  28.712366673612955
|  |  r2 :  28.712366673612955
|  |  N1
|  |  |  [[10, 61], [10, 61], [30, 51]]
|  |  N2
|  |  |  p1 :  [32, 79]
|  |  |  p2 :  [14, 65]
|  |  |  r  :  13.682105101189654
|  |  |  r1 :  13.682105101189654
|  |  |  r2 :  13.682105101189654
|  |  |  N1
|  |  |  |  [[21, 75]]
|  |  |  N2
|  |  |  |  [[32, 79]]
|  |  |  N3
|  |  |  |  [[11, 75], [25, 65], [14, 65],
             [16, 75]]
|  |  |  N4
|  |  |  |  []
|  |  |  N5
|  |  |  |  []
|  |  N3
|  |  |  [[19, 41], [11, 36]]
|  |  N4
|  |  |  []
|  |  N5
|  |  |  []
|  N4
|  |  []
|  N5
|  |  []
N4
|  [[97, 26]]
N5
|  []
```

FIGURE B.1 – Arbre-IM obtenu à partir des vecteurs du tableau B.1 par l'algorithme 32 de construction en lot

```
p1 :  [93, 64]
p2 :  [43, 15]
r  :  42.004285495649135
r1 :  62.81719509815764
r2 :  68.0
N1
|  [[55, 53], [68, 46], [58, 45]]
N2
|  p1 :  [99, 53]
|  p2 :  [54, 67]
|  r  :  28.276492003075628
|  r1 :  31.78049716414141
|  r2 :  31.827660925679098
|  N1
|  |  [[80, 63]]
|  N2
|  |  p1 :  [93, 64]
|  |  p2 :  [97, 26]
|  |  r  :  22.925967809451357
|  |  r1 :  22.925967809451357
|  |  r2 :  22.925967809451357
|  |  N1
|  |  |  [[88, 45]]
|  |  N2
|  |  |  [[93, 64], [99, 53], [99, 63]]
|  |  N3
|  |  |  [[97, 26], [90, 37], [91, 30],
          [96, 40]]
|  |  N4
|  |  |  []
|  |  N5
|  |  |  []
|  N3
|  |  p1 :  [52, 63]
|  |  p2 :  [80, 77]
|  |  r  :  18.782971010998235
|  |  r1 :  18.782971010998235
|  |  r2 :  18.782971010998235
|  |  N1
|  |  |  []
|  |  N2
|  |  |  [[60, 78], [54, 67], [52, 63],
          [63, 64], [53, 64]]
|  |  N3
|  |  |  [[80, 77], [69, 76], [72, 78],
          [75, 69]]
|  |  N4
|  |  |  []
|  |  N5
|  |  |  []
|  N4
|  |  [[86, 82], [94, 84]]
|  N5
|  |  [[77, 89]]
N3
|  p1 :  [59, 15]
|  p2 :  [19, 41]
|  r  :  28.624465060503752
|  r1 :  28.624465060503752
|  r2 :  32.31098884280702
|  N1
|  |  [[32, 17]]
|  N2
|  |  [[43, 15], [59, 15], [59, 15],
       [62, 22], [46, 21]]
|  N3
|  |  p1 :  [24, 23]
|  |  p2 :  [30, 51]
|  |  r  :  17.181385275931625
|  |  r1 :  18.384776310850235
|  |  r2 :  17.181385275931625
|  |  N1
|  |  |  [[23, 36], [30, 35]]
|  |  N2
|  |  |  [[24, 23], [31, 33]]
|  |  N3
|  |  |  [[19, 41], [30, 51]]
|  |  N4
|  |  |  [[11, 36]]
|  |  N5
|  |  |  []
|  N4
|  |  []
|  N5
|  |  [[49, 53]]
N4
|  [[53, 81], [32, 79], [50, 81],
    [62, 99]]
N5
|  p1 :  [10, 61]
|  p2 :  [42, 66]
|  r  :  19.432961688841974
|  r1 :  19.432961688841974
|  r2 :  19.432961688841974
|  N1
|  |  [[25, 65]]
|  N2
|  |  p1 :  [21, 75]
|  |  p2 :  [10, 61]
|  |  r  :  10.682696288858914
|  |  r1 :  10.682696288858914
|  |  r2 :  10.682696288858914
|  |  N1
|  |  |  []
|  |  N2
|  |  |  [[11, 75], [21, 75], [16, 75]]
|  |  N3
|  |  |  [[10, 61], [10, 61], [14, 65]]
|  |  N4
|  |  |  []
|  |  N5
|  |  |  []
|  N3
|  |  [[42, 66]]
|  N4
|  |  []
|  N5
|  |  []
```

FIGURE B.2 – Arbre-IM obtenu à partir des vecteurs du tableau B.1 par l'algorithme 36 de construction incrémental

Algorithme 47 Recherches kNN aléatoire dans un arbre-IM

```
Q = GénérerVecteurs(10, 2, 10, 99)
k = 3

for q in Q:
    print();
    print('kNN de ', q)
    kNNTri = sorted([(d(e, q), e) for e in V],
                    key=lambda c: c[0])[0:k]
    print('kNN sur tri           : ', kNNTri)
    kNNIML = kNNIM(IML, q, k, d)
    print('kNN sur IM lot  : ', kNNIML)
    kNNIMI = kNNIM(IMI, q, k, d)
    print('kNN sur IM inc. : ', kNNIMI)
    print('kNN sur tri     - distances : ', [d for (d, _) in kNNTri])
    print('kNN sur IM lot  - distances : ', [d for (d, _) in kNNIML])
    print('kNN sur IM inc. - distances : ', [d for (d, _) in kNNIMI])
    if [d for (d, _) in kNNTri] == [d for (d, _) in kNNIML] \
                                == [d for (d, _) in kNNIMI]:
        pass
    else:
        print('*** PROBLEM ***', q)
```

moment du calcul (approximatif) de deux éléments les plus éloignés.

Le code de l'algorithme 47 génère dix requêtes aléatoires et effectue la recherche de trois plus proches voisins sur les deux arbres construits ainsi qu'en utilisant un algorithme de tri standard. Bien entendu, les résultats doivent coïncider au moins sur les distances (il pourrait y avoir des éléments à égale distance d'une requête donnée) et aux erreurs d'arrondis arithmétiques près.

Les résultats, identiques entre eux, sont reportés sur la figure B.3.

B.5 Instrumentation du code

L'« instrumentation » du code consiste à ajouter des capteurs afin de mesurer les performances des algorithmes. Ici, nous voulons compter le nombre d'accès à des objets, le nombre de calculs de distances et le nombre de comparaisons entre distances.

L'instrumentation du code de la recherche est présenté sur l'algorithme 48. On y déclare cinq compteurs en tant que variables globales (mises en évidence par des majuscules). L'algorithme 43 est alors repris en introduisant les incrémentations de ces compteurs, ce qui donne une version « cachée » de la recherche. Dans les nœuds internes, le décompte du nombre de comparaisons de distance se décompose comme suit :

- 1 pour la limitation de r_q avant la boucle ;

```
kNN de  [82, 52]
kNN sur tri     :  [(9.219544457292887, [88, 45]), (11.180339887498949, [80, 63]), (15.231546211727817, [68, 46])]
kNN sur IM lot  :  [(9.219544457292887, [88, 45]), (11.180339887498949, [80, 63]), (15.231546211727817, [68, 46])]
kNN sur IM inc. :  [(9.219544457292887, [88, 45]), (11.180339887498949, [80, 63]), (15.231546211727817, [68, 46])]
kNN sur tri     - distances :  [9.219544457292887, 11.180339887498949, 15.231546211727817]
kNN sur IM lot  - distances :  [9.219544457292887, 11.180339887498949, 15.231546211727817]
kNN sur IM inc. - distances :  [9.219544457292887, 11.180339887498949, 15.231546211727817]

kNN de  [13, 98]
kNN sur tri     :  [(23.08679276123039, [11, 75]), (23.194827009486403, [16, 75]), (24.351591323771842, [21, 75])]
kNN sur IM lot  :  [(23.08679276123039, [11, 75]), (23.194827009486403, [16, 75]), (24.351591323771842, [21, 75])]
kNN sur IM inc. :  [(23.08679276123039, [11, 75]), (23.194827009486403, [16, 75]), (24.351591323771842, [21, 75])]
kNN sur tri     - distances :  [23.08679276123039, 23.194827009486403, 24.351591323771842]
kNN sur IM lot  - distances :  [23.08679276123039, 23.194827009486403, 24.351591323771842]
kNN sur IM inc. - distances :  [23.08679276123039, 23.194827009486403, 24.351591323771842]

kNN de  [13, 17]
kNN sur tri     :  [(12.529964086141668, [24, 23]), (19.0, [32, 17]), (19.1049731745428, [11, 36])]
kNN sur IM lot  :  [(12.529964086141668, [24, 23]), (19.0, [32, 17]), (19.1049731745428, [11, 36])]
kNN sur IM inc. :  [(12.529964086141668, [24, 23]), (19.0, [32, 17]), (19.1049731745428, [11, 36])]
kNN sur tri     - distances :  [12.529964086141668, 19.0, 19.1049731745428]
kNN sur IM lot  - distances :  [12.529964086141668, 19.0, 19.1049731745428]
kNN sur IM inc. - distances :  [12.529964086141668, 19.0, 19.1049731745428]

kNN de  [65, 27]
kNN sur tri     :  [(5.830951894845301, [62, 22]), (13.416407864998739, [59, 15]), (13.416407864998739, [59, 15])]
kNN sur IM lot  :  [(5.830951894845301, [62, 22]), (13.416407864998739, [59, 15]), (13.416407864998739, [59, 15])]
kNN sur IM inc. :  [(5.830951894845301, [62, 22]), (13.416407864998739, [59, 15]), (13.416407864998739, [59, 15])]
kNN sur tri     - distances :  [5.830951894845301, 13.416407864998739, 13.416407864998739]
kNN sur IM lot  - distances :  [5.830951894845301, 13.416407864998739, 13.416407864998739]
kNN sur IM inc. - distances :  [5.830951894845301, 13.416407864998739, 13.416407864998739]

kNN de  [52, 56]
kNN sur tri     :  [(4.242640687119285, [49, 53]), (4.242640687119285, [55, 53]), (7.0, [52, 63])]
kNN sur IM lot  :  [(4.242640687119285, [49, 53]), (4.242640687119285, [55, 53]), (7.0, [52, 63])]
kNN sur IM inc. :  [(4.242640687119285, [55, 53]), (4.242640687119285, [49, 53]), (7.0, [52, 63])]
kNN sur tri     - distances :  [4.242640687119285, 4.242640687119285, 7.0]
kNN sur IM lot  - distances :  [4.242640687119285, 4.242640687119285, 7.0]
kNN sur IM inc. - distances :  [4.242640687119285, 4.242640687119285, 7.0]

kNN de  [40, 93]
kNN sur tri     :  [(15.620499351813308, [50, 81]), (16.1245154965971, [32, 79]), (17.69180601295413, [53, 81])]
kNN sur IM lot  :  [(15.620499351813308, [50, 81]), (16.1245154965971, [32, 79]), (17.69180601295413, [53, 81])]
kNN sur IM inc. :  [(15.620499351813308, [50, 81]), (16.1245154965971, [32, 79]), (17.69180601295413, [53, 81])]
kNN sur tri     - distances :  [15.620499351813308, 16.1245154965971, 17.69180601295413]
kNN sur IM lot  - distances :  [15.620499351813308, 16.1245154965971, 17.69180601295413]
kNN sur IM inc. - distances :  [15.620499351813308, 16.1245154965971, 17.69180601295413]

kNN de  [49, 92]
kNN sur tri     :  [(11.045361017187261, [50, 81]), (11.704699910719626, [53, 81]), (14.7648230602334, [62, 99])]
kNN sur IM lot  :  [(11.045361017187261, [50, 81]), (11.704699910719626, [53, 81]), (14.7648230602334, [62, 99])]
kNN sur IM inc. :  [(11.045361017187261, [50, 81]), (11.704699910719626, [53, 81]), (14.7648230602334, [62, 99])]
kNN sur tri     - distances :  [11.045361017187261, 11.704699910719626, 14.7648230602334]
kNN sur IM lot  - distances :  [11.045361017187261, 11.704699910719626, 14.7648230602334]
kNN sur IM inc. - distances :  [11.045361017187261, 11.704699910719626, 14.7648230602334]

kNN de  [49, 33]
kNN sur tri     :  [(12.36931687685298, [46, 21]), (15.0, [58, 45]), (17.029386365926403, [62, 22])]
kNN sur IM lot  :  [(12.36931687685298, [46, 21]), (15.0, [58, 45]), (17.029386365926403, [62, 22])]
kNN sur IM inc. :  [(12.36931687685298, [46, 21]), (15.0, [58, 45]), (17.029386365926403, [62, 22])]
kNN sur tri     - distances :  [12.36931687685298, 15.0, 17.029386365926403]
kNN sur IM lot  - distances :  [12.36931687685298, 15.0, 17.029386365926403]
kNN sur IM inc. - distances :  [12.36931687685298, 15.0, 17.029386365926403]

kNN de  [56, 28]
kNN sur tri     :  [(8.48528137423857, [62, 22]), (12.206555615733702, [46, 21]), (13.341664064126334, [59, 15])]
kNN sur IM lot  :  [(8.48528137423857, [62, 22]), (12.206555615733702, [46, 21]), (13.341664064126334, [59, 15])]
kNN sur IM inc. :  [(8.48528137423857, [62, 22]), (12.206555615733702, [46, 21]), (13.341664064126334, [59, 15])]
kNN sur tri     - distances :  [8.48528137423857, 12.206555615733702, 13.341664064126334]
kNN sur IM lot  - distances :  [8.48528137423857, 12.206555615733702, 13.341664064126334]
kNN sur IM inc. - distances :  [8.48528137423857, 12.206555615733702, 13.341664064126334]

kNN de  [33, 55]
kNN sur tri     :  [(5.0, [30, 51]), (12.806248474865697, [25, 65]), (14.212670403551895, [42, 66])]
kNN sur IM lot  :  [(5.0, [30, 51]), (12.806248474865697, [25, 65]), (14.212670403551895, [42, 66])]
kNN sur IM inc. :  [(5.0, [30, 51]), (12.806248474865697, [25, 65]), (14.212670403551895, [42, 66])]
kNN sur tri     - distances :  [5.0, 12.806248474865697, 14.212670403551895]
kNN sur IM lot  - distances :  [5.0, 12.806248474865697, 14.212670403551895]
kNN sur IM inc. - distances :  [5.0, 12.806248474865697, 14.212670403551895]
```

FIGURE B.3 – Résultat de dix recherches 3-NN

Algorithme 48 Recherche kNN dans un arbre-IM en Python, version « instrumentée »

```
LEAF_ACCESSES         = 0
INNER_NODE_ACCESSES   = 0
ELEMENT_ACCESSES      = 0
DISTANCE_COMPUTATIONS = 0
DISTANCE_COMPARISONS  = 0

from math import ceil, log

def kNNIMInstrumentée2 (N, q, k, d,
                        rq = float('Infinity'), A = []):
    global LEAF_ACCESSES
    global INNER_NODE_ACCESSES
    global ELEMENT_ACCESSES
    global DISTANCE_COMPUTATIONS
    global DISTANCE_COMPARISONS
    if type(N) is list:
        LEAF_ACCESSES         += 1
        ELEMENT_ACCESSES      += len(N)
        DISTANCE_COMPUTATIONS += len(N)
        DISTANCE_COMPARISONS  += len(N) * ceil(log(k, 2))
        return sorted(A + [(d(e, q), e) for e in N],
                      key=lambda c: c[0])[0:k]
    else:
        INNER_NODE_ACCESSES   += 1
        ELEMENT_ACCESSES      += len(N)
        DISTANCE_COMPUTATIONS += 2
        DISTANCE_COMPARISONS  += 1 + \
                                 2 + 2 + 2 + 3 + 3 + \
                                 2 + \
                                 5 * ceil(log(5, 2)) + \
                                 2 * 5
        (p1, p2, r, r1, r2, N1, N2, N3, N4, N5) = N
        d1 = d(p1, q)
        d2 = d(p2, q)
        rq = min(rq, A[-1][0]) if len(A) == k else rq
        for (Ni, Ci, rmini, rmaxi) in
            sorted([(N1, lambda rq: d1 <= r + rq and
                                    d2 <= r + rq,
                         max(d1 - r, d2 - r),
                         min(d1 + r, d2 + r)),
                    ...
                    (N5, lambda rq: d2 <= r2 + rq and
                                    not d1 + rq <= r and
                                    not d2 + rq <= r,
                         d2 - r2,
                         d2 + r2)],
                   key=lambda t: t[3]):
            if Ci(rq):
                A = kNNIMInstrumentée2(Ni, q, k, d, min(rq, rmaxi), A)
                rq = min(rq, A[-1][0]) if len(A) == k else rq
        return A
```

Algorithme 49 Recherche kNN dans un arbre-IM en Python, version « instrumentée » englobante

```
def kNNIMInstrumentée (N, q, k, d,
                       rq = float('Infinity'), A = []):
    global LEAF_ACCESSES
    global INNER_NODE_ACCESSES
    global ELEMENT_ACCESSES
    global DISTANCE_COMPUTATIONS
    global DISTANCE_COMPARISONS
    LEAF_ACCESSES         = 0
    INNER_NODE_ACCESSES   = 0
    ELEMENT_ACCESSES      = 0
    DISTANCE_COMPUTATIONS = 0
    DISTANCE_COMPARISONS  = 0
    result = kNNIMInstrumentée2(N, q, k, d, rq, A)
    print('AccesFeuille', LEAF_ACCESSES)
    print('AccesNoeuds', INNER_NODE_ACCESSES)
    print('AccesElements', ELEMENT_ACCESSES)
    print('CalculDistance', DISTANCE_COMPUTATIONS)
    print('ComparaisonDistance', DISTANCE_COMPARISONS)
    return result
```

- $2+2+2+3+3$ (au pire) pour les tests associés aux fonctions lambda (certains étant redondants par rapport à une implémentation plus efficace et certains étant également court-circuités à l'exécution) ;
- 2 pour le calcul des distances minimales et maximales à l'intersection des deux boules dans le cas associé à N_1 ;
- $5 \times \lceil \log_2 5 \rceil$ pour le tri des fils ;
- 2×5 pour les calculs des minima à l'intérieur de la boucle (certains n'ayant pas lieu puisque la condition peut être fausse).

Dans cet algorithme nous déterminons donc un *majorant* à la complexité effective.

Cette version peut être appelée à partir d'une version qui réinitialise les compteurs avant chaque recherche (cf. algorithme 49).

Sur un test dont les paramètres sont :

- génération de 1 000 vecteurs de dimension 4 ;
- $c_{\max} = 100$;
- $d = L_2$;
- $\alpha = 0,6$;
- génération de 100 requêtes ;

- $k = 5$;

nous obtenons un arbre-IM constitué de 33 feuilles et 8 nœuds internes.

Les cardinaux des feuilles sont donnés sur le tableau B.2. On constate de nouveau que les feuilles des régions IV et V contiennent peu d'éléments comparées aux autres régions. On remarque aussi que l'arbre, d'une profondeur maximale de trois, est relativement bien équilibré. Les feuilles sont en moyenne remplies à un tiers de leur capacité maximale.

L'exécution de la centaine de requête donne les résultats reportés sur le tableau B.3. Bien que le cardinal des feuilles ait été choisi très grand (100 vis-à-vis de 1 000 au lieu de 31 pour la racine carrée ou de 10 pour le logarithme), bien qu'une génération aléatoire uniforme ne permette pas de regrouper de manière significative les éléments, nous constatons que la recherche des cinq plus proches voisins est quatre fois plus rapides, en moyenne, que le parcours intégral avec l'algorithme « *k*-tri », et n'atteint pas la moitié dans le pire des cas.

Cela termine cette annexe présentant de manière très succincte la méthodologie de développement et de test.

(a) dans un parcours en préordre de l'arbre

Feuille	Cardinal
11	25
12	34
13	51
14	0
15	0
21	53
221	32
222	46
223	78
224	0
225	1
231	36
232	81
233	55
234	0
235	0
24	7
25	24
31	55
321	37
322	71
323	70
324	1
325	0
331	18
332	55
333	54
334	2
335	0
34	12
35	4
4	56
5	42

(b) dans l'ordre décroissant des cardinaux

Feuille	Cardinal
232	81
223	78
322	71
323	70
4	56
233	55
31	55
332	55
333	54
21	53
13	51
222	46
5	42
321	37
231	36
12	34
221	32
11	25
25	24
331	18
34	12
24	7
35	4
334	2
225	1
324	1
14	0
15	0
224	0
234	0
235	0
325	0
335	0

TABLE B.2 – Cardinal des feuilles d'un arbre-IM de 1 000 vecteurs de dimension 4 construit incrémentalement avec $c_{\max} = 100$ et $\alpha = 0,6$

Requête	#feuilles	#nœuds internes	#éléments	#distances	#comparaisons	TOTAL
[4009, 7266, 8374, 4092]	2	2	79	63	259	511
[5151, 6576, 8709, 6236]	2	2	79	63	259	511
[4919, 6489, 5080, 3972]	2	2	79	63	259	511
[3097, 7665, 1903, 4068]	2	2	79	63	259	511
[4986, 7193, 2675, 8973]	3	4	101	69	347	623
[5989, 5250, 2065, 6528]	2	2	96	80	310	630
[3098, 7984, 3566, 5574]	3	4	154	122	506	994
[4005, 8967, 3618, 3866]	4	4	172	140	560	1120
[3652, 9943, 6402, 3240]	5	4	172	140	560	1120
[4196, 8446, 1517, 5674]	4	4	172	140	560	1120
[4140, 8327, 1558, 8751]	4	4	172	140	560	1120
[3428, 9814, 6752, 9200]	5	4	172	140	560	1120
[4308, 9069, 7005, 3875]	4	4	172	140	560	1120
[3112, 9620, 8557, 8484]	4	4	172	140	560	1120
[2486, 7554, 8598, 4687]	3	4	177	145	575	1155
[2959, 6828, 7791, 8859]	3	4	177	145	575	1155
[3518, 6726, 2562, 5938]	3	4	177	145	575	1155
[3340, 6934, 1703, 9442]	3	4	177	145	575	1155
[2937, 7513, 5135, 1547]	3	4	177	145	575	1155
[7983, 4515, 9777, 2849]	3	4	186	154	602	1218
[8400, 4445, 6957, 1050]	3	4	186	154	602	1218
[8451, 4536, 8120, 3798]	3	4	186	154	602	1218
[8735, 4343, 9956, 6394]	3	4	186	154	602	1218
[6755, 7380, 2147, 8715]	6	3	195	171	618	1302
[6957, 7684, 3292, 7284]	6	3	195	171	618	1302
[6450, 6832, 1931, 3269]	6	3	195	171	618	1302
[5531, 1417, 1298, 9454]	5	4	205	173	659	1351
[9169, 6602, 5057, 2744]	5	4	208	176	668	1372
[1968, 6663, 5276, 3051]	4	4	209	177	671	1379
[1719, 6738, 3663, 9978]	4	4	209	177	671	1379
[9306, 4215, 9511, 7797]	4	4	223	191	713	1477
...						
[5993, 8045, 4792, 7944]	8	4	277	245	875	1855
[8494, 6171, 6124, 9502]	6	4	278	246	878	1862
[1345, 3133, 5984, 7177]	9	4	280	248	884	1876
[1601, 5096, 8736, 3868]	8	4	281	249	887	1883
[1315, 3769, 2872, 8376]	8	4	281	249	887	1883
[1765, 5176, 1769, 9148]	8	4	281	249	887	1883
[9923, 2544, 1869, 8891]	6	4	284	252	896	1904
[9195, 3137, 8248, 9871]	5	4	284	252	896	1904
[6261, 1956, 1426, 9819]	6	4	286	254	902	1918
[6228, 2346, 8267, 5387]	6	4	286	254	902	1918
[6287, 2749, 6034, 1879]	5	4	286	254	902	1918
[6196, 3425, 9054, 4023]	5	4	286	254	902	1918
[9035, 9822, 5338, 5425]	8	4	288	256	908	1932
[9092, 9217, 4390, 9996]	8	4	288	256	908	1932
[8273, 1316, 9227, 8050]	6	4	289	257	911	1939
[3026, 1084, 3342, 8371]	8	4	294	262	926	1974
[4409, 3631, 7780, 8078]	8	4	294	262	926	1974
[3844, 2615, 3063, 9502]	8	4	294	262	926	1974
[4498, 3281, 1046, 5271]	8	4	294	262	926	1974
[4587, 1205, 9623, 5434]	8	4	294	262	926	1974
[9217, 2037, 1242, 1644]	6	4	295	263	929	1981
[8687, 7444, 3614, 9473]	8	4	313	281	983	2107
[9960, 7338, 9205, 2794]	8	4	313	281	983	2107
[8729, 7627, 5872, 7391]	9	4	313	281	983	2107
[5016, 3551, 9049, 9327]	8	4	320	288	1004	2156
[1386, 1475, 6145, 6621]	10	5	345	305	1090	2310
[2433, 2462, 3548, 3113]	10	5	345	305	1090	2310
[1355, 1926, 7776, 5389]	10	5	345	305	1090	2310
[8631, 9788, 4816, 7853]	10	5	356	316	1123	2387
[8572, 9557, 1481, 3009]	10	5	356	316	1123	2387
[3466, 4978, 3024, 1458]	11	4	360	328	1124	2436
[9658, 2622, 8084, 4872]	7	6	385	337	1221	2569
min	2	2	79	63	259	511
moyenne	5,82	3,94	242,26	210,74	770,12	1 613,08
écart-type	2,12	0,57	63,92	60,76	196,26	439,01
médiane	6	4	252	220	800	1 680
max	11	6	385	337	1 221	2 569

TABLE B.3 – Extrait de la recherche de 100 vecteurs aléatoires dans un arbre-IM avec $k = 5$ ordonnés suivant la complexité totale croissante, à comparer à une complexité d'un « *k*-tri » égale à $1\,000 \times 4 + 1\,000 \times \log_2 5 \simeq 6\,322$

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Zineddine KOUAHLA

Indexation dans les espaces métriques

Index arborescent et parallélisation

Metric Spaces Indexing

Tree-based Index and Parallelism

Résumé
L'indexation et la recherche efficiente de données complexes constitue un besoin croissant face à la taille et à la variété des bases de données actuelles. Nous proposons une structure d'index arborescent basée sur un partitionnement d'un espace métrique à base de boules et d'hyper-plans. Les performances de cet index sont évaluées expérimentalement sur des collections de complexités intrinsèques différentes. La parallélisation de l'algorithme de recherche des *k* plus proches voisins est également effectuée afin d'encore améliorer les performances.

Mots clés
Indexation, *k* plus proches voisins, espaces métriques, parallélisme.

Abstract
The efficient indexing and searching of complex data is an increasing need in order to face the size and diversity of current databases. We introduce a tree-based indexing technique that partitions a metric space thanks to balls and hyper-planes. The performances of this index structure are experimentally evaluated on chosen datasets presenting different intrinsic difficulties. Also, we parallelise the *k* nearest neighbour search algorithm in order to further improve the performances.

Key Words
Indexing, *k* nearest neighbours, metric spaces, parallelism.