# Construction Incrémentale de Graphes de Voisinage avec Accès Réduit aux Disques

Hakim Hacid*, Tetsuya Yoshida**

*Alcatel-Lucent Bell Labs France
Centre de Villarceaux Route de Villejust, 91620 Nozay, France
hakim.hacid@alcatel-lucent.com,
**Hokkaido University,
Graduate School of Information Science and Technology,
N-14 W-9, Sapporo 060-0814, Japan
yoshida@meme.hokudai.ac.jp

**Résumé.** La recherche efficace de voisinage dans un espace multidimensionnel a été étudiée dans plusieurs domaines tels que la reconnaissance de formes et l'exploration de données. Les graphes de voisinage sont des structures géométriques basées sur le concept de proximité des éléments de données, et ils peuvent être utilisés pour déterminer les plus proches voisins de données multidimensionnelles. Le problème est que, outre la construction couteuse de ces graphes, leur mise à jour est aussi difficile en raison de l'insertion ou la suppression d'un élément. Dans cet article nous proposons d'adapter les graphes de voisinage pour l'indexation de plus grandes quantités de données multidimensionnelles. Nous proposons une modification d'une méthode existante de mise à jour locale de graphes de voisinage de telle sorte à ce qu'elle puisse considérer des accès réduits aux disques afin de prendre en compte l'évolution des bases de données. Cette proposition est étendue pour faire de la construction incrémentale de graphes de voisinage. Des évaluations de l'approche ont été effectuées et les résultats montrent que notre approche est prometteuse.

## 1 Introduction

Une des exigences pour traiter des données multidimensionnelles à grande échelle telles que des données multimédias, des données temporelles, des données spatiales, etc., est de fournir une solution efficace pour rechercher le voisinage. En plus de l'hétérogénéité des données, leur grande quantité et leur haute dimensionnalité rendent cette tache de recherche assez difficile. Rechercher le voisinage efficacement est une question clé dans le traitement automatique de données multidimensionnelles. Divers algorithmes et techniques dans différents domaines de recherche ont été proposés. Par exemple, *k-means* (MacQueen (1967)), Cartes de Kohonen (Somervuo and Kohonen (1999)), et *k-NN* sont des solutions largement utilisées dans l'exploration de données ou dans la majorité des techniques d'indexation de bases de données (Gaede and Günther (1998)).

Parmi les modèles proposés pouvant faire face au probléme de la recherche de voisinage dans un espace multidimensionnel, les graphes de voisinage (Preparata and Shamos (1985); Toussaint (1980); Gabriel and Sokal (1969)) sont illustratifs. Cependant, outre la construction de graphes de voisinage, leur mise à jour, en raison de l'insertion ou la suppression d'éléments de données, peut aussi devenir coûteuse. Une méthode de mise à jour locale a été proposée dans (Hacid and Zighed (2005)) pour atténuer ce problème. Cette méthode construit une hyper-sphère pouvant localiser les données pouvant être concernées par la mise à jour du graphe. Toutefois, ce travail n'a pas explicitement adressé le cas où la taille des données devient très importante. Ce problème est d'autant plus important que les données ne rentrent pas forcément en mémoire centrale, nécessitant aussi des accès disques supplémentaires pouvant réduire le rendement de la méthode ainsi proposée. Il est donc nécessaire de réduire les accès aux disques pour garantir un certain niveau de qualité dans la mise à jour, voire dans la construction du graphe.

Nous proposons dans ce travail d'utiliser un graphe de voisinage comme une structure d'indexation d'une base de données multidimensionnelle. Pour faire face à des masses de données qui dépassent la taille de la mémoire principale, nous construisons un graphe de voisinage pour un jeu de données et nous chargeons uniquement la structure du graphe, et non l'ensemble des données, dans la mémoire principale. Sur la base de cette structure, nous proposons (i) une modification de la méthode de mise à jour locale du graphe de voisinage de sorte à ce qu'elle puisse prendre en compte l'accès réduit aux disques, et (ii) une méthode incrémentale de construction de graphes de voisinage intégrant un accès réduit aux disques. Afin de réduire les accès aux disques, nous définissons et utilisons une limite supérieure pour la méthode proposée dans (Hacid and Zighed (2005)) dans la construction d'une hyper-sphère pour situer la mise à jour. Nous avons en outre élaboré des mécanismes pour faire face à plusieurs problèmes avec la borne supérieure, et proposé d'autres limites pour la méthode de mise à jour. Plusieurs évaluations ont été menées pour évaluer l'approche proposée et les résultats des expériences montrent l'intérêt de notre approche.

Le reste de cet article est organisé comme suit : la Section 2 introduit brièvement différents types de graphes de voisunage. La Section 3 décrit les problèmes des approches existentes, et détaille la méthode proposée pour réaliser la mise à jour locale avec accès limités aux disques. Les évaluations sont rapportées dans la Section 4. Enfin, nous concluons et présentons quelques perspectives dans la Section 5.

# 2 Graphes de voisinage pour la recherche de voisinage

La recherche de voisinage est largement utilisée dans les systèmes de gestion de bases de données pour effectuer un prétraitement des données afin de localiser les données les plus proches les unes des autres. Ainsi, une fois ce voisinage identifié, il devient facile d'interroger et retrouver des données lors d'une recherche d'informations par exemple.

## 2.1 Notations et définitions

Tout au long de cet article, nous utiliserons des lettres en majuscule gras pour dénoter les ensembles d'objets (données). Soit $\boldsymbol{V}$ un ensemble d'objets dans un espace à $p$-dimensions

$\boldsymbol{R}^p$ où $\boldsymbol{R}$ représente l'ensemble des nombres réels. Pour un ensemble $\boldsymbol{V}$, $|\boldsymbol{V}|$ représente sa cardinalité.

Un graphe $G(\boldsymbol{V}, \boldsymbol{E})$ consiste en un ensemble de nœuds $\boldsymbol{V}$ et un ensemble d'arrêtes $\boldsymbol{E}$ définies sur $\boldsymbol{V} \times \boldsymbol{V}$. L'ensemble $\boldsymbol{E}$ peut être interprété comme une représentation d'une relation binaire sur l'ensemble $\boldsymbol{V}$. Un couple de nœuds $(v_i, v_j)$ est une relation binaire définie par le graphe $G(\boldsymbol{V}, \boldsymbol{E})$ si et seulement si la paire $(v_i, v_j) \in \boldsymbol{E}$. Intuitivement, chaque individu est traduit en nœud dans le graphe $G$ et une paire de noeuds $(v_i, v_j)$ est dans $\boldsymbol{E}$ si et seulement si ils sont *directement* connectés par un lien dans le graphe. La fonction $N(v)$ retourne l'ensemble des nœuds connectés au nœud $v \in \boldsymbol{V}$. Formellement, $N(v) = \{w | w \in \boldsymbol{V} \wedge (v, w) \in \boldsymbol{E}\}$.

Nous considérons que la fonction de distance

$$d : \boldsymbol{V} \times \boldsymbol{V} \rightarrow \boldsymbol{R}^+ \cup \{0\} \tag{1}$$

peut être spécifiée pour les données à traiter en se basant sur sur une mesure de distance (Toussaint (1991)). Dans ce qui suit, nous considérons que la fonction de distance $d$ est spécifiée et nous y référons explicitement comme $d$ dans la suite du manuscrit.

Les graphes de voisinage, ou graphes de proximité, sont des structures géométriques qui sont définies en se basant sur la proximité mutuelle entre une paire d'objets. Formellement, une fonction de proximité $f_{prox} : \boldsymbol{V} \times \boldsymbol{V} \rightarrow \{1, 0\}$ est définie par rapport aux caractéristiques des données à traiter et le type du voisinage considéré.

**Definition 1 (Graphe de voisinage)** *Supposons un ensemble de données $\boldsymbol{V}$ et une fonction de proximité $f_{prox} : \boldsymbol{V} \times \boldsymbol{V} \rightarrow \{1, 0\}$. Un graphe de voisinage $G(\boldsymbol{V}, \boldsymbol{E})$ pour $\boldsymbol{V}$ est défini comme suit :*

$$(v_i, v_j) \in \boldsymbol{E} \quad Ssi \quad f_{prox}(v_i, v_j) = 1 \tag{2}$$

*pour toute paire de nœuds $v_i, v_j \in \boldsymbol{V}$, $v_i \neq v_j$.*

Intuitivement, une paire de nœuds $(v_i, v_j)$ forme une arête dans le graphe de voisinage $G(\boldsymbol{V}, \boldsymbol{E})$ pour l'ensemble $\boldsymbol{V}$ si les nœuds respectent la contrainte de proximité qui est représentée par $f_{prox}$.

## 2.2 Exemple de graphe de voisinage

Même pour le même ensemble de données $\boldsymbol{V}$, nous pouvons définir différents types de graphes de voisinage en utilisant différentes fonctions de proximité. Nous introduisons dans ce qui suit les définitions suivantes.

**Definition 2 (Hyper-sphère $H(v_i, v_j)$)** *Une hyper-sphère dans $\boldsymbol{R}^p$, centerée sur $v_i$ avec un rayon $r = d(v_i, v_j)$, est dénotée par $H(v_i, v_j)$. Pour tout objet $w$ in $\boldsymbol{R}^p$,*

$$w \in H(v_i, v_j) \quad Ssi \quad d(v_i, w) \leq r \tag{3}$$

Nous pouvons considérer d'autres types de régions pour une paire de nœuds dans $\boldsymbol{V}$. Par exemple, nous pouvons considérer une "lunule" comme suit :

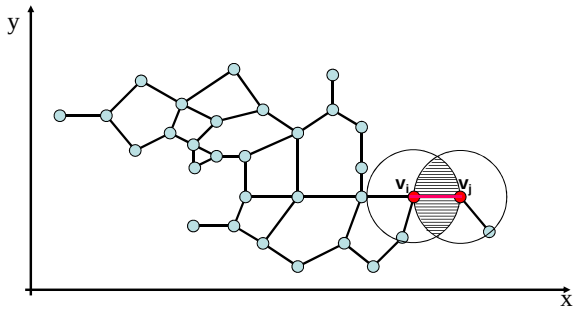


FIG. 1: Un exemple d'un graphe de voisins relatifs dans un plan.

**Definition 3 (Lunule $A(v_i, v_j)$)** *Pour tout $v_i$, $v_j \in \boldsymbol{R}^p$, une lunule $A(v_i, v_j)$ in $\boldsymbol{R}^p$ est définie comme l'intersection de deux hyper-sphères $H(v_i, v_j)$ et $H(v_j, v_i)$. Formellement :*

$$A(v_i, v_j) = H(v_i, v_j) \cap H(v_j, v_i) \tag{4}$$

*Pour tout individu $w$ dans $\boldsymbol{R}^p$, $w$ est à l'intérieur de la lunule $A(v_i, v_j)$ si et seulement si $(d(v_i, w) \leq d(v_i, v_j)) \wedge (d(v_j, w) \leq d(v_i, v_j))$ .*

En considérant la définition précédente d'une lunule, nous pouvons ainsi considérer un autre type de graphes de voisinage appelé "le graphe des voisins relatifs" (RNG) (Jaromczyk and Toussaint (1992)).

**Definition 4 (Graphe des voisins relatifs)** *Pour un ensemble de données $\boldsymbol{V}$ et une fonction de distance spécifiée $d$, un graphe de voisins relatifs est défini en se basant sur la fonction de proximité suivante : $f_{RNG} : \boldsymbol{V} \times \boldsymbol{V} \rightarrow \{1, 0\}$*

$$f_{RNG}(v_i, v_j) = \begin{cases} 1 & A(v_i, v_j) \cap \boldsymbol{V} = \phi \\ 0 & sinon \end{cases} \tag{5}$$

Ainsi, un individu $v_i \in \boldsymbol{V}$ est voisin d'un autre nœud $v_j \in \boldsymbol{V}$ dans un RNG si et seulement si aucun autre nœud dans $\boldsymbol{V}$ n'est à l'intérieur de la lunule $A(v_i, v_j)$. La Figure 1 illustre un exemple d'un graphe de voisins relatifs dans un espace bi-dimensionnel.

## 2.3 Principe des algorithmes de construction des graphes de voisinage

Plusieurs algorithmes pour la construction des graphes de voisinage ont été proposés. Les algorithmes que nous citons ci-après concernent la construction du graphe des voisins relatifs étant donné que c'est la structure principale dans nos travaux. L'une des approches communes aux différents algorithmes est l'utilisation des techniques de raffinement. Dans ce type d'approches, le graphe est construit par étapes. Chaque graphe est construit à partir d'un graphe précédent, contenant toutes les connexions, en éliminant un certain nombre d'arêtes qui ne vérifient pas la propriété de voisinage du graphe à construire. L'élagage (élimination des arrêtes) se fait généralement en tenant compte de la fonction de construction du graphe ou à travers des propriétés géométriques.

Le principe de construction des graphes de voisinage consiste à chercher pour chaque point si les autres points de l'espace sont dans son voisinage. Le coût de cette opération est d'une complexité de $O(n^3)$($n$ étant le nombre de points dans l'espace). Toussaint (Toussaint (1991)) a proposé un algorithme de complexité $O(n^2)$. En utilisant les voisins géographiques (Octant neighbors), (Katajainen (1988)) a proposé également un algorithme de complexité $O(n^2)$. Smith (Smith (1989)) a proposé un algorithme de complexité $O(n^{23/12})$.

Ces algorithmes sont de moindre complexité comparés à l'algorithme standard ($< O\left(n^3\right)$ ). Toutefois, ces derniers se concentrent sur la construction globale du graphe. C'est pourquoi ces algorithmes ne supportent pas et n'offrent pas des opérations d'insertion et de suppression de points dans le graphe. Or, ces opérations sont vitales pour une structure d'indexation, c'est pour cela que ces algorithmes ne sont pas non plus souhaitables pour l'utilisation du graphe comme structure d'indexation.

Ainsi, étant donné qu'un index de données modernes doit supporter beaucoup d'interactions, traduites par des insertions, des suppressions et des interrogations fréquentes, nous proposons dans ce qui suit des algorithmes capables de prendre en considération ces interactions, tout en préservant les propriétés initiales du graphe nécessaires pour l'application de techniques de data mining.

# 3 Construction de graphes avec un accès réduit aux disques

Avant de présenter et de discuter la proposition de cet article, nous allons tout d'abord faire un rappel de l'approche initiale de mise à jour locale de graphes de voisinage basée sur le repérage des individus potentiellement concernés par l'opération de mise à jour.

## 3.1 Approche initiale de mise à jour locale

L'approche initiale de mise à jour locale de graphes de voisinage (Hacid and Zighed (2005)) construit un graphe de voisinage $G(\boldsymbol{V}, \boldsymbol{E})$ en se basant sur la localisation de l'individu inséré ainsi que les arêtes potentiellement affectées par l'insertion. Cette méthode réalise ceci en deux étapes principales :

**1)** déterminer une région dans $\boldsymbol{R}^p$ (une hyper-sphère $SR$ avec un rayon $r$ centrée sur l'individu inséré $q$) qui contiendrait potentiellement les objets les plus proches de $q$.

**2)** calculer la mise à jour de $G$ et appliquer les modifications sur $\boldsymbol{E}$.

L'hyper-sphère dans (1) est formalisée comme suit :

**Definition 5 (Hyper-sphère $SR(q,r)$)** *Une hyper-sphère $SR(q,r)$ est définie comme celle centrée sur $q \in \boldsymbol{R}^p$ avec un rayon $r$. pour tout objet $w$ in $\boldsymbol{R}^p$,*

$$w \in SR(q,r) \;\; ssi \;\; d(q,w) \leq r \tag{6}$$

Le rayon $r$ de l'hyper-sphère $SR$ est déterminé de telle sorte que tous les voisins du plus proche voisin de $q$ soient inclus dans $SR$. La mise à jour de $\boldsymbol{E}$ est conduite ensuite dans (2) entre les individus récupérés via la phase précédente.

Le rayon et l'hyper-sphère pour une requête $q$ sont déterminés comme suit (Hacid and Zighed (2005)). Soit $v_{q1}$ le *plus proche voisin* de $q$, et soit $v_{q2}$ le voisin le *plus éloigné* de $v_{q1}$ avec leurs distances respectives. Ainsi, le rayon $r_{nf}$ est défini comme suit :

$$d_1 = d(q, v_{q1}) \quad (7)$$
$$d_2 = d(v_{q1}, v_{q2}) \quad (8)$$
$$r = (d_1 + d_2)(1 + \epsilon) \quad (9)$$

où $\epsilon \in [0, 1]$ est un paramètre de relaxation qui peut être initialisé en fonction de la connaissance des données (la dispersion par exemple) ou en se basant sur la connaissance du domaine. Dans ce qui suit, les valeurs concrètes de $r$ et de $SR$ sont représentées en utilisant les suffixes adéquats comme : $r_{nf}$ et $SR_{nf}$.

La complexité de la méthode proposée dans (Hacid and Zighed (2005)) est de $O(n + n'^3)$ où $n$(=$|\boldsymbol{V}|$) est le nombre d'objets dans la base de données, $n'$ est le nombre d'objets dans $SR_{nf}$ ($n' \ll n$). Les termes correspondent à : 1) la recherche du rayon $r_{nf}$ de $SR_{nf}$, qui nécessite $O(n)$, et 2) l'ajout et la suppression d'arêtes $SR_{nf}$, qui nécessite $O(n'^3)$. Le deuxième terme correspond au temps requis pour la mise à jour de $\boldsymbol{E}$ dans $SR_{nf}$. Le traitement de (2) est réalisé selon la méthode standard de construction de graphes de voisinage.

Dans cet article, nous appelons la fonction LocalInsert($G(\boldsymbol{V}, \boldsymbol{E})$,$q$,$\epsilon$) la méthode prposée dans (Hacid and Zighed (2005)). Elle retourne un graphe en insérant un élément de la requête $q$ dans $G(\boldsymbol{V}, \boldsymbol{E})$ avec le paramètre $\epsilon$. Plusieurs expériences ont été menées et les résultats vérifient que LocalInsert ($G(\boldsymbol{V}, \boldsymbol{E})$,$q$, $\epsilon$) construit le graphe correctement pour les données en question, i.e. $\boldsymbol{V}$ et $q$ (Hacid and Zighed (2005)).

Le problème majeur de l'approche précédente est qu'elle procède en essayant de trouver tout d'abord le plus proche voisin $v_{q1}$ pour un élément requête $q$ en vérifiant tous les objets. Lorsqu'on est en présence d'une grosse grosse masse de données, cela peut aussi prendre beaucoup de temps. Ainsi, il est souhaitable de réduire le nombre d'accès aux disques pour supporter l'évolution de la mise à jour. La section suivante décrit notre solution pour traiter ce problème.

## 3.2 Passage à l'échelle avec réduction des accès disques

Avec la quantité de plus en plus croissante des données disponibles dans les bases de données, une méthode d'indexation doit prendre en considération l'aspect accès disques. Les méthodes d'insertion et de suppression locales telles qu'elles ont été présentées dans la section précédente sont efficaces pour la mise à jour locale et la construction rapides d'un graphe de voisinage. Toutefois, quand la masse de données est trop importante, les performances de ces méthodes peuvent être affectées. Pour permettre à ces méthodes d'offrir de bonnes performances lorsque nous traitons de grandes bases de données, il est nécessaire de réduire au maximum leur dépendance de la taille globale de ces dernières. Ainsi, nous allons présenter dans ce qui suit d'autres améliorations visant à rendre ces méthodes les moins dépendantes possible de la base de données. Pour cela, nous nous basons sur le calcul de certains indicateurs statistiques sur l'ensemble du graphe.

Le problème auquel nous voulons faire face ici est le cas où la taille des données est très importante et que celles-ci ne tiennent pas en mémoire centrale. Pour cela, il est important et

nécessaire de considérer l'accès aux données sur les mémoires secondaires qui est généralement une opération très coûteuse comparée aux accès à la mémoire centrale.

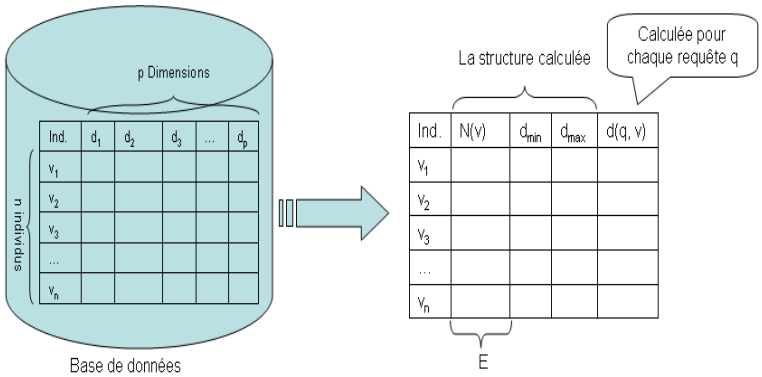


FIG. 2: Illustration du principe général de la réduction des accès disques

Dans la Figure 2, les données de la base de données sont représentées de manière compacte sous forme de graphe de voisinage. En plus du calcul et du stockage du graphe, nous calculons aussi deux indicateurs statistiques représentés par les colonnes $d_{min}$ et $d_{max}$. Ces deux colonnes calculent la distance minimale (resp. maximale) entre chaque individu $v$ donné et tous ses voisins $N(v)$ dans le graphe $G(\boldsymbol{V}, \boldsymbol{E})$. Ces deux paramètres sont calculés pour une utilisation ultérieure. La colonne $d(q, v)$ est utilisée, quant à elle, pour stocker la distance d'un point requête (à son arrivée dans le système) par rapport à tous les individus de la base de données. Les autres indicateurs sont calculés indépendamment de la requête.

A l'arrivée d'un point requête, les valeurs de la colonne $d(q, v)$ sont calculées pour une utilisation ultérieure. Il est nécessaire à ce niveau d'accéder à la base de données pour effectuer ces calculs. Toutefois, une fois cette distance calculée, il n'est plus nécessaire d'accéder à la base de données et il suffira ainsi d'utiliser la structure ainsi stockée en mémoire pour des traitements ultérieurs.

### 3.2.1 Insertion locale dans un graphe avec un accès réduit aux disques

Comme cela a été mentionné plus haut, la méthode d'insertion locale dans sa version initiale nécessite deux accès à la base de données (représentés par le paramètre $2n$ dans l'expression de sa complexité). Le premier accès est nécessaire pour repérer le plus proche voisin du point requête, tandis que le deuxième sert pour la recherche des individus contenus dans l'hyper sphère $SR$. Quand la taille de la base de données augmente, ceci peut avoir un effet négatif sur les performances de la méthode, c'est pourquoi la réduction de ces accès est nécessaire.

Pour atteindre cet objectif, nous utilisons des limites (supérieures et inférieures) pour déterminer le rayon de l'hyper sphère. L'idée est la suivante : en construisant le graphe $G(\boldsymbol{V}, \boldsymbol{E})$, en plus de la structure du graphe, nous récupérons aussi deux valeurs : $\bar{d}_{min}$ représentant la plus grande distance entre un point et son plus proche voisin dans toutes les connexions du graphe [1], et $\bar{d}_{max}$ représentant la plus grande distance entre un point et son plus loin voisin

1. Cette valeur constitue ainsi une borne supérieure de $d_1$ dans la formule du rayon de la formule 7.

dans toutes les connexions du graphe [2]. Formellement, nous aurons :

$$\bar{d}_{min} = \underset{v_i \in \boldsymbol{V}}{\text{argmax}} \ \underset{v_j \in N(v_i)}{\text{argmin}} \ d(v_i, v_j) \quad (10)$$

$$\bar{d}_{max} = \underset{v_i \in \boldsymbol{V}}{\text{argmax}} \ \underset{v_j \in N(v_i)}{\text{argmax}} \ d(v_i, v_j) \quad (11)$$

A partir de cette définition, nous pouvons extraire les propriétés servant de correspondance entre les paramètres de la méthode de base en ayant un point requête en entrée. Ces propriétés sont présentées ci-après :

**Property 1** $\forall v_i, v_j \in V, v_i \neq v_j$,

$$d_1 = d(q, v_{q1}) \leq \bar{d}_{min} \quad (12)$$

$$d_2 = d(v_{q1}, v_{q2}) \leq \bar{d}_{max} \quad (13)$$

En nous basant sur cette propriété, nous pouvons redéfinir le rayon de l'hypersphère délimitant l'espace de recherche. En intégrant ces nouveaux paramètres, l'équation est la suivante :

$$r_{\bar{d}} = (\bar{d}_{min} + \bar{d}_{max})(1 + \epsilon) \quad (14)$$

En faisant toujours un parallèle avec la méthode de base et en considérant la définition de $r_{\bar{d}}$ dans l'équation (14), la propriété suivante est alors vérifiée :

**Property 2** $\forall q \in V$,

$$\begin{aligned} r_{nf} &= (d_1 + d_2)(1 + \epsilon) \\ &\leq (\bar{d}_{min} + \bar{d}_{max})(1 + \epsilon) = r_{\bar{d}} \end{aligned} \quad (15)$$

En utilisant $\bar{d}_{min}$ et $\bar{d}_{max}$, nous pouvons calculer $r_{\bar{d}}$, qui peut être considéré comme une borne supérieure pour $r_{nf}$ pour tout $q \in \boldsymbol{V}$, et construire ainsi l'hypersphère $SR_{\bar{d}}$ avec $r_{\bar{d}}$ centrée dans $q$. Ainsi, à partir de la définition de $SR_{\bar{d}}$, nous pouvons avoir la propriété suivante :

**Property 3**

$$SR_{nf} \subseteq SR_{\bar{d}} \ pour \ \forall q \in V.$$

A partir de là, étant donné qu'il n'est plus nécessaire de rechercher le plus proche voisin $v_{q1}$ du point requête $q \in \boldsymbol{V}$ (et par conséquent le plus loin voisin $v_{q2}$), nous pouvons alors supprimer un accès au disque. La complexité de la méthode basée sur $\bar{d}_{min}$ et $\bar{d}_{max}$ peut alors être exprimée comme suit :

$$O(n + n'^3_{\bar{d}}) \quad (16)$$

où $n = |\boldsymbol{V}|$ et $n_{\bar{d}}$ est le nombre d'individus dans $SR_{\bar{d}}$.

En nous basant sur les propriétés précédentes, nous pouvons alors émettre la proposition suivante :

2. Cette valeur constitue ainsi une borne supérieure de $d_2$ dans la formule du rayon de la formule 8.

**Proposition 1** *Vu que la méthode initiale d'insertion locale peut construire un graphe de voisinage correct et mettre à jour le voisinage correctement, la nouvelle méthode basée sur les $SR_{\bar{d}}$ peut aussi construire un graphe correct pour $q \in \boldsymbol{V}$.*

**Preuve** A partir de la propriété 3, tous les individus se trouvant dans $SR_{nf}$ sont aussi contenus dans $SR_{\bar{d}}$, $\forall q \in \boldsymbol{V}$. Ainsi, vu qu'il est suffisant de mettre à jour le voisinage à l'intérieur de $SR_{nf}$ centrée sur $q$, la méthode basée sur $SR_{\bar{d}}$ est aussi suffisante pour mettre à jour correctement le graphe en mettant à jour le voisinage entre les individus dans $SR_{\bar{d}}$ centrée dans $q$.
Les Figures 3 et 4 illustrent la nouvelle méthode basée sur $\bar{d}_{min}$ et $\bar{d}_{max}$.

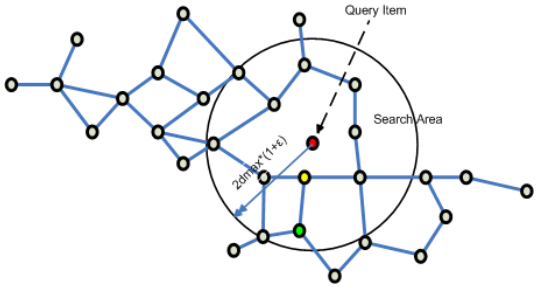


FIG. 3: La première phase dans la nouvelle méthode de mise à jour locale

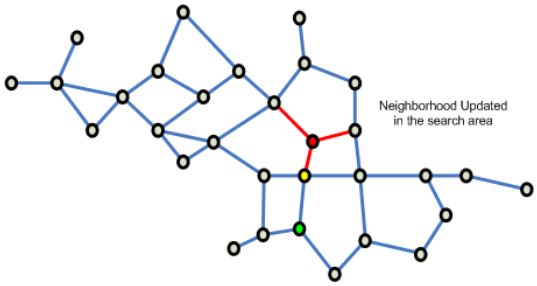


FIG. 4: La seconde phase dans la méthode de mise à jour locale qui applique la mise à jour uniquement à l'intérieur de l'hypersphère.

### 3.2.2 Problème associé à la nouvelle méthode de mise à jour locale

La méthode proposée ci-dessus est intéressante et rapide en supprimant un accès à la mémoire secondaire qui est une opération coûteuse en terme de temps de calcul. Toutefois, il y a une situation où cette méthode, opérant toute seule, ne peut pas mettre à jour correctement le graphe de voisinage. Ce problème est illustré dans les figures 5 et 6. Ainsi, cette utilisation naïve de la méthode peut insérer correctement un individu dans le graphe $G(V, E)$ pour un point requête $q \notin V$ si $r_{nf} \leq r_{\bar{d}}$. Toutefois, pour un individu $q \notin V$ avec $r_{nf} > r_{\bar{d}}$, $SR_{\bar{d}} \cap V = \phi$ et la méthode ne peut donc pas localiser l'individu et les relations de voisinage à mettre à jour.

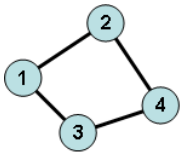


FIG. 5: Le graphe initial sur lequel $\bar{d}_{min}$ et $\bar{d}_{max}$ sont calculées.

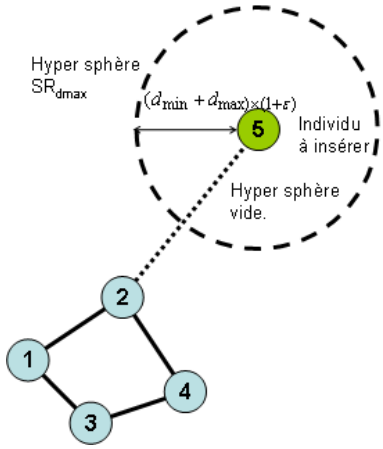


FIG. 6: Une nouvelle requête $q$ est insérée dans $G$. Si $d(q,v) > \bar{d}, \forall v \in V$, $SR_{\bar{d}} \cap V = \phi$ et donc $G$ ne peut être mis à jour correctement en utilisant uniquement $SR_{\bar{d}}$.

Les problèmes cités précédemment peuvent survenir car les différentes propriétés et propositions se vérifient pour les requêtes $q \in V$, et même pour ($q \notin V$ et $r_{nf} \leq r_{\bar{d}}$). Cette proposition ne peut toutefois pas être vérifiée pour une requête $q \notin V$ en général. Ceci est dû au fait que $r_{nf}$ peut être plus large que $r_{\bar{d}}$ pour un nouveau point quelconque $q \notin V$.

Pour résoudre ce problème nous proposons donc de combiner $SR_{\bar{d}}$ pour $G(V,E)$ et $SR_{nf}$ (dans LocalInsert($G(V,E)$,$q$,$\epsilon$)) de telle sorte que le cas où $SR_{\bar{d}} \cap V = \phi$ peut être pris en compte. Dans cette nouvelle approche, la méthode basée sur $\bar{d}$ est privilégiée et est utilisée quand $SR_{\bar{d}} \cap V \neq \phi$. Dans le cas où $SR_{\bar{d}} \cap V = \phi$, la méthode initiale est invoquée.

Un autre problème susceptible d'être rencontré lors de l'utilisation de $SR_{\bar{d}}$ est que $\bar{d}_{min}$ et $\bar{d}_{max}$ peuvent être une estimation trop large de $d_1$ et $d_2$, i.e., $r_{nf} \ll r_{\bar{d}}$ et donc $|SR_{nf}| \ll |SR_{\bar{d}}|$. Il peut arriver surtout que $|SR_{\bar{d}}| \simeq |V|$. Ceci signifie que l'utilisation de l'hyper sphère $SR_{\bar{d}}$ s'avère inefficace pour localiser la zone à mettre à jour dans un graphe de voisinage pour un point requête $q$ donné.

Pour apporter une solution à ce problème, nous utilisons $SR_{\bar{d}}$ pour construire une hyper sphère similaire à $SR_{nf}$, et utilisons l'hyper sphère pour localiser la zone de mise à jour pour le point requête $q$. La différence de la forme originale de $SR_{nf}$ dans LocalInsert($G(V,E)$,$q$,$\epsilon$) est que nous construisons l'hyper sphère uniquement en considérant les individus qui sont dans $SR_{\bar{d}}$ et non pas dans toute la base de données $V$. Ainsi, du moment que $SR_{\bar{d}} \subseteq V$ se vérifie, il est garanti que le nombre de points vérifiés pour la construction de l'hyper sphère est plus petit

**Algorithm 1** LocalInsert$_{d_{max}}$ ($G(V,E)$,$q$,$\bar{d}_{min}$,$\bar{d}_{max}$,$\epsilon$) basé sur $\bar{d}_{min}$ et $\bar{d}_{max}$

**Require:** $G(V,E)$ ;
**Require:** $q$ ; //*Le point requête à insérer.*
**Require:** $\bar{d}_{min}$ ;
**Require:** $\bar{d}_{max}$ ;
**Require:** $\epsilon$ ; //*Le paramètre de relaxation*

1: $r_{\bar{d}} = (\bar{d}_{min} + \bar{d}_{max})(1+\epsilon)$ ;
2: $V_{\bar{d}} = \{v \in V | d(q,v) \leq r_{\bar{d}}\}$ ;
3: $v_{q1}$ = Le plus proche voisin de $q$ dans $G$ ;
4: $d_1 = d(q, v_{q1})$ ;
5: $v_{q2}$ = Le plus loin voisin de $v_{q1}$ dans $G$ ;
6: $r_{nf} = (d_1 + d_2) \times (1+\epsilon)$ ;
7: **if** ($V_{\bar{d}} \neq \phi$) **then**
8: $\quad V^{SR} = \{v \in \boldsymbol{V}_{\bar{d}} |\ r_{nf} > d(q,v)\}$ ;
9: **else**
10: $\quad V^{SR} = \{v \in V |\ r_{nf} > d(q,v)\}$ ;
11: **end if**
12: Ajouter $q$ à $V$ ;
13: Mettre à jour $E$ en utilisant la méthode standard *seulement* pour le sous graphe $G_{sub}$ de $G$ ($G_{sub} \subseteq G$ et $G_{sub}$ est constitué de $V^{SR}$ et des arêtes correspondantes à $V^{SR}$).
14: **return** $G(V,E)$ mis à jour ;

que celui des points contenus dans toute la base de données. Par conséquent, les performances sont réduites dans tous les cas.

La méthode que nous proposons, appelée LocalInsert$_{d_{max}}$($G(V,E)$,$q$,$\bar{d}_{min}$,$\bar{d}_{max}$,$\epsilon$), est résumée dans l'Algorithme 1. Pour éviter l'augmentation de la complexité de la méthode proposée, quand on accède aux données pour trouver les individus dans $SR_{\bar{d}}$ à la ligne 2, nous récupérons aussi le plus proche voisin $v_{q1}$ de $q$ à la ligne 3. Nous procédons ensuite à la mise à jour, à la ligne 10, seulement quand $SR_{\bar{d}} \cap V = \phi$ et ce sans accéder aux données encore une fois pour trouver $v_{q1}$. En outre, et comme cela a été décrit précédemment, les distances calculées à la ligne 2 sont stockées et sont utilisées (lignes 8 et 10) pour éviter des calculs et des accès inutiles.

Ainsi, les propriétés suivantes sont vérifiées pour cette nouvelle stratégie de mise à jour locale.

**Proposition 2** *Les performances de LocalInsert$_{d_{max}}$($G(V,E)$,$q$,$\bar{d}_{min}$,$\bar{d}_{max}$,$\epsilon$) sont garanties d'être équivalentes ou supérieures à celles de LocalInsert ($G(V,E)$,$q$,$\epsilon$).*

**Preuve**

Par définition, étant donné que $SR_{\bar{d}} \subseteq V$ se vérifie, $|SR_{\bar{d}}| \leq |V|$. Si $SR_{\bar{d}} \cap V = \phi$ alors LocalInsert$_{d_{max}}$($G(V,E)$,$q$,$\bar{d}_{min}$,$\bar{d}_{max}$,$\epsilon$) est équivalente à LocalInsert ($G(V,E)$,$q$,$\epsilon$), et ses performances sont alors équivalentes aussi à la première. D'un autre côté, si $SR_{\bar{d}} \cap V \neq \phi$, vu que $SR_{\bar{d}} \subseteq V$ se vérifie, $V^{SR}$ à la ligne 8 dans l'algorithme 1 est plus petite que $V^{SR}$ à la ligne 10, qui est équivalente à l'hypersphère dans LocalInsert($G(V,E)$,$q$,$\epsilon$). Ainsi, les performances de la première sont équivalentes ou supérieures à la méthode initiale.

Dans le pire des cas (seulement quand $SR_{\vec{q}} \cap V = \phi$), la complexité de la nouvelle méthode est équivalente à celle de LocalInsert($G(V, E)$,$q$,$\epsilon$). Toutefois, ce cas arrive uniquement quand la requête spécifiée est située à l'extérieur de tous les points du graphe. Ce point peut être considéré, à cet instant là, comme un *point aberrant* (*outlier*) par rapport à l'ensemble des points de $V$. En plus de l'insertion locale, cette approche est étendue pour la construction incrémentale des graphes.

### 3.2.3 Une heuristique pour l'amélioration de la suppression locale

Toujours dans un esprit de limitation de la dépendance des méthodes proposées par rapport à la taille des bases de données, nous avons proposé d'améliorer aussi la méthode de suppression locale dans un graphe de voisinage. Nous proposons alors une nouvelle méthode, basée certes sur une heuristique, mais totalement indépendante de la taille de la base de données.

Cette méthode est basée sur des observations que nous avons menées sur la structure globale des graphes de voisinage. L'idée est alors de supprimer la lecture faite par la méthode initiale (représentée par $O(n)$ dans $O(n + n'^3)$). Pour cela, nous nous basons sur le concept des couches de voisinage. En effet, nous avons émis l'hypothèse qui consiste à dire que, lors de la suppression d'un point dans un graphe, les relations de voisinage pouvant être affectées sont celles des voisins directs ainsi que celles de quelques couches liées à cette première couche de voisinage (voir Figure 7).

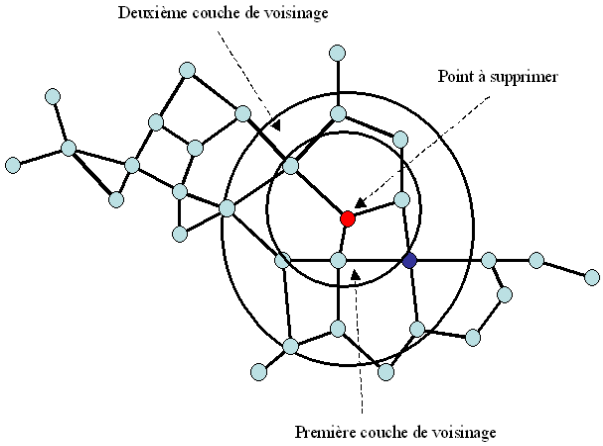


FIG. 7: Illustration de l'heuristique de suppression locale.

Ainsi, l'idée ici est de pouvoir déterminer un nombre minimum de couches de voisinage pouvant être impliquées dans la mise à jour, tout en assurant une mise à jour correcte. Nous avons alors vérifié cette hypothèse par des expérimentations et des observations sur les graphes. Nous avons conclu que notre hypothèse se vérifie en utilisant uniquement deux couches de voisinage : soit $v_i \in \boldsymbol{V}$, le voisinage de $v_i$ est défini comme $N(v_i) = \{v_j \in \boldsymbol{V} / v_i \neq v_j \wedge (v_i, v_j) \in \boldsymbol{E}\}$. Ainsi, $N(v_i)$ constitue la première couche de voisinage (i.e. points ayant une relation directe avec le point $v_i$). De ce fait, nous pouvons définir l'ensemble des points pouvant être affectés par la suppression d'un point dans le graphe par la formule : $N(N(vi))$.

### 3.3 Construction incrémentale du graphe avec un accès réduit aux disques

L'Algorithme 1 est basé sur un graphe initial avec lequel la mise à jour du graphe est réalisée. Toutefois, que ce passerait-il s'il n'y a pas de graphe disponible pour la mise à jour ? Etant donné que la méthode proposée est capable de mettre à jour la structure $G(\boldsymbol{V}, \boldsymbol{E})$ tout en conservant la propriété de voisinage, il semble assez aisé de la généraliser pour la construction incrémentale de graphes de voisinage. Par exemple, considérons le graphe $G(\boldsymbol{V_{new}}, \boldsymbol{E})$ où $\boldsymbol{V_{new}} = \{v_1, v_2\}$ et $\boldsymbol{E} = \{(v_1, v_2)\}$, à chaque itération nous tirons un sommet $v \in \boldsymbol{V}$ et l'ajoutons dans $G$ à l'aide de l'Algorithme 1 et décrémentons $\boldsymbol{V}$ comme suit : $\boldsymbol{V} \backslash \{v\}$.

Cependant, cette méthode "naïve" ne peut pas construire les graphes de voisinage correctement. Ceci est dû au fait que l'insertion d'un élément de données $v$ dans $G(\boldsymbol{V}, \boldsymbol{E})$ exige la *mise à jour* de $\bar{d}_{min}$ et $\bar{d}_{max}$ pour les sommets insérés $\boldsymbol{V} \cup \{v\}$ en plus de la mise à jour de $\boldsymbol{E}$. Par ailleurs, une mise à jour naïve de ces bornes à chaque insertion d'un nouvel élément $q$ en calculant à chaque fois la distance $d(q, v)$ pour $\forall v \in \boldsymbol{V}$ n'est pas acceptable, étant donné que le traitement prend $O(n)$ avec $n = |\boldsymbol{V}|$ pour chaque insertion.

Pour résoudre le problème pour le graphe courant $G(\boldsymbol{V}, \boldsymbol{E})$, nous introduisons une autre structure de données pour gérer l'ensemble des *bornes* de $\boldsymbol{E}$. Cette structure de données stocke l'ensemble des bornes de $\boldsymbol{E}$ par rapport à leur longueur de telle sorte que nous puissions toujours obtenir la bonne limite pour les données actuelles $\boldsymbol{V}$ indépendamment de la suppression ou de l'insertion d'un noeud. La contrainte imposée sur la structure de données est que toutes les bornes pour le graphe courant $G(\boldsymbol{V}, \boldsymbol{E})$ doivent être triées (par ordre décroissant) par rapport à leur longueur, et l'ordonnancement doit être conservé correct à chaque fois que l'insertion ou la suppression de sommets se produit. Ceci peut être aisément réalisé en utilisant un standard dans le commerce tel que B-arbre (Bentley (1975)).

Notre algorithme incrémental de construction de graphes de voisinage est résumé dans l'Algorithme 2. Dans cet algorithme, sauf pour $|\boldsymbol{V}| < 2$, $\boldsymbol{E}$ est initialisé avec une arête reliant deux sommets, qui sont prélevés aléatoirement de $\boldsymbol{V}$ (lignes 4 à 9), et crée les structures de données pour l'ensemble des arêtes $\boldsymbol{E}$ (ligne 10). Après cela, nous appliquons l'Algorithme 1 pour les données restantes dans $\boldsymbol{V}$ jusqu'à ce que $\boldsymbol{V}$ soit vide (ligne 15). Notons que $\bar{d}_{min}$ et $\bar{d}_{max}$ sont obtenus en utilisant la structure (lignes 14 et 15) pour qu'ils soient ensuite mis à jour (ligne 17). A la fin, l'algorithme retourne un graphe de voisinage correct correspondant à l'ensemble $\boldsymbol{V}$ selon la propriété de voisinage spécifiée pour le graphe $G$.

Pour ce qui est de la complexité de $IncNGC_{d_{max}}(\boldsymbol{V}, \epsilon)$, celle ci peut être nettement plus intéressante que celle de la méthode naïve. A chaque $i$-ème itération, la mise à jour des structures pour $\boldsymbol{E_i}$ peut être effectuée en $O(\log |\boldsymbol{E_i}|)$. Supposons $n'_i = |\boldsymbol{v_i}^{SR}| (\ll n_i = |\boldsymbol{v_i}|)$, où $\boldsymbol{E_i}^{SR}$ est l'ensemble des arêtes de $\boldsymbol{v_i}^{SR}$ à chaque $i$-ème itération. Ici, $n_i$ représente le nombre de données qui figurent déjà dans le graphe de voisinage $G$ à la $i$-ème itération, et $n'_i$ représente la quantité de données localisées dans $SR_{\bar{d}}$. L'insertion (suppression) des arêtes $\boldsymbol{E_i}^{SR}$ dans la structure nécessite $O(|\boldsymbol{E_i}^{SR}| \log |\boldsymbol{E_i}|)$, vu que la hauteur de l'arbre dans les structures est proportionnelle à $O(\log |\boldsymbol{E_i}|)$ et le nombre d'arêtes à insérer est $|\boldsymbol{E_i}^{SR}|$. De nos expérimentations préliminaires (Hacid and Zighed (2005); Hacid and Yoshida (2007)), $|\boldsymbol{E}|$ est généralement *linéaire* à $n(= |\boldsymbol{V}|)$ [3], $O(\boldsymbol{E_i}^{SR})$ peut être approché par $O(n_i)$.

En résumé, à chaque $i$-ème itération, $IncNGC_{d_{max}}(\boldsymbol{V},$ epsilon $)$ prend $O(n_i + n'^3_i)$ pour la mise à jour d'un graphe de voisinage à l'aide de l'Algorithme 1, $O(n'_i \log n_i)$ pour la

3. En moyenne, $|N(v)| = O(10) \sim O(10^2)$ pour $\forall v \; in \boldsymbol{V}$ où $|\boldsymbol{V}| = O(10^3) \sim O(10^4)$

**Algorithm 2** IncNGC$_{d_{max}}(\boldsymbol{V},\epsilon)$ : Construction incrémentale de graphes de voisinage

**Require:** $\boldsymbol{V}$ ;
**Require:** $\epsilon$ ;
1: **if** $|V| < 2$ **then**
2: **return** $G(\boldsymbol{V}, \phi)$ ;
3: **else**
4: $v_1$ = a data item from $\boldsymbol{V}$ ;
5: $v_2$ = another data item from $\boldsymbol{V}$ ; //$v_1 \neq v_2$
6: $\boldsymbol{V}$ = $\boldsymbol{V}\backslash\{v_1, v_2\}$ ;
7: $\boldsymbol{V_{new}}$ = $\{v_1, v_2\}$ ;
8: $E = \{(v_1, v_2)\}$ ;
9: create $DB_{\bar{d}_{min}}$ and $DB_{\bar{d}_{max}}$ for $\boldsymbol{E}$ ;
10: **for** $i = 3$ *to* $n$ **do**
11: $v_i$ = a data item from $\boldsymbol{V}$ ;
12: $\boldsymbol{V}$ = $\boldsymbol{V}\backslash\{v_i\}$ ;
13: $\bar{d}_{min} = \underset{v_i \in \boldsymbol{V_{new}}}{\text{argmax}} \ \underset{v_j \in N(v_i)}{\text{argmin}}\, d(v_i, v_j)$ ; *//utilize $DB_{\bar{d}_{min}}$ for the calculation*
14: $\bar{d}_{max} = \underset{v_i \in \boldsymbol{V_{new}}}{\text{argmax}} \ \underset{v_j \in N(v_i)}{\text{argmax}}\, d(v_i, v_j)$ ; *//utilize $DB_{\bar{d}_{max}}$ for the calculation*
15: $G(\boldsymbol{V_{new}}, \boldsymbol{E}) = LocalInsert_{\bar{d}}(G(\boldsymbol{V_{new}}, \boldsymbol{E}), v_i, \bar{d}_{min}, \bar{d}_{max}, \epsilon)$ ;
16: Update $DB_{\bar{d}_{min}}$ and $DB_{\bar{d}_{max}}$ based on the updated $\boldsymbol{E}$ ;
17: **end for**
18: **end if**
19: **return** $G(\boldsymbol{V_{new}}, \boldsymbol{E})$ ; *// $\boldsymbol{V_{new}}$ should be equal to $\boldsymbol{V}$*

mise à jour de $\bar{d}_{min}$ et $\bar{d}_{max}$. Ainsi, il faut $O(\sum_{i=3}^{n}[n_i + n_i'^3 + (n_i' \log n_i)])$, au total, qui est (nettement) plus faible que celle de la méthode naïve nécessitant $O(\sum_{i=3}^{n}[n_i + n_i'^3])$.

# 4 Evaluations

Comme décrit dans dans la Section 3, théoriquement, la méthode proposée a la propriété de *garantir certaines améliorations* par rapport à la méthode précédente (Hacid and Yoshida (2007)). Du point de vue de la validité, il est clair que la modification de la méthode (i.e., $LocalInsert_{\bar{d}}$) donne des résultats corrects comme indiqué précédemment. Cette partie ne sera donc pas discutéé dans cette section. Ainsi, nous mettons l'accent sur l'évaluation des performances de calcul, i.e., temps d'exécution, de l'approche proposée dans le but de l'illustrer et de la comparer avec d'autres approches. Les résultats seront discutés et décrits selon les deux aspects suivants : (1) le temps d'exécution et le passage à l'échelle de la méthode d'insertion locale modifiée ; (2) le temps d'exécution de la méthode de construction incrémentale. Notons que ces résultats sont également influencés par l'implémentation mise en œuvre. Pour montrer les différences entre $LocalInsert_{\bar{d}}$ et le méthode proposée antérieurement dans (Hacid and Yoshida (2007)), que nous appelons $LocalInsert$, les résultats sont présentés sur le même graphique afin de faciliter leur compréhension et d'analyse.

## 4.1 Configuration des évaluations

Etant donné que nous nous concentrons sur le temps de calcul pour gérer les ensembles de données en fonction de leur taille, nous avons utilisé des ensembles de données synthétiques avec la taille spécifiée dans les évaluations ci-dessous. Pour mener des comparaisons avec notre approche précédente, nous avons suivi le même protocole d'évaluation que celui décrit dans (Hacid and Zighed (2005)). Les temps d'exécution ont été mesurés sur une machine avec un processeur de 1,8 GHz, 512 Mo de mémoire et un système Windows XP.

## 4.2 Passage à l'échelle de $LocalInsert_{\bar{d}}$

L'évaluation du passage à l'échelle se réfère au comportement des méthodes proposées par rapport à la taille ($n_i$) et la dimension $p$ de l'ensemble de données. Nous avons généré des séries de données artificielles ayant une dimension $p = 250$. Les ensembles de données varient de $5 \times 10^3$ individus à $40 \times 10^3$, avec un intervalle de $2,5 \times 10^3$. A chaque itération, $10$ individus sont tirés aléatoirement de l'ensemble des données correspondantes, et réinsérés dans le graphe. Les expérimentations ont été menées en considérant les situations où le sommet qui doit être inséré dans un graphe de voisinage réside à l'intérieur du graphe. Pour chaque graphe correct $G(\boldsymbol{V}, \boldsymbol{E})$, un sommet $v \in \boldsymbol{V}$ est choisi aléatoirement $\boldsymbol{V}$, et le graphe sans $v$ (soit $G'(\boldsymbol{V} \backslash \{v\}, \boldsymbol{E'})$) est construit. Puis, $v$ est inséré dans $G'$ en utilisant la méthode proposée. Le temps d'exécution a été enregistré pour ce processus. Les résultats obtenus sont présentés dans la Figure 8.

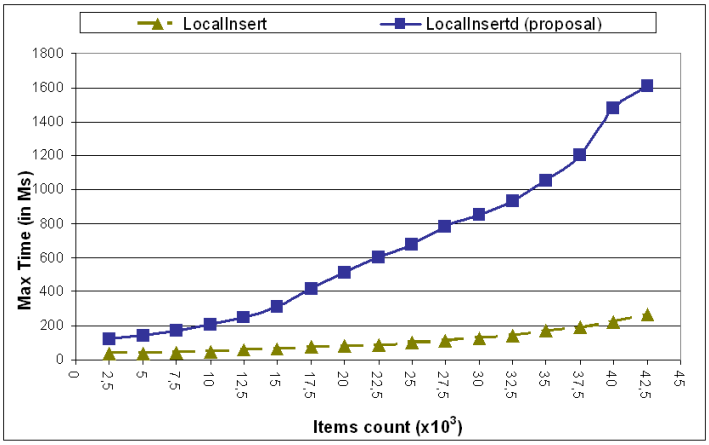


FIG. 8: Comparaison des temps d'exécution suivant deux stratégies.

## 4.3 Temps d'exécution de la construction incrémentale

Nous avons ensuite évalué le comportement de la construction incrémentale. L'idée est simple : disposer d'un ensemble de données, appliquer des stratégies différentes pour construire un graphe de voisinage et enregistrer les temps d'exécution. Comme dans la Section 4.2, nous

avons généré des ensembles de données artificielles ayant une dimension $p = 250$ et une taille de $5, 10, 20, 40, 50, 75, 100, 150 \times 10^3$. Nous avons comparé notre méthode de construction incrémentale (Algorithme 2) avec une autre version proposée dans (Hacid and Yoshida (2007)) et avec l'Algorithme de construction naïve (avec $O(n^3)$). Les résultats obtenus sont illustrés dans la Figure 9.

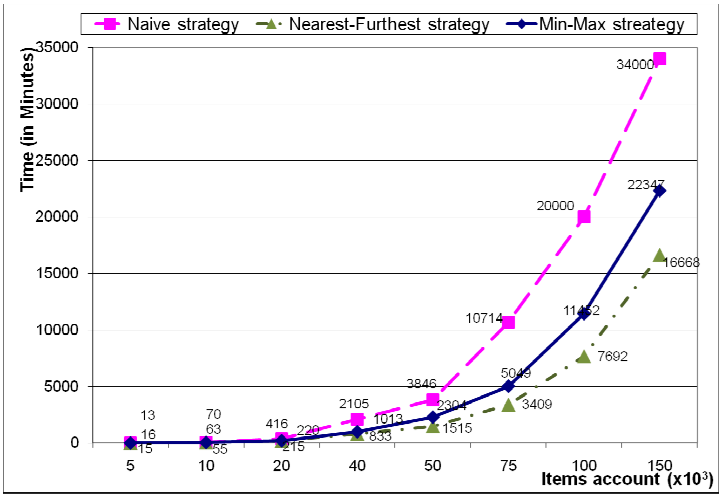


FIG. 9: Comparaison des temps d'exécution de trois approches

## 4.4 Discussions

La Figure 8 montre que les deux méthodes ($LocalInsert_{\bar{d}}$ et $LocalInsert$) est linéaire par rapport à la taille des données. En outre, la Figure 9 montre que la construction de graphes de voisinage fondée sur les méthodes supplémentaires est efficace comparée à l'approche naïve. Toutefois, en ce qui concerne le temps d'exécution dans le monde réel, les résultats indiquent que $LocalInsert_{\bar{d}}$ n'améliore pas la performance par rapport à $LocalInsert$.

Bien que la propriété théorique de l'Algorithme 1 indiquée dans la proposition 2, $r_{\bar{d}}$ peut être une estimation très prudente pour $r_{nf}$ en fonction des caractéristiques de l'ensemble de données. Ainsi, la taille de $SR_{\bar{d}}$ peut être assez grande pour garantir cette propriété. La localisation des données à traiter pour les mise à jour n'est pas encore suffisante à la ligne 2 de l'Algorithme 1. En outre, les évaluations dans les Figures 8 et 9 ne prennent pas explicitement en compte le coût (temps d'exécution) des accès disques, ce qui dépend de l'architecture de la machine et le système d'exploitation utilisés. En particulier, l'accès aux disques se produit lorsque la taille des données dépasse la taille de la mémoire principale : lorsque les données traitées sont échangées avec l'espace de stockage secondaire et de nouveaux éléments de données sont échangés dans la mémoire. Ainsi, actuellement, il n'est pas clair dans quelle mesure l'utilisation d'un graphe de voisinage en tant que structure d'indexation dans la mémoire principale aide pour identifier le voisinage. Il serait nécessaire de mesurer le nombre d'accès disque et le temps pris pour l'accès aux disques. Cela permettrait de clarifier l'effet de l'utilisation d'un graphe de voisinage en tant que structure d'indexation.

## 5 Conclusion

Cet article propose d'utiliser la structure de graphe de voisinage pour l'indexation à grande échelle de données multidimensionnelles. Du point de vue de la réduction des accès aux disques, nous avons proposé une méthode modifiée de recherche de voisinage pour sa mise à jour sur la base de statistiques sur les données. Nous avions défini des limites (supérieure et inférieure) pour la méthode proposée dans (Hacid and Zighed (2005)), et les ont utilisées pour la construction d'une hyper-sphère localisant la mise à jour. Pour faire face à certains problèmes dans l'utilisation de la borne supérieure, nous avons aussi mis au point plusieurs mécanismes et les ont incorporés dans la méthode de mise à jour. En outre, nous avons également proposé une méthode incrémentale pour la construction basée sur des statistiques calculées sur le graphe de voisinage. Plusieurs expériences ont été menées pour évaluer l'approche proposée et les résultats indiquent que notre approche est prometteuse. Toutefois, ils ont aussi révélé que plus d'efforts devraient être menés pour l'amélioration de sa mise en œuvre.

Dans l'avenir immédiat, nous pensons étudier l'effet d'utiliser un graphe de voisinage en tant que structure d'indexation et d'optimiser sa mise en œuvre et son implémentation en considérant les analyses de ce travail. Nous prévoyons également d'étendre la méthode proposée pour traiter la suppression d'éléments de données (Hacid and Yoshida (2007)), en plus de l'insertion.

## Références

Bentley, J. L., 1975. Multidimensional binary search trees used for associative searching. Commun. ACM 18 (9), 509–517.

Gabriel, K. R., Sokal, R. R., 1969. A new statistical approach to geographic variation analysis. Systematic zoology 18, 259–278.

Gaede, V., Günther, O., 1998. Multidimensional access methods. ACM Comput. Surv. 30 (2), 170–231.

Hacid, H., Yoshida, T., 2007. Incremental neighborhood graphs construction for multidimensional databases indexing. In : Proc. of the 20th Canadian Conference on Artificial Intelligence. pp. 405–416.

Hacid, H., Zighed, D. A., 2005. An effective method for locally neighborhood graphs updating. In : DEXA. pp. 930–939.

Jaromczyk, J. W., Toussaint, G. T., 1992. Relative neighborhood graphs and their relatives. P-IEEE 80, 1502–1517.

Katajainen, J., 1988. The region approach for computing relative neighborhood graphs in the lp metric. Computing 40, 147–161.

MacQueen, J. B., 1967. Some methods for classification and analysis of multivariate observations. In : Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability", Berkeley, University of California Press, 1. pp. 281–297.

Preparata, F., Shamos, M. I., 1985. Computationnal Geometry-Introduction. Springer-Verlag, New-York.

Smith, W. D., 1989. Studies in computational geometry motivated by mesh generation. PhD thesis, Princeton University.

Somervuo, P., Kohonen, T., 1999. Self-organizing maps and learning vector quantization for feature sequences. Neural Processing Letters 10 (2), 151–159.

Toussaint, G. T., 1980. The relative neighborhood graphs in a finite planar set. Pattern recognition 12, 261–268.

Toussaint, G. T., 1991. Some insolved problems on proximity graphs. D. W Dearholt and F. Harrary, editors, proc. of the first workshop on proximity graphs. Memoranda in computer and cognitive science MCCS-91-224. Computing research lab. New Mexico, state university Las Cruces.

## Summary

Efficient neighborhood search in a multi-dimensional space has been studied in several fields such as pattern recognition and data mining. Neighborhood graphs are geometrical structures based on the concept of proximity of data items, and they can be utilized to determine the nearest neighbors for multi-dimensional data. One problem is that, besides the construction of neighborhood graphs, their update due to the insertion or deletion of data items can also become expensive. To alleviate this problem, a localization-based method was previously proposed. However, it might not be scalable because it first tries to find out the nearest neighbor for a queried data item by checking all the data items, which can be time consuming when the size of dataset gets large. In this paper we propose to go one step further in the usage of neighborhood graphs for indexing larger multi-dimensional data. By utilizing the structure of a graph for data indexing, we propose a modified neighborhood graph update method so that it can work with reduced data access to realize the scalability. Furthermore, we also propose an incremental neighborhood graph construction method based on statistics of the neighborhood graph. Experiments for evaluating the proposed approach were conducted, and the results indicate that our approach is promising.