Recherche d'information par le contenu II

Master2 - Partage de données à grande échelle - GMIN307

Alexis JOLY - alexis.joly@inria.fr

November 1st 2012

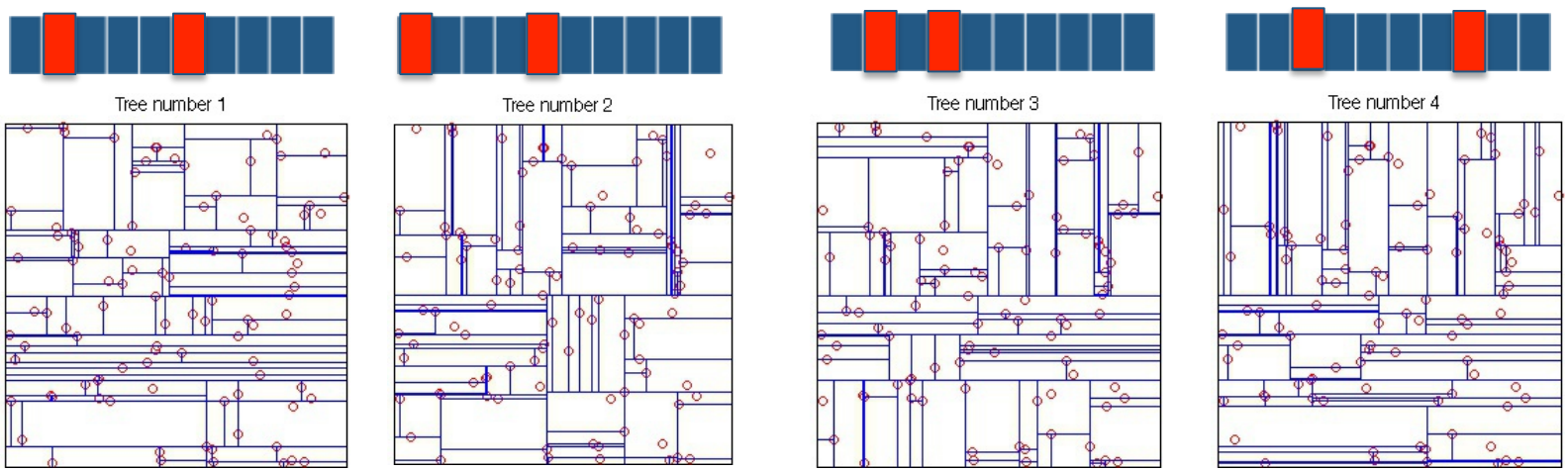
Présentation du randomized kd-tree

Motivation

Recherche exact dans les arbres (branch & band ou best bin first) rentable que si dim<<15 à cause de la malédiction de la dimension

Au lieu de construire un seul arbre, on **construit L arbres de dimension réduite** en ne considérant qu'un **sous ensemble de composantes dans chaque arbre**

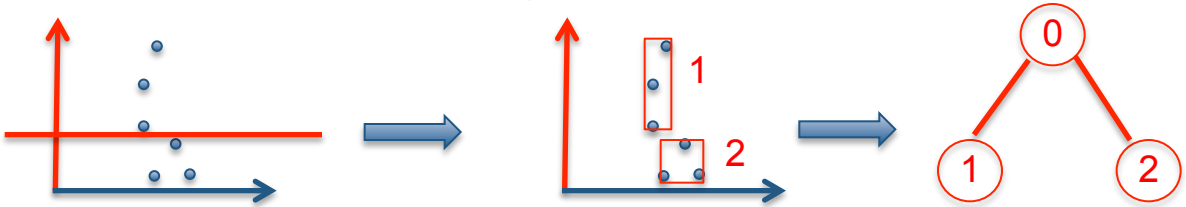
Les sous-espaces sont tirés aléatoirement **parmi les composantes** ou par **rotation aléatoire** de l'espace pour que les recherches dans chaque arbre soit **indépendantes** (complémentaires !!)



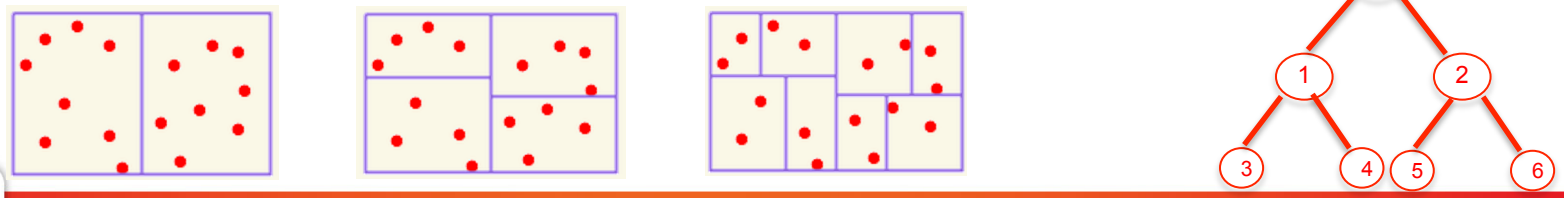
Construction d'un kd-tree classique

1. Parmi les d dimensions, on sélectionne celle pour laquelle la variance des points de la base est maximale

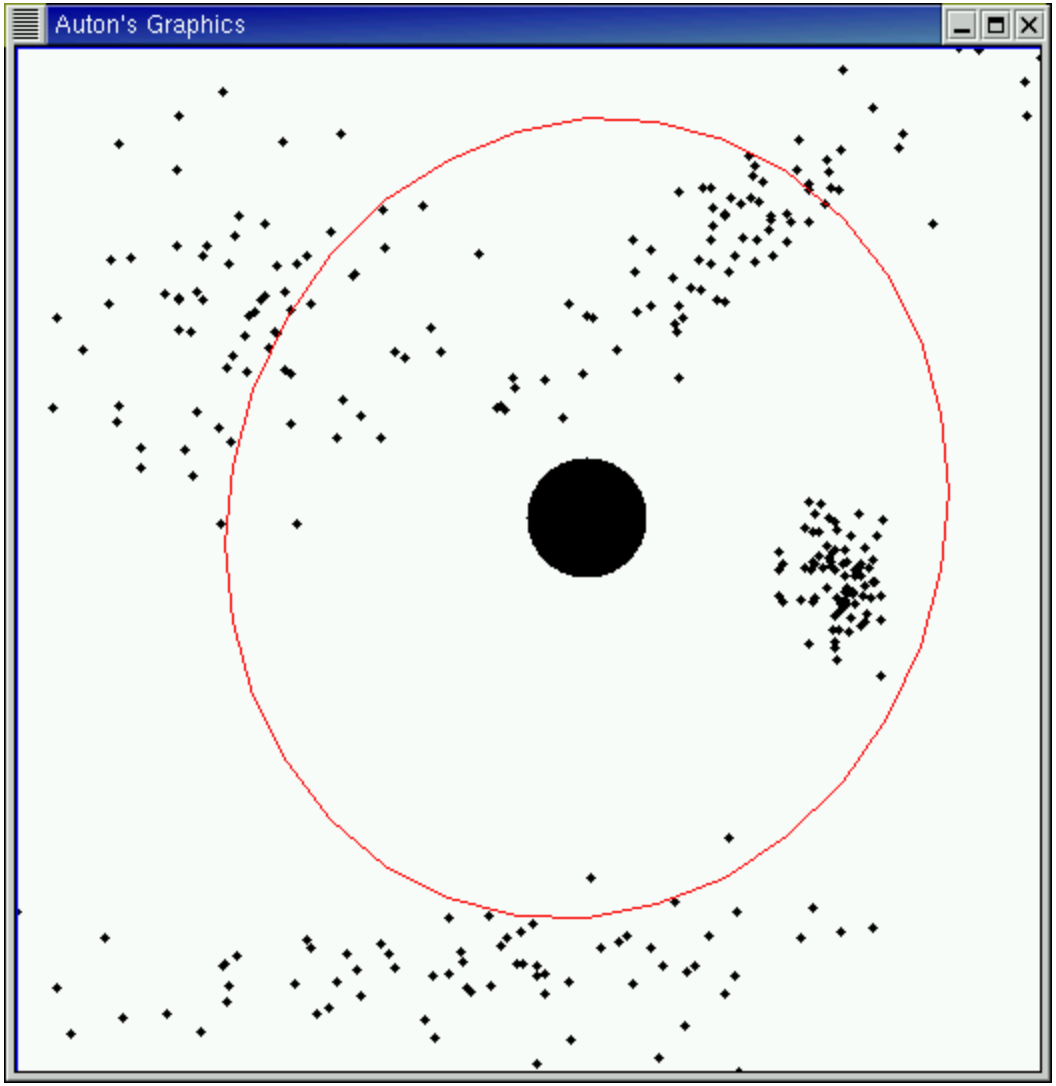
2. On coupe les données au niveau de la médiane et on calcul les paramètres de la forme englobante



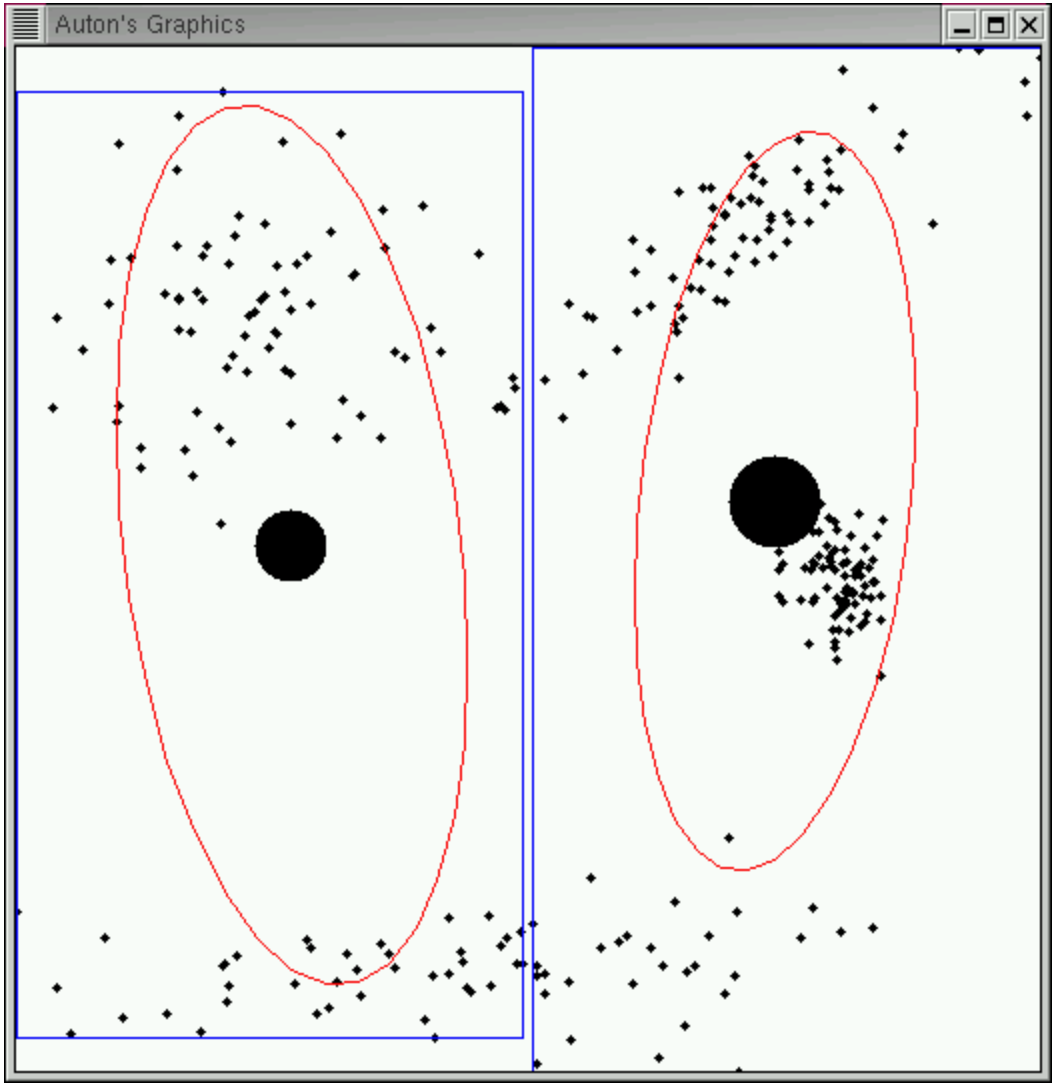
3. On réitère récursivement dans chaque partition fille.



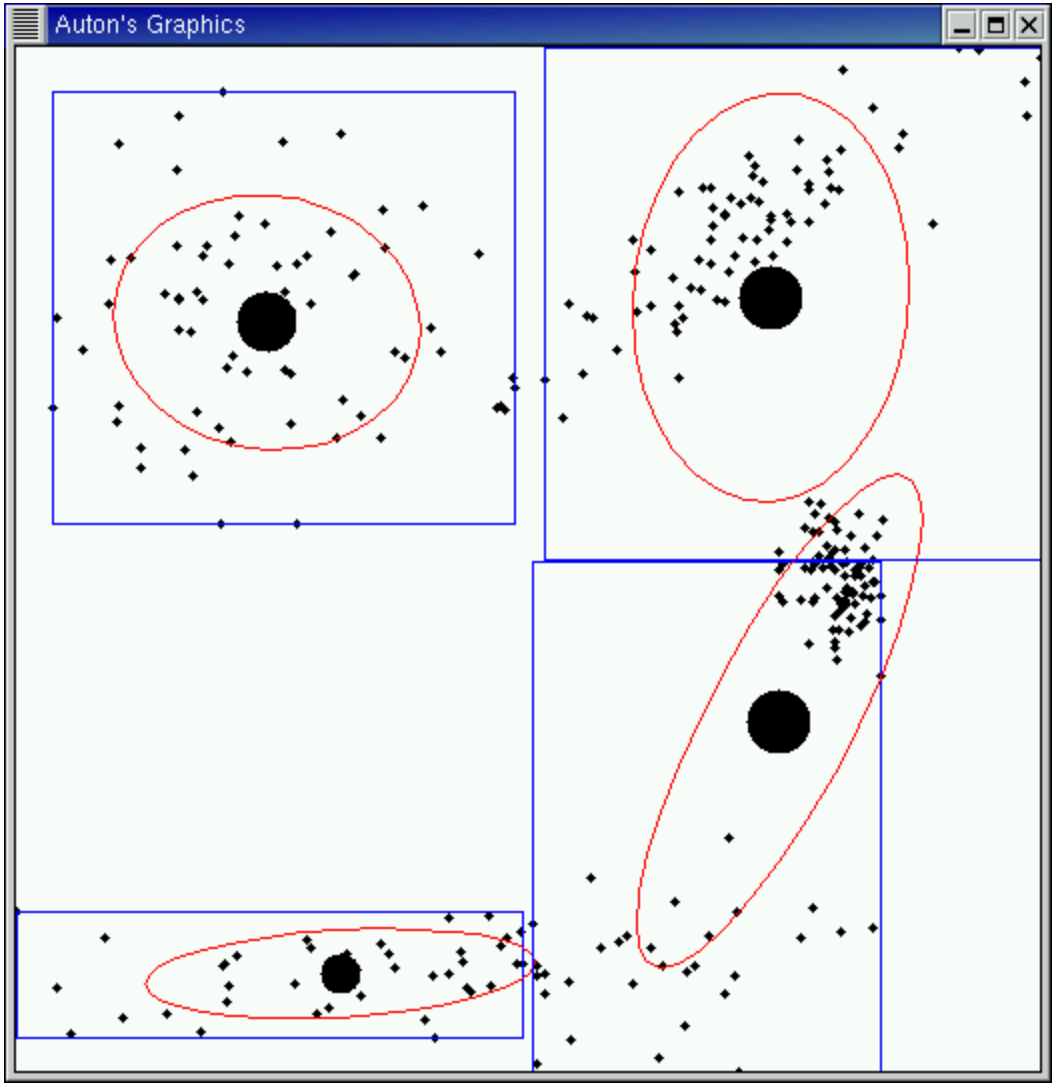
Construction d'un kd-tree 2D



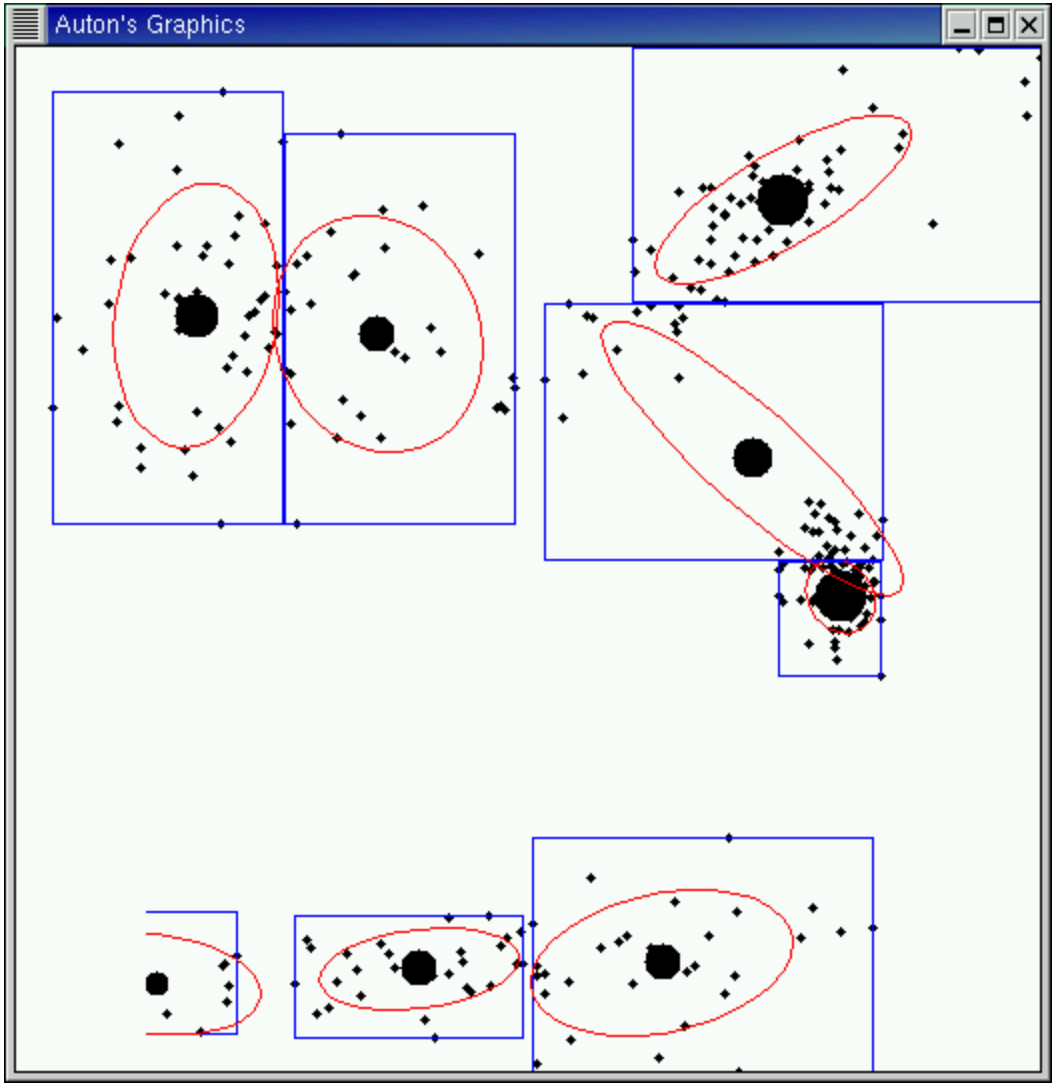
Construction d'un kd-tree 2D



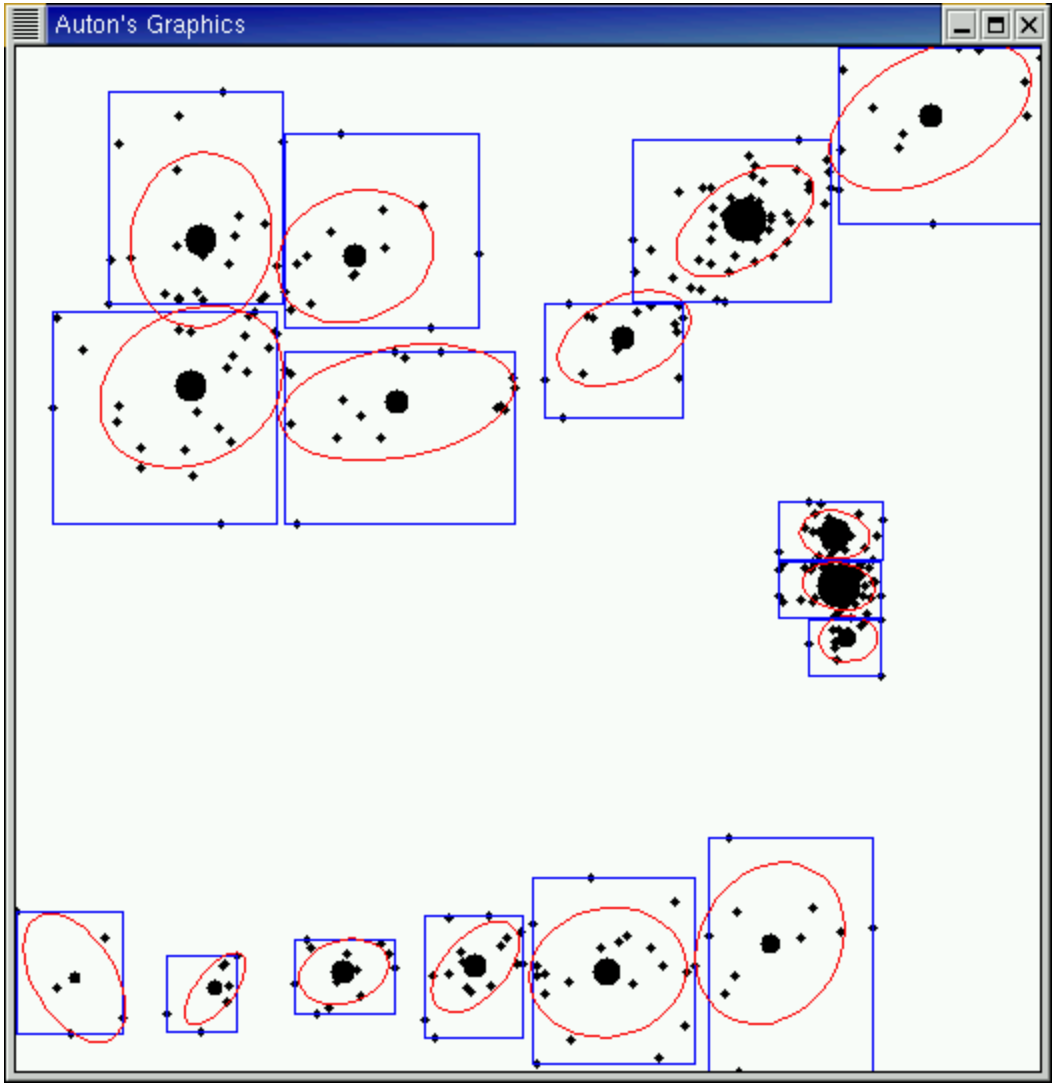
Construction d'un kd-tree 2D



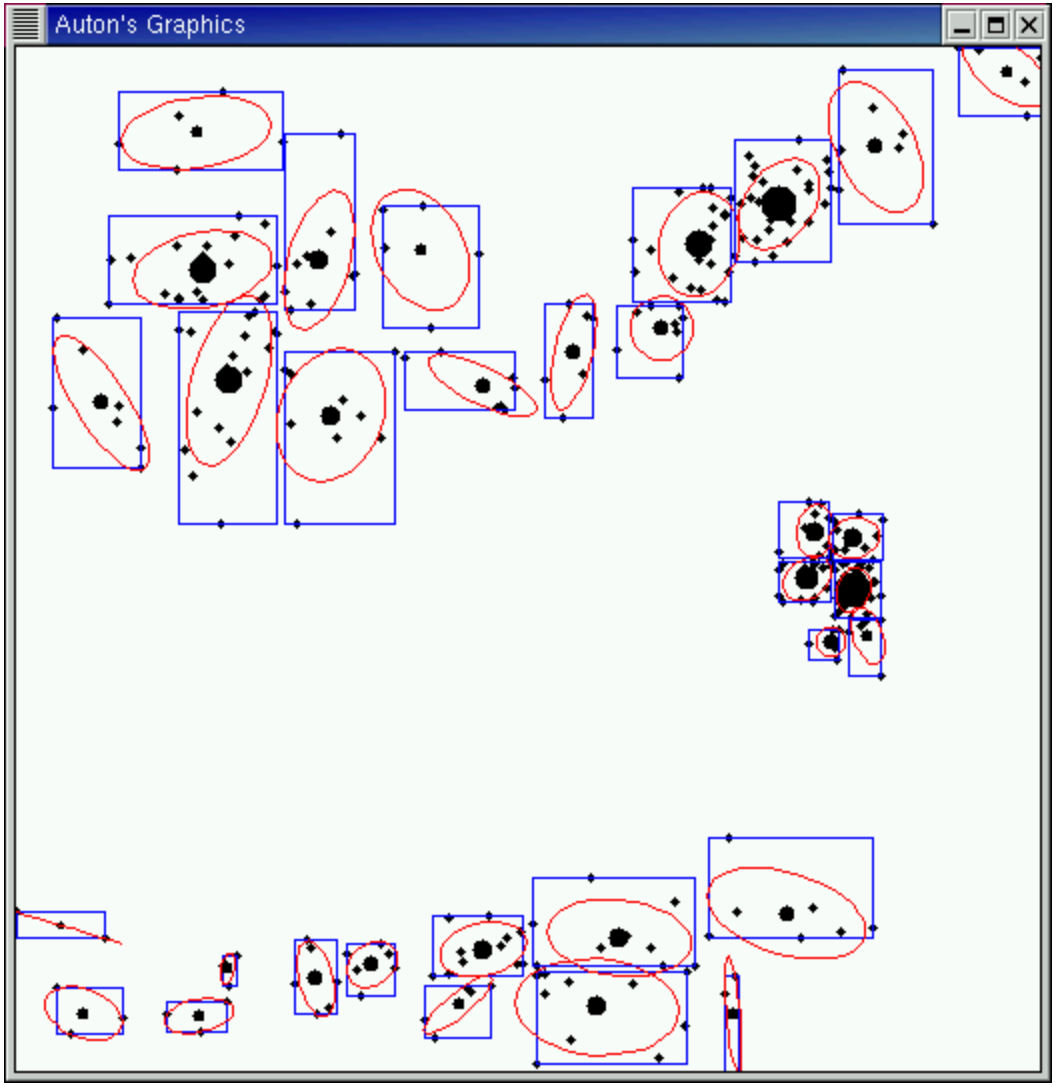
Construction d'un kd-tree 2D



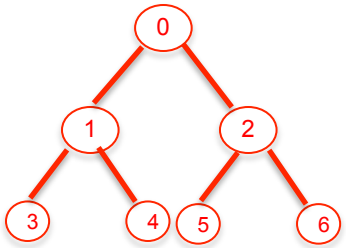
Construction d'un kd-tree 2D



Construction d'un kd-tree 2D



Construction d'un kd-tree classique



L'arbre résultant est un arbre **binaire balancé**

Binaire = chaque nœud a exactement deux fils

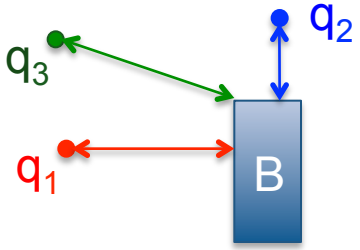
Balancé = Il y a toujours autant de points dans chacune des deux branches filles

La profondeur de l'arbre est log2(n) si on poursuit la procédure de construction jusqu'à ce que les feuilles ne contiennent plus qu'un point

Il peut être plus rentable de s'arrêter avant et de parcourir séquentiellement tous les points d'une feuille au moment de la recherche

Recherche dans un kd-tree classique

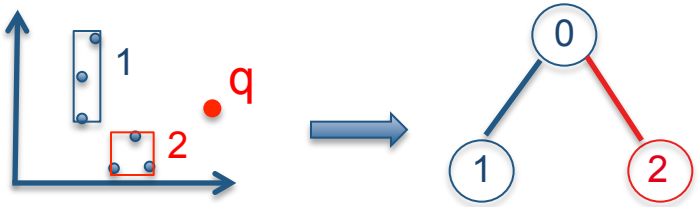
Borne min d'une région par rapport à la requête:

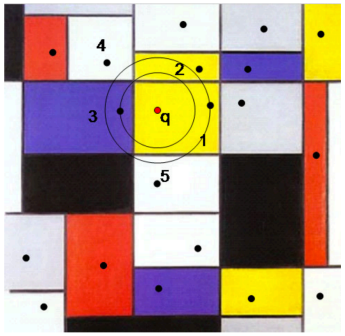


$$d_{min}(q, B)$$

Algorithme de recherche = Branch and Bound

Parcours de l'arbre en profondeur, à chaque noeud, on choisit la région fille la plus proche de la requête:

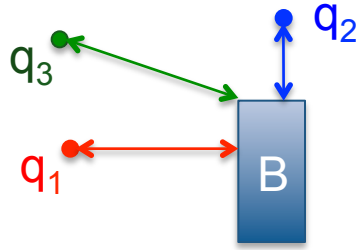




Lorsque l'on rencontre un p' tel que d(p',q) > r_{NN} mise à jour du rayon de recherche : r_{NN}=d(p',q)

Recherche dans un kd-tree classique

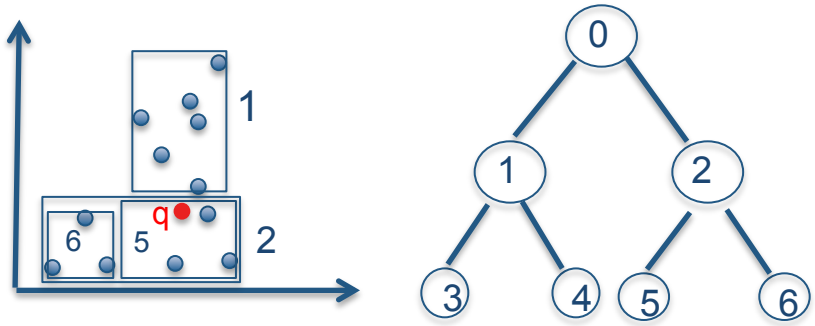
Borne min d'une région par rapport à la requête:



$$d_{min}(q, B)$$

Algorithme de recherche = Best Bin First

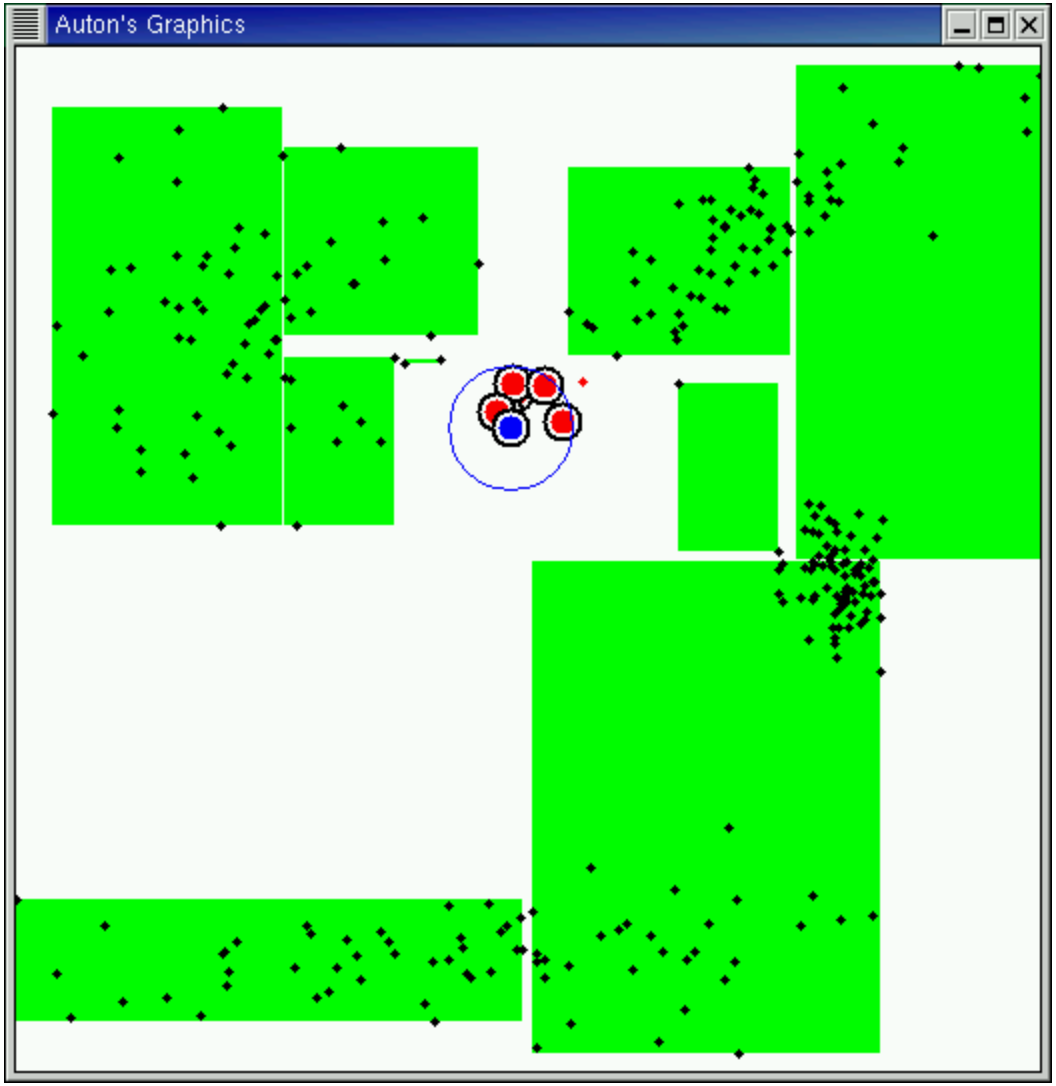
On maintient les régions à parcourir dans une priority queue:



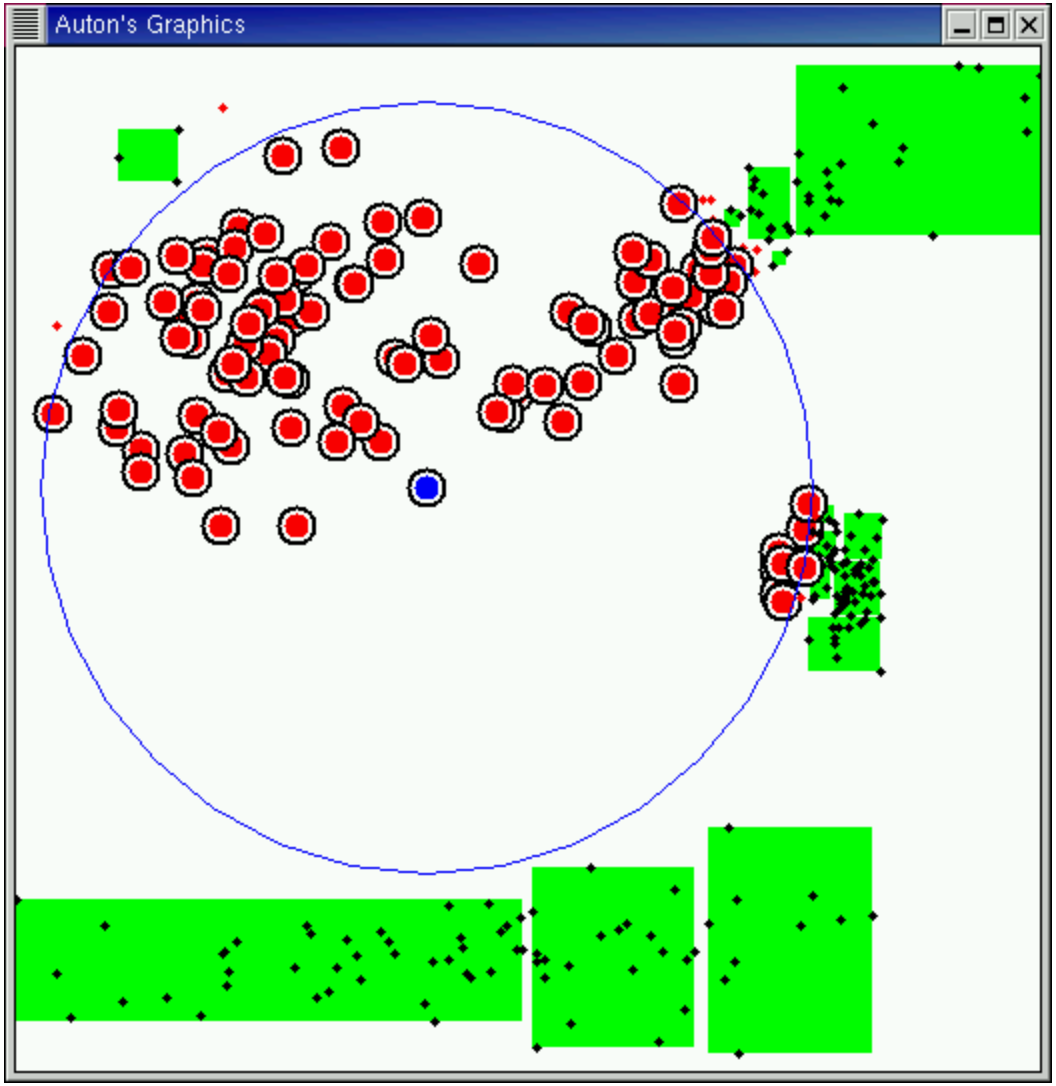
Priority Queue:
It1 {0}
It2 {2,1}
It3 {5,1,6}

Lorsque l'on rencontre un p' tel que d(p',q) > r_{NN} mise à jour du rayon de recherche : r_{NN}=d(p',q)

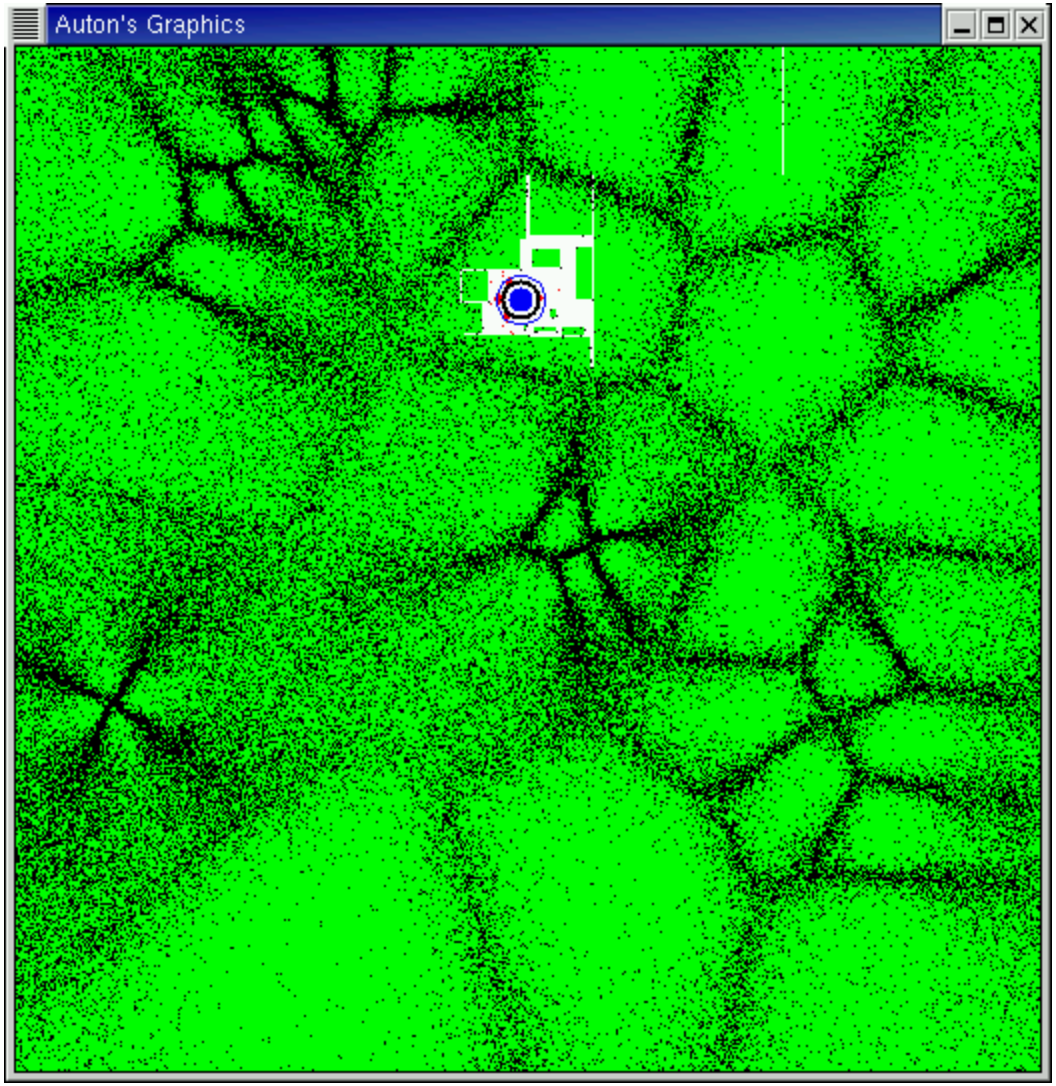
Range Query: petit rayon



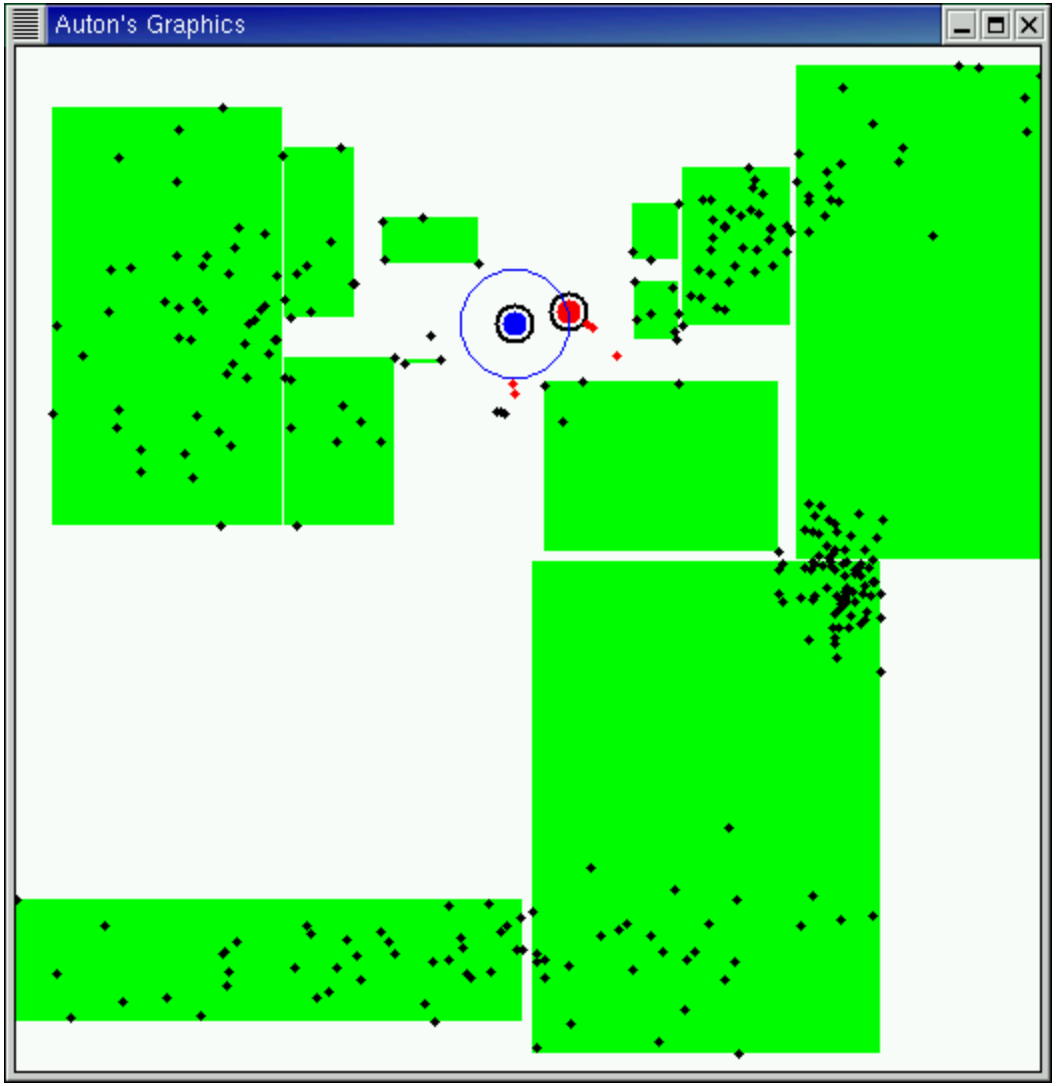
Range Query: grand rayon



Range Query: grande base

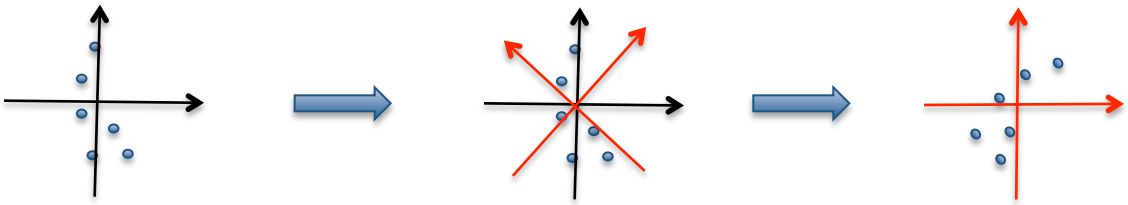


Recherche du Plus Proche Voisin



Construction d'une forêt d'arbre "randomisés" par rotation aléatoire

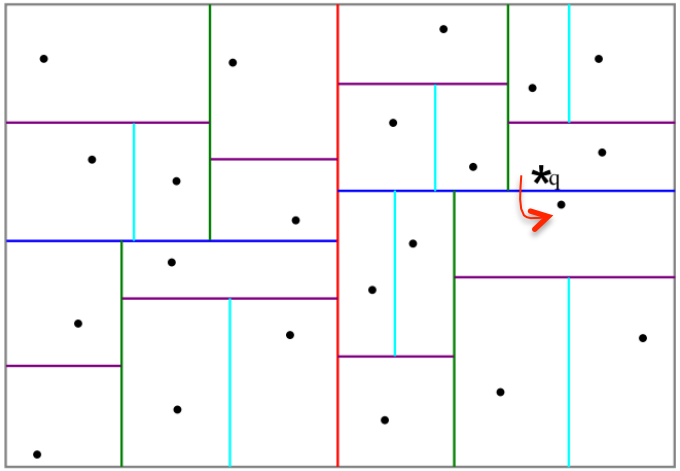
1. Appliquer une rotation aléatoire sur les données

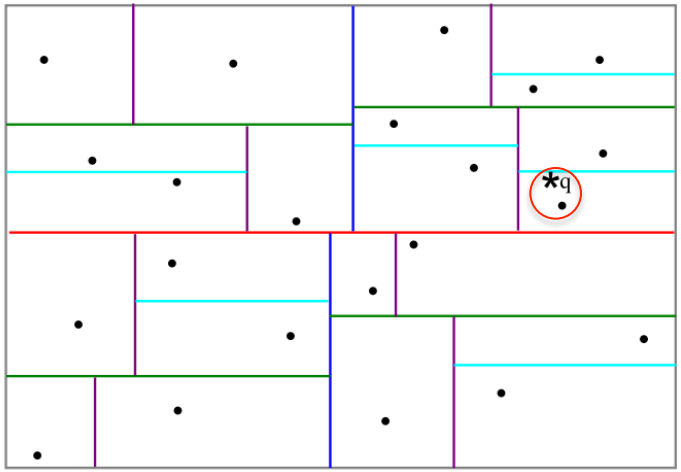


$$x' = R.x$$

2. Construire un kd-tree classique sur l'espace résultant
3. Répéter L fois pour construire L arbres

Intuition:



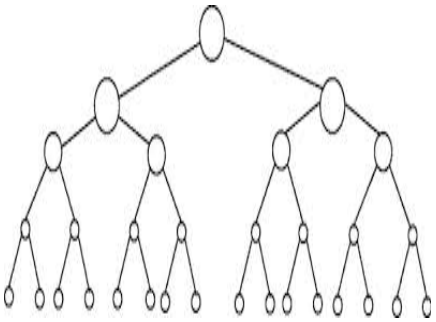


Construction d'une forêt d'arbre "randomisés" par tirage de composantes aléatoires

1. Tirage aléatoire d'une **série de comosantes** parmi les D composantes ayant la plus forte variance (exemple D=5)

{2,3,2,1,4,4,1,2,0,3,0,1,0,4,2,4,1,0…}

2. Construire un kd-tree en utilisant la série d'axes déterminée aléatoirement



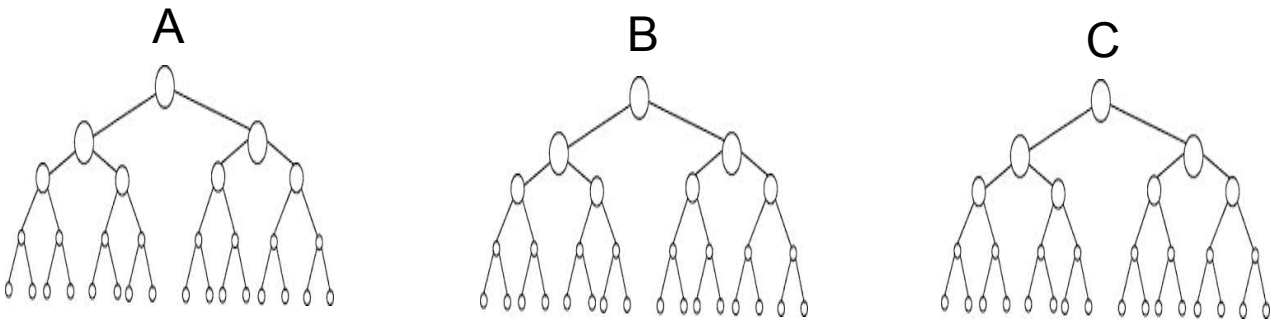
Split selon l'axe n°2
Split selon l'axe n°3
Split selon l'axe n°2
Split selon l'axe n°1
Split selon l'axe n°4

3. Répéter L fois pour construire L arbres

Avantages = construction et recherche plus simples & plus rapides car on travaille directement dans l'espace d'origine (pas de projection matricielle avant split ou best bin selection)

Recherche dans une forêt de kd-tree randomisés

On parcours tous les arbres en parallèle et on maintient une queue de priorité unique pour tous les arbres



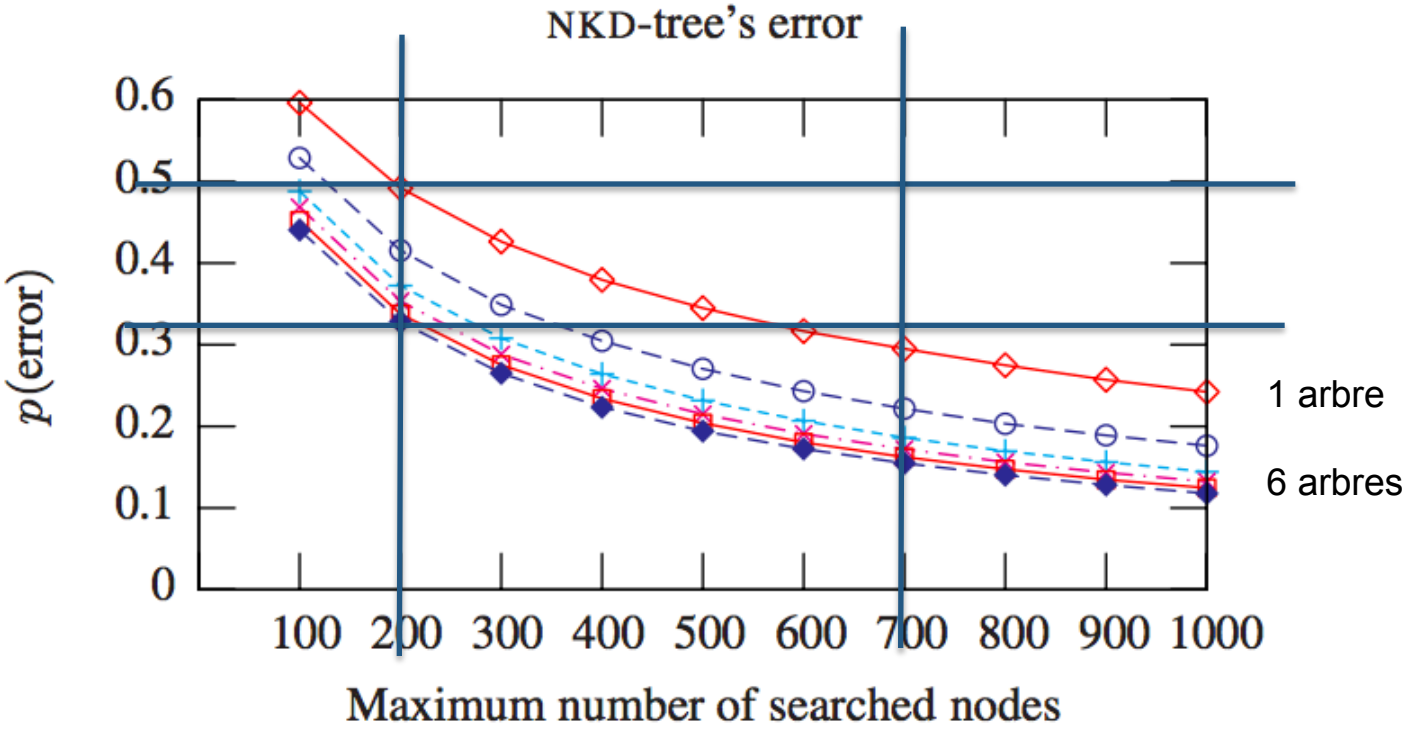
Priority Queue (construite en fonction de d(q,B)):

It1	{B0, A0, C0}
It2	{B2, A0, B1, C0}
It3	{A0, B4, B1, C0, B3}
It4	{B4, B1, A1, C0, A2, B3}
….	

Condition d'arrêt = nombre de feuilles visitées pour lesquelles on mis à jour le r_{NN}

Impact du nombre d'arbres et du nombre d'itérations

Il est plus rentable d'augmenter le nombre d'arbres que d'augmenter le nombre de nœuds visités (nombre d'itération de la priority queue)



Optimum sur le nombre d'arbres

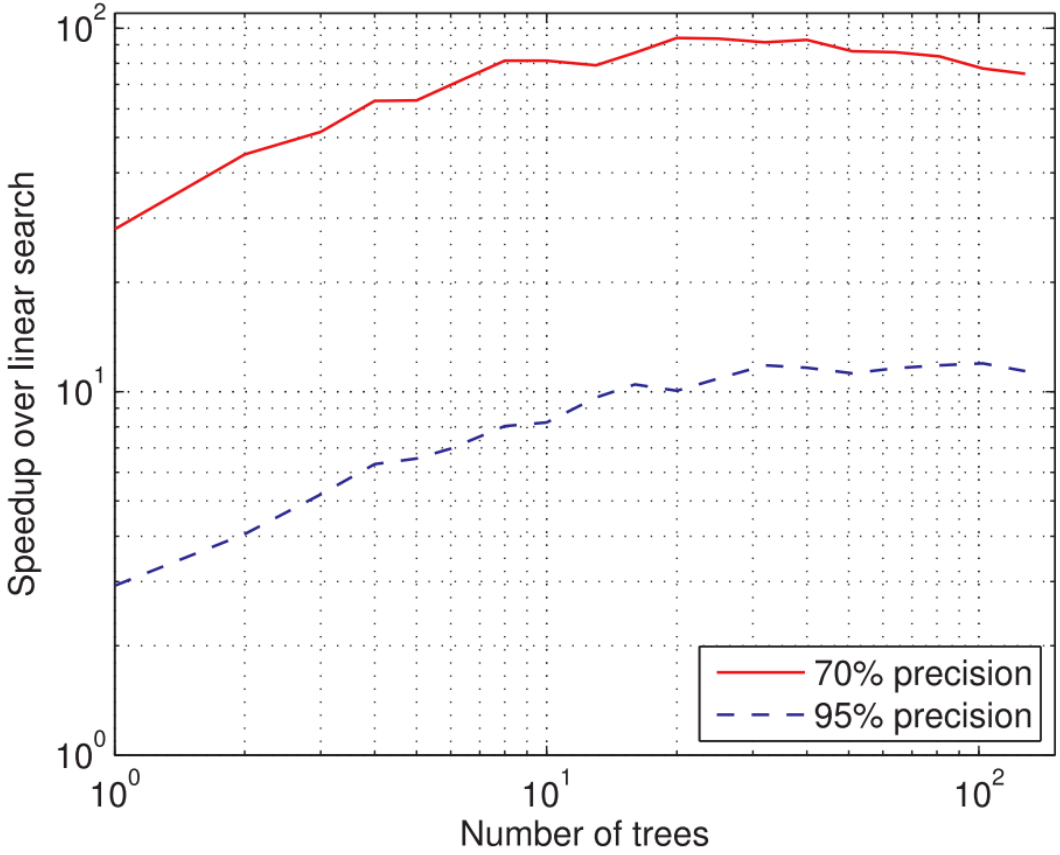


Fig. 1. Speedup obtained by using multiple randomized kd-trees (100K SIFT features data set).

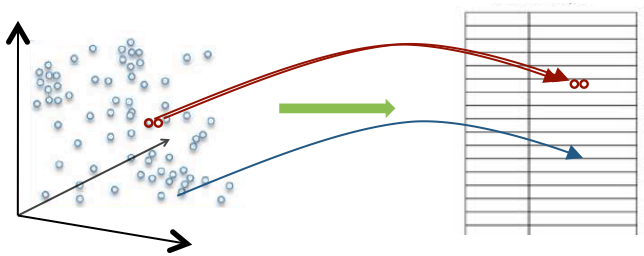
Etude de LSH

LSH: Intro

LSH (Localité Sensitive Hashing), méthode de hachage introduite par Gionis, Indyk et Motwani en 1998 pour la recherche approximative dans les espaces de grande dimensions

Sensible à la localité = la probabilité de collision des hash code est forte si les vecteurs sont proches et faible si les vecteurs sont éloignés

Formellement, selon Gionis et al. en 1998



$\mathcal{F}$ est une famille de fonctions $h : \mathbb{R}^d \rightarrow S$ satisfaisant les conditions suivantes pour deux points quelconques $p, q \in \mathbb{R}^d$ et une fonction h choisie aléatoirement parmi la famille $\mathcal{F}$:

- si $d(p,q) \leq R$, alors $Pr_{h \in H}[h(p) = h(q)] \geq P_1$
- si $d(p,q) \geq cR$, alors $Pr_{h \in H}[h(p) = h(q)] \leq P_2$

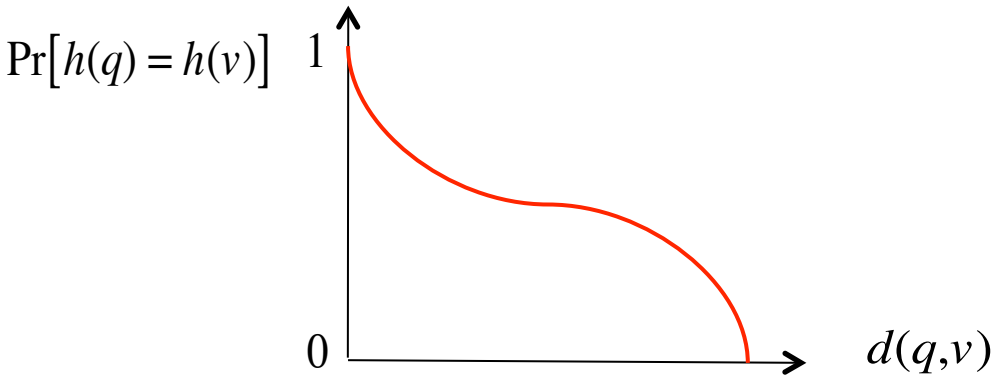
LSH: Intro

LSH (Localité Sensitive Hashing), méthode de hachage introduite par Gionis, Indyk et Motwani en 1998 pour la recherche approximative dans les espaces de grande dimensions

Pour certaines familles LSH, la probabilité de collision peut même s'exprimer comme une fonction décroissant avec la distance

$$\Pr\left[h_\theta(q) = h_\theta(v)\right]_{p_\theta} = f(d(q,v))$$

- θ is i.i.d drawn from a known **data independent distribution**
- f(d) is the sensitivity function (monotonically increasing from 0 to 1)



LSH sensible au produit scalaire

Exemple: famille LSH sensible au produit scalaire

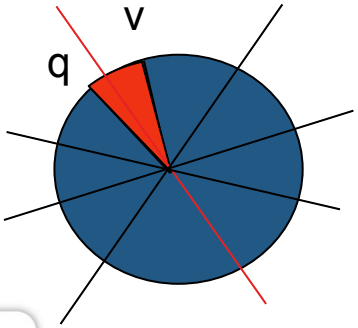
$$h(x) = \mathrm{sgn}(w^T x) \qquad p_w = N(0,I)$$

C'est le cas que l'on a déjà rencontré = projections aléatoires + quantification scalaire binaire

On peux montrer que la fonction de sensibilité vaut

Pour tous q, v $\in \mathbb{R}^d$ $\qquad \Pr\left[h_w(q) = h_w(v)\right]_w = 1 - \frac{1}{\pi}\cos^{-1}\left(\frac{q^T v}{\|q\|\|v\|}\right)$

Interprétation dans le cas normé:



$$\Pr\left[h_w(q) \neq h_w(v)\right]_w = angle(q,v)/\pi = \frac{1}{\pi}\cos^{-1}\left(q^T v\right)$$

$$= \frac{1}{\pi}\cos^{-1}\left(1 - \frac{\left(d(q,v)\right)^2}{2}\right)$$

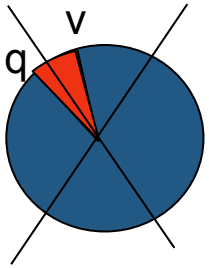
LSH sensible au produit scalaire

Exemple: famille LSH sensible au produit scalaire

$$h(x) = \mathrm{sgn}(w^T x) \qquad p_w = N(0,I) \qquad \Pr\left[h_w(q) = h_w(v)\right]_w = 1 - \frac{1}{\pi}\cos^{-1}\left(\frac{q^T v}{\|q\|\|v\|}\right)$$

On forme un hash code de D bits en utilisant D fonctions de hachage de la famille $\mathcal{F}$

$$z(x) = [h_1(x),\ldots,h_D(x)] = sign(Wx)$$



z(q) = 1 hash code = [0 1 0 1 0 1 1 0 …0 0 1 0]

z(v) = 1 hash code = [1 0 1 1 1 1 0 1 …0 1 0 0]

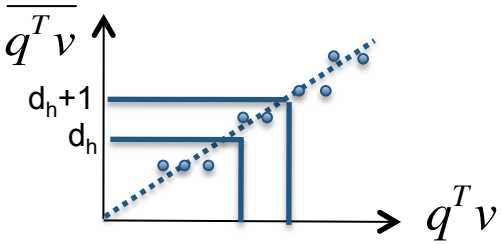
$$\approx \frac{D}{\pi}\cos^{-1}\left(\frac{q^T v}{\|q\|\|v\|}\right)$$

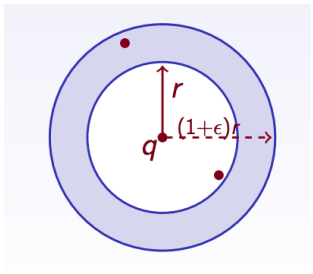
LSH sensible au produit scalaire

La distance de Hamming entre deux hash code est un estimateur de la probabilité de collision et donc du produit scalaire entre q et v

$$\overline{q^T v} = \|q\|\|v\|\cos\left(\frac{\pi}{D} d_h(z(q),z(v))\right)$$

Variance





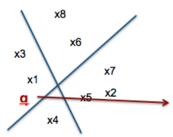
La variance $\sigma(k,q^T v)$ de l'estimateur décroit avec le nombre de bits et converge vers le produit scalaire exact

$$\lim_{k\to\infty} \overline{q^T v} = q^T v$$

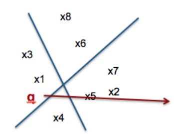
Les k plus proches voisins dans l'espace de Hammoing convergent vers les k plus proches voisins exactes

LSH: algorithme de base pour l'indexation et la recherche

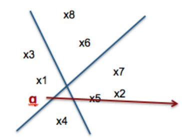
- Tables multiples: création de L tables dont les clé sont générées par k fonctions de hachage (hash code de taille k pour chaque table). Taille index O(Ln)
- Recherche = accès simple dans chaque table O(L)



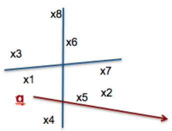
Clé	Valeur (ensemble de vecteurs)
z1	Id6:X6,Id8:X8
z2	Id2:X2, Id5:X5, Id7:X7
z3	Id1:X1,Id3:X3
z4	Id4:X4



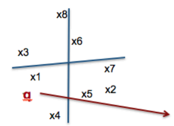
Clé	Valeur (ensemble de vecteurs)
z1	Id6:X6,Id8:X8
z2	Id2:X2, Id5:X5, Id7:X7
z3	Id1:X1,Id3:X3
z4	Id4:X4



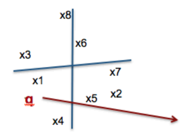
Clé	Valeur (ensemble de vecteurs)
z1	Id6:X6,Id8:X8
z2	Id2:X2, Id5:X5, Id7:X7
z3	Id1:X1,Id3:X3
z4	Id4:X4



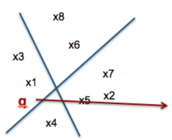
Clé	Valeur (ensemble de vecteurs)
z1	Id3:X3,Id8:X8
z2	Id6:X6
z3	Id2:X2, Id5:X5, Id7:X7
z4	Id1:X1, Id4:X4



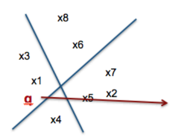
Clé	Valeur (ensemble de vecteurs)
z1	Id3:X3,Id8:X8
z2	Id6:X6
z3	Id2:X2, Id5:X5, Id7:X7
z4	Id1:X1, Id4:X4



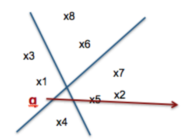
Clé	Valeur (ensemble de vecteurs)
z1	Id3:X3,Id8:X8
z2	Id6:X6
z3	Id2:X2, Id5:X5, Id7:X7
z4	Id1:X1, Id4:X4



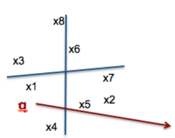
Clé	Valeur (ensemble de vecteurs)
z1	Id6:X6,Id8:X8
z2	Id2:X2, Id5:X5, Id7:X7
z3	Id1:X1,Id3:X3
z4	Id4:X4



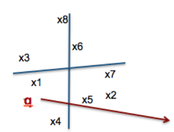
Clé	Valeur (ensemble de vecteurs)
z1	Id6:X6,Id8:X8
z2	Id2:X2, Id5:X5, Id7:X7
z3	Id1:X1,Id3:X3
z4	Id4:X4



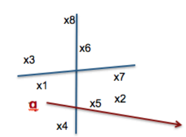
Clé	Valeur (ensemble de vecteurs)
z1	Id6:X6,Id8:X8
z2	Id2:X2, Id5:X5, Id7:X7
z3	Id1:X1,Id3:X3
z4	Id4:X4



Clé	Valeur (ensemble de vecteurs)
z1	Id3:X3,Id8:X8
z2	Id6:X6
z3	Id2:X2, Id5:X5, Id7:X7
z4	Id1:X1, Id4:X4



Clé	Valeur (ensemble de vecteurs)
z1	Id3:X3,Id8:X8
z2	Id6:X6
z3	Id2:X2, Id5:X5, Id7:X7
z4	Id1:X1, Id4:X4



Clé	Valeur (ensemble de vecteurs)
z1	Id3:X3,Id8:X8
z2	Id6:X6
z3	Id2:X2, Id5:X5, Id7:X7
z4	Id1:X1, Id4:X4

LSH: algorithme de base

Probabilité de collision dans une table de k bits:

$$\Pr\left[h_w(q) = h_w(v)\right]_w = f(d(q,v)) \Rightarrow \Pr\left[z(q) = z(v)\right]_W = \left(f(d(q,v))\right)^k$$

Fonction de sensibilité

Probabilité de non collision dans une table:

$$\Pr\left[z(q) \neq z(v)\right]_W = 1 - \left(f(d(q,v))\right)^k$$

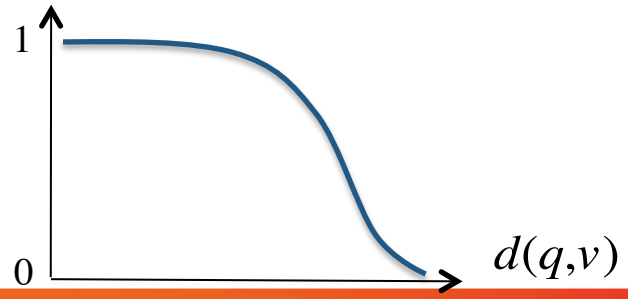
Très faible

Probabilité de non collision dans L tables:

$$\left(1 - \left(f(d(q,v))\right)^k\right)^L$$

Probabilité de collision dans au moins une table:

$$1 - \left(1 - \left(f(d(q,v))\right)^k\right)^L$$

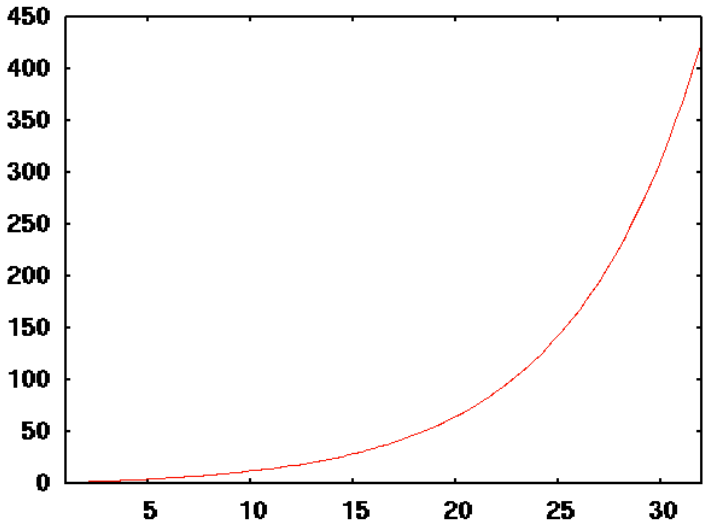


LSH: algorithme de base

Si on veux retrouver α (e.g 95%) des v tels que d(q,v)<r, il faut que la probabilité de collision dans au moins une table soit supérieure à α

$$1-\left(1-\left(f(r)\right)^{k}\right)^{L} > \alpha \quad \Rightarrow \quad L > \frac{\ln(1-\alpha)}{\ln(1-f(r)^{k})}$$

L (nombre de tables)



A qualité α constante et rayon r constant, le nombre de tables doit croitre pour compenser l'augmentation de k

k (nombre de bits par table)

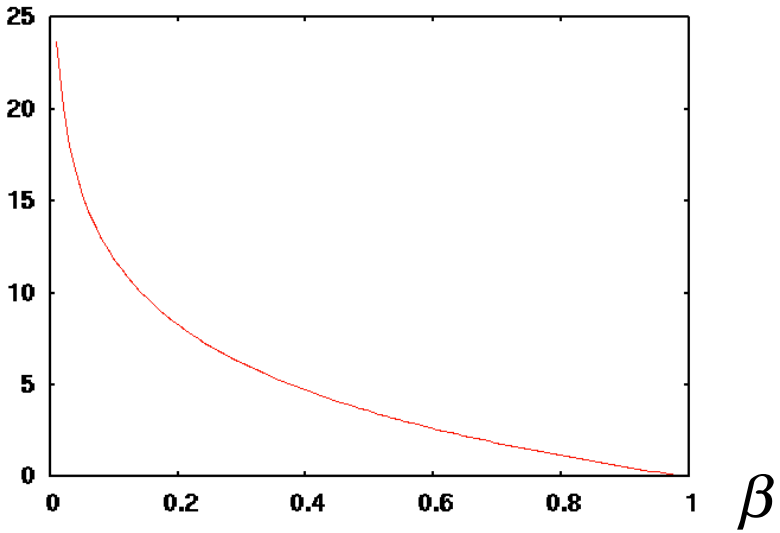
LSH: algorithme de base

D'un autre côté, l'augmentation de k permet de réduire la probabilté de fausses collisions dans chaque table. Si on veux que la probabilité de collision soit faible lorsque d(q,v)>r+ε

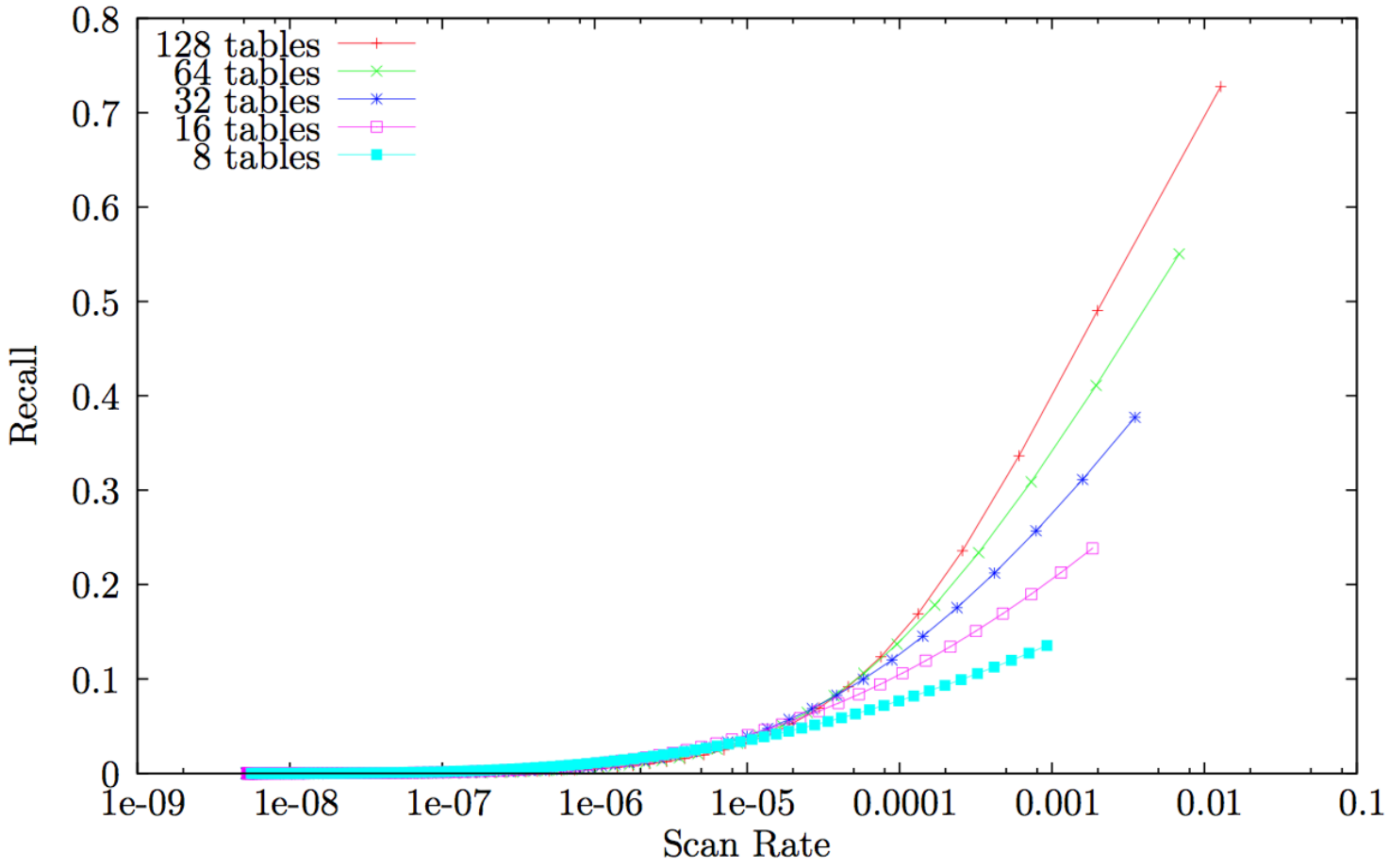
Proba de collision dans une table

$$n\left(f((1+\varepsilon)r)\right)^k < \beta \quad \Rightarrow \quad k > \frac{\ln(\beta)}{\ln(f((1+\varepsilon)r))}$$

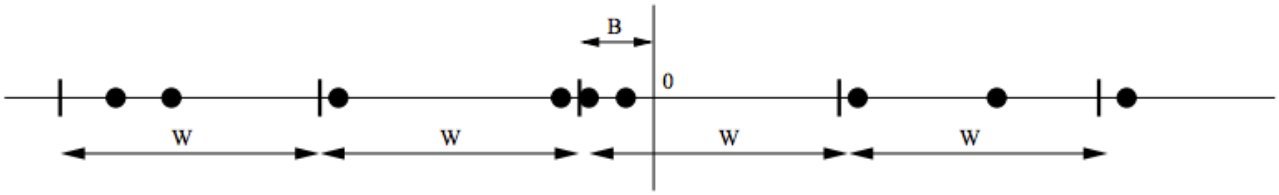
k (nombre de bits par table)



Impact du nombre de tables



LSH sensible aux distances L_p



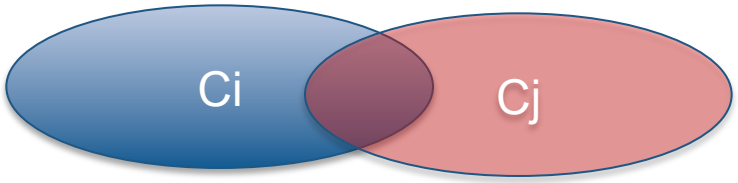
- Consider $h_{\boldsymbol{a},b} \in \mathcal{H}^w$, $h_{\boldsymbol{a},b}(\boldsymbol{v}) : \mathcal{R}^d \rightarrow \mathcal{N}$
- $\boldsymbol{a}$ is a d dimensional random vector whose each entry is drawn from a p-stable distr
- b is a random real number chosen uniformly from $[0, w]$ (random shift)
- $h_{\boldsymbol{a},b}(\boldsymbol{v}) = \lfloor \frac{\boldsymbol{a} \cdot \boldsymbol{v} + b}{w} \rfloor$

LSH sensible à la distance de Jaccard

MinHash: the min-wise independent permutations locality sensitive hashing scheme

Jaccard distance = mesure de similarité entre des ensembles d'objets

$$sim_J(C_i, C_j) = \frac{|C_i \cap C_j|}{|C_i \cup C_j|}$$



- View sets as columns of a matrix; one row for each element in the universe. a_{ij} = 1 indicates presence of item i in set j
- Example

C_1	C_2
0	1
1	0
1	1
0	0
1	1
0	1

$sim_J(C_1, C_2) = 2/5 = 0.4$

Hash function=

- Randomly permute rows
- Hash $h(C_i)$ = index of first row with 1 in column C_i
- Suprising Property

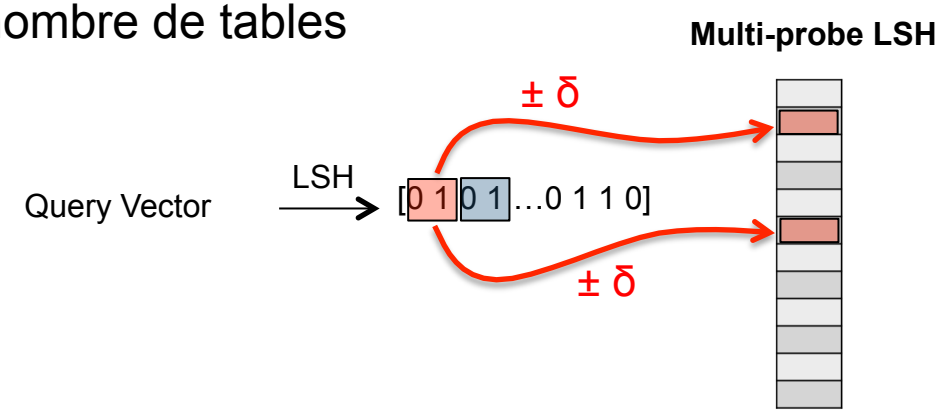
$$P[h(C_i) = h(C_j)] = sim_J(C_i, C_j)$$

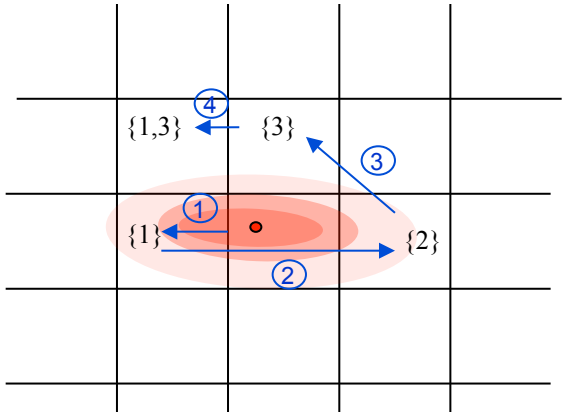
Limitations de LSH et variantes

Problème de LSH: espace mémoire et multi-probe LSH

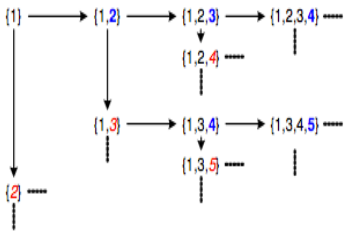
LSH peux nécessiter un grand nombre de tables pour atteindre des qualités suffisantes ce qui rend la méthode problématique lorsque les données sont grandes

→ Utilisation d'accès multiples dans chaque table pour réduire le nombre de tables





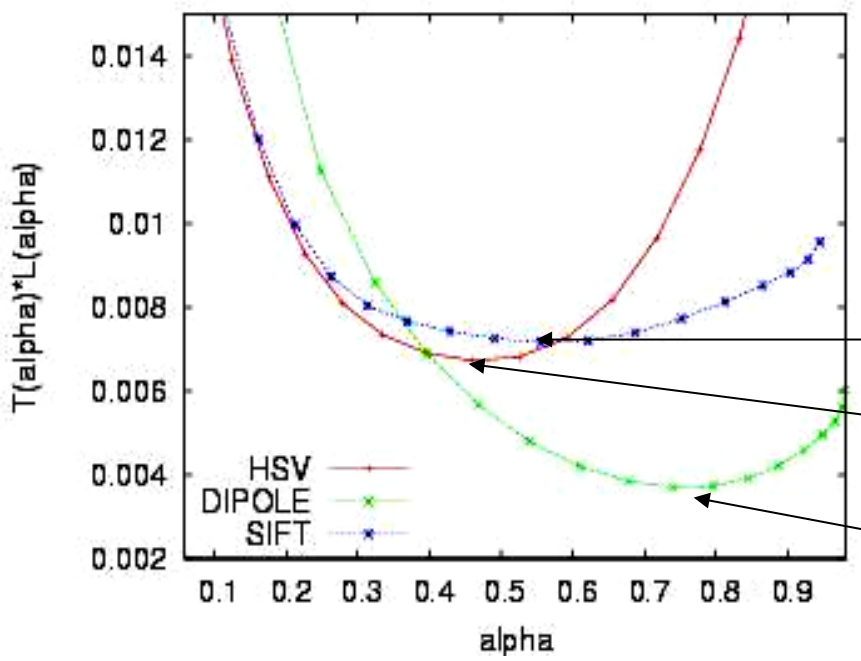
→ Algorithme glouton pour les bucket de la plus probable à la moins probable



Nombre optimal de tables << LSH

$$\alpha_T = 1-(1-\alpha)^L \longrightarrow L = \frac{\ln(1-\alpha_T)}{\ln(1-\alpha)}$$

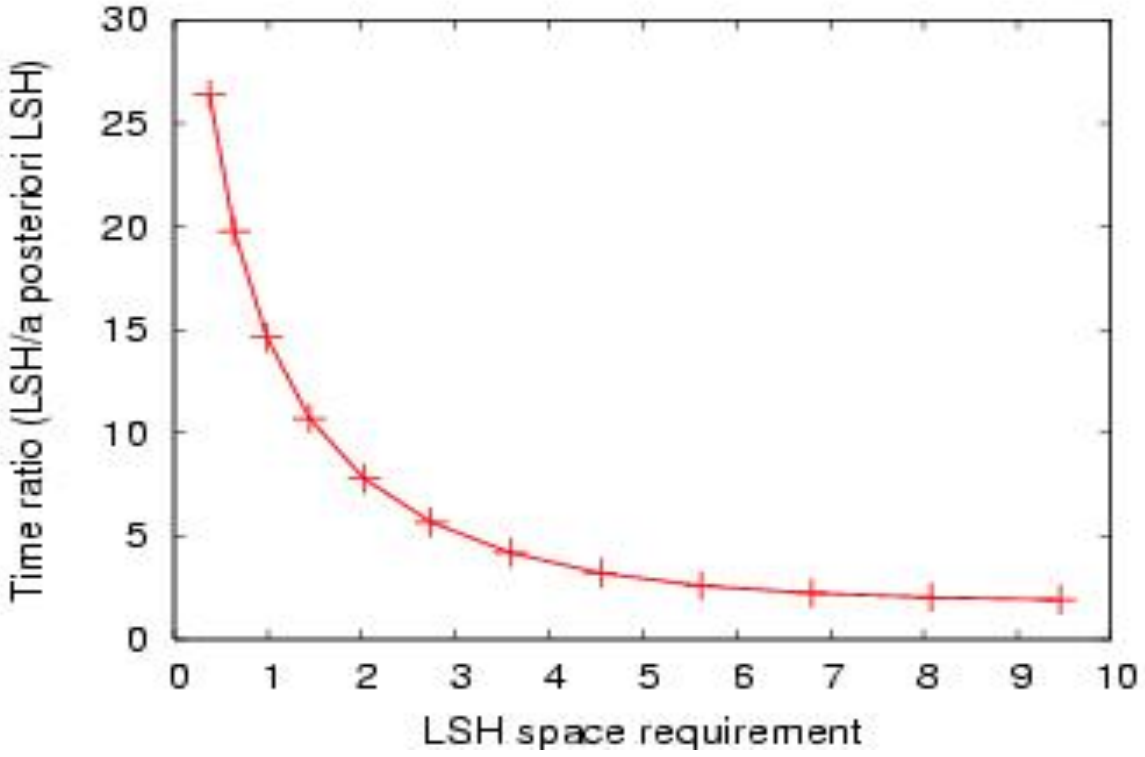
$$\alpha >> \alpha_{LSH}$$



	Multi-probe	LSH
HSV	L=4	L=253
DIPOLE	L=5	L=253
SIFT	L=2	L=36

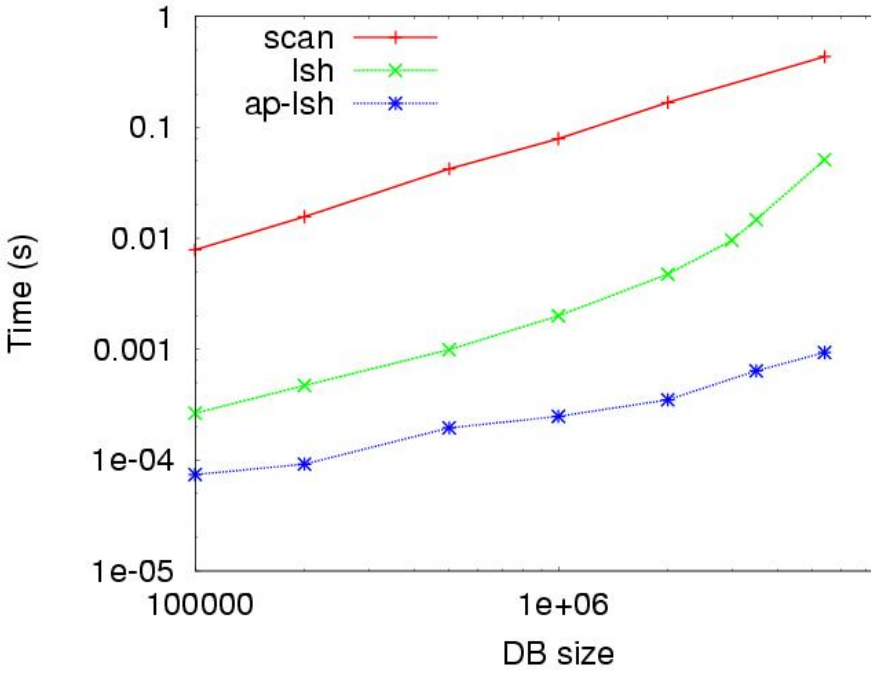
Time/Space performances vs. LSH

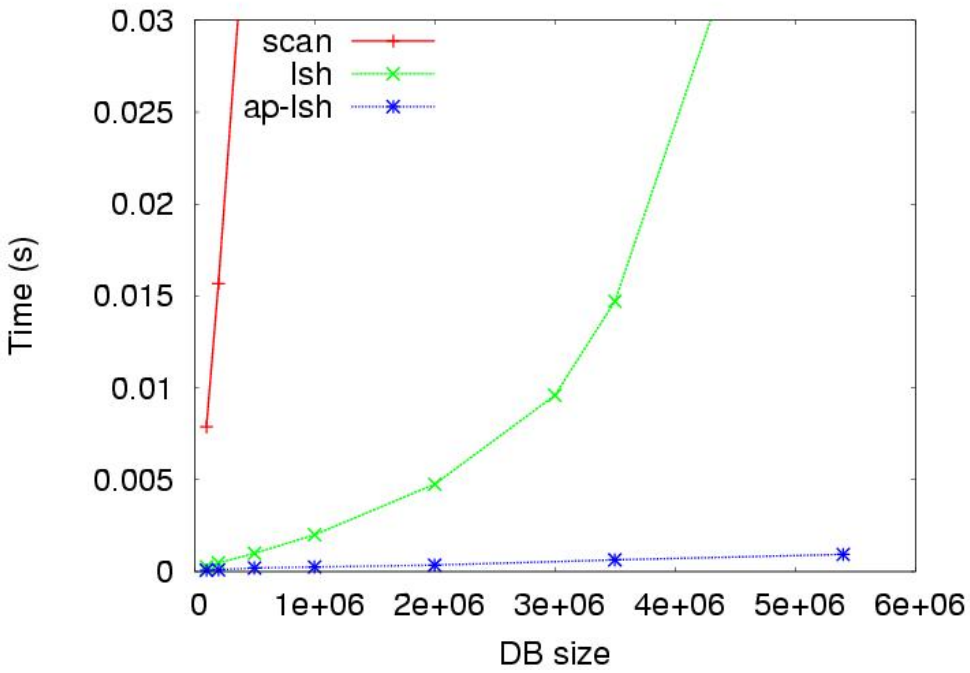
- Time Gain vs. LSH space requirements



Influence de la taille de la base

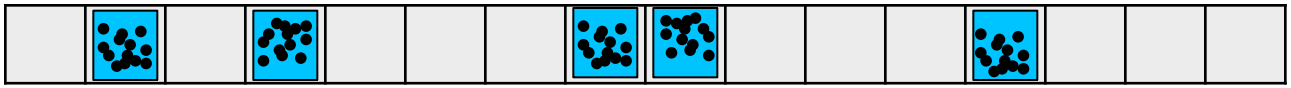
Limited memory (4 Gb)





Autre problème de LSH: mauvais balancement

Sur certaines données réelles, LSH peux conduire à des partitions loin de l'uniformité. Certaines bucket peuvent concentrer presque tous les points:



→ Augmentation du nombre de fausses collisions
→ Augmentation du temps de raffinement
→ Problématique dans les contexte distribués (load balancing)

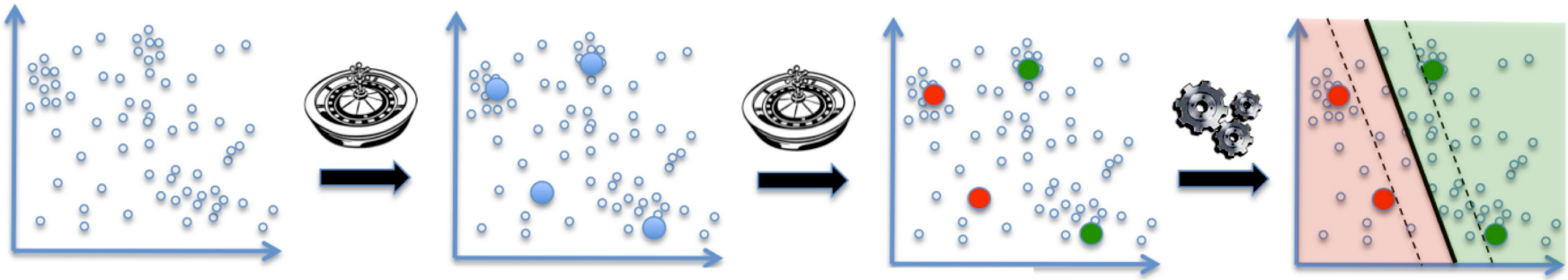
Solution: utiliser des fonctions de hâchage dépendant des données. De nombreuses fonctions « data dependent » ont été proposées récemment dans la littérature (spectral-hashing, KLSH, RMMH, etc.)

Random Maximum Margin Hashing

RMMH = une famille de fonctions de hachage basée sur le **partitionnement aléatoire des données**

Avantage = fonctions indépendantes + adaptabilité aux données

Basé sur **un concept théorique nouveau** = **l'apprentissage aléatoire de classifieurs** (1 par fonction de hachage = 1 par bit)



M échantillons aléatoires

Labels positifs et négatifs purement aléatoires !!

SVM

$$h(\mathbf{x}) = sgn(\mathbf{w}_m.\mathbf{x} + b_m)$$

Random Maximum Margin Hashing

Pourquoi ça marche ?

Cas extrême: M (nombre d'échantillons) = N (toute la base)

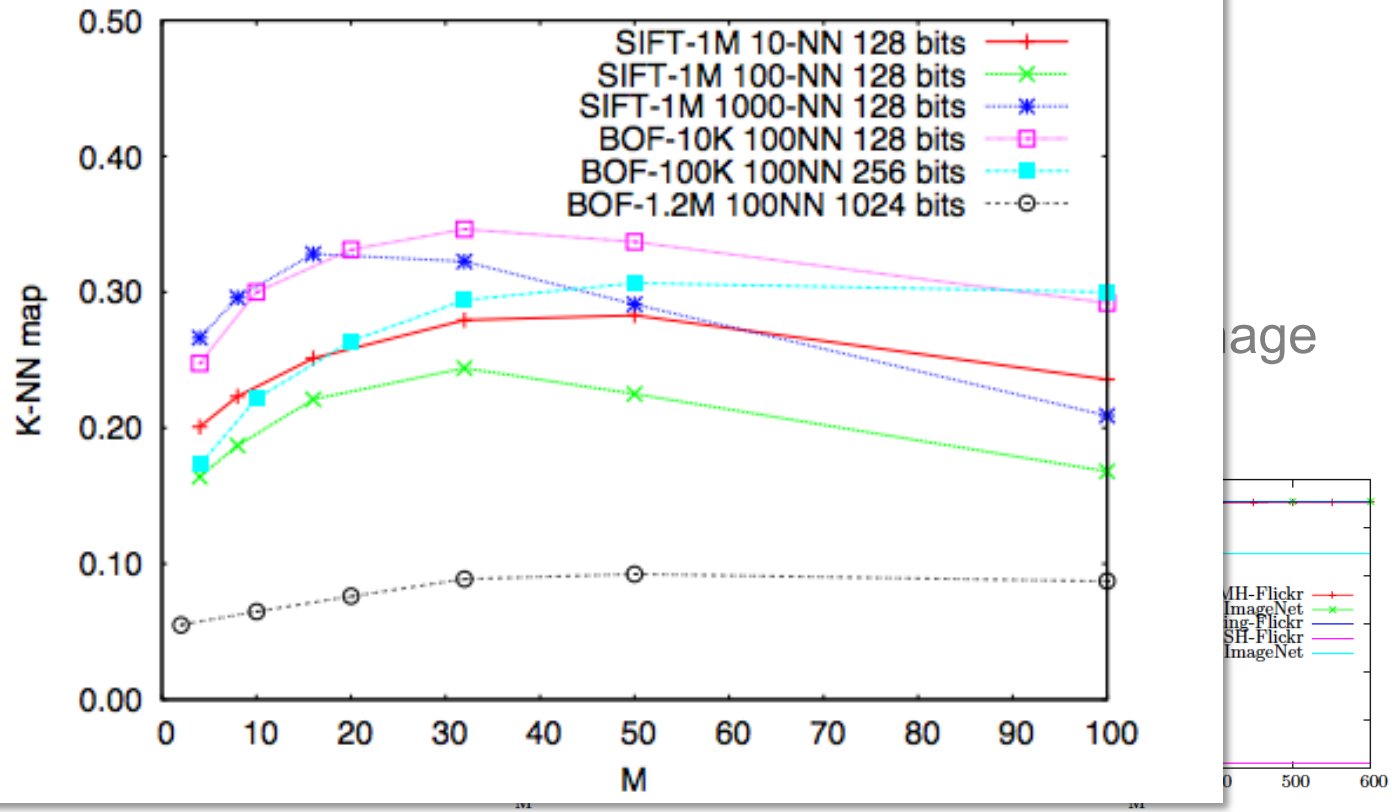
Uniformit

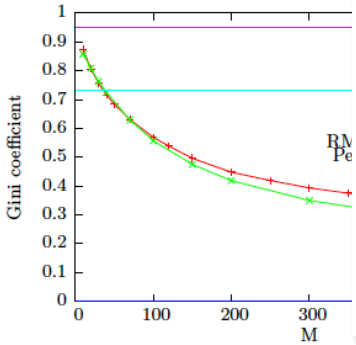
Overfitti

M trop faible:

M optimal = **c**

age





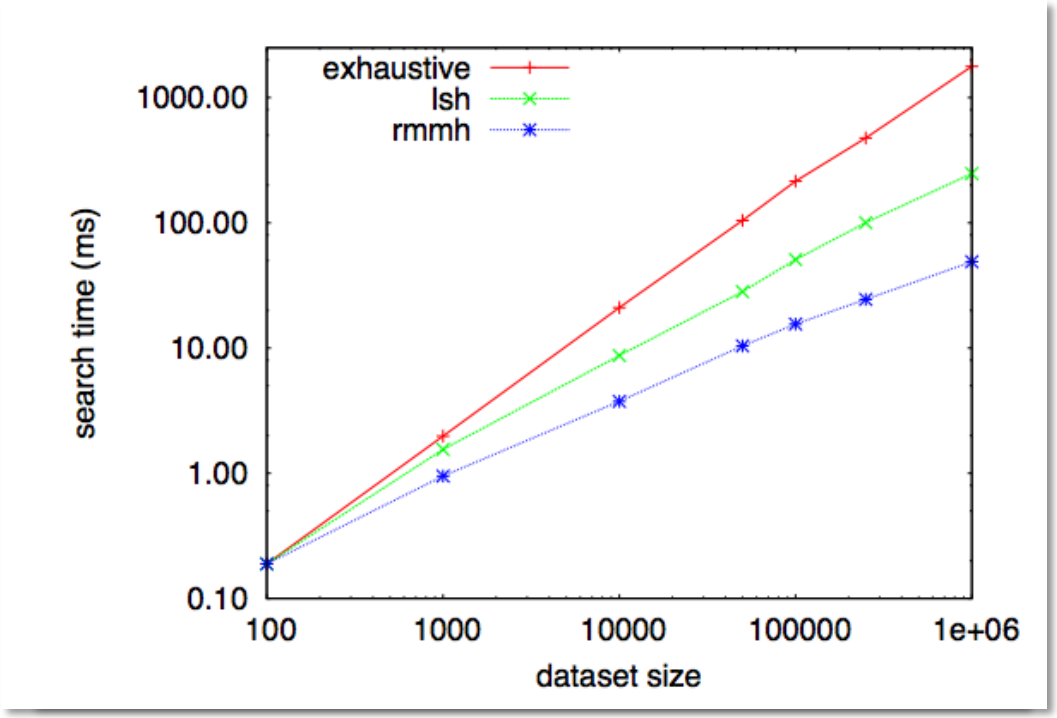
(a) Gini coefficient (b) Avg. Max. bucket size (c) Number of non empty buckets

Random Maximum Margin Hashing

Evaluation on ImageNet

1.2M images described by BovW featrues (dim 1000)

Supervised classification across 1000 semantic categories, 1000-NN



Présentation de deux librairies: FLANN et VLfeat

VL-Feat

VLFeat.org

VL-Feat est une librairie généraliste de vision par ordinateur avec un focus particulier sur la **recherche d'information par le contenu visuel**

VL-Feat contient un **grand nombre d'algorithmes** pour

1. l'extraction de descripteurs visuels (SIFT, Fisher vector, VLAD, MSER)
2. l'indexation et la recherche approximative de k-PP
3. L'apprentissage et la classification (SVM, etc.)

VL-Feat a été développé par un consortium d'universitaires internationaux et est sous licence open source BSD

VL-Feat est écrit en C++ mais intègre aussi des API pour C, MATLAB et Octave

URL: http://www.vlfeat.org/

VL-Feat:algorithmes

Local feature frames	Covariant feature detectors
HOG features	SIFT detector and descriptor
Dense SIFT	LIOP local descriptor
MSER feature detector	Distance transform
Fisher Vector and VLAD	Gaussian Mixture Models
K-means clustering	Agglomerative Infromation Bottleneck
Quick shift superpixels	SLIC superpixels
Support Vector Machines (SVMs)	KD-trees and forests
Plotting AP and ROC curves	Miscellaneous utilities
Integer K-means	Hierarchical integer k-means

VL-Feat: kd-tree

VL-Feat intègre un module de recherche de k plus proches voisins dans des kd-tree classiques

Le search se fait avec un algorithme **best-bin-first**

Exemple de construction d'un arbre 2D avec l'API matlab:

```
X = rand(2, 100) ;
```

```
kdtree = vl_kdtreebuild(X) ;
```

Exemple de recherches de PPV d'une requête Q (toujours en 2D):

```
Q = rand(2, 1) ;
[index, distance] = vl_kdtreequery(kdforest, X, Q) ;
```

```
[index, distance] = vl_kdtreequery(kdtree, X, Q, 'NumNeighbors', 10) ;
```

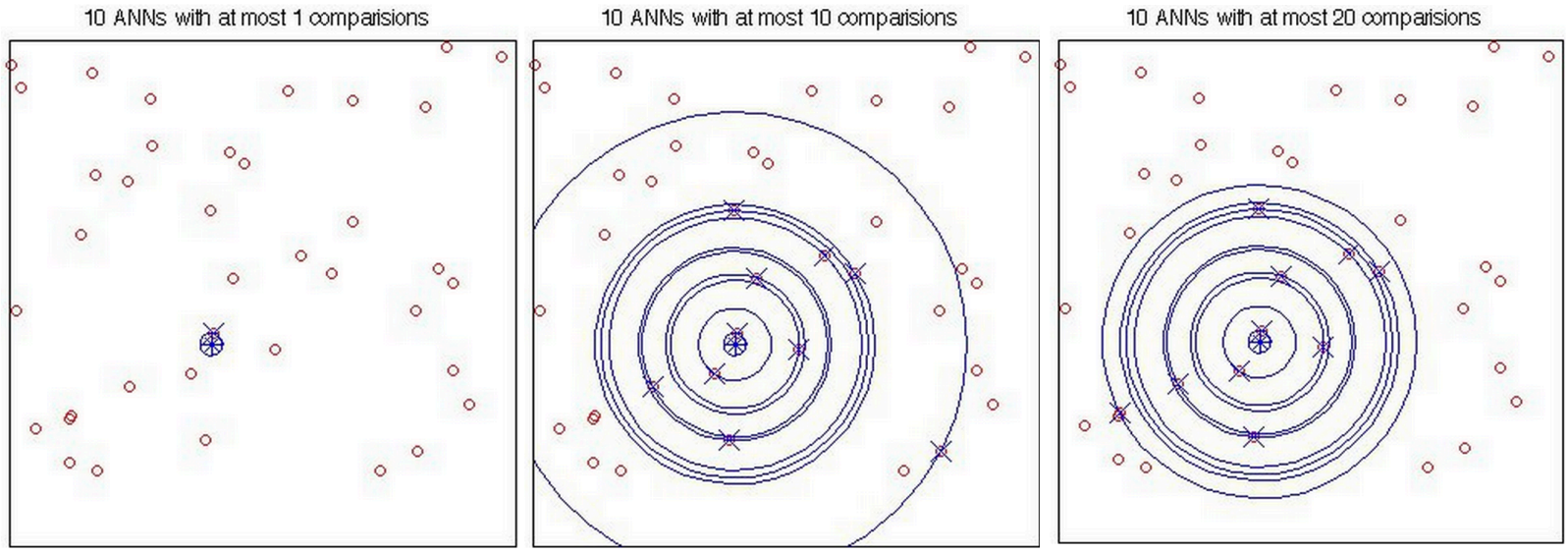
« index » contient l'identifiant des plus proches voisins, « distance » les distances associées

VL-Feat: randomized kd-tree

Exemple de recherche **approximative** de PPV d'une requête Q (toujours en 2D):

```
Q = rand(2, 1) ;
```

```
[index, distance] = vl_kdtreequery(kdtree, X, Q, 'NumNeighbors', 10, 'MaxComparisons', 15) ;
```



VL-Feat: randomized kd-trees forest

VL-Feat intègre un module de recherche de k plus proches voisins dans **une forêt de kd-tree aléatoires** (avec la méthode des axes aléatoires).

Exemple de construction et de recherche dans une forêt d'arbres 2D avec l'API matlab:

```
kdtree = vl_kdtreebuild(X, 'NumTrees', 4) ;
[index, distance] = vl_kdtreequery(kdtree, X, Q) ;
```

Quelques méthodes de l'API C++:

VlKDForest *	**vl_kdforest_new** (vl_type dataType, **vl_size** dimension, **vl_size** numTrees, **VlVectorComparisonType** normType) Create new KDForest object. More...
void	**vl_kdforest_build** (**VlKDForest** *self, **vl_size** numData, void const *data) Build KDTree from data. More...
vl_size	**vl_kdforest_query** (**VlKDForest** *self, **VlKDForestNeighbor** *neighbors, **vl_size** numNeighbors, void const *query) Query the forest. More...
void	**vl_kdforest_set_max_num_comparisons** (**VlKDForest** *self, **vl_size** n) Set the maximum number of comparisons for a search. More...

VL-Feat: k-means

VL-Feat intègre également des algorithmes de clustering en particulier de nombreuses variantes de **kmeans** hierarchical k-means

Caculer k-means avec matlab (avec l'algorithme de Loyd par défaut):

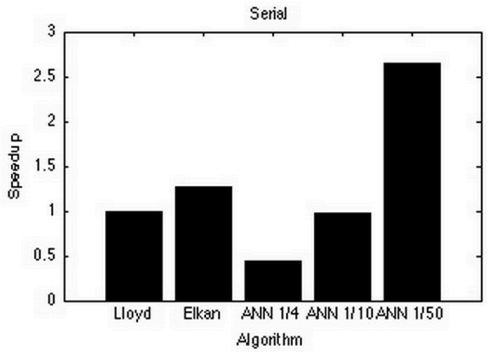
```
numData = 5000 ;
dimension = 2 ;
data = rand(dimension,numData) ;
```

```
numClusters = 30 ;
[centers, assignments] = vl_kmeans(data, numClusters);
```

"centers" contient les centres de clusters (30 vecteurs), "assignments" les affectations de chaque point à chaque cluster

Versions optimisées: by setting the `'Algorithm'` parameter to `'Lloyd'`, `'Elkan'` or `'ANN'`

→ La version 'ANN' utilise une forêt de kd-tree randomisés pour affecter les points au cluster les plus proches



FLANN

FLANN

FLANN est une librairie de **recherche approximative de k plus proches voisins** pour les espaces de grande dimensionalité

FLANN contient un **ensemble d'algorithmes** pour la recherche de k-PP ainsi qu'un système permettant de **choisir automatiquement** l'algorithme **optimal et les paramètres optimaux pour un ensemble de données particulier**

FLANN a été développé par l'université de British Columbia et est sous licence open source BSD

FLANN est écrit en C++ mais intègre aussi des API pour C, MATLAB et Python

URL: http://www.cs.ubc.ca/research/flann/

FLANN

Les 4 principaux algorithmes

1. Exhaustive Search
2. Multi-probe Locality Sensitive Hashing
3. Randomized kd-trees
4. Priority search k-means tree

FLANN

Les 4 principaux algorithmes

1. Exhaustive Search
2. Multi-probe Locality Sensitive Hashing
3. Randomized kd-trees
4. **Priority search k-means tree**

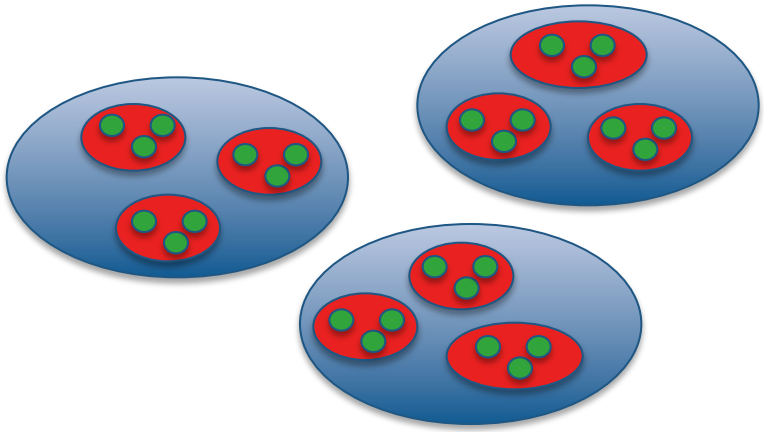
Priority Search K-means tree

Exploite plus efficacement la structure naturelle des données en les regroupant par proximité selon toutes les composantes (contrairement au kd-tree qui partitionne selon une composante)

Construction d'un k-means tree

```
Function hk_means(X,k) {
        if (size(X)<k) leaf node = X;
        else [X1, …,Xk]=K-means(X,k);
        For m=1 à k {
                create node Xm;
                hk_means(Xm,k)
        }
```

→ Complexity O(n.k.log(n))



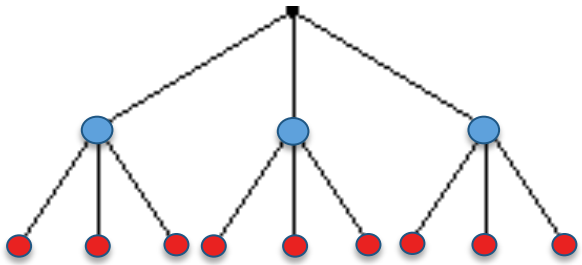


Illustration de l'impact de k (branching factor)

K=4 K=32 K=128

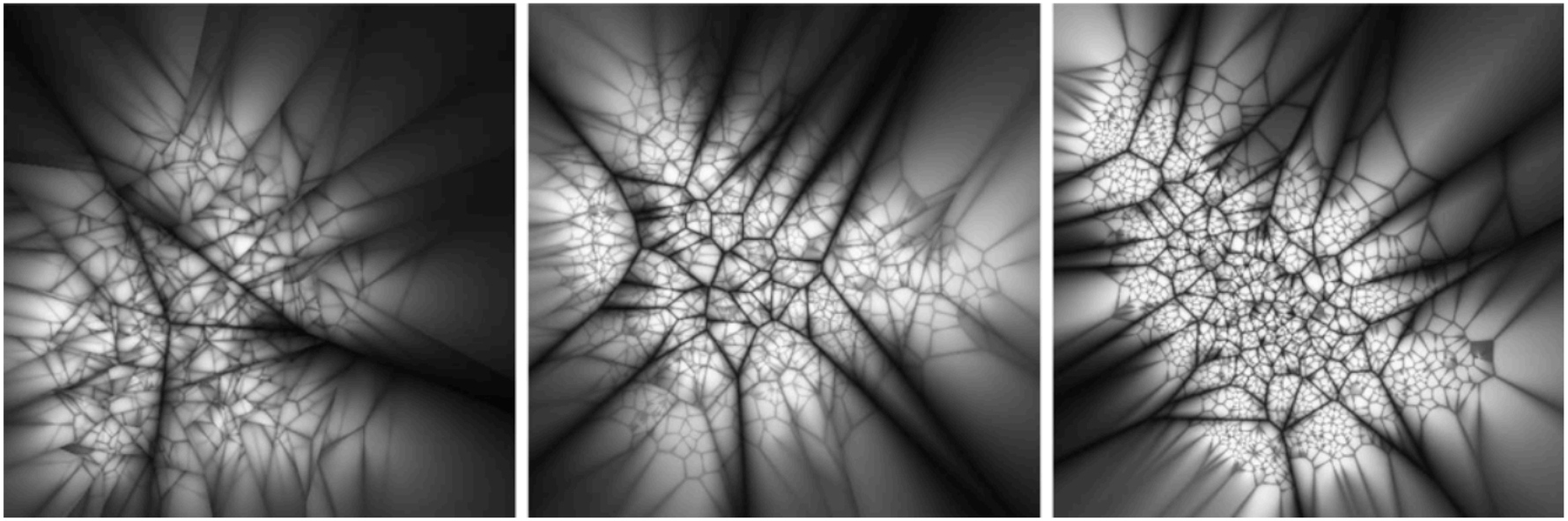
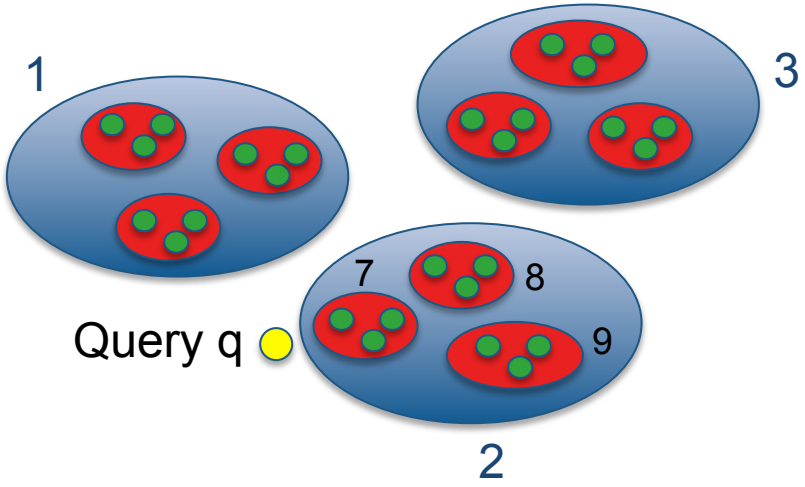
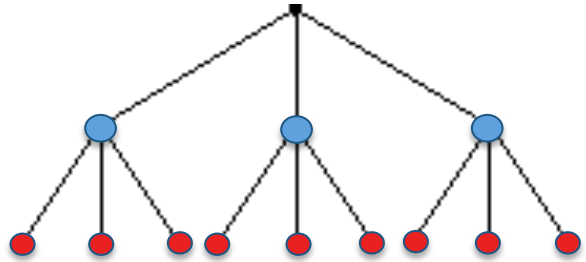


Fig. 3. Projections of priority search k-means trees constructed using different branching factors: 4, 32, 128. The projections are constructed using the same technique as in [26], gray values indicating the ratio between the distances to the nearest and the second-nearest cluster centre at each tree level, so that the darkest values (ratio ≈ 1) fall near the boundaries between k-means regions.

Priority Search K-means tree

On maintient les régions à parcourir dans une priority queue:





Priority Queue:

It1	{0}
It2	{2,1,3}
It3	{7,1,8,9,3}

La distance de la requête à un noeud est calculée par la distance au centre de gravité du cluster

Lorsque l'on rencontre un p' tel que $d(p',q) > r_{NN}$ mise à jour du rayon de recherche : $r_{NN}=d(p',q)$

FLANN: quick start

- C++

```
// file flann_example.cpp

#include <flann/flann.hpp>
```

- Python

```
from pyflann import *
from numpy import *
from numpy.random import *

dataset = rand(10000, 128)
testset = rand(1000, 128)

flann = FLANN()
result,dists = flann.nn(dataset,testset,5,algorithm="kmeans",
               branching=32, iterations=7, checks=16);
```

```
index.knnSearch(query, indices, dists, nn, flann::SearchParams(128));

flann::save_to_file(indices,"result.hdf5","result");
```

1

FLANN: LSH parameters

```
struct LshIndexParams : public IndexParams

{

LshIndexParams(unsigned int table_number = 12,//nombre
de tables de hachage

unsigned int key_size = 20, //nombre de bits par table

unsigned int multi_probe_level = 2);//nombre de switch de
bits pour le multi-probe

};
```

FLANN: Priority Search Kmeans parameters

```
struct KMeansIndexParams : public IndexParams

{

KMeansIndexParams( int branching = 32, //nombre de clusters par noeud

int iterations = 11, //nombre d'itérations pour chaque appal de k-means

flann_centers_init_t centers_init = FLANN_CENTERS_RANDOM,//type
d'initialisation

float cb_index = 0.2 );

};
```

FLANN: search parameters

```
struct SearchParams

{

int checks;//nombre de feuilles max à visiter (“early stopping”)
float eps;//valeur de epsilon pour recherche à (1+epsilon) près
bool sorted;//tri des voisins ou non lorsqu’on fait une requête à un rayon près
int max_neighbors;//nombre max de voisins pour requête à un rayon près
tri_type use_heap;//utilisation ou non d’un max heap pour la recherche de knn
int cores;//nombre de coeurs affectés à la recherche

};
```

FLANN: Sélection automatique du meilleur algorithme

Les 4 principaux algorithmes: Exhaustive Search, Multi-probe Locality Sensitive Hashing, Randomized kd-trees, Priority search k-means tree

Problème:

Le choix de l'algorithme optimal dépend des données: dimensionalité, taille et structure

Chaque algorithme a lui même des paramètres ayant une influence forte sur les performances

Espace de paramètres de grande dimensionalité –> impossible de tester toutes les combinaisons

FLANN: Sélection automatique du meilleur algorithme

Formalisation du problème:

1. Un ensemble de paramètres à optimiser:

$$\theta = (\theta_1, \theta_2, \ldots, \theta_m)$$

2. Une fonction de coût:

Temps de recherche — Temps de construction — Coût mémoire : taille index / taille données

$$c(\theta) = \frac{s(\theta) + w_b b(\theta)}{\min_{\theta \in \Theta}(s(\theta) + w_b b(\theta))} + w_m m(\theta)$$

3. Problème: $\min_{\theta \in \Theta} c(\theta)$

FLANN: Sélection automatique du meilleur algorithme

L'estimation de $c(\theta)$ pour une valeur de θ donnée se fait de manière empirique en construisant l'index et en recherchant un certain nombre d'échantillons (proportionnel au nombre de points dans la base).

Si m=5 paramètres $\theta = (\theta_1, \theta_2, ..., \theta_5)$ et qu'on veux tester 20 valeurs par paramètre:

nb d'estimation empiriques = 20^5 = 3,200,000

Soit 3,200,000 construction d'index O(n log n) et n fois 3,200,000 recherches O(log n) ou O(n)

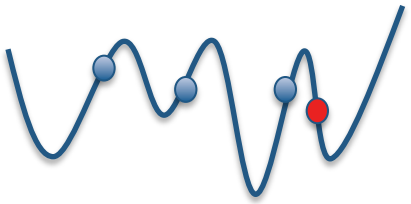
→ Impossible, des siècles de calcul

FLANN: Sélection automatique du meilleur algorithme

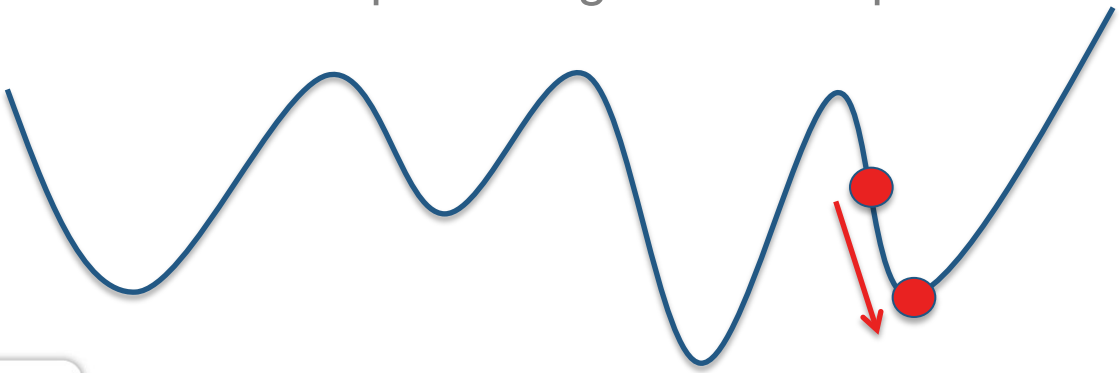
Solution FLANN = estimation globale grossière + raffinement local par

Estimation globale grossière, par exemple m=2 paramètres (dont le type d'algorithme et le param principal de chaque algo) + 4 valeurs par paramètres:

nb d'estimation empiriques = 4^2 = 16



Raffinement local par un algorithme d'optimisation (de type simplex)



FLANN: Sélection automatique du meilleur algorithme

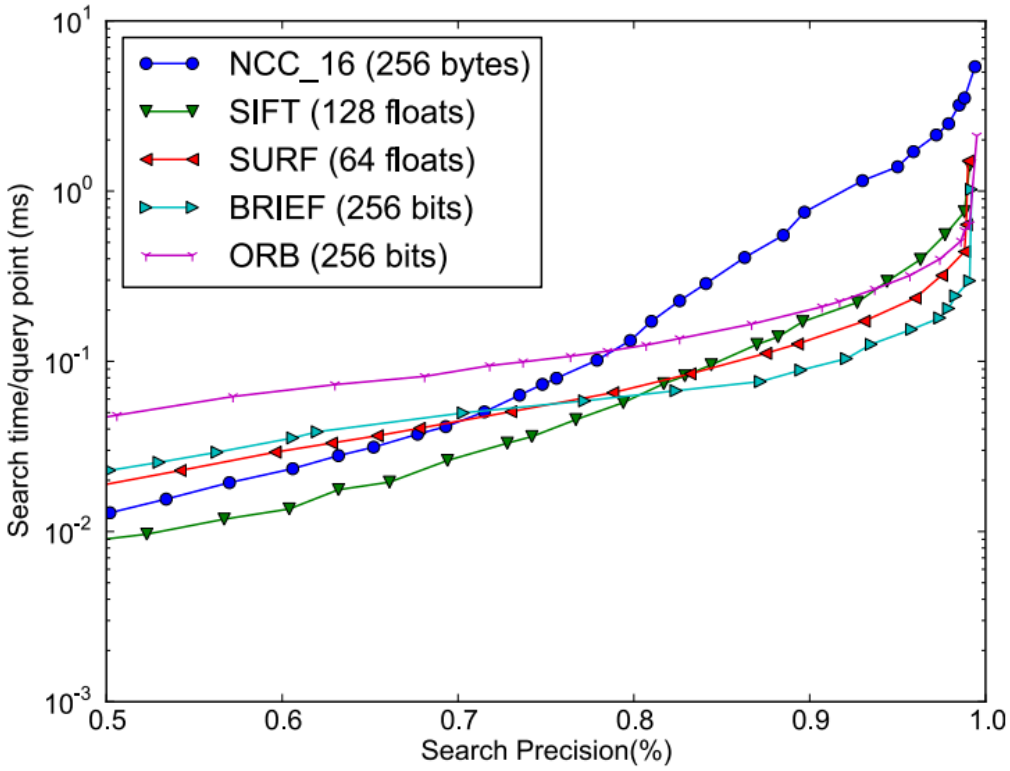
```
struct AutotunedIndexParams : public IndexParams
{
AutotunedIndexParams( float target_precision = 0.9,//précision par rapport aux NN exacts

float build_weight = 0.01,//pondération du temps de construction de l'index dans la fonction de coût

float memory_weight = 0, //pondération de l'utilisation mémoire de l'index dans la fonction de coût

float sample_fraction = 0.1 ); //fraction de la base à utiliser pour l'estimation de la fonction de coût (nombre de requêtes)
};
```

FLANN: temps de recherche brutes en fonction de la qualité



FLANN: illustration sélection auto

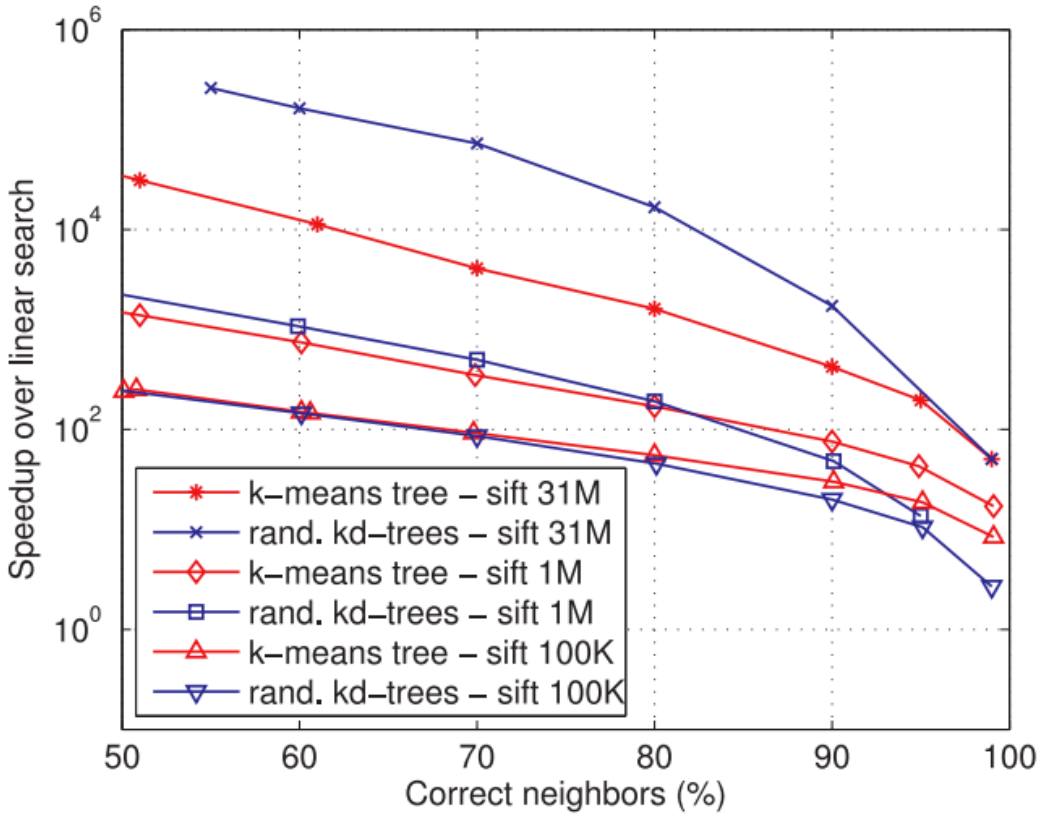
Algorithmes choisis par la sélection automatique de FLANN

Importance temps construction

Importance utilisation mémoire

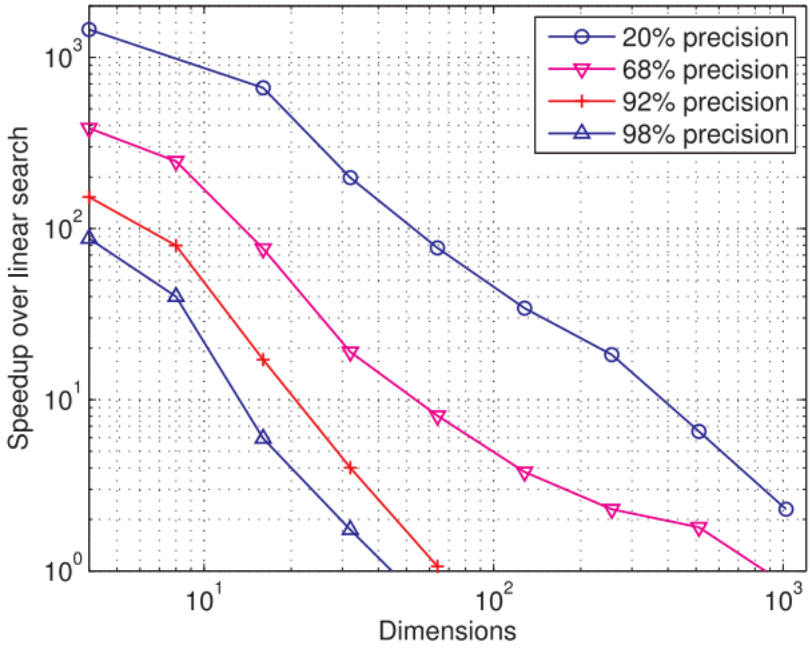
$Pr.(\%)$	w_b	w_m	Algorithm Configuration	Dist. Error	Search Speedup	Memory Used	Build Time
60%	0	0	k-means, 16, 15	0.096	181.10	0.51	0.58
	0	1	k-means, 32, 10	0.058	180.9	0.37	0.56
	0.01	0	k-means, 16, 5	0.077	163.25	0.50	0.26
	0.01	1	kd-tree, 4	0.041	109.50	0.26	0.12
	1	0	kd-tree,1	0.044	56.87	0.07	0.03
	*	∞	kd-tree,1	0.044	56.87	0.07	0.03
90%	0	0	k-means, 128, 10	0.008	31.67	0.18	1.82
	0	1	k-means, 128, 15	0.007	30.53	0.18	2.32
	0.01	0	k-means, 32, 5	0.011	29.47	0.36	0.35
	1	0	k-means, 16, 1	0.016	21.59	0.48	0.10
	1	1	kd-tree,1	0.005	5.05	0.07	0.03
	*	∞	kd-tree,1	0.005	5.05	0.07	0.03

FLANN: peformances vs. taille de la base

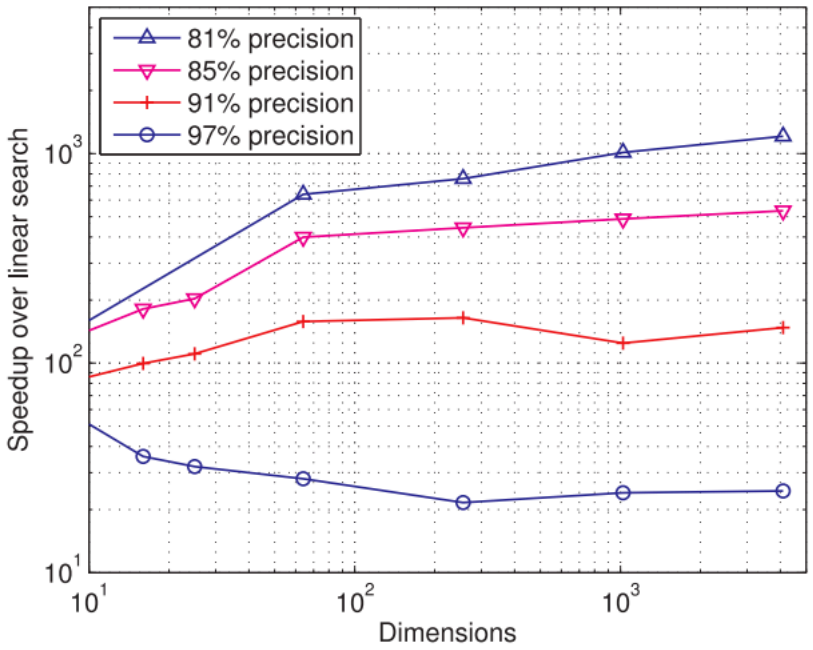


FLANN: peformances vs. dimensionalité

Données synthétiques uniformes



Données réelles



FLANN: distributed search

FLANN permet de traiter les requêtes en parallèle sur plusieurs machines

En partitionnant aléatoirement et équitablement les données entre les machines

Algorithm 5 Searching a distributed index on a comput cluster

Input: query Q, search parameters P

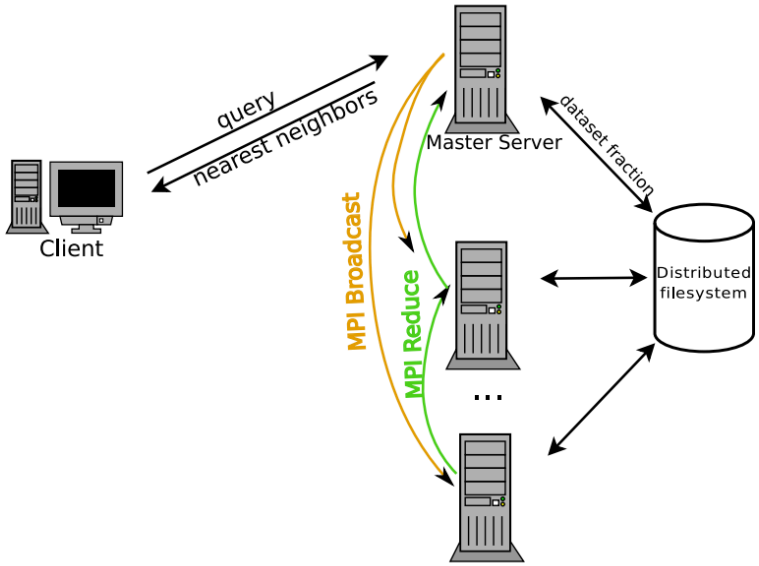
procedure SEARCHINDEX(Q,P)

1: MPI_broadcast(Q,P) // broadcast query and parameter to all processes

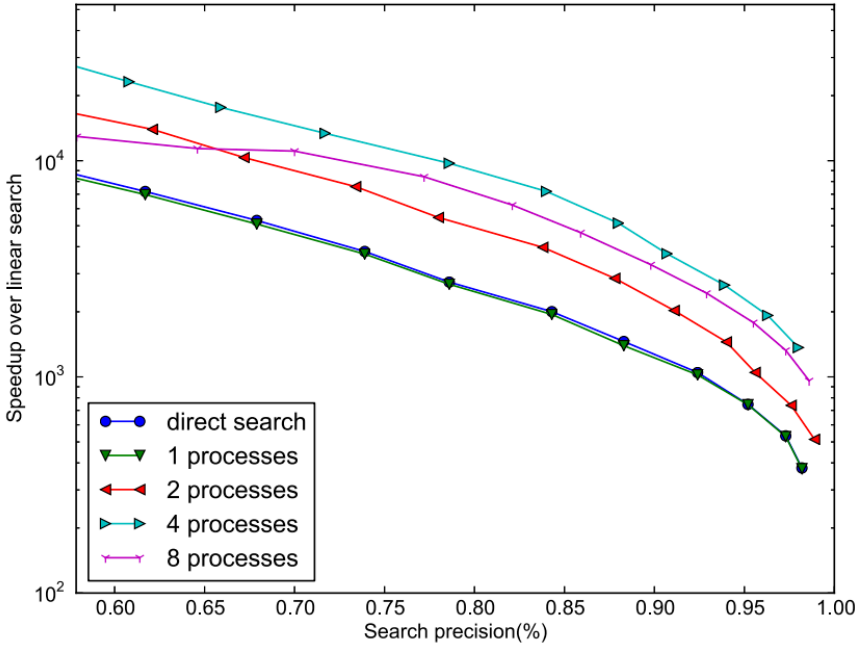
2: $NN_i \leftarrow$ run nearest neighbor search with query Q an parameters P on each process i

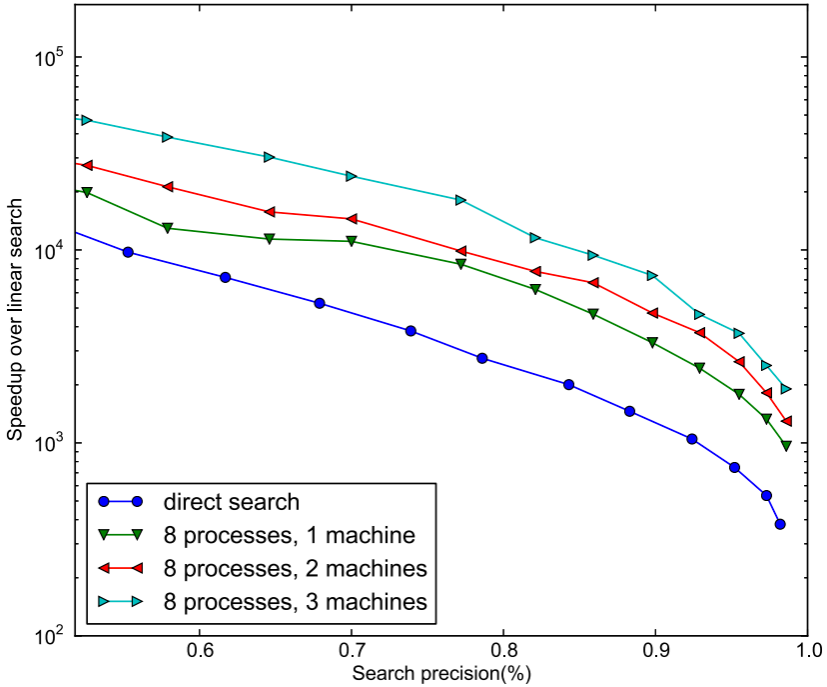
3: $NN \leftarrow$ MPI_reduce(NN_i) // merge results using a MP reduce operation

4: **return** NN



FLANN: distributed search





Pour aller plus loin avec FLANN

FLANN - Fast Library for Approximate Nearest Neighbors

User Manual

Marius Muja, mariusm@cs.ubc.ca
David Lowe, lowe@cs.ubc.ca

January 24, 2013

http://www.cs.ubc.ca/research/flann/uploads/FLANN/flann_manual-1.8.4.pdf

KNN-graph approximation

Graphes des plus proches voisins

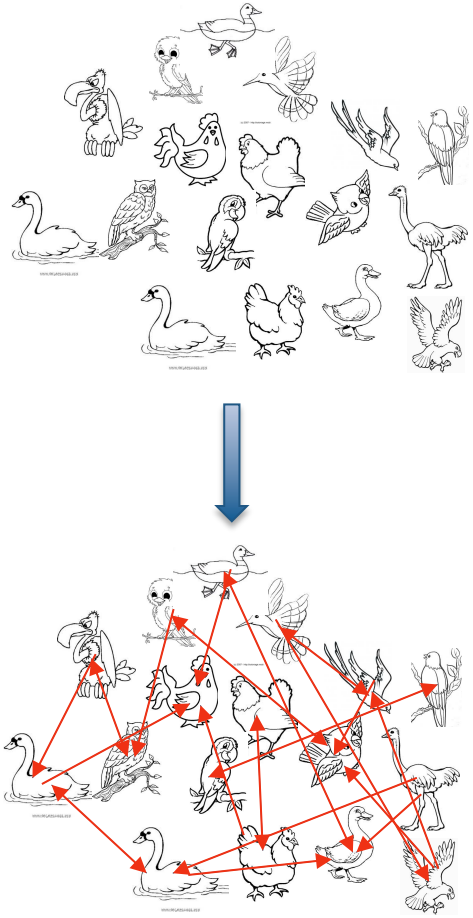
Structure clé … pour diverses applications

Notamment:

- Suggestion, complétion dans les moteurs de recherche
- Filtrage collaboratif
- Analyse media sociaux

Impliquant:

- Des objets complexe:
  - Grandes dimensions
  - Mesures de similarités complexes
- Bases de données à échelle mondiale



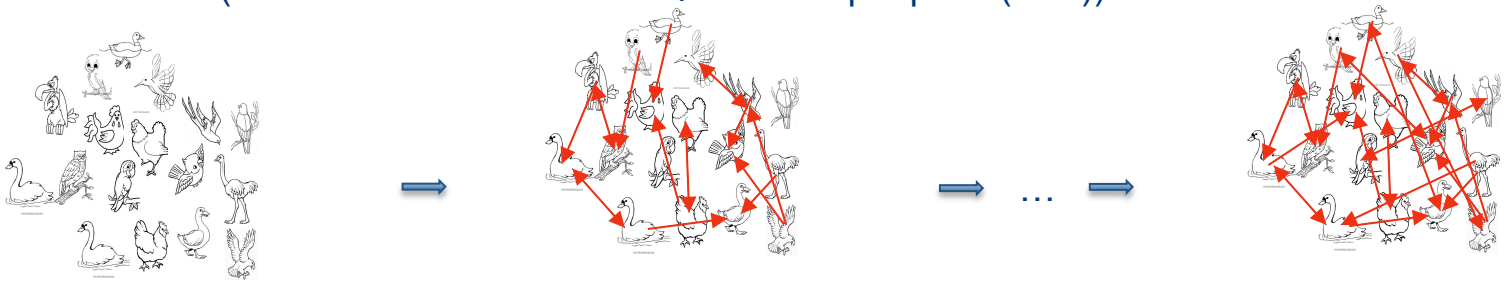
État de l'art

Approches itératives (brute force, processer les nœuds de manière **itérative**):

- Exacte: recherche **exhaustive** des plus proches voisins $O(n^2)$)

Approches récursives:

- NN-Descent* (méthode de référence complexité empirique $O(n^{1.14})$)

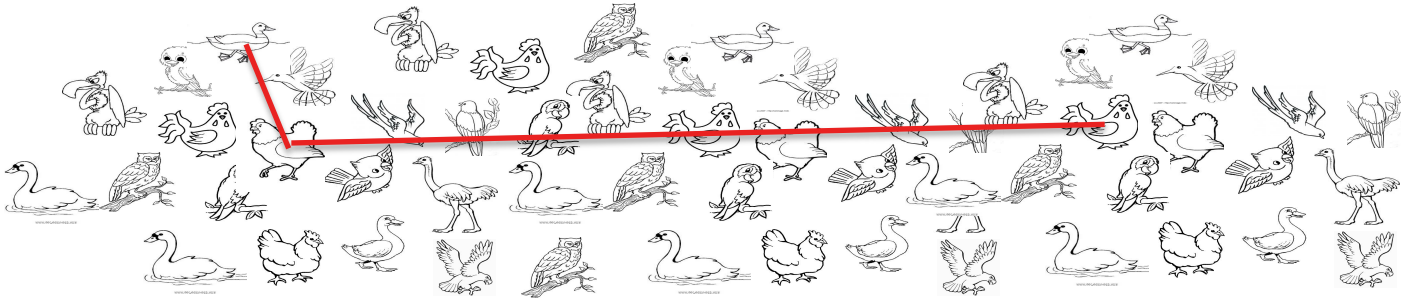


- Approche par jointure dans des tables de hachage sensible à la localité, performances similaires mais plus facilement distribuables

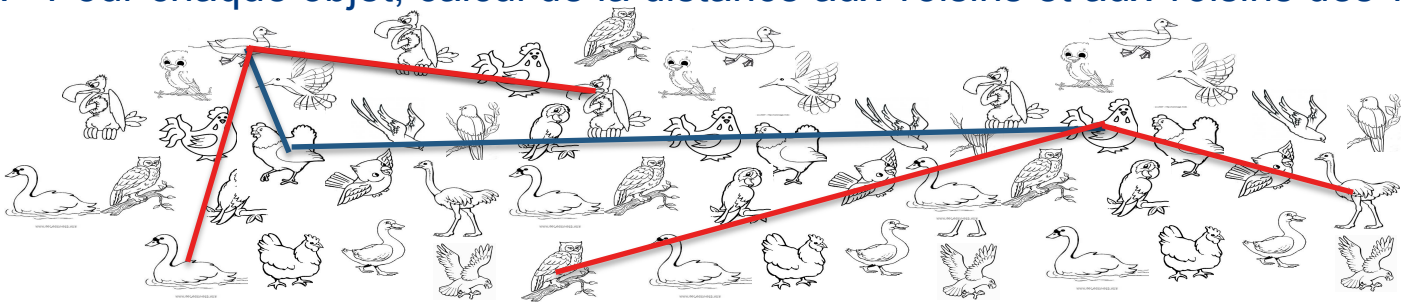
	T_1	T_2	T_3	T_4	T_5	T_6	T_7
00			a,b,c,d	c	a,b,c,d	b,d	
01				a,b			
10		a,b,d		d		a,c	a,c
11	a,b,c,d	c					b,d

NN descent: principe

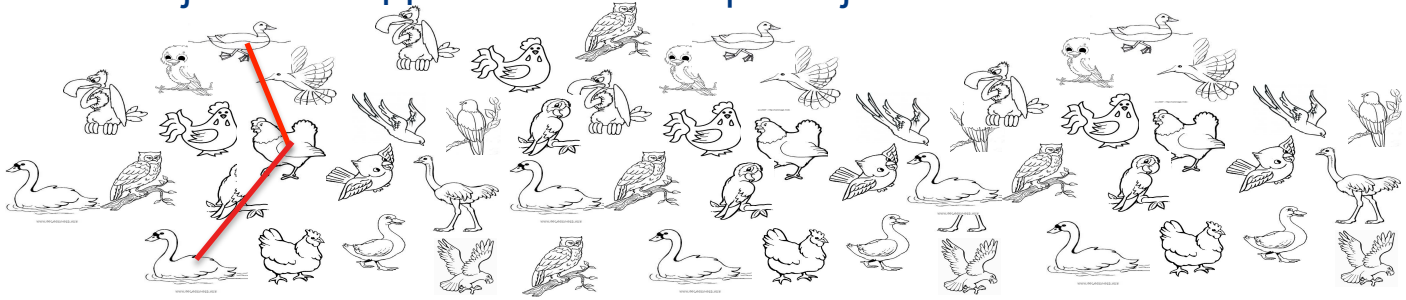
1. Tirage aléatoire de k voisins pour chaque objet (ex: k=2)



2. Pour chaque objet, calcul de la distance aux voisins et aux voisins des voisins



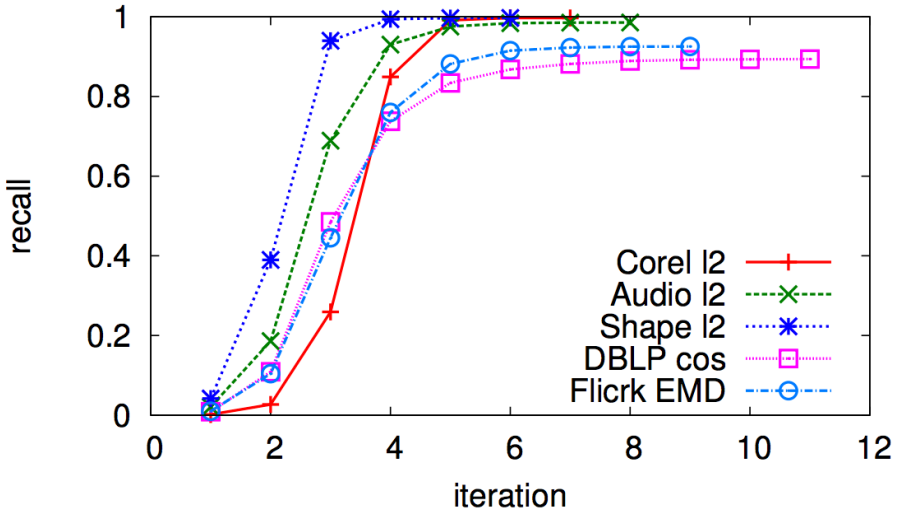
3. Mise à jour des k-pp voisins de chaque objet



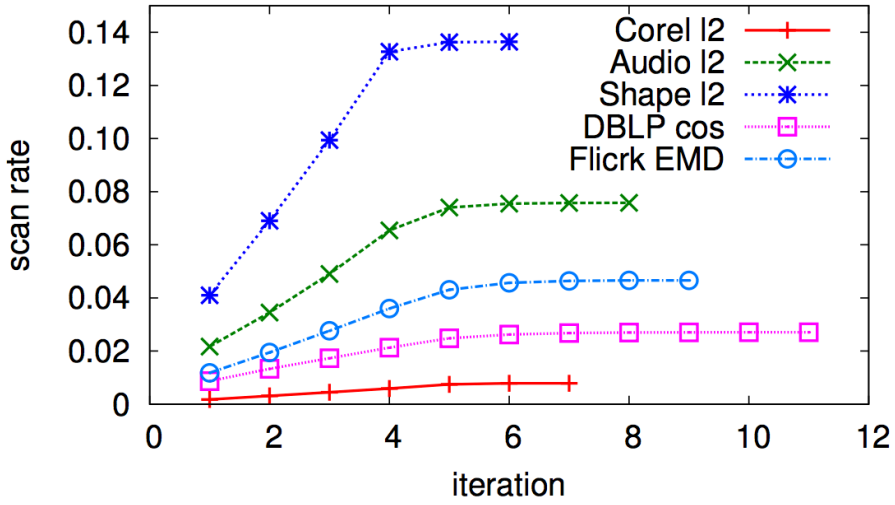
4. Répéter m fois

NN descent: performances

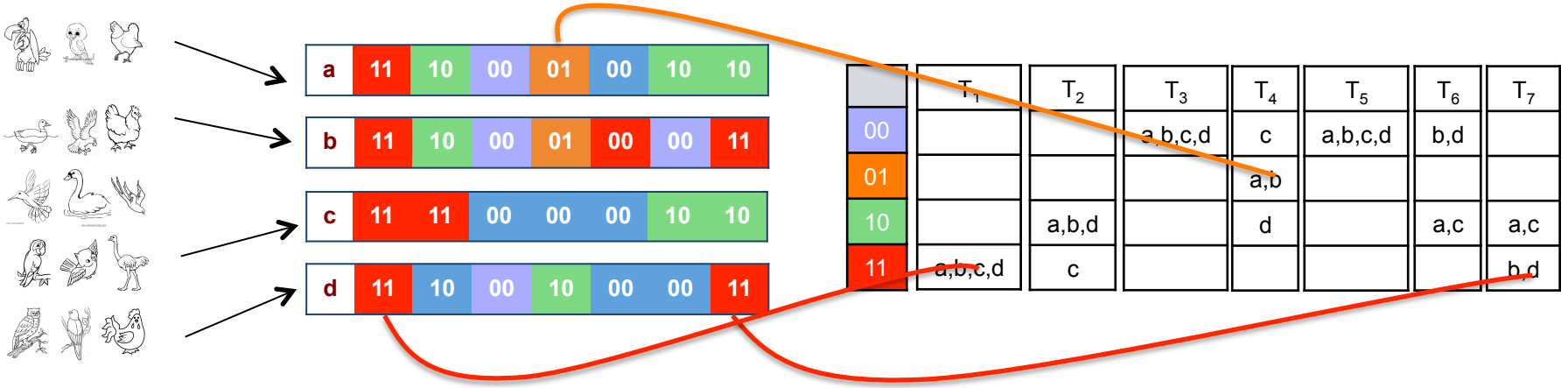
Pourcentage des k-NN exactes retrouvés en fonction du nombre d'itération



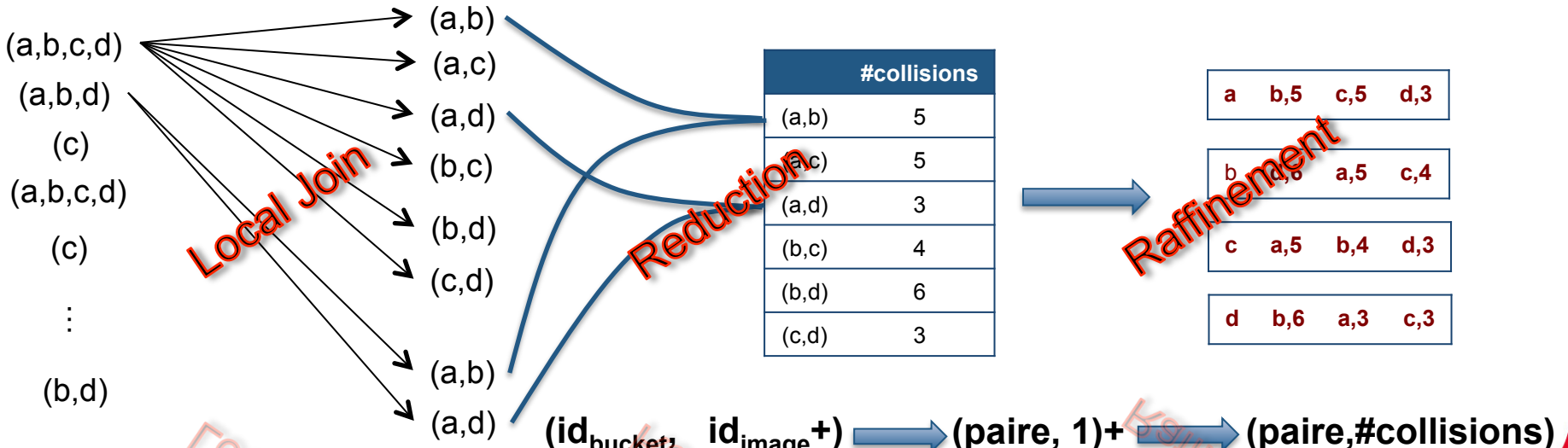
Fraction de la base parcourue en fonction du nombre d'itération



Approche par jointure de tables de hachage sensible à la localité



$(id_{Image},$  $)$ → $((id_{Table}, id_{Bucket}), id_{Image})+$ → $((id_{Table}, id_{Bucket}), id_{Image}+)$



$(id_{bucket}, id_{image}+)$ → $(paire, 1)+$ → $(paire, \#collisions)$

Introduction TP