Résumé d'article - Cours d'Apprentissage et Fouille de données
Alan Tucholka

An introductory tutorial on kd-trees
Andrew W. Moore
Carnegie Mellon University

Cet extrait de thèse est une introduction aux structures de type kd-tree, il présente également une implémentation explicite et détaillée d'un algorithme de recherche du plus proche voisin.

Introduction aux kd-trees

Un kd-tree est une structure de données en arbre binaire pour stocker un ensemble fini de points d'un espace de dimension k. Chaque nœud contient : (i) les coordonnées d'un point de l'espace (nommé dom) à k dimensions, (ii) une dimension de séparation (nommée split), (iii) une branche gauche (respectivement droite) pour les points à gauche (respectivement à droite) de l'hyperplan de séparation. Un point est la branche gauche du nœud si et seulement si sa valeur sur la dimension split est inférieure à la valeur de la même dimension du point dom.

Recherche du plus proche voisin en utilisant les kd-trees

Dans cette section, nous verrons une description de l'algorithme que propose l'auteur pour la recherche du plus proche voisin en utilisant un kd-tree.

Une première approximation consiste à trouver le point de la feuille du kd-tree qui contient la cible. S'il existe un point plus proche, il doit se trouver obligatoirement dans l'hypersphère centrée sur la cible et passant par ce point (cf. le cercle figure 1). Passons maintenant au noeud parent de la feuille. Il n'est pas nécessaire de calculer la distance entre ce noeud et tous les noeuds de la branche non explorée s'il n'y a pas d'intersection entre l'hypersphère et l'hyperrectangle non exploré (cf. rectangle gris figure 1). L'auteur présente dans l'article comment implémenter les opérations sur les hyperrectangles.

Regardons plus en détail l'algorithme présenté dans le tableau 1. L'algorithme est appelé récursivement avec 4 variables : le kd-tree, la cible, l'hyperrectangle contenant l'arbre, une distance maximum. L'appel initial prend en paramètre l'hyperrectangle englobant tous les points, et une distance maximum placée à l'infini.

- L'étape 1 gère le cas trivial ou l'arbre est vide.
- L'étape 4 coupe l'hyperrectangle actuel en deux suivant l'axe de coupe du noeud courant.
- Les étapes 5 à 7 déterminent quel hyperrectangle contient la cible.
- L'étape 8 appelle récursivement la fonction avec les paramètres correspondants à la branche de l'arbre contenant la cible.

A partir de maintenant, nous avons une approximation du point le plus proche. Tout point plus proche doit reposer dans l'hypersphère décrite dans le paragraphe précédent.

- L'étape 10 teste s'il y a une intersection entre l'hypersphère et l'hyperrectangle.

Si c'est le cas :

- L'étape 10.1 calcule la distance entre le noeud courant et la cible.
- L'étape 10.2 appelle récursivement la fonction sur la branche encore non explorée.

Performances

L'algorithme parcourt l'arbre en profondeur jusqu'à arriver à une feuille, il ne va inspecter un noeud qu'une seule fois. Le nombre de noeuds parcouru est donc compris entre log(N) et N. Cependant, N est exponentiellement dépendant de k, le nombre de dimensions.

ANNEXES

Algorithm:	Nearest Neighbour in a *k*d-tree
Input:	**kd**, of type **kdtree** **target**, of type **domain vector** **hr**, of type **hyperrectangle** **max-dist-sqd**, of type **float**
Output:	**nearest**, of type **exemplar** **dist-sqd**, of type **float**
Pre:	*Is-legal-kdtree*(**kd**)
Post:	Informally, the postcondition is that **nearest** is a nearest exemplar to **target** which also lies both within the hyperrectangle **hr** and within distance $\sqrt{\textbf{max-dist-sqd}}$ of **target**. $\sqrt{\textbf{dist-sqd}}$ is the distance of this nearest point. If there is no such point then **dist-sqd** contains infinity.
Code:	
1.	if **kd** is empty then set **dist-sqd** to infinity and exit.
2.	**s** := **split** field of **kd**
3.	**pivot** := **dom-elt** field of **kd**
4.	Cut **hr** into two sub-hyperrectangles **left-hr** and **right-hr**. The cut plane is through **pivot** and perpendicular to the **s** dimension.
5.	**target-in-left** := $\textbf{target}_\textbf{s} \leq \textbf{pivot}_\textbf{s}$
6.	if **target-in-left** then
6.1	**nearer-kd** := **left** field of **kd** and **nearer-hr** := **left-hr**
6.2	**further-kd** := **right** field of **kd** and **further-hr** := **right-hr**
7.	if **not target-in-left** then
7.1	**nearer-kd** := **right** field of **kd** and **nearer-hr** := **right-hr**
7.2	**further-kd** := **left** field of **kd** and **further-hr** := **left-hr**
8.	Recursively call **Nearest Neighbour** with parameters (**nearer-kd**,**target**, **nearer-hr**,**max-dist-sqd**), storing the results in **nearest** and **dist-sqd**
9.	**max-dist-sqd** := minimum of **max-dist-sqd** and **dist-sqd**
10.	A nearer point could only lie in **further-kd** if there were some part of **further-hr** within distance $\sqrt{\textbf{max-dist-sqd}}$ of **target**. if this is the case then
10.1	if $(\textbf{pivot} - \textbf{target})^2 < \textbf{dist-sqd}$ then
10.1.1	**nearest** := (**pivot**, **range-elt** field of **kd**)
10.1.2	**dist-sqd** := $(\textbf{pivot} - \textbf{target})^2$
10.1.3	**max-dist-sqd** := **dist-sqd**
10.2	Recursively call **Nearest Neighbour** with parameters (**further-kd**,**target**, **further-hr**,**max-dist-sqd**), storing the results in **temp-nearest** and **temp-dist-sqd**
10.3	If **temp-dist-sqd** < **dist-sqd** then
10.3.1	**nearest** := **temp-nearest** and **dist-sqd** := **temp-dist-sqd**
Proof:	Outlined in text

Tableau 1 - Algorithme de recherche du plus proche voisin.

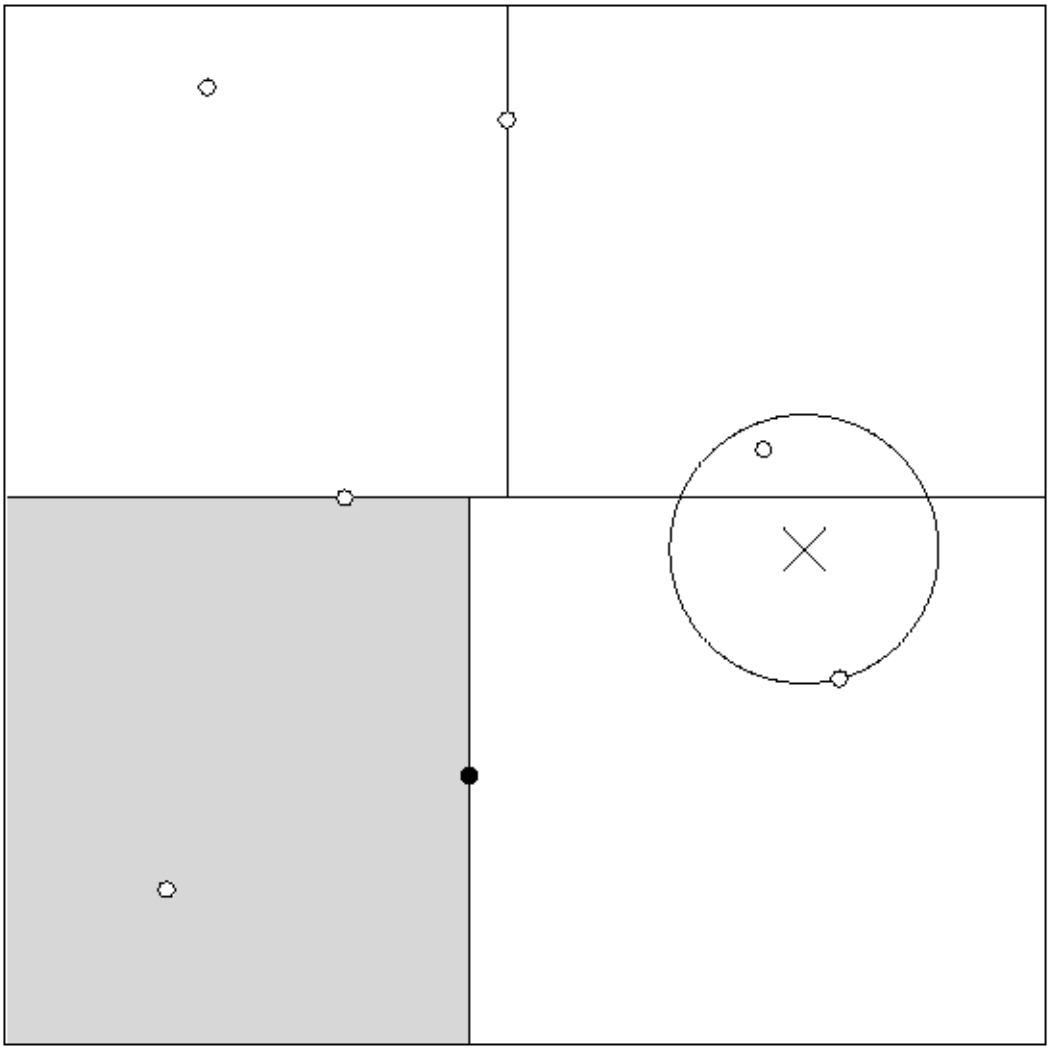

Figure 1 - Le point noir est le parent du premier point le plus proche.
Il est inutile de parcourir l'hyperrectangle en gris.