An intoductory tutorial on kd-trees

Andrew W. Moore
Carnegie Mellon University
awm@cs.cmu.edu

Extract from Andrew Moore's PhD Thesis: *Efficient Memory-based Learning for Robot Control*
PhD. Thesis; Technical Report No. 209, Computer Laboratory, University of Cambridge. 1991.

【算法】kd-tree

张楚珩
清华大学 交叉信息院博士在读

kd-tree 是一种实现多维 k-nearest neighbor（kNN）的方法。

原文传送门

Moore, Andrew W. "An intoductory tutorial on kd-trees." (1991).

特色

一种用来找多维向量最近邻的数据结构和相应算法，可用于 kNN。这篇文章除了讲了基本原理和算法，还给出了扩展性能的分析。

过程

1. 符号定义

数据集 E 包括若干样本点，每个样本点包含一个 domain vector $\in \mathbb{R}^k$ 和一个值属于值域 $\mathbb{R}^k$ 内的值。给定一个 target domain vector $\mathbf{d}$，它的它的最近邻为 $(\mathbf{d}',\mathbf{r}') \in E$，并且 $None\text{-}nearer(E,\mathbf{d},\mathbf{d}')$。

$$None\text{-}nearer(\mathbf{E},\mathbf{d},\mathbf{d}') = \forall(\mathbf{d}'',\mathbf{r}'') \in \mathbf{E} \quad |\mathbf{d}-\mathbf{d}'| \leq |\mathbf{d}-\mathbf{d}''| \qquad (6.1)$$

这里使用 L2 norm。

在此定义下，一个最简单的查找最近邻的方法就是给定一个 target domain vector 之后，遍历代表数据集的表格，然后找到最近的一个点。该方法复杂度为 $O(N)$，其中 N 表示树的规模。

2. kd-tree 数据结构

数据集可以被储存为一颗 kd-tree，这样每次查找一个最近邻的时候能够在 $O(\log N)$ 内找到。kd-tree 是一颗二叉树，其每个节点包含如下属性：

Field Name:	Field Type	Description
dom-elt	**domain-vector**	A point from k_d-d space
range-elt	**range-vector**	A point from k_r-d space
split	**integer**	The splitting dimension
left	**kd-tree**	A kd-tree representing those points to the left of the splitting plane
right	**kd-tree**	A kd-tree representing those points to the right of the splitting plane

Table 6.2: The fields of a kd-tree node

前两个属性为该节点储存的相应数据，该节点把其子节点们划分为两颗子树，其中左子树中每个节点上的 domain vector 的第 split 维的数值都小于该节点上 domain vector 的第 split 维数值，反之，则相应的数据在右子树中。

由此可以总结得到 kd-tree 和相应数据集（exemplar set）直接的对应关系：

$$exset\text{-}rep : k\text{d-tree} \rightarrow \text{exemplar-set} \tag{6.3}$$

which maps the tree to the exemplar-set it represents:

$$\begin{aligned} exset\text{-}rep(\text{empty}) &= \phi \\ exset\text{-}rep(< \mathbf{d}, \mathbf{r}, -, \text{empty}, \text{empty} >) &= \{(\mathbf{d}, \mathbf{r})\} \\ exset\text{-}rep(< \mathbf{d}, \mathbf{r}, \text{split}, \text{tree}_{left}, \text{tree}_{right} >) &= \\ exset\text{-}rep(\text{tree}_{left}) \cup \{(\mathbf{d}, \mathbf{r})\} \cup exset\text{-}rep(\text{tree}_{right}) & \end{aligned} \tag{6.4}$$

以及一颗合法 kd-tree 的判定规则

$$\begin{aligned} &Is\text{-}legal\text{-}kdtree(\text{empty}) = \text{true} \\ &Is\text{-}legal\text{-}kdtree(< \mathbf{d}, \mathbf{r}, -, \text{empty}, \text{empty} >) = \text{true} \\ &Is\text{-}legal\text{-}kdtree(< \mathbf{d}, \mathbf{r}, \text{split}, \text{tree}_{left}, \text{tree}_{right} >) = \\ &\quad \forall(\mathbf{d}', \mathbf{r}') \in exset\text{-}rep(\text{tree}_{left}) \;\; d'_{\text{split}} \leq d_{\text{split}} \wedge \\ &\quad \forall(\mathbf{d}', \mathbf{r}') \in exset\text{-}rep(\text{tree}_{right}) \;\; d'_{\text{split}} > d_{\text{split}} \wedge \\ &\quad Is\text{-}legal\text{-}kdtree(\text{tree}_{left}) \wedge \\ &\quad Is\text{-}legal\text{-}kdtree(\text{tree}_{right}) \end{aligned} \tag{6.5}$$

可以看出，kd-tree 中的每一个节点都相当于把某个 domain 空间在某个维度上“切了一刀”，把两个子空间再分配给相应的子树。kd-tree 和空间的对应关系如下图所示。可以看到，每个节点（可能是一个空节点）都代表了一个子空间，而该子空间的形状为一个超长方体。并且，叶子节点所代表的超长方体互不包含；一颗子树上的所有超长方体的并集等于该子树父节点所代表的超长方体。

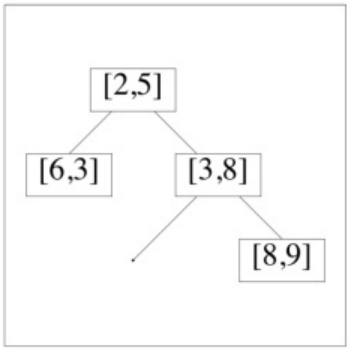

Figure 6.1

A 2d-tree of four elements. The splitting planes are not indicated. The [2,5] node splits along the y = 5 plane and the [3,8] node splits along the x = 3 plane.

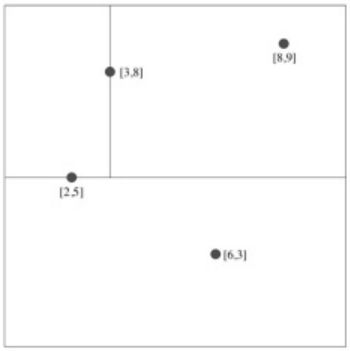

Figure 6.2

How the tree of Figure 6.1 splits up the x,y plane.

3. 构建 kd-tree 的方法

构建一颗 kd-tree 只需要递归构建就可以了，如下图所示。

Algorithm:	Constructing a *kd*-tree
Input:	**exset**, of type **exemplar-set**
Output:	**kd**, of type **kdtree**
Pre:	None
Post:	**exset** = *exset-rep*(**kd**) ∧ *Is-legal-kdtree*(**kd**)
Code:	
1.	If **exset** is empty then return the empty kdtree
2.	Call pivot-choosing procedure, which returns two values: **ex** := a member of **exset** **split** := the splitting dimension
3.	**d** := domain vector of **ex**
4.	**exset'** := **exset** with **ex** removed
5.	**r** := range vector of **ex**
6.	**exsetleft** := $\{(\mathbf{d}', \mathbf{r}') \in \mathbf{exset'} \mid d'_{\text{split}} \leq d_{\text{split}}\}$
7.	**exsetright** := $\{(\mathbf{d}', \mathbf{r}') \in \mathbf{exset'} \mid d'_{\text{split}} > d_{\text{split}}\}$
8.	**kdleft** := recursively construct *kd*-tree from **exsetleft**
9.	**kdright** := recursively construct *kd*-tree from **exsetright**
10.	**kd** := < **d**, **r**, **split**, **kdleft**, **kdright** >
Proof:	By induction on the length of **exset** and the definitions of *exset-rep* and *Is-legal-kdtree*.

Table 6.3: Constructing a *kd*-tree from a set of exemplars.

难点在于给定一颗（子）树上的数据时，如何确定使用哪个数据点的哪个维度进行分割。文章提出，分割的主要目标有两点：一是需要使得整棵树比较平衡，这样才能够保证查找到每个叶子的时候复杂度为 $O(\log N)$ ；二是需要使得整棵树划分出来的超长方体比较“方”，而非狭长状，这样能够尽可能减少找最近邻的时候查询到节点的数目（后面会看到这一点）。

那么应该找哪个维度和哪个数据点进行分割呢？有以下两种方案：

- 找方差最大的维度进行分割，找这个在维度上的数值取中位数的样本进行分割；
- 找当前树节点对应超长方体最长的那个维度进行分割，找最接近该维度上中间位置的样本进行分割；

其中前一种方案更着重优化第一个目标，后一种方案更着重优化第二个目标。

4. kd-tree 中节点的添加和删除

在 kd-tree 中添加节点比较容易，先找到一个叶子节点，使得该叶子节点所代表的超长方体包含待添加的节点。如果该叶子节点为空节点，直接把待添加节点写入即可；如果该叶子节点不空，则添加两个子节点，其中另一个子节点为空。节点划分的维度可以选择为叶子节点所代表超长方体最长的维度。

在 kd-tree 中删除节点会十分麻烦。如果待删除的节点为叶子节点，可以直接将该叶子删除，如果待删除节点为中间的节点，就基本上需要重新构建其对应的子树。文章中提到，可以暂时标记该节点为“删除”，然后在下一次重建（如果有）的时候，再删除该节点。

5. 查找最近邻

Algorithm:	Nearest Neighbour in a *kd*-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

Table 6.4: The Nearest Neighbour Algorithm

算法如图所示，注意这里该算法的目标只找到最近的一个节点。这是一个递归的算法。考虑该函数运行在 kd-tree 的某一个中间节点上。其中 1-4 为一些准备工作，5-7 行把包含待查询的 domain vector 的子树标记为 near-kd，反之则标记为 far-kd。接下来在 8 行中递归调用函数得到 near-kd 中的最近邻个节点。接下来，第 10 行判断还有没有必要去查找 far-kd 。如果 far-kd 所决定的超长方体里面最近的点都比要求查找的半径或者刚刚已经找到的最近点还远，那么就没必要去查找了。如果还有查找的必要，则先查询当前节点，然后递归查询far-kd。

值得说一下的是第 10 行如何判断 far-kd 还有没有再去查询的必要。先计算出 far-kd 所决定的超长方体中距离待查询点最近的点，计算方法如下

$$p_i = \begin{cases} hr_i^{\min} & \text{if } t_i \le hr_i^{\min} \\ t_i & \text{if } hr_i^{\min} < t_i < hr_i^{\max} \\ hr_i^{\min} & \text{if } t_i \ge hr_i^{\max} \end{cases} \tag{6.6}$$

其中， p 为求得的最近的点， t 为待查询的点， $\{[hr_i^{\min}, hr_i^{\max}]\}$ 为 far-kd 对应的超长方体。计算 p 和 t （ $\mathbf{target}$ ）之间的距离，然后与 $\mathbf{max\text{-}dist\text{-}sqd}$ 比较即可。

6. 算法拓展性能

算法在什么情况下表现比较差呢？一个直观的例子如下图所示，当待查询的点距离很多点距离都差不多的时候，就需要访问多个不同的节点，从而降低性能。

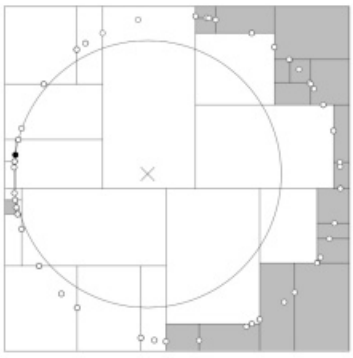

Figure 6.6

A bad distribution which forces almost all nodes to be inspected.

有比较近似的分析表明，最坏情况下，需要访问的期望节点数和数据集的大小无关。由此，每次查询的性能仍为 $O(\log N)$ 。

该文章主要做了一些实验，来说明下面一些因素对于性能的影响：

- N ：数据集大小；
- k_d ：domain vector 的维度；
- d_{distrib} ：domain vector 的分布，主要考虑 domain vector 分布的实际维度。比如，如果数据分布在一个三维空间上的一个球面上，虽然 $k_d = 3$ ，但是其数据实际分布的维度为 2；
- d_{target} ：查询向量的分布，主要考虑查询的向量是均匀地分布在整个 domain space 中的，还是和数据集中的数据分布一样。

主要结论如下：

数据集大小对于性能的影响不大，即具有较好的扩展性。

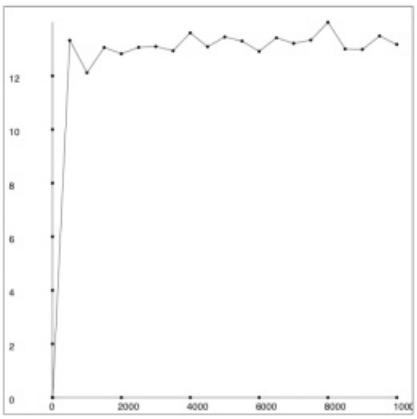

Figure 6.7

Number of inspections required during a nearest neighbour search against the size of the kd-tree. In this experiment the tree was four-dimensional and the underlying distribution of the points was three-dimensional.

对应性能的影响非常大的是数据分布的实际维度，而非 k_d

如果数据分布的实际维度增大，性能下降极快。

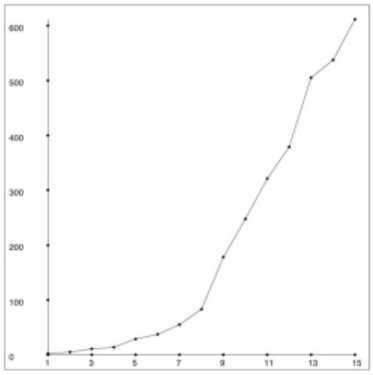

Figure 6.9

Number of inspections graphed against tree dimension. In these experiments the points had an underlying distribution with the same dimensionality as the tree.

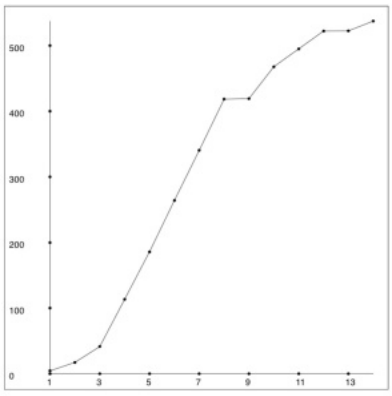

Figure 6.10

Number of inspections graphed against underlying dimensionality for a fourteen-dimensional tree.

如果数据分布的实际维度不变，而只是增加 k_d ，性能下降不大。

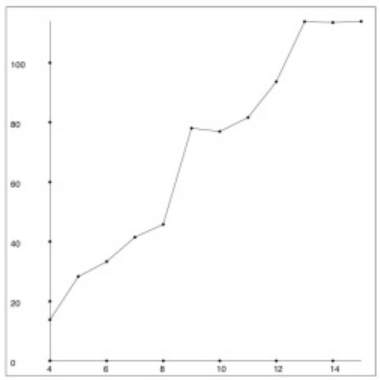

Figure 6.11

Number of inspections graphed against tree dimension, given a constant four dimensional underlying distribution.

如果查询向量的分布和数据集中数据分布不一致，会较大幅度损害性能

这一点可以参考前面那个最坏情况的例子。

7. 扩展

上述算法描述了如何找到一个给定 range 内最近的一个点。该算法可以较容易拓展到该 range 内的所有点，以及找到最近的 Q 个点。

发布于 2019-07-08

算法

文章被以下专栏收录

强化学习前沿
读呀读paper

进入专栏