# [k-d tree算法](https://www.cnblogs.com/eyeszjwang/articles/2429382.html)

k-d树（k-dimensional树的简称），是一种分割k维数据空间的数据结构。主要应用于多维空间关键数据的搜索（如：范围搜索和最近邻搜索）。

## 应用背景

SIFT算法中做特征点匹配的时候就会利用到k-d树。而特征点匹配实际上就是一个通过距离函数在高维矢量之间进行相似性检索的问题。针对如何快速而准确地找到查询点的近邻，现在提出了很多高维空间索引结构和近似查询的算法，k-d树就是其中一种。

索引结构中相似性查询有两种基本的方式：一种是范围查询（range searches），另一种是K近邻查询（K-neighbor searches）。范围查询就是给定查询点和查询距离的阈值，从数据集中找出所有与查询点距离小于阈值的数据；K近邻查询是给定查询点及正整数K，从数据集中找到距离查询点最近的K个数据，当K=1时，就是最近邻查询（nearest neighbor searches）。

特征匹配算子大致可以分为两类。一类是线性扫描法，即将数据集中的点与查询点逐一进行距离比较，也就是穷举，缺点很明显，就是没有利用数据集本身蕴含的任何结构信息，搜索效率较低，第二类是建立数据索引，然后再进行快速匹配。因为实际数据一般都会呈现出簇状的聚类形态，通过设计有效的索引结构可以大大加快检索的速度。索引树属于第二类，其基本思想就是对搜索空间进行层次划分。根据划分的空间是否有混叠可以分为Clipping和Overlapping两种。前者划分空间没有重叠，其代表就是k-d树；后者划分空间相互有交叠，其代表为R树。（这里只介绍k-d树）

## 实例

先以一个简单直观的实例来介绍k-d树算法。假设有6个二维数据点{（2,3），（5,4），（9,6），（4,7），（8,1），（7,2）}，数据点位于二维空间内（如图1中黑点所示）。k-d树算法就是要确定图1中这些分割空间的分割线（多维空间即为分割平面，一般为超平面）。下面就要通过一步步展示k-d树是如何确定这些分割线的。

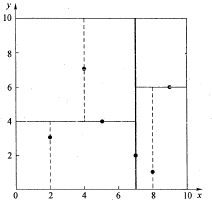


图1  二维数据k-d树空间划分示意图

k-d树算法可以分为两大部分，一部分是有关k-d树本身这种数据结构建立的算法，另一部分是在建立的k-d树上如何进行最邻近查找的算法。

## k-d树构建算法

k-d树是一个二叉树，每个节点表示一个空间范围。表1给出的是k-d树每个节点中主要包含的数据结构。

表1  k-d树中每个节点的数据类型

|  |  |  |
| --- | --- | --- |
| 域名 | 数据类型 | 描述 |
| Node-data | 数据矢量 | 数据集中某个数据点，是n维矢量（这里也就是k维） |
| Range | 空间矢量 | 该节点所代表的空间范围 |
| split | 整数 | 垂直于分割超平面的方向轴序号 |
| Left | k-d树 | 由位于该节点分割超平面左子空间内所有数据点所构成的k-d树 |
| Right | k-d树 | 由位于该节点分割超平面右子空间内所有数据点所构成的k-d树 |
| parent | k-d树 | 父节点 |

从上面对k-d树节点的数据类型的描述可以看出构建k-d树是一个逐级展开的递归过程。表2给出的是构建k-d树的伪码。

表2  构建k-d树的伪码

|  |
| --- |
| 算法：构建k-d树（createKDTree） |
| 输入：数据点集Data-set和其所在的空间Range |
| 输出：Kd，类型为k-d tree |
| 1.If Data-set为空，则返回空的k-d tree |
| 2.调用节点生成程序：  　　（1）确定split域：对于所有描述子数据（特征矢量），统计它们在每个维上的数据方差。以SURF特征为例，描述子为64维，可计算64个方差。挑选出最大值，对应的维就是split域的值。数据方差大表明沿该坐标轴方向上的数据分散得比较开，在这个方向上进行数据分割有较好的分辨率；  　　（2）确定Node-data域：数据点集Data-set按其第split域的值排序。位于正中间的那个数据点被选为Node-data。此时新的Data-set' = Data-set\Node-data（除去其中Node-data这一点）。 |
| 3.dataleft = {d属于Data-set' && d[split] ≤ Node-data[split]}     Left\_Range = {Range && dataleft}     dataright = {d属于Data-set' && d[split] > Node-data[split]}     Right\_Range = {Range && dataright} |
| 4.left = 由（dataleft，Left\_Range）建立的k-d tree，即递归调用createKDTree（dataleft，Left\_     Range）。并设置left的parent域为Kd；     right = 由（dataright，Right\_Range）建立的k-d tree，即调用createKDTree（dataleft，Left\_     Range）。并设置right的parent域为Kd。 |

　　以上述举的实例来看，过程如下：

　　由于此例简单，数据维度只有2维，所以可以简单地给x，y两个方向轴编号为0,1，也即split={0,1}。

　　（1）确定split域的首先该取的值。分别计算x，y方向上数据的方差得知x方向上的方差最大，所以split域值首先取0，也就是x轴方向；

　　（2）确定Node-data的域值。根据x轴方向的值2,5,9,4,8,7排序选出中值为7，所以Node-data = （7,2）。这样，该节点的分割超平面就是通过（7,2）并垂直于split = 0（x轴）的直线x = 7；

　　（3）确定左子空间和右子空间。分割超平面x = 7将整个空间分为两部分，如图2所示。x < =  7的部分为左子空间，包含3个节点{（2,3），（5,4），（4,7）}；另一部分为右子空间，包含2个节点{（9,6），（8,1）}。

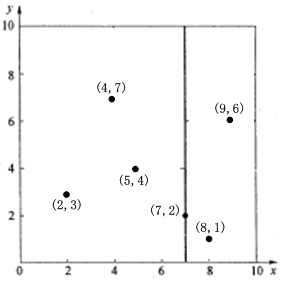


图2  x=7将整个空间分为两部分

如算法所述，k-d树的构建是一个递归的过程。然后对左子空间和右子空间内的数据重复根节点的过程就可以得到下一级子节点（5,4）和（9,6）（也就是左右子空间的'根'节点），同时将空间和数据集进一步细分。如此反复直到空间中只包含一个数据点，如图1所示。最后生成的k-d树如图3所示。

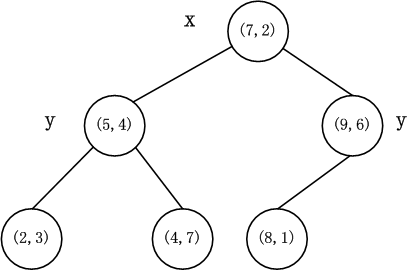


图3  上述实例生成的k-d树

　　注意：每一级节点旁边的'x'和'y'表示以该节点分割左右子空间时split所取的值。

## k-d树上的最邻近查找算法

　　在k-d树中进行数据的查找也是特征匹配的重要环节，其目的是检索在k-d树中与查询点距离最近的数据点。这里先以一个简单的实例来描述最邻近查找的基本思路。

　　星号表示要查询的点（2.1,3.1）。通过二叉搜索，顺着搜索路径很快就能找到最邻近的近似点，也就是叶子节点（2,3）。而找到的叶子节点并不一定就是最邻近的，最邻近肯定距离查询点更近，应该位于以查询点为圆心且通过叶子节点的圆域内。为了找到真正的最近邻，还需要进行'回溯'操作：算法沿搜索路径反向查找是否有距离查询点更近的数据点。此例中先从（7,2）点开始进行二叉查找，然后到达（5,4），最后到达（2,3），此时搜索路径中的节点为<（7,2），（5,4），（2,3）>，首先以（2,3）作为当前最近邻点，计算其到查询点（2.1,3.1）的距离为0.1414，然后回溯到其父节点（5,4），并判断在该父节点的其他子节点空间中是否有距离查询点更近的数据点。以（2.1,3.1）为圆心，以0.1414为半径画圆，如图4所示。发现该圆并不和超平面y = 4交割，因此不用进入（5,4）节点右子空间中去搜索。

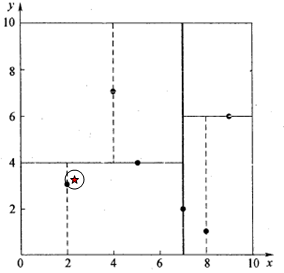


图4  查找（2.1，3.1）点的两次回溯判断

　　再回溯到（7,2），以（2.1,3.1）为圆心，以0.1414为半径的圆更不会与x = 7超平面交割，因此不用进入（7,2）右子空间进行查找。至此，搜索路径中的节点已经全部回溯完，结束整个搜索，返回最近邻点（2,3），最近距离为0.1414。

　　一个复杂点了例子如查找点为（2，4.5）。同样先进行二叉查找，先从（7,2）查找到（5,4）节点，在进行查找时是由y = 4为分割超平面的，由于查找点为y值为4.5，因此进入右子空间查找到（4,7），形成搜索路径<（7,2），（5,4），（4,7）>，取（4,7）为当前最近邻点，计算其与目标查找点的距离为3.202。然后回溯到（5,4），计算其与查找点之间的距离为3.041。以（2，4.5）为圆心，以3.041为半径作圆，如图5所示。可见该圆和y = 4超平面交割，所以需要进入（5,4）左子空间进行查找。此时需将（2,3）节点加入搜索路径中得<（7,2），（2,3）>。回溯至（2,3）叶子节点，（2,3）距离（2,4.5）比（5,4）要近，所以最近邻点更新为（2，3），最近距离更新为1.5。回溯至（7,2），以（2,4.5）为圆心1.5为半径作圆，并不和x = 7分割超平面交割，如图6所示。至此，搜索路径回溯完。返回最近邻点（2,3），最近距离1.5。k-d树查询算法的伪代码如表3所示。

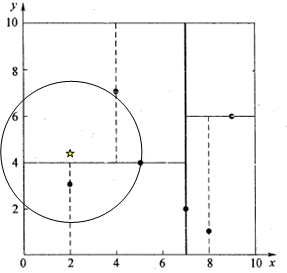


图5  查找（2，4.5）点的第一次回溯判断

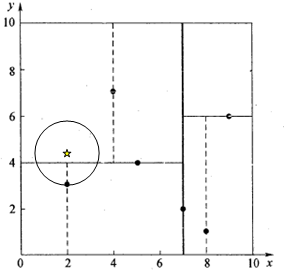


图6  查找（2，4.5）点的第二次回溯判断

表3  标准k-d树查询算法

|  |
| --- |
| 算法：k-d树最邻近查找 |
| 输入：Kd，    //k-d tree类型       target  //查询数据点 |
| 输出：nearest， //最邻近数据点       dist      //最邻近数据点和查询点间的距离 |
| 1. If Kd为NULL，则设dist为infinite并返回 |
| 2. //进行二叉查找，生成搜索路径     Kd\_point = &Kd；                   //Kd-point中保存k-d tree根节点地址     nearest = Kd\_point -> Node-data；  //初始化最近邻点     while（Kd\_point）     　　push（Kd\_point）到search\_path中； //search\_path是一个堆栈结构，存储着搜索路径节点指针   /\*\*\* If Dist（nearest，target） > Dist（Kd\_point -> Node-data，target）     　　　　nearest  = Kd\_point -> Node-data；    //更新最近邻点     　　　　Max\_dist = Dist(Kd\_point，target）；  //更新最近邻点与查询点间的距离  \*\*\*/     　　s = Kd\_point -> split；                       //确定待分割的方向     　　If target[s] <= Kd\_point -> Node-data[s]     //进行二叉查找     　　　　Kd\_point = Kd\_point -> left；     　　else     　　　　Kd\_point = Kd\_point ->right；     nearest = search\_path中最后一个叶子节点； //注意：二叉搜索时不比计算选择搜索路径中的最邻近点，这部分已被注释     Max\_dist = Dist（nearest，target）；    //直接取最后叶子节点作为回溯前的初始最近邻点 |
| 3. //回溯查找     while（search\_path != NULL）     　　back\_point = 从search\_path取出一个节点指针；   //从search\_path堆栈弹栈     　　s = back\_point -> split；                   //确定分割方向     　　If Dist（target[s]，back\_point -> Node-data[s]） < Max\_dist   //判断还需进入的子空间     　　　　If target[s] <= back\_point -> Node-data[s]     　　　　　　Kd\_point = back\_point -> right；  //如果target位于左子空间，就应进入右子空间     　　　　else     　　　　　　Kd\_point = back\_point -> left;    //如果target位于右子空间，就应进入左子空间     　　　　将Kd\_point压入search\_path堆栈；     　　If Dist（nearest，target） > Dist（Kd\_Point -> Node-data，target）     　　　　nearest  = Kd\_point -> Node-data；                 //更新最近邻点     　　　　Min\_dist = Dist（Kd\_point -> Node-data,target）；  //更新最近邻点与查询点间的距离 |

上述两次实例表明，当查询点的邻域与分割超平面两侧空间交割时，需要查找另一侧子空间，导致检索过程复杂，效率下降。研究表明N个节点的K维k-d树搜索过程时间复杂度为：tworst=O（kN1-1/k）。

## 后记

以上为了介绍方便，讨论的是二维情形。像实际的应用中，如SIFT特征矢量128维，SURF特征矢量64维，维度都比较大，直接利用k-d树快速检索（维数不超过20）的性能急剧下降。假设数据集的维数为D，一般来说要求数据的规模N满足N»2D，才能达到高效的搜索。所以这就引出了一系列对k-d树算法的改进。有待进一步研究学习。