k-d tree 算法

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。

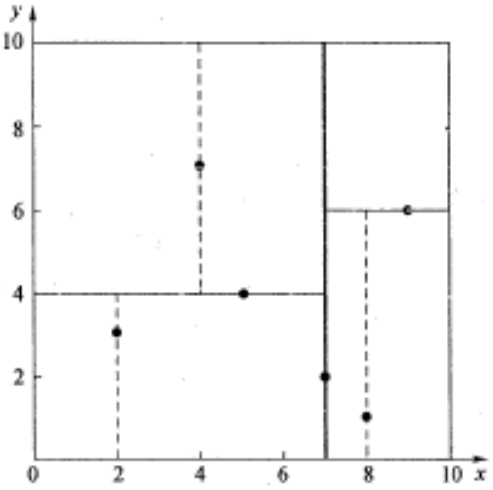

图 1　二维数据 k-d 树空间划分示意图

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）}。

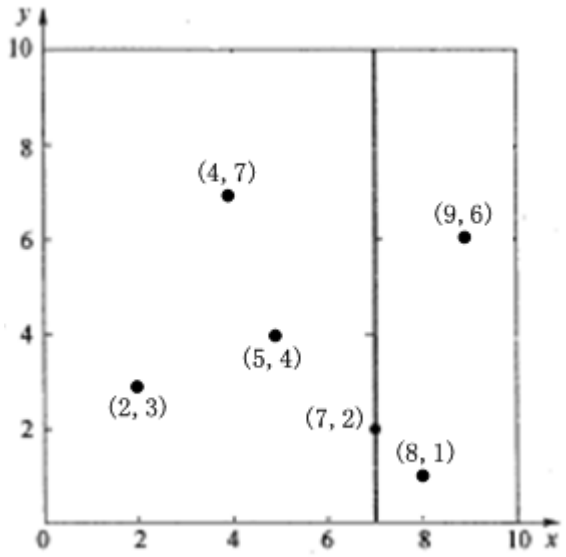

图 2　x=7 将整个空间分为两部分

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。

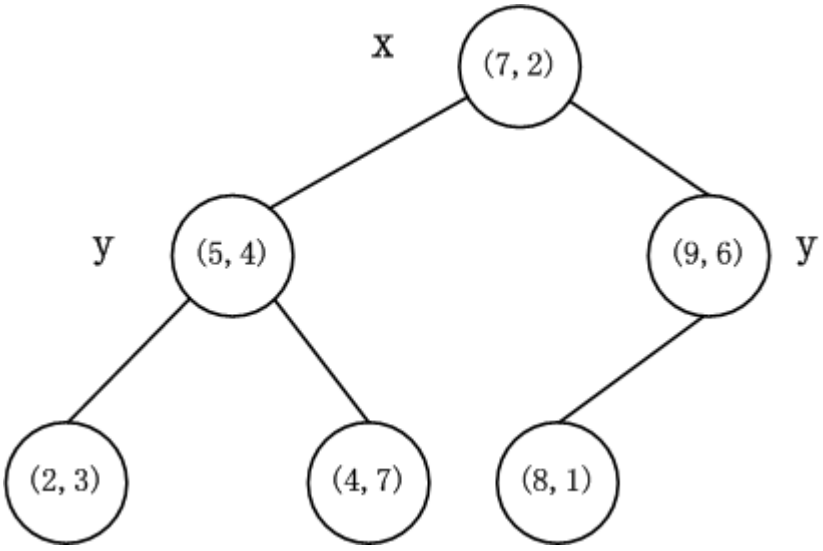

图 3　上述实例生成的 k-d 树

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。

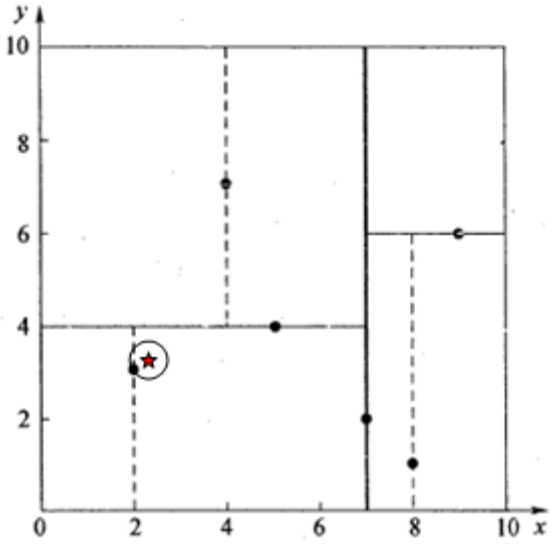

图 4　查找（2.1，3.1）点的两次回溯判断

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。

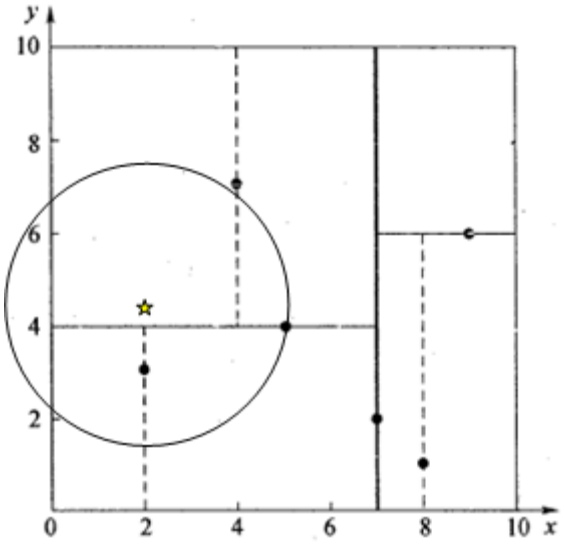

图 5　查找（2，4.5）点的第一次回溯判断

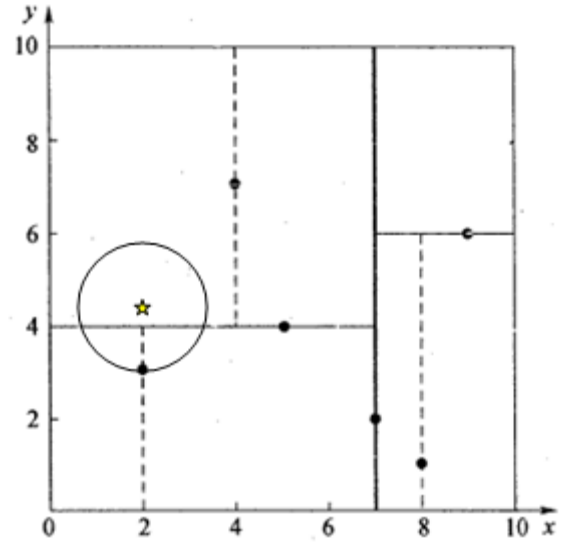

图 6　查找（2，4.5）点的第二次回溯判断

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：$t_{worst}=O（kN^{1-1/k}）$。

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 $N \gg 2^D$，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。