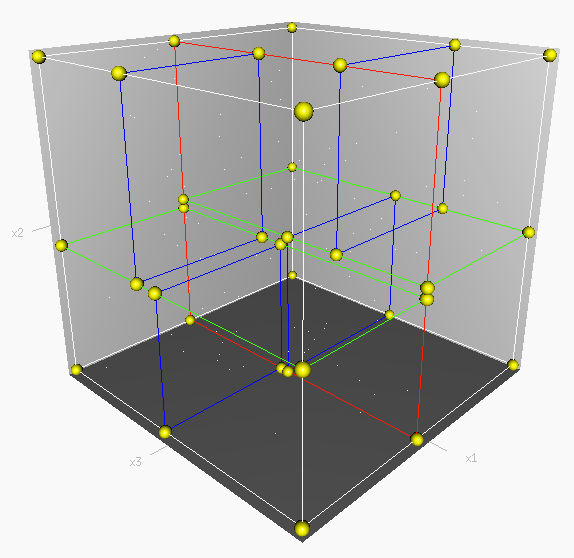
一、如何高效率地实现k近邻法？

  在SIFT图像特征匹配等应用中，需要在高维特征空间中快速找到距离目标图像特征最近邻的那个特征点，往往需要进行比较的特征向量的数量很大，如果进行朴 素最近邻搜索，也就是依次计算目标点和每一个待匹配特征的距离，然后再算出最短距离这样的策略，那么特征匹配算法的时间复杂度将会高得令人难以接受。因 此，我们需要借助一种存储和表示k维数据的数据结构，既能够方便地存储k维数据，又能够进行高效率的搜索。

二、k-d树的基本思想

  k-d树由斯坦福大学本科生Jon Louis Bentley于1975年首次提出。k-d树是每个节点都为k维点的二叉树。其中k表示存储的数据的维度，d就是dimension的意思。所有非叶子节点可 以视作用一个超平面把空间分割成两部分。在超平面左边的点代表节点的左子树，在超平面右边的点代表节点的右子树。超平面的方向可以用下述方法来选择：每个 节点都与k维中垂直于超平面的那一维有关。因此，如果选择按照x轴划分，所有x值小于指定值的节点都会出现在左子树，所有x值大于指定值的节点都会出现在 右子树。这样，超平面可以用该x值来确定，其法矢为x轴的单位向量。一个三维空间内的3-d树如下所示：



  当特征空间维度大于20时，k-d tree算法的性能会剧烈下降，对于高维数据，David Lowe在1997的一篇文章中提出一种近似算法best-bins-first，可以有效改善这种情况。

kd 树是一种对k维空间中的实例点进行存储以便对其进行快速检索的树形结构。kd树从本质上来说是二叉树，表示对k维空间的一个划分。构造kd树相当于不断地 用垂直于坐标轴的超平面切分k维空间，构成一系列的k维超矩形区域，kd树的每一个结点都对应于一个超矩形区域，非叶结点的左右子树分别表示划分得到的两 个区域。在2维情形，当划分超平面平行于x轴时，在划分超平面以下的数据点将存储在此划分结点的左子树，在超平面以上的点存储在此划分结点右子树；若划分 超平面平行于y轴，在划分超平面左侧的数据点将存储在此划分结点的左子树，在超平面右侧的点存储在此划分结点右子树。

构造kd树的方法：首先构造根节点，根节点对应于整个k维空间，包含所有的实例点，（至于如何选取划分点，有不同的策略。最 常用的是一种方法是：对于所有的样本点，统计它们在每个维上的方差，挑选出方差中的最大值，对应的维就是要进行数据切分的维度。数据方差最大表明沿该维度 数据点分散得比较开，这个方向上进行数据分割可以获得最好的分辨率；然后再将所有样本点按切分维度的值进行排序，位于正中间的那个数据点选为分裂结 点。）。然后利用递归的方法，分别构造k-d树根节点的左右子树。在超矩形区域上选择一个坐标轴（切分维度）和一个分裂结点，以通过此分裂结点且垂直于切 分方向坐标轴的直线作为分隔线，将当前超矩形区域分隔成左右或者上下两个子超矩形区域，对应于分裂结点的左右子树的根节点。实例也就被分到两个不相交的区 域中。重复此过程直到子区域内没有实例点时终止。终止时的结点为叶结点。

通常依次选择坐标轴对空间切分，选择训练实例点在选定坐标轴上的中位数为切分点，这样得到的kd树是平衡的，但并不一定能保证检索的效率最优。

拥有n个已知点的kD-Tree，其复杂度如下：

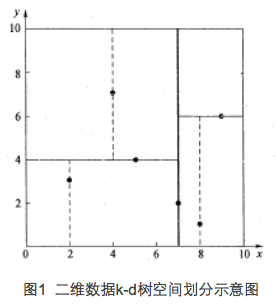
1. 构建：O(log2n)
2. 插入：O(log n)
3. 删除：O(log n)
4. 查询：O(n1-1/k+m) m---每次要搜索的最近点个数

三、k-d tree的实现

  k-d tree是英文K-dimension tree的缩写，是对数据点在k维空间中划分的一种数据结构。k-d tree实际上是一种二叉树。每个结点的内容如下:

|  |  |  |
| --- | --- | --- |
| 域名 | 类型 | 描述 |
| dom\_elt | kd维的向量 | kd维空间中的一个样本点 |
| split | 整数 | 分裂维的序号，也是垂直于分割超面的方向轴序号 |
| left | kd-tree | 由位于该结点分割超面左子空间内所有数据点构成的kd-tree |
| right | kd-tree | 由位于该结点分割超面右子空间内所有数据点构成的kd-tree |

  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。  
  
  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。

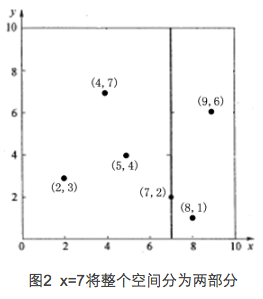


由于此例简单，数据维度只有2维，所以可以简单地给x，y两个方向轴编号为0,1，也即split={0,1}。

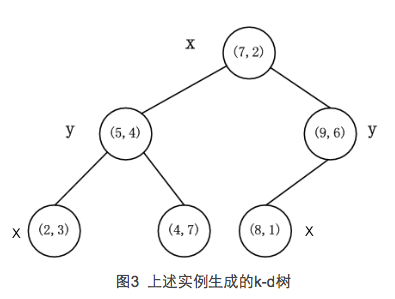
　　（1）确定split域的首先该取的值。分别计算x，y方向上数据的方差得知x方向上的方差最大，所以split域值首先取0，也就是x轴方向；

　　（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）}。



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。



从上面的表也可以看出k-d tree本质上是一种二叉树，因此k-d tree的构建是一个逐级展开的递归过程。

**[cpp]** [view plaincopy](http://blog.csdn.net/zhl30041839/article/details/9277807)

1. 算法：createKDTree 构建一棵k-d tree
3. 输入：exm\_set 样本集
5. 输出 : Kd, 类型为kd-tree
7. 1. 如果exm\_set是空的，则返回空的kd-tree
9. 2.调用分裂结点选择程序（输入是exm\_set），返回两个值
11. dom\_elt:= exm\_set中的一个样本点
13. split := 分裂维的序号
15. 3.exm\_set\_left = {exm∈exm\_set – dom\_elt && exm[split] <= dom\_elt[split]}
17. exm\_set\_right = {exm∈exm\_set – dom\_elt && exm[split] > dom\_elt[split]}
19. 4.left = createKDTree(exm\_set\_left)
21. right = createKDTree(exm\_set\_right)

k-d tree最近邻搜索算法

  如前所述，在k-d tree树中进行数据的k近邻搜索是特征匹配的重要环节，其目的是检索在k-d tree中与待查询点距离最近的k个数据点。

  最近邻搜索是k近邻的特例，也就是1近邻。将1近邻改扩展到k近邻非常容易。下面介绍最简单的k-d tree最近邻搜索算法。

  基本的思路很简单：首先通过二叉树搜索（比较待查询节点和分裂节点的分裂维的值，小于等于就进入左子树分支，等于就进入右子树分支直到叶子结点），顺着 “搜索路径”很快能找到最近邻的近似点，也就是与待查询点处于同一个子空间的叶子结点；然后再回溯搜索路径，并判断搜索路径上的结点的其他子结点空间中是 否可能有距离查询点更近的数据点，如果有可能，则需要跳到其他子结点空间中去搜索（将其他子结点加入到搜索路径）。重复这个过程直到搜索路径为空。下面给 出k-d tree最近邻搜索的伪代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/zhl30041839/article/details/9277807)

1. 算法：kdtreeFindNearest /\* k-d tree的最近邻搜索 \*/
3. 输入：Kd /\* k-d tree类型\*/
5. target /\* 待查询数据点 \*/
7. 输出 : nearest /\* 最近邻数据结点 \*/
9. dist /\* 最近邻和查询点的距离 \*/
11. 1. 如果Kd是空的，则设dist为无穷大返回
13. 2. 向下搜索直到叶子结点
15. pSearch = &Kd
17. while(pSearch != NULL)
18. {
19. pSearch加入到search\_path中;
20. if(target[pSearch->split] <= pSearch->dom\_elt[pSearch->split]) /\* 如果小于就进入左子树 \*/
21. {
22. pSearch = pSearch->left;
23. }
24. else
25. {
26. pSearch = pSearch->right;
27. }
28. }
29. 取出search\_path最后一个赋给nearest
31. dist = Distance(nearest, target);
32. 3. 回溯搜索路径
34. while(search\_path不为空)
35. {
36. 取出search\_path最后一个结点赋给pBack
38. if(pBack->left为空 && pBack->right为空) /\* 如果pBack为叶子结点 \*/
40. {
42. if( Distance(nearest, target) > Distance(pBack->dom\_elt, target) )
43. {
44. nearest = pBack->dom\_elt;
45. dist = Distance(pBack->dom\_elt, target);
46. }
48. }
50. else
52. {
54. s = pBack->split;
55. if( abs(pBack->dom\_elt[s] - target[s]) < dist) /\* 如果以target为中心的圆（球或超球），半径为dist的圆与分割超平面相交， 那么就要跳到另一边的子空间去搜索 \*/
56. {
57. if( Distance(nearest, target) > Distance(pBack->dom\_elt, target) )
58. {
59. nearest = pBack->dom\_elt;
60. dist = Distance(pBack->dom\_elt, target);
61. }
62. if(target[s] <= pBack->dom\_elt[s]) /\* 如果target位于pBack的左子空间，那么就要跳到右子空间去搜索 \*/
63. pSearch = pBack->right;
64. else
65. pSearch = pBack->left; /\* 如果target位于pBack的右子空间，那么就要跳到左子空间去搜索 \*/
66. if(pSearch != NULL)
67. pSearch加入到search\_path中
68. }
70. }
71. }

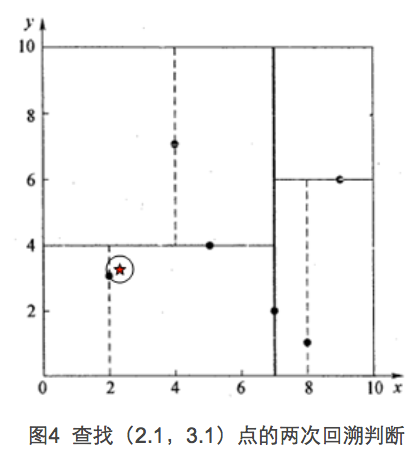
假设我们的k-d tree就是上面通过样本集{(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)}创建的。

我们来查找点 (2.1,3.1)，在(7,2)点测试到达(5,4)，在(5,4)点测试到达(2,3)，然后search\_path中的结点为<(7,2), (5,4), (2,3)>，从search\_path中取出(2,3)作为当前最佳结点nearest, dist为0.141；

然后回溯至(5,4)，以(2.1,3.1)为圆心，以dist=0.141为半径画一个圆，并不和超平面y=4相交，如下图，所以不必跳到结点(5,4)的右子空间去搜索，因为右子空间中不可能有更近样本点了。

于是在回溯至(7,2)，同理，以(2.1,3.1)为圆心，以dist=0.141为半径画一个圆并不和超平面x=7相交，所以也不用跳到结点(7,2)的右子空间去搜索。

至此，search\_path为空，结束整个搜索，返回nearest(2,3)作为(2.1,3.1)的最近邻点，最近距离为0.141。



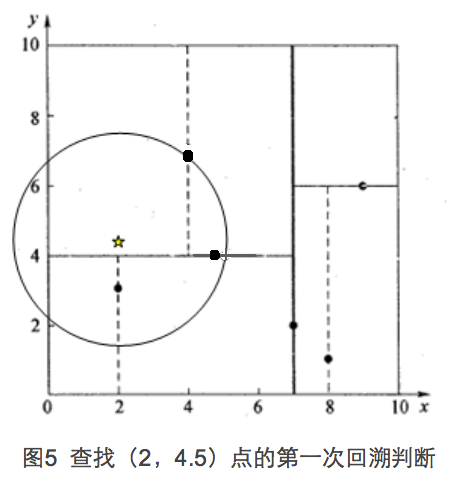
再举一个稍微复杂的例子，我们 来查找点(2,4.5)，在(7,2)处测试到达(5,4)，在(5,4)处测试到达(4,7)，然后search\_path中的结点为< (7,2), (5,4), (4,7)>，从search\_path中取出(4,7)作为当前最佳结点nearest, dist为3.202；

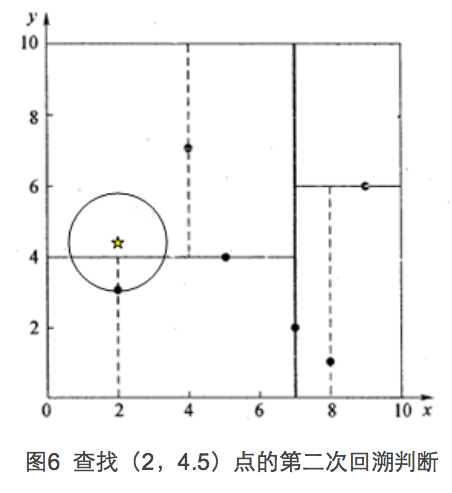
然后回溯至(5,4)，以 (2,4.5)为圆心，以dist=3.202为半径画一个圆与超平面y=4相交，如下图，所以需要跳到(5,4)的左子空间去搜索。所以要将(2,3) 加入到search\_path中，现在search\_path中的结点为<(7,2), (2, 3)>；另外，(5,4)与(2,4.5)的距离为3.04 < dist = 3.202，所以将(5,4)赋给nearest，并且dist=3.04。

回溯至(2,3)，(2,3)是叶子节点，直接平判断(2,3)是否离(2,4.5)更近，计算得到距离为1.5，所以nearest更新为(2,3)，dist更新为(1.5)

回溯至(7,2)，同理，以(2,4.5)为圆心，以dist=1.5为半径画一个圆并不和超平面x=7相交, 所以不用跳到结点(7,2)的右子空间去搜索。

至此，search\_path为空，结束整个搜索，返回nearest(2,3)作为(2,4.5)的最近邻点，最近距离为1.5。





  以下是k-d树的c++代码实现，包括建树过程和搜索过程。算法main函数输入k-d树训练实例点，算法会完成建树操作，随后可以输入待查询的目标点， 程序将会搜索K-d树找出与输入目标点最近邻的训练实例点。本程序只实现了1近邻搜索，如果要实现k近邻搜索，只需对程序稍作修改。比如可以对每个结点添 加一个标记，如果已经输出该结点为最近邻结点，那么就继续查找次近邻的结点，直到输出k个结点后算法结束。

**[cpp]** [view plaincopy](http://blog.csdn.net/zhl30041839/article/details/9277807)

1. #include <iostream>
2. #include <algorithm>
3. #include <stack>
4. #include <math.h>
5. using namespace std;
6. /\*function of this program: build a 2d tree using the input training data
7. the input is exm\_set which contains a list of tuples (x,y)
8. the output is a 2d tree pointer\*/

11. struct data
12. {
13. double x = 0;
14. double y = 0;
15. };
17. struct Tnode
18. {
19. struct data dom\_elt;
20. int split;
21. struct Tnode \* left;
22. struct Tnode \* right;
23. };
25. bool cmp1(data a, data b){
26. return a.x < b.x;
27. }
29. bool cmp2(data a, data b){
30. return a.y < b.y;
31. }
33. bool equal(data a, data b){
34. if (a.x == b.x && a.y == b.y)
35. {
36. return true;
37. }
38. else{
39. return false;
40. }
41. }
43. void ChooseSplit(data exm\_set[], int size, int &split, data &SplitChoice){
44. /\*compute the variance on every dimension. Set split as the dismension that have the biggest
45. variance. Then choose the instance which is the median on this split dimension.\*/
46. /\*compute variance on the x,y dimension. DX=EX^2-(EX)^2\*/
47. double tmp1,tmp2;
48. tmp1 = tmp2 = 0;
49. for (int i = 0; i < size; ++i)
50. {
51. tmp1 += 1.0 / (double)size \* exm\_set[i].x \* exm\_set[i].x;
52. tmp2 += 1.0 / (double)size \* exm\_set[i].x;
53. }
54. double v1 = tmp1 - tmp2 \* tmp2;  //compute variance on the x dimension
56. tmp1 = tmp2 = 0;
57. for (int i = 0; i < size; ++i)
58. {
59. tmp1 += 1.0 / (double)size \* exm\_set[i].y \* exm\_set[i].y;
60. tmp2 += 1.0 / (double)size \* exm\_set[i].y;
61. }
62. double v2 = tmp1 - tmp2 \* tmp2;  //compute variance on the y dimension
64. split = v1 > v2 ? 0:1; //set the split dimension
66. if (split == 0)
67. {
68. sort(exm\_set,exm\_set + size, cmp1);
69. }
70. else{
71. sort(exm\_set,exm\_set + size, cmp2);
72. }
74. //set the split point value
75. SplitChoice.x = exm\_set[size / 2].x;
76. SplitChoice.y = exm\_set[size / 2].y;
78. }
80. Tnode\* build\_kdtree(data exm\_set[], int size, Tnode\* T){
81. //call function ChooseSplit to choose the split dimension and split point
82. if (size == 0){
83. return NULL;
84. }
85. else{
86. int split;
87. data dom\_elt;
88. ChooseSplit(exm\_set, size, split, dom\_elt);
89. data exm\_set\_right [100];
90. data exm\_set\_left [100];
91. int sizeleft ,sizeright;
92. sizeleft = sizeright = 0;
94. if (split == 0)
95. {
96. for (int i = 0; i < size; ++i)
97. {
99. if (!equal(exm\_set[i],dom\_elt) && exm\_set[i].x <= dom\_elt.x)
100. {
101. exm\_set\_left[sizeleft].x = exm\_set[i].x;
102. exm\_set\_left[sizeleft].y = exm\_set[i].y;
103. sizeleft++;
104. }
105. else if (!equal(exm\_set[i],dom\_elt) && exm\_set[i].x > dom\_elt.x)
106. {
107. exm\_set\_right[sizeright].x = exm\_set[i].x;
108. exm\_set\_right[sizeright].y = exm\_set[i].y;
109. sizeright++;
110. }
111. }
112. }
113. else{
114. for (int i = 0; i < size; ++i)
115. {
117. if (!equal(exm\_set[i],dom\_elt) && exm\_set[i].y <= dom\_elt.y)
118. {
119. exm\_set\_left[sizeleft].x = exm\_set[i].x;
120. exm\_set\_left[sizeleft].y = exm\_set[i].y;
121. sizeleft++;
122. }
123. else if (!equal(exm\_set[i],dom\_elt) && exm\_set[i].y > dom\_elt.y)
124. {
125. exm\_set\_right[sizeright].x = exm\_set[i].x;
126. exm\_set\_right[sizeright].y = exm\_set[i].y;
127. sizeright++;
128. }
129. }
130. }
131. T = new Tnode;
132. T->dom\_elt.x = dom\_elt.x;
133. T->dom\_elt.y = dom\_elt.y;
134. T->split = split;
135. T->left = build\_kdtree(exm\_set\_left, sizeleft, T->left);
136. T->right = build\_kdtree(exm\_set\_right, sizeright, T->right);
137. return T;
139. }
140. }

143. double Distance(data a, data b){
144. double tmp = (a.x - b.x) \* (a.x - b.x) + (a.y - b.y) \* (a.y - b.y);
145. return sqrt(tmp);
146. }

149. void searchNearest(Tnode \* Kd, data target, data &nearestpoint, double & distance){
151. //1. 如果Kd是空的，则设dist为无穷大返回
153. //2. 向下搜索直到叶子结点
155. stack<Tnode\*> search\_path;
156. Tnode\* pSearch = Kd;
157. data nearest;
158. double dist;
160. while(pSearch != NULL)
161. {
162. //pSearch加入到search\_path中;
163. search\_path.push(pSearch);
165. if (pSearch->split == 0)
166. {
167. if(target.x <= pSearch->dom\_elt.x) /\* 如果小于就进入左子树 \*/
168. {
169. pSearch = pSearch->left;
170. }
171. else
172. {
173. pSearch = pSearch->right;
174. }
175. }
176. else{
177. if(target.y <= pSearch->dom\_elt.y) /\* 如果小于就进入左子树 \*/
178. {
179. pSearch = pSearch->left;
180. }
181. else
182. {
183. pSearch = pSearch->right;
184. }
185. }
186. }
187. //取出search\_path最后一个赋给nearest
188. nearest.x = search\_path.top()->dom\_elt.x;
189. nearest.y = search\_path.top()->dom\_elt.y;
190. search\_path.pop();

193. dist = Distance(nearest, target);
194. //3. 回溯搜索路径
196. Tnode\* pBack;
198. while(search\_path.size() != 0)
199. {
200. //取出search\_path最后一个结点赋给pBack
201. pBack = search\_path.top();
202. search\_path.pop();
204. if(pBack->left == NULL && pBack->right == NULL) /\* 如果pBack为叶子结点 \*/
206. {
208. if( Distance(nearest, target) > Distance(pBack->dom\_elt, target) )
209. {
210. nearest = pBack->dom\_elt;
211. dist = Distance(pBack->dom\_elt, target);
212. }
214. }
216. else
218. {
220. int s = pBack->split;
221. if (s == 0)
222. {
223. if( fabs(pBack->dom\_elt.x - target.x) < dist) /\* 如果以target为中心的圆（球或超球），半径为dist的圆与分割超平面相交， 那么就要跳到另一边的子空间去搜索 \*/
224. {
225. if( Distance(nearest, target) > Distance(pBack->dom\_elt, target) )
226. {
227. nearest = pBack->dom\_elt;
228. dist = Distance(pBack->dom\_elt, target);
229. }
230. if(target.x <= pBack->dom\_elt.x) /\* 如果target位于pBack的左子空间，那么就要跳到右子空间去搜索 \*/
231. pSearch = pBack->right;
232. else
233. pSearch = pBack->left; /\* 如果target位于pBack的右子空间，那么就要跳到左子空间去搜索 \*/
234. if(pSearch != NULL)
235. //pSearch加入到search\_path中
236. search\_path.push(pSearch);
237. }
238. }
239. else {
240. if( fabs(pBack->dom\_elt.y - target.y) < dist) /\* 如果以target为中心的圆（球或超球），半径为dist的圆与分割超平面相交， 那么就要跳到另一边的子空间去搜索 \*/
241. {
242. if( Distance(nearest, target) > Distance(pBack->dom\_elt, target) )
243. {
244. nearest = pBack->dom\_elt;
245. dist = Distance(pBack->dom\_elt, target);
246. }
247. if(target.y <= pBack->dom\_elt.y) /\* 如果target位于pBack的左子空间，那么就要跳到右子空间去搜索 \*/
248. pSearch = pBack->right;
249. else
250. pSearch = pBack->left; /\* 如果target位于pBack的右子空间，那么就要跳到左子空间去搜索 \*/
251. if(pSearch != NULL)
252. // pSearch加入到search\_path中
253. search\_path.push(pSearch);
254. }
255. }
257. }
258. }
260. nearestpoint.x = nearest.x;
261. nearestpoint.y = nearest.y;
262. distance = dist;
264. }
266. int main(){
267. data exm\_set[100]; //assume the max training set size is 100
268. double x,y;
269. int id = 0;
270. cout<<"Please input the training data in the form x y. One instance per line. Enter -1 -1 to stop."<<endl;
271. while (cin>>x>>y){
272. if (x == -1)
273. {
274. break;
275. }
276. else{
277. exm\_set[id].x = x;
278. exm\_set[id].y = y;
279. id++;
280. }
281. }
282. struct Tnode \* root = NULL;
283. root = build\_kdtree(exm\_set, id, root);
285. data nearestpoint;
286. double distance;
287. data target;
288. cout <<"Enter search point"<<endl;
289. while (cin>>target.x>>target.y)
290. {
291. searchNearest(root, target, nearestpoint, distance);
292. cout<<"The nearest distance is "<<distance<<",and the nearest point is "<<nearestpoint.x<<","<<nearestpoint.y<<endl;
293. cout <<"Enter search point"<<endl;
295. }
296. }

来源： <<http://blog.csdn.net/zhl30041839/article/details/9277807>>

[**k-d tree的优化查找算法BBF**](http://www.cnblogs.com/eyeszjwang/articles/2437706.html)

　　BBF（Best Bin First）是一种改进的k-d树最近邻查询算法。从前两篇标准的k-d树查询过程可以看出其搜索过程中的“回溯”是由“查询路径”来决定的，并没有考虑 查询路径上数据点本身的一些性质。BBF的查询思路就是将“查询路径”上的节点进行排序，如按各自分割超平面（称为Bin）与查询点的距离排序。回溯检查 总是从优先级最高的（Best Bin）的树节点开始。另外BBF还设置了一个运行超时限制，当优先级队列中的所有节点都经过检查或者超出时间限制时，算法返回当前找到的最好结果作为近 似的最近邻。采用了best-bin-first search方法就可以将k-d树扩展到高维数据集上。

　　下面我们通过大牛Rob Hess基于OpenCV的SIFT实现中的相关代码来具体学习下BBF算法。

/\*  
Finds an image feature's approximate k nearest neighbors in a kd tree using  
Best Bin First search.  
  
@param kd\_root root of an image feature kd tree  
@param feat image feature for whose neighbors to search  
@param k number of neighbors to find  
@param nbrs pointer to an array in which to store pointers to neighbors  
    in order of increasing descriptor distance  
@param max\_nn\_chks search is cut off after examining this many tree entries  
  
@return Returns the number of neighbors found and stored in nbrs, or  
    -1 on error.  
\*/  
//参数和返回值参看以上注释  
//基于k-d tree + bbf的k近邻查找函数  
int kdtree\_bbf\_knn( struct kd\_node\* kd\_root, struct feature\* feat, int k,  
struct feature\*\*\* nbrs, int max\_nn\_chks )  
{  
struct kd\_node\* expl;　　　　　　//expl是特征k-d tree中的一个节点  
struct min\_pq\* min\_pq;         //min\_pq是优先级队列  
struct feature\* tree\_feat, \*\* \_nbrs;//tree\_feat是一个SIFT特征，\_nbrs中存放着查找出来的近邻特征节点  
struct bbf\_data\* bbf\_data;　　　//bbf\_data是一个用来存放临时特征数据和特征间距离的缓存结构  
int i, t = 0, n = 0;　　　　　　 //t是运行时限，n是查找出来的近邻个数  
if( ! nbrs  ||  ! feat  ||  ! kd\_root )  
    {  
        fprintf( stderr, "Warning: NULL pointer error, %s, line %d\n",  
                \_\_FILE\_\_, \_\_LINE\_\_ );  
return -1;  
    }  
  
    \_nbrs = calloc( k, sizeof( struct feature\* ) );　　//给查找结果分配相应大小的内存  
    min\_pq = minpq\_init();　　　　　　　　　　　　　　　　　//min\_pq队列初始化  
    minpq\_insert( min\_pq, kd\_root, 0 );　　　　　　　　　//将根节点先插入到min\_pq优先级队列中  
while( min\_pq->n > 0  &&  t < max\_nn\_chks )       //min\_pq队列没有回溯完且未达到时限  
    {  
        expl = (struct kd\_node\*)minpq\_extract\_min( min\_pq );//从min\_pq中提取优先级最高的节点（并移除）  
if( ! expl )  
        {  
            fprintf( stderr, "Warning: PQ unexpectedly empty, %s line %d\n",  
                    \_\_FILE\_\_, \_\_LINE\_\_ );  
goto fail;  
        }  
  
        expl = explore\_to\_leaf( expl, feat, min\_pq );　　　　//从expl节点开始查找到叶子节点（下详）　  
if( ! expl )  
        {  
            fprintf( stderr, "Warning: PQ unexpectedly empty, %s line %d\n",  
                    \_\_FILE\_\_, \_\_LINE\_\_ );  
goto fail;  
        }  
  
for( i = 0; i < expl->n; i++ )　　//开始比较查找最近邻  
        {  
            tree\_feat = &expl->features[i];  
            bbf\_data = malloc( sizeof( struct bbf\_data ) );  
if( ! bbf\_data )  
            {  
                fprintf( stderr, "Warning: unable to allocate memory,"  
" %s line %d\n", \_\_FILE\_\_, \_\_LINE\_\_ );  
goto fail;  
            }  
            bbf\_data->old\_data = tree\_feat->feature\_data;  
            bbf\_data->d = descr\_dist\_sq(feat, tree\_feat);　　//计算叶子节点特征和目标特征的距离  
            tree\_feat->feature\_data = bbf\_data;  
            n += insert\_into\_nbr\_array( tree\_feat, \_nbrs, n, k );//判断并插入符合条件的近邻到\_nbrs中  
        }  
        t++;  
    }  
　　 //释放内存并返回结果  
    minpq\_release( &min\_pq );  
for( i = 0; i < n; i++ )  
    {  
        bbf\_data = \_nbrs[i]->feature\_data;  
        \_nbrs[i]->feature\_data = bbf\_data->old\_data;  
        free( bbf\_data );  
    }  
    \*nbrs = \_nbrs;  
return n;  
  
fail:  
    minpq\_release( &min\_pq );  
for( i = 0; i < n; i++ )  
    {  
        bbf\_data = \_nbrs[i]->feature\_data;  
        \_nbrs[i]->feature\_data = bbf\_data->old\_data;  
        free( bbf\_data );  
    }  
    free( \_nbrs );  
    \*nbrs = NULL;  
return -1;  
}

 　 　整个kdtree\_bbf\_knn函数包括了优先级队列的建立和k邻近查找两个过程。其中最关键的两个数据结构就是min\_pq优先级队列和\_nbrs 存放k邻近结果的队列。min\_pq优先级队列是按照各节点的分割超平面和目标查询特征点之间的距离升序排列的，第一个节点就是最小距离（优先级最高）的 节点。另外\_nbrs中也是按照与目标特征的距离升序排列，直接取结果的前k个特征就是对应的k近邻。注意：上述代码中的一些数据结构的定义以及一些对应 的函数如：minpq\_insert，minpq\_extract\_min， insert\_into\_nbr\_array， descr\_dist\_sq等在这里就不贴了，详细代码可参看Rob Hess主页（<http://blogs.oregonstate.edu/hess/>）中代码参考文档。

　　下面来详细看看函数explore\_to\_leaf是如何实现的。

/\*  
Explores a kd tree from a given node to a leaf.  Branching decisions are  
made at each node based on the descriptor of a given feature.  Each node  
examined but not explored is put into a priority queue to be explored  
later, keyed based on the distance from its partition key value to the  
given feature's desctiptor.  
  
@param kd\_node root of the subtree to be explored  
@param feat feature upon which branching decisions are based  
@param min\_pq a minimizing priority queue into which tree nodes are placed  
    as described above  
  
@return Returns a pointer to the leaf node at which exploration ends or  
    NULL on error.  
\*/  
//参数和返回值参看以上注释  
//搜索路径和优先级队列的生成函数  
static struct kd\_node\* explore\_to\_leaf( struct kd\_node\* kd\_node, struct feature\* feat,  
struct min\_pq\* min\_pq )  
{  
struct kd\_node\* unexpl, \* expl = kd\_node;　　//unexpl中存放着优先级队列的候选特征点  
　　　　　　　　　　　　　　　　　　　　　　　　　　　　　//expl为开始搜索节点  
double kv;　　　　　　　　　　　　　　　　　　　　 //kv是分割维度的数据  
int ki;　　　　　　　　　　　　　　　　　　　　    //ki是分割维度序号  
while( expl  &&  ! expl->leaf )  
    {  
        ki = expl->ki;　　　　　　　　　　　　　　　 //获得分割节点的ki，kv数据  
        kv = expl->kv;  
  
if( ki >= feat->d )  
        {  
            fprintf( stderr, "Warning: comparing imcompatible descriptors, %s" \  
" line %d\n", \_\_FILE\_\_, \_\_LINE\_\_ );  
return NULL;  
        }  
if( feat->descr[ki] <= kv )　　　　　　　　//目标特征和分割节点分割维上的数据比较  
        {  
            unexpl = expl->kd\_right;　　　　　　  //小于右子树根节点成为候选节点　  
            expl = expl->kd\_left;               //并进入左子树搜索  
        }  
else  
        {  
            unexpl = expl->kd\_left;　　　　　　　 //大于左子树根节点成为候选节点  
            expl = expl->kd\_right;　　　　　　　  //并进入右子树搜索  
        }  
　　　　 //将候选节点unexpl根据目标与分割超平面的距离插入到优先级队列中  
if( minpq\_insert( min\_pq, unexpl, ABS( kv - feat->descr[ki] ) ) )　　  
        {  
            fprintf( stderr, "Warning: unable to insert into PQ, %s, line %d\n",  
                    \_\_FILE\_\_, \_\_LINE\_\_ );  
return NULL;  
        }  
    }  
  
return expl;　　//返回搜索路径中最后的叶子节点  
}

　　从explore\_to\_leaf 函数的实现中可以看到，优先级队列和搜索路径是同时生成的，这也是BBF算法的精髓所在：在二叉搜索的时候将搜索路径另一侧的分支加入到优先级队列中，供 回溯时查找。而优先级队列的排序就是根据目标特征与分割超平面的距离ABS( kv - feat->descr[ki] )

注 意：是目标特征和分割超平面间的距离，不是候选节点和分割超平面的距离。如还是上两篇例子中的数据，查找（2,4.5）的k近邻，当搜索到（5,4）节点 时，应将（4,7）节点加入搜索路径而将（2,3）节点选为优先级队列的候选节点，优先级的计算是：abs（4 - 4.5） = 0.5。

来源： <<http://www.cnblogs.com/eyeszjwang/articles/2437706.html>>