華東理工大學

模式识别大作业

题　　目　Ants

学　　院　信息科学与工程

专　　业　控制科学与工程

组　　员　史亿帆

指导教师　赵海涛

完成日期：　2019 年 12 月 12 日

模式识别作业报告——Ants

组员：史亿帆

题目来源：http://acm.hdu.edu.cn/showproblem.php?pid=5809

一、题意

筷子先生在火星上发现了许多蚂蚁和蚁巢。这些蚂蚁的行为似乎与地球上的蚂蚁大不相同，因此筷子先生建了一个实验室来研究它们。在实验室里，有一块平地和 N 个蚁巢，从 1 到 N 编号。第 i 个巢的位置是(xi, yi)，没有两个巢在同一位置。经过一段时间的观察，筷子先生发现了所有蚂蚁都遵守的一些规律:

1、当蚂蚁在一个蚁巢 p 处时，它总是会移动到另一个离 p 最近的蚁巢。如果有多个蚁巢与 p 的距离最小，它就会移动到一个 x 坐标值较小的蚁巢。如果 x 相同，则选择 y 坐标值较小的那个。当蚂蚁从一个巢穴移动到另一个巢穴时，它总是沿着连接它们的线段移动。

2、蚂蚁从不停止，也就是说，当蚂蚁到达一个巢穴时，它会立即移动到下一个巢穴。所以，当然，一只蚂蚁可能会无数次地造访它的巢穴。

3、所有的蚂蚁都以同样的速度移动。

如果两只蚂蚁相遇了怎么办？他们会打架吗?筷子先生对这些问题很好奇。但首先，他应该知道两只蚂蚁是否会相遇。所以他考虑了下面的问题:给定两个不同的蚁巢，如果两只蚂蚁同时从这两个蚁巢中移动，它们会在移动的过程中相遇吗？请注意，所有的蚂蚁和巢穴都可以视为点。

输入：输入以一个整数 T (T <= 10)开始，表示测试用例的数量。每个 case 以一个整数 N 和 Q 开始(2 <= N <= 100000, 1 <= Q <= 100000)，分别表示蚁巢的数量和查询的数量。以下 N 行，每一行包含两个整数 xi 和 yi (-1000000000 <= xi, yi <= 1000000000)，表示第 i 个蚂蚁窝的位置。以下 Q 行各包含两个整数 i 和 j (1 <= i, j <= N, i≠j)，表示两个给定的蚁巢。

输出：对于每种情况，在一行中输出“case #X:”，其中 X 是从 1 开始的 case 编号。然后，对于每个查询，如果两个蚂蚁将相遇，则在一行中输出“YES”；否则输出“不”。

时间限制：6000/3000 MS (Java/Others)

内存限制：131072/131072 K (Java/Others)

二、题目分析

题目的大致意思是有很 n 个蚂蚁窝，蚂蚁窝里面的蚂蚁运动的时候有一个规

律，就是每次往距离它最近的蚂蚁窝走。当两对蚂蚁窝距离相同时，坐标小的那个更近。而且蚂蚁运动严格是走直线，现在有 q 个询问，每个询问给出两个蚂蚁窝的编号，问这两个蚂蚁窝的蚂蚁是否能够相遇。

首先，要解决的问题是找到最近的蚂蚁窝，考虑到蚂蚁窝的数量最多可能有 10^5 个，且限时 3s，遍历所有点的 $O(N^2)$的算法肯定会超时，考虑使用 K 最邻近分类算法即 KNN 中的优化算法 KD 树，其算法复杂度为 O(NlogN)，这种算法在数据集很大的情况下性能尤为优异。

其次，在找到了每个蚂蚁要前往的蚂蚁窝之后，我们需要知道任意两个蚂蚁窝中的蚂蚁是否会相遇。从题目中可以发现，“当蚂蚁到达一个巢穴时，它会立即移动到下一个巢穴”，即一只蚂蚁可能会无数次地造访它的巢穴。也就是说只要两只蚂蚁在前往新蚂蚁窝的途中相遇了那么他们一定会前往同一个蚂蚁窝，所以我们只需要知道任意两个蚂蚁是否会前往同一个窝即可。对于判断两个点是否在一个连通分量里，通常使用并查集进行查询和维护。

三、算法分析

1、KNN 和 KD 树

K 最近邻分类算法，即 KNN 是数据挖掘分类技术中最简单的方法之一。所谓 K 最近邻，就是 k 个最近的邻居的意思，说的是每个样本都可以用它最接近的 k 个邻居来代表。Cover 和 Hart 在 1968 年提出了最初的邻近算法。KNN 是一种分类算法，它输入基于实例的学习，属于懒惰学习即 KNN 没有显式的学习过程，也就是说没有训练阶段，数据集事先已有了分类和特征值，待收到新样本后直接进行处理。与急切学习相对应。KNN 是通过测量不同特征值之间的距离进行分类。

思路是：如果一个样本在特征空间中的 k 个最邻近的样本中的大多数属于某一个类别，则该样本也划分为这个类别。KNN 算法中，所选择的邻居都是已经正确分类的对象。该方法在定类决策上只依据最邻近的一个或者几个样本的类别来决定待分样本所属的类别。

KNN 算法三个基本要素：

（1）k 值的选取。（在应用中，k 值一般选择一个比较小的值，一般选用交叉验证来取最优的 k 值）

（2）距离度量。（Lp 距离：误差绝对值 p 次方求和再求 p 次根。欧式距离：p=2 的 Lp 距离。曼哈顿距离：p=1 的 Lp 距离。p 为无穷大时，Lp 距离为各个维度上距离的最大值）

（3）分类决策规则。（也就是如何根据 k 个最近邻决定待测对象的分类。k 最近邻的分类决策规则一般选用多数表决）

k 近邻模型实际上是基于训练数据集对特征空间进行切割划分，根据这个划分，来搜索包含搜索点最小的特征空间中的超矩形区域，利用多数表决确定所属的类，以此来预测输出值。

本质上讲，k 近邻模型也是一种分类器。所谓回归，也是利用最小二乘等方法将空间切割为数个子区域后，叶节点返回对应区域的输出，最后得到一个近似连续的线性模型。这与概率统计中的线性回归实际上异曲同工。统计学的线性回归也是采取最小二乘法导出正规方程组，整理得到回归系数，将离散数据拟合为一个直线或线性超平面以此预测其他点的输出值。

当我们的数据量很小的时候，我们可以使用线性扫描，计算输入实例和每一个训练实例的距离，但是当数据集很大时，这种方法就不太可行了。下面我们来介绍一下当数据量很大情况下，对于快速寻找 k 最近邻的算法——KD 树。

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数据结构。KD 树是是一种二叉树，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将 K 维空间切分，构成一系列的 K 维超矩形区域。KD 树的每个结点对应于一个 k 维超矩形区域。利用 KD 树可以省去对大部分数据点的搜索，从而减少搜索的计算量。

KD 树算法主要包括建树和查询两部分。

建树的基本思路：在每个根节点处选取该根节点对应的坐标 l=j mod k+1(jj 为根节点的深度, k 为空间维数)的值的中位数，将子区域以该坐标的中位数为“标准”分割为两个子区域，该中位数对应的切分点作为根节点，然后对左右两个子区域再递归进行刚才的步骤。

查询的基本思路：

（1）从 kd 树根节点出发，依次向下访问，若 searching point 当前维度的坐标小于切分点的坐标，访问左节点，反之访问右节点，直到子节点为叶节点以此叶节点为当前最近邻；

（2）回溯，递归地向上回退，若该节点保存的切分点比当前最近邻更近，则将该切分点置为当前最近邻。检查该节点另一个子节点（相对于回溯点）对应的区域是否与一个以搜索点为球心、以搜索点与当前最近邻之间的间距为半径的球体相交，若不相交，继续向上回溯，若相交，则从该子节点开始进行最近邻搜索；

（3）当回退到根节点，搜索结束。

2、并查集

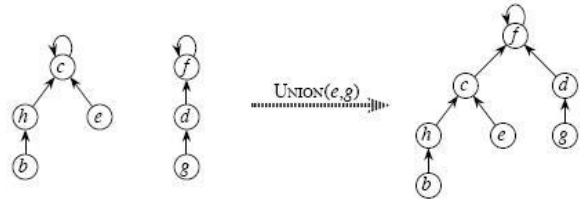
并查集是一种树形结构，又叫“不相交集合”，保持了一组不相交的动态集合，每个集合通过一个代表来识别，代表即集合中的某个成员，通常选择根做这个代表。

并查集是一种非常简单的数据结构，它主要涉及两个基本操作，分别为：合

并两个不相交集合以及判断两个元素是否属于同一个集合。

（1）合并两个不相交集合 Union(x,y)

合并操作很简单：先设置一个数组 Father[x]，表示 x 的“父亲”的编号。那么，合并两个不相交集合的方法就是，找到其中一个集合最父亲的父亲（也就是最久远的祖先），将另外一个集合的最久远的祖先的父亲指向它。



上图为两个不相交集合，在右图中 Father(b):=Father(g)

```
inline int find(int x){
    if(x!=f[x])
    f[x]=find(f[x]);
    return f[x];
}
```

（2）判断两个元素是否属于同一集合 Find_Set(x)

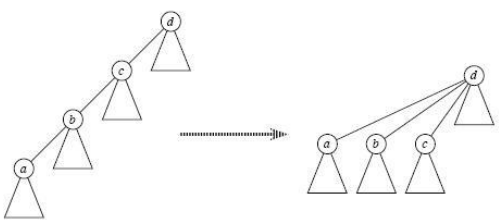
本操作可转换为寻找两个元素的最久远祖先是否相同。可以采用递归实现。

```
inline void join(int x,int y){
     int fx=find(x);
     int fy=find(y);
     if(fx!=fy){
     f[fx]=fy;}
}
```

对并查集的优化：

（1）Find_Set(x)时，路径压缩

寻找祖先时，我们一般采用递归查找，但是当元素很多亦或是整棵树变为一条链时，每次 Find_Set(x)都是 O(n)的复杂度。为了避免这种情况，我们需对路径进行压缩，即当我们经过“递推”找到祖先节点后，”回溯”的时候顺便将它的子孙节点都直接指向祖先，这样以后再次 Find_Set(x)时复杂度就变成 O(1)了，如下图所示。可见，路径压缩方便了以后的查找。



（2）Union(x,y)时，按秩合并

即合并的时候将元素少的集合合并到元素多的集合中，这样合并之后树的高度会相对较小。

```
int find(int x){
    return f[x]==x?x:find(f[x]);
}
```

四、代码实现

考虑到读入的数据量较大，使用快速读入来取代 scanf 的读入方式。因为 getchar()比 scanf 要快,故为了加快读入,可以用 getchar()代替 scanf。

思路：利用 getchar()将数字读入,若为空格或\n 结束,第一个字符判断一下数字的正负,然后每读入一个数字就将当前数*10 并加上它。

```
template <class T>
inline bool read(T &ret) {
    char c; int sgn;
    if(c=getchar(),c==EOF) return 0; //EOF
    while(c!='-'&&(c<'0'||c>'9')) c=getchar();
    sgn=(c=='-')?-1:1;
    ret=(c=='-')?0:(c-'0');
    while(c=getchar(),c>='0'&&c<='9') ret=ret*10+(c-'0');
    ret*=sgn;
    return 1;
}
```

构建 KD 树中节点的结构体：

```
struct Point
{
    int x[2], id;
    bool operator < (const Point& p) const{
        if(x[0] == p.x[0]) return x[1] < p.x[1];
        return x[0] < p.x[0];
    }
}
```

其中，x[0]表示 x 轴的值，x[1]表示 y 轴的值，id 记录节点的初始编号。对小于< 运算进行了重定义，这是为了能够直接将两个节点进行比较，满足优先比较 x 轴

的值，再比较 y 轴的值。

KD 树的构建：

```
void build(int l, int r, int d)
{
    if(l >= r) return;
    int mid = l + r >> 1;
    cur_d = d & 1;
    nth_element(node + l, node + mid, node + r + 1, cmp);
    build(l, mid - 1, d + 1);
    build(mid + 1, r, d + 1);
}
```

其中 cur_d 表示当前分割的维度，0 表示 x 轴，1 表示 y 轴。在 c++的 stl 库中，提供了 nth_element 这样一个函数，它的用法是 nth_element(a+l,a+k,a+r)。这样它会使 a 这个数组中区间[l,r)内的第 k 小的元素处在第 k 个位置上(相对位置)，但是它并不保证其他元素有序。该排序的时间复杂度为 O(N)。

KD 树的查询：

```
void query(int l, int r, int d)
{
    if(l > r) return;
    int mid = l + r >> 1;
    ll d_sq = dis_sq(node[mid], n_point);
    if (d_sq && (d_sq < n_dis || d_sq == n_dis && node[mid] < point[n_id])) {
        n_id = node[mid].id;
        n_dis = d_sq;
    }
    cur_d = d & 1;
    ll radius = (ll)(n_point.x[cur_d] - node[mid].x[cur_d])*(n_point.x[cur_d] -
node[mid].x[cur_d]);
    if (n_point.x[cur_d] < node[mid].x[cur_d]) {
        query(l, mid - 1, d + 1);
        if (radius <= n_dis) {
            query(mid + 1, r, d + 1);
        }
    }
    else {
```

```
            query(mid + 1, r, d + 1);
            if (radius <= n_dis) {
                query(l, mid - 1, d + 1);
            }
        }
}
```

其中，n_dis 表示当前查询到的最近距离，在每次调用 query(l, r, d)前，将 n_dis 设置为无限大，这里取 0x3f3f3f3f3f3f3f3fll；n_id 记录最近点的编号；radius 则是计算查询点到分裂面的距离的平方。比较距离的时候，其实只要比较距离的平方就可以了，这样节省了开方的时间，加快运行速度。

并查集的构建：

```
int fa[MAXN];
int find(int x)
{
    return x == fa[x] ? x : fa[x] = find(fa[x]);
}
```

题目中给了一组非常简单的测试数据，只能用来验证大致思路是否正确，但不能验证算法的复杂度是否满足要求。

测试数据：

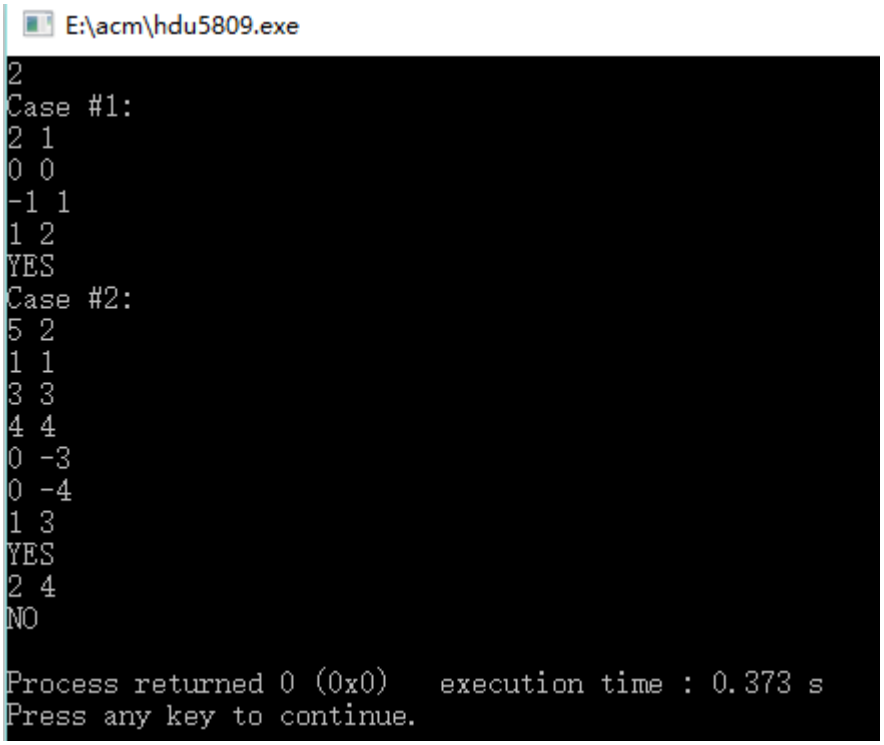
Sample Input

```
2
2 1
0 0
-1 1
1 2
5 2
1 1
3 3
4 4
0 -3
0 -4
1 3
2 4
```

Sample Output

```
Case #1:
YES
Case #2:
YES
NO
```

进行测试，得到结果：



可以看到与样例的输出一致。进一步检验算法的复杂度。

这里考虑自己构建最大的数据量进行测试分析，数据的构建如下：

```
int T = 10;
printf("10\n");
while(T--) {
    int a = 1000000000;
    printf("100000 100000\n");
    for(int i = 0; i < 100000; ++i) {
        printf("%d %d\n", random(1000000000), random(1000000000));
    }
    for(int i = 0; i < 100000; ++i) {
        printf("%d %d\n", random(100000), random(100000));
    }
}
```

通过 5.78s 的时间大约形成了 32686KB 的文件，进行本地测试，结果为：

测试数据为 10 组 100000 个蚂蚁窝，100000 次查询，满足 3000ms 的限时。

提交到网上进行判题，结果为：

31723140	2019-12-02 02:28:34	Accepted	5809	1606MS	6516K	2876 B	G++	shyf1301	
58	shyf1301	1606MS	6516K	2876B	G++	2019-12-02 02:28:34			

五、总结

在之前搞算法的时候就遇到过这道题，但是由于当时知识的欠缺无法解决，在本学期学习了模式识别的课程后，了解到了 KNN 算法，这是一种寻找邻近点的算法，在阅读了李航的《统计学习方法》后了解到了实现 KNN 算法的一种优化算法 KD 树，这种优化算法非常适合处理数据量很大的情况，这正是解决这道题的关键。二维的 KD 树可以看做是一个二叉树，在我看来用到的思想是分治法，于是便考虑试着去写一下 KD 树。

KD 树首先通过不断划分空间的方法，规划一条 logN 级别的搜索路径来探测最近点。也就是说，对当前 N 个点，在某个维度选取中位点，然后将点集在这个维度以中位点坐标分成两部分，往目标点所在的那部分去查询。然后递归进入相应的那一半区间重复此操作。这样就可以在这条渐进路上得到一个最小值。 但显然有这样一种情况，跨过分界线的点更优。对于这种情况有个必要条件就是目标点到分解线的距离比已知的最小值要小。所以 KD 树是在回溯的时候在以目标点为圆心，当前最小值为半径做圆，如果和分界线相交那么在递归另外一边。

按照这个思路写出了 KD 树在 C++中的实现，在刚写完的代码中没有在每次查询之初对 n_dis 进行重新赋值，导致了无法找到正确的最邻近的蚂蚁窝，经过调试后发现了这个问题。现在很多算法都进行了打包，我们在使用的时候只要直接调包即可，这大大提升了我们编码的速度，但却无法使我们深入理解算法的实现方法。通过这次自己动手编写 KD 树算法，并结合最美妙的算法——并查集，解决了一个数据量庞大的问题，可以说受益匪浅。

六、展望

由于时间有限，本次作业使用的是最基本的 KD 树算法，实现效果较为理想，但注意到使得 KD 树处理查询问题时复杂度从 O(N)退化到 O(logN)的关键是存在跨越分界线的点更优的情况，因此减少跨越分界线的点的情况便可以提高算法的效率。考虑可以在划分的时候选择尽可能稀疏的维度，来避免在靠近根的地方产生额外分支。至于选择哪一维度，可以考察在不同维度下的点的方差。同时这种算法可以考虑应用到群体机器人的规划中。

七、文件说明

ants_in.cpp 测试数据的生成

ants.cpp 主程序

in.txt 测试数据

out.txt 测试数据输出