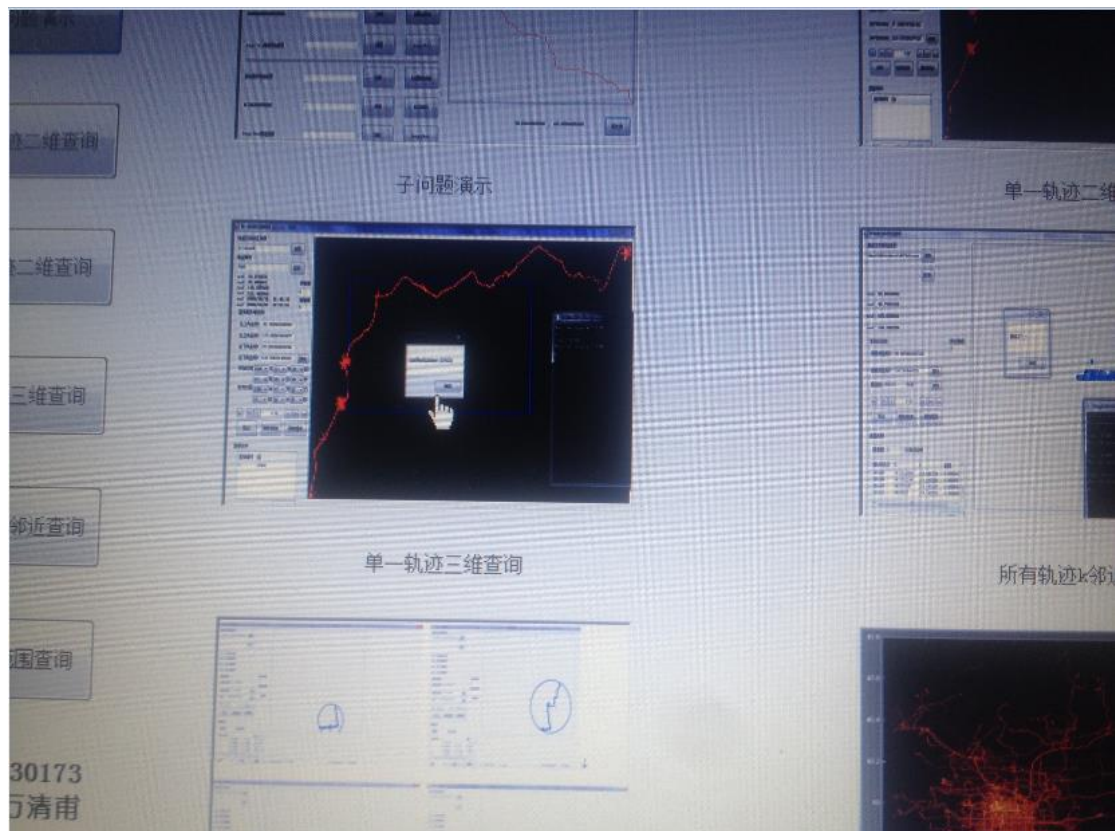
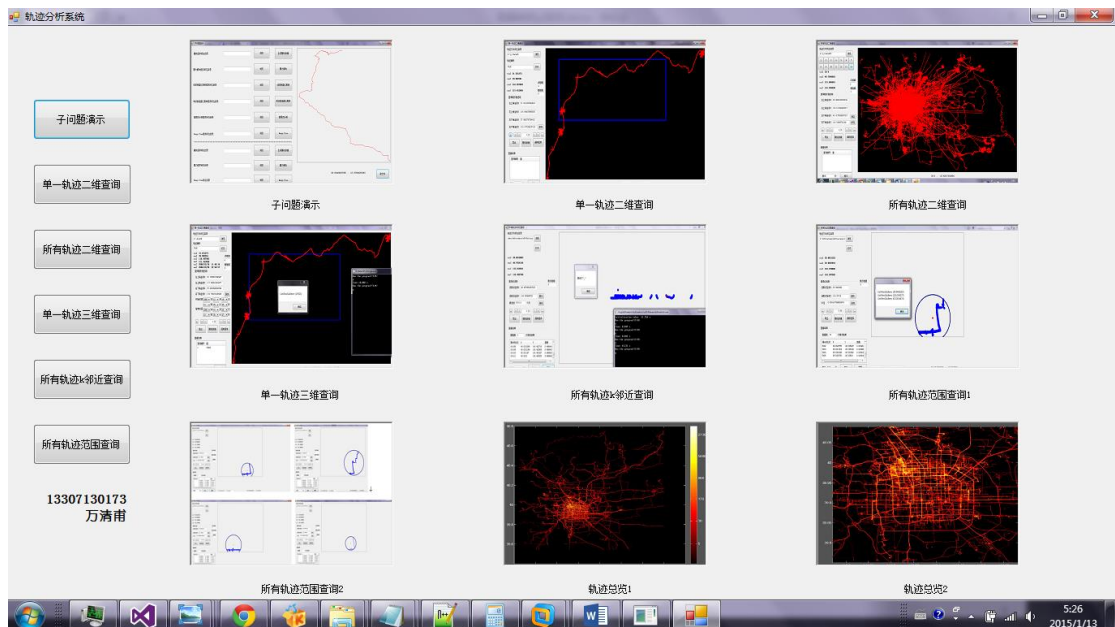


数据结构 pj 报告

1. 摘要

主界面



移动到大框框中后鼠标会变成手形，单击进入相关程序。

本程序分析数据来源于北京市出租车在04/2007-08/2012的18670条轨迹，去除错误数据后共有18507条轨迹，所有轨迹中的点纬度范围在 $39.6^{\circ}\text{N} - 40.7999983^{\circ}\text{N}$ 经度范围在 $115.800001^{\circ}\text{E} - 116.999998^{\circ}\text{E}$ 。

主要有以下几个功能：

- 1) 单一轨迹二维查询
- 2) 所有轨迹二维查询
- 3) 单一轨迹三维查询
- 4) 所有轨迹k邻近查询
- 5) 所有轨迹范围（圆形区域）查询

2. 子问题演示

2.1 问题1：一条轨迹上有 n 个点， m 组询问，每组询问是一个搜索矩形框，每次询问轨迹上有多少点在矩形框中。

1.1.1 暴力解法（复杂度 $O(n * m)$ ）

1.1.2 线段树套红黑树

n 轨迹中点数 nx 这条轨迹中横坐标不同的坐标数 ny 纵坐标不同的坐标数 m 为查询次数
查询矩形左下角坐标为 (xa,xb) 右上角坐标为 (xc,xd)

- 1) 将轨迹按横坐标排序离散化 坐标保存在 $tx[1]..tx[nx]$ 中
- 2) 将轨迹按纵坐标排序离散化 坐标保存在 $ty[1]..ty[ny]$ 中
- 3) 按照横坐标建立一颗线段树 每个线段树区间的节点上再挂一颗平衡树

$[l,r]$ 区间所对应的平衡树保存了所有横坐标排序后在 $[l,r]$ 的点，这些点在这个线段树区间对应的平衡树中按照纵坐标排序后的下标（离散化后的结果）按照从小到大的顺序储存在平衡树中（相当于是本来普通排序先按 x 为第一关键字， y 为第二关键字，线段树就是对 x 的排序，平衡树是对 y 的排序）
不知道有没有描述清楚

这样的话空间复杂度是 $n \log_2 nx$ （每个节点都最多在 $\log_2 nx$ 个区间中出现，所以也在至多 $\log_2 nx$ 颗平衡树中出

现；此外线段树空间复杂度也是 $n \log_2 nx$ ）次查询的时候，最多进入 $\log_2 nx$ 个区间，每个区间还要深入进入对应的平衡树找到这棵树中最小的纵坐标 $\geq xb$ 的节点，以及最大的纵坐标 $\leq xd$ 的节点，这两个节点在平衡树中序遍历中出现先后顺序之差+1就是说对应的这颗线段树区间（横坐标在 $tree[x].l$ 到 $tree[x].r$ 中的所有点）中纵坐标在 xb 到 xd 之间的点数，累加到 ans 中。由于深入平衡树

查找这两个节点分别需要最多 $\log_2 n$ 次 所以单次查询的复杂度为 $\log_2 n \log_2 nx$ 的复杂度

1.1.3 树状数组套红黑树

本来以为树状数组常数小会比上面方法快的，测试了一下发现更慢。

1.1.4 象限四分树

1) PR四分树

四个象限每个象限点数均等。

本来以为会很快，实验结果发现不是这样。

2) 普通四分树

每个象限只有一个点。

1.1.5 带分层折叠的区域树 Range Tree with Fractional Cascading

具体见 Range Search.pptx

1.1.6 效率对比

| 序号 | 算法 | n (点数) | m (询问数) | 时间 |
|----|-----------|--------|---------|------|
| 1 | 暴力 | 300000 | 1000000 | 210s |
| 2 | 线段树套红黑树 | 300000 | 1000000 | 14s |
| 3 | 树状数组套红黑树 | 300000 | 1000000 | 28s |
| 4 | PR四分树 | 300000 | 10000 | 15s |
| 5 | 普通四分树 | 300000 | 100000 | 7s |
| 6 | 带分层折叠的区域树 | 300000 | 1000000 | 5s |

可见：

区域树大概是线段树套红黑树的3倍，线段树套红黑树大概是树状数组套红黑树的2倍，树状数组套红黑树大概是普通四分树的5倍，普通四分树大概是暴力的3倍，暴力大概是PR四分树的7倍，并且这种差距随着询问数不断增加越来越明显。

区域树是效率最高的。

2.2 问题2:

一条轨迹上有n个点，m组询问，每次选取一个搜索立方体，询问轨迹上有多少点X坐标在(x1, x2)，Y坐标在(y1, y2)，Z坐标在(z1, z2)范围内。（Z表示时间）

2.2.1 暴力

2.2.2 二维Range Tree扩展到三维即可

主要程序模块还是一样的，稍微改一下代码即可。

实验结果：

| 序号 | 算法 | n（点数） | m（询问数） | 时间 |
|----|-----------|--------|---------|-----|
| 1 | 暴力 | 300000 | 10000 | 22s |
| 2 | 带分层折叠的区域树 | 300000 | 1000000 | 22s |

区域树是暴力的100倍，并且暴力算法复杂度和点数有关，而区域树复杂度只与查询数有关。

2.3 截图



2.4 总览

| 序号 | 算法 | exe所在目录 | 完成时间 | 行数 |
|----|--------------|---|------------|-----|
| 1 | 暴力（二维） | G:\pj\线段树（树状数组）套红黑树\pre | 2014/12/22 | 27 |
| 2 | 线段树套红黑树（二维） | G:\pj\线段树（树状数组）套红黑树\pre\sts\x64\Release | 2014/12/22 | 329 |
| 3 | 树状数组套红黑树（二维） | G:\pj\线段树（树状数组）套红黑树\pre\sts2\sts2\x64\Release | 2014/12/23 | 293 |
| 4 | PR四分树（二维） | G:\pj\象限四分树2\preequal\preequal\x64\Release | 2014/12/25 | 241 |

| | | | | |
|---|-------------------|--|------------|-----|
| 5 | 普通四分树 (二维) | G:\pj\象限四分树1\pre2\x64\Release | 2014/12/25 | 194 |
| 6 | 带分层折叠的区域树 (二维) | G:\pj\Range Tree with Fractional Cascading(2D)\prerangetree\prerangetree\x64\Release | 2014/12/27 | 250 |
| 7 | 暴力(三维) | G:\pj\三维查询暴力\pre3D | 2014/12/27 | 28 |
| 8 | 带分层折叠的区域树 (三维) | G:\pj\Range Tree with Fractional Cascading(3D)\prerangetree3D\prerangetree3D\x64\Release | 2014/12/27 | 378 |

3. 单一轨迹二维查询

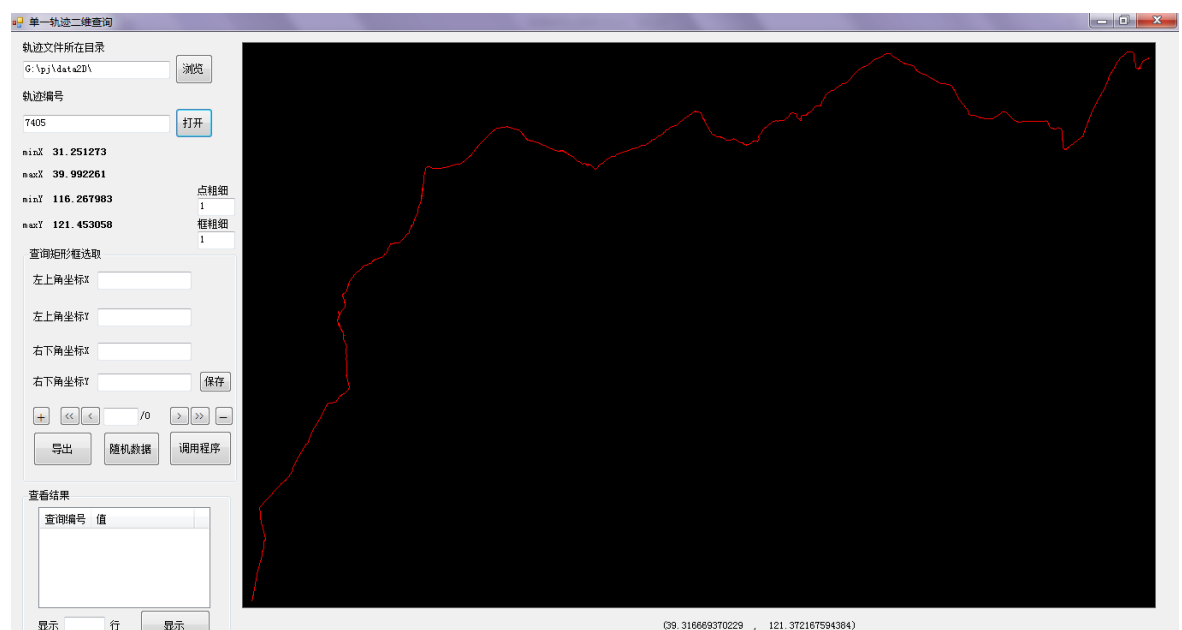
3.1 设置轨迹所在目录及轨迹编号

G:\pj\data2D\

(cpp路径为G:\pj\2Dsingle\2Dsingle\2Dsingle\2Dsingle.cpp)

保存了1.in到18507.in。

单击“打开”按钮。



```
minX 39.98632
maxX 39.987355
minY 116.302033
maxY 116.304461
```

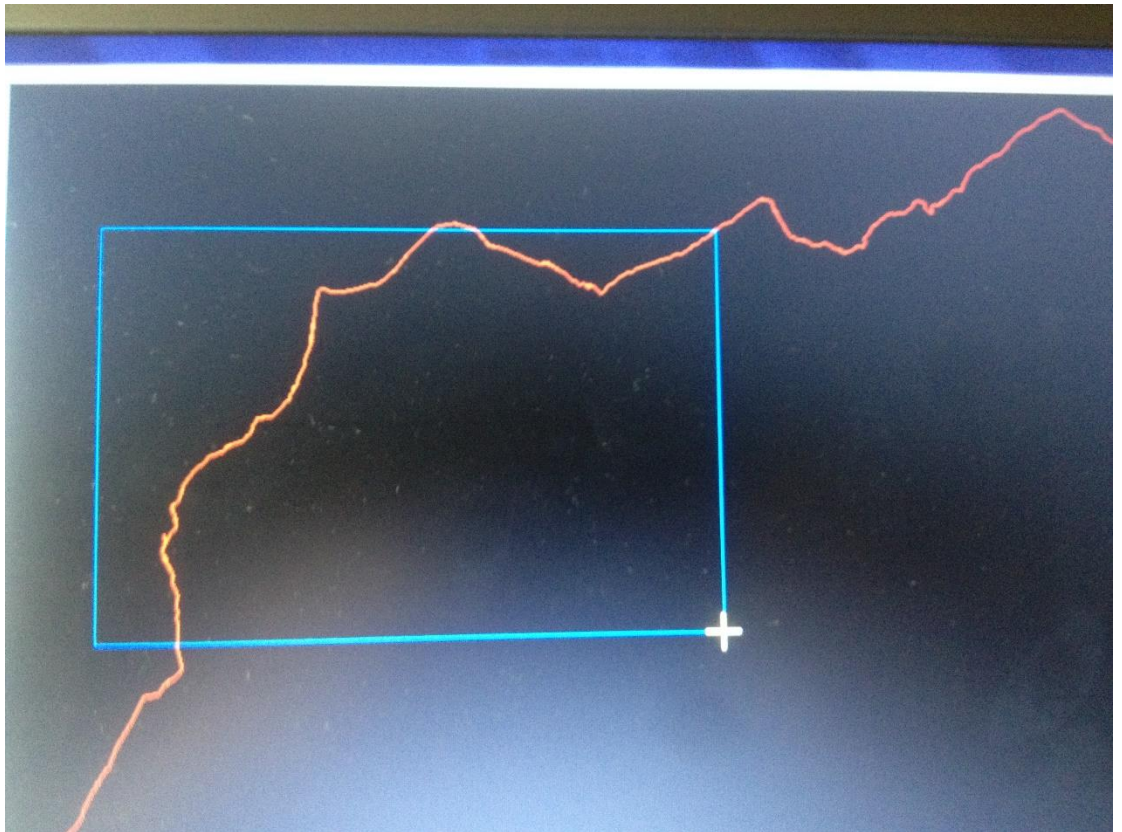
显示了这条轨迹的坐标范围。

鼠标在该区域移动，最底下的一行显示当前鼠标所在点的坐标。

```
(39.9868523139313 , 116.304040550702)
```

3.2 设置矩形框

用鼠标可以在平面上选取一个矩形框。（每次矩形自动重绘制）



查询矩形框选取

```
左上角坐标X 39.983776126145
左上角坐标Y 116.313385714509
右下角坐标X 39.9845980824428
右下角坐标Y 116.33231624493
```

会显示当前选取框的左上角和右下角坐标。

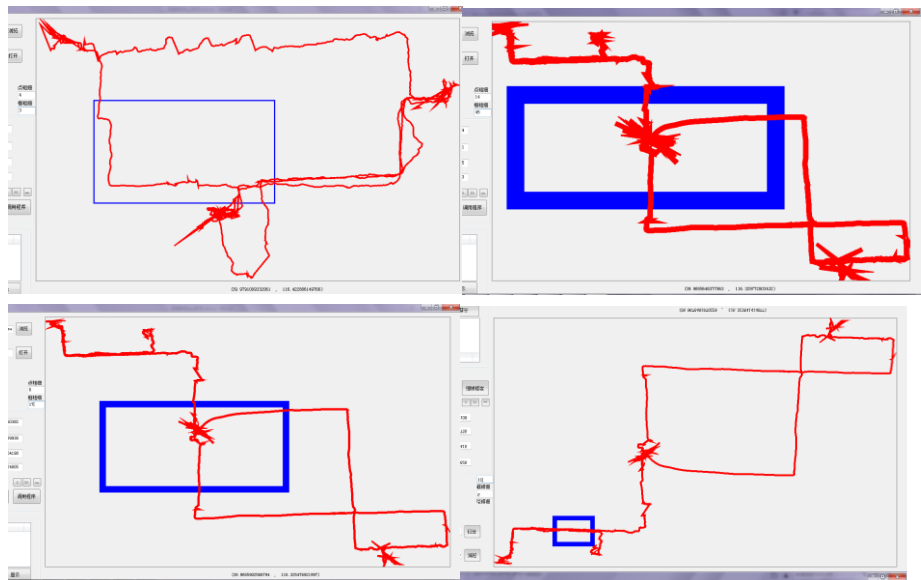
点粗细

2

框粗细

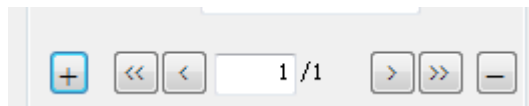
1

可以修改点的粗细和矩形框的粗细（可以为实数）每次修改后下次重画时就能看到改变。



算是一个附加功能。

3.3 选取多个矩形框



+表示新增一个询问并保存当前矩形框

-表示删除一个询问

< 表示查询上一个询问

> 表示查询下一个询问

<< 表示跳到第一个询问

>> 表示跳到最后一个询问

按“-”按钮后

查询矩形框选取

左上角坐标X 39.9644375076336

左上角坐标Y 116.414502322933

右下角坐标X 39.9733909370229

右下角坐标Y 116.420728686427

+

<<

<

3 / 3

>

>>

-

导出

随机数据

调用程序

查询矩形框选取

左上角坐标X 39.9644375076336

左上角坐标Y 116.414502322933

右下角坐标X 39.9733909370229

右下角坐标Y 116.420728686427

+

<<

<

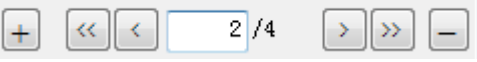
2 / 2

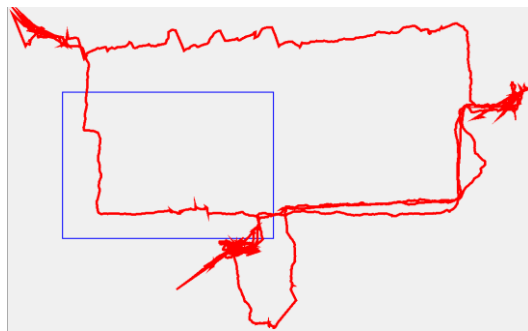
>

>>

-

变成

在  文本框中输入要查询的某组询问，
按回车后会在右边显示查询的矩形框。




39.9693165582061

116.32501323869

39.9726564007634

116.328286397816

在 修改坐标可以及时看到右边矩形框的变化。

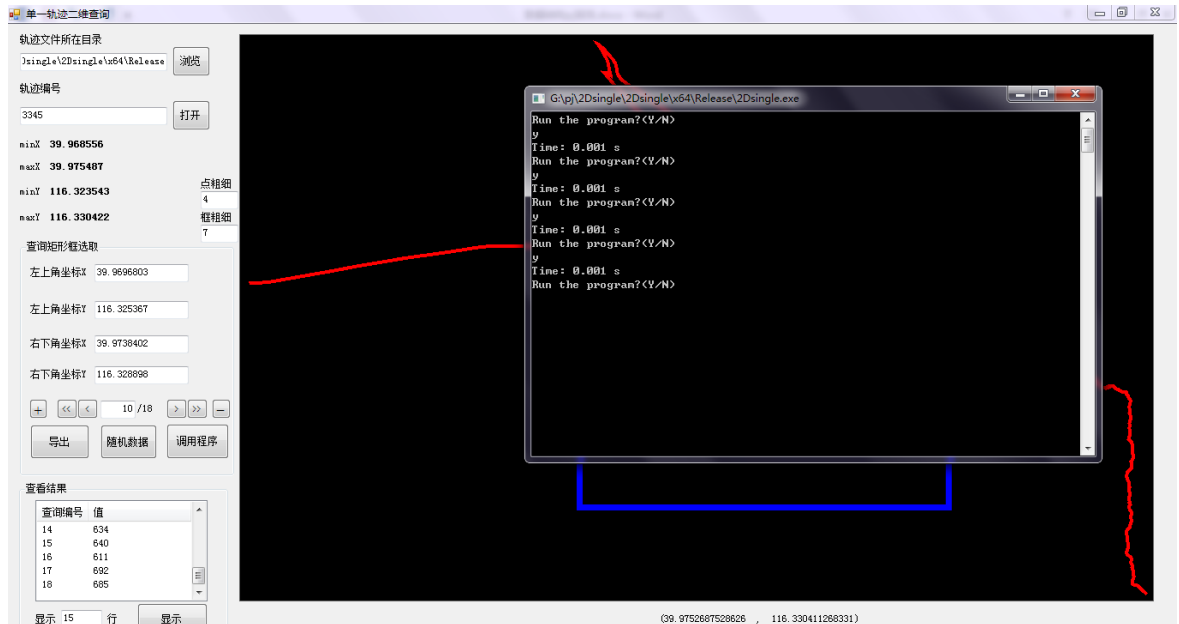
按  按钮保存当前询问矩形框。



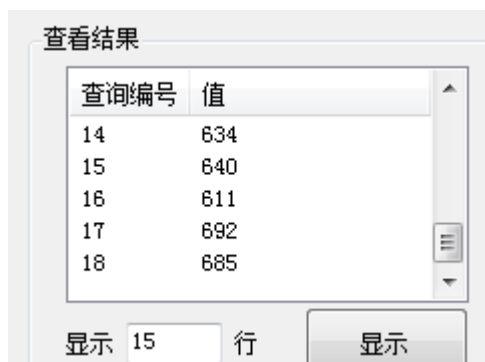
3.4 导出数据调用程序显示结果



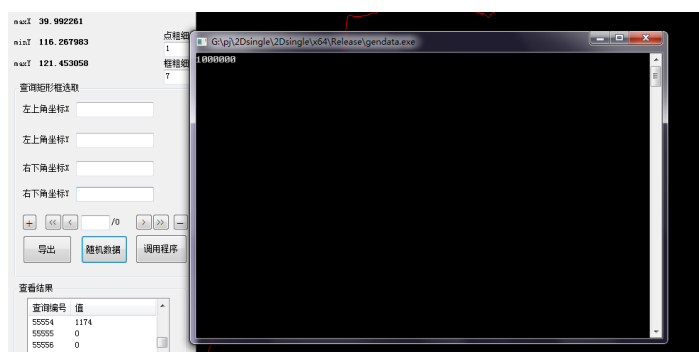
按“导出”、“调用程序”后会调用一个c++程序，每次输入Y可以反复查询：

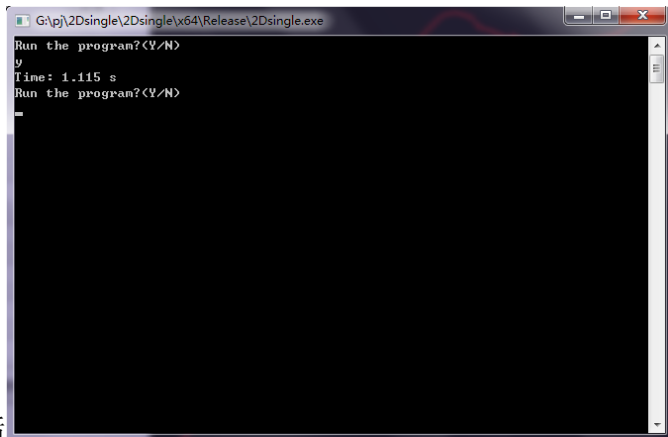


按“显示”可以将查询结果输出到列表框中，“值”显示的是对于这组询问这条轨迹中有多少点在搜索矩形框中。（列表加载比较慢其实可以用notepad++或者记事本打开）



按“随机数据”后程序产生一个随机生成的若干矩形框：

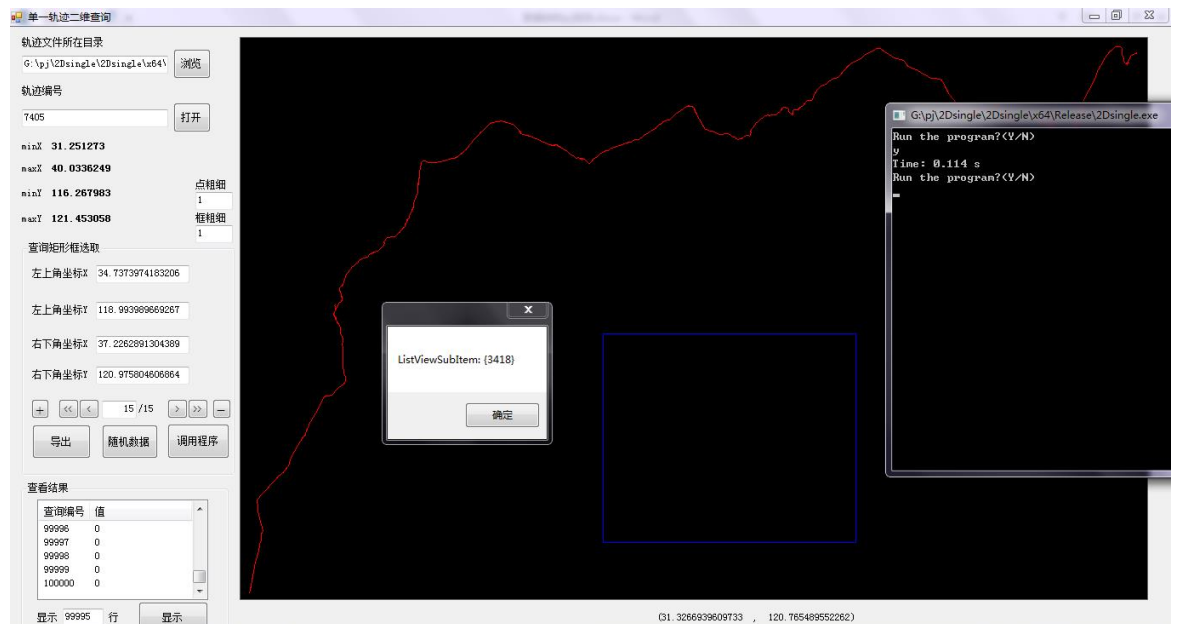




按“调用程序”后

对于7405.in 92465个点 1000000组询问 需要1.115s

3.5 总体效果



4. 所有轨迹二维查询

4.1 设置轨迹所在目录及轨迹编号

设置目录为G:\pj\data2D(cpp在

G:\pj\prerangetreeall\prerangetreeall\prerangetreeall\prerangetreeall.
1. cpp)

每次询问一个矩形框，问所有18507条轨迹有多少条和这个矩形相交。（轨迹中至少有一个点在矩形内）

由于c#画图DrawLines有bug 一次性画很多点会莫名其妙挂掉。

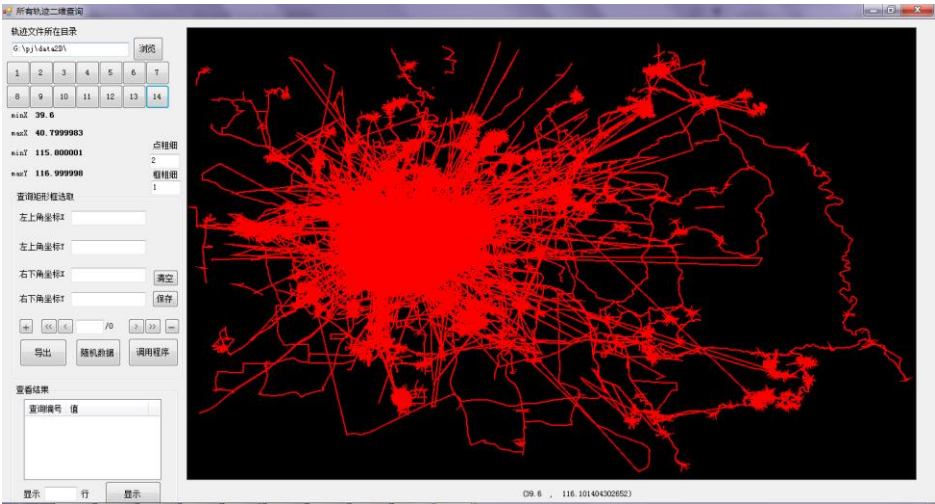
于是就设置了14个按钮画16114028个点。

DrawLines函数画线比较慢。

有考虑过在bitmap位图上作图 但是后来时间不够就没改。

画的图很多点都聚集在一块，本来想设置缩放、移动功能的，但是缩放涉及到变换矩阵，以前图像处理写过一次双三次插值，觉得太麻烦效率不高也不是本题重点就没有加进去这个功能。

画完图后是这样，比较丑T_T



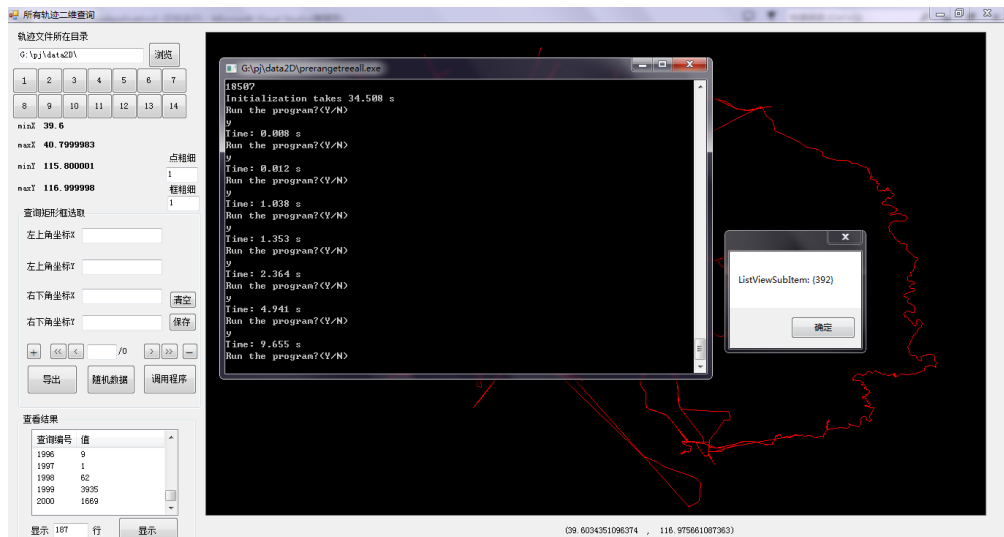
选矩形框的时候每次选完后不会重画这么多个点，因为这个画图函数太慢了

单击“调用程序” 首先对18507条轨迹进行预处理。

预处理需要34.508s。

测试效果如下 效率应该还是比较高的。（平均1s 200—250组询问）

| Q | Time |
|------|--------|
| 1 | 0.008s |
| 2 | 0.012s |
| 200 | 1.038s |
| 250 | 1.353s |
| 500 | 2.364s |
| 1000 | 4.941s |
| 2000 | 9.755s |



5. 单一轨迹三维查询

5.1 设置轨迹所在目录及轨迹编号

类似二维查询 设置为G:\pj\data3D (cpp路径)

G:\pj\3Dsingle\3Dsingle\3Dsingle\3Dsingle.cpp)

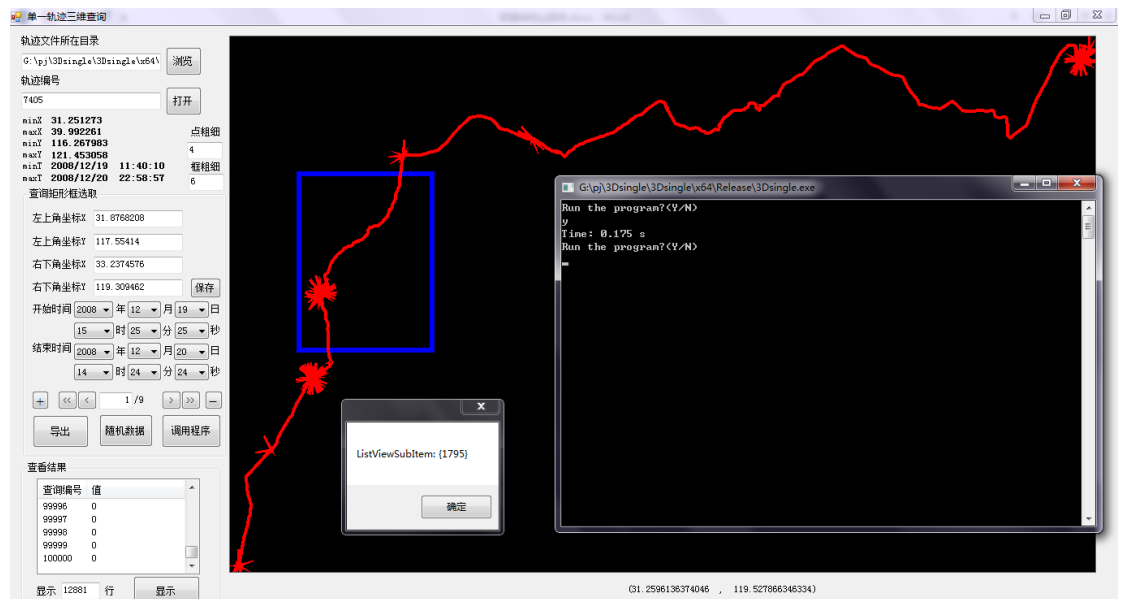
5.2 其它同上

截图

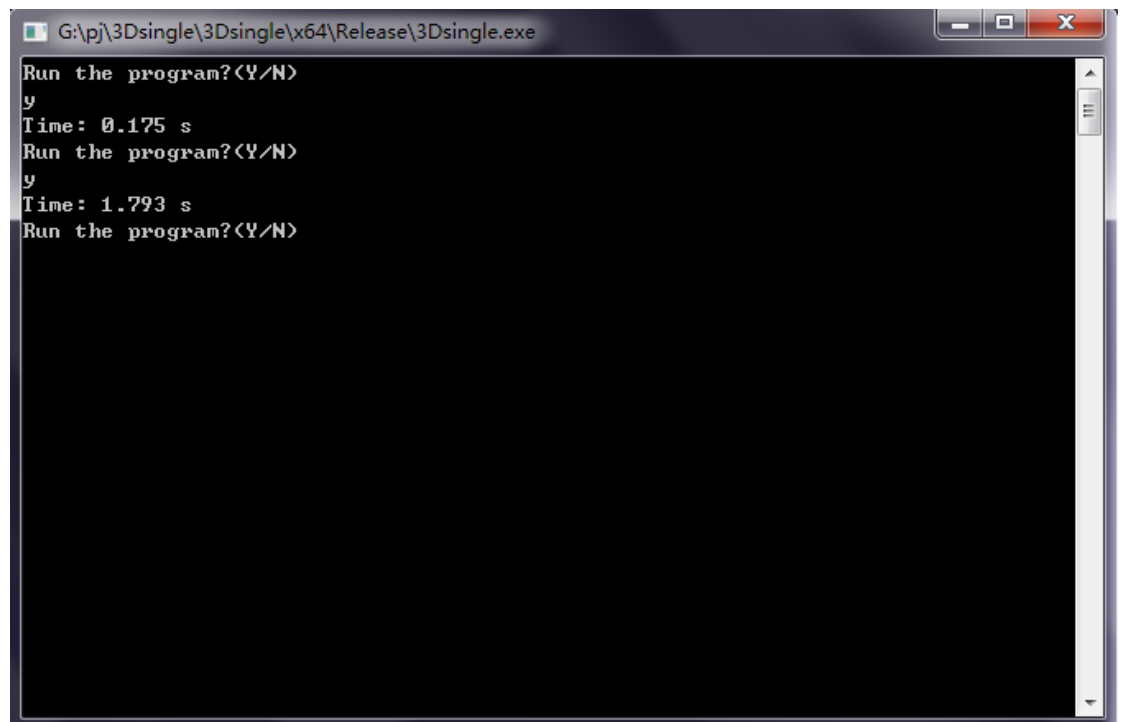


5.3 随机数据测试

100000组询问 0.175s



1000000组询问 1.793s

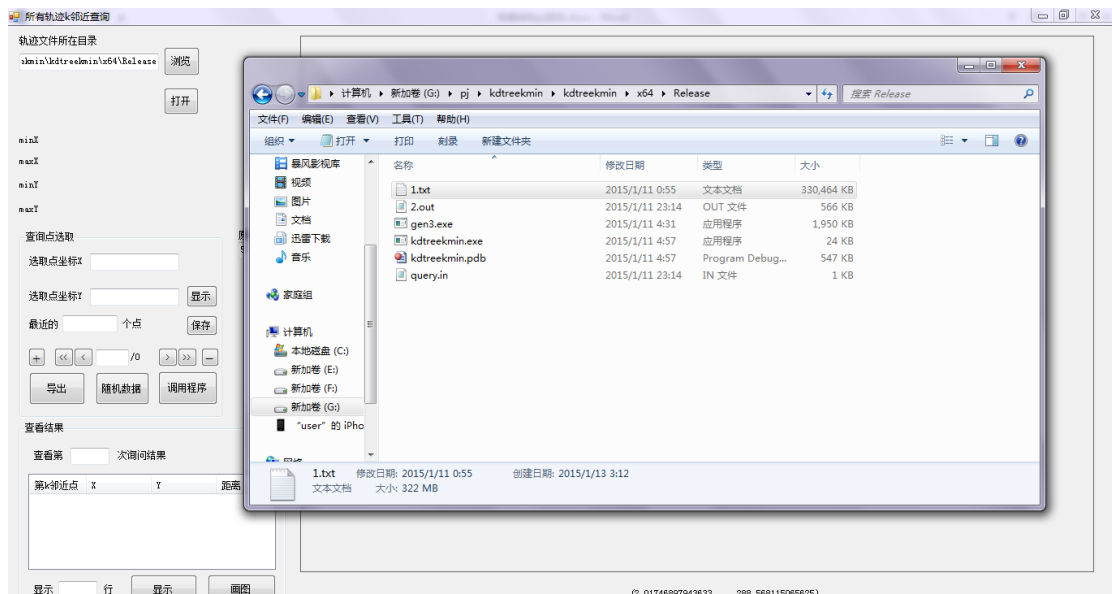


6. 所有轨迹k邻近查询

6.1 解决的问题

对平面上任意一点(x,y)询问所有轨迹中的点离(x,y)距离最近的k个点（欧式距离） 这个问题是范围查询：以(x,y)为圆心，r为半径的圆内有多少点（即求距离(x,y)在r范围内的点数）的子问题。

6.2 打开文件



轨迹文件所在目录

`akmin\kdtreemin\x64\Release`

程序每次读取的是轨迹文件所在目录

(G:\pj\kdtreemin\kdtreemin\x64\Release) 下的 1. txt 文件。(全部轨迹

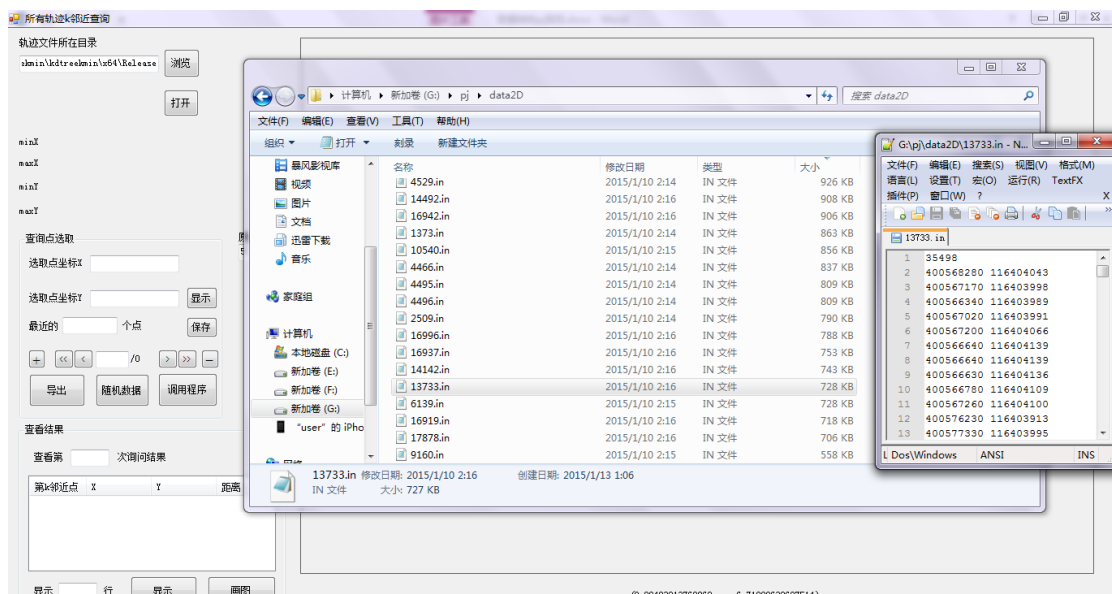
数据 G:\pj\data2Dpoint\data2Dpoint.in)

如果只需要读取某条特定轨迹，只要从 G:\data2D\ 中选一个 in 文件复制

轨迹文件所在目录

`akmin\kdtreemin\x64\Release`

下并保存为 1. txt 文件。



minX 39.6019083
maxX 40.7934316
minY 115.818641
maxY 116.992789

单击“打开”后 显示经纬度坐标范围
此时所有点不显示。

6.3 测试

初始化需要51.766s;
查询离红点最近的161111个点 算法运行时间为0.235s。

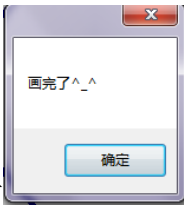
在 查看第 1 次询问结果 文本框中输入要查询的询问编号

| 第k邻近点 | X | Y | 距离 |
|--------|-----------|------------|----------|
| 119080 | 40.054597 | 116.397597 | 0.433092 |
| 119081 | 40.054595 | 116.397539 | 0.433092 |
| 119082 | 40.054806 | 116.403043 | 0.433092 |
| 119083 | 40.054598 | 116.397636 | 0.433092 |
| 119084 | 40.054582 | 116.397141 | 0.433092 |

显示 行 显示 画图

列表框里显示第k邻近点

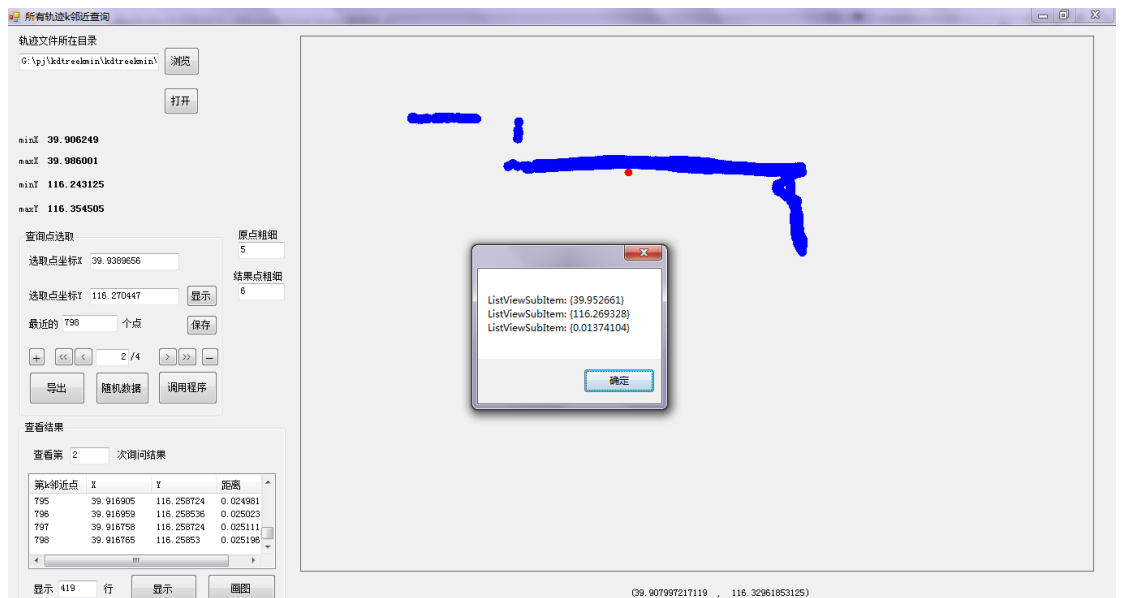
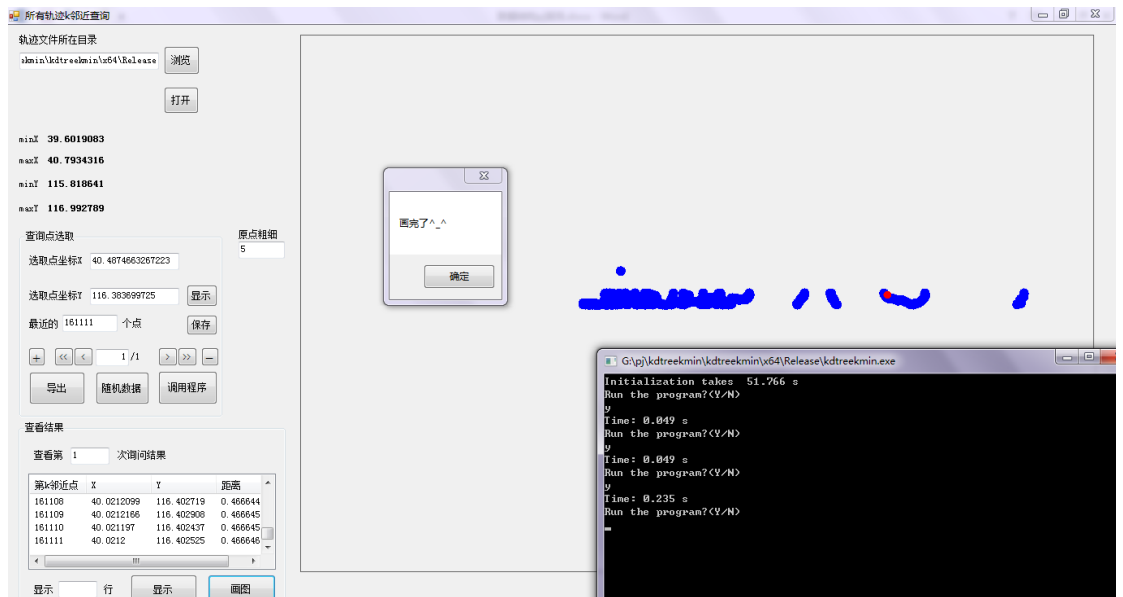
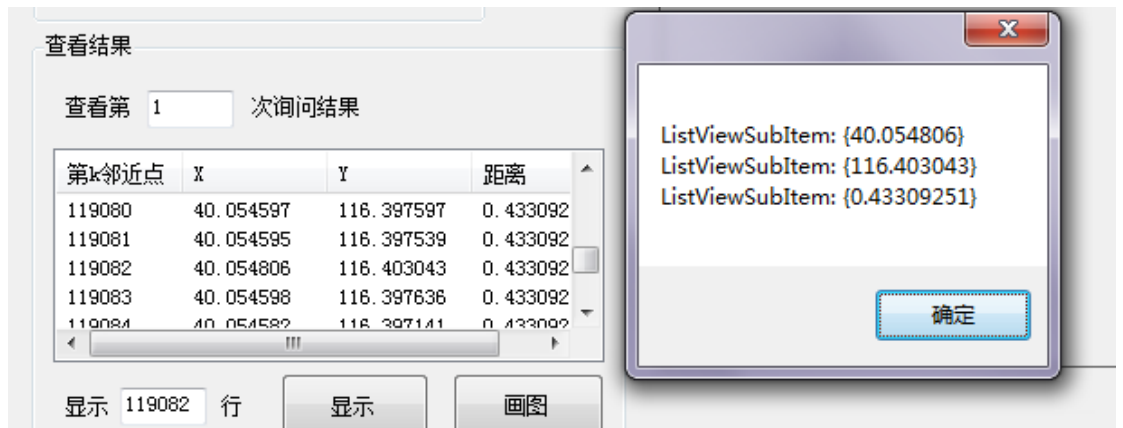
的X Y坐标及距离查询点的距离
先单击“显示” 然后再单击“画图” 还是由于c#画图的bug

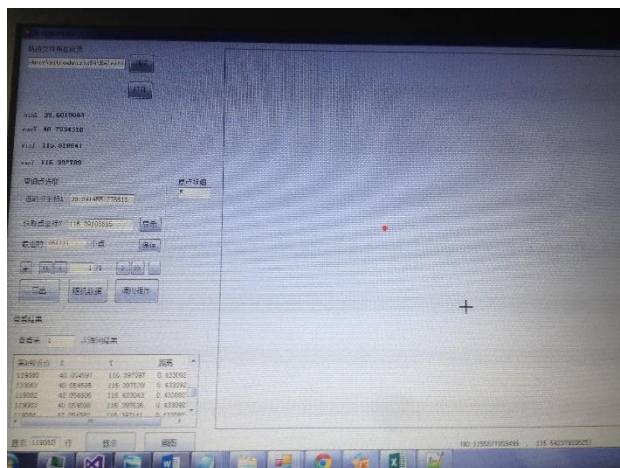


每次画图按钮画1000个点 多次单击画完整张图后会有提示

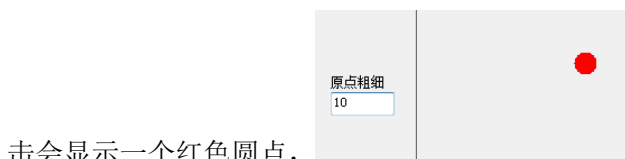
在 显示 119082 行 中输入要显示的第k邻近点

消息框三行分别为X Y坐标及距离查询点的距离



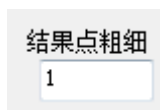


鼠标呈十字形，在任一点单

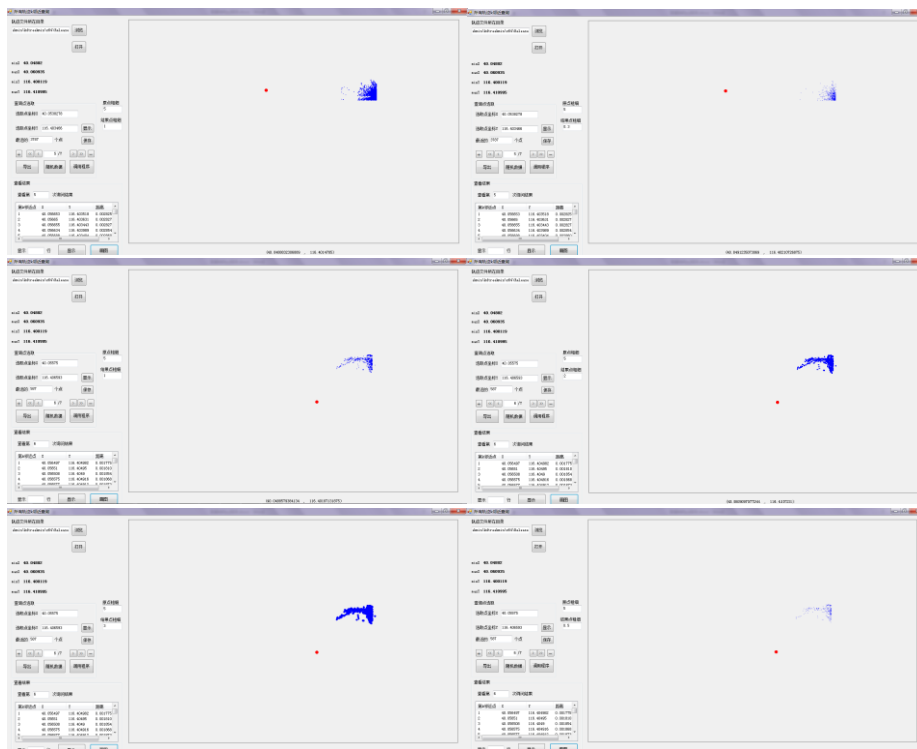


击会显示一个红色圆点，

可以设置圆点粗细。



可以控制结果点粗细 对比下图



6.4 算法实现

用的是最简单的没有优化过的kd-tree，维护一个大小为k的priority_queue。虽然这个功能和轨迹分析关系不大，但是感觉在地图里面还是挺有用的。比如问距离某个点最近的k个电影院（书店、购物广场、超市、医院、银行……）再扩展到距离某个点一定范围内有多少饭店……

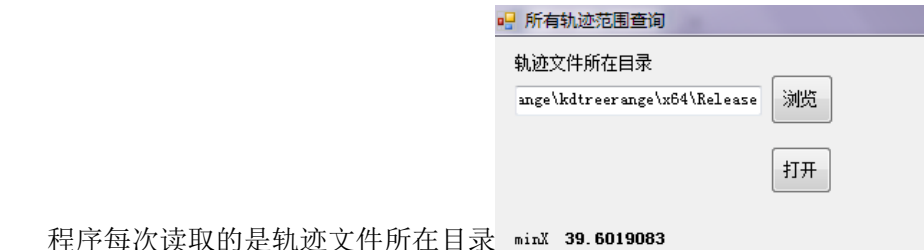
k邻近问题是个经典问题，貌似在机器学习、图像处理特征匹配上都有用，以后实验室可能会用到，就实现了一下。

7. 所有轨迹范围查询

7.1 解决的问题

承上，这里解决的是一个半径区域内有多少个点的问题。

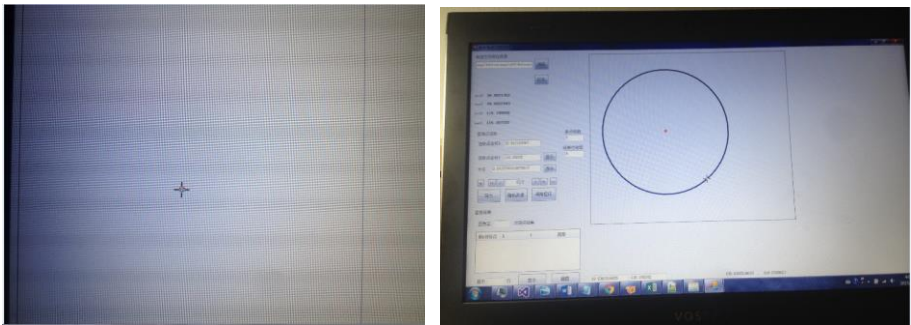
7.2 打开文件



G:\pj\kdtreerange\kdtreerange\x64\Release

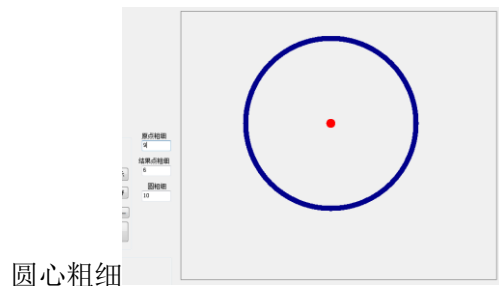
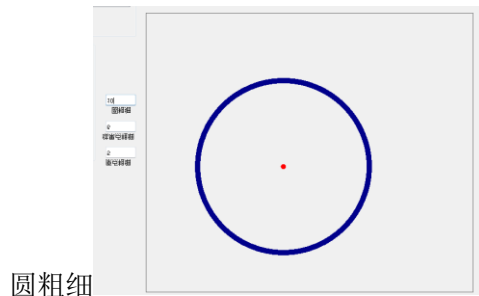
7.3 选半径

在框中单击一个点不放后移动鼠标就能框定一个圆



或者在 设置，
可以立即看到半径、圆心变化。
单击“保存”按钮后保存该组询问。

7.4 粗细



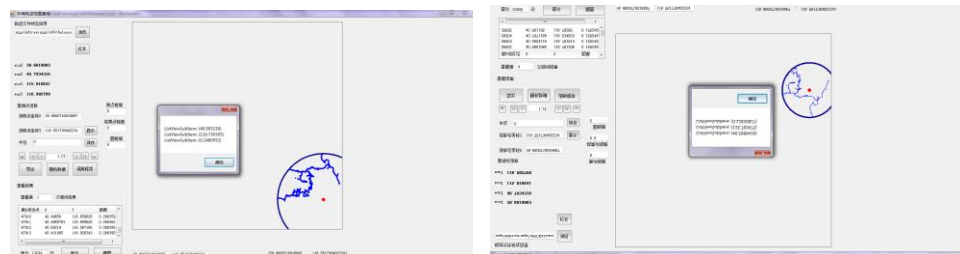
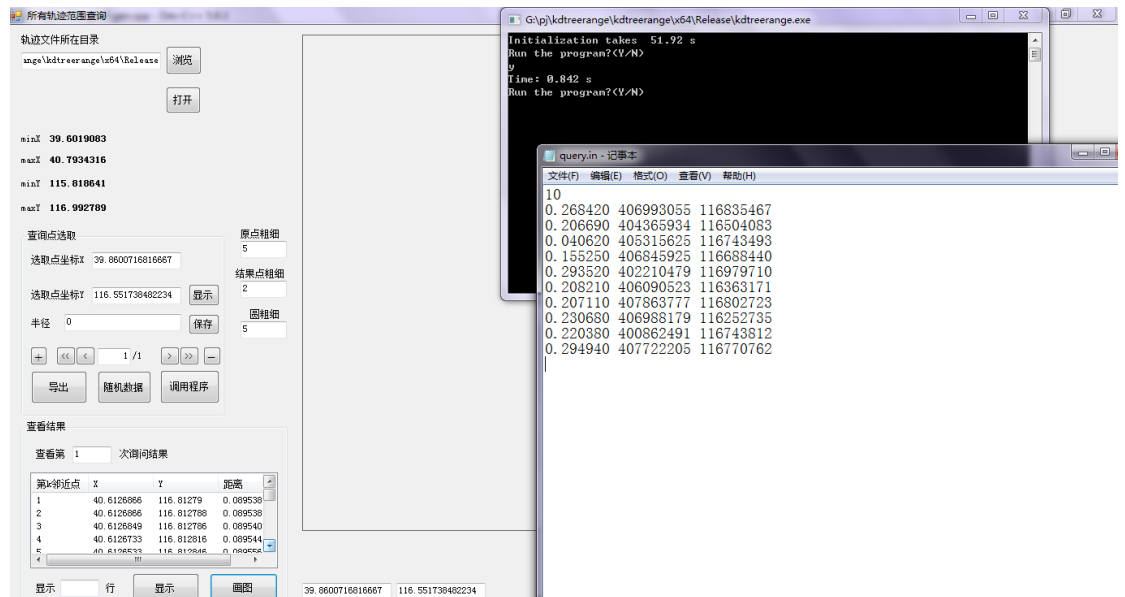
设置合适的粗细:



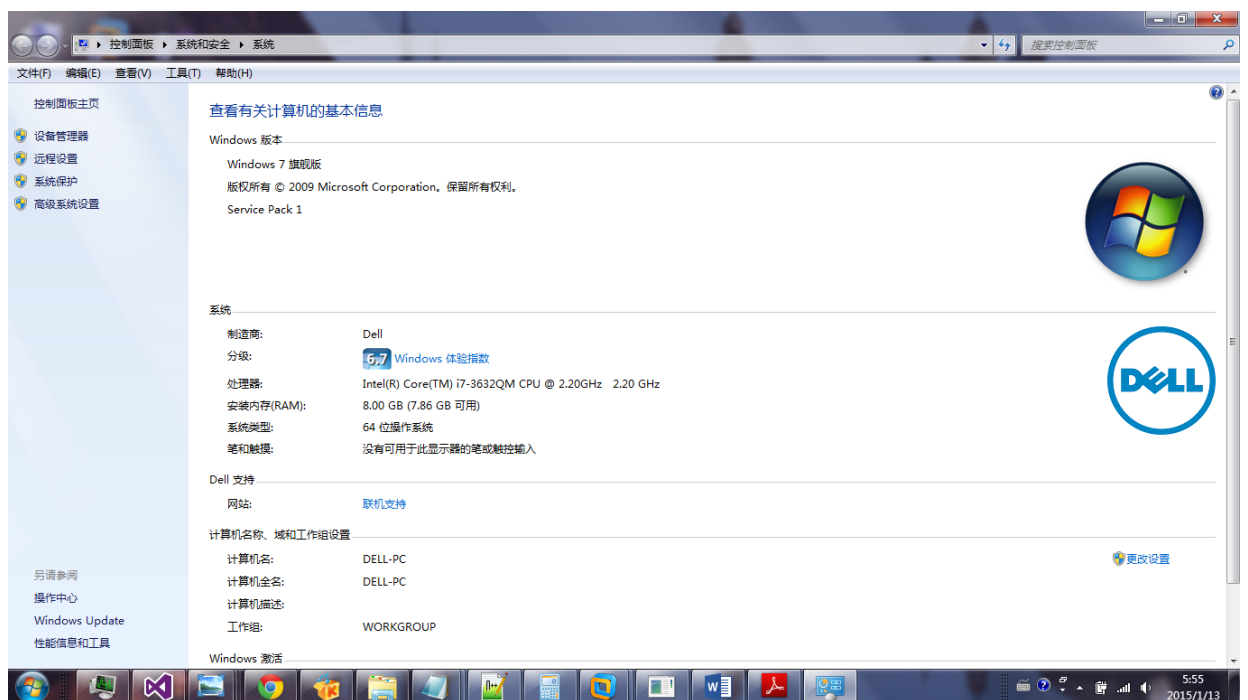


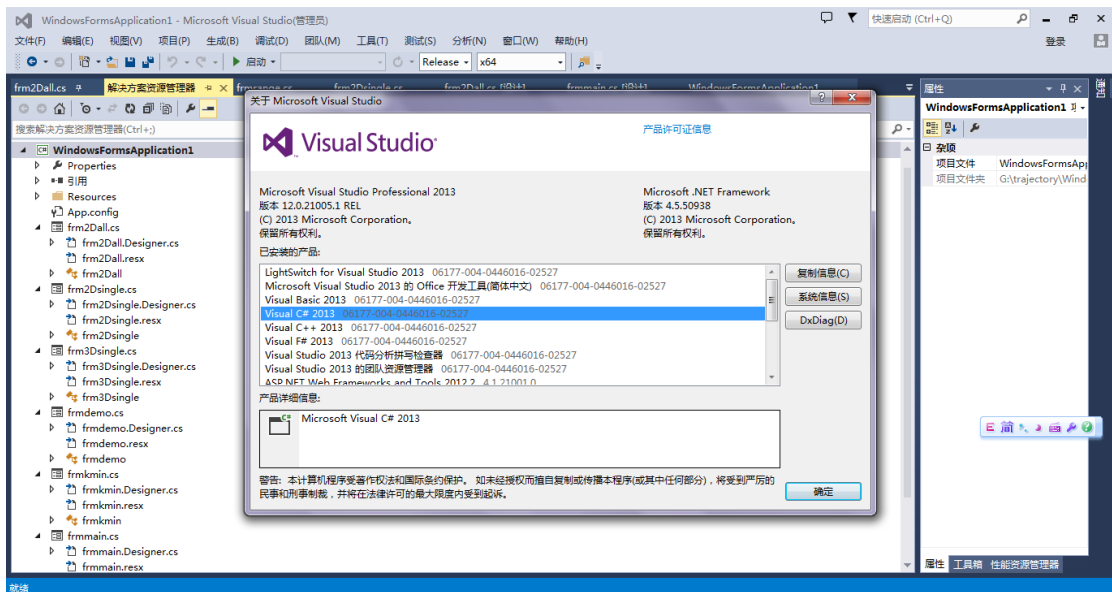
7.5 测试

输入数据是所有轨迹 10组询问



8. 实验环境



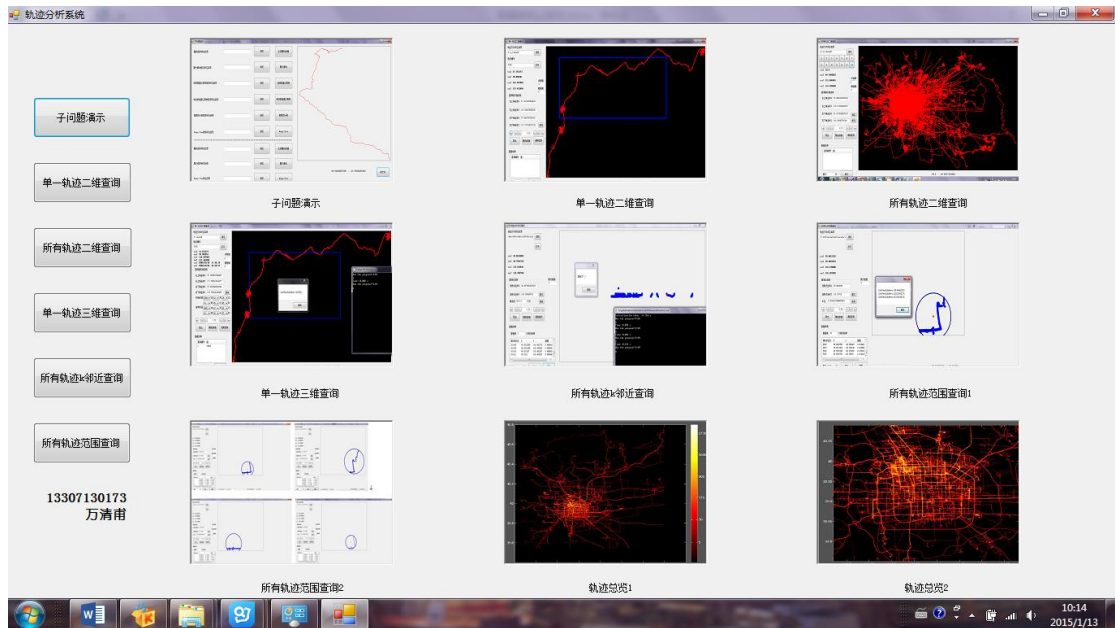


| | | | |
|--------------------------------------|-----------------|------------------|--------|
| setup.exe | 2015/1/13 10:12 | 应用程序 | 422 KB |
| WindowsFormsApplication1.application | 2015/1/13 10:12 | ClickOnce 应用程... | 6 KB |

setup.exe即为安装程序












安装后即可运行



9. 文件夹总览

| | | |
|--|-----------------|-----|
| 2Dsingle | 2015/1/12 21:13 | 文件夹 |
| 3Dsingle | 2015/1/12 23:38 | 文件夹 |
| data2D | 2015/1/13 7:09 | 文件夹 |
| data2Dpoint | 2015/1/13 3:13 | 文件夹 |
| data3D | 2015/1/13 7:10 | 文件夹 |
| kdtree | 2015/1/13 7:34 | 文件夹 |
| kdtreebf | 2015/1/13 6:35 | 文件夹 |
| kdtreekmin | 2015/1/13 3:12 | 文件夹 |
| kdtreerange | 2015/1/13 3:57 | 文件夹 |
| kdtreerangebf | 2015/1/13 6:35 | 文件夹 |
| prerangetreall | 2015/1/13 6:38 | 文件夹 |
| pretraj | 2015/1/13 6:52 | 文件夹 |
| Range Tree with Fractional Cascading(2D) | 2015/1/12 18:35 | 文件夹 |
| Range Tree with Fractional Cascading(3D) | 2015/1/12 19:00 | 文件夹 |
| trajectory | 2015/1/13 7:40 | 文件夹 |
| 三维查询暴力 | 2015/1/12 19:00 | 文件夹 |
| 线段树 (树状数组) 套红黑树 | 2015/1/12 17:03 | 文件夹 |
| 象限四分树1 | 2015/1/12 17:21 | 文件夹 |
| 象限四分树2 | 2015/1/12 17:18 | 文件夹 |

10. 主要程序总览

| 名称 | 修改日期 | 类型 | 大小 |
|---|------------------|-----------------------|-------|
| C# frm3Dsingle.Designer.cs | 2015/1/13 0:10 | Visual C# Source file | 56 KB |
| C# frm2Dall.Designer.cs | 2015/1/13 1:50 | Visual C# Source file | 39 KB |
| C# frmrange.Designer.cs | 2015/1/13 7:15 | Visual C# Source file | 35 KB |
| C# frmkmin.Designer.cs | 2015/1/13 3:36 | Visual C# Source file | 33 KB |
| C# frm3Dsingle.cs | 2015/1/13 0:27 | Visual C# Source file | 32 KB |
| C# frm2Dsingle.Designer.cs | 2015/1/13 0:08 | Visual C# Source file | 31 KB |
| C# frm2Dall.cs | 2015/1/13 6:16 | Visual C# Source file | 26 KB |
| C# frmrange.cs | 2015/1/13 7:29 | Visual C# Source file | 26 KB |
| C# frmdemo.Designer.cs | 2015/1/9 2:48 | Visual C# Source file | 25 KB |
| C# frm2Dsingle.cs | 2015/1/13 0:11 | Visual C# Source file | 24 KB |
| C# frmkmin.cs | 2015/1/13 4:14 | Visual C# Source file | 20 KB |
| C# frmmain.Designer.cs | 2015/1/13 5:23 | Visual C# Source file | 19 KB |
| C# frmdemo.cs | 2015/1/9 14:15 | Visual C# Source file | 17 KB |
|  3Dsingle.cpp | 2015/1/10 4:34 | C++ Source File | 12 KB |
|  prerangetreeall.cpp | 2015/1/11 21:09 | C++ Source File | 9 KB |
|  sts.cpp | 2014/12/26 17:47 | C++ Source File | 8 KB |
|  sts2.cpp | 2014/12/23 0:38 | C++ Source File | 7 KB |
|  preequal.cpp | 2014/12/25 0:19 | C++ Source File | 7 KB |
|  2Dsingle.cpp | 2015/1/9 16:26 | C++ Source File | 7 KB |
|  pre2.cpp | 2014/12/25 2:18 | C++ Source File | 6 KB |
|  kdtreerange.cpp | 2015/1/11 20:46 | C++ Source File | 6 KB |
|  kdtreekmin.cpp | 2015/1/11 4:29 | C++ Source File | 6 KB |
| C# frmmain.cs | 2015/1/13 5:28 | Visual C# Source file | 4 KB |
| C# Program.cs | 2015/1/9 12:49 | Visual C# Source file | 1 KB |

| 序号 | 算法 | cpp所在目录 | 语言 | 行数 |
|----|-------------------|---|-----|-----|
| 1 | 暴力（二维） | G:\pj\线段树（树状数组）套红黑树\pre\bf.cpp | C++ | 27 |
| 2 | 线段树套红黑树 （二维） | G:\pj\线段树（树状数组）套红黑树 \pre\sts\sts\sts.cpp | C++ | 329 |
| 3 | 树状数组套红黑树 （二维） | G:\pj\线段树（树状数组）套红黑树 \pre\sts\sts\sts2.cpp | C++ | 293 |
| 4 | PR四分树 （二维） | G:\pj\象限四分树 2\preequal\preequal\preequal\preequal.cpp | C++ | 241 |
| 5 | 普通四分树 （二维） | G:\pj\象限四分树1\pre2\pre2\pre2.cpp | C++ | 194 |
| 6 | 带分层折叠的区域树 （二维） | G:\pj\Range Tree with Fractional Cascading(2D)\prerangetree\prerangetree\prerangetree\prerangetree.cpp | C++ | 250 |
| 7 | 暴力（三维） | G:\pj\三维查询暴力\pre3D\bf.cpp | C++ | 28 |
| 8 | 带分层折叠的区域树 （三维） | G:\pj\Range Tree with Fractional Cascading(3D)\prerangetree3D\prerangetree3D\prerangetree3D\prerangetree3D.cpp | C++ | 378 |
| 9 | kd-tree求k邻近点 | G:\pj\kdtreemin\kdtreemin\kdtreemin\kdtreemin .cpp | C++ | 221 |
| 10 | kd-tree求半径范围内的点 | G:\pj\kdtreerange\kdtreerange\kdtreerange\kdtreerange.cpp | C++ | 231 |
| 11 | 主界面 | G:\pj\trajectory\WindowsFormsApplication1\WindowsFormsApplication1\frmmain.cs | C# | 108 |
| 12 | 子问题演示 | ...\frmdemo.cs | C# | 459 |
| 13 | 单一轨迹二维查询 | ...\frm2Dsingle.cs | C# | 601 |
| 14 | 所有轨迹二维查询 | ...\frm2Dall.cs | C# | 700 |
| 15 | 单一轨迹三维查询 | ...\frm3Dsingle.cs | C# | 722 |
| 16 | 所有轨迹k邻近查询 | ...\frmkmin.cs | C# | 497 |
| 17 | 所有轨迹范围查询 | ...\frmrange.cs | C# | 617 |

11. 优点

界面比较容易操作，可以选定多个搜索矩形框当作若干组询问，并查看任一个矩形。列表框输出查询结果，可以不滑动滚动条显示任意一次询问的结果。

本pj使用的数据结构算法效率均比较高。

12. 存在问题及未来工作

12.1 单一轨迹二维（三维）查询只查询了有多少点，没有输出具体点的坐标并画在屏幕上，其实要输出所有点只有在现有程序上稍作改动即可，但是考虑到输出只能一个个输出而且屏幕上已经画了轨迹上所有点，所以就只输出了有多少个点在矩形内。

12.2 对于所有轨迹的二维查询，因为画线函数效率太低，选定搜索矩形框后原本轨迹上的点就被我强制清除了，不太利于操作。

12.3 由于三维Range Tree空间复杂度是 $n(\log_2 n)^2$ 开销特别大（二维查询已经用到超过8G内存 开始用虚拟内存了） 所以只对单一轨迹进行了三维查询 听一个同学说他用了可持久化平衡树可以在一定时间范围内查询任意多边形 在三维查询上Range Tree时间复杂度是 $m(\log_2 n)^2$ m 为询问数 空间换时间 貌似有同学用Grid和四分树搞 不知道效率上和Range Tree比起来如何，有空可以看一下R-tree（包括R+和R*两个变种）和Grid这两种数据结构。

12.4 画图函数用的c#自带的最基本的画线画椭圆画矩形函数，画所有轨迹时一坨点聚集在一块，画圆和线的时候很容易看到不平滑的边缘。如上所说，可以考虑像地图一样任意缩放、移动显示框。而且DrawLines函数效率比较低，经常出bug，可以考虑保存成bitmap画图，或是使用openGL,openCV等等。

12.5 程序本身存在很多bug

VS自身会提示会出现的异常 在测试过程中也出了很多bug加了一部分try catch语句 但是由于代码太长 懒得一个一个加try catch了 基本上是不会出现大问题的。

```

public void repaint1(Color col)
{
    double t1 = Convert.ToDouble(xx1[nowm] - minx) / Convert.ToDouble(maxx - minx);
    t1 = t1 * (this.pic1.Width - 20);
    double t2 = Convert.ToDouble(yy1[nowm] - miny) / Convert.ToDouble(maxy - miny);
    t2 = t2 * (this.pic1.Height - 20);
    double t3 = Convert.ToDouble(xx2[nowm] - minx) / Convert.ToDouble(maxx - minx);
    t3 = t3 * (this.pic1.Width - 20);
    double t4 = Convert.ToDouble(yy2[nowm] - miny) / Convert.ToDouble(maxy - miny);
    t4 = t4 * (this.pic1.Height - 20);
    if (col == Color.Blue) g.DrawRectangle(new Pen(col, (float)pointwidth2), (float)(t1 + 10), (float)(t2 + 10), (f
    else g.DrawRectangle(new
}
public void repaint2(Color co
{
    try
    {
        double t1 = Convert.ToDouble(double.Parse(txt1.Text) * 1000000.0 - minx) / Convert.ToDouble(maxx - minx);
        t1 = t1 * (this.pic1.Width - 20);
    }
    catch (System.ArgumentNullException)
    {
        // 异常: System.ArgumentNullException
    }
}

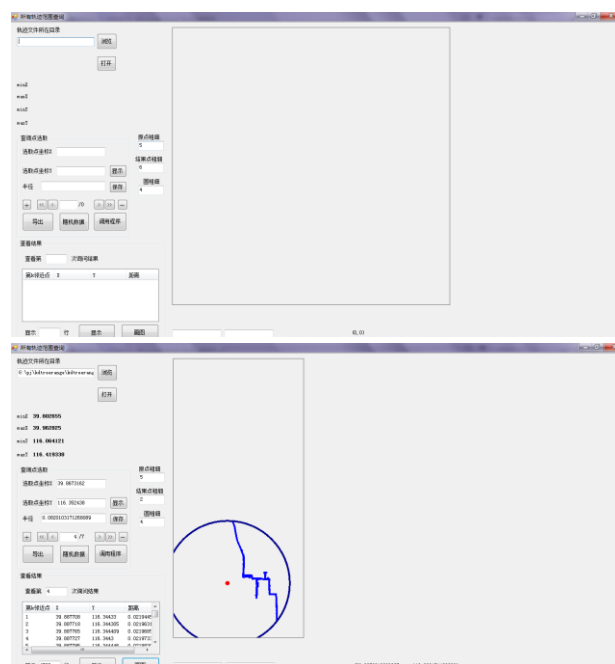
```

void Graphics.DrawRectangle(Pen pen, float x, float y, float width, float height) (+ 2 重载)
 绘制由坐标对、宽度和高度指定的矩形。

12.6 列表框查看结果基本上当有20000行的时候就要等几秒,不知道是不是ListView
 控件本身的问题,有的时候不要查询具体有哪些点在距离(x,y)一个半径范围
 内,只要知道有多少点。

12.7 显示框大小是根据输入轨迹文件的X坐标最大值maxX-最小值minX
 和Y坐标最大值maxY-最小值minY之间的比值设置的。

因为屏幕上水平方向和竖直方向同等距离对应的数据中X坐标差值和Y坐标差
 值相等,才能保证在屏幕上框一个圆形,这个圆形内的点对应数据中的点距离
 圆心的确在一个半径范围内,否则就会出现比例尺比例不对的问题。所以每次
 按“打开”后显示框都会自动调整宽度和高度,但是这样会导致大小不统一。
 如下图所示。



12.8 写程序的时候比较任性,UI代码比较冗长,很多可以写到一个函数(比如屏幕
 上坐标和轨迹数据坐标之间的一一对应关系)的代码由于写的时候懒就直接复

制粘贴了，导致UI代码量非常大。

12.9 VS的debug模式和release模式有时候运行结果不一样，不明原因。

12.10 有空可以尝试比较一下kd-tree, R-tree, Grid, Quad, Tree Range Tree等数据结构。

13. 总结

13.1 12.22-12.27写完了线段树套红黑树、树状数组套红黑树、PR四分树、普通四分树、带分层折叠的区域树

1.9-1.11写完了kd-tree及所有UI

1.12-1.13做了些测试和改进

13.2 高中竞赛的时候写过线段树套平衡树（treap），前段时间刚写过扩展作业红黑树，于是自然而然想到线段树套红黑树，以前一直觉得树状数组套平衡树会更快一些，经过试验发现并不是这样。

13.3 后来我觉得算法还不够快 是log的平方级别，听同学大体讲了下他R-tree的思路觉得不适合这道题。

13.4 尝试了一下四分树，发现是个大坑，同学也说是这样，效率非常低。（也许是我写的有问题？网上说可以控制树的搜索深度然后再暴力搜索觉得不太靠谱，有点讨巧的意味）

13.5 无意中看到网上有篇讲范围搜索（Range Search）的博客，

前一阵把搜索引擎的RangeQuery的逻辑重新写了一遍，我写的时候就感觉很不对劲，我们的搜索引擎采用的是一种非常怪异的实现，至少我没在别的搜索引擎里见过，或是在资料中看到过。我要解决的是二维坐标查询，比如你想知道你周围五公里内的医院在什么地方，暴力解决方法就是把所有医院坐标得到，把x坐标循环过滤一遍，再把y坐标循环过滤一遍。其实这还好，因为一个城市一共也没多少医院，但如果调用方把坐标查询写前面，也就是先过滤x和y坐标，再过滤医院，那就悲剧了。

简单点的办法就是把x和y坐标有序地保存，然后用二分查找定位到x-2.5km, x+2.5km, y-2.5km, y+2.5km，然后取x-2.5km到x+2.5km的posting list和y-2.5km到y+2.5km的posting list做and操作就可以了。

但是还能不能再快呢？这个问题我想了想也没什么头绪，偶然发现了RTree这个数据结构，我感觉这才是正道。

下面是Range Searching的翻译。原文地址：<http://www.cs.ucsb.edu/~suri/cs235/RangeSearching.pdf>

发现其实就是线段树套一个线段树（排序数组） 不加优化的Range Tree效率是和线段树套红黑树基本一样的 作者讲了一个预处理加指针的优化，其实不是什么神奇的东西。

试验了一下发现效率还是非常高的。

而且这种方法可以轻松扩展到高维，写起来很容易。时间复杂度非常低，唯一的缺点是耗空间。关于其动态修改、删除操作我没有研究过，感觉上不是太支持。

- 13.6** 这次pj极大地锻炼了代码能力。虽然期中之前用java写了一个几千行的UI，但是那个UI没有什么算法或者数据结构，而且复制粘贴了很多。搞竞赛时候数据结构题代码顶多也就100-300多行，而且是一个个孤立的控制台程序，一个个黑框框，这是第一次用界面展示数据结构。由于C++有STL库，release模式开O2优化开关后程序加快很多，而且C++用的比较熟练，数据结构的核心代码选择了C++。但是因为不会MFC建工程，之前在mac虚拟机下用netbeans写JAVA的UI不是太顺手，最后决定在windows下用VS写c#。之前用c#写过一点图像处理的UI，感觉还是很方便的，基本语法都是互通的，缺点就是代码没有c++简洁。
- 13.7** 统计了一下核心部分数据结构的代码一共2192行，UI的代码一共3704行（不包括c#自动生成的Designer.cs），总共5896行。写UI没什么技术含量，基本不用动脑子，但是特别麻烦，经常牵一发而触全身，遇到过各种奇奇怪怪的bug，写完了以后比较有成就感，相比一直写的控制台程序直观了很多。还有一点很深的感悟就是写数据结构的时候，一个清晰的逻辑很重要，算法框架大体了解了之后可以事先在纸上模拟运行一下，对于搞不清楚的边界数据仔细想想怎么处理，基本上把能想到的细节怎么实现都想好以后再去写代码。与其花时间调试不如花时间想清楚后再写，这次pj数据结构部分基本上80%的时间都是在写，调试时间并不多，和以前一直调不出情绪急躁相比进步了很多，整个过程心态比较平和。写完高效算法之后我用随机程序生成了若干组数据和暴力程序对拍，保证了所有数据结构代码都是完全正确的。
- 13.8** 有待改进的地方是，因为界面的代码改了以后就能及时看到改变，写UI的时候比较随意，想到哪写到哪，复制粘贴了很多代码，这些代码本可以写成若干个函数以供反复调用。
- 13.9** 写代码的效率有待提高，代码还不够简洁。

总而言之，写完pj收获颇丰。

13级计算机科学与技术专业

万清甫

13307130173

2015年1月13日星期二