基于梯形图的点定位

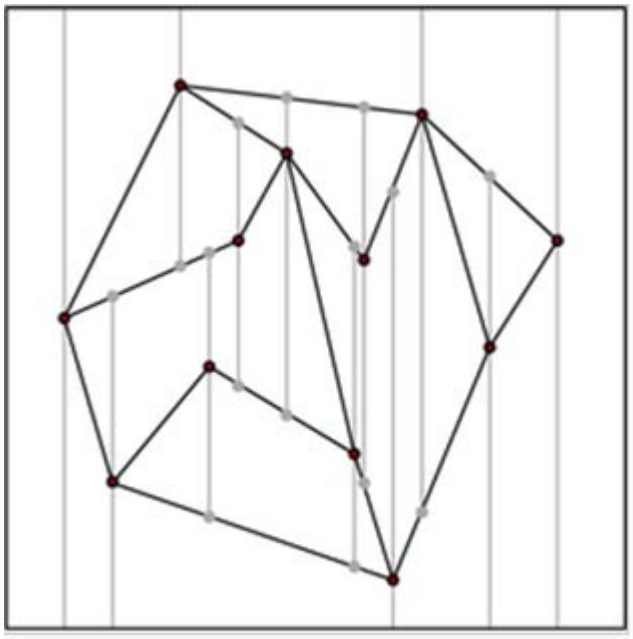
背景介绍

在很多应用中，都需要使用点定位的概念，例如在航行中，需要知道船只所在区域的水文情况 ，船只可与通过 gps 得到其位置坐标，然后根据点定位的方法与给定的位置坐标，就可以通过计算机计算得到所在区域。

梯形图的概念

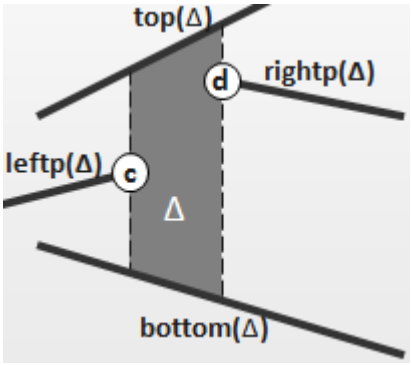
梯形图是对平面区域的一个划分，首先用一个包围盒将所有不相交的线段包围起来，然后在每一个顶点处，发出向上向下的两条射线，这些射线或与包围盒相交，或与线段相交，从而形成对平面的一个划分，每个子平面都是一个梯形或者三角形（可以认为退化的梯形）。

如图下所示：



A trapezoidal decomposition.

每一个梯形图单元如下图所示，由两条边以及两个端点组成。



在算法中我们对梯形图以及相关几何结构的定义如下：

线段：

```
struct Segment
{
    DataPoint startPoint;
    int startPointPos;
    DataPoint endPoint;
    int endPointPos;
};
```

梯形图的半边

```
struct TrapeziaEdge
{
    double x1;
    double y1;
    double x2;
    double y2;
};
```

在整个算法中关键是维护梯形图的搜索结构，其中搜索结构如下

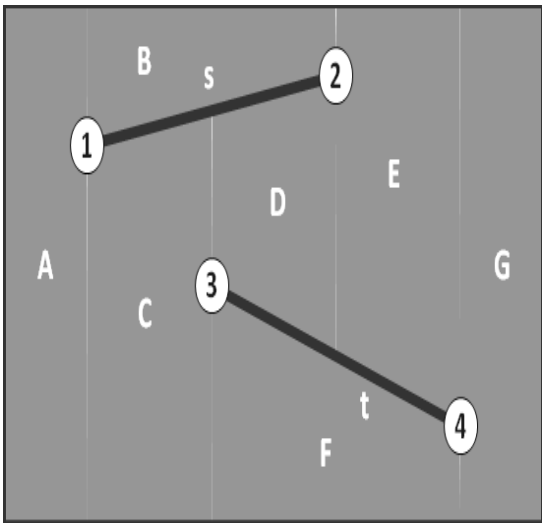
搜索结构是一颗树，分为叶子节点和内部节点，内部节点又分为端点节点和边节点，在程序中定义如下

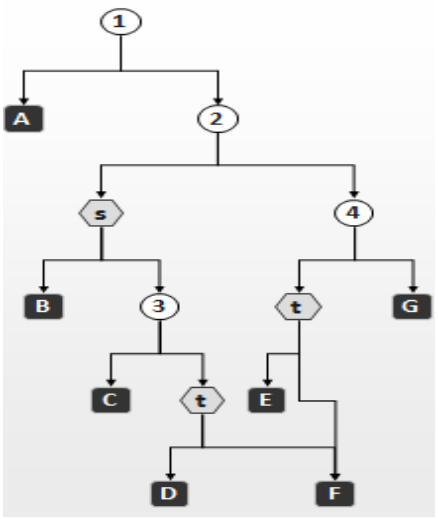
```
struct Node
{
    int type;                  //0...x_node point,   1...y_node segment,  2...leaf trapezia
    int pointsPos;
    int segmentsPos;
    int trapeziasPos;
    Node *leftchild;
    Node *rightchild;
    Node() {type=-1; pointsPos=-1; segmentsPos=-1; trapeziasPos=-1; leftchild=NULL;
rightchild=NULL;}
};
```

其中 0 表示端点节点，1 表示线段节点，2 表示梯形

叶子节点表示一个梯形，端点节点(约定为 x 节点)存储的是线段的端点，边节点(约定为 y 节点)存储的是一条边，在算法中我们使用了一个半边结构来表示一个梯形图。

我们以下图为例说明梯形图中树的结构





算法原理

我们使用了随机增量式的算法。该算法通过维护梯形图来实现。下面我们将分别梯形图算法的一些细节。我们假定

首先介绍梯形图的查找算法，对于端点节点我们需要判断的是位于端点的左边还是右边，而对于线段节点则要判断位于线段的上方还是下方，当访问到叶子节点时，该叶子节点所存贮的梯形就是所要查询的点所在的梯形。算法伪代码如下：

```
1.    p 为待查询的节点
2.    Node find(p, node*)
3.     {   if node.type=叶子节点
4.           return node
5.        else if node.type=端点节点
6.                 If   p.x <node.x
7.                      return find(p,node.left)
8.                 else p.x>=node.x
9.                      return find(p,node.right)
10.       else if   node.type=边节点
11.                if node 位于线段的右边
12.                     return find(p,node.right)
13.                else
14.                     return find(p,node.left)
15.    }
```

我们先给出真个算法的伪码，然后再来介绍各个算法的子步骤。

算法 TRAPEZIODALMAP(S)

输入：一组共 n 条互不相交的线段

输出：梯形图 T(S),以及与之对应的，限制于一个包围框之内的查找结构 D

1. 构造一个包围框 R，其大小必须足以容纳 S 中的所有线段
   根据 R，初始化相应的梯形图结构 T 以及查找结构 D
2. 将 S 中的所有线段随意打乱，得到一个随机序列：$s_1, s_2, \ldots, s_n$
3. for i <-1 to n

4 do 找到与s_i相交的所有梯形$\Delta_0,\Delta_1,\ldots,\Delta_k$从 T 中删去

5 将$\Delta_0,\Delta_1,\ldots,\Delta_k$从 T 中删去

将它们替换为由s_i的引入而新生出来的若干梯形

6 将与$\Delta_0,\Delta_1,\ldots,\Delta_k$对应的叶子从 D 中删去

对应与每个新生成的梯形，生成一匹新的叶子

将新生成出的叶子与已有的内部节点相连接

我们通过以下算法来求得与s_i相交的所有梯形

算法 FOLLOWSEGMENT(T,D, s_i)

输入： 梯形图 T，与 T 相对应的查找结构 D，以及新引入的一条线段s_i

输出： 由所有与s_i相交的梯形组成的一个序列 $\Delta_0,\Delta_1,\ldots,\Delta_k$

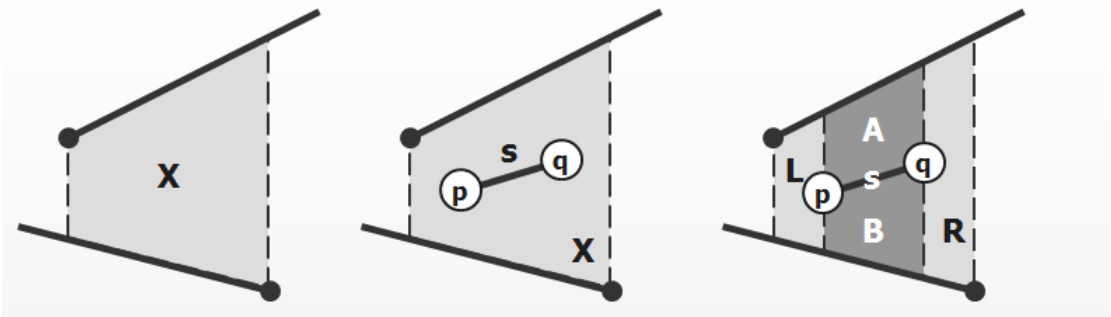
1. 分别令 p 和 q 为s_i的左、右端点
2. 在查找结构 D 中对 P 进行查找，最终找到梯形Δ_0
3. j <-0
4. while(q 位于 rightp(Δ_j)的右侧)
5. do if (rightp(Δ_j)位于s_i的上方)
6. then 令Δ_{j+1}为Δ_j的右下方邻居
7. else 令Δ_{j+1}为Δ_j的右上方邻居
8. j<-j+1
9. return ($\Delta_0,\Delta_1,\ldots,\Delta_j$)

对于给定的一条线段，根据上面的算法，当端点落入梯形图内的情况，梯形图内的线段可以分为以下三种种情况进行讨论。

我们是这样规定的，每进来一条线段，我们先把 x 值小的点作为起点，x 值大的点作为终点。我们先判断起点所在梯形和终点所在梯形，如果这两个梯形的编号相等则当作情况一来处理，落在不同梯形中则按照情况二来处理。这里我们处理了特殊情况，如果起点在垂线上判断为在右边，如果终点在垂线上则判断为在左边。如果起点与已有线段起点重合，我们则在所加的那条线段上取离起点很少位移的一点，然后搜索它所在的梯形为线段起点所在的梯形。

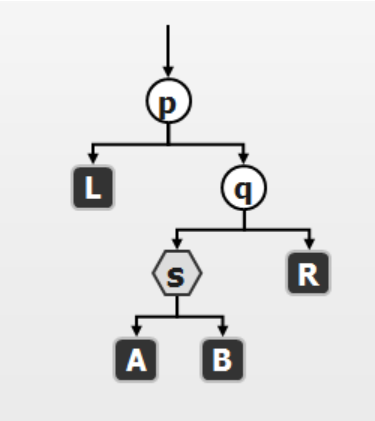
情况 1:线段落在查询区域内有两个端点

如下图所示，这时 x 被划分为四个新的梯形:



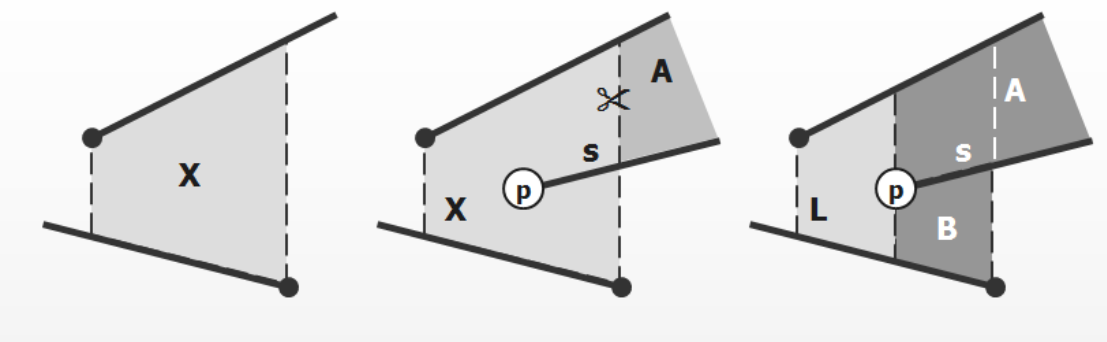
首先找出在 p 点和 q 点正上方和正下方的半边，进行裂边，如果此边不在边界上还要修改对边以及相邻梯形的边结构。我们这里处理了存在于 p，q 的 x 相等的端点的情况，也处理了 p 和 q 在边界和端点上的情况。这些情况下面也加以考虑了。

我们将所在梯形节点换为一个表示四个子梯形的子树，其子树的结构如下



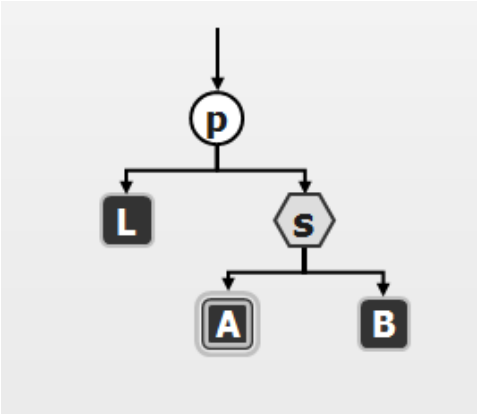
当发生这种情况时，我们对 x 进行替换。

情况 2：线段落在区域内一个节点

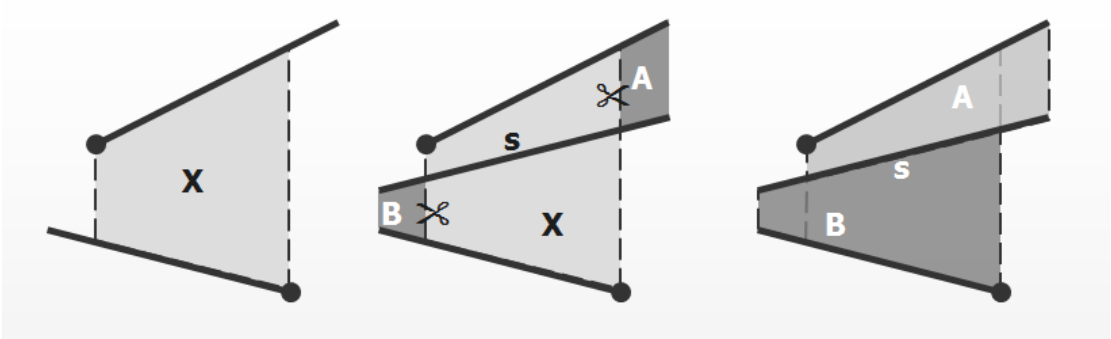


这种情况要合并梯形，我们是这样确定下次应该合并哪个梯形的。如果线段节点在线段下面（也就是线段右边），则下次和合并线段上面的梯形，我们把上面梯形节点指针记录下来，并且把两个公共端点记录下来。

构造的子树的结构如下

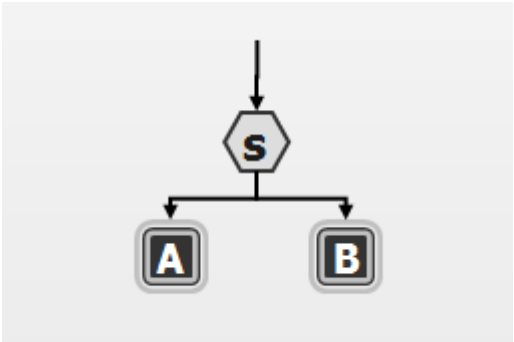


情况 3：没有端点落在区域内



根据上次确定的合并方向和指针还有两个节点，我们合并梯形。并且记录下来右边的线段端点是在上面还是下面以确定下次合并的方向。合并后两个孩子指针指向同一个节点。线段经过了多少个梯形在树形结构中就出现多少个这个线段的节点。

其子树的结构如下



当s_i完全落入一个梯形Δ当中时，我们将Δ划分为四个新的梯形 L,A,B,R.我们可以在常数时间内确定四个新的梯形的位置，并对原有的树做相应的修改。
若s_i与两个甚至多个梯形相交时，首先从s_i的两个端点发出垂直延长线，将Δ_0，Δ_k一分为三。然后，要将当前与s_i相交的各条垂直延长线截短。借助存储在$\Delta_0,\Delta_1,\ldots,\Delta_k$中的信息，可在正比于相交梯形数目的时间内，完成这一步操作。

为更新树，必须删除对应于$\Delta_0,\Delta_1,\ldots,\Delta_k$的叶子，然后为每个新的梯形分别生成一匹叶子，最后还要引入若干新的内部节点。若点落在Δ_0内部，则用一个对应于s_i左端点的 x 节点，以及另一个对应于s_i本身的 y 节点，来替换原先对应于Δ_0的那匹叶子。类似得，若s_i的右端点落在Δ_k的内部，则用一个对应于s_i右端点的 x 节点，以及一个对应于线段s_i本身的 y 节点，来替换原先对应于Δ_k的那匹叶子。最后，与$\Delta_1,\Delta_2,\ldots,\Delta_{k-1}$相对应的所有叶子，都将被替换为同一个 y 节点，该节点对应于s_i。新生成的每个内部节点的出弧，都需要正确地指向对应的新叶子。

随机线段的生成

我们的线段的随机生成算法增量式的算法，即不断的随机的生成一个线段，然后检查改线段是否与已知的线段集相交，若相交，则取与已知线段相交的最近的点的偏移为新的线段的端点。

伪代码如下：

i <- 0

Q <- Ø
while(i<n)
do 随机的生成两点并连接成线段
对 i 个已知的线段求交
if 对这 i 个线段没有交点
将改线段插入 Q
i<i+1
else 取所有线段和原有线段的交集并对线段两端点进行缩进偏移

复杂度分析

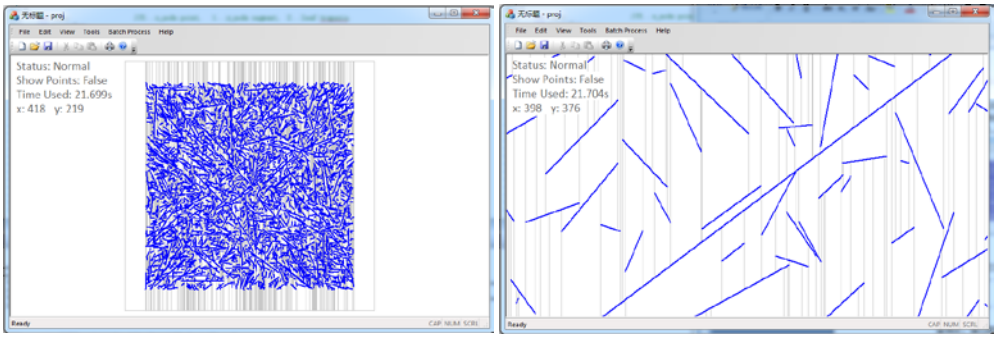
建树所需要花费的时间为O(nlogn)
所花费的空间大小为 O(n)
点定位的期望时间为 O(nlogn)

程序功能使用说明：

程序运行模式

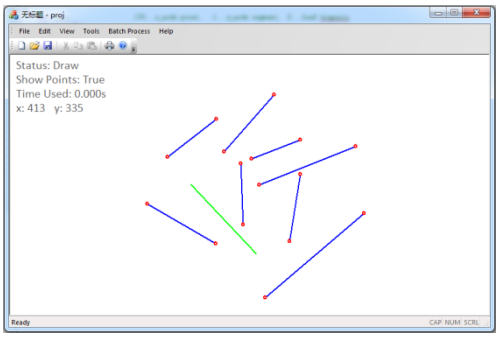
1. NORMAL

正常运行状态可以进行图像的拖动，缩放操作，按住 shift 用坐标左键拖动可以放大缩小图像，按住 ctrl 用鼠标左键进行拖动可以平移整个图像，这可以在屏幕中点和线段过多时更好的看清楚图像细节。如下图：



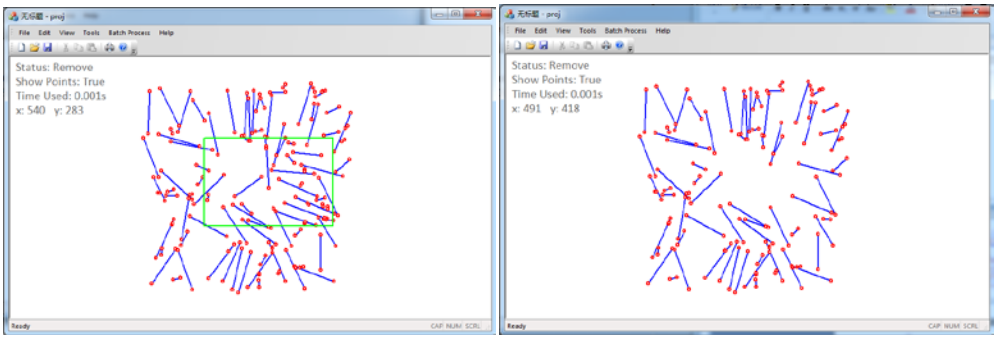
2. DRAW

手工输入不相交的线段，通过鼠标左键画线即可，如果输入的线段相交，则画出的线段不会被加入。



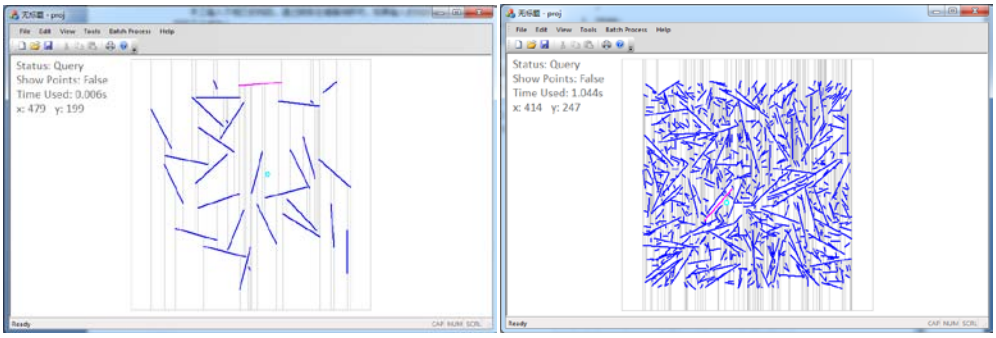
3. REMOVE

删除输入线段，使用鼠标左键画一个矩形，在矩形内部的线段都将被删除。如下图所示：



4. QUERY

查询点，通过鼠标左键在屏幕上点。查询后该店上面的点用不同颜色标出。如下图所示：



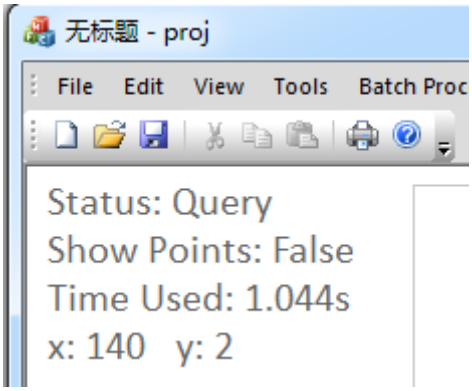
界面左上角显示了当前程序运行的状态：

Status：表示当前状态；

Show Points: FALSE 表示不显示线段的端点，TRUE 表示显示线段的端点；

Time Used:表示当前操作所用的时间，包括生成随机数的时间和构造梯形图的时间；

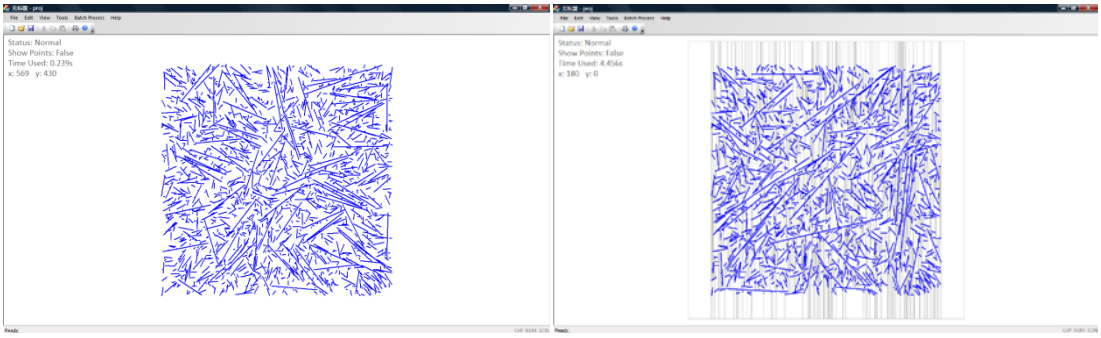
X: Y: 表示的是 X 和 Y 的坐标。



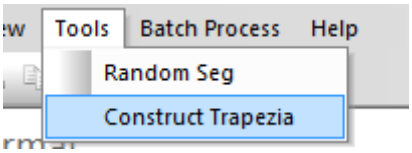
程序功能

1. 随机线段的生成

随机线段生成算法有两个参数，第一个控制生成线段的条数，第二个控制最长线段的长度，在（0，1）区间里，随机生成 2000 个最大长度为 0.2 的点后结果以及构造的梯形图如下图：



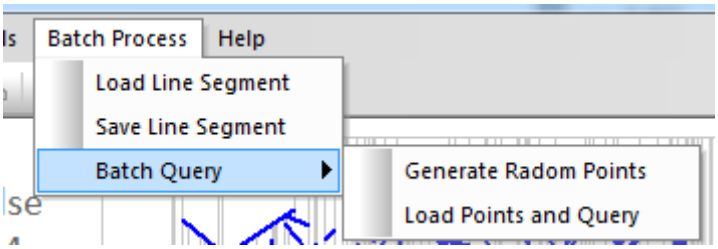
2. 构造梯形图



点击 Tools->Construct Trapezia 即可，程序会自动清除上一次构造的梯形图，根据现有线段集重新构造。

3. 批处理功能

工具栏中批处理菜单下包含了批处理的功能。



Load Line Segment 为读取线段集,Save Line Segment 为保存当前的线段集。

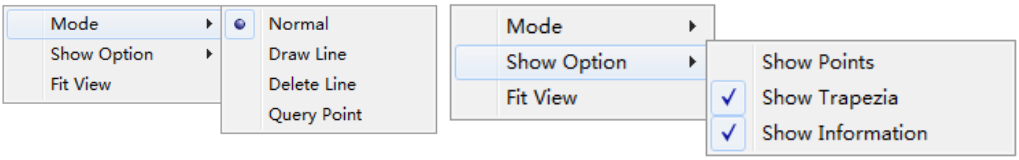
Batch Query->Generate Radom Points 表示随机生成查询点，可以选择生成数量，然后保存到文件。

Batch Query->Load Points and Query 通过已经生成好的梯形图进行查询，然后将查询结果保存到文件。

查询所用的时间都将在左上角的信息栏中显示出来。

界面设定

点击右键后可以设定界面的显示以及程序的状态。



在 Mode 模式下有以下几个状态，分别为 Normal, Draw Line, Delete Line ,Query Point.可以进行程序运行状态的转换。

Show option 下有以下几个选项分别为 Show Points, Show Trapezia 和 Show Information.

Show Points 为显示所有线段的端点

Show Trapezia 为显示梯形图

Show Information 为显示左上角的状态信息

Fit View:为对窗口内容进行调整以让其适应窗口的大小。

数据格式说明：

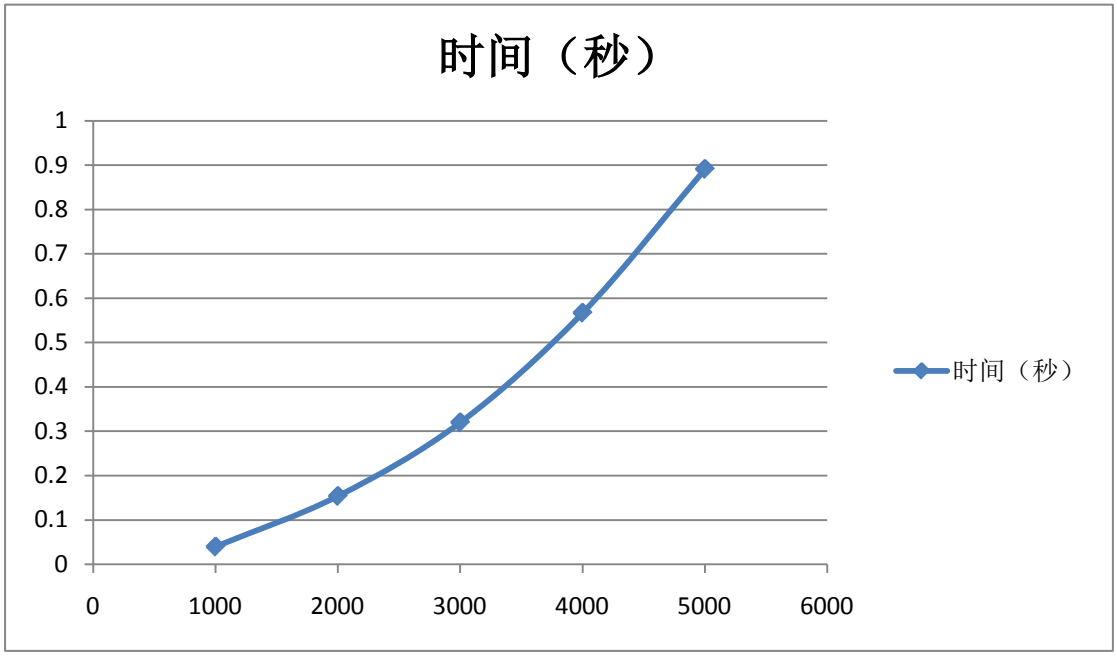
程序中我们对于点和线段集使用了自己定义的文件格式如下：

1. 点集文件。命名为*.pts，文件第一行是一个整数 n，表示点的个数，下面一共有 n 行，每行两个浮点数，表示每个点的 x 和 y 坐标。（下图左）
2. 线段集文件。命名为*.seg，文件第一行是一个整数 n，表示线段的条数，下面一共有 n 行，每行 4 个浮点数，表示每条线段两个端点的 x 和 y 坐标（下图右）

试验结果

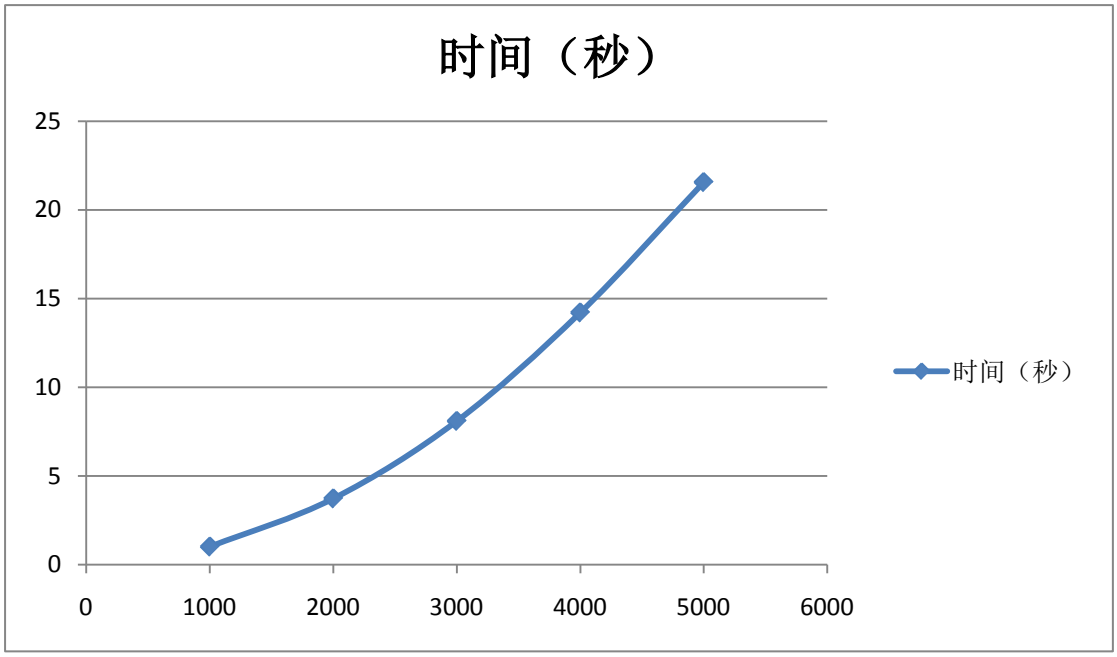
当最大线段长度设为 0.2 时，线段数目从 1000 变到 5000 时生成随机数的时间

	1000	2000	3000	4000	5000
0.2	0.04	0.154	0.32	0.567	0.891



当最大线段长度设为 0.2，线段数目从 1000 变到 5000 时构造梯形图的时间为有

	1000	2000	3000	4000	5000
0.2	1.018	3.732	8.096	14.207	21.552



观察图标可以发现，复杂度以 O(NlogN)的速度增长。