基于 Voronoi 图的最近邻查找算法实现

计算几何课程实验

高　岳　2006310456

孙建树　2006210793

周云涛　2006310427

目录

问题描述

最近邻问题可以这样描述：

给定一个点集 S，对于某个输入的查询点 q，查找 S 中与之最接近的点 p。

最近邻查找的算法有很多，在本次的实验中，我们关注的是基于 voronoi 图的最近邻查找算法。

根据 Voronoi 的特性，我们想出了一个比较简单的最近邻查询算法（并将其命名为“GarbilItree”），并预计该算法的复杂度为 $O(\sqrt{n})$；作为比较，我们另外实现了一种基于 Voronoi 的最近邻查找算法 Monotone Subdivisions ,该算法的搜索复杂度是 $O(\log^2 n)$。在实验中，我们对这两种算法和不使用 Voronoi 的线性查找算法（作为标准算法）在正确性、速度上进行了测试和比较。

算法分析

Voronoi 图构造

Voronoi 图的构造使用 Incremental Construction。数据结构使用 DCEL 来表图。此算法的时间复杂度是 $O(n^2 \log n)$。

Non-Voronoi 算法

这是一种最简单，也是最可靠的最近邻查找算法。其过程是计算集合 S 中每一个点与查询点 q 的距离，然后找出其中距离最短的那个点。这种算法的复杂度是 $O(n)$。我们实现

这个算法的目的是将其作为标准算法（在下面称为 standard 算法），来检验其他两个算法（Garbilltree 和 Monotone Subdivisions）的正确性，以及作性能上的比较。

Garbilltree 算法

算法提出

在 Voronoi 图中，判断某一个点与哪个 site 最接近，只需要看这个点落在哪个 cell 内。很容易想到的一种办法是，从某一个 site 出发，沿着一条与目标点距离逐渐缩短的路径查找，最后就可以找到包含目标点的 cell。

算法流程

在构造完 Voronoi 图之后，对于每一个查询点 q，做如下的操作：

1、随机选一个 site $p_0 = p_i (1 \le i \le n)$

2、计算 p_0 到 q 的距离 D_0，以及 p_0 所在的 cell 的所有 m 个邻居的 site $p_{01}, p_{02}, p_{03}, ..., p_{0m}$ 到 q 的距离 $D_{01}, D_{02}, D_{03}, ..., D_{0m}$；

3、计算 $D_0, D_{01}, D_{02}, D_{03}, ..., D_{0m}$ 中的最小值 $D_{\min}$，若 $D_{\min} = D_0$，则算法结束，p_0 即为到点 q 距离最近的点；

4、设 $D_{\min} = D_{0j} (1 \le j \le m)$，则令 $p_0 = p_{0j}$，转 2

正确性分析

算法可以在有限步结束。从算法描述可以看出，每次迭代 D_0 都要单调递减，又因为 site 的点数有限，所以算法可以结束。

算法能够收敛到正确解。设 site 点 p 的邻居 site 点分别为 $p_1, p_2, p_3, ..., p_m$，下面证明

p 是距离查询点 q 最近的 site 点，当且仅当，$|p-q| \leq |p_i-q|, i=1,2,3...m$。

必要性自然成立。

反设必要性不成立，即存在 p 点满足$|p-q| \leq |p_i-q|, i=1,2,3...m$，且存在一个 site 点 p'，使得$|p-q'| \leq |p-q|$。因为 p'的存在，所以 q 点不在 p 点的 Voronoi Cell 内。设组成 p 点 Voronoi Cell 的 Voronoi Edge 是逆时针方向（也就是 DCEL 的表示方向），则必存在一个边 e，使得 p 在 e 的 ToRight 方向。（Voronoi Cell 是凸的，所以如果 p 对所有的边都是 ToLeft，则 p 就在这个 Cell 内部了）。设这个边 e 对应的相邻的 site 点是 p_k，则 p_k 也在 e 的 ToRight 方向，所以$|p_k-q| < |p-q|$，这与$|p-q| \leq |p_i-q|, i=1,2,3...m$ 矛盾。所以必要性成立。

综上所述，算法的正确性可以保证。

复杂度分析

可以用距离的比较次数来衡量算法的复杂度。

算法的最坏情况下的复杂度为 O(n)，比如下图中的情况，在随机初始点和查询点不在一个半平面的时候，迭代一步后会找到中间的点，然后再查邻居需要 n-1 次比较。所以算法对这个例子的时间复杂度为 O(n)。

Monotone Subdivisions 算法

算法思想和流程

Monotone Subdivisions的思想是先对E^2构造出一组关于Y轴方向（竖直方向）单调的Separator，这些Separator将整个Voronoi平面分成N个部分（N是Voronoi图中Site的个数），相邻的两个Separator之间有且仅有一个Face。Separator是由一些首尾相连的HalfEdge构成的折线，并且是关于X轴反方向（水平方向）单调折线。当查询某个点属于那个Site的Face时候，可以判断查询点是在Separator上面还是下面，用这些单调的Separator进行binary search，可以确定查询点是在哪两个相邻Separator之间，从而就可以知道查询点是那个Site的Face中。

构造 Separators：

我们用Voronoi图的dcel数据结构来构造我们的Separator(S_1，S_2，…,S_{N-1})，其中N是Voronoi图的Site个数。构造方法分两步：

1．找到lowest_then_rightmost_site，用它的face周围的halfEdges0 构成Separator0。[1]①

2．从右到左遍历 halfEdges0 的每个 halfEdge00，找到这个 halfEdge00 的twinEdge 的 incFace00。我们可以想象加入这个 incFace00 后的新的 halfEdges1。但是我们需要检测这个 halfEdges1 是否单调。事实上，我们只要验证这个 halfEdges1 中属于 incFace00 周围的那些 halfEdges_tmp 组成的折线是否单调。

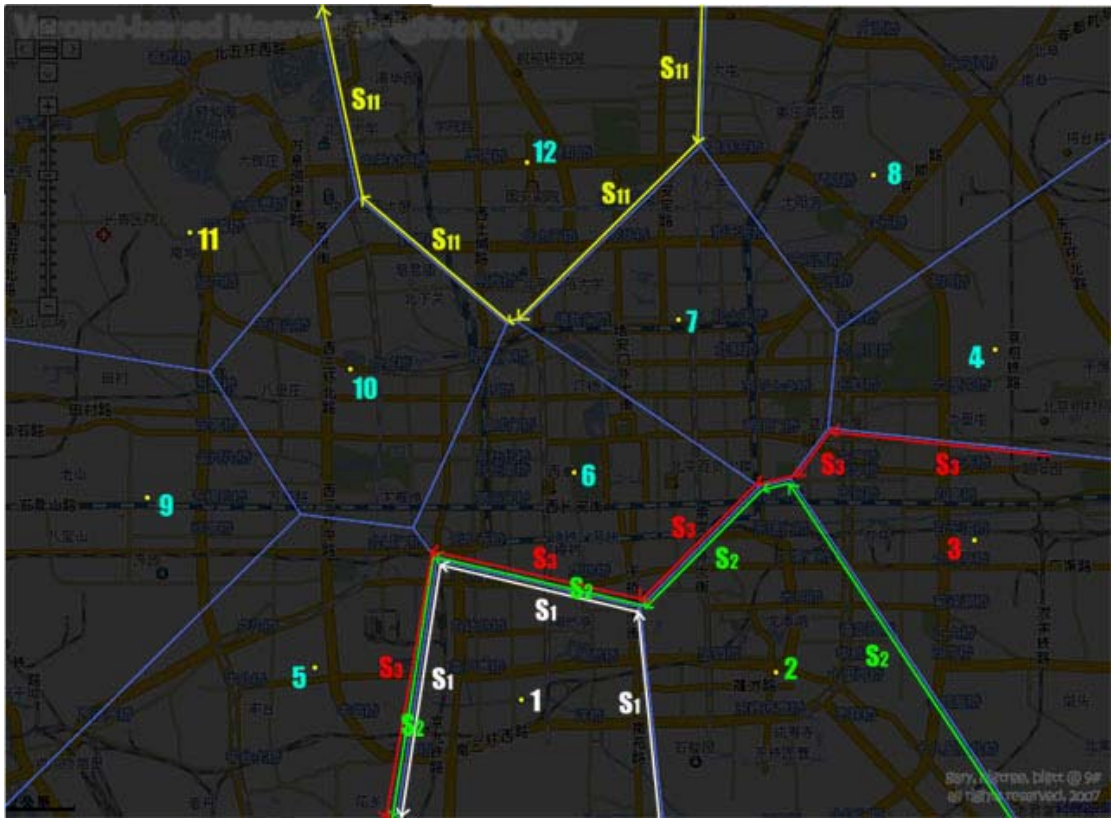
3．如果不单调，那么继续遍历 halfEdges0，令 halfEdge00 在折线中的下一个 halfEdge 为 halfEdge00，转 2.

4．如果单调，那么新的 halfEdges1 满足单调条件，用他构成新的 Separator1，令 halfEdges0 为 Separator1 的折线，转 2.

5．如果在 2 中从右到左遍历 halfEdges0 所有的 halfEdges，检测后都不单调，则算法结束。

构造的过程如下图：S_1是最初构造的Separator。S_1，S_2，…，S_{11}是逐渐构造的Separator。图中的Site的number是逐渐加入的face，从而构造新的Separator。

[1] 显然，lowest_then_rightmost_site的face周围的halfEdges构成的折线肯定是单调的。可证明。



图

两次 Binary Search：

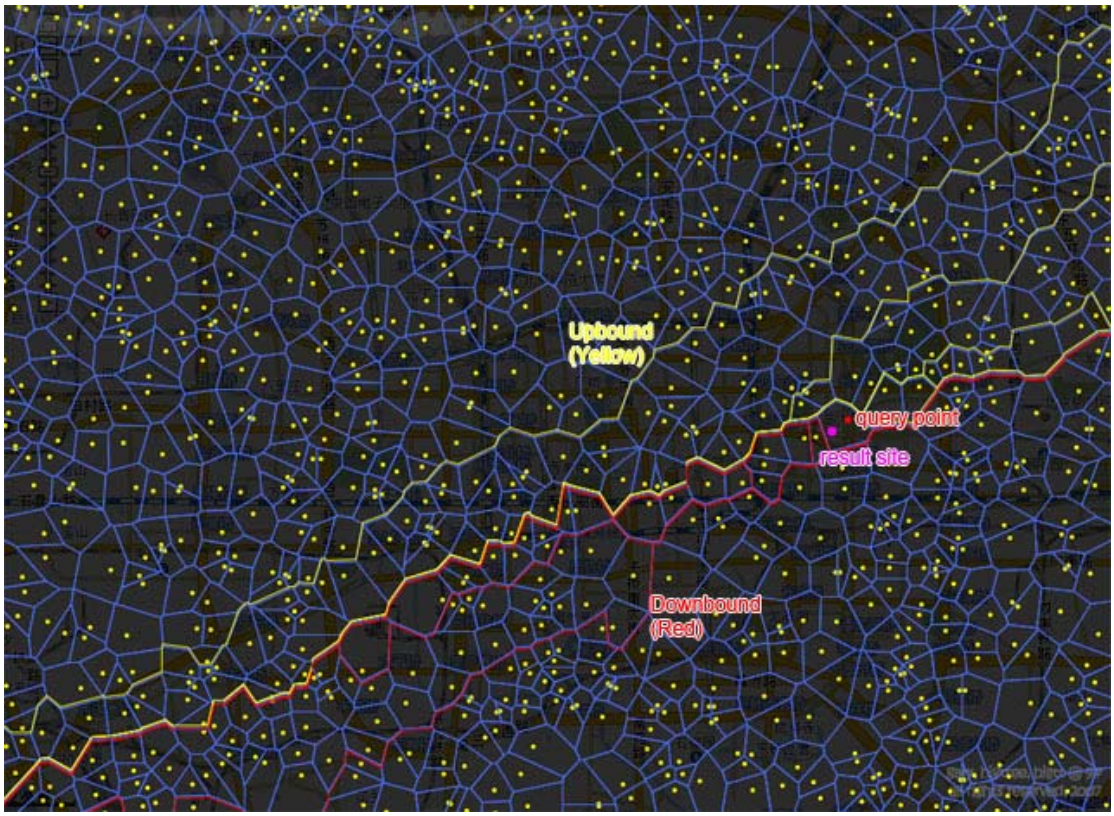
对于用户的输入的 QueryPoint q，我们可以进行两次的 Binary Search 来查找 q 属于那个 face。

比如我们有个子函数 boolean down(QueryPoint q, Separator s)可以判断，q 是否在 s 的下面（把 s 看成从右到左的折线，那么即判断 q 是否是 s 的 ToLeft）。由于这些 Separator 是单调的，所以 Binary Search 后可以得到包含 q 的两条相邻的 Separator。而两个 Separator 之间只有一个 face，结果就是这个 face 的 site 点。

子函数 down()也需要一次 Binary Search 来判断是否 ToLeft。具体就是二分查找到 q 的横坐标在 s 中某个 halfEdge 的首末端点横坐标之间，然后再判断 q 是否是这个 halfEdge 的 ToLeft。

下图是具体 query point 的查找过程，其中 Yellow 线是从上面不断逼近 query point

的线，Red 线是从下面不断逼近 query point 的线，最后两组线逼近相邻后，他们之间的 face 的 result site 即为所求。



复杂度分析

我们主要看 query 过程的时间复杂度和空间复杂度。

两次Binary Search的时间复杂度是O(logm * logn)。其中n是Voronoi图的site个数，m是Separator中halfEdge个数的平均数。对所有Separator进行二分查找点的时间复杂度是O(logn)，而对每个$Separator_i$和查询点之间进行比较的子函数down()的时间复杂度是O($logk_i$)，k_i是$Separator_i$中halfEdge的个数，所以总的时间复杂度就是O(logm * logn).

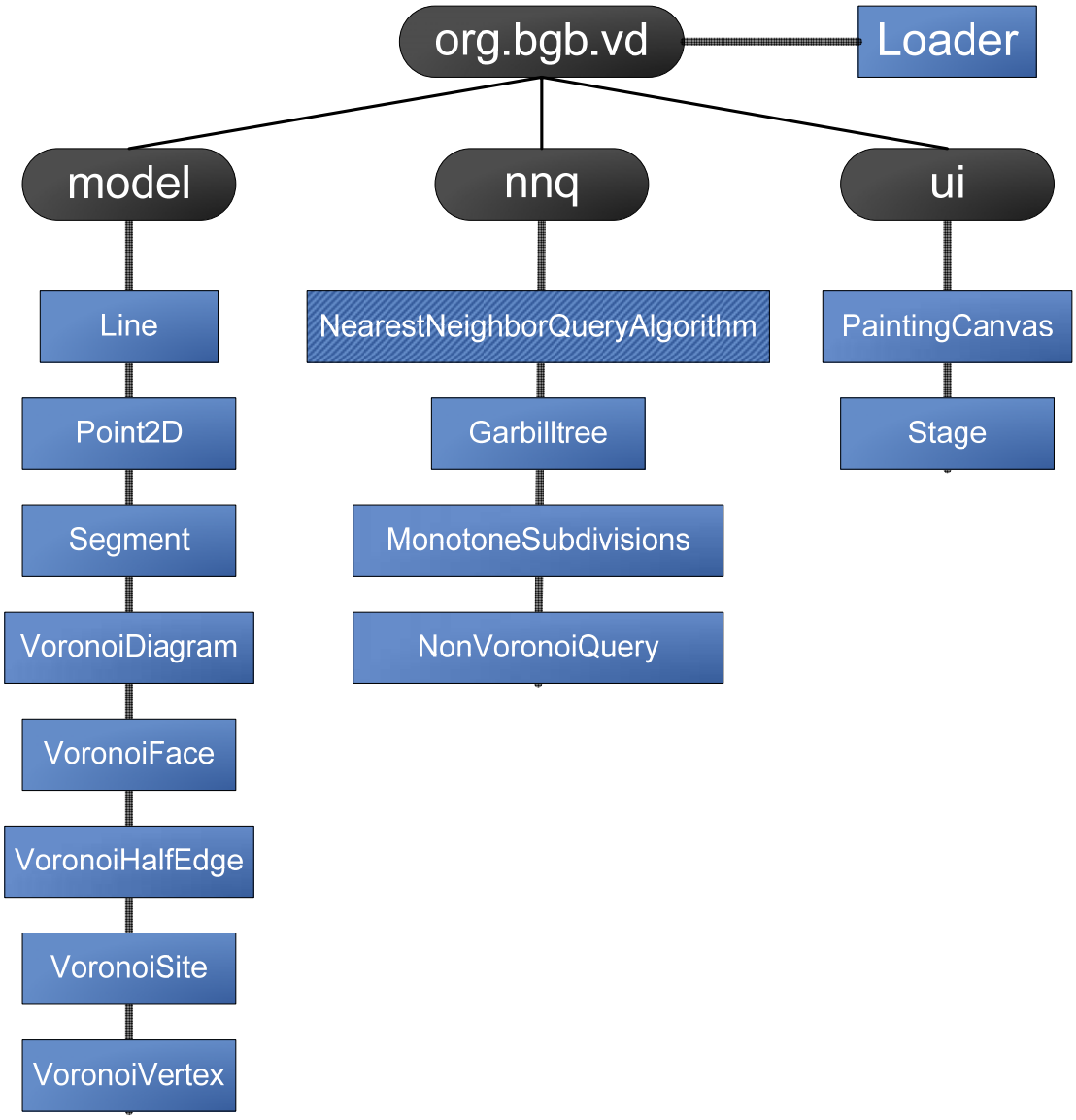
空间复杂度是O(n^2)。

实现简介

整体框架

我们的实验使用的开发环境是 Java 5.0 ,使用的 IDE 是 Eclipse 3.1 ,版本控制 :SVN。

程序所包含的包和类如下所示：



其中，Loader 类是系统的入口，用于启动程序；model 包下面的类表示了算法用到的各种模型，主要是用于表示 Voronoi 的各种数据结构，还有一些抽象的对象类（点、线、线段

等）;nnq 包下面的类则是表示各种最近邻查询算法。为了能够对三种算法进行统一的操作，而又尽可能使算法之间独立 ,在 nnq 包中定义了一个 NearestNeighborQueryAlgorithm 的 Interface，其中定义了 query 和 animatedQuery 两个函数，对于每一个最近邻查找的算法，只需要用一个类来实现该接口即可，我们所讨论的三种算法即对应其中的三个类 Garbilltree，MonotoneSubdivisions，NonVoronoiQuery；在 ui 包中包含了两个界面相关的类，PaintingCanvas 是一个用于作图的类，Stage 则为整个界面的实现类。

Voronoi 构造

Incremental 构造算法的基本构架如课件所讲。不过程序要处理边界的特殊情况。

在 Voronoi 图别上的 Cell 是不闭合的，可以证明会有两条相应的 Voronoi Edge 是射线而不是线段。程序中处理的办法是，在 DECL 中，当 edge->nextEdge 为 null 时，则这条边就是射线，e->twinEdge->originVertex 是一个非常大的坐标值（程序中用的是 1e8）。这样 e 和 e-twinEdge 就定义了这个射线。这种“开口”的 cell 导致了算法在求直线和 Cell 相交的时候，不能从某个边出发，只使用 nextEdge 来遍历各个边求交。因为有可能到射线的地方导致结果不对。所以查找的时候要先用 nextEdge 向一个方向，如果求交未果且到了射线结束，就用 prevEdge 再反方向查找，如果仍然没有找到交点则结束。还有一种退化的情况就是有“两个开口”。比如初始点在一条直线上，则所有的分割线都平行，每个 cell 在两个方向“开口”。这种情况我们程序无法处理。

Garbilltree 算法实现

由于我们的程序使用 DCEL 来表示 Voronoi 图，所以 Garbilltree 的实现比较简单。

按照之前的算法描述，该算法最关键的是对某个 site 的相邻 site 的遍历的实现。在 DCEL 中对 site 的相邻 site 遍历代码如下（假设对 current 这个 site 的邻居 site 进行遍历）：

```
edge = current.getFace().getIncEdge();       // 找出一条起始边

site = edge.getTwinEdge().getIncFace().getSite();    // 找到该起始边所对应的另一个 site

firstSite = site;  // 记录第一个 site，作为判断终止的条件

do
{
    double dist = site.getPoint().distanceTo(queryPoint);

    if (dist<minDist) // 找最小距离

    {
        minDist = dist;
        minSite = site;
    }

    edge = edge.getNextEdge(); // 遍历下一个边

    if (edge==null)
    {
        break;
    }

    site = edge.getTwinEdge().getIncFace().getSite();  // 取下一个边对应的 site

}
while (site!=firstSite);
```

MonotoneSubdivision 算法实现

MonotoneSubdivision 的算法比较复杂，但大致思想比较简单，主要一些细节比较复杂，主要是在边界的地方会出现退化情况。

构造 Separators

```
public void initialize(VoronoiDiagram vd)
```

```
    {
        // first separator
        VoronoiFace f0 = this.lowestFace();
        s0 = this.firstSeparator(f0);
        this.separators.add(s0);

        boolean found = false;
        do
        {

            s0 = this.separators.get(this.separators.size() - 1);
            halfEdges = s0.getMonotoneHalfEdge();

            found = false;
            for (int i = 0; i < halfEdges.size(); i++)
            {
                // if can add a new face
                if (this.addFace(halfEdges.get(i)))
                {
                    s1          =          this.newSeperator(this.newHalfEdges,
halfEdges.get(i));
                    this.separators.add(s1);
                    found = true;
                    break;
                }
            }
        }
        while (found);
    }
```

Query 过程

```
    public Point2D query(VoronoiDiagram vd, Point2D queryPoint)
    {
        // number of points too small
        if (vd.getSites().size() == 0)
        {
            return null;
        }
        if (vd.getSites().size() == 1)
        {
            return vd.getSites().get(0).getPoint();
        }
```

```
        // number of points normal
        int Y = 0;
        int Y1 = 0;
        int Y2 = this.separators.size() - 1;
        // top
        ...
        // bottom
        ...
        // y binary search
        while (Y2 - Y1 > 1)
        {
            Y = (Y1 + Y2) / 2;
            halfEdges0 = this.separators.get(Y).getMonotoneHalfEdge();
            if (this.down(halfEdges0, queryPoint)) // down
            {
                Y2 = Y;
            }
            else
            {
                Y1 = Y;
            }
        }
        // middle
        p = this.separators.get(Y2).getLastFace().getSite().getPoint();
        return p;
    }
```

这个算法出现的主要问题有：

1. 判断一个点 p 是否是一个 Separator 的 ToLeft。

这个 ToLeft 的完成最后是判断 p 和 Separator 的某个 halfEdge 的 ToLeft。而当 halfEdge 是竖直直线，并且 p 的横坐标和与 halfEdge 两端点横坐标相同，且 p 不再 halfEdge 上时，ToLeft 会无法判断，因为返回 True 或者 False 都有可能。如下图：

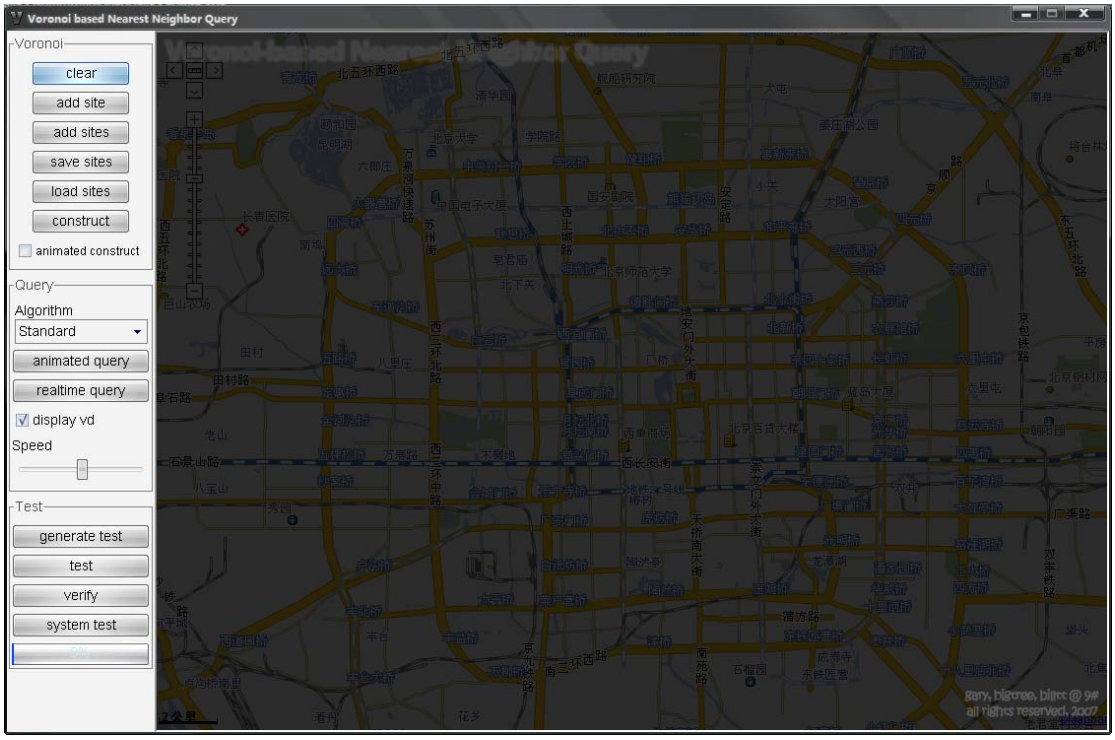
我的解决方法，就是交给他相邻的那个边去判断。

2. Java 中 int 转 double 的存储问题。

我们由于有图形界面，所以都用 int 来保存点的坐标。但是当我们把相同 int 分别转为两个 double 的时候发现得到的 double 值不一样。这也是在 halfEdge 为竖直直线是出错。我们的方法，就是自己写一个比较两个 double 大小的函数。

界面设计及使用说明

本实验的程序界面如下图所示：



在以上界面中，左边是工具栏，右边是作图区域（使用北京地图作为背景）。

工具栏分成三个部分，一是 voronoi 作图的部分，二是查询的部分，三是测试部分。下面分别作介绍：

1、voronoi 作图部分

a) clear：清除作图区域的所有 sites 和 voronoi 图

b) add site：向图中添加 site，点击该按钮之后，会进入添加 site 的状态，此时用鼠标点击可直接在作图区域内添加 site，添加结束后点击“stop”即可。

c) add sites：向图中添加若干个（随机）点。点击之后，输入需要添加的点数，然后程序会自动随机添加该数量的点。

d) save sites：将当前作图区域内的 sites 位置信息记录在文件中

e) load sites：将保存的 sites 信息载入当前作图区域，原有的 sites 信息将不被保存。

f) construct：基于当前的 sites 构造 voronoi 图。

g) animated construct：该选项表示是否显示 monotone subdivisions 算法的动态初始化过程（即 separator 的构造过程）。

2、查询部分

该部分提供的功能是使用各种最近邻查询算法进行查询

a) algorithm：选择查询所需要的算法，该选项除了影响下面的 animated query 和 realtime query 之外，还在 Test 中生效

b) animated query :点击该按钮之后,在作图区域中使用鼠标单击输入一个查询点，然后程序会按照选定的最近邻算法动态的演示查询过程；

c) realtime query：点击该按钮之后，将鼠标移动到作图区域中，程序会实时的查询鼠标当前位置的最近邻，并显示出来；

d) display vd：该选项表示是否在实时查询的过程中显示 voronoi 图，如果显示的话，速度可能会比较慢（sites 很多的时候）。

e) speed：该滑动条控制 animated query 的速度，另外对构造时 monotone 的初始化过程的演示速度也进行控制

3、测试部分

这部分进行各种算法的正确性和性能的测试，基本测试方法是随机产生一定数量的测试点（查询点），然后用各种算法对这些点进行查询，最后统计各种算法所用的时间，同时将 Garbilltree 和 monotone subdivisions 的结果与标准算法的结果比较，以考查其正确性

a) generate test：生成测试数据。点击后输入需要生成的数据点数，然后选择保存的文件，程序会将生成的测试点保存在文件里；

b) test：测试某种算法的性能和正确性。在构造完 voronoi 图之后，首先选择需要

测试的算法（在查询部分的 algorithm 列表中选择），然后点击该按钮，选择测试数据，然后在选择保存的文件，程序会使用测试数据中的点进行测试，并将查询的结果保存在.result 文件中，将所用的时间保存在.result.time 文件中；

c) verify：用于验证算法正确性，其本质是检验两个 result 文件中的结果是否一致。点击之后，先选择测试数据，然后分别选择待检验的算法生成的 result 文件和标准算法生成的 result 文件，若两者一致，则提示“Verification OK”，否则提示不一致的点数，以及在文件 diff.txt 中列出所有不一致的点。

d) system test：对三种算法进行比较系统的测试。即自动使用不同 site 数量的 voronoi 图对三种算法进行测试，然后将性能指标输出到文件。首先在作图区域中加入一定数量的 sites，然后点击该按钮，输入需要的步数，选择测试文件，然后程序会自动进行测试，并将结果保存在 test 文件夹下的 systemtest.txt 中。比如当前有 100 个 sites，选择的步数为 4 的话，系统就会使用 25、50、75、100 个 sites 组成的 voronoi 图对三种算法进行性能的测试。

实验结果及分析

测试说明

测试的具体方法在上面的界面介绍中已经介绍过了，我们的测试分为两个部分，一是验证 Garbilltree 和 Monotone Subdivisions 算法的正确性，当然该验证是从实际统计的角度来看的，理论的验证请参考前面的算法分析部分；二是性能测试，即测试三种算法的查询速度。

在以下的所有测试中，我们使用的测试数据有两种，一是 1000000 个测试点，保存在文件 1M.test 中，另一种是 100000 个点，保存在 100k.test 中；voronoi 图使用一个有 2000 个 site 的图，保存在 2000.site 中。

正确性测试

正确性测试使用的测试集是 1M.test

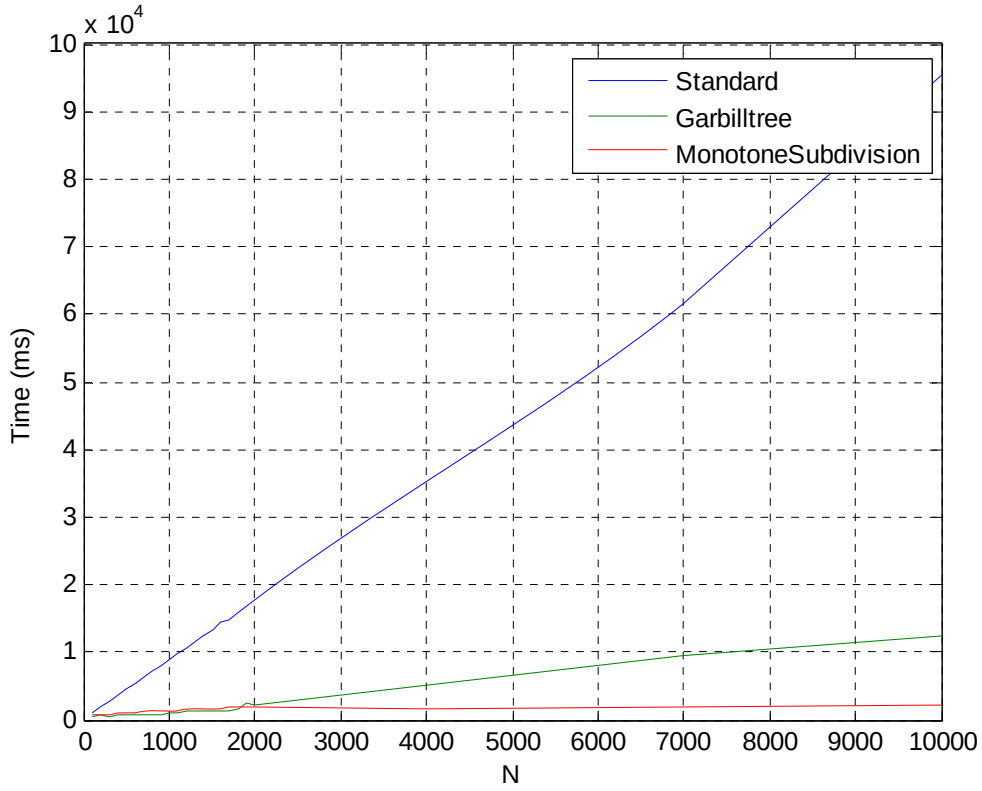
通过使用 verify 功能进行校验 ,发现 garbilltree 和 mono-subdiv 算法的测试结果均是 100%与 standard 匹配。

性能测试及对比

使用 3 种算法，对不同规模的 n 进行计算，所用的时间如下表（单位：毫秒）

N	Standard	Garbilltree	MonotoneSubdivision
100	1156	422	750
200	1953	625	766
300	2750	531	875
400	3594	609	984
500	4469	656	1031
600	5469	750	1125
700	6187	766	1187
800	7172	844	1266
900	8156	875	1313
1000	8844	1000	1390
1100	9781	1016	1453
1200	10563	1172	1500
1300	11437	1234	1578
1400	12407	1250	1656
1500	13234	1266	1688
1600	14468	1454	1735
1700	14906	1453	1812
1800	15953	1563	1844
1900	16765	2469	1906

2000	17656	2047	1953
4000	35234	5125	1688
7000	61485	9360	1921
10000	95547	12359	2171



当 n 比较小的时候，Garbilltree 算法优于 MonotoneSubdivision。但是当 n 变大到一定规模后可以看出 MonotoneSubdivision 好于 Garbilltree。由于 MonotoneSubdivision 复杂度是 $O(\log^2 n)$，Standard 复杂度是 O(n)。所以可以知道 Garbilltree 方法介于两者之间。

我们用实验数据尝试进行拟合，但是由于测试数据有限（n 很大以后的数据很难测试），另外对时间的衡量存在误差。所以最终我们没有得到很好的拟合结果，在此无法给出显式的复杂度的公式。