计算几何大实验报告

多边形内部点集可见图的计算

Computing the Visibility Graph of Points Within a Polygon

指导教师：　邓俊辉

参与学生：　贾云涛　2004211034

杨永亮　2004310471

张弛　2004211045

1．Introduction:【问题介绍】

可见图（visibility graph）是一个基本的几何结构，有着广泛的应用，包括照明（illumination）和渲染（rendering），运动计划（motion planning），模式识别（pattern recognition）和神经原网络（sensor network）。

定义 Q 为一个由 n 个顶点组成的多边形，定义 P = {s1,...,s_m}为 Q 内部的 m 个点（定点）；P 中的点可以位于 Q 的边界上或者在 Q 的内部。P 相对于 Q 的可见图表示为 $VG_Q(P)$，定义为一个以 P 为顶点以 P 内相对 Q 可见的点之间的连线为边的图。所谓 P 内相对 Q 可见的点指两点之间的连线位于 Q 的内部。我们的主要问题是如何快速有效的计算 $VG_Q(P)$。

关于可见图（visibility graph），已经有了很多的研究。以前最优的结果为计算 Q 内所有定点的可见图（P = V(Q)）：如果 Q 为简单多边形，时间复杂度为 O(n+k)；如果 Q 包括空洞，时间复杂度为 O(nlogn+k)，k 为输出的大小。在我们参考的最新的研究中，P 不特定为 Q 的顶点，Mitchell 04[1]提出了一种输出相关的算法：如果 Q 是一个简单多边形，时间复杂度近似与输入大小(m+n)和输出大小 k 成线性。

在 Mitchell 04[1]的研究中还提出了一种强可见图（robust visibility graph）的概念，描述相对于 p 几乎可见的点 q 有可能不会被一个站在 p 点的摄像机捕捉到的情况，这依赖与摄像机的分辨率。为了使得 q 点相对于 p 点鲁棒可见（robustly visible），我们要求无论 q 点在一个半径为 r 的圆域（disk）Br(q)如何移动，同时保证在 Q 内部，都要相对于 p 点可见。圆域的半径与 p、q 的距离相关，依赖与参数α，其中α定义为圆域相对于 p 点的的角度。

另外在很多实际的应用中，可见性往往都会受到距离的约束，在 Mitchell 04[1]的研究中也提到了一个受距离约束的可见图（distance-restrict visibility graph）的求解算法，复杂度与输出相关。

2. Preliminaries:【问题定义】

Q 为平面上的一个 n 顶点的多边形，比如：Q 是一个 R^2 的连通子集，其边界为 ∂Q 且由 n 条直线段组成。我们考虑 Q 为平面上的一个闭区域（即 Q 包括它的边界 ∂Q ）并假设 Q 的内部是连通的。我们用 h 表示 Q 内部的连通域（hole）的个数；这样 ∂Q 包括 h＋1 段封闭边界。当 h 为 0 里 Q 是一个简单多边形。

我们用 $P \subset Q$ 表示 Q 内部的 m 个点。定义点集 P 为定点集。对于其中任意两点， $p,q \in Q$，为相互可见，当且仅当 $\overline{pq} \subset Q$ 。P 相对于 Q 的可见图(visibility graph) $VG_Q(P)$ 定义为一个以定点集 P 为顶点且以所有相互可见的顶点连线为边组成的图。我们用 k 表示 $VG_Q(P)$ 的边的个数。

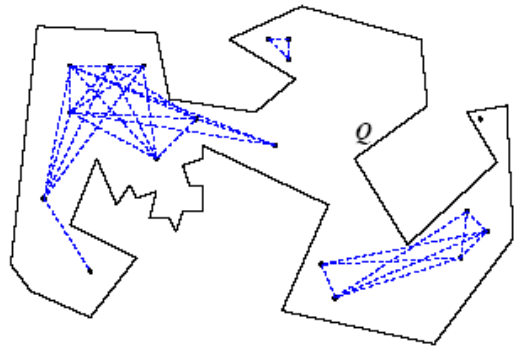


Figure 1. The visibility graph $\mathrm{VG}_Q(\mathcal{P})$ is shown with dashed edges joining pairs of visible sites of $\mathcal{P}$ within a polygon Q.

3．Related Work:【相关工作】

此前计算可见图（visibility graph）的算法只讨论了点集 P 为多边行 Q 所有定点的情况。第一个构造可见图（visibility graph）$VG_Q(V)$的算法时间复杂度为 $O(n^2 logn)$，它主要是对 Q 的每一个定点进行放射线扫描。Welzl 和 Asano 将时间复杂度改进到 $O(n^2)$，在最坏的情况下最优但不是输出相关（ouput-sensitive）。Hershberger 给出了一个对简单多边形 Q 计算可见图（visibility graph）的最优 O(n+k)的输出相关的算法。对于广义的多边形（带有空洞），Overmars 和 Welzl 给出了一个相对简单 O(klogn)时间复杂度，O(n)空间复杂度的算法。接着，Ghosh 和 Mount 给出了一个 O(k+nlogn)时间复杂度，O(k)空间复杂度的算法。Pocchiola 和 Vegter 利用可见复杂度（visibility complex）的概念将空间复杂度改进到最优，他们的算法需要 O(k+nlogn)的时间复杂度和 O(n)的空间复杂度。

4．Algorithm:【基本算法描述】

将 Q 用一条对角线 ξ 分割成两个子多边形 Q_1 和 Q_2。于是 P 中的所有站点被分割为两个子集，其中 $P_1 \subset Q_1$， $P_2 \subset Q_2$，同时在分割的时候考虑两个子集等势的因素（例如 $|P_1|,|P_2| \geq m/4$）。接着我们递归的计算 $VG_{Q_1}(P_1)$ 和 $VG_{Q_2}(P_2)$。然后还需要考虑在 $VG_Q(P)$ 中且穿过切割对角线 ξ 的那些定点 s_i, s_j，其中 $s_i \in Q_1, s_j \in Q_2$。对于每个站点 $s_i \in P$，我们确定切割对角线的子线段 $\sigma_i \subseteq \xi$，使得该子线段上的所有点对于 s_i 均可见。用 L_i 表示所有过 s_i 并且和 σ_i 相交的无穷直线。这些直线组成了一个对偶图上的“double wedge”（也叫做“line segment”） L_i^*（中间经过任意一个 point-line 对偶变换）。于是 $s_i \in Q_1$ 对于 $s_j \in Q_2$ 可见，就意味着 $L_i \bigcap L_j \neq \phi$，也就是说有唯一的一条直线 $l \in L_i \bigcap L_j$，l 即为那条过 s_i 和 s_j 的直线。如下图所示：

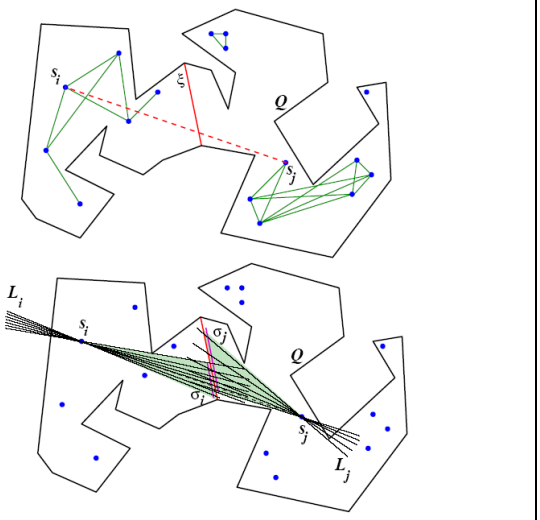


Figure 2. Illustration of the cut ξ, the visible subsegments σ_i, and the set of lines L_i. Site s_i sees site s_j if there is a line in common to L_i and L_j.

进一步，在二值图中，s_i 和 s_j 的可见性问题变成了判断 line segment L_i^* 和 L_j^* 是否相交的问题，交点 l^* 对应于原图中过 s_i 和 s_j 的直线 l 。

于是我们可以通过计算“line segment”来求所有过 s_i 并且和 σ_i 相交的无穷直线，通过“red-blue segment”的求交问题来片段两个站点 s_i 和 s_j 是否可见，其中“red segment”代表了 P_1 中的“segment” L_i^*，“blue segment”代表了 P_2 中的“segment” L_j^*。

于是问题的难点转化为 red-blue segment 求交问题，对于 red segment union 和 blue segment union 连通的情况，目前为止最好的算法 K red-blue intersection 的复杂度为 $O(K+r+b)$ ，r 为 red segment 的个数，b 为 blue segment 的个数。（如果 union 为不连通的情况，求交的算法复杂度在 $O((r+b)^{4/3}\log(r+b)+K)$ 的量级）。

我们也可以去掉 red-blue segment 求交限制，对于所有的 dual segment L_i^* 求交，而不管它们对应

的站点 s_i 是否位于同一个子多边形中，同样可以用是否有交点来判断两个站点是否相交。但是这个方法存在一个问题，即对于递归的分割，两个站点 s_i 和 s_j 很可能又位于同一个子多边形中，这样就会带来重复计算。

考虑到递归树的高度，一个可见站点对 s_i 和 s_j 在被对角线分隔之前至多能同时位于 $O(\log m)$ 个子多边形中，也只可能被重复计算 $O(\log m)$ 次。于是总的算法复杂度如下：

Proposition 1. 简单的 divide-and-conquer 算法的复杂度为 $O(n+m\log m\log mn+k\log m)$ 。

5. Improved Algorithm:【改进算法描述】

这种算法对前面简单的 divide and conquer 算法进行了改进，主要体现在去掉上面算法中的 $O(\log m)$ 和输出 k 的乘积项，这个乘积项是由可见定点对的 over reporting 造成的。

改进算法中利用了一个由 Quibas 和 Hershberger 提出的查询结构，这个查询结构能解决多边形内部的最短路径问题，经过修改后可以用在我们的问题中。

要建立一个得到最短路径的查询结构，我们首先对多边形 Q 进行层次分割，即递归的将多边形用对角线分成两个子多边形。每次用于分割的对角线构成了一棵二叉树，在这棵二叉树当中，左右孩子对角线分割了由父对角线分隔的左右两个子多边形。我们再利用因子图（factor graph）来增强这个分解二叉树，采用在分解过程中对处于同一个子多边形上的对角线增加一条连接边的方式。也就是说，对于已经分割好的多边形 Q 中的两条对角线，如果这两条对角线被factor graph中的一条边相连，当且仅当它们在 Q 中有一条路径相连，这条路径不会经过二叉树中深度小于自己的的对角线。对于factor graph中的每条边，Guibas和Hershberger计算了一个hourglass的查询结构，该结构能够处理以下查询：

给定一个点 p 和一个连接 d_1 和 d_2 的hourglass h，寻找一条从点 p 到 d_2 上的点的最短

路，而且假定这条最短路通过 d_1。该查询需要的时间正比于两条对角线在二叉树中的高度和。建立factor graph的及其查询结构的时间正比于所有的对角线个数。

为了能在判断 P 中定点可见性问题中使用该结构，我们首先选择能使左右两个子多边形中定点个数平衡的一系列对角线进行分割。其中一些对角线的端点可能会落在在多边形的边内部一点，而不是多边形的顶点，所以多边形的复杂度增长为 $\Theta(m+n)$。递归分割直到每个叶子多边形中只有一个定点。我们可以在 $O(m+n)$ 的时间内建立因子图（factor graph）和查询结构。我们只需要那些“开”的“漏斗（hourglass）”，即互相可见的对角线。这样，我们可以利用如下类型的查询：对于给定的一点 p 和一个hourglass h，确定h所关联的所有对角线相对于p可见的部分，如果存在。为了回答这个可见性查询问题，我们确定一个点，使得由p出发经过该点的射线与h的某一边相切。寻找此切线问题是最短路查询的一个基本问题，所以可以在O(log(m+n))=O(logmn)时间内基于此查询结构完成。

其次计算互相可见的定点对。我们通过构造一个对偶线段集合来标示该定点光束的可见区域。考虑从定点s出发的光束r，以及r最初的在多边形Q内部可见的对偶线段i。这个初始的线段i会和周围D条对角线相交。如果D不为空，设d为D中深度最小的对角线元素，我们将光束r与d相关。也就是说，我们将多边形内第一个可能切到该光线的对角线与该光线相关。如果从s_1出发的光束直接射到定点s_2，那样s_1就也与这条对角线相关，因为这条光束跨越了分割s1和s2之间的一条对角线。如果我们像这样把所有定点s_1发出的光束关联到相关的对角线，我们就对从s_1出发的所有光束进行了一个分割。在与定点相关的对角线集合中不能存在同样深度的对角线，这样每一个定点的光束的分割大小都是O(logm)。计算这些分割可以在O(logmlogmn)时间内完成。

最后计算可见定点对的集合。我们从考虑每一条对角线出发，计算与该对角线d相关的所有光束分割的交点。然后在对偶空间，光束分割就对应于一条线段，这样我们就可以在$O(k_1+m_1\log m_1)$时间内计算出所有线段的交点，其中m_1对应于与对角线d相关的对偶线段个数，k_1是交点的个数。

引理 1. 确定的定点对只会被发现一次或者两次。

求交运算的总复杂度为$O(k+m\log^2 m)$。这样求得k对可见点的总时间为O(n+mlogmlogmn+k)。

定理 1.一个n个顶点的简单多边形Q内m个定点集合P的可见图$VG_Q(P)$可以在

O(n+mlogmlogmn+k)时间内完成，k为最终可见图的定点对的数目。

6. Plan:【实现计划】

6.1 数据结构

1. 简单多边形 Q 采用 DCEL(double connected edge link)结构表示。
2. 改进的多边形 Q 对角线剖分采用 Factor Graph 表示。
3. 定点与对角线对应关系的数据结构：每一个点记录了它的光束分割对应的对角线集合；每一条对角线记录了所有与它对应的点集。

6.2 具体算法

1. 改进的多边形对角线剖分算法[3]。

   包括：多边形的简单对角线剖分算法和平衡层次剖分算法。
2. 改进的最短路经查询算法[2]。
3. 线段集合的求交算法。

6.3 实现目标

1. 基本目标：基于简单多边形 Q 内部点集 P 的可见图的计算。

7. Implementation Work:【实现工作】

7.1 杨永亮 2004310471

7.1.1 简单多边形的结构

简单多边形结构在实验中是用类 Polygon2D 来描述的，Polygon2D 的基本数据类型定义如下：

```
class Polygon2D
{
public:
    Function Declarations;
public:
    Vertex2D* vertices;                    // 顶点数组
    int numOfVertices;                     // 顶点个数
    Edge2D* edges;                         // 边数组
    int numOfEdges;                        // 边个数
    Triangle* triangles;                   // 三角片数组
    int numOfTriangles;                    // 三角片个数
};
```

简单多边形的输入分两种：一是用户用鼠标手工输入，二是从文件导入，本质上是一个顶点序列，所以多边形结构的基本维护就是用一个顶点数组来进行的。在没有进行三角剖分之前，多边形结构只有顶点数组进行了初始化。进行了三角剖分后的结构要组织成一个 Dcel 结构。这时候再初始化边和三角形数组。

图 7.1.1.1 给出了多边形输入后的结果：

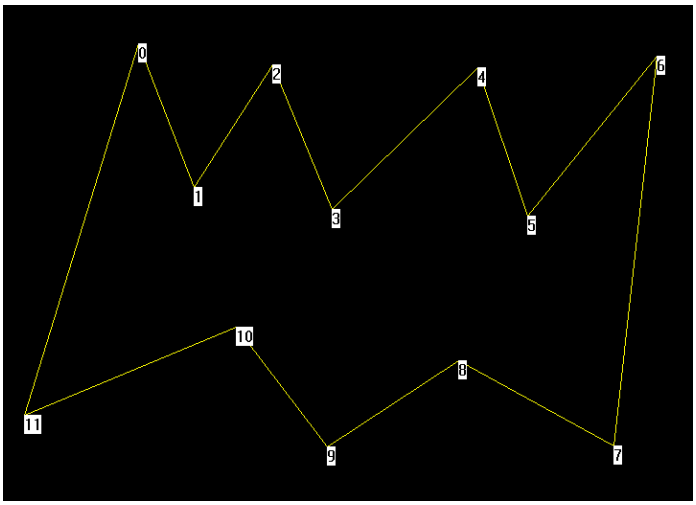


图 7.1.1.1 多边形输入的结果

7.1.2 三角剖分

一旦我们对简单多边形进行了三角剖分，那么对于一个三角形内的所有点（site）来说，它们必然相互可见。所以我们可以把他们直接连接起来，对于不在同一个三角形内的点，我们再用 FactorGraph 的方法来进行判断。

Bernard Chazelle 在 1990 年发表文章说明对于简单多边形的三角剖分可在 O(n)的时间内完成，这个算法只是理论上的很好的突破，还从来没有人将它实现，因此，它的具体的实用价值还没有体现出来。又考虑到我们课程实验的核心是在构造可见性图上，所以我们没有在三角剖分的问题上做过多的纠缠。我们采用了 *Godfried Toussaint* 提出的一个有效的简单多边形三角剖分的算法。*Godfried Toussaint* 仅仅利用了一些几何知识，对多边形的一些特征进行分析，设计出一个简单且也容易实现的算法，它不需要排序和运用平衡二叉树这样的复杂数据结构。这个算法的复杂度是 $O(n(1+t_0))$ ，其中 $t_0< n$，t_0 度量了由算法得到的三角化的形状复杂性，更精确地说，**t_0 表示所有三角剖分后完全由内对角线组成的三角形的个数**。这个算法的最坏情况为 $O(n^2)$。但是，它不仅是第一个计算复杂性是输出复杂性的函数（即输出敏感）的三角剖分算法（这具有很好的理论意义），还有另外一个贡献是，

批注 [Jordan1]:
这里测试用的多边形规模太小，看不出效率的差别

批注 [Jordan2]:
该算法由陈小雕实现，实现有问题实测规模分别为 1001、1111、1300 或 3200 的多边形，三角剖分时均死机

它引进了一种新的度量三角形的形状复杂性的方法。这在以后有人发展了这个概念，用α来表示多边形的形状复杂度，并且应用到模式识别和图象分析中。

简单多边形三角剖分的结果如图 7.1.2.1 所示：

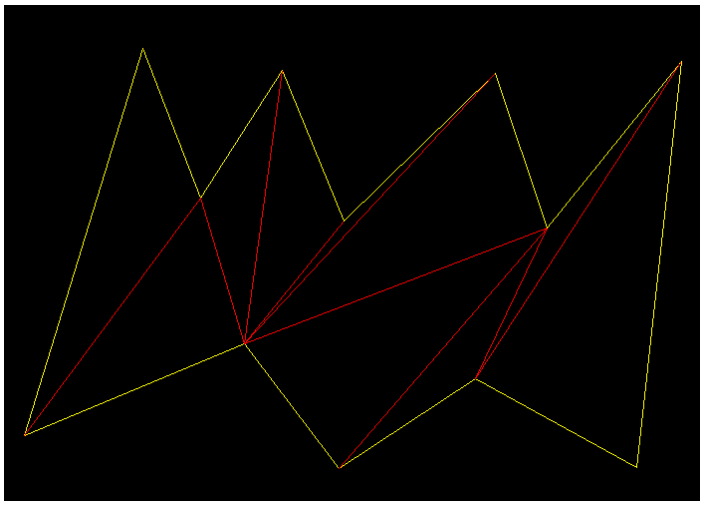


图 7.1.2.1 简单多边形三角剖分

7.1.3 Dcel 结构

上一步进行三角剖分的输出是一个对角线数组，记录了三角剖分中新加入的每条对角线。为了后面对偶图的构建和平衡对角线的剖分，我们要构造出三角剖分后的简单多边形的 Dcel 结构。我们给出的 Dcel 结构由下面点、线、面（三角形）的数据来维护：

```
class Vertex2D
{
public:
    CPoint m_vPosition;                 // 顶点坐标
    int m_iIndex;                       // 顶点标号
    int m_iPolyEdgeTo;                  // 从该顶点出发的多边形边号
    int m_iPolyEdgeFrom;                // 进入该点的多边形的边号
    std::list<int> toEdgeList;          // 从该顶点出发的半边集合
    std::list<int> fromEdgeList;        // 进入该顶点的半边集合
```

```
};
class Edge2D
{
public:
    int m_Vertex[2];            // 起点和终点的编号
    int m_iTwinEdge;            // TwinEdge编号，如果没有为-1
    int m_iTriangle;            // 该有向边所属的面号，在该边左方
};
class Triangle
{
public:
    int m_Vertex[3];                // 三个顶点的编号
    int m_Edge[3];                  // 三条线段的编号
    std::list<CPoint> innerpoints;  // 内点数组
};
```

7.1.4 对偶图的构建

有了 Decl 结构我们就可以在线形时间内构造出简单多边形三角剖分结果的对偶图，该对偶图是一个树形结构，每个节点对应了一个三角形。构造对偶图的目的是为了后面进行平衡对角线分解。我们采用的方法是从多边形顶点编号为 0->1 的有向边所在的三角形开始向两边搜索，直到达到最大深度，即到达多边形边界终止。对偶图的结果如图 7.1.4.1 所示：

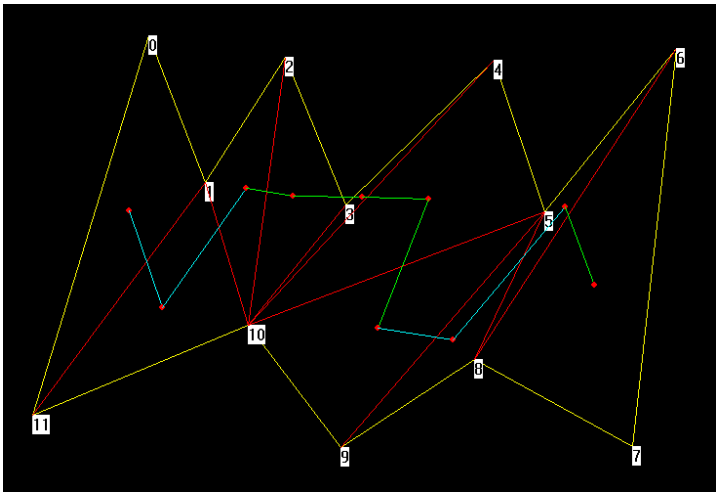


图 7.1.4.1 三角剖分对偶图的构建

7.1.5 平衡对角线剖分

有了三角剖分的对偶图结构（树形结构），我们就可以在该结构上对树进行平衡分解。所谓分解也就是说截掉树中的一个树枝，把原来的树分成两棵子树。其中一棵的根还为原来树的根，另一棵树的树根为截掉树枝的子节点。而我们的分解要做到平衡，这是服务于后面构造 Hourglass 时的查找复杂度。平衡的定义是指每棵子树的节点数目不能少于原来树节点个数的α倍，例如$\alpha = 0.3$，这样的平衡分解可以在线性的时间内完成。由于之前三角剖分的算法平均复杂度已经达到了 O(nlogn)，所以预处理的时间复杂度已经不可能回到线性。所以我们实现的是精确的平衡分解，也就是说在所有分解中的最优解，我们的方法是从上至下计算每个节点的平衡系数ρ，该平衡系数定义为如果截去该节点和它的父节点中间的树枝，所产生的以该节点为根的树的节点数和原来树的节点数的比值，那么这个比值越接近 0.5，说明在该节点处分解最接近平衡。在计算了每个节点的平衡系数之后，然后再对树从下到上回溯，选出平衡系数最大的节点，在该处分解。这样的平衡分解每次计算和选择平衡系数最大的节点是对树的一次遍历，复杂度为 O(n)，由于是严格的平衡分解，所以要搜索分解 logn 次，总的复杂度为 O(nlogn)。下图给出图 7.1.5.1 所示的对偶图的平衡对角线剖分结果：

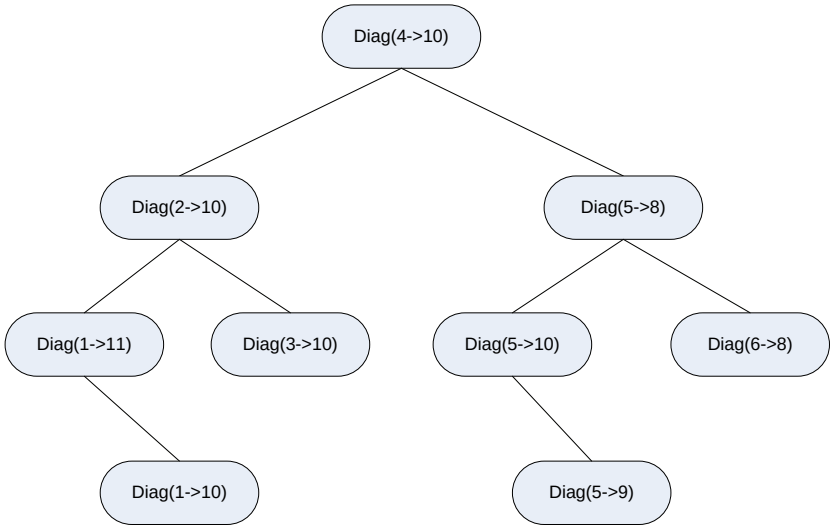


图 7.1.5.1 平衡对角线剖分

7.1.6 点定位

对简单多边形内可见图的计算的复杂度为$O(n+m\log m\log mn+k\log m)$ [1]，其中 m 代表了简单多边形内点的个数。如果对多边形做了三角剖分之后，由于同一个三角形内的点一定互相可见，而在后面构造 Hourglass 的时候我们也可以把同一个三角形内的点统一进行处理，于是复杂度可以减小到$O(n+m'\log m'\log m'n+k\log m')$（其中 m'为三角形的数目）。于是，我们就需要判定每个点在哪个三角形里，就要用到点定位(point location)。在本实验中我们使用的是梯形图(Trapezoid-Map)随机增量算法，将所有的多边形的边和三角剖分的对角线随机加入梯形图中得到查询结构。由于在 Dcel 结构中我们记录了边所对应的三角形，所以我们得到梯形的上边 top(Δ)，通过上边能找到点位于哪个三角形中。构造梯形图和查询结构的期望复杂度为 O(nlogn)，查询复杂度为 O(logn)，所以该步的复杂度为 O(nlogn)＋mO(logn)。

7.2 张弛 2004211045

7.2.1 链（Chain）

链由一组依次首尾衔接的半边组成，对起始和末尾端点构成一个下凸包。

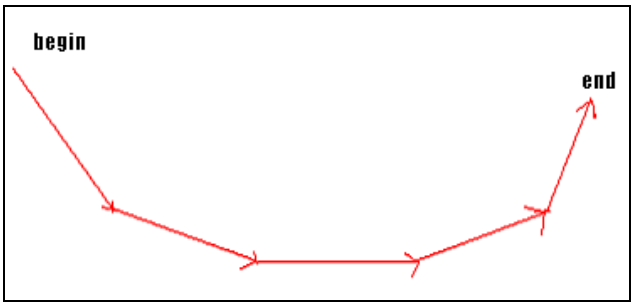


图 7.2.1.1 链

1. 数据结构

实现链的数据结构是一棵二叉搜索树，由前面三角剖分和平衡化的工作，我们得到每个二叉树都是平衡的。

链有两种不同的类型：

1) 原子链：对应于一条半边。

2) 衍生链：两个链（原子的或衍生的）合并而成的下凸包。

在二叉树中，一个原子链对应一个含有该半边的叶子节点；一个衍生链所对应节点，包括构成它的两个子链的公共切线 tan，和分别指向这两个子链对应节点的指针。另外，为了有效构造与输出，每个节点含有两条恢复链表，用来记录该链与构成它的两个子链的差异。在以该节点为根的子树所对应的链中，与 tan 相邻的两条半边，称为邻接半边。而记录它们的节点，称为邻接节点。两个恢复链表分别记录由该节点到邻接节点的路径。

恢复链表（Retrieval List）中的每个元素，由两部分构成，对左恢复链表而言，pointer 指向左子树中的某个节点，partner 指向该节点在另一恢复链表中的位置。具体而言，对于一棵以 v 为根的子

树，u 是它的左邻接节点，s,t 是它左恢复链表上的两个相邻记录。它们有如下性质：

1)．在以 v 为根的子树中，s.pointer 是 t.pointer 的祖先，而且 t.pointer 将被记录在 s.pointer 的右恢复链表中。

2)．t.partner 指向 t.pointer 在 s.pointer 右恢复链表中的位置。

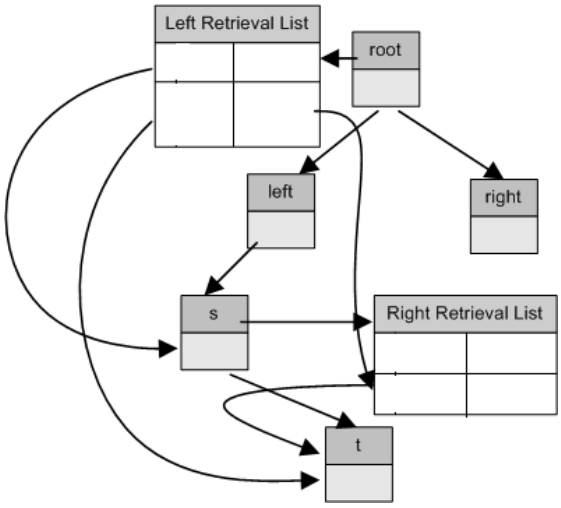


图 7.2.1.2 Chain 的数据结构

2．算法

1) 构造：

对于原子链，直接令切线为对应半边，左、右恢复链表为空即可。对于衍生链，则是两个子链的合并，将在下面给出。

2) 合并：

类似动态凸包算法，我们通过把两个子链进行合并得到新的链。关键步骤在于求出两条子链 left, right 的公共切线。首先考察 left.tan 与 right.tan 首尾衔接形成的切线，与子链中的邻接边是否冲突。如果存在冲突，递归的计算以该邻接边对应临接节点为根的子树，所对应子链，与另一侧的公共切线，并记录递归走向。这样，该递归调用结束后，我们得到一条位于两子链中间的半边。之后，考察凸性是否满足，对于与该切线不满足凸性的子链，由上一步的记录，判断是应该向它的祖先节点

递归，还是向它另一侧的邻接节点递归，同时修改记录。这样类似 zig-zag 算法，我们最终得到连接两条子链得公共切线。

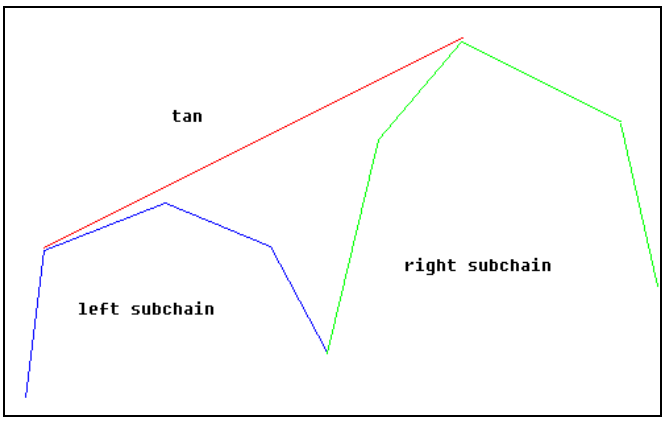


图 7.2.1.2 把左、右两个子链合并为一个新的链

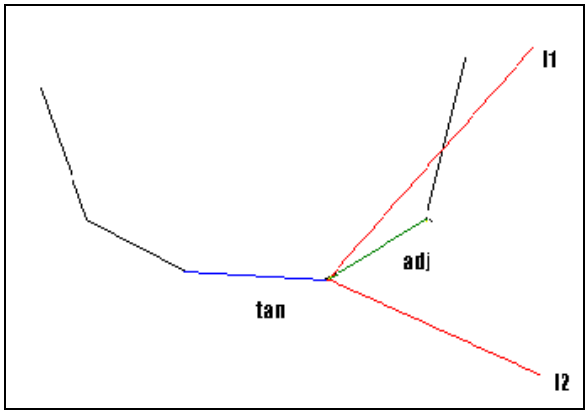


图 7.2.1.3 l1 与邻接边 adj 发生冲突，l2 破坏了链的凸性。

接下来，构造新生成节点的左，右恢复链表。通过在计算公共切线时，记录的递归走向，考察递归路径上的各个节点对应切线，是否在新生成的链中可见。由文献[2]，可以知道，只需要考察上一个可见切线，和新生成的切线是否对其构成“遮挡”。判断遮挡的依据是，线段 a 被线段 b 遮挡，当且仅当线段 a 的端点序号都落在 b 端点序号之间。对于端点不是多边形顶点的情况，在把它所在三角剖分中，不属于漏斗中的点的序号赋予它。对于恢复链表中除去链首的每个元素，在上一元素

指向节点的另一侧恢复链表中，依次搜索，以确定 partner 的位置。

3) 求点到链的切线：

求一点到一链的切线，以点在左侧为例。首先考察该点到 tan 的左端点的射线。如果它与左邻接边发生冲突，则返回该点与左邻接节点为根的子树对应链的切线，而左邻接节点即由根节点左恢复链表的链尾指向。否则，如果凸性被破坏，则相应的返回该点与右邻接节点为根的子树对应链的切线。如果，不破坏前面两条原则，则该切线即为所求切线。

4) 遍历：

通过左、右恢复链表，可以顺时针依次遍历一个链中的所有半边。

首先，遍历切线左侧的所有半边，接着输出切线，然后遍历切线右侧所有半边。以左侧为例，设根节点为 r，左恢复链表的链首元素所指节点为 s，则位于 s 中存储的切线左侧的所有可见半边，即以 s 为根子树对应链中左侧的所有半边，而由 s.tan 到 r.tan 的所有半边，可以通过如下方法遍历：

对于节点 r 中左恢复链表中的某一元素指向节点 s，它的下一元素指向节点 t。遍历由 s.tan 到 r.tan，首先由 partner 得到 t 在 s 节点右恢复链表中的位置，在以 s 为根子树对应链中，遍历 s.tan 到 t.tan 的半边，然后输出 t.tan，接下来遍历 t.tan 到 r.tan 之间的可见半边。

3. 复杂度分析

1) 空间复杂度：

对于每个原子链，显然空间复杂度为 O(1)。

对于衍生链，构成它的子链并不增加它的空间存储量。切线存储是常数的，而恢复链表的长度不会超过子树的高度。对于最终可归结为 N 个原子链的衍生链，它的空间复杂度为 O(logN)。

2) 构造时间复杂度：

构造原子链的时间复杂度为 O(1)。

构造衍生链，可以分为求公共切线，和构造恢复链表两部分。求公共切线，由于左、右子树每层最多访问两次，故是复杂度位 O(logN)。而构造恢复链表，链表中每一元素的 pointer 部分都将在常数时间内构造，因此是 O(logN)的。构造 partner 部分时，搜索节点 t 在节点 s 中的位置，搜索范围为 s 恢复链表中，节点高度在 s，t 高度之间的部分，因此所有搜索范围之和不超过根节点 r 的高度，因

此时间复杂度也是 O(logN)。故构造一个由 N 个原子链构成衍生链的时间复杂度为 O(logN)。

3) 求切线时间复杂度：

类似于链合并中求公共切线，为 O(logN)。

4) 遍历时间复杂度：

从前面对于遍历算法的描述，容易看出遍历链上每一半边的时间都是常数的，即遍历一个链的时间正比于链的长度。

7.2.2 Hourglass

漏斗分为开与闭两种。在本程序中，我们只关心开的情况。

对于一个开漏斗而言，它由两条对角线，和连接两对角线的链(Chain)组成。

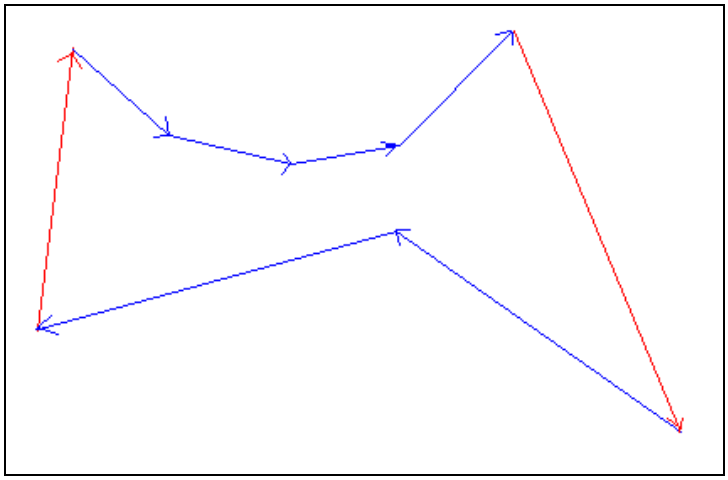


图 7.2.2.1 Hourglass

1. 数据结构：

漏斗的数据结构，即为记录两条左、右对角线的半边，和上、下链。

2. 算法

1) 构造：

类似链（Chain），漏斗也有原子和衍生两种情况。

对于原子情形，两条对角线同属于一个三角剖分，上、下链一个退化为点，另一个是三角形的一个半边。

对于衍生情形，即求两个子漏斗的合并。

2) 合并

漏斗的合并，首先判断合并后将消失的对角线，即分别在两个子漏斗中互为孪生半边的两个对角线。然后，确定每条子链合并的对象和顺序，通过前文介绍的链的合并算法，即可得到漏斗的合并。

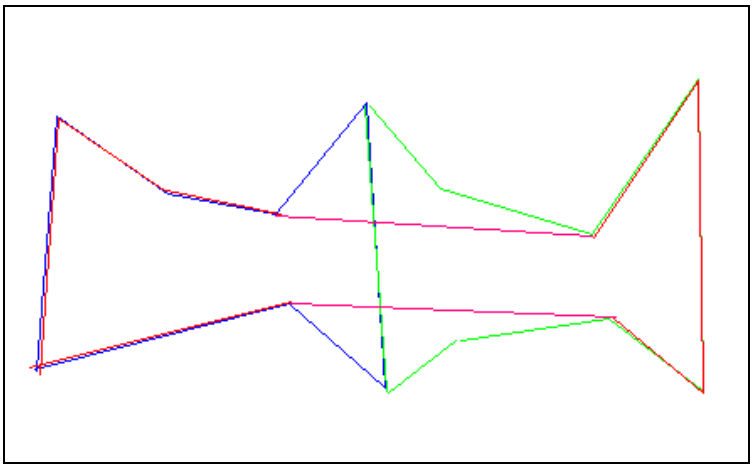


图 7.2.2.2 Hourglass 的合并操作

3) 开闭判断

我们只关心漏斗是开的情况，对于合并后闭的情况，需要把它们找出。

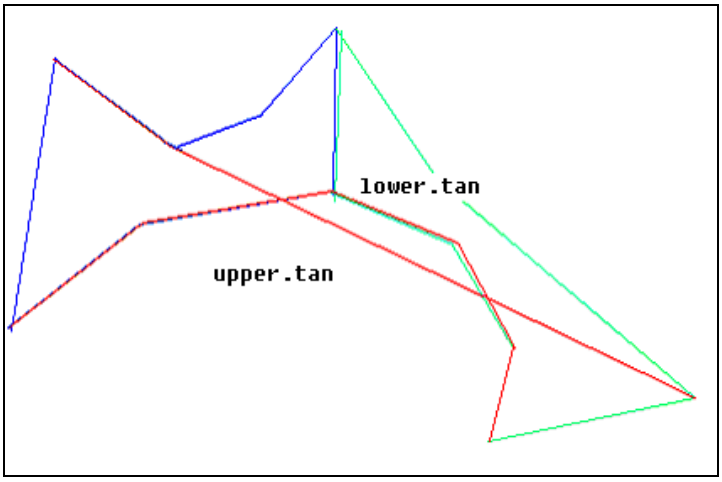


图 7.2.2.3 Hourglass 的开闭判断

漏斗是闭的，当且仅当上、下链中的切线 tan 或者相交，或者上切线在下切线下方。对于第一种情况，我们只要对两条切线进行线段求交，即可判断。对于第二种情况，考虑到合并对角线所在直线必然穿过上、下两条切线。只需要计算两切线与它的交点位置，就可以知道漏斗是否为开。

4) 求射线

求一点到漏斗的射线范围，即求解一点到上、下两链的切线。值得注意的是，求出两条切线后，要判断它们是否使新的类漏斗（一条对角线退化为点）为闭。

3. 复杂度分析

1) 空间复杂性：

空间复杂性主要来自于上、下链表，为 O(logN)。

2) 构造时间复杂度：

对于原子的，显然为 O(1)。

对于衍生漏斗。首先，判断孪生半边，O(1)。然后，构造上、下链表，根据上文分析，时间复杂度为 O(logN)。而判断开闭，时间为 O(1)。总时间复杂度为 O(logN)。

3) 求射线

对上、下链求切线为 O(logN)。开、闭判断为 O(1)。总时间消耗为 O(logN)。

7.2.3 Factor Graph

1 数据结构：

程序中使用 factor graph 来存储多边形中的对角线及它们之间的漏斗(hourglass)。

Factor graph 通过一棵二叉树实现，经过前面平衡化之后，该二叉树是平衡二叉树。它的中间节点存储一条对角线，叶子节点存储一个三角剖分后形成的三角形。

在程序中，只有那些互为祖先/后代的节点，它们存储的对角线可能进行漏斗构造。因此，在每个节点中，设置一组指针数组，用来存储它和它所有祖先的所含对角线形成的漏斗。需要说明的是，考虑到可能大量的点集中在少数三角形中，因此只有在需要构造漏斗时，才构造一个漏斗，并把它存储到数组中相应位置。然后，通过设置标记，使当它作为其它漏斗的子漏斗时，可以直接访问。同时还要记录该节点位于祖先节点的左子树，还是右子树。对于落在多边形中的某个点，首先通过点定位确定它的所在三角形。对于三角形所在叶子节点的所有祖先节点中存储的对角线，都需要和该点进行射线求解。

一个子节点对应的对角线，将把多边形分为两个部分，其中一侧对父对角线可见。在节点中需要记录这一方向，以便构造漏斗和计算射线。

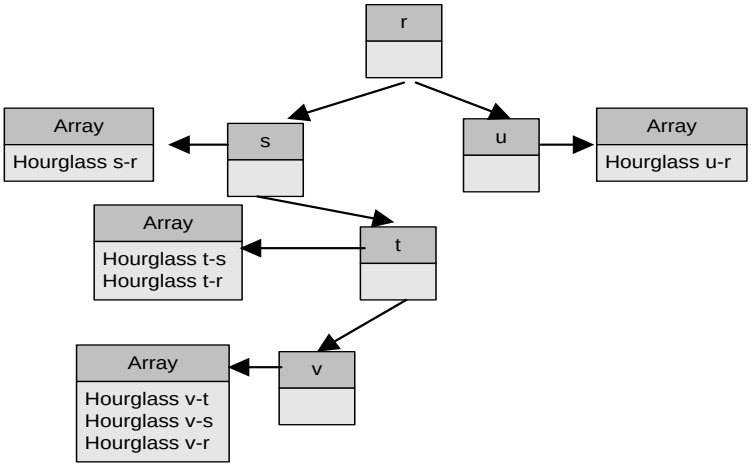


图 7.2.3.1 Factor Graph 的数据结构

2 算法

1) 求点到某条对角线的射线

对于已经定位到某个三角形中的点 p，需要求出它与它祖先节点所存储的对角线 l 的射线。根据节点数组中对应该祖先的元素，判断三角形 p 对对角线 l 是否可见，如果不是，根据数组中其他元素的方向和可见方向，判断出把 p 和 l 分开的最低的对角线，它必然是三角形中某边。求它和 l 构成的漏斗，并计算点对该漏斗的射线。对于可见情况，两条射线即 p 到 l 端点的射线。

2) 构造两条对角线之间的漏斗

求两条对角线 a,d 之间的漏斗，其中 a 是 d 的祖先。首先通过 d 中的数组，判断 a,d 是否可见，如果可见它们必然是某一三角形的两边，对应一个原子漏斗。否则，通过 d 中数组得到分开 a,d 的最高对角线 s，分别求它和 a,d 形成的漏斗，在把它们进行合并，就得到了 a,d 之间的漏斗。值得注意的是，对于点到对角线求射线和对角线之间求漏斗，我们都必须判断结果的开闭，剔除闭的情况。同时，对于一个漏斗，如果合并它所使用的某个子漏斗是闭的，它也将是闭的。

3 复杂度分析

1) 空间复杂性：

由于并非每个漏斗都必须求出，因此空间复杂度和点的分布相关。这里我们给出最坏情况下的空间复杂度。

每个节点都和它的祖先节点间有一个漏斗,最多有 O(NlogN)个漏斗，而每个漏斗最多占用 O(logN)的空间。因此，总的空间复杂度不会超过 $O(N\log^2 N)$。

2) 构造的时间复杂度：

对于构造 O(NlogN)个漏斗，而每个漏斗的构造时间不会超过 O(logN)，因此构造总时间不会超过 $O(N\log^2 N)$。

3) 点到对角线求射线的时间复杂度：

对于 M 个点求射线，每个点需要对它的不超过 O(logN)的对角线求射线，而每次求射线的时间不会超过 O(logN)。因此，总共消耗的时间不超过 $O(M\log^2 N)$。

7.3 贾云涛 2004211034

7.3.1 算法原理以及分析

1. 问题描述

对于下图的配置，判断点 a、b 关于对角线 AB 是否可见。

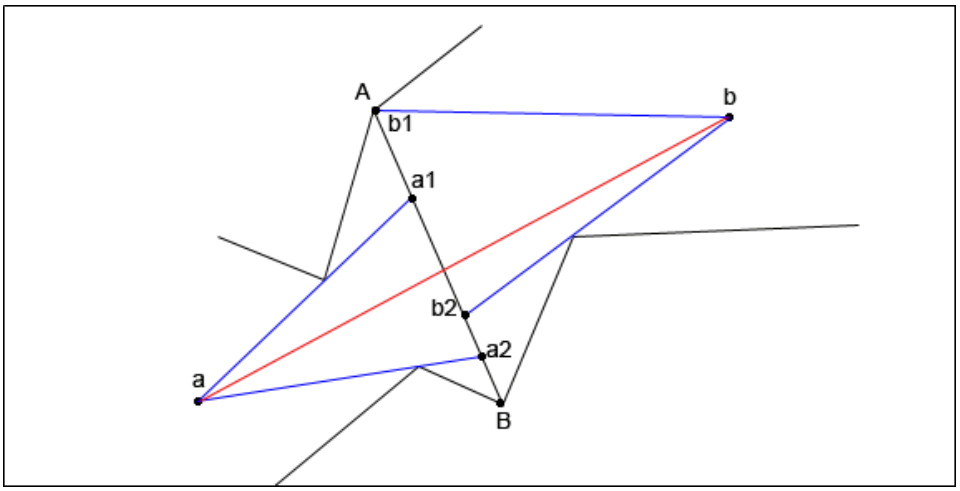


图 7.3.1.1 问题描述

2. 已知（向上接口）

1) Factor Graph：一棵包含所有代测点的二叉树。

2) Hourglass：提供待测点到某条对角线的可见区域。

3) DCEL：包含多变形和对角线的数据结构。

3. 分析

对于上图的配置，为了判断 a、b 两点关于 AB 这条对角线是否可见，我们首先要计算点 a 和点 b 相对于 AB 直线的可见区域（通过 Hourglass 的查询的得到），如图中蓝色直线所组成的楔形。a、b 可见当且仅当楔形 a_1aa_2 与楔形 b_1bb_2 有一条共同的直线，如图中红线所示。

在对偶空间中，楔形 a_1aa_2 可以表示为一条线段，假设为 La；楔形 b_1bb_2 对应的线段为 Lb。则如

果存在一条直线穿过楔形 a_1aa_2 和楔形 b_1bb_2，就意味着在对偶空间里面线段 La 和线段 Lb 存在一个共同的交点。

因而问题转化为下列问题：

1) 首先计算楔形区域在对偶空间中的线段。
   a) 分别计算直线 aa_1 和 aa_2 的对偶 $D(aa_1)$、$D(aa_2)$，并组成直线 La。
   b) 分别计算直线 bb_1 和 bb_2 的对偶 $D(bb_1)$、$D(bb_2)$，并组成直线 Lb。
   c) 算法复杂度为 O(mlogn)，m 为点的数目，n 为多变形的规模，logn 表示 factor graph 平衡三角剖分树的树高。只执行一次。
2) 然后计算所有关于此对角线的线段的交点并记录。
   a) 算法复杂度为 O((mlogm+k)logn)，logn 表示 factor graph 平衡三角剖分树的树高，mlogm+k 为线段求交的复杂度。只执行一次。
   b) 实现中采用的是由 Mulmuley 提出的最优的随机线段求交算法，复杂度为 nlogn+k。

如果点 a、b 关于对角线 AB 的线段存在交点，就表示 a、b 点相对于对角线 AB 可见；否则，不可见。

7.3.2 对偶运算

1 算法

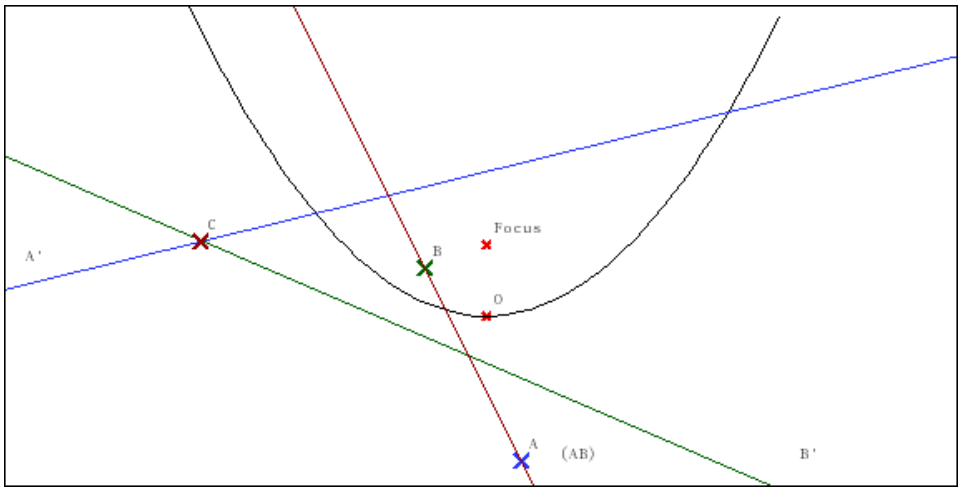


图 7.3.2.1 对偶变换

我们采用的是基于抛物线的平面对偶方法。

如上图所示：平面上点 A = (x_A, y_A)（图中蓝色点），我们定义它的对偶 dual A’为这条直线：$y = x_A\ x - y_A$（图中蓝色直线）。反过来，如果已知一直线 L 的方程式为 y = a x + b，则定义它的对偶为 (a,-b) 这个点。

但是简单的采用上述对偶算法会存在下面的问题：

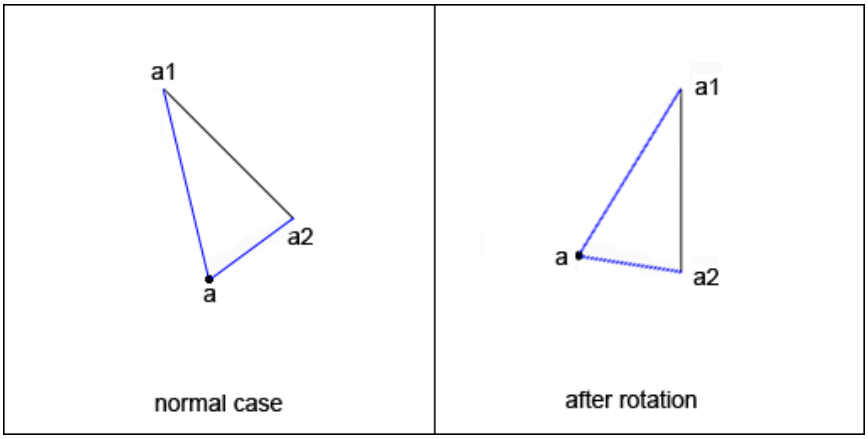


图 7.3.2.2 改进对偶变换

如上图中的左图所示，它表示通常的情况下，对角线并不是竖直的。在这种情况下，在 a_1aa_2 的楔形里面包括了一条竖直的射线，它对应的斜率为无穷，这表示它经过对偶变换之后为无穷远点，此时就无法求出该楔形对偶变换之后所对应的线段。

在这种情况下，我们的解决办法如上图中的右图所示，对整个 configuration 绕对角线的某一个端点作旋转，使得旋转后的对角线为竖直上下。此时任意点相对于该对角线的楔形可见区域中不再包括竖直方向的直线，从而可以计算出该楔形对应的线段。

批注 [Jordan3]:
这些知识，没有在期末考试第 7 题中得到运用，可惜

2 数据结构

2.1 对角线列表（DiagonalMap）

记录对多边形平衡三角剖分而得到的所有对角线（Diagonal），采用 std lib 的 hash map 实现。主要操作包括插入、查找、遍历。

2.2 对角线（Diagonal）

在计算可见图的时候，我们会把每一个定点的 360 度可见区域分成几个可见间隔。每一个间隔对应于某一条对角线，那么对角线里面记录的就是这些定点相对于它的可见区域（楔形）对偶变换之后的线段。即对角线里面记录的是一个线段列表（SegmentMap）。对角线的主要操作为插入线段。

2.3 线段列表（SegmentMap）

采用 std lib 的 hash map 实现，主要操作包括插入、查找、遍历。

3 复杂度分析

3.1 空间复杂度

在对一个定点的可见区域进行分割的时候，它只可能对在 factor graph 上面处于它父节点的对角线做贡献，所以一个定点的可见区域在所有的对角线的记录中最多只出现 factor graph 树高的次数，即 O(logn)次，因此总的空间复杂度为 O(mlogn)。

3.2 时间复杂度

对偶变换的时间复杂度为 O(1)，对 hash map 的插入和查找都可以常数时间内完成，因而总的时间复杂度取决于操作的次数。因为一个定点的可见区域在所有的对角线的记录中最多只出

现 0(logn)次，因为总的时间复杂度为 0(mlogn)。

7.3.3 线段求交

Mulmuley 于 1988 年提出了一个随机的增量式算法来解决线段求交问题。这也是我们实现中所采用的算法。一方面该算法的时间复杂度为 O(nlogn+k)，空间复杂度为 O(n+k)，已经达到最优；另一方面相比于其他最优的算法，该算法只需要相对简单的数据结构。

Mulmuley 的随机算法对于求交的线段有一个限制：不能存在竖直的线段。这个条件在我们的问题中自然的就满足了，因此也非常适合。

1 算法原理

算法在运行过程中始终维护了一个由线段和经过线段端点直到另外一条线段的竖直的 attachment 所构成的平面划分。每一个划分的平面利用围成它的所有顶点的一个逆时针的链表来标记。为了方便的对划分进行遍历，每一个顶点都有一个指针，指向在下一个划分中同样位置的顶点。以上为算法所需要的大致的数据结构。

对于输入的所有线段，算法首先对他们的所有端点排一个序，根据这个顺序去构造每一个端点的 attachment，从而给出一个初始的划分。接着对所有线段进行随机的插入操作。因为每一个线段的端点都有一个到穿过它的 attachment 的指针，这样随机的插入一条线段以后，我们可以很快的查找到该线段穿过的第一个划分，然后我们依次遍历该线段所穿过的所有划分，并把这些划分再分解，如果线段穿过了一个原划分中的一个 attachment，那么该线段就切断了这个 attachment，此时需要进行划分的合并。

该算法的伪代码描述为：

```
////////////////////////////////////////////////////////////////////////////////////////
/////          Mulmuley's Line segment intersection Algorithm
/////
{
    Sort endpoints of all segments and create initial partition
    For all segments s in random order do
        { insert segment into partition }
```

```
        While current point is not the right endpoint of s do
            Create new vertices above and below the entry point of s
            (u, v) = vertices where s leaves the current face
            If s passes through attachment above or below associated endpoint then
            Split current face at (u, v)
            Contract attachment
            Merge faces above or below attachment
        Else
            Split current face (u, v)
        End if
        Transition to next face s passes through
    End while
 End for
}
```

2 数据结构

2.1 线段（Segment）

线段为求交的输入数据，它主要由两个端点（EndPoint）a、b 组成，而且要求 a<b。主要操作为相交判断，相等判断，线段比较，对偶变换等。

2.2 端点（EndPoint）

端点除了维护了高精度的横纵坐标以外，还保存了该端点对应的竖直方向的辅助线（Attachment），另外该点还保存了到自己构成的线段的指针。主要操作包括：端点比较，相等判断，对偶变换等。

2.3 辅助线（Attachment）

由以上的算法说明可知，辅助线主要用来根据线段端点对平面进行切分。它是穿过线段端点的一条竖直线段，上下端点为该线段与另外一条线段或者所有线段的包围盒（BoundBox）的交点。因而它由一个端点（穿过）坐标和首尾两个剖分顶点（Vertex）指针组成。

2.4 剖分顶点（Vertex）

剖分顶点表示的是平面所有划分的顶点位置，因而它本身包括一个位置坐标。另外算法中涉及到划分的遍历，因而剖分顶点中还包括指向前一个划分和后一个划分的指针。

2.5　划分遍历（Traversal，方法，非数据结构）

主要操作包括：根据输入线段，快速找出左端点所在的划分；计算输入线段离开当前划分的交点；根据与上一个划分的交点找出下一个划分等。

2.6　几何操作（Geom，方法，非数据结构）

主要操作为：判断点与直线关系；直线与直线关系；线段与线段求交；点是否在划分内等操作。

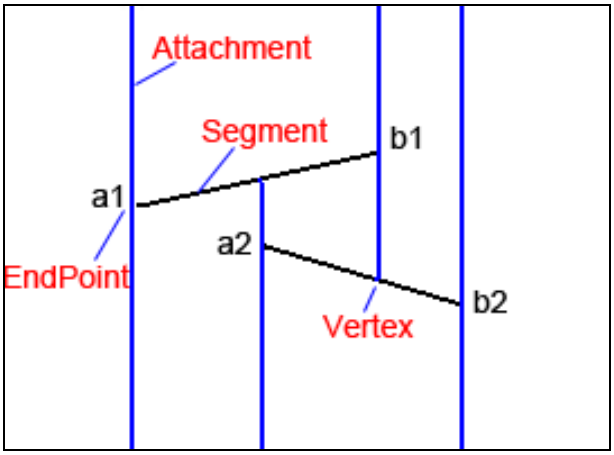


图 7.3.3.1 平面划分的各个元素

3　复杂度分析

3.1　空间复杂度

算法空间占用主要包括：线段（Segment）＋端点（Endpoint）＋辅助线（Attachment）＋剖分顶点（Vertex）。

假设线段数目为 n，则前两者空间复杂度为 O(n)；辅助线由穿过线段端点的直线与上下线段或者包围盒求交而得，因而空间复杂度为 O(2n)=O(n)；剖分顶点为辅助线（线段）的上下端点，空间复杂度为 O(4n)=O(n)。所以总的空间复杂度为 O(n)。

由上一节对偶运算的复杂度分析可知，线段的数目为 O(mlogn)，所以在可见图计算的算法实现中，线段求交的空间复杂度为 O(mlogn)。

3.2 时间复杂度

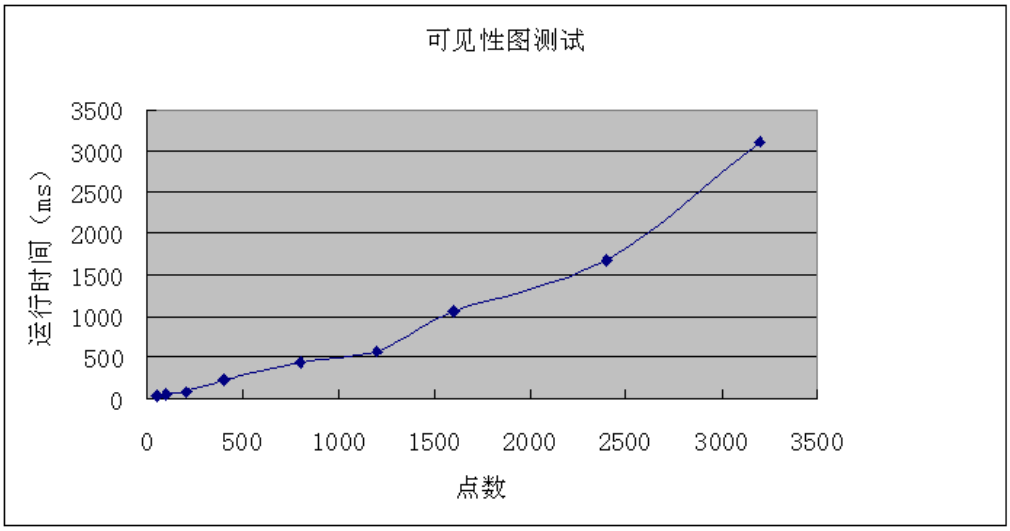
该算法为一个随机算法，通过统计分析的论证，该算法的时间复杂度为 O(nlogn)，这已经达到了线段求交复杂度的理论下界。

我们简单的分析一下：算法开始利用排序构造了一个初始划分，复杂度为 O(nlogn)，接着算法随机的插入所有 n 条线段。每条线段都依次遍历它所穿过的所有划分，每次遍历的操作主要包括：划分的顺序定位，线段离开划分交点的计算，上述操作都可以在常数时间内完成。因此算法的时间复杂度为：O(nlogn)+n*(遍历划分的平均个数*O(1))，即时间复杂度取决于插入每条线段所遍历划分的平均个数。通过概率分析，随机的插入一条线段所遍历划分的个数的期望值为 logn，因此算法总的时间复杂度为 O(nlogn)。

结合输入直线的数目为 O(mlogn)，所以在可见图计算的算法实现中，线段求交的时间复杂度为 O(mlogn(logm+loglogn))。

批注 [Jordan4]:
后面的测试数据，并未做这样的回归分析

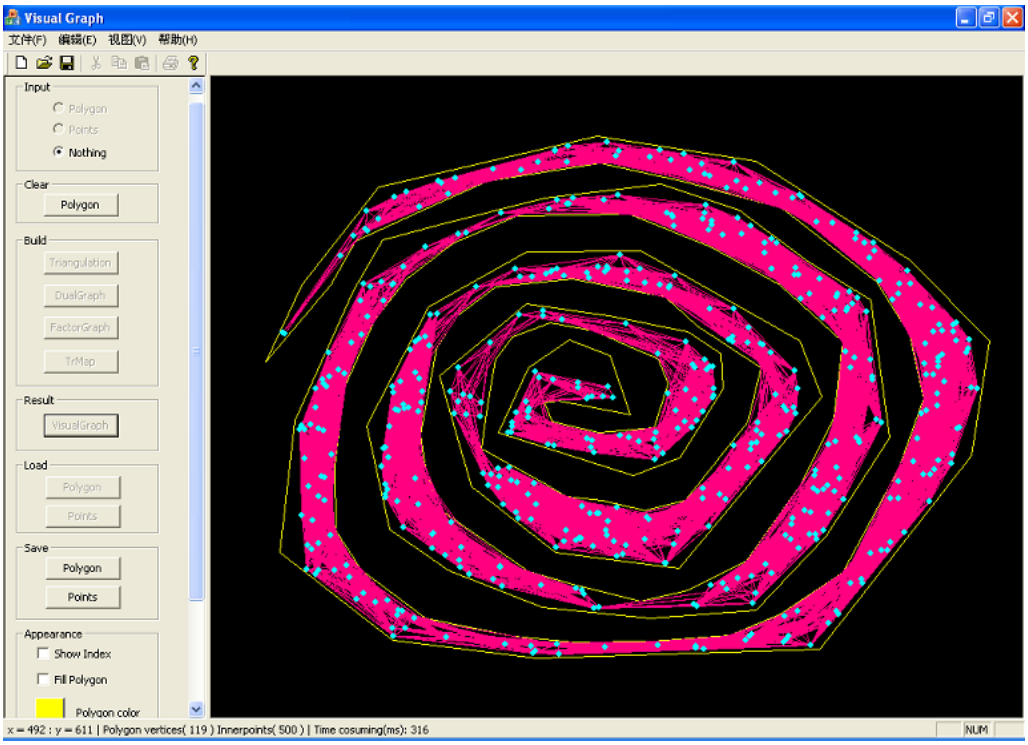
8. Result:【测试结果】



点数	50	100	200	400	800	1200	1600	2400	3200
时间/ms	26	45	88	221	444	571	1053	1683	3117

注：　　多边形以./data/test_pg13.poly 为例

批注 [Jordan5]:
该多边形只有 119 个顶点



9.　　Future Work：【进一步工作】

进一步的工作包括：

1. 基于简单多边形 Q 内部点集 P 的带距离约束的可见图的计算。
2. 基于简单多边形 Q 内部点集 P 的鲁棒强可见图的计算。

10．Reference:【参考文献】

[1] B. Ben-Moshe, O. Hall-Holt, M. J.Katz, Computing the Visibility Graph of Points Within a Polygon. SCGA04 2004.3

[2] L. J. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. J. Comput. Syst. Sci., 39(2):126{152, Oct. 1989.

[3] L. Guibas, J. Hershberger, 3. Leven, M.Sharir, R. Tarjan: "Linear eime algorithms for visibility and shortest path problems inside simple polygons," Proceedings of the 2nd ACM Symposium on Computational Geometry (1986), pp. 1-13.

[4] K. Mulmuley. A fast planar partition algorithm, i. In Proc. 29th Annu. IEEE Sympos. Found. Comp. Sci., 10, pages 580–589, 1988.

[5] K. Mulmuley. Computational geometry: an introduction through randomized algorithms. Prentice-Hall, 1994.

[6] S. Kanthak. Implementations of Algorithms For Line-Segment Intersection.

[7] L. Guibas, J. Hershberger, 3. Leven, M. Sharir, R. Tarjan: "Linear eime algorithms for visibility and shortest path problems inside simple polygons," Proceedings of the 2nd ACM Symposium on Computational Geometry (1986), pp. 1-13.

[8] M. Overmars and H. van Leeuwen: "Maintenance of configurations in the plane," Journal of Computer and System Sciences, vol. 23 (1981), pp. 166-204.

[9] P. K. Agarwal. Partitioning arrangements of lines: II. Applications. Discrete Comput. Geom., 5:533-573, 1990.

[10] T. Asano, T. Asano, L. J. Guibas, J. Hershberger, and H. Imai. Visibility of disjoint polygons. Algorithmica, 1:49-63, 1986.

11. Mail：【联系方式】

11.1 贾云涛：

jiayuntao00@mails.tsinghua.edu.cn

11.2 杨永亮：

yangyongliang00@mails.tsinghua.edu.cn

11.3 张弛：

zangcy00@mails.tsinghua.edu.cn