《计算几何》**Project** 之报告

Folding and One Straight Cut on Simple Polygon

单　　　位：　国家 **CAD** 工程中心

指 导 教 师：　邓 俊 辉

作　　　者：　丁 俊 勇（**006128**）

朱 旭 平（**006130**）

郭　　镔（**006134**）

问题简介及选题目的

Folding-and-Cutting Problem 源于日本的一种民间艺术——折纸手工（Origami），它所要解决的问题是：任意给定一张纸和一个图形，如何通过一定方式将给定的纸折叠得到一个平面图案，在该图案上面沿着某一条直线剪一刀，把剩余的一部分展开得到的图形正是给定的图形。这个问题从提法上看很简单，其实不然，从我们所了解的资料中可以看出，对于该问题的研究有很多方面。例如问题的可解性研究，折线图案（Crease Pattern）的生成，折叠过程的模拟等等。综合所掌握的资料，我们选择该问题的一种比较简单的情况作为该次《计算几何》Project 的研究题目，一方面考虑到这个问题比较有意思，另一方面，有难度的题目才具有挑战性。

问题描述

我们对上面 Folding-and-Cutting 问题的输入作了一个限定，即我们只处理简单多边形的情况。描述如下：

☺ 输入：简单多边形 P

☺ 输出：给出折叠方案（Crease Pattern and Moutain_Valley Assignment）

举一个具体的例子如下：如图 1 中，在给定的纸张上面给定一个五角星，通过 1、2、3 三个步鄹的折叠，如 4 用剪刀剪一刀，我们就可以得到一个五角星的图案。

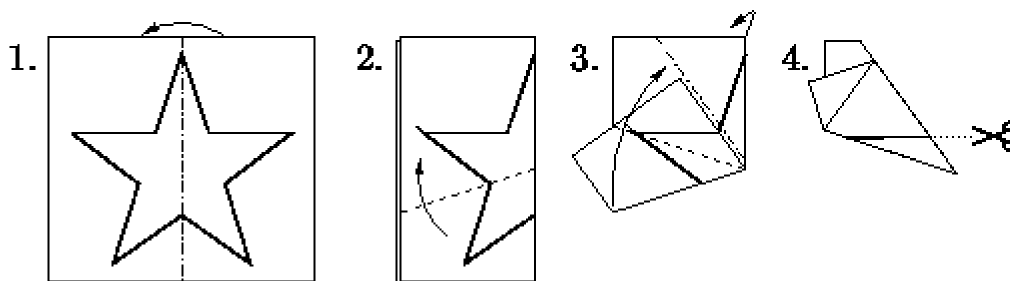


图 1　如何折叠一张方形纸，通过一刀剪能够得到一个五角星形状

算法及实现

Part One —— Skeleton 的构造

Skeleton 是平面直线图形 G 的一种几何结构，这个概念的提出也有一段时间了，而且在很多方面都有广泛的应用，折纸问题（Folding）只是它的一个应用之一。Skeleton 是一个和“medial axis”非常类似的概念，对于凸多边形来说，它们是等价的。它不是用一个距离函数来定义的（not a Voronoi－diagram-like concept），而是通过一种收缩过程来定义。设想，

多边形 P 的各条边界通过一种自平行的方式，分别以相同的速度向多边形 P 的内部和外部收缩和扩张，这样，多边形的各个顶点将沿着各个顶角的角平分线移动，这些移动的轨迹构成的图形就称为该多边形的“Skeleton”。由于这种特殊的定义方式，使得一些计算 Voronoi－diagram-like concept 的经典算法，如分治法（divide-and-conquer）和类似归纳的方法（inremental insertion）都无法利用。可能是这个问题的计算和实现都比较复杂（或者我们掌握的资料还不够多），我们查阅到的大部分文章，都侧重于讨论 Straight Skeleton 的特性和应用，对于算法的设计却很少提及。有一篇文章**《Straight Skeleton for General Polygonal Figures in the Plane》**,O.Aichholzer and F.Aurenhammer，中提出了一种算法的思想，综合考虑其直观性和算法复杂性，我们决定试一试。

从 Skeleton 的定义我们可以看到，理论上来说，它的构造过程可以通过模拟来实现，即给定一个合适的时间间隔 Δt，每过 Δt，我们分析 G 中各个顶点的运动和位置，看有没有有效的事件发生（如是否有两个顶点运动到了同一个位置，是否有一个顶点运动到了 G 中的某一条边上面等等），如果有事件发生，则对拓扑结构作相应的改变（如顶点的位置、运动方向等），如此循环，直至不可能再有有效的事件发生为止。这个过程直观上想简单明了，可是从算法的角度上考虑，其精确性和有效性都不能够构成一个可行的算法（如 Δt 究竟取多大？）。上面的文章中提出的思想就是把这个模拟的构造过程量化到了几个时间点上，变成了一种可行的算法描述。它的**总体思想**大致是这样的：***首先，***对于包含输入直边图形 G（我们这里指的就是简单多边形）的整个平面做一个有限制的三角剖分（就是可以对该剖分加上哪些边必须保留，哪些边不能连等限制条件）；***然后，***求出输入简单多边形各个顶点上向内和向外的角平分线方向，将原始多边形链复制两条，一条上的所有顶点运动方向赋值为相应顶点的内角平分线方向（该链称为内运动前沿（Inner Wavefront）），另一条上的所有顶点运动方向赋值为相应顶点的外角平分线方向（该链称为外运动前沿（Outer Wavefront））；***接着，***在前两步结果的基础上，每一个三角形的三个顶点，依据三角形在当前 Wavefront 的内外情况，可以从 Wavefront 上对应的顶点各自得到一个运动矢量（包括长度和方向），这样，每一个三角形的三个顶点的运动明确了，就可以建立一个按照消亡时间递增排队的三角形队列，我们可以很容易的分析得到，只有这些三角形的消失，才会引起前面模拟过程中提到的有效事件的发生（也就是可能产生了 Skeleton 结构中的 Skeleton 点（Skeleton Node）），这样就把整个模拟的连续过程量化到了这些三角形消失的时刻点上；***最后，***就是根据这个三角形优先队列进行循环，每消失一个三角形，则对前面的 Wavefront 做一定的修改（如合并点、分裂等等），如果有新的 Skeleton Node 产生则记录，更新三角形队列，…… 循环直至三角形队列中的所有三角形都不消失为止。下面将对以上各个步鄹的详细情况和实现中遇到的一些问题做一个详略得当的阐述。

☆ ***原始多边形的三角化***

这里需要注意的是，三角化不仅仅是对多边形内部进行，而是对整个平面进行，所以这里的三角形和我们平时所说的三角形不太一样，意义更为广泛。从《计算几何》所

学的知识，我们可以定义三角形为：平面上任意三条直线的某半空间（halfspaces）交成的 polyhedron，如果该 polyhedron 是封闭的（即为一个 Polytope），我们称之为封闭三角形（Bounded Triangle），这就是平时意义上所说的三角形；如果区域不封闭，则称之为开放三角形（Unbounded Triangle）。

三角化算法可以任意选择，我们尝试过用经典的 Delaunay 三角化算法，但是由于编程中遇到一些内存上的问题，考虑到时间紧迫，而该算法只是整个工作中的一小个步鄹，所以我们退而求其次，采用了另外一种复杂度稍高，但是效果很接近 Delaunay 的三角化算法，而且可以对三角化结果作一定的限定（对于 Delaunay 三角化，我们整理了一些相关的英文资料，在理解的基础上翻译成了中文，放在附录里面）。**算法描述**如下：

第一步： 把要三角化的所有点的点集记为 P（n 个点），求出 P 中所有点两两之间的距离用二维数组 Dis[n][n]表示。求出距离的最大与最小值，记为 Max、Min.。设定搜索范围 r_min、r_max。设定搜索的步长 p 为(Max-Min)/m。其中 m 在三角化中为常数，由点的个数及点集的特征可自己指定。m 越大，三角化效果越好，但速度会变慢。第一次可取 r_min=Min-p；r_max=Min 即可；

第二步： 对每三个点进行判断，当三点的两两距离都小于等于 r_max，且至少有一条边的距离大于 r_min 时，如果三点不共线并且连成的三角形不与任何已有的边(用数组 e[]记录)相交并且没有点在三角形内部（或边上），那么把这三个点连成三角形。执行如下操作：1、把这三个点的序号加入用以记录已构成三角形的数组 Tri[]。2、如果这三角形的边以前还未连接的（用二维数组 E[i][j]记录边是否已连接），把边加入数组 e[]中，同时把 E[i][j]设为 True，表示该边已连接；

第三步： r_min=r_min+p;

r_max= r_max+p。

如果 r_max<=Max,转第二步；否则结束。

通过这种算法，在 Tri[]数组中就保持了所有三角化得来的三角形。这里，如果要想规定某些边必须连接，则可以在三角化前把这些边加入数组 e[]中。如果要想规定某些边不可以连接，只需在算法中加入数组 ne[]来先保存不可连接的边，在第二步时保证三角形的边都不在 ne[]数组中即可。

该三角化算法的特点，决定了三角化后的结果其实是对原始多边形顶点的凸包的一个三角化结果，所以为了加入那些 unbounded 的三角形，我们对原始多边形顶点求了一个凸包，采用的是 Graham Scan 算法（这里不再详细阐述）。

☆ *内外 Wavefront 的生成和三角形消亡优先队列的生成*

有了上面的三角化结果，下一步要建立该算法的核心结构——三角形消亡的优先队列。首先，将原始多边形链复制两份（Inner Wavefront 和 Outer Wavefront），分别作

为初始的运动状态；然后在此基础上，每一个三角形的三个顶点依据其在多边形的内外情况各自都有一个运动矢量；通过分析各个三角形顶点的运动矢量，我们便可以计算各个三角形消亡的相关信息。由于三个顶点的运动方向多种多样，如果采用枚举分析的方法，要想全面的概括所有可能的情况是很困难的，这里我们把该问题转化为一个代数问题来求解。

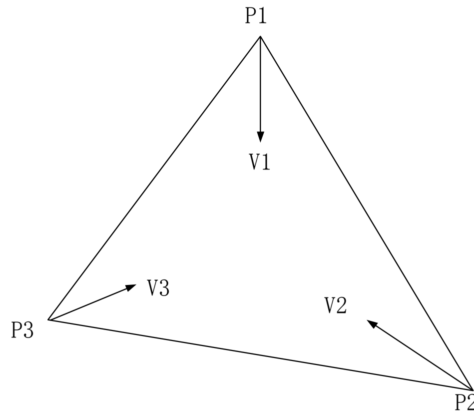


图 1　三角形运动形态分析

如图，顶点 Pi 的运动方程为：$P_i^{'} = P_i + t * V_i$ ，其中，Pi 为顶点的初始位置，t 为时间，Vi 为顶点 Pi 的运动矢量（包括大小和方向）。当三个顶点共线时（如果可能的话），三角形消失；如果无论怎样运动，三个点都没法共线，那么该三角形永远不消失。据此，可以列出时间 t 的二次方程如下形式：$at^2 + bt + c = 0$，通过分析该二次方程解的情况就可以判断三角形的消亡情况并精确的求出消亡的时间。

☆ *三角形形态的变化分析以及 Skeleton 结构的生成*

通过分析，三角形消亡时可能发生三种事件：Flip Event、Edge Event 和 Split Event（参见图 2）。

（1） Flip Event：当 Wavefront 上的某个顶点穿越一条三角形的边（s）并且该边不是 Wavefront 上的边，则发生该事件；
处理方法：将 s 移走，加入边 t，这种情况不产生 Skeleton 的点；

（2） Edge Event：当 Wavefront 上的两个顶点运动到同一个点时发生该事件；
处理方法：产生新的 Skeleton 点（即该二点碰头的点），将 Wavefront 上的这两个点合并，并且更新三角形消失队列；

（3） Split Event：当 Wavefront 上的某个顶点穿越一条 Wavefront 上的边（e），则发生该事件；
处理方法：将被切分的边切分成 e'、e'',原 Wavefront 分裂成两段。

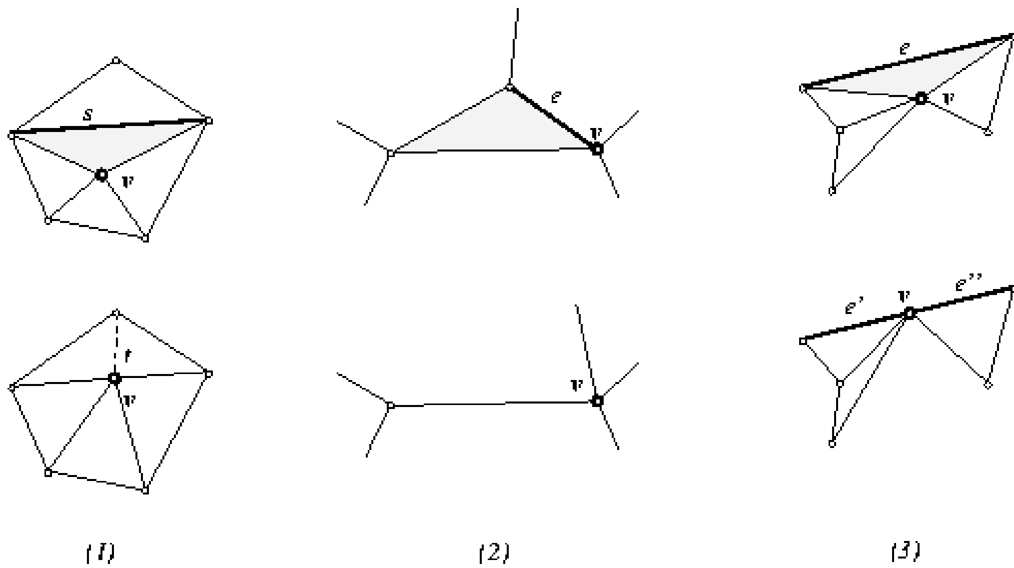


图 2　　三角形消失时发生的事件

（1）Flip Event　　（2）Edge Event　　（3）Split Event

通过逐个的分析三角形消亡队列中的三角形消失情况对 Wavefront 做相应的改变，并一步步产生 Skeleton 点，直至消亡队列中的所有三角形都不消失。

Part Two —— Perpendiculars 的生成

1. **Perpendiculars 的定义与意义**

(1)定义

- 在得到 skeleton 之后，对于平面内的任意一个点 ***p***，都有可能求到一条 Perpendiculars，称这个 Perpendiculars 与点 ***p*** 相关。
- 一条 Perpendiculars 由一些线段和射线组成，其中的每一条线段或射线称为一条 Perpendicular edge，每一条 Perpendicular edge 都有一个 skeleton face 与之相对应。
- Perpendicular edge 与 Perpendiculars 的关系：对于一个点 ***p***，它属于一个 skeleton face ***f***（如果 ***p*** 在这个 skeleton face 的边界上，那么它将有可能属于两个 skeleton face，随便选择一个即可），令 ***l*** 为过点 ***p*** 且与 ***f*** 内的 cut edge（定理保证该 cut edge 必然存在）垂直的直线，令 ***m*** 为 ***l*** 与 ***f*** 求交所得的线集中（可能有多条射线，线段）与点 ***p*** 直接相连的那一条线。那么与点 ***p*** 相关的 Perpendiculars 包含线 ***m*** 以及和 ***m*** 的端点相关的 Perpendiculars。
- 与一个 skeleton vertex 相关的 Perpendiculars 称为 real perpendiculars

(2)意义

我们最后要求的 mountain-valley 分配只可能发生在 skeleton edge 或者 real perpendiculars 上。具体的分配方法见后。

2. 本实验中 Perpendiculars 的生成算法

(1) 概述

Perpendiculars 的定义实际上是一个递归性质的定义。可以把它改造成递推的定义，本实验中我采用递推的算法一步一步的求 real perpendiculars。

(2) 总体流程

设置一个工作栈 S，S 中的每一个元素记录着一个点和一个 skeleton face，表示将要求出与这个点和这个面相关的 perpendicular edge。

a. 取一个没有处理过的 skeleton vertex，把它的信息和所在的 skeleton face 压入 S（一般有多个 skeleton face，分别压入）。

b. 如果 S 非空，则从 S 中取出一个元素，用其中的信息求出一条 perpendicular edge，如果该 edge 存在，那么进一步看该 edge 与 skeleton face 的交点，把这些交点信息和这些交点所处的其他 skeleton face 压入 S。加入这些信息的过程称为 AddNode 过程。
   如果 S 为空，表明已经求完一条 Perpendiculars，输出，退出。

c. 转 b

(3) 子算法

a. skeleton face 的求出

从一条 skeleton edge 出发，一直左旋或右旋知道转回自己，从而得到一个 skeleton face。在生成 skeleton face 的过程中，我们记录了大量的相关信息到相关的点线面结构中，比如一个 skeleton face 中包含的 cut edge 地信息等。

b. 从一个工作栈元素求出一条 perpendicular edge

首先作出过点的垂线，然后使用线面求交算法，这里我采用的是扫描线算法的变形，因为我不需要求到所有的交集，只需要与点相连的那一段。

c. AddNode 过程

判断一个点属于哪几个 skeleton face，根据我们定义的很好的点线面数据结构很快可以得到，然后剔除掉那些已经得到 perpendicular edge 的 face，即可加入工作栈。

注意：本过程中我们进行了是否会形成 circular corridor 的判断。若有 circular corridor 的话，本算法将无法求解出最后的 mountain valley 分配，所以如果判断出存在 circular corridor 的情况，我们将报告错误。

当我们认为应该加入工作栈的元素实际上以前已经加入过的时候，

实际上就是circular corridor的情况，此时若不报错将会死循环。

3. 疑难解决——无限范围的定义与有限范围计算的问题

正如在开题时提到的，纸张大小的设定，具体纸张的大小不是无限的，事实上，对于纸的大小只要求它能够包含输入的多边形。

而且事实上，我们的计算能力也只能局限在有限的一个范围，我们不会计算整个无限平面内的信息。对于 skeleton edge，由于 O(n)的关系，可以求出整个无限平面内的 skeleton edge；但是对于 real perpendiculars，它往往是发散的，而不是收敛的，也就是说，一个平面内往往会有无数的 perpendicular edge，我们不可能都求出来。

后面的 mountain valley 分配算法可以看到，有用的 perpendicular edge 只是那些落在我们设定的纸张内 edge。

假设纸张是一个矩形R。问题在于对于这样一个有无限个perpendicular edge的情况，我如何在使用上面的递推算法求perpendicular edge时知道：从现在开始再往下计算的所有perpendicular edge都不会和我们设定的纸张R相交，从而递推可以终止了。

这里，我证明了一个定理。

【定理 1】使用上面的递推算法计算一条 real perpendicular 的时候，如果先后两次计算一个 skeleton face 内得到的 perpendicular edge，以及这过程中计算过的 skeleton face 内得到的 perpendicular edge,都没有和我们设定的纸张 R 相交，并且，后计算得到的 edge 在先计算得到的 edge 的外面，则以后计算的 skeleton face 内得到的 perpendicular edge，也不会和 R 相交。

具体严格证明过程略。主要思路是使用反证法，假设以后有一条 perpendicular edge 落到 R 内，由于后计算的 edge 在先计算的 edge 的外面的情况将会在以后一直得到保留（很容易由参考文献中 perpendiculars 的性质推论得到），所以这条落在 R 内的 perpendicular edge 将会和另外一个由几条由 perpendicular edge 组成的折线段相交，且交点不是这条 edge 的端点。这将会违背 perpendiculars 的性质，从而导致矛盾。

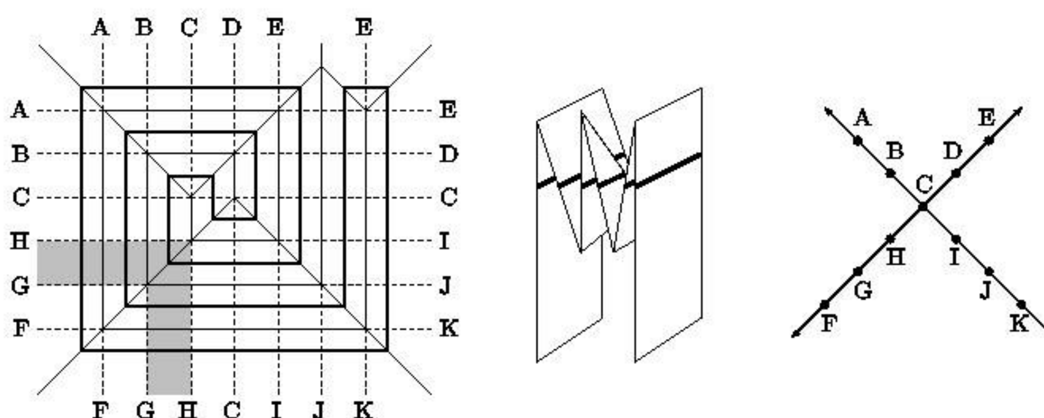
上面的定理保证了我们可以计算有限范围的 perpendicular edge 之后，就可以得到所有落在纸张 R 内的 perpendicular edge。实际的算法中，我们设了一个计数器，每当求出与 R 有交的 perpendicular edge 时，计数器就加 1。在每个 skeleton face 中，记录下了刚刚经过这个 face 的时候的计数器的值，这样我们只要比较计数器与上次记下的 face 内的计数器的值是否相等，如果相等的话，我们再做一些别的简单的判断就可以确定是不是有必要继续计算下去了（就是说是不是需要继续调用 AddNode 过程了）。

Part Three —— Folding

到现在为止，我们已经得到了 Skeletons 和 Real Perpendiculars,下面的工作就是要判断它们到底如何折叠，折叠方式不外乎三种，向外折（mountain）、向内折（valley）或着不折，所以 Folding 的工作就是要确定每一条边的折叠方式。

首先，对于 skeleton edge，是必须要折的，要么是 mountain,要么是 valley。具体确定规则可参考论文①的 4.1 小节有关 skeleton edge 的 mountain-valley assignment 部分，对于本试验所处理的简单多边形可以简化处理方法，即对于所有凸角顶点所发出的 skeleton 都可以定义为 mountain,相反对于凹角顶点所发出的 skeleton，就定义为 valley；对完全在多边形内部的 skeleton 定义为 mountain，完全在多边形外部的 skeleton 则定义为 valley。这样就可以满足论文中所提出的规则。

下面就该确定 Real Perpendiculars 的折叠方式了。首先提出一个树的结构,对于没有环形 corridor 的图来说，如果将每个 Real Perpendicular 看成树中一个结点的话，那么相邻两条 Real Perpendicular 所夹的 corridor 就是对应两个结点的边，如下图所示：共有 A、B、C…J、K 共 11 条 Real Perpendicular,对应于右边树模型中 11 个结点，左图中 H 和 G 之间的 corridor 用深色区域画出，对应于树中的 HG 边。其实，如果我们定义纸的平面为 xy 平面，那么我们将左图折叠成三位模型后，从 z 轴方向所看到的就应该是右图中的树模型，也就是说，同一条 Real Perpendicular 在空间中应该折成一条垂线，这样从 z 方向看就是一个点，而 corridor 也就被折成一条边，而且双边的 corridor 对应于树中一条线段，单边的 corridor 对应于一条射线，如右图中 A、E、F、K 四个点向外发出的箭头，这里称为叶子。

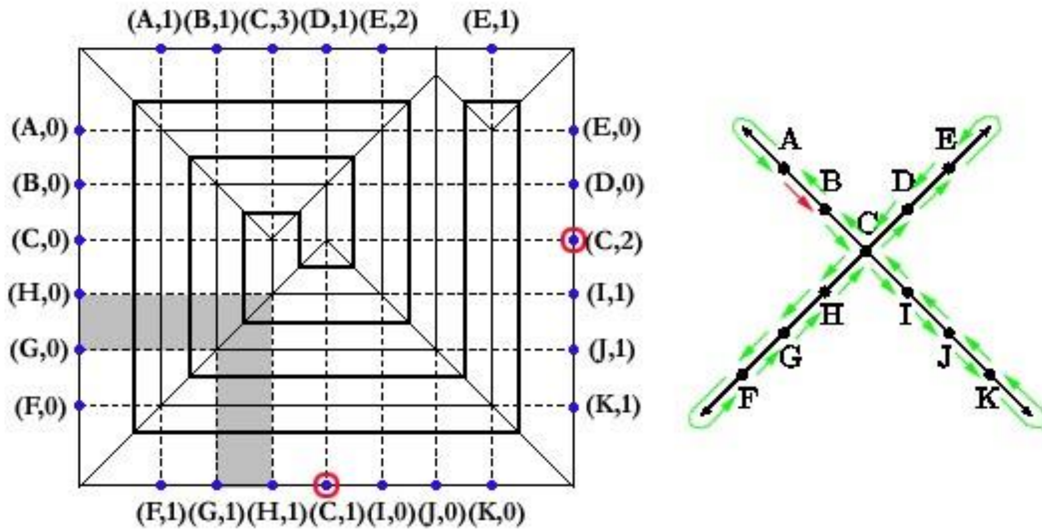


有了这个树的模型，我们就可以直观地得出折叠方法，即在平面上将一个树折成一条直线，我们可以以任意一点为根结点，然后向下拉直。例如：我们以 A 为根，那么 C 就有三个子树，只要把这三个子树压成一条直线即可，因此我们需要将角 HCI 和角 ICD 变成零，而对应于左图就是要把 corridor HC 和 corridor CI 之间的一部分 Real Perpendicular 进行折叠。可以看出，选择不同的根结点所得出的折叠方法是不同的，即解法不唯一，不同的解法需要折叠的 Real Perpendicular 也不同。

下面我们讨论如何得到树模型，以及选定解法后，如何具体确定每段 Real Perpendicular 的 mountain-valley。

首先给出生成树的算法：

初始化：由于我们要考虑的折叠平面是有界的，那么对于任意一条 Real Perpendicular，如果它不是环形的，它一定会与边界有交点。同时由于每条 Real Perpendicular 都是从一个 Vertex 发出的几条 Perpendicular，而每一条 Perpendicular 都应该与边界有交点，它可能多次出入边界，而最终不会留在区域内部，所以我们取它与边界的最后一个交点，一定是从内到外与边界相交，我们把该交点用一个二元组表示（I,i），其中 I 表示 Real Perpendicular 的标号，而 i 表示从 Vertex 发出的第 i 条 Perpendicular，如下图所示。



任取一点作为起点（根结点），如（A,0），顺时针方向扫描边界上的所有点。

```
i       生成新结点 TreeNow=A;
        Pnext=Proot=(A,0);
ii      do
        {
                Pnow=Pnext;
                Pnext=Pnow;
                If(Pnow 的 I 值未出现)
                {
                        生成新结点 I ;
                        将 I 加为 TreeNow 的子结点；
                        TreeNow＝I;
                }
                else
                {
                        if(Pnow 的 I 值＝Pnext 的 I 值)
                        {
                                生成叶子结点加为 TreeNow 的子结点；
                        }
                        else if（Pnow 的 I 值＝＝TreeNow 的父结点）
```

```
            {
                TreeNow=TreeNow->pFather;（回溯）
            }
            else
            {
                error(有环形的 perpendicular 存在)；
                return；
            }
        }
    }until(Pnext=Proot)
```

该算法扫描边界上所有的点，从而生成以起点为根的树，起扫描过程如右图所示，其中红色箭头为起始点。

有了树的模型后，就可以很容易的选出需要折叠的结点，即所有子结点数大于 1 的结点，该例中只有点 C，而对应原图中，需要折叠的点应该是（C，1）和（C，2）。这样，我们就需要遍历整个树，然后找到大于等于两个子的结点，再根据子结点的相邻关系，找出原图中需要折叠的点，这样就相当于我们从原图中先生成一棵树，再根据树来标记原图中的部分点，这样显然很繁琐，完全可以把这两步简化成一步执行。即当我们的树中新生成一个子结点时，看看父结点是否已经有子，如果有，就标记当前点 Pnow 为 Valley，这是因为需要把该点两边的 corridor 折到一起。另外，还有一种特殊情况，即无子结点，这样的结点对应与原图实际上是一个 perpendicular 的两边为同一个 corridor，显然应该标记为 Mountain,这样原来的算法就修改为：（其中粗体字为新加内容）

```
i   生成新结点 TreeNow=A;
    Pnext=Proot=(A,0);
ii  do
    {
        Pnow=Pnext;
        Pnext=Pnow;
        If(Pnow 的 I 值未出现)
        {
            生成新结点 I ;
            如果 TreeNow 已经有子,标记 Pnow 为 Valley；
            将 I 加为 TreeNow 的子结点；
            TreeNow＝I;
        }
        else
        {
            if(Pnow 的 I 值＝Pnext 的 I 值)
            {
                如果 TreeNow 已经有子,标记 Pnow 为 Valley；
                生成叶子结点加为 TreeNow 的子结点；
            }
            else if（Pnow 的 I 值＝＝TreeNow 的父结点）
            {
                如果 TreeNow 已经无子,标记 Pnow 为 Mountain；
```

```
            TreeNow=TreeNow->pFather;（回溯）
        }
        else
        {
            error(有环形的 perpendicular 存在);
            return;
        }
    }
}until(Pnext=Proot)
```

新算法运行后，将（C，1）、（C，2）两点标记为 Valley。

这样，我们以这些被标记为 valley 的点为起点，延 perpendicular 回溯，交替标记每段 perpendicular edge 为 valley、mountain、valley、mountain...直至 Vertex 结束；同理，以被标记为 mountain 的点为起点，延 perpendicular 回溯，交替标记每段 perpendicular edge 为 mountain、valley、mountain、valley...直至 Vertex 结束。

到此为止我们就完成了对 Skeletons 和 Real Perpendiculars 折叠方式的判定。

下面是一个完整的演示：

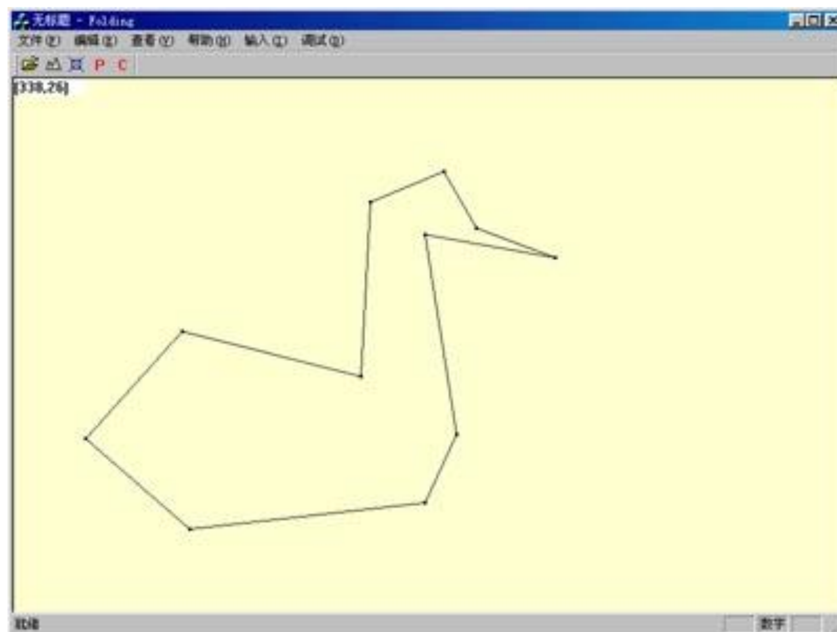
实验结论

由于时间关系，没能将折叠过程演示出来，的确非常遗憾，目前只能将最后的平面折叠方案显示出来，而且在真正折叠时，必须先将所有的折边全部折好，然后一下完成，对于普通用户来讲也不是一件容易的事，可见可视化工作还是相当必要的。但是经过我们再三思考，实在没有时间和精力完成这项工作，还请老师体谅。

下面是一个完整的演示：

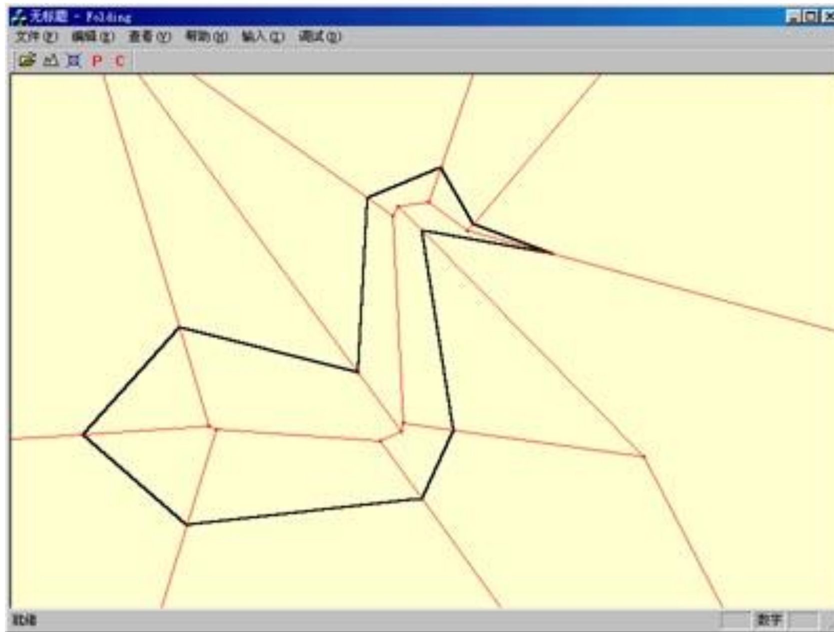
第一步：输入原始简单多边形

方法：先单击多边形按钮（左数第二个），然后用鼠标输入：左键选取所有点后单击右键结束。



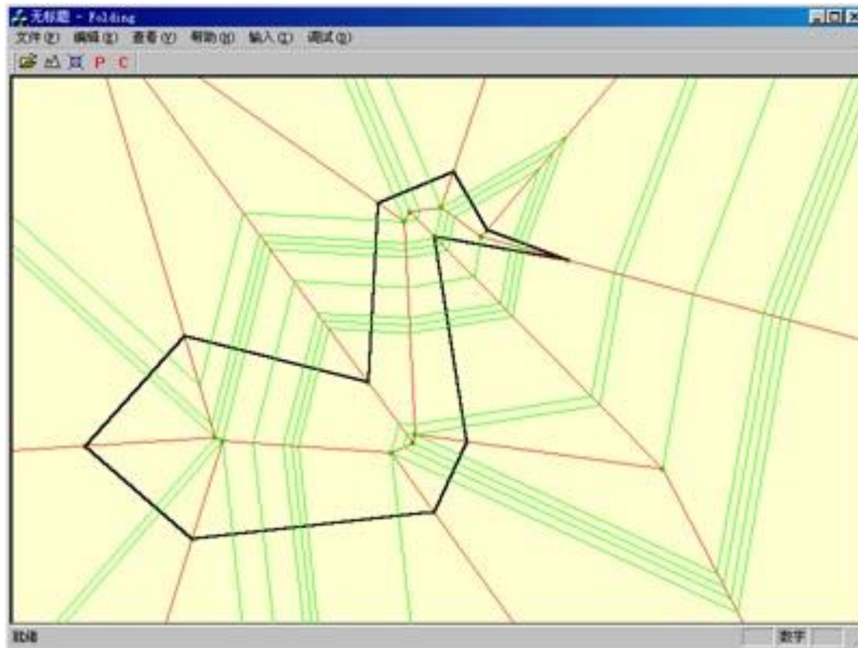
第二步：生成 Skeleton

方法：单击 Skeleton 按钮（左数第三个）,红色实线为 skeleton。



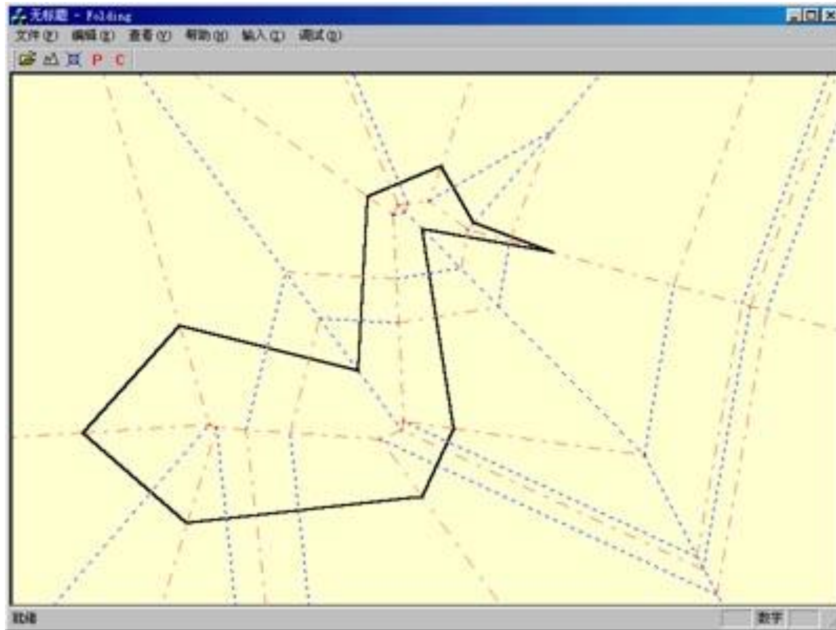
第三步：生成 Perpendicular

方法：单击 Perpendicular 按钮（左数第四个），绿色实线为 perpendicular。



第四步：生成折叠线

方法：单击 Crease 按钮（左数第五个），蓝色虚线为向内折线（valley）,棕色点划线为向外折线（mountain）。当然内外是相对的，反过来也可以。



参考文献

下面将列出我们搜集到并在不同程度上研读过的相关参考文献。

[1] E.D.Demaine , M.L.Demaine , and Lubiw. Folding and cutting paper . In *Proc. Japan Conf. Discrete and Computational Geometry* , Tokyo , 1998

[2] E.D.Demaine , M.L.Demaine , and Lubiw. Folding and one straight cut suffice . A 2_page paper (An outline of a Tech . Rep. CS-98-18 , University of Waterloo, 1998，我们没有找到这个 Tech. Rep. ，我们与 E.D.Demaine 联系过，他说还没有完成)

[3] O.Aichholzer and F.Aurenhammer. Straight skeletons for general polygonal figures in the plane. In *Proc. 2nd Int. Computing and Combinatorics.Conf.*, pp.117-126, Hong Kong, 1996

[4] O.Aichholzer and F.Aurenhammer,D.Alberts, and B. Gartner.A novel type of skeleton for polygons. *JUCS*, 1(12):752-761,1995

[5] E.D.Demaine and M.L.Demaine . Computing extreme origami bases. Tech. Rep. CS-97-22,University of Waterloo, May 1997.

[6] R.J.Lang. A computational algorithm for origami design . In Proc. 12th Symp. Computational Geometry. pp. 98—105, Philadelphia, May 1996.

[7] M.Bern, E. Demaine, D.Eppstern and B.Hayes. A disk-packing algorithm for an origami magic trick. In *Proc. Int. Conf. Fun with Algorithms*, Italy, June 1998.

附录 —— Delaunay 三角化算法

平面三角化的方法有很多种，其中 Delaunay 三角化算法是一种经典且成熟的算法，它进行三角化的原则是：使得所有三角形的最小角最大。

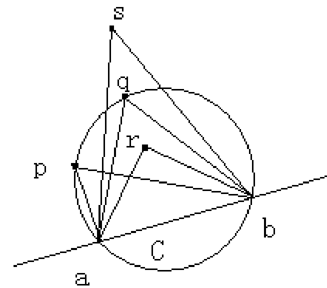
1、 基本概念和基本定理

设 P ＝ $\{p_1 , p_2 , \ldots , p_n\}$为平面上的一个离散点集。为了定义 P 上的一个三角划分，

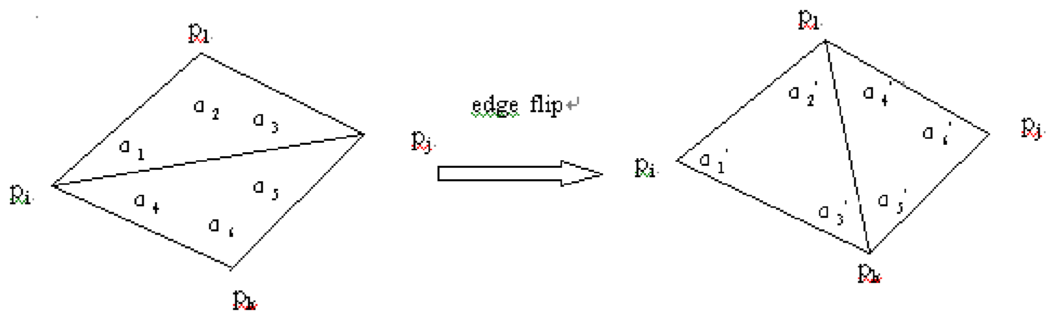
我们首先在点集 P 上定义一个概念——***最大平面划分 S***，最大平面划分 S 是这样定义的：它是点集 P 上的一个划分，如果在不损坏其平面性的基础上，已经不可能再加入一条连接点集 P 上任意两个顶点的边。也就是说，任何不在划分 S 中的边，必然与划分 S 中的边相交。点集 P 的一个三角划分就被定义为点集 P 的一个最大平面划分。

- ***定理 1***

设 C 为一个圆，l 为一条与 C 交于点 a 和点 b 的直线，p，q，r 和 s 是位于 l 同一边的几个点。假设 p 和 q 落在圆 C 上，r 在圆 C 内部，s 在圆 C 的外部，那么：$\angle arb > \angle apb = \angle aqb > \angle asb$



现在，我们考虑 P 的一个三角划分 T 中的一条边 $e = \overline{p_i p_j}$。如果 e 不是三角划分中边界上的一条边的话，那么必然是两个三角形 $p_i p_j p_k$ 和 $p_i p_j p_l$ 的交界边。如果这两个三角形形成一个凸四边形，那么我们可以通过将 $\overline{p_i p_j}$ 从 T 中移走而加入新的边 $\overline{p_k p_l}$ 而得到一个新的三角划分 T'。我们称这种操作



为***边滑动（edge flip）***，三角划分 T 与 T' 的唯一不同之处在于（如下图所示）A（T）中的六个角 $\alpha_1,...,\alpha_6$，被 A（T'）中的六个角 $\alpha_1',...,\alpha_6'$ 所取代。

在此定理的基础上，我们定义：

如果 $\min_{1 \le i \le 6} \alpha_i < \min_{1 \le i \le 6} \alpha_i'$，那么我们就称边 $e = \overline{p_i p_j}$ 是***非法的***。换句话说，如果我们能够通过局部的边滑动来增大最小角的大小，那么该边就是非法的。

- ***定理 2***

设边 $\overline{p_i p_j}$ 为三角形 $p_i p_j p_k$ 和 $p_i p_j p_l$ 的交界边，设 C 为三角形 $p_i p_j p_k$ 的外接圆。边 $\overline{p_i p_j}$ 是不合法的当且仅当点 p_l 落在圆 C 的内部。而且，如果点

p_i, p_j, p_k, p_l 构成一个凸四边形，并且不在同一个圆上，那么，在边 $\overline{p_i p_j}$ 和 $\overline{p_k p_l}$ 中必然有一条边是不合法的。

我们定义：一个***合法的三角划分***是一个不包含任何一条非法边的三角划分。经过观察，我们得出这样的结论，任何一个角度最佳的三角划分是合法的。计算一个合法的三角划分是很容易的，对于给定的任意一个三角划分，我们只需对其进行边滑动操作直至所有的边都合法。

- ***定理 3***

设 P 是平面上的一个点集。

（i） 如果点 $p_i, p_j, p_k \in P$ 是 P 的 Delaunay 三角划分中一个三角形的三个顶点，当且仅当穿过 p_i, p_j, p_k 的圆 C 内部不包含任意其他的点；

（ii）两个点 $p_i, p_j \in P$ 构成了 P 的 Delaunay 三角划分中的一条边，当且仅当存在一个闭合的圆盘 C，它包含 p_i和p_j 在它的边界上但是不包含 P 中任何其他的点。

由定理 3，我们可以推导出 Delaunay 三角划分的下面一些性质：

- ***定理 4***

设 P 是平面上的一个点集，T 为 P 的一个三角划分。那么 T 是一个 Delaunay 三角划分当且仅当 T 中任何一个三角形的外接圆内部不包含 P 中的任意一个点。

- ***定理 5***

设 P 是平面上的一个点集。P 上的一个三角划分 T 是合法的当且仅当 T 是 P 上的一个 Delaunay 三角划分。

- ***定理 6***

设 P 是平面上的一个点集。任何一个角度最佳的三角划分是 Delaunay 三角划分。而且 P 上任意一个 Delaunay 三角划分使得 P 上所有的三角划分中最小角最大。

2、 *Delaunay 三角化算法步鄹*

我们首先给出算法主要部分的精确描述如下：

Algorithm DELAUNAYTRIANGULATION(P)

Input. A set P of n point in the plane.

Output. A Delaunay triangulation of P.

1. Let $p_{-1}, p_{-2}, and\ p_{-3}$ be a suitable set of three points such that P is contained int the triangle $p_{-1}p_{-2}p_{-3}$.

2. Initialize T as the triangulation consisting of the single triangle $p_{-1}p_{-2}p_{-3}$.

3. Compute a random permutation $p_1, p_2, ..., p_n$ of P.

4. **for** r ← 1 **to** n

5. **do** (*Insert p_r into T:*)

6. Find a triangle $p_i p_j p_k \in T$ containing p_r.

7. **if** p_r lies in the interior of the triangle $p_i p_j p_k$

8. **then** Add edges from p_r to the three vertices of $p_i p_j p_k$,thereby

splitting $p_i p_j p_k$ into three triangles.

9. LEGALIZEEDGE(p_r, $\overline{p_i p_j}$,T)

10. LEGALIZEEDGE(p_r, $\overline{p_j p_k}$,T)

11. LEGALIZEEDGE(p_r, $\overline{p_k p_i}$, T)

12. **else** (*p_r lies on an edge of $p_i p_j p_k$,say the edge $\overline{p_i p_j}$ *)

13. Add edge from p_r to p_k and to the third vertex p_l of the other

triangles incident to $\overline{p_i p_j}$ into four triangles.

14. LEGALIZEEDGE(p_r, $\overline{p_i p_l}$,T)

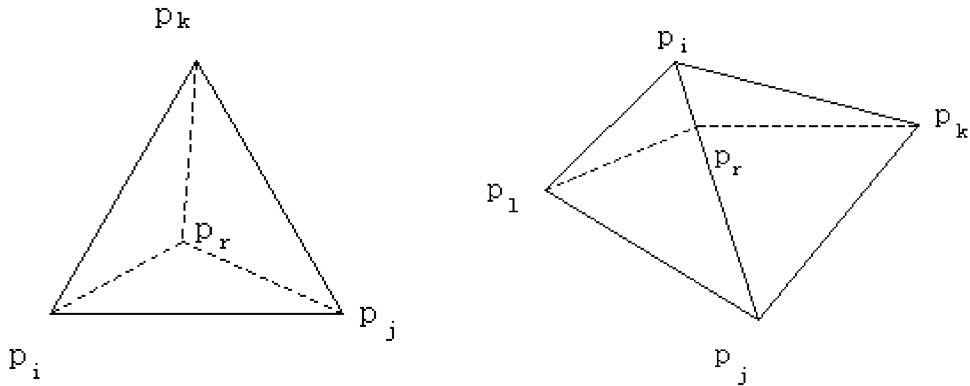
15. LEGALIZEEDGE(p_r, $\overline{p_l p_j}$,T)

16. LEGALIZEEDGE(p_r, $\overline{p_j p_k}$,T)

17. LEGALIZEEDGE(p_r, $\overline{p_k p_i}$,T)

18. Discard $p_{-1}, p_{-2}, and\ p_{-3}$ with all their incident edges from T.

19. **return** T.



附图 1　对 Delaunay 三角化算法的注解

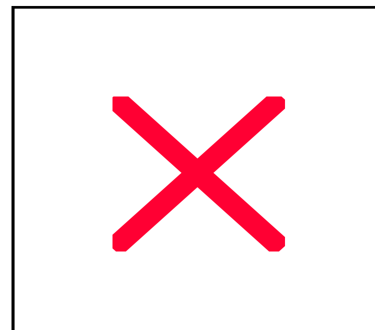
下面，我们就详细的解释该算法的各个步骤。我们的算法是从一个包括点集 P 的很大的三角形 $p_{-1}p_{-2}p_{-3}$ 开始的，也就是说，我们实际上是在点集 $P\cup\Omega$ 上进行三角划分的，其中的 $\Omega:=\{p_{-1},p_{-2},p_{-3}\}$。等我们进行三角划分后，通过去除点 p_{-1},p_{-2} 和 p_{-3} 以及与它们相关的边就可以得到 P 上的 Delaunay 三角划分。不过需要注意的是 p_{-1},p_{-2} 和 p_{-3} 的选取要距离点集 P 很远以防止它们破坏 P 上的 Delaunay 三角划分中的三角形（因为如果它们距离点集 P 很近的话，一旦最后把它们从划分中去除之后，有可能造成三角划分中的三角形缺边，即有可能形成不是三角形的多边形），尤其是我们必须保证它们不能够落在 P 中任意三个点确定的圆 C 内部。有了这样的假设，算法就可以开始进行了。接下来的步骤由图 1 的注解可以很容易的理解，在这里要详细解释一下 LEGALZIEEDGE 函数，它可以用形式化描述语言描述如下：

LEGALIZEEDGE(p r，$\overline{p_i p_j}$,T)

1.　（*The point being inserted is p r ,and $\overline{p_i p_j}$ is the edge of T that may need to be flipped.*）

2.　**if** $\overline{p_i p_j}$ is illegal

3.　　**then**　Let $p_i p_j p_k$ be the triangle adjacent to $p_r p_i p_j$ along $\overline{p_i p_j}$.



4.　　　(*Flip $\overline{p_i p_j}$; *) Replace $\overline{p_i p_j}$ with $\overline{p_r p_k}$

5.　　　LEGALIZEEDGE(p r，$\overline{p_i p_K}$,T)

6.　　　LEGALIZEEDGE(p r，$\overline{p_k p_j}$,T)

上面算法中，判断$\overline{p_i p_j}$是否合法，我们可以运用定理 2，但是由于p_{-1}, p_{-2}和 p_{-3}的加入，我们需要采取一些特殊的措施。下面我们就讨论一下p_{-1}, p_{-2}和 p_{-3}的选取以及如何判断一条边是否合法。如前所述，我们一方面要将p_{-1}, p_{-2}和 p_{-3}选得很远，因为我们不希望由于它们的存在而影响 P 上最后的 Delaunay 三角划分。另一方面，我们又不希望因此而引入很大的坐标值。通常，我们选择$p_{-1} = (3M, 0), p_{-2} = (0, 3M)$和 $p_{-3} = (-3M, -3M)$，其中 M 是点集 P 中绝对值最大的坐标值，这能保证点集 P 中的所有点都包含在三角形$p_{-1} p_{-2} p_{-3}$中。然而，当我们检验一条边是否合法时，我们并不使用这些坐标。相反的，我们认为p_{-1}落在 P 中任何三个点所确定的圆之外，p_{-2}落在点集$P \cup \{p_{-1}\}$中任何三个点所确定的圆之外，p_{-3}则落在点集$P \cup \{p_{-1}, p_{-2}\}$中任何三个点所确定的圆之外。这样，我们就可以导出下面的判断一条边$\overline{p_i p_j}$是否合法的规则：

情况（i）：i 和 j 都是负的。

这种情况下，我们认为$\overline{p_i p_j}$是合法的，因为我们需要保持大三角形$p_{-1} p_{-2} p_{-3}$的边直至最后。

对于其他的情形，我们设p_k和p_l是以$\overline{p_i p_j}$为交界边的两个三角形的另外两个顶点。

情况（ii）：下标 i , j , k , l 都是正的。

这是最为常见的一种情况。待测的点中没有一个是特殊的点，所以$\overline{p_i p_j}$是非法的当且仅当p_l落于p_i, p_j和 p_k所定义的圆中；

情况（iii）：如果下标 i , j , k , l 中只有一个是负值。

我们并不希望一个特殊的点破坏 P 点集上的 Delaunay 边。因此，如果 i 或者 j 是负值（也就是说，p_i或者p_j是特殊的点），那么我们认为$\overline{p_i p_j}$是非法的，它将被$\overline{p_k p_l}$替代；否则，我们就认为$\overline{p_i p_j}$是合法的。

情况（iv）：下标 i , j , k , l 中有两个是负值。

这种情况下，i 和 j 中以及 k 和 l 中各有一个是负值。因为如果 i 和 j 都是负值，那么则回到情况（i）,而 k 和 l 又不可能都是负值，因为p_k和p_l中有一个是刚刚作为p_r加入进来的点。所以，如果下标 i 和 j 中负的那个值比 k 和 l 中负的那个值小，那么，我们认

为 $\overline{p_i p_j}$ 是合法的，否则 $\overline{p_i p_j}$ 就是不合法的。

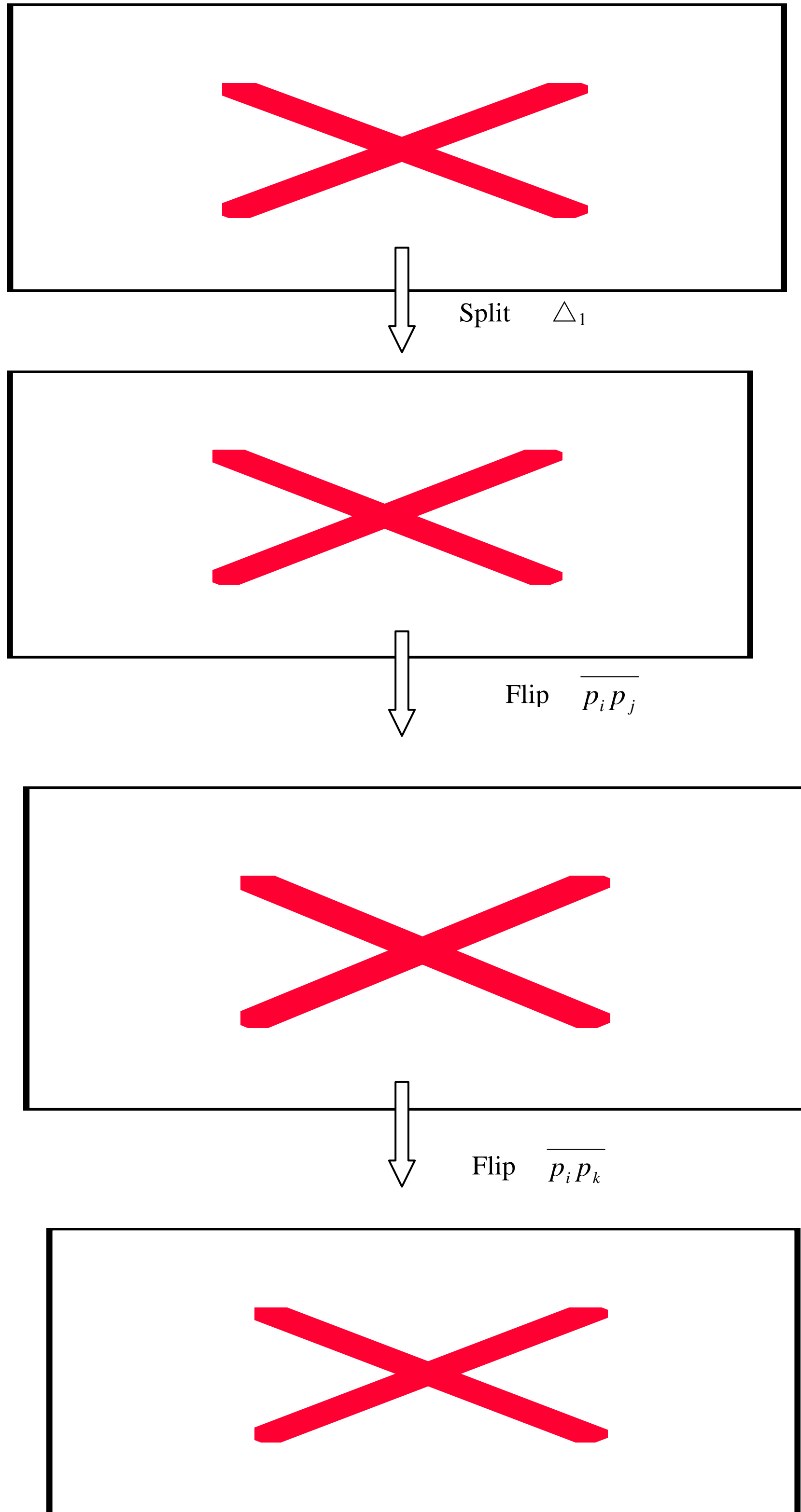
情况（v）：下标 i , j , k , l 中有三个是负值。

这种情况是不会出现的，i 和 j 不可能都是负值，因为这样就回到情况（i）；k 和 l 也不可能都是负值，因为 p_k和p_l 中有一个是刚刚作为 p_r 加入进来的点。

综合上面的 5 种情况，LEGALIZEEDGE(p r ， $\overline{p_i p_j}$,T)中判断 $\overline{p_i p_j}$ 是否合法的那一步（**if** $\overline{p_i p_j}$ is illegal）就解决了。

对于 Delaunay 三角化算法，我们还剩一个问题要讨论，就是对于任何一个新加入的点 p_r 我们如何判断它包含在哪个三角形内部，既 Delaunay 三角化中的第 6 个步骤（Find a triangle $p_i p_j p_k \in T$ containing p r ）。对于这个问题，我们可以用最简单的方法——对于目前的三角划分，一个三角形一个三角形地判断 p r 包含在哪个三角形内部，不过一旦三角形很多的情况下，这种方法的效率是非常低下的，所以我们在进行三角化的同时，建立了一棵搜索树。这棵树的叶子结点是当前三角划分 T 中的三角形，而树的非叶子结点则代表在先前步鄹中曾经存在过的三角形，后来被分裂了或替换了。我们的搜索树是这样建立起来的：在 Delaunay 三角化算法中的第 2 行中，我们初始化树 D 为只有一个叶子结点的树，这个叶子结点对应三角形 $p_{-1}p_{-2}p_{-3}$。下面，设想在加入某一个点后，我们把当前三角划分中的一个三角形 $p_i p_j p_k$ 分裂为三个（或两个）三角形。那么，对于搜索树 D 的相应改变是：在 D 中加入三个（或两个）新的叶子结点，然后把原来的叶子结点 $p_i p_j p_k$ 变为一个中间结点，并且让它指向这三个（或两个）新的叶子结点。类似的，如果我们在边滑动中用两个新的三角形 $p_k p_i p_l$ 和 $p_k p_l p_j$ 代替三角形 $p_k p_i p_j$ 和 $p_i p_j p_l$ 的话，我们为两个新的三角形建立两个新的叶子结点，然后把被替代的三角形变为中间结点并让它们分别指向刚才建立的两个新的叶子结点。图 2 显示了加入一个点后引起的搜索树 D 的变化。我们可以观察到，当我们把一个叶子结点变为了中间结点后，它向外发出的指针最多有三个。

运用搜索树 D，我们可以确定下一个加入的点 p r 所在的三角形。做法是这样的：我们从 D 的根结点（与三角形 $p_{-1}p_{-2}p_{-3}$ 相对应）开始，我们首先检查 p r 落于根的三个孩子中的哪一个孩子三角形中，然后，我们再检查 p r 落于这个孩子三角形的哪一个孩子三角形中，如此传递下去，直至我们到达叶子结点为止，这个叶子结点就是我们所要找的点，它所对应的三角形就是包含点 p r 的目前三角划分中的三角形。由此可见，通过这样的搜索树，寻找包含点 p r 的三角形的速度就大大加快了。



附图 2　加入一个点后引起的搜索树 D 的变化