Steiner Tree Problem

目录

1. 问题背景

1.1 概述

斯坦纳树(Steiner Tree Problem)问题，是组合优化中一个历史悠久的问题。早在 17 世纪初，法国数学家费马就曾提出过费马点问题，即对平面上任意给定的三个点，如何求出一个点，使得该点到这三点的距离之和最小。费马点问题在后来被瑞士科学家斯坦纳（J.Steiner,1796-1863）推广为：在平面上任意给定 n 个点 $A_1,A_2,\cdots\cdots,A_n$，如何求得一点 P 使得 A_i 至 P 点的距离之和$\sum_{i=1}^{n}|A_iP|$最小。1750 年，英国的辛普森(T.Simpson)进一步将这一问题推广至一个加权问题，即对任意给定的三个正数 a_1,a_2,a_3 和平面上的三个点 A_1,A_2,A_3，如何求得一点 P，使 $a_1|A_1P|+a_2|A_2P|+a_3|A_3P|$最小。

直到，1934 年和 1941 年，亚尔尼克(V.Jarnik)和克斯勒(M.Kossler)与库朗(R.Courant)和罗宾斯（H.Robbins）进一步提出了现在所说的斯坦纳树问题：对于平面上给定的 n 个点 $A_1,A_2,\cdots\cdots,A_n$，如何求出一个连接这 n 个点的最小网络。即所求的不是一个点 P，而是由一些线段构成的连接这 n 个点的一个最小连通网络。这一问题被称为斯坦纳最小树问题（Steiner Minimum Tree Problem,SMTP），简称为斯坦纳树问题。使得问题的应用范围大大扩大，难度也大为增加。

斯坦纳树问题在大规模集成电路设计、道路交通规划设计等领域都有着广泛的运用。图 1 展示了一个 3 个点的斯坦纳树的例子。图中的 t_1, t_2, t_3 是单位正三角形的三个顶点。(a) 为一棵长度是 2 的最小生成树，(b)为一棵长度为$\sqrt{3}$的斯坦纳树。

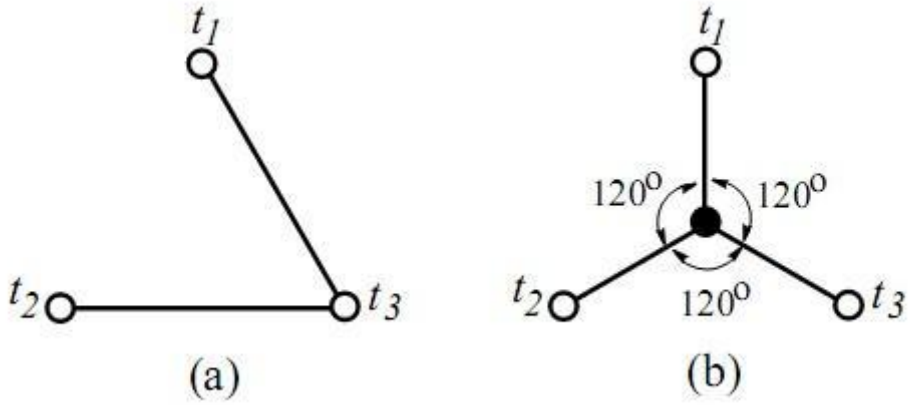


Fig 1. (a) 最小生成树 (b) 斯坦纳树
(来源：Dingzhu Du and Xiaodong Hu[1])

斯坦纳树在大规模集成电路及无线传感器网络（物联网）中具有广泛的应用。

1.2 斯坦纳树的构造

对于斯坦纳树构造的认识，科学家有着由简到繁的递进式推进过程。

意大利的托里切利最早解决了 n=3 的斯坦纳树构造问题。 托里切利指出：若在∆ABC 的三条边上分别向外作一等边三角形，并对每一三角形作一外接圆，则此三圆交于一点 P，即为所求之点。但这只适用于三点所构成的三角形内角均小于 120 ℃的情形。1647 年，意大利的卡瓦列里（F.B.Cavalieri）进一步证明了上述作图中，在 P 点的三个交角∠APB,∠BPC 和∠CPA 均为 120 °。1834 年海嫩（F.Heinen）提出并解决了存在一内角≥120 ℃的情形。此种情况为一退化情况，此时的点 P 应选在三角形最大内角的角顶。

对于 n=4 的情况要比 n=3 时复杂得多，对于三点来说，斯坦纳树的可能连接方式有 4 种，但是对于四点来说，可能的连接方式就达到了 31 种[2]。对此，1978 年波拉克（H.O.Pollak）给出了一波拉克定理。

对于 n=4 时，一般存在 2 株斯坦纳树，总长一般不相等，通过波拉克定理可不必求出两株树再进行比较，但对于 n≥5 时，还没有发现这样的方法。并且已证明寻求 SMT(X)为一 NP 难题。目前所用的方法基本是枚举法。此时点集所组成的归类迅速增大。如当 n=6 时，其数为 5625，而当 n=8 时则达到 2643795。因而目前这类问题主要有两种解决途径：一是对 X 的性质加些限制：二是寻求问题的近似解。

2. 设计方案

前面已经提到对于 n 点的斯坦纳树构造为 NP 难题，在本项目中，我们使用启发式算法对斯坦纳树问题进行求解。

3. 系统设计

3.1 开发环境

Windows XP + Visual studio 2005 + MFC + OpenGL。

3.2 编译与链接

工程中使用了 MFC 自带的链接库，不需要特殊操作，但需要链接 OPENGL 的库文件与头文件。

3.3 总体框架

系统主要分为三大模块：

① 系统框架模块

这部分是整个程序框架，负责数据的输入及输出，与其余两大模块进行连接发，完成所有控件操作。

② 核心算法模块

这部分主要实现核心算法，对输入点集构建斯坦纳树，并将结果返回给系统。

③ 图形绘制模块

这部分主要完成图形界面的绘制，实现对算法结果的图形演示。

4. 数据结构

本工程中用到的一些基本数据结构如下：

```
// 点的结构
struct Point
{
    // 坐标
    int x;
    int y;

    // 控制信息
    bool inTree;
    bool steinerPoint;
    int round; // 表示改点是第几轮被加进来的；初始点的轮数为0

    // 边链表头指针
    void * edgeHead;
    int edgeNumber;

    // 改点所在连通分支树的头指针，用来在Kruskal算法中查找和合并连通分支
    Point * setHead;

    // 点号
    int pointID;

    // 所属连通分支号
    int setID;
};

// 边的结构
struct Edge
{
    Point * start;
    Point * end;
    float length;

    int round; // 表示改点是第几轮被加进来的；初始点的轮数为0

    int a;

    bool inMST;
};
```

在算法运行的不同阶段，我们还使用了特定的数据结构。下面依次介绍。

4.1Delaunay 三角剖分中的数据结构

这一部分借用了之前第四组同学的代码。在计算三角剖分的过程中，使用了DCLE结构。该结构要求Edge 表示一条有向半边，存储向关联的起始Vertex、终止Vertex、所在Facet、下一条Edge、上一条Edge 等信息；Vertex 中存储一条相关Edge，该Edge 的起始Vertex 就是该Vertex。相关结构的定义如下：

```
//顶点信息
class Vertex
{
public:
    //坐标信息
    double x;
    double y;

    //该边指向从当前顶点出发的任意一条半边
    HalfEdge* pEdge;

    //构造器和析构器
    Vertex(double newX, double newY);
    Vertex(double newX, double newY, void * newPoint);
    Vertex(void);
    ~Vertex(void);

    //比较操作符，便于排序，
    //原则：先比较横坐标，再比较纵坐标，坐标小者返回true
    bool operator<(Vertex& v);
    bool operator<(Vertex* pV);

    //计算到另一点的距离(平方)
    double distance(Vertex& v);
    double distanceSquare(Vertex& v);

    void * point;
};

//半边信息
class HalfEdge
{
public:
    //半边起点在vector里的编号
    //Vertex* ori;
    int oriNum;
    //孪生边
    HalfEdge* twin;
    //沿着顺时针方向遇到的第一条同起点半边
    HalfEdge* pre;

    //沿着逆时针方向遇到的第一条同起点半边
    HalfEdge* next;
private:
    HalfEdge(void);
public:
    //构造一个半边而不构造其对应的孪生边，其pre和next都被初始化为NULL
    HalfEdge(int pOriNum, HalfEdge* pre, HalfEdge* next, vector<Vertex>& vertexSet);

    //在构造了当前半边的同时也构造了其孪生边,当前半边及其孪生边的pre和next都被初始化为NULL
    HalfEdge(int pOriNum, int pDestNum, vector<Vertex>& vertexSet);

    //析构器
    ~HalfEdge(void);

    bool operator!=(const HalfEdge& edge);
};
```

4.2Kruskal 算法

为了保证Kruskal算法可以在O(nlogn)的时间内完成，我们在排序过程中使用了堆排序算法，并且为Kruskal算法的连通分支合并过程设计了专门的数据结构：

```
// 点链表结点的结构
struct PointNode
{
    Point * self;
    PointNode * next;
};

// 点链表的结构
struct PointList
{
    PointNode * head;
    PointNode * rear;
};

// 连通分支集合的结构
struct Set
{
    PointList pointList;
    int pointNumber;
};
```

对于每一个连通分支，例如Set s，s.pointList中包含有位于该连通分支内的全部点(Point)。除s.pointList中的第一个点外，s.pointList中的每一个点的setHead指针指向s.pointList的第一个点。这样，在集合合并的过程中，例如集合A和B，只需将A的pointList的头指针连接到B的pointList链表的尾部，同时修改A的第一个点的setHead，使其指向B的pointList的第一个点。这样，可以保证集合的合并在常数时间内完成。

同时，对于查找一个点坐在的连通分支的操作，只需不断地向前查找其setHead指针，当setHead->next为空时，其所代表的点坐在的集合便是所求的集合。

4.3 计算 Steiner Minimal Trees

在计算Steiner Minimal Trees的过程中，除了以上提到过的基本数据结构，我们还使用了边链表结构：

```
// 边链表结点的结构
struct EdgeNode
{
    Edge * self;
    float angle;
    EdgeNode * next;
};
```

在计算Steiner Minimal Trees的过程中，由于需要频繁加入新的Steiner Point，并删除2度的Steiner Point，因此我们采用点－边双链表的结构保存计算结果，即，将当前点集（包括初始出入点和新加入的Steiner Point）组织成一个链表，同时每个点包含一个由端点为该点自身的边组成的边链表。

5. 算法介绍

5.1 相关知识

首先介绍几个已被证明的定理（具体证明请见参考文献一）：

1. 在欧氏空间中，一棵斯坦纳树中的任何一个点的度数都小于等于三，其中任何一个斯坦纳点的度数正好为三，这是由于如果一个斯坦纳点（v1）的度数为一，则可以立即删除它，如果为二(假设与其相关的两条边为 v1v2,v1v3)，则可以连接 v2v3, 并去除 v1v2,v1v3 使总距离更短，如果度数为四，则与其相关的四个角度中必有一个度数小于 120 度，则可以通过构造费马点使总距离更小；
2. 在欧氏空间中，一个斯坦纳树中最多有 N-2 个斯坦纳点；
3. $MST \subseteq RNG \subseteq DT$,其中 MST 是最小生成树，RNG 是相关邻居树（假设两个点 v1,v2 则它们称为相关邻居，如果：$\|v1-v2\| \le \|v1-w\| or \|v1-v2\| \le \|w-v2\|$），DT 是 Delaunay 三角剖分得到的图；
4. 如果 T＝（V, E）是欧氏空间中对于有限点的一棵斯坦纳树，则其中任意一个斯坦纳点都是其邻居的费马点；
5. 对于欧氏空间，与斯坦纳树中的斯坦纳点相关的三个角都为 120 度。
6. 对于三个点 A,B,C，如果其有一个角度大于等于 120 度，则其费马点即 120 度角所在的点，如果所有角度都小于 120 度，则其费马点可以通过以 AB,BC,AC 为边向外分别画出正三角形 ABC',BCA',ACB', 则 AA',BB',CC'的交点即为费马点。

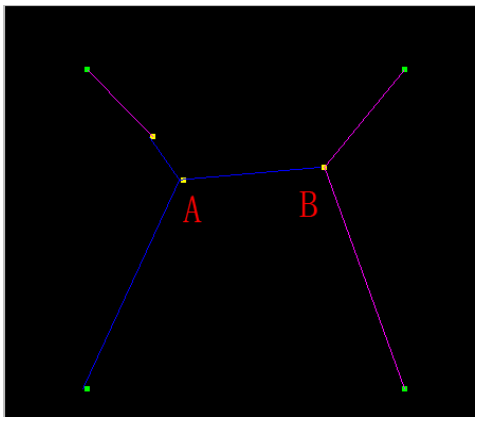
5.2 算法思想

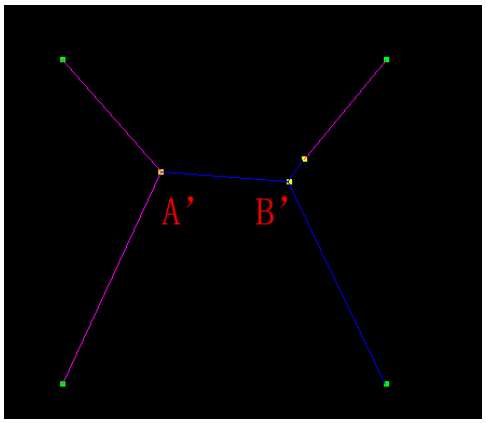
在此算法中，给定一个点集，我们先构造点集的 Delaunay 三角剖分，通过得到的三角剖分结果，寻找最小生成树。

在得到的最小生成树中，我们进行多轮扫描，一轮的定义如下：

首先将当前所有点都加入到一个队列中，从队头开始处理每个点，对于每个点，查看其是否有小于 120 度的角，如果有，则求出其费马点，并将新求出的费马点加入到队列，加入后还需要查看是否产生了度数为二的斯坦纳点（不可能产生度数为一的点），如果产生了，则需要将此点移除，并新加一条连接与此点相关的两个点的边；如果没有，则将此点从队列中移除。

按照此算法，算法的每个步骤都会朝着 SMT 前进，即其距离和会减小（无论是生成费马点还是去除二度点，都会使得距离减小），但是却不会最终达到 SMT，即其会无限循环下去（如果精度足够高的话），原因见下图示例：



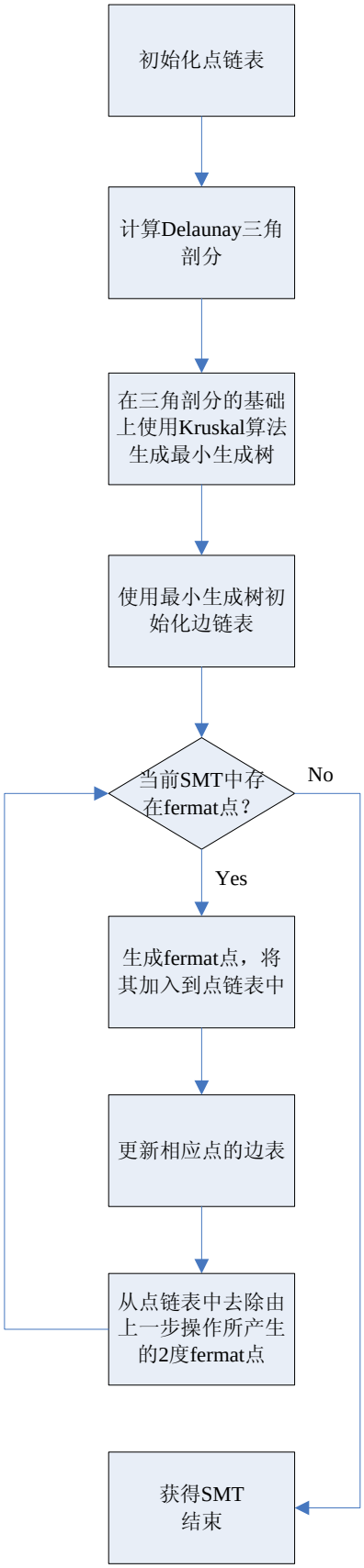


以上两图为算法过程中两个相邻的过程，即左图发生后就发生了右图，左图中 A 角为 120 度，但 B 角小于 120 度，在 A 点左上方的斯坦纳点去除后（去除二度点），会以 B 角为基础求费马点，然后得到右图，在右图中 A’角小于 120 度，B’角为 120 度，接着以 A’角为基础求费马点，可知 A”将得于 120 度，B”将小于 120 度，如此循环,可以证明（如归纳法），两个角都永远无法都达到 120 度，即算法将无法终止。

在真实的程序中，程序不能永远循环下去，只能找到一个次优解，即满足某个精度要求即可。

在我们的算法中，我们由用户决定何时终止算法（即演示的时候按下 CANCLE 键即可终止算法），并且由于机器的显示只能显示像素极的，我们以像素级为最高精度要求，当在像素级别无法找到更优解时（即找到费马点），即终止程序，并将结果显示出来。

5.3 算法流程图：


初始化点链表
计算Delaunay三角剖分
在三角剖分的基础上使用Kruskal算法生成最小生成树
使用最小生成树初始化边链表
当前SMT中存在fermat点？
No
Yes
生成fermat点，将其加入到点链表中
更新相应点的边表
从点链表中去除由上一步操作所产生的2度fermat点
获得SMT结束

5.4 关于求解费马点

Fermat点问题是计算组合优化中的一个经典问题。对于尺规作图求解fermat点，目前已有确定性的精确算法。但是，对于给定三个点的坐标，通过代数方法求得fermat点的解析解，是一个比较困难的问题。我们查阅了大量文献，没有找到代数求解fermat点的解析公式，也未能找到一个多项式时间的算法来精确求解fermat点。

为了满足项目需要，我们采用像素遍历的方法求解fermat点。即，对于一个给定的三角形，我们依次遍历这个三角形内部的全部像素点，对于每个像素点，计算该点到三角形三个顶点的距离和。当三角形内部的全部像素点均被遍历后，我们便可以取到三个顶点距离和最小的那个点作为该三角形的fermat点。

这种方式实质上类似于Brute-Force方法，看起来具有极高的时间复杂度。但是，通过对问题的深入分析，我们发现，对于我们的整体算法来说，该方法具有一种十分优良的性质，对于点链表的没一趟扫描，通过该方法求得全部fermat点的计算量的总和是一个常数！

这是因为，在计算Steiner Minimal Tree的过程中，我们实际上是把空间分割成了若干不想交的三角形的集合（考虑我们是在求得Minimal Spanning Tree的基础上计算Steiner Minimal Tree的，而Minimal Spanning Tree恰是Delaunay三角剖分的子图）。我们所要做的，就是在每一个三角形中求解一个fermat点（如果存在的话），然后更新点集（也就是更新了点集的三角剖分，这时的点集已经不同于初始输入时的情形了），直到不能找到新的fermat点为止。

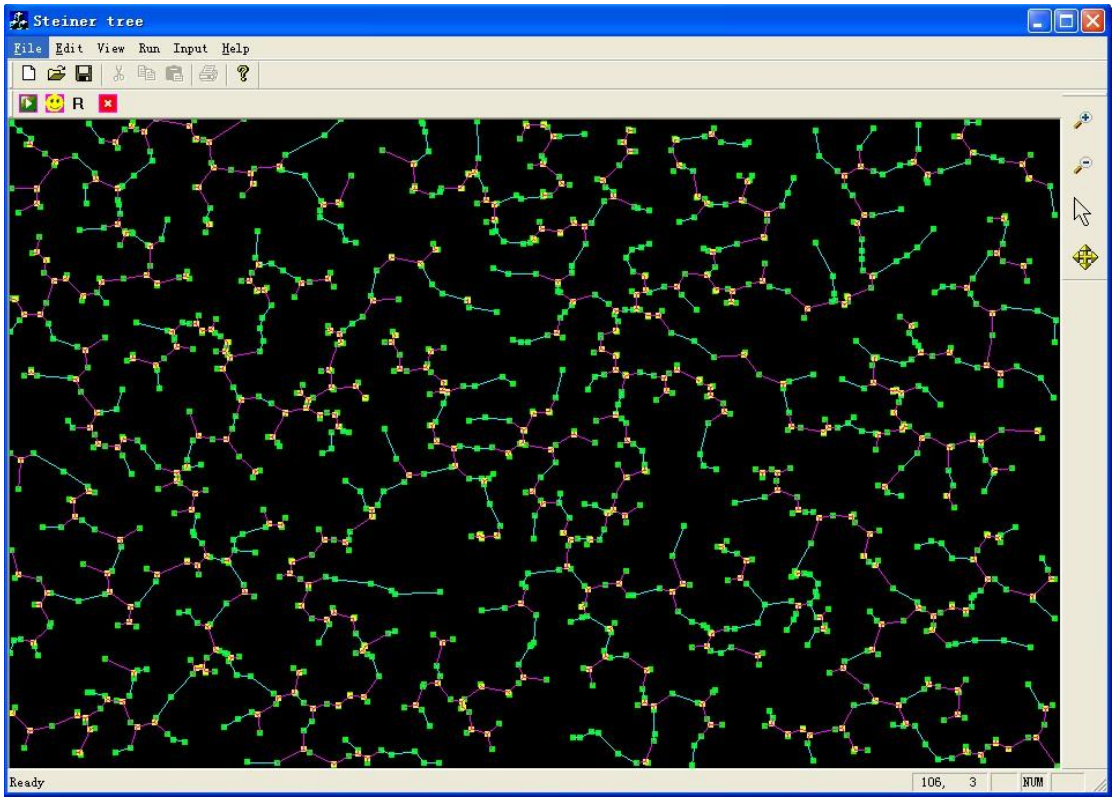
我们在实现的过程中，是通过对点链表的扫描来求解Steiner Minimal Tree的。那么在对点链表的每一趟遍历中，从遍历开始到结束，我们实际上恰好将当前点集所围成的凸包内的全部像素遍历了一遍，不多也不少。由于屏幕所限，点集所围成的凸包内的像素数会以常数为界。这就是说，对于点链表的趟扫描，其操作的总和为一个常数！

在测试程序的过程中，我们发现，对于n个初始点的输入，使用我们的算法求解Steiner Minimal Tree时对点链表所进行的总的扫描的趟数均落在[0.5n, 1.5n]之内，即扫描的趟数为O(n)量级。这样，由于每趟扫描只是进行了常数次操作，那么在计算出最小生成树之后，求解Steiner Minimal Tree实际上只是具有O(n)的时间复杂度。

虽然我们未能证明这个结论的正确性，但是通过大量数据的验证，我们观测到在绝大多数情况下（特别是随机生成的点），计算出最小生成树后求解Steiner Minimal Tree的时间大致等于计算最小生成树之前计算Delaunay三角剖分的时间，也就是说，它至少是不慢于O(nlogn)的时间复杂度的。

6. 运行结果与分析

6.1 界面整体外观



1000 个点的斯坦纳树

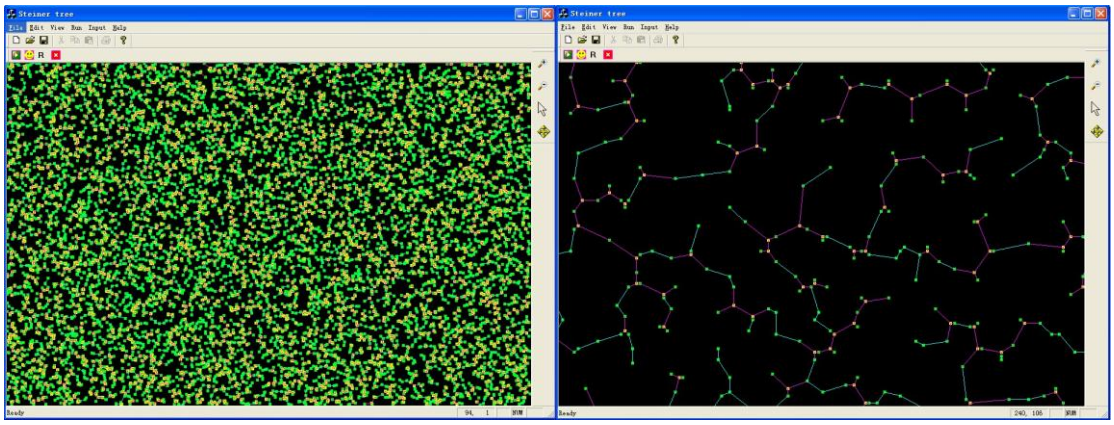
在上图中，绿色点为初始输入点，黄色点为斯坦纳点，粉色边为保留下来的最小生成树中的边，青色边为因斯坦纳点的引入而新加入的边。

6.2 使用手册

-  （或者“File”→“New”）：新建功能。清空界面和数据集，重新初始化程序。
-  （或者“File”→“Open”）：打开功能。从指定格式的文本文件中读入点集。
-  （或者“File”→“Save”）：保存功能。将界面中显示的点集数据以指定格式保存至文本文件中。
-  (或者“Run”→“Run”)：运行功能。依据界面的上的点集数据，生成斯坦纳树，并将结果显示在界面中。
-  （或者“Run”→“Play”）：单步执行功能。单步演示斯坦树的生成过程。

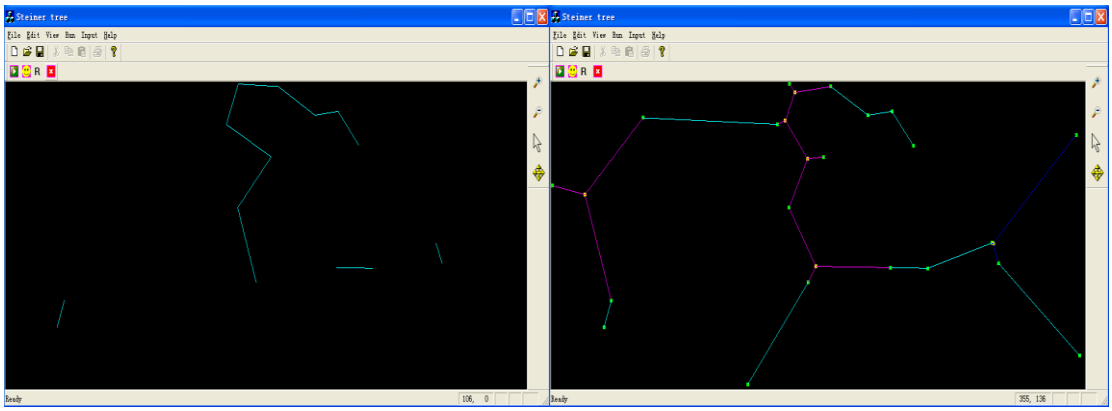
-  （或者“Run”→“Cancle”）:取消功能。在运行过程中取消运行，在演示过程中停在某个次优解处。
-  （或者“Input→ “Random”）随机生成功能。在弹出的对话框中填入点数，界面将自动生成所需的随机点集。
-  （或者“View”→“Room Out”） 放大功能。点击此图标后，用鼠标左键点击界面中需放大的部分，界面即可放大显示部分细节。
-  （或者“View”→“Room In”） 缩小功能。点击此图标后，用鼠标左键点击界面中需缩小的部分，即可获得更大的视图范围。
-  (或者“View”→“Normal”)还原功能。点击此图标后，界面将恢复至原始大小的图形界面。
-  （或者“View”→“Move”）移动功能。点击此图标后，可用鼠标左键拖动界面，以观察所需部分的图形界面。

6.3 部分运行结果



10000 个点的斯坦纳树　　经放大后的图形界面

动态演示的过程如下图所示：



演示过程　　演示过程

7. 进一步的工作方向

对于我们来说，Steiner Spanning Tree 问题是一个全新的领域。小组成员从搜索相关资料文献、学习相关理论知识、设计数据结构与算法到最终编码实现，经历了一个比较艰辛的过程。在这个过程中，我们对 Steiner Spanning Tree 问题有了更加深入的理解，同时我们也发现了在最初的数据结构与算法设计中所存在的一些问题。由于项目时间所限，我们没有在程序中将这些问题全部解决。根据我们项目中所存在的问题，我们将在今后的工作中针对如下方面进行改进：

a) 对本项目所用算法的改进

i. 使用事件队列代替线性扫描

在寻找 Steiner Point 的过程中，我们采用了线性扫描的方式，即不断地对整个点链表进行扫描，直到不能找到新的 Steiner Point 为止。这种方式可能会带来很多无用的重复扫描。针对这个问题，我们认为可以借鉴扫描线算法中事件队列的思想，通过一定程度的判断，仅将有可能产生 Steiner Point 的点或边记入事件队列。这样，就有可能大幅度降低扫描过程的时间复杂度。

ii. 优化 fermat 点的求解方法

Fermat 的解析求解时一个非常困难的问题。在本工程中，为了在较快的时间内进行演示，我们采用了像素扫描的方法。虽然这种方法可以保证在对整个点链表的扫描中仅会扫描常数多个像素，但是对于比较大的显示空间，效率仍然比较低。而且，这种方法也不具有通用性。

在今后的工作中，我们会进一步改进寻找 fermat 点的方法，提高求解 fermat 点的效率。

b) 探索更加优质、高效的算法

本工程所使用的算法本质上属于启发式一类，改算法能够保证在较快时间内收敛，但不能保证结果的全局最优性。

目前，对于二维欧式空间的 Minimal Steiner Tree 问题，虽然存在多项式级别的确定性解法，但是，其复杂度为 $O(n^4)$甚至更高，以至于不能应用较大规模的数据。

在今后的工作中，我们将进一步寻求更加高效、的算法。

8. 项目中遇到的问题及解决方案

1） 对Steiner Minimal Tree问题缺乏了解

在进行这个项目之初，小组三人对Steiner Minimal Tree问题（以下简称SMT问题）均不熟悉。为了较好地完成此次项目，我们查阅并阅读了大量相关文献，其中包括两本专门介绍SMT问题的书籍“Steiner Minimal Trees[1]”和“Steiner Tree Problems in Computer Communication Networks[2]”，还阅读了许多论文。

通过阅读文献，我们对SMT问题有了深入理解，这为我们最终完成本次项目奠定了重要的基础。

2） 无OpenGL基础

在上一个项目中，我们在显示图像的时候使用了MFC自身的作图功能。这样画出的图像不够美观，效率也比较低。为了更加高效、优美地将运算结果显示出来，我们决定在本次项目中使用MFC+OpenGL的方式进行绘图。

由于小组成员均无OpenGL基础，我们从零开始学习了OpenGL的相关原理及操作。通过对OpenGL的学习，我们顺利实现了项目初期设定的目标，较好地实现了图像显示的功能。

3） 平滑显示的问题

由于MFC是通过调用onDraw()函数的方式进行屏幕上图像的绘制的，这样在数据量较大的情况下，onDraw()函数就会运行较长时间，导致在这段时间内用户无法控制显示面板的各项功能。这将带来很差的用户体验。

为了解决这个问题，我们将显示图像的核心代码做成若干线程，在onDraw()函数中有条件地启动这些线程。这样，我们就实现了计算与显示的分离，使得用户可以在程序运行的任何时刻都可以观测到当前运行的结果，并能对运行状态进行自由的控制。

在多线程编程的过程中，我们遇到了若干临界区并发访问以及线程间死锁的问题。通过深入的分析与仔细调试，我们妥善地解决了这些问题。

4） 坐标变换的问题

用户在进行输入或观看演示的过程中可能频繁地调整显示窗口的大小。这样，由于调整窗口后点集的坐标空间发生了变化，最终计算与显示的结果可能与正确结果差之甚远。在项目开发初期，我们没有考虑这个问题，导致实际测试的时候图像显示产生了各种不可预期的错误结果。

为了解决这个问题，我们将图像坐标进行了一致性映射。在计算的过程中，将所有点的坐标统一映射到一个标准的坐标空间，在显示的时候根据显示窗口的大小将点的坐标映射到显示空间。这样，我们就解决了坐标不一致的问题。

9. 我们的收获

通过本次项目，我们学到了很多东西，受益颇丰。下面，我们对我们的收获做一个总结。

i. 深化了知识结构

在完成本次项目的过程中，我们深入学习了Steiner Minimal Tree问题。小组成员对SMT问题经历了一个从无知到了解到比较深刻的理解的过程。同时，在学习SMT问题的过程中，我们也对之前学习过的一些知识，比如Delaunay三角剖分、最小生成树、线程间的并发与同步等问题有了更加深刻的理解。

通过本次项目，我们深化并扩展了自己的知识体系结构，相信这对我们今后的学习与研究工作将会产生非常有益的影响。

ii. 提高了编程能力

在编程实现的过程中，我们学习了MFC、OpenGL、多线程编程等之前不熟悉或知之甚少的编程技巧。通过实际编程，我们比较熟练地掌握这些技巧，大大提高了我们编写程序解决实际问题的能力。

iii. 提高了团队协作能力

由于我们是以团队的形式共同完成项目，因此团队成员间的沟通与协作就显得至关重要。在本次项目开发过程中，小组成员之间进行了充分、有效的沟通，通过制定计划、定期开会、统一函数接口等方式保证了成员间的有效协作，为最终顺利完成项目提供了保障。

通过本次项目，我们提高了自己的团队协作能力，这对于今后实际步入工作岗位来说，将是一笔宝贵的财富。

10. 特别致谢

在项目开发过程中，我们受到了很多帮助，我们对曾经帮助过我们的老师和同学表示衷心的感谢。感谢邓俊辉老师对我们的关心以及知识上的指导。感谢第一次实验中第四组同学的辛勤劳动（虽然我们不知道你们是谁，但你们关于Delaunay三角剖分的代码实现为我们带来了非常大的帮助）。感谢王俊同学对图形用户界面的设计所提出的宝贵意见。

11. 参考文献

[1] Dietmar Cieslik, Steiner Minimal Trees, KLUWER ACADEMIC PUBLISHERS, 1998

[2] Dingzhu Du, Xiaodong Hu, Steiner Tree Problems in Computer Communication Networks, World Scientific, 2008.

[3] W.D.Smith. How to find Stiner Minimal Trees in Euclidean d-Space Algorithmica, 7:137-178, 1992.
[4] D.Z.Du. On Greedy Heuristics for Steiner Minimum Trees. Algorithmica, 13:381-386, 1995.
[5] J.E. Wieselthier, G.D.Nguyen, and A. Ephremides, On the construction of energy-efficient broadcast and multicast trees in wireless networks, Proceddings of the 19th Annual Joint Conference of the IEEE Computer and Communications Socienties (INFOCOM), 2000, pp. 585-594.
[6] 越民义, 最小网络——斯坦纳树问题. 上海科学技术出版社, 2006.11.
[7] 堵丁柱，谈谈斯坦纳树.数学通报，1995.1.