直角多边形的矩形最小剖分

——计算几何实验终期报告

冯哲　张京城

2004 年 12 月

摘 要

本系统实现并改进了一个直角多边形的矩形最小剖分问题的解决方案。经过工业例子的实际测试，运行时间符合工业的要求。

文中，首先给问题的背景分析；其次，给出了算法和原理；第三部分，说明了我们在实现这一系统中所遇到的问题及解决方法；在第四部分，介绍了系统的设计框架及重要的数据结构；第五部分是对算法复杂度和测试结果的分析；最后给出全篇的结论。

1 问题背景

该问题是由 VLSI 设计问题引出的，目前有两类 VLSI 问题中需要直角多边形的矩形最小剖分。一类是，在光刻掩模板的过程中，所采用的光刻头是一个可伸展的矩形框，每次在掩模板上，刻出一个矩形的电路。掩模板的制造成本与制作时间相关联，而制造时间这里由光刻次数所决定，由此引出对掩模板的矩形最小剖分问题[1]；另一类问题，存在于对布局过程中，如何合理的划分出通道区域[5]。

对这一问题的研究主要是在 80 年代，由于在 80 年代末，这一问题已得到较好的解决，后来很少有人继续研究。

在我们研究的这篇论文以前的研究中，多是采用显式的构造二部图，来找最大match，从而得到一组不相关的弦，复杂度都比较高，代表的有Hopcroft and Karp's algorithm，他们的算法复杂度为$O(n^{2.5})$[6]；Imai and Asano's algorithm，他们的算法复杂度为$O(n^{0.5}N)$ time 式中 N=min(m,nlogn]，m是所构造的二部图的边数[7]。而我们所研究的论文，它隐式的构造二部图，充分的利用了二部图的性质，将算法复杂度降到了O（nloglogn）。但该篇论文也有不足之处，它之所以能够取得如此低的算法复杂度，是因为它在预处理中采用了Tarjan在论文[2]中提出的finger search的数据结构，但在论文[2]中已经指出，该数据结构虽然能够降低复杂度，但在实现方面有很多困难。因此，在后来的一篇文章中，论文[3]中，作者提出了一种复杂度为O(nlogk)，其中k是多边形中有倒置关系的水平边与竖直边的数目。

在下面的章节中，首先给出算法和原理；其次，说明我们在实现这一系统中所遇到的问题及解决方法；在第四部分，介绍了系统的设计框架及重要的数据结构；第五部分是对算法复杂度和测试结果的分析；最后给出全篇的结论。

2 原理与算法

2.1 原理

定义 1 凹角 内角度数为 270^o 的角称为凹角。

定义 2 弦 当两个凹角的顶点处于同一水平线或竖直线时，连接这两个凹角的顶点的对角线，称为弦。

定理 1 在不存在弦的直角多边形中，它的剖分得到的矩形数目是一定的。

证明： 如果一个直角多边形的剖分结果中有凹角存在，则它一定没有剖分结束，因为任何一个矩形的内角都为 90^o。因此，矩形剖分的过程实质就是去凹角的过程。对任何一个凹角，从顶点引出一条水平线或是竖直线，都可以把它给破坏掉，然而，在不存在弦的直角多边形中，每一条从凹角顶点引出的对角线，只能构破坏掉一个凹角。由此可知，最初的多边形的凹角数目一定，引入的剖分线的数目就一定，因此，最后剖分所得到的矩形数目也一定。

命题得证。

由定理 1，我们可以知道，如果想获得最小的矩形剖分，就必须充分的利用弦的性质，即一次可以同时去掉两个凹角。

本算法的出发点就是找到一组最大无关弦集合，先用他们对多边形（本文中的多边形只指直角多边形）进行初步剖分，再对剖分结果进行普通剖分。

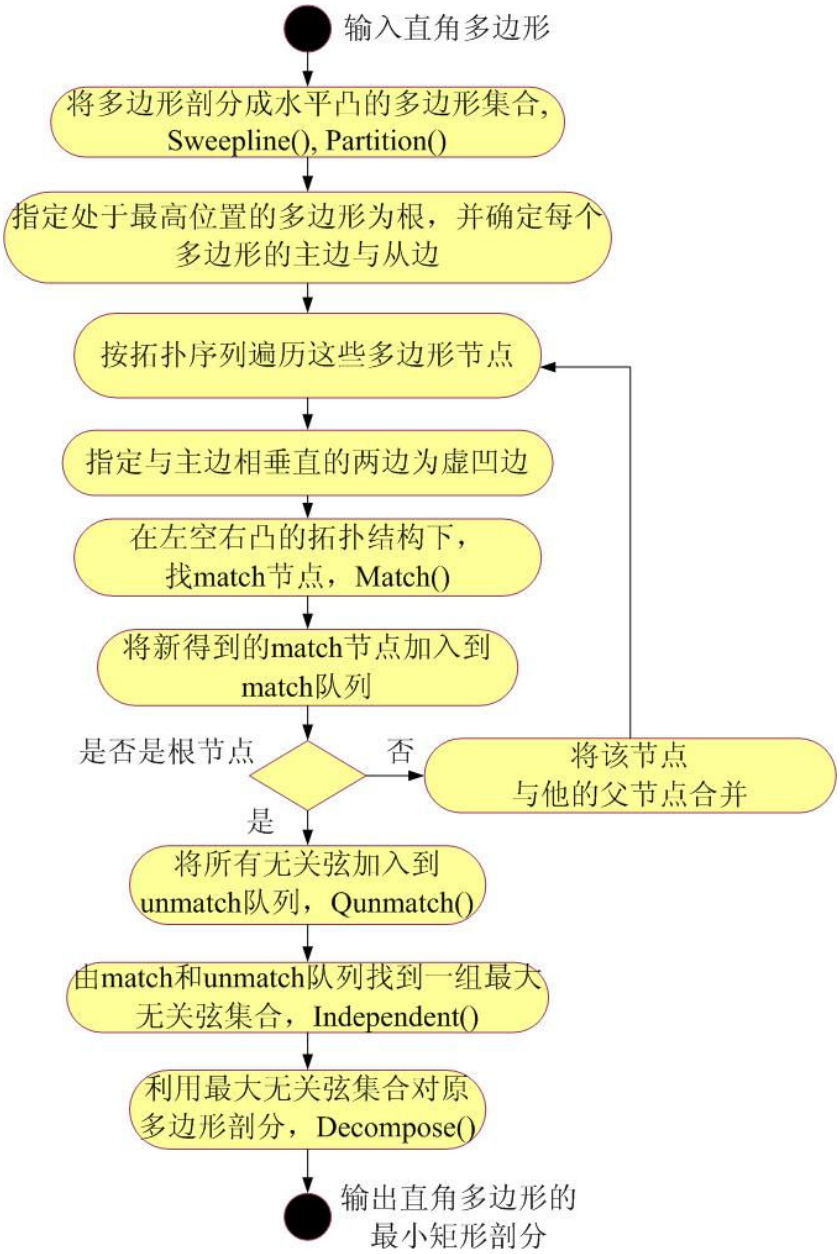
显然，弦有水平弦和竖直弦，他们之间必有相交的情况，也有一些弦，不域任何的其他弦相交，这些弦一定会被加入到最后最大不相关的弦集合中。关键就是如何从那些相交的弦中取出最大的一组不相交的弦。这一过程是通过二部图的 match 来实现的，与以往的算法不同，本论文采用了隐式的表示二部图的方法。

论文中详细的分析了二部图是如何保证了剖分的性质，在所构造的算法中，来隐式的保持这种性质，以保证剖分的最优性，这一部分，论文[1]中有详尽的证明，此不赘述。下面介绍所采用的算法：

2.2 算法

首先，论文中通过一条引理，确定在一定的拓扑结构下，所找到的 match 节点，与二部图找到的完全一致，这种拓扑结构称为左空右凸（left-free & right-convex）。因此，在算法的全过程中，每每要处理添加 match 节点时，都会保持着这种结构添加下去。

算法的总体结构图如下所示：



下面就每一部分的具体算法进行具体说明：

2．2．1　平面扫描找剖分点算法 Sweepline( )

2．2．1.1　算法目标

用平面扫描找到所有上凸和下凸的 Reflex 边延长后与多边形边的交点。

2．2．1.2　算法概述

a．算法处理的事件分为两类：

（1）所有垂直边的两个顶点：处理上顶点时将该垂直边插入平衡二叉树，作为状态保存。处理下顶点时将该垂直边从状态中删除。

（2）凸边。处理这类事件时，从状态中查找在水平方向离它两个端点最近的两条边，作为算法的结果存在 Partion 结构中，保存下来。

b．算法特殊性：

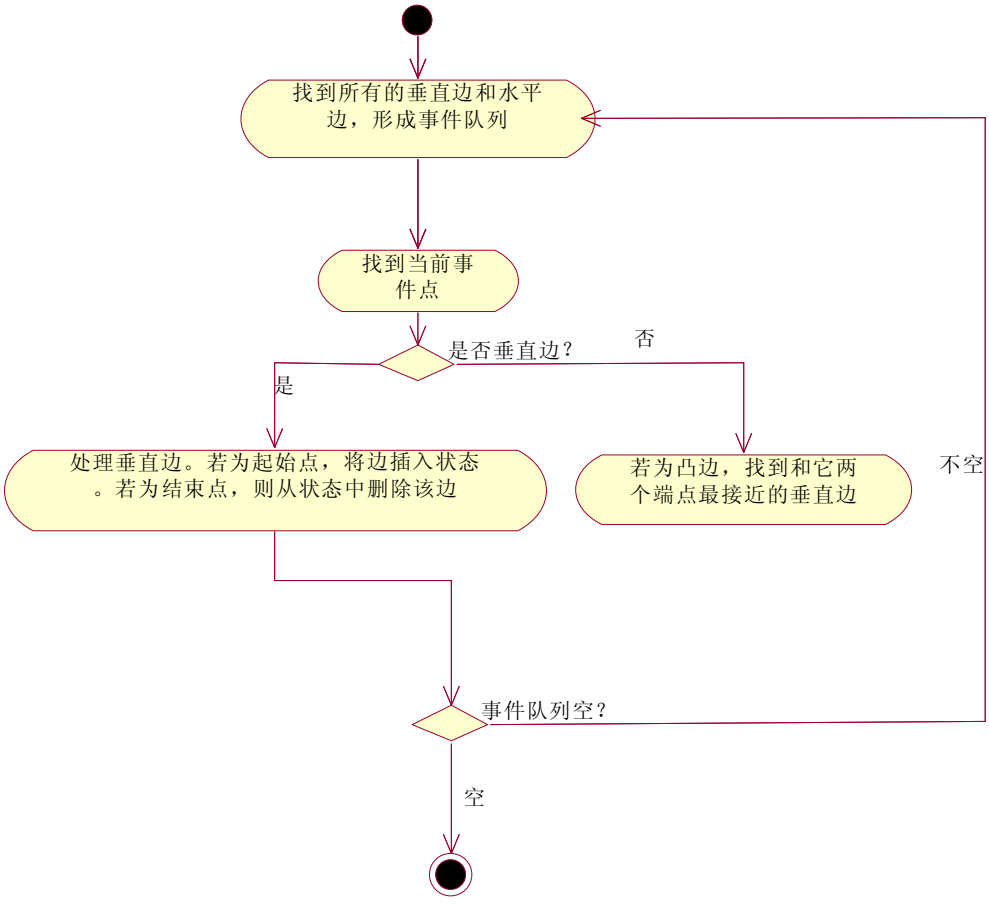
所有事件可在开始时确定，处理中不会产生新的事件。

c．算法用数据结构

（1）用链表存储事件队列

（2）用平衡二叉树存储状态

d．算法流程图



2．2．1.3　算法时间和空间复杂度分析：

a．设 n 为多边形顶点数。找到所有的垂直边和凸边只需遍历多边形一遍，时间复杂度为 $O(n)$，对事件点进行归并排序需时 $O(n\log n)$。使用平衡二叉树存

储状态，因此每次插入和删除只需 $O(\log n)$ 时间，而每条边最多执行插入、删除和查找一次，因此，处理所有事件仅需时间 $O(n\log n)$。因此，总的时间复杂度为 $O(n\log n+n)=O(n\log n)$。

b．存储事件队列需要空间为 $O(n)$，平衡二叉树占用空间最大为 $O(n)$，则总的空间复杂度为 $O(n)$。

2．2．2　将多边形剖分为水平凸子多边形的剖分算法 Partition( )

2．2．2.1 算法目标

通过凸边对应的最近的垂直边，将多边形剖分成水平凸子多边形。将这些水平凸多边形组织成一棵可以拓扑有序访问的孩子兄弟树。

2．2．2.2 算法概述

a．算法用数据结构

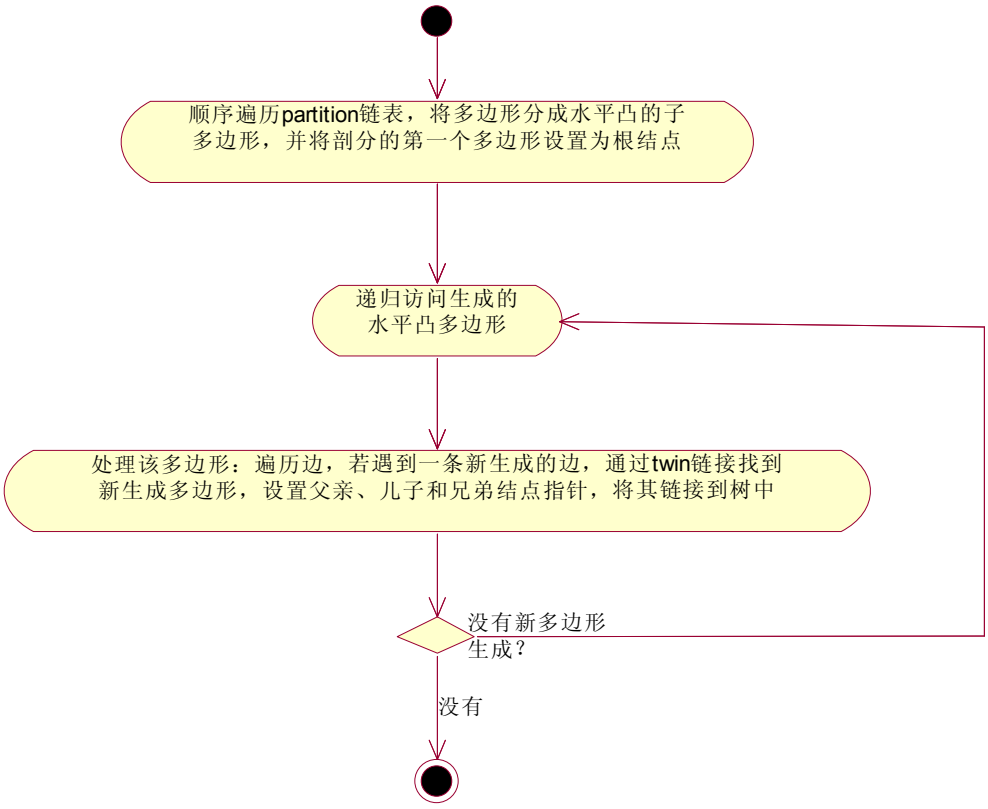
将子多边形存在孩子兄弟树中，为了访问方便，为每个结点设置父亲指针。

b．算法基本思想

将最高凸边剖成的多边形默认为根结点。顺序遍历 Partion 链表，将多边形剖分成多个水平凸的子多边形，将新生成的始末端点重合的边相互链接起来，作为 twin 边存在。

从根结点多边形的某条边开始遍历该水平凸子多边形，遇到一条新生成的边，则通过 twin 链接找到相邻的子多边形，建立新结点，设置好父亲、儿子和兄弟链接，然后递归遍历这个找到的子多边形。

c．算法流程图



2．2．2.3 算法复杂度分析

a.顶点数为 n 。

b.在将多边形剖分成水平凸多边形时，每个凸多边形新生成两条边，而每个水平凸多边形至少有两条原多边形的边，这些边不会被别的水平凸多边形所拥有，所以，总共生成的凸子多边形个数不超过 n 。因此，最终各个子多边形的总边数不超过 $O(n)$ 。

c.遍历一个子多边形时，每条边只处理一次，每次处理时，最多设置三个指针，需常数时间。因此，遍历每个子多边形时，时间复杂度为 $O(n_i)$ ，其中 n_i 为子多边形的边数。每个多边形仅访问一次，所以最终总的时间复杂度不超过总的边数，为 $O(n)$ 。

d.整个过程中孩子兄弟树需 $O(n)$ 的存储空间，所有的子多边形需 $O(n)$ 空间，递归访问需系统栈为 $O(n)$ ，因此，总的空间复杂度为 $O(n)$ 。

2．2．3 找到所有的 match 节点 Match( )

2．2．3.1 算法目标

输入是一个水平凸的多边形，要求找出它的所有 match 节点，按一定顺序相交的水平弦和竖直弦对。

2．2．3.2 算法概述

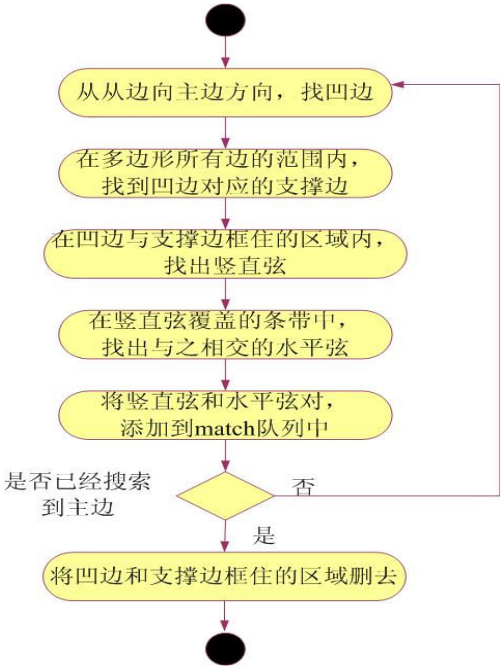
a. 算法思想

主要通过所搜到对应的凹边和支撑边，继而通过 zigzag，顺序找到竖直弦，在竖直弦扫过的区域内，找到的一个与之相交的水平弦；如是处理，直到处理完凹边和支撑边所包含的小区域，此时，由于这个区域内 的所有竖直弦全被处理了，可以将这部分区域删去，以去掉由对应的凹边所引起的 水平方向的非单调性。

b. 数据结构

为了保证不重复的搜索竖直弦所对应的水平弦，引入了一个双向队列 pdqhcnode，用于记录前面已搜到的存在于竖直弦对应的区域那的水平弦，及在搜索竖直弦的过程中，新添加的水平弦

c. 算法流程图



2．2．3.3 算法复杂度分析

a. 时间复杂度

对凹边的搜索只有一遍，搜过的水平凸的多边形的边不会搜第二次，因此这部分时间是线性的；对凹边所对应的支撑边的搜索看似 $O(n^2)$，但实际上，对每个边检验是否是支撑边也最多有一次，因为每次 处理完凹边支撑边对后，其所括住的区域要被删去，因此这部分是线性时间内可以完成的；同理，对竖直弦的搜索也是如此；关键是看，如何找到竖直弦对应的水平弦，这里引入了数据结构 pdqhcnode，用于累计前面已搜到的存在于竖直弦对应的区域那的水平弦，及在搜索竖直弦的过程中，新添加的水平弦，由此可以保证这一算法的整体时间复杂度为 O(n)。

b. 空间复杂度

这部分的存储结构仍若采用已有的边的链表，这部分的空间复杂度为 O(n)，新引入的数据结构 pdqhcnode,只用于记录所有的水平弦，其最大值不超过 n，故而这部分算法整体的空间复杂度为 O(n)。

2．2．4 后序遍历孩子兄弟树找无关弦的算法 Qunmatch( )

2．2．4.1 算法目标

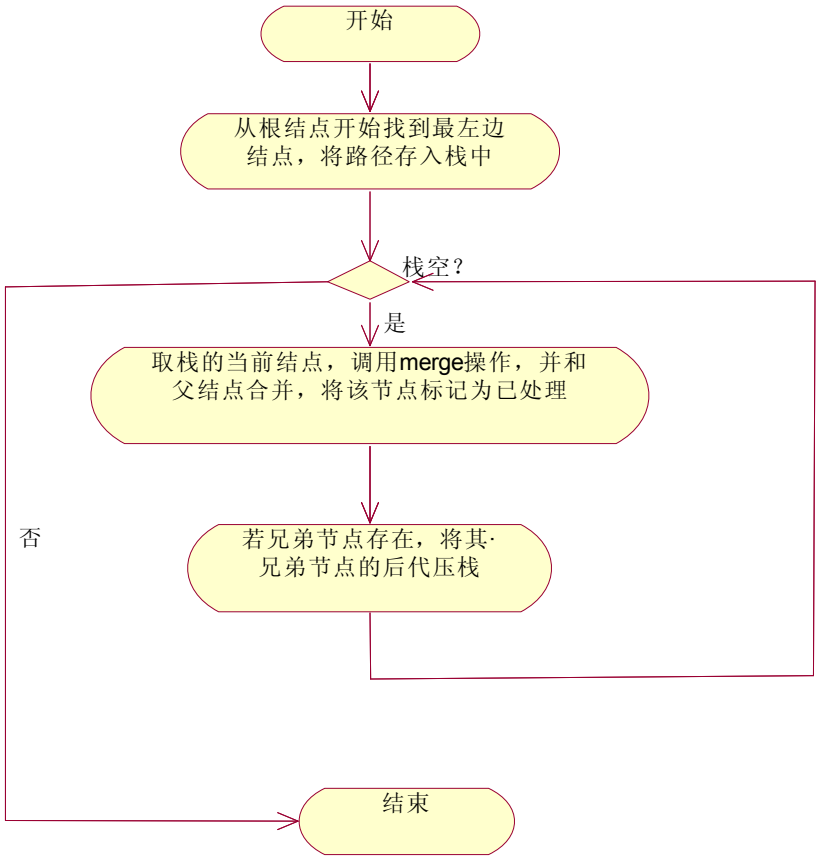
后序遍历生成的孩子兄弟树，遍历时实现找相交弦的操作。

2．2．4.2 算法概述

a．算法用数据结构

存储树结点的栈

b.算法流程图



2．2．4.3 算法复杂度分析

a.时间复杂度

设多边形顶点数目为n，则剖分成的水平凸多边形数不超过n，因此后序遍历时访问的结点数为$O(n)$。每个结点进行的操作是 merge 算法，该算法时间复杂度为$O(n_i)$，其中n_i是每个节点所代表多边形的顶点数。所以总的时间复杂度为$O(m+\sum_{i=1}^{m} n_i)$其中m为孩子兄弟树的结点数。按照上一个算法的论述，生成的所有水平多边形顶点总数不超过$O(n)$。所以，总的时间复杂度为$O(n)$。

b.空间复杂度

算法所用的数据结构为孩子兄弟树和栈，因此，总的空间复杂度$O(n)$。

2．2．5　找到一组最大无关弦集合 Independent( )

2．2．5.1 算法目标

由 match 和 unmatch 队列构造一组最大的不相关弦集合。

2．2．5.2 算法概述

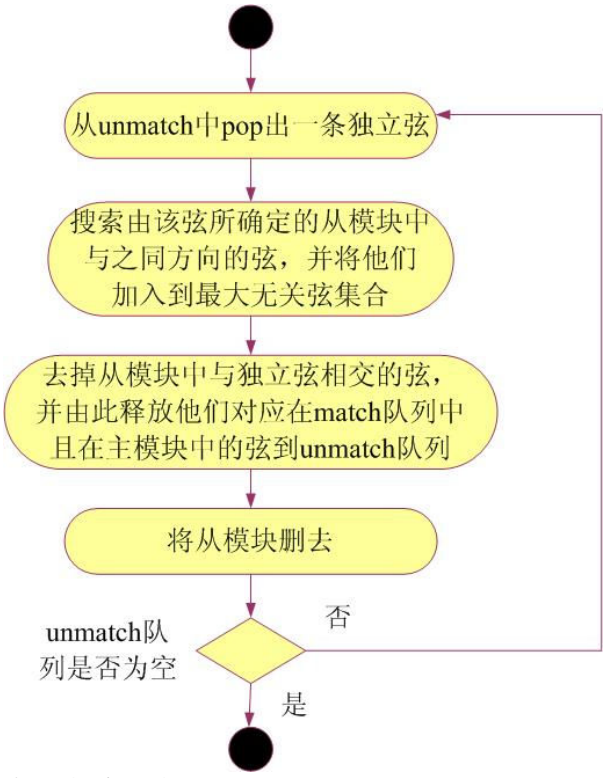
a.　算法思想

unmatch 队列中的弦是一定会被加入到最大的不相关弦集合中，关键在于如何从 match 队列中取出合适的弦。方法是，取出与当前已有的弦最兼容的。

b.　数据结构

不相关弦的集合，用单向队列存储。

c.　算法流程图



2．2．5.3 算法复杂度分析

a.　时间复杂度

由于对每个弦的处理最多为两次（从 match 队列中释放到 unmatch 队列，再从 unmatch 队列中被 pop 出来），对每个边的检测最多为一次，处理后便随着从模块被删除了，因此，这部分算法整体的时间复杂度为 O(n)

b. 空间复杂度

不相关弦的集合长度小于所有弦的数目，所有弦的数目又小于 n，故这部分整体的空间复杂度为 O(n)。

2．2．6 将多边形剖分为无弦子多边形算法 Decompose( )

2．2．6.1 算法目标

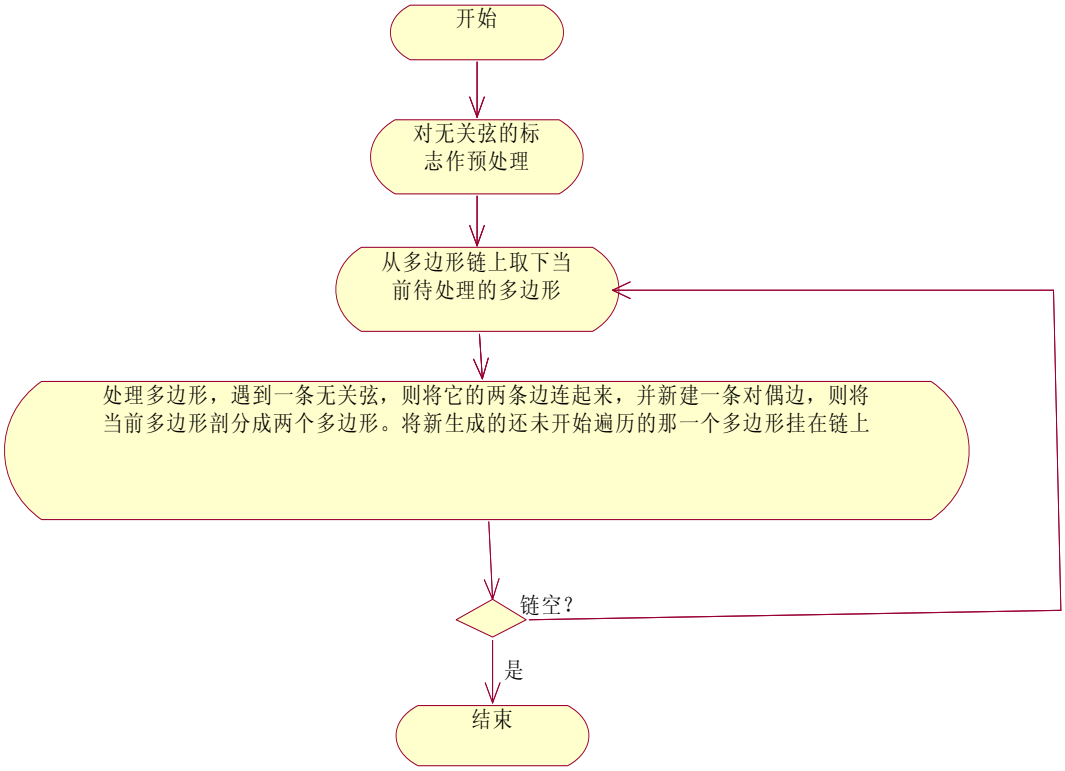
将多边形剖分成不存在弦的多边形

2．2．6.2 算法描述

a．算法思想

算法通过 merge 算法找到最大无关弦集合，利用递归算法连接这些弦，则可以得到一系列的没有弦的多边形，从这些多边形能得到最终结果。

b.算法流程图



2．2．6.3 算法复杂度分析

a. 时间复杂度分析

设 n 指多边形的顶点数。剖分的主要操作是对多边形做遍历，在遍历中连接找到的最大无关弦。最大无关弦的数量级不超过 $O(n)$，因此除遍历外的操作时间复杂度为 $O(n)$。而当我们剖分多边形时，每次总是将一个多边形剖分成两个多边形。每次总是把没遍历过的挂在链上，因此确保了遍历多边形时每个边仅仅遍历一遍，时间复杂度不超过 $O(n)$。综上所述，总的时间复杂度为 $O(n)$。

b.空间复杂度分析

该算法仅仅用一个链表保存剖分的多边形链，其个数上限为O(n)。剖分过程中新建的边是最大无关弦的两倍，复杂度为O(n)。所以，总的空间复杂度为O(n)。

3 难点及解决方法

3.1 定向问题

要处理封闭多边形的问题，定向至关重要。刚开始只是用点来存储多边形，因此处理问题时涉及到大量的计算和判断。为了解决这个问题，调整了多边形的存储方式，用边来存储，在边的结构中存储有该边对屏幕坐标的方向信息，这些信息可以在预处理时以线性时间一次计算出来。而增加的空间复杂度不超过$O(n)$，其中n为多边形的顶点数，所以这个存储结构并没有增加该程序的空间复杂度。

3.2 剖分问题和多边形拓扑排序

原算法是要在$O(n\log\log n)$时间内将正交多边形剖分成多个水平凸多边形，同时按照父子关系将这些多边形组织成一棵树便于找相交弦时用。由于原论文使用的算法相当复杂，所以我们采用平面扫描算法来实现。在开始解决这个问题时，准备在一次平面扫描的过程中决定子多边形的拓扑顺序，但是遇到很大的问题。因为在扫描到任何一条剖分边时，我们很难在$O(\log n)$的时间内确定它在已剖分的多边形中的相对位置。而且这个位置在剖分过程中还可能动态改变。

后来的解决办法是默认第一个剖分掉的多边形为树根，以此来确定各个多边形的拓扑顺序，将其组织成一棵孩子－兄弟树，以方便后序遍历。所以，才用了两遍扫描的算法，第一遍，确定多边形的所有剖分边。然后从默认的树根开始，以递归方式建立那棵孩子－兄弟树，拓扑顺序由递归过程和孩子－兄弟树的结构决定。这样就在$O(\log n)$的时间内解决了问题。

3.3 输入正交边多边形的问题

要画出一个正交多边形是个很困难的问题。所以，为了解决这个问题，用了两个办法：一是提供了数据与文本文件之间的导入导出功能，这样可以通过编辑文本文件，来得到复杂图形，可以方便的产生很多弦。二是，在绘图方面，使用了坐标调整方法。使用点定位和坐标的自动调整来实现多边形的闭合，还用对齐坐标到 5 的倍数的方法来产生更多的弦。

4 系统设计与数据结构

4.1 系统设计

程序主要分为两部分：前端和后端。

4.1.1 前端

前端主要用来与用户交互，进行绘图，显示剖分的过程和结果。主要使用微软的 MFC 类库来实现，基于文档－视图结构。

4.1.2 后端

后端用来处理数据，是全部用自己写的数据结构代码来实现。主要包括四个部分：

第一部分，预处理。主要的功能为将输入多边形剖分成水平凸的多边形，形成一棵孩子兄弟树，便于找最大无关弦时后序遍历。

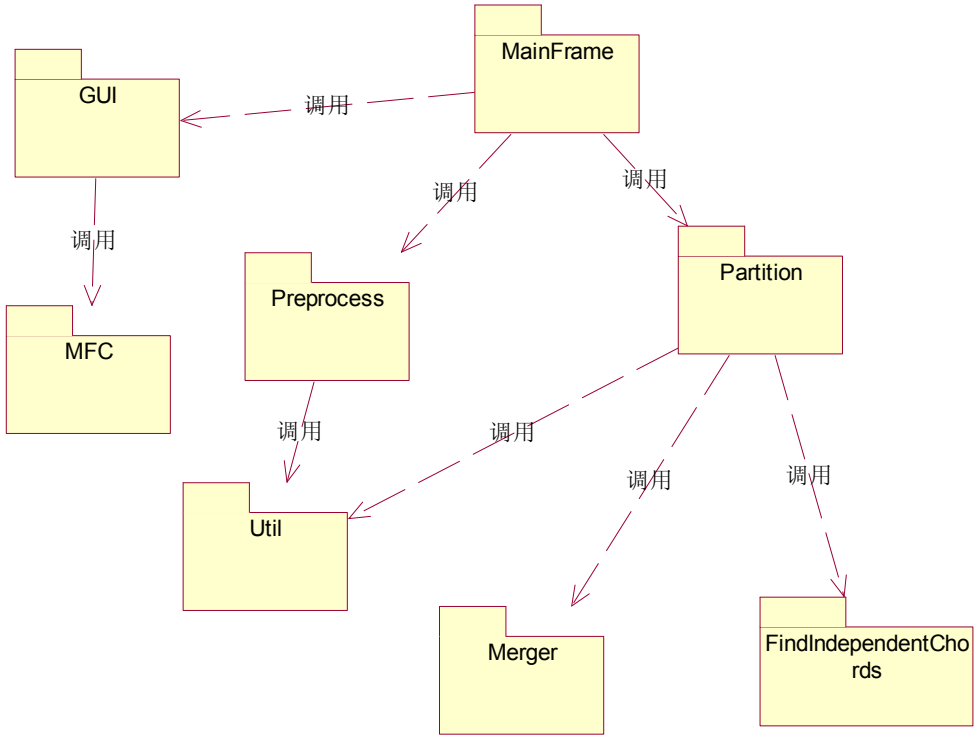
第二部分，找最大无关弦。这一部分首先找到相交弦，然后利用这个相交弦，结合二分图的性质，则可以得到最大无关弦。

第三部分，剖分。这一部分利用找到的最大无关弦，进行最后的剖分。

第四部分，工具类。该部分主要包括复杂的数据结构，主要为平衡二叉树。

4.1.3 系统框架结构

模块之间的调用关系如下图所示



4.2 数据结构

4.2.1 存储基本几何信息的数据结构

4.2.1.1 存储多边形点的结构

```
typedef struct Vertex{
int intX;
int intY;
int intInd;
int intChk;
int intSeq;
int intVerticalVisible;
struct Edge* pPrevEdge;
struct Edge* pNextEdge;
struct Chord* pchdHorizon;
struct Chord* pchdVertical;
struct Vertex* pNext;
```

```
 struct Vertex* pPrevNode;
 struct Vertex* pNextNode;
 VertexType vtType;
} Vertex, *PVertex;
```

4.2.1.2 存储多边形边的结构

```
typedef struct Edge{
 int cut;
 int bound;
 DirectionType drtDir;
 EdgeType etType;
 PVertex pvtStart;
 PVertex pvtEnd;
 PVertex pLeft;
 PVertex pRight;
 PVertex pTop;
 PVertex pBottom;
 struct Edge* pHitTop;
 struct Edge* pHitBottom;
 DQHonChordNode* pdqhcNode;
 struct Edge* pReflexto;
 struct Edge* pAdajPrev;
 struct Edge* pAdajNext;
 struct Edge* pNext;
 struct Edge* pTwin;
}Edge,*PEdge;
```

4.2.1.3 存储多边形弦的结构

```
typedef struct Chord{
 ChordType chdType;
 int intSeq;
 int intVisible;
 int intUse;
 int intLength;
 PVertex pvtLeft;
 PVertex pvtRight;
 PVertex pvtTop;
 PVertex pvtBottom;
 PVertex pvtStart;
 PVertex pvtEnd;
 PMatchNode pIntersectNode;
 struct Chord* pNext;
 struct Chord* pPrev;
 struct Chord* pNextIndep;
 struct Chord* pPrevIndep;
}Chord,*PChord;
```

4.2.1.4 存储多边形的结构

```
typedef struct CPolygon{
public:
```

```
 PolygonType plgConvexType;
 int inUse;
PEdge pMastBound;
PEdge pFirstSlaver;
PEdge pLastSlaver;
PEdge pLeftChain;
PEdge pRightChain;
PEdge pTopSupportEdge;
PEdge pBottomSupportEdge;
PEdge pVirtualReflexLeft;
PEdge pVirtualReflexRight;
PChord pFirstChord;
PChord pLastChord;
PEdgeVertexPair pLeftPair;
PEdgeVertexPair pRightPair;
PPartion ptnList;
CPolygon* pNext;
}CPolygon;
```

4.2.2 处理中使用的数据结构

4.2.2.1 平面扫描算法中存储状态的平衡二叉树

```
template <class Type> class AVLTree{
public:
  struct AVLNode{
      Type data;
        AVLNode *left,*right;
        int balance;
        AVLNode():left(NULL),right(NULL),balance(0){}
              AVLNode(Type d,AVLNode* l=NULL,AVLNode*
                r=NULL):data(d),left(l),right(r),balance(0){}
         };
protected:
  Type RefValue;
  AVLNode* root;
  int Insert(AVLNode* &tree,Type x,int &taller);
  int Delete(AVLNode* &tree,Type x,int &shorter,AVLNode* parent);
  void RotateLeft(AVLNode* Tree,AVLNode* &NewTree);
  void RotateRight(AVLNode* Tree,AVLNode* &NewTree);
  void LeftBalance(AVLNode* &Tree,int &taller);
  void RightBalance(AVLNode* &Tree,int &taller);
  int Height(AVLNode* t)const;
public:
  AVLTree():root(NULL){}
  AVLTree(Type Ref):RefValue(Ref),root(NULL){}
  int Insert(Type x){int taller; return Insert(root,x,taller);}
  int Delete(Type x){int shorter;return Delete(root,x,shorter,NULL);}
  int FindMaxLower(Type dt,Type &Res);
  int FindMinHigher(Type dt,Type &Res);
  int Height() const;
};
```

4.2.2.2 存储多边形结构按拓扑有序访问的兄弟树

```
typedef struct TriNode{
      CPolygon* pData;
         TriNode *child,*nextSibling;
         TriNode* parent;
}TriNode;
```

4.2.2.3 存储剖分结果的数据结构

```
typedef struct Partion{
PEdge pLeft;
PEdge pRight;
PEdge lstEdge;
Partion* pNext;
int single;
Partion():pLeft(NULL),pRight(NULL),lstEdge(NULL),single(-1){};
}Partion,*PPartion;
```

4.2.2.4 存储剖分节点的数据结构

```
class Node{
public:
 CPolygon* pPolygon;
 QMatch* pQMatch;
 QUnMatch* pQUnMatch;
 PEdge pLeft;          //always be ver
 PEdge pRight;         //always be ver
 PEdge pLeftSupport;
 PEdge pRightSupport;
 PEdge pUpperChain;      //always be hon
 PEdge pLowerChain;      //always be hon
 DQHonChordNode* pdqhcNode;
public:
 KEEPRULE();         //keep vertical and honrizon alternately
 Node(CPolygon &P,QMatch &M,QUnMatch
&U):pPolygon(&P),pQMatch(&M),pQUnMatch(&U),pLeft(NULL),

   pRight(NULL),pLeftSupport(NULL),pRightSupport(NULL),pUpperChain(NUL
L),pLowerChain(NULL)
 {
       pdqhcNode = new DQHonChordNode;
       KEEPRULE();
       MATCHING();
 };
 Left_UpperChain_Move_Up();
 Left_UpperChain_Move_Down();
 Left_LowerChain_Move_Up();
 Left_LowerChain_Move_Down();
 Right_UpperChain_Move_Up();
 Right_UpperChain_Move_Down();
 Right_LowerChain_Move_Up();
```

```
Right_LowerChain_Move_Down();
int Top_Bottom_Left();
int Top_Bottom_Right();
int Bottom_Top_Left();
int Bottom_Top_Right();
MATCHING();
DESTROYRULE();
~Node();
};
```

4.2.2.5 存储 match 的数据结构

```
class QMatch {
public:
 PMatchNode pHead;
 PMatchNode pEnd;
public:
 QMatch():pHead(NULL),pEnd(NULL){};
 PMatchNode Pop();
 Push(PMatchNode);
 ~QMatch();
};
```

4.2.2.6 存储 unmatch 的数据结构

```
class QUnMatch {
public:
 PChord pHead;
 PChord pEnd;
public:
 QUnMatch():pHead(NULL),pEnd(NULL){};
 PChord Pop();
 Push(PChord);
 ~QUnMatch();
};
```

4.2.2.7 存储最大无关弦集合的数据结构

```
class IndepSet
{
public:
 CPolygon* pPolygon;
 QIndep* pQIndep;
 PUnMatchNode pOrientSet;
 PUnMatchNode pNHSet;
 PUnMatchNode pWSet;
 QMatch* pQMatch;
 QUnMatch* pQUnMatch;
 POpenLink SB;          //construct in SHRINK_TEST();
 PChord pU;
public:
 IndepSet(CPolygon &P,QMatch &QM,QUnMatch
&QUM):pPolygon(&P),pQMatch(&QM),pQUnMatch(&QUM),
```

```
        pOrientSet(NULL),pNHSet(NULL),pWSet(NULL)
{
        pQIndep = new QIndep;
        SB = new OpenLink;
        CONSTRUCT();
};
VISIBLE_CHORD();
int SHRINK_TEST();
TRAVERSE();
SHRINK();
CONSTRUCT();
~IndepSet();
};
```

5 复杂度分析与测试

5.1 复杂度分析

在每一步具体的算法中已有详尽的复杂度分析，这里只是综合起所有分步，看整体算法的复杂度。

5.1.1 时间复杂度分析

不难看出，整个算法时间复杂度的瓶颈为 Sweepline( )函数所确定的 O(nlogn)。其它所有的处理都可以在 O(n)时间内完成, 故而算法的时间复杂度为 O(nlogn)。

5.1.2 空间复杂度分析

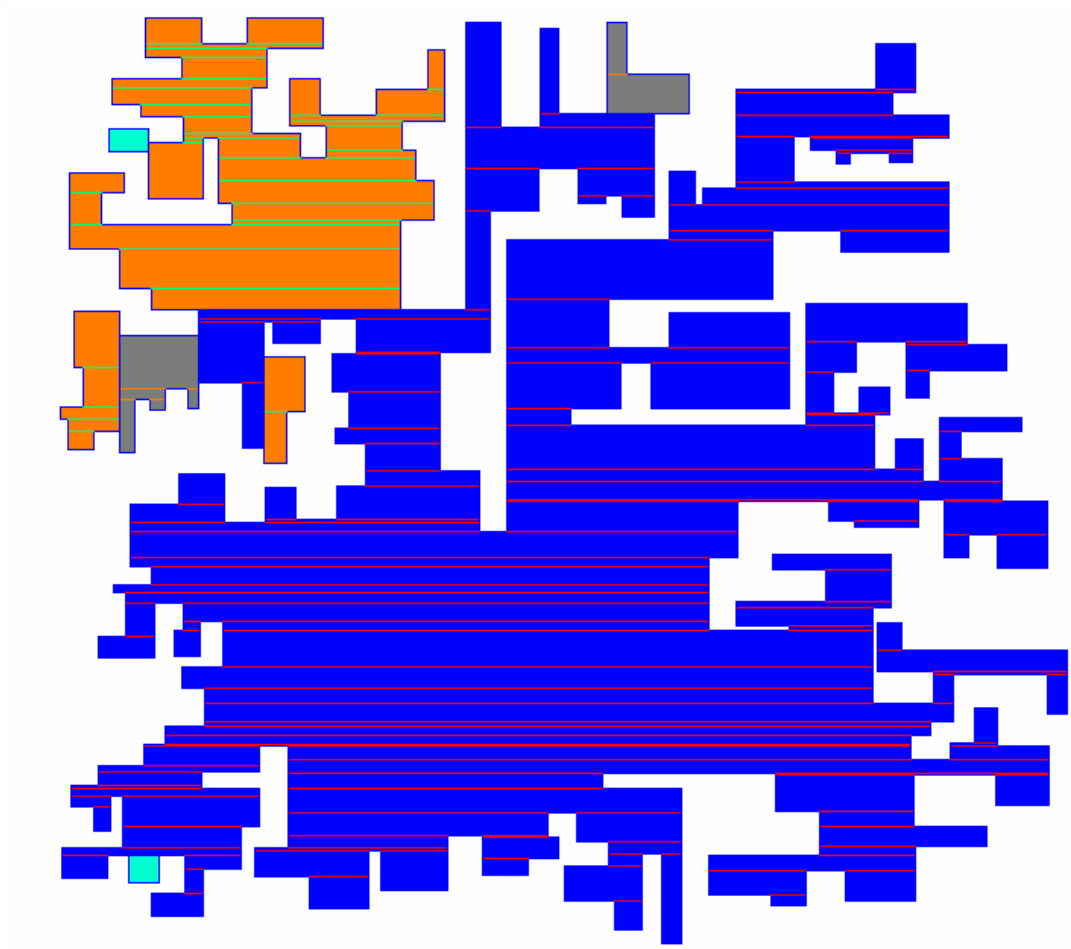
综合所有的处理来看，都可以在 O(n)的空间内完成，故而算法的空间复杂度为 O(n)。

5.2 测试

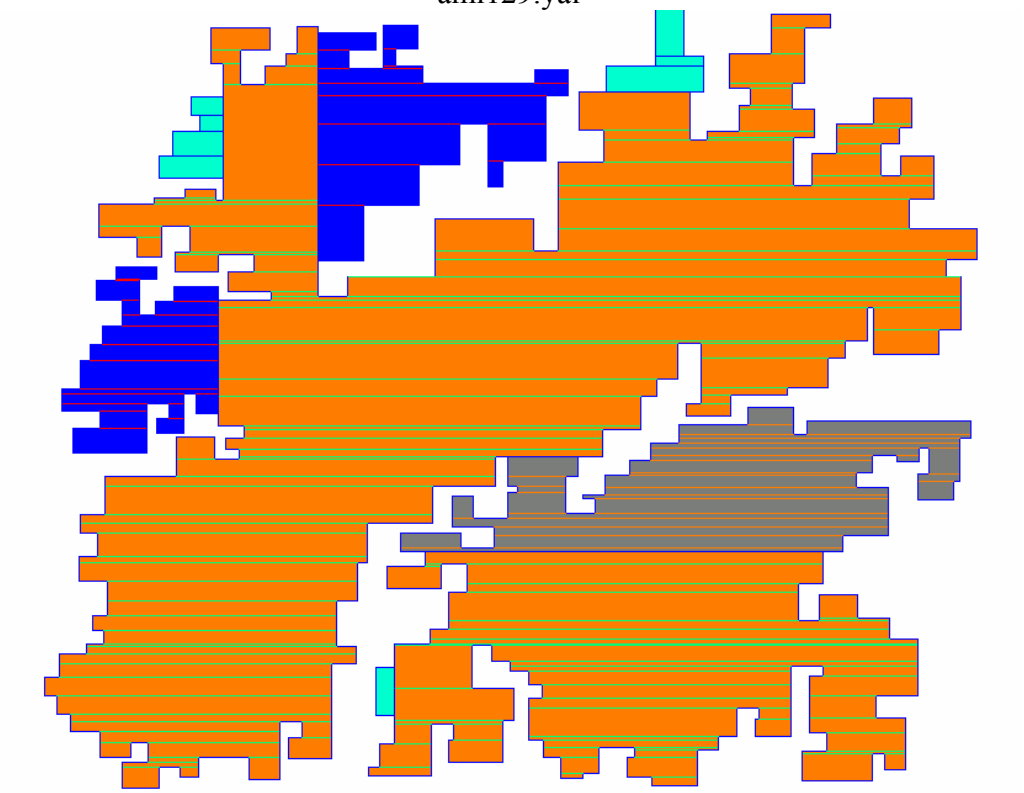
本论文所研究的对于掩模板的光刻，一般是对全芯片（full-chip）级的，即是 block 间的，而非 block 内部 cell 间的。二者的区别在于，前者的模块数较少。目前通常为 100 个模块左右。

在本文中所采用的例子为 BBL 的 yal 文件。

经过该系统分别得到的结果如下所示：



ami129.yal



Ami189.yal

6 结论

6.1 针对检查时的改进

检查时，存在的不足是，（1）显示效果可以进一步增强（2）测试例子最好采用工业实例。

针对以上两点，这段时间又对程序进行了调整：（1）显示方面：即使被 shrink 调的部分仍会出现在显示区，利于理解程序在处理节点时的过程。（2）例子方面：经与实验室的老师商定，采用了 BBL 的 yal 文件，主要测试了 ami129.yal 和 ami189.yal 两个代表性例子。

6.2 结果分析

两个例子的运行时间都<0.01s，完全符合工业上的要求。

6.3 结论

原论文中采用了不易实现的 finger search 数据结构，致使尽管其在复杂度上有很大的优势，也终究未被广泛的采用。在我们的实现过程中，采用了替换策略，用 sweep line 的方法，虽然复杂度由 O(nloglogn)降到了 O(nlogn)，但能够实现了，而且运行时间也可以接受，比该论文以前提出的现在工业仍采用的方法，复杂度降了很多，因此很有实际意义。

参考文献

[1]W. T. Liou, J.J.M. Tan and R.C.T. Lee Minimum partitioning simple rectilinear polygons in O(nloglogn) – time. In Proc.5th Annu ACM Sympos. Comput. Geom., pages 344—353 , 1989
[2]R.E. Tarjan, and C.J. Van Wyk, An O(nloglogn) time Algorithm for Triangulating Simple Polygons, SIAM Journal on Computing 17(1988), 143-178.
[3]S. Nahar and S. Sahni Fast algorithm for polygon decomposition. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 7:473—483, 1988
[4]Polygon Decomposition (technical report) J. Mark Keil Department of Computer Science University of Saskatchewan Saskatoon. Sask,. Canada. S7N 5A9 May 14, 1996
[5] A. Lingas, R. Pinter, R. Rivest and A. Shamir. Minimum edge length partitioning of rectilinear polygons. In Proc20th Allerton Conf. Commun. Control Comput,. Pages 53—63,1982
[6] E. Hopcroft, and R.M. Karp An $n^{5/2}$ Algorithm for Maximum Matchings in bipartite Graph' SIAM Journal on Computing 2(1973), 225-231.
[7] H. Imai, and T. Asano, 'Efficient Algorithms for Geometric Graph Search Problems', SIAM Journal on Computing 15(1986), 478-494.

作者

冯哲 计算机系 04 硕 z-feng04@mails.tsinghua.edu.cn
张京城 计算机系 04 硕 jc-zhang04@mails.tsinghua.edu.cn