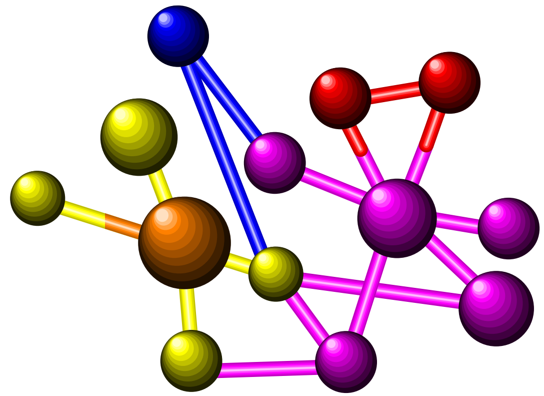
西北工业大学

本科专业课程讲座



计算几何算法与应用

Computational Geometry Algorithms and Applications

课程总结

第一讲 导言

1、何为计算几何？

2、凸包的例子

3、退化及鲁棒性

1、何为计算几何？

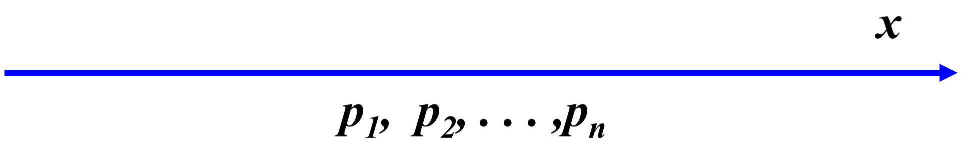
- 计算几何（ Computational Geometry ），针对处理几何对象的算法及数据结构的系统化研究，重点在于“渐进快速的精确算法”

几何算法的建立过程

- **第一阶段：**忽略对正在处理的几何概念理解有干扰的问题，例如多点共线、线段垂直等退化情况，初步建立算法步骤
- **第二阶段：**对前一阶段所设计的算法进行调整，使之对于退化情况也能正确处理
  - ✓ 除开引入大量的特殊情况外，还可以考虑对问题的几何性质进行再次分析，将各种特例归结到一般情况中去
- **第三阶段：**具体实现，需要考虑基本的操作，如测试某个点是位于一条有向直线的左侧、右侧、还是线上
  - ✓ 对实数进行精确运算时不现实的，由此带来对算法的调整，使之能够检测可能出现的不一致性，并采取适当的措施以避免程序崩溃。但如此一来，就不能保证算法的输出一定正确——鲁棒性问题

2、凸包的例子

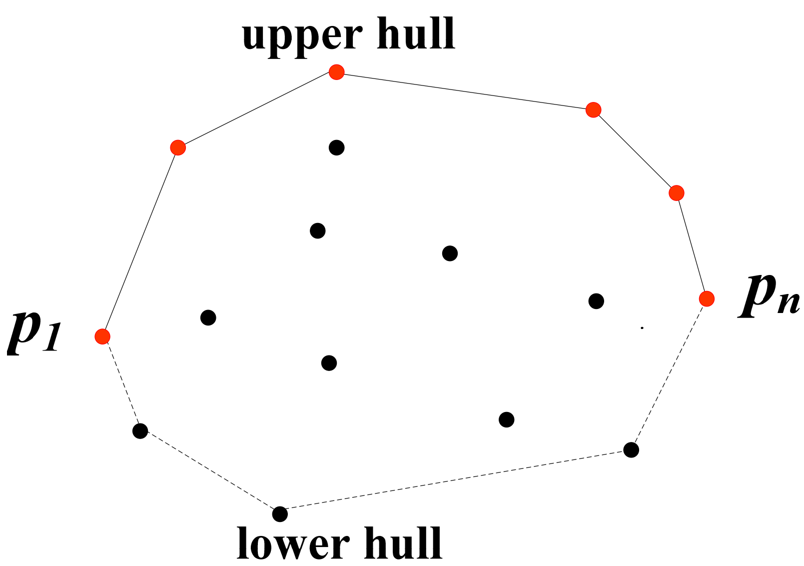
- **递增式策略：**逐一引入P中各点，每增加一个点，都相应更新当前的解；沿用几何上的习惯，按由左向右的次序加入各点



2、凸包的例子

算法的改进

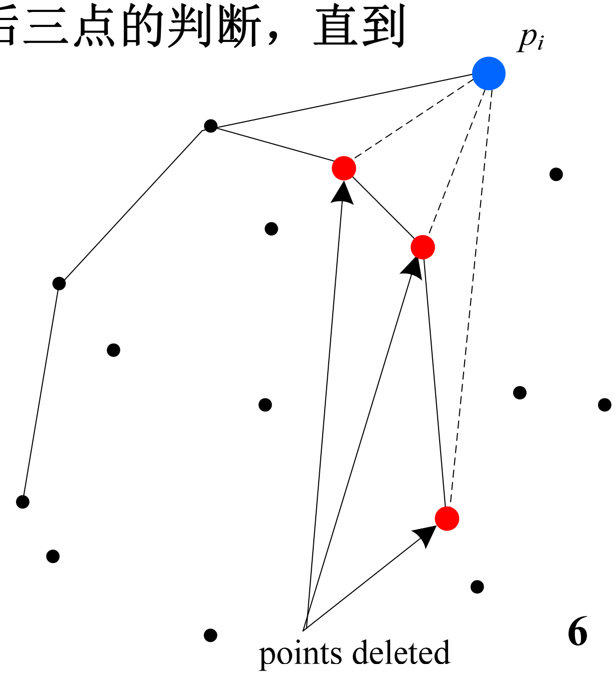
- 首先，自左向右计算上凸包，就是从左端顶点p_1出发，沿着凸包顺时针进行到最右端顶点p_n之间的那段
- 然后，自右向左计算下凸包



2、凸包的例子

算法的改进

- 沿多边形边界顺时针行进，在每个顶点都要改变方向，若是凸多边形，则必然每次都是向右转。根据这一点，在新引入p_i之后，进行如下处理：
  - 令L_{upper}为从左向右存放上凸包各顶点的一个列表，首先将p_i接在L_{upper}的最后，然后判断L_{upper}最后三个点是否形成右拐，若是左拐则要将中间的顶点从上凸包中删去。重复进行最后三点的判断，直到构成一个右拐或仅剩下两个点



2、凸包的例子

退化情况的处理

- 隐含假设：所有点的x坐标互异。改进：采用字典序而不仅仅是x的坐标来排序，即首先按照x坐标排序，若存在多个点x坐标一样，则再按照y坐标排序

采用字典序

- 对于数字1、2、3、…、n的排列，不同排列的先后关系是从左到右逐个比较对应数字的大小来决定的。例如：对于5个数字的排列12354和12345，排列12345在前，排列12354在后。按照这样的规定，5个数字的所有的排列中最前面的是12345，最后面的是54321

第一讲：导言—算法分析体系及计量

1、算法分析的评价体系

2、算法的时间复杂性

3、时间复杂度的估算

4、算法的空间复杂性

5、NP完全问题

6、算法分析实例

1、算法分析的评价体系

算法评价

- **算法的维护的方便性：** 可读性、通用性、可重用性和可扩充性
- **算法运行的时间和空间效率：** 占用空间少而且执行时间短

算法分析: 对设计出的每一个算法，利用数学工具讨论其复杂度

2、算法的时间复杂性

与算法时间相关的因素

- 采用的数据结构
- 算法采用的数学模型
- 算法设计的策略
- 问题的规模
- 实现算法的程序设计语言
- 编译算法产生的机器代码的质量
- 计算机执行指令的速度

3、时间复杂度的估算

算法效率的衡量方法

算法 = 控制结构 + 原操作(固有数据类型的操作)

算法的执行时间=Σ 原操作的执行次数 * 原操作

- 一个算法所耗费的时间,除了与所用的计算软、硬件环境有关外，主要取决于算法中指令重复执行的次数，即与语句频度相关

3、时间复杂度的估算

常见的时间复杂度表示

- **常见算法时间复杂度：**

  O(1): 表示算法的运行时间为常量

  O(n): 表示该算法是线性算法

  $O(\log_2 n)$: 二分查找算法

  $O(n^2)$: 对数组进行排序的各种简单算法，例如选择排序。

  $O(n^3)$: 做两个**n**阶矩阵的乘法运算

  $O(2^n)$: 求具有**n**个元素集合的所有子集的算法

  O(n!): 求具有**N**个元素的全排列的算法

$$O(1)<O(\log_2 n)<O(n)<O(n^2)<O(2^n)$$

6、算法分析实例

- 【例2】抽象地考虑以下递归方程，且假设n=2^k，T(1)=O(1)，则迭代求解过程如下：

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + 2 \\
&= 2\left(2T(\frac{n}{2^2}) + 2\right) + 2 \\
&= 4T(\frac{n}{2^2}) + 4 + 2 \\
&= 2^3 T(\frac{n}{2^3}) + 8 + 4 + 2 \\
&\cdots\ \cdots \\
&= 2^k T(1) + \sum_{i=1}^{k} 2^i \\
&= 2^k O(1) + (2^k - 1) = n(O(1) + 1) - 1 \\
&= O(n)
\end{aligned}
$$

6、算法分析实例

- **一般地，当递归方程为：T(n)= aT(n/c)+O(n)，T(n)的解为：**

  ① $O(n)$ ，**a<c且c>1 时**

  ② $O(n\log_c n)$ ，**a=c且c>1时**

  ③ $O(n^{\log_c a})$ ，**a>c且c>1时**

第二讲：线段求交：专题图叠合

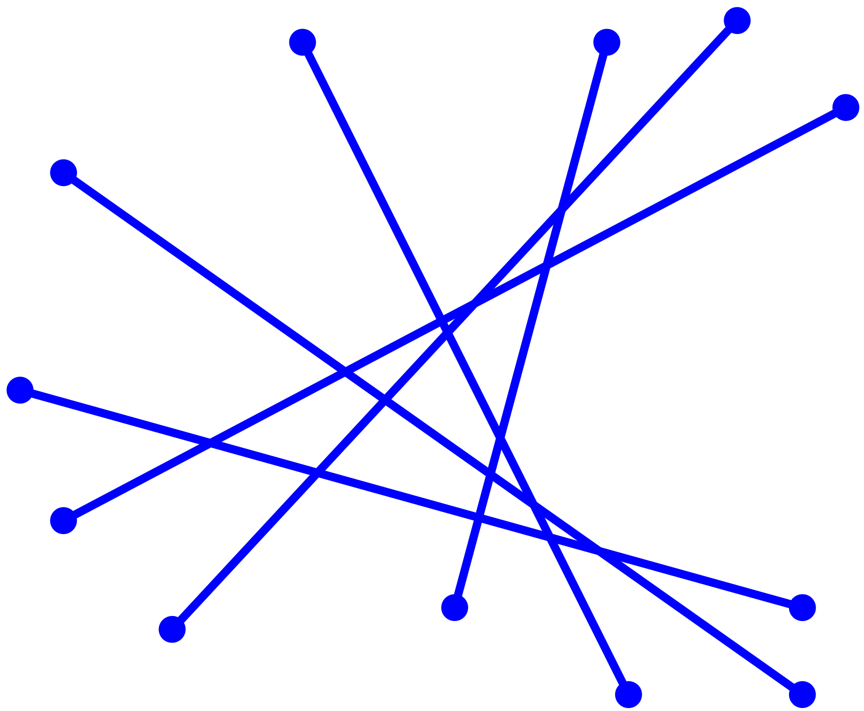
1、线段求交

2、双向链接边表

3、计算子区域划分的叠合

1、线段求交

"输出敏感的" 算法



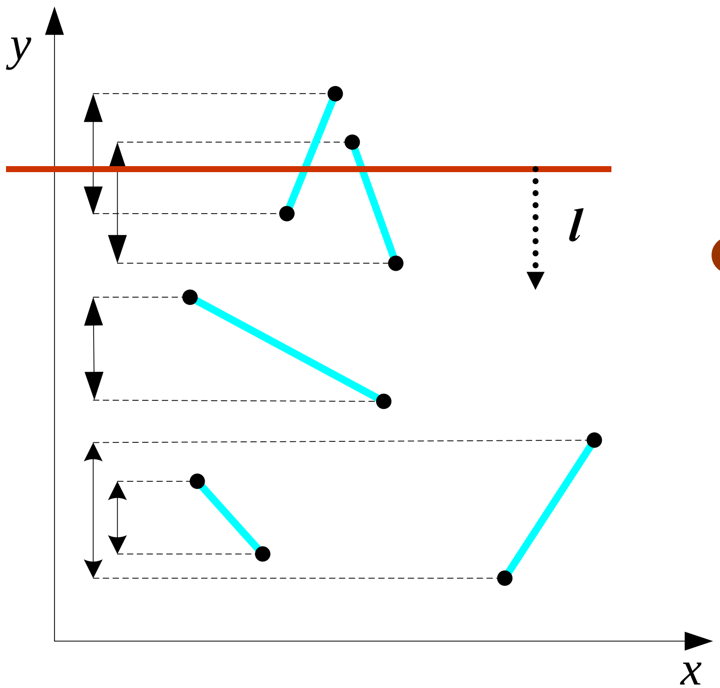
- 如图所示，每两条线段都相交，无论采用什么算法，<u>至少都需要Ω(n^2)时间(组合)</u>
- 在实际环境中，大多数的线段要么根本不与其它线段相交，要么只与少数线段相交，<u>交点总数远远达不到平方量级</u>

- 我们希望得到的算法，其运行时间不仅取决于输入线段的数目，还取决于实际交点的数目。这样的算法称为<u>"输出敏感的" 算法</u>（**output-sensitive algorithm)**

1、线段求交

平面扫描法

- 为找出这些线段对，可以想象用<u>一条直线*l*，从一个高于所有线段的位置起，自上而下地扫过整个平面</u>。在这条假想的直线扫过平面的过程中，跟踪记录所有与之相交的线段，以找出所需的所有线段对



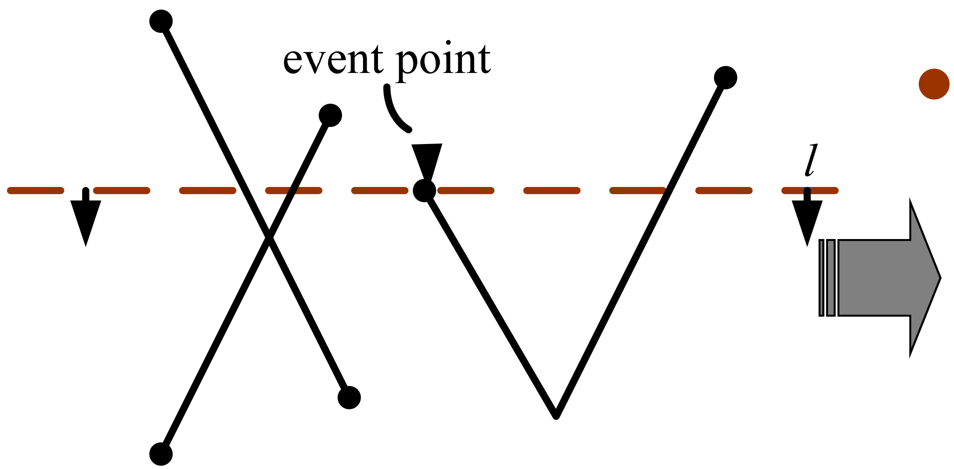
线段在y轴上的正交投影

- 得出一个解决线段求交问题的输出敏感的算法——<u>平面扫描算法（**plane sweep algorithm**）</u>

1、线段求交

平面扫描法

- *l* 被称为扫描线（sweep line），与当前扫面线相交的所有线段构成的集合，称为扫描线的状态（status）。随着扫描线的向下推进，它的状态不断变化，但变化并不是连续的

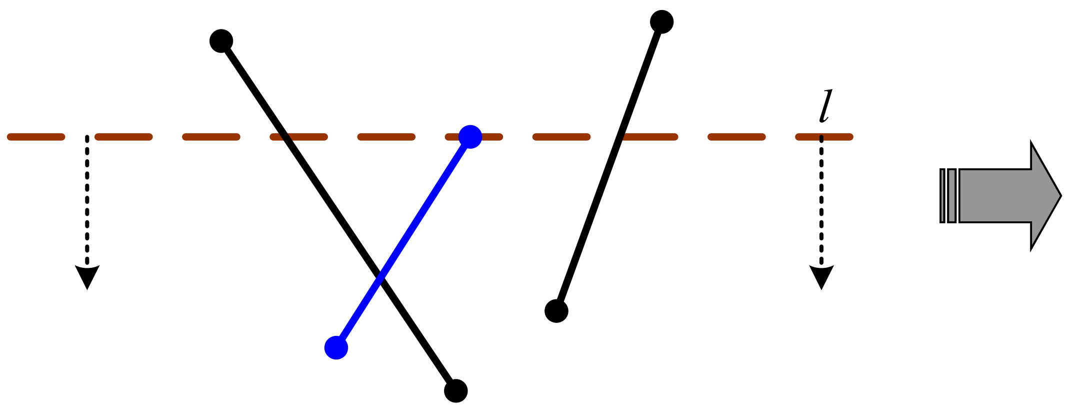


- 只有在某些特定位置，才需要对扫描线的状态进行更新，这些位置被称为平面扫描算法的事件点（event point），如线段端点

1、线段求交

平面扫描法

- 观察：与同一扫描线相交的两直线，在水平方向上仍然有可能相距很远，多数情况下并不相交
- 在考虑到水平方向的临近性后，我们可以沿着扫描线，将与之相交的所有线段自左向右排序。这样，只有当其中的某两条线段沿水平方向相邻时，才需要对其进行测试



每引入一条线段，只需要将其端点相邻的两条线段进行测试

1、线段求交

算法时间复杂度

- 在处理每一个事件时，我们至多对扫描线状态结构和事件队列执行常数个操作（每个操作需要 O(log$n$) 的时间）
- 扫面过程中所有 $2n+I$ 个事件处理的总时间为：

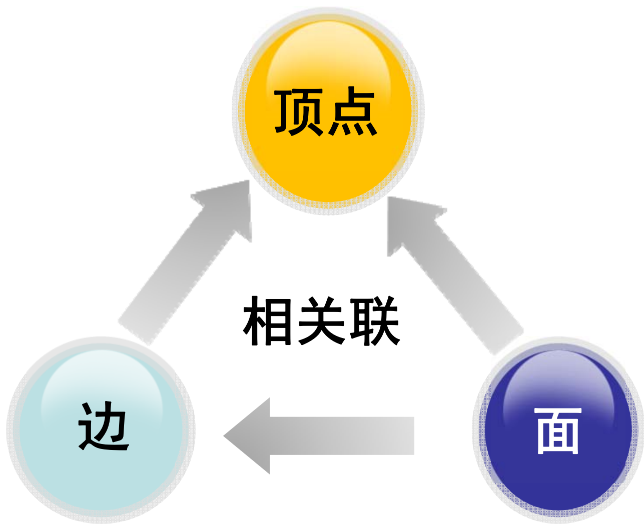
$$O((2n+I)\log n) = O((n+I)\log n) = O(n\log n + I\log n)$$

【定理 2.4】 给定由平面上任意 n 条线段构成的一个集合 S。可以在 $O(n\log n + I\log n)$时间内，使用 $O(n)$空间，报告出 S 中各线段之间的所有交点，以及与每个交点相关的所有线段。其中，I为实际的交点总数

2、双向链接边表

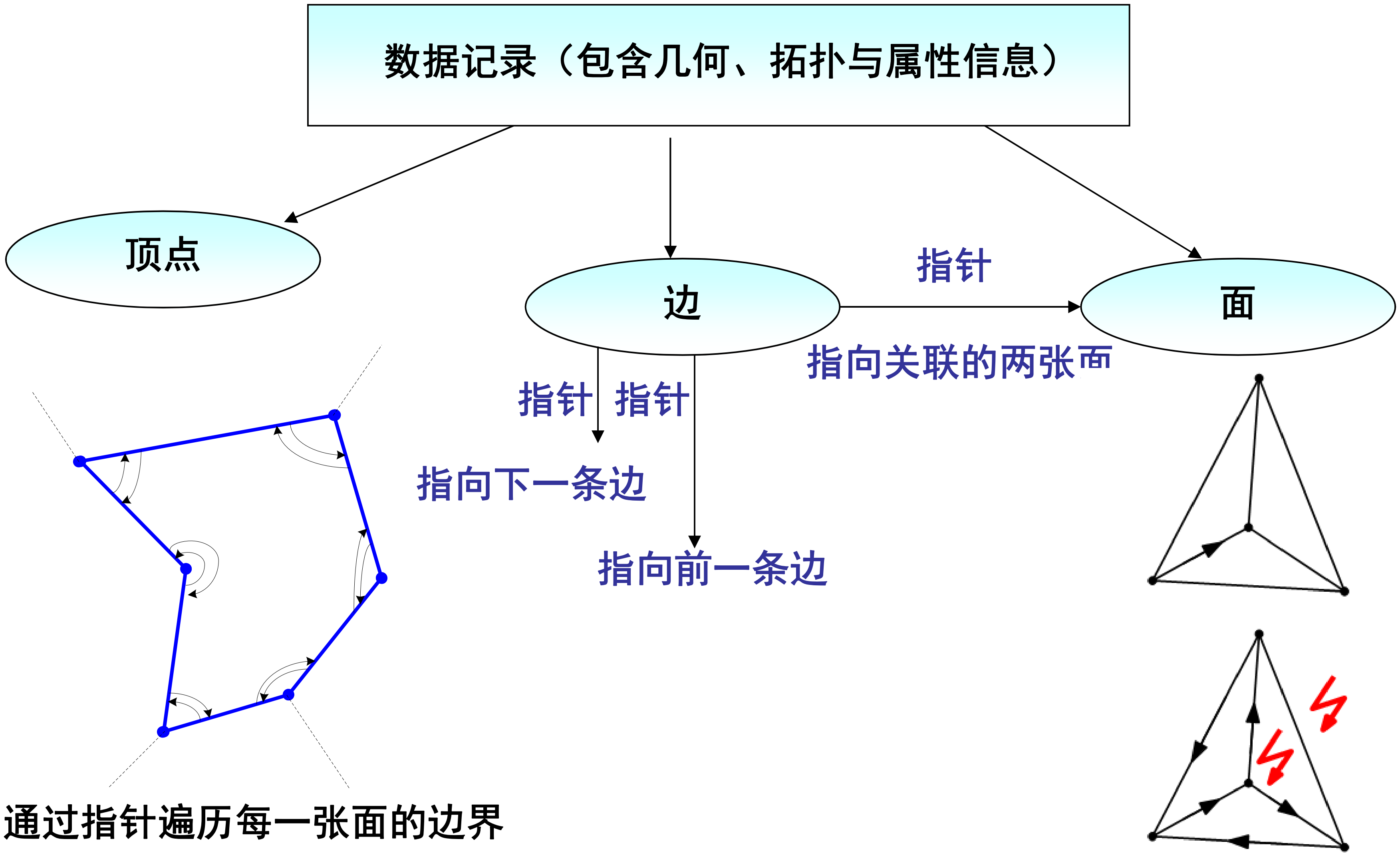
平面嵌入

- 一个子区域划分的复杂度，就是构成该子区域划分的<u>顶点、边和面的总数</u>



2、双向链接边表

双向链接边表——基本思想

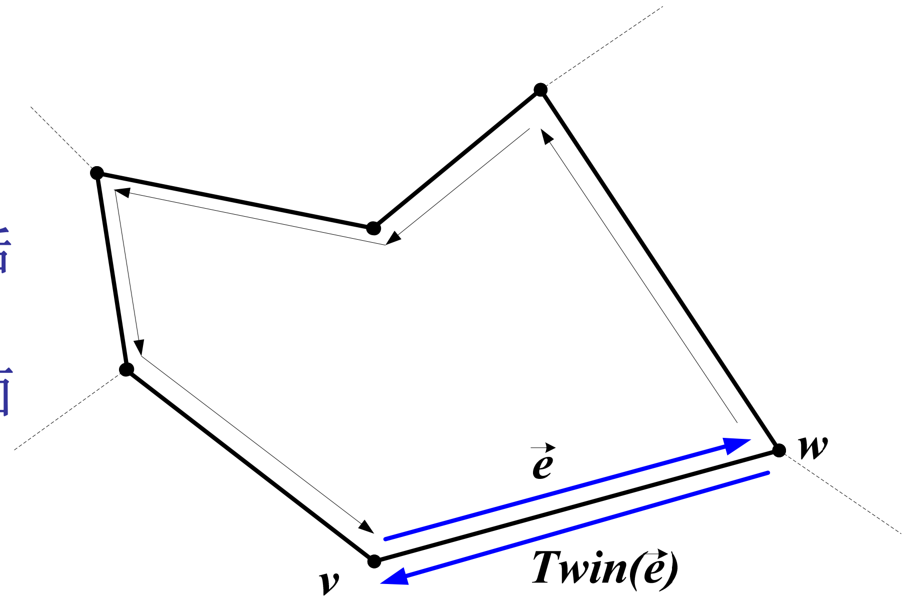


通过指针遍历每一张面的边界

2、双向链接边表

双向链接边表——半边

- 将每条边的两端分别视为一条半边（half-edge）
  - ✓ 任何一条半边都有唯一的一条后继半边、唯一的一条前驱半边
  - ✓ 每条半边就只隶属于唯一一张面的边界



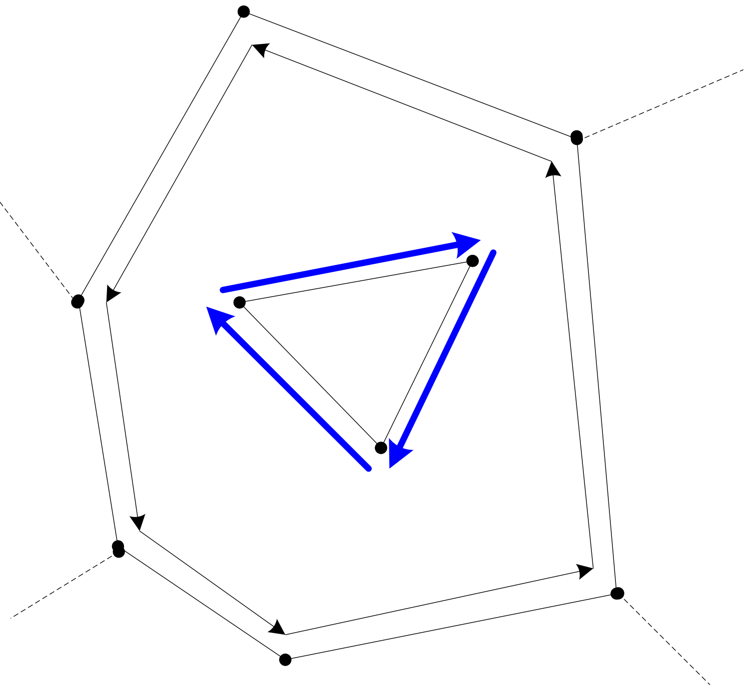
- 每一条边都被分成两条半边——它们互为孪生兄弟（twin）。在为每一条半边指定后继半边时，总是依照同一原则：后继半边的方向，应该能够沿逆时针方向遍历其对应的面

> 如果观察这沿着这一方向前行，每条半边所参与围成的那张面，总是位于其左侧

2、双向链接边表

双向链接边表——空洞

- 对空洞的边界来说，上面所说的性质不成立
- 如果能够在定义半边方向时，使得与之关联的面总在同一侧，就会更加方便。
  - ✓ 若是空洞，就按顺时针方向来遍历其边界。这样，无论是哪张面，对于构成其边界的那些半边来说，该面总是位于其左侧

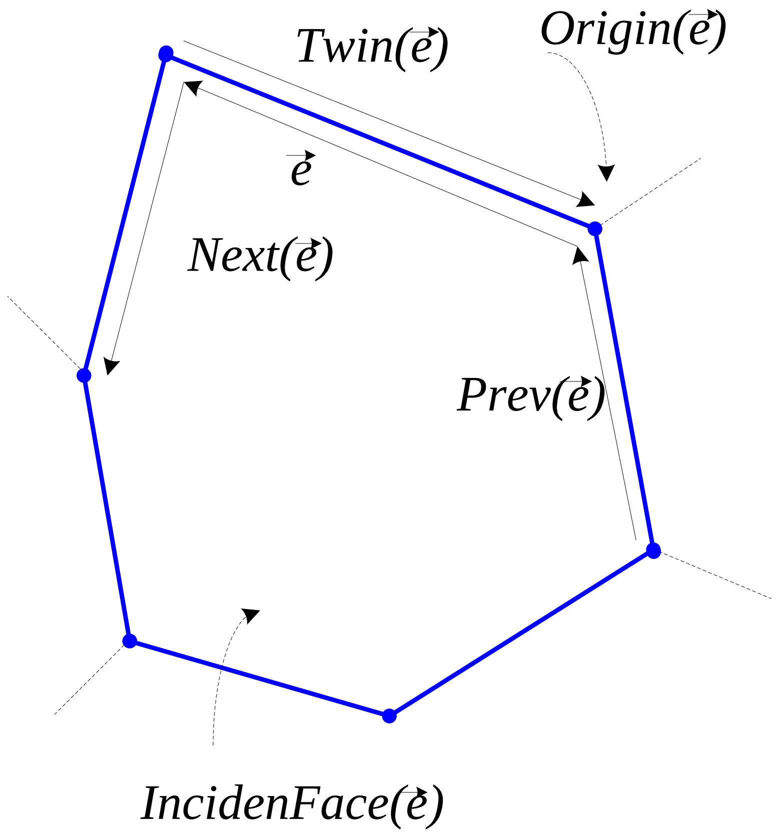


对于存在空洞的面，仅通过一个指针并不能保证访问到边界上所有的半边

2、双向链接边表

双向链接边表—数据结构

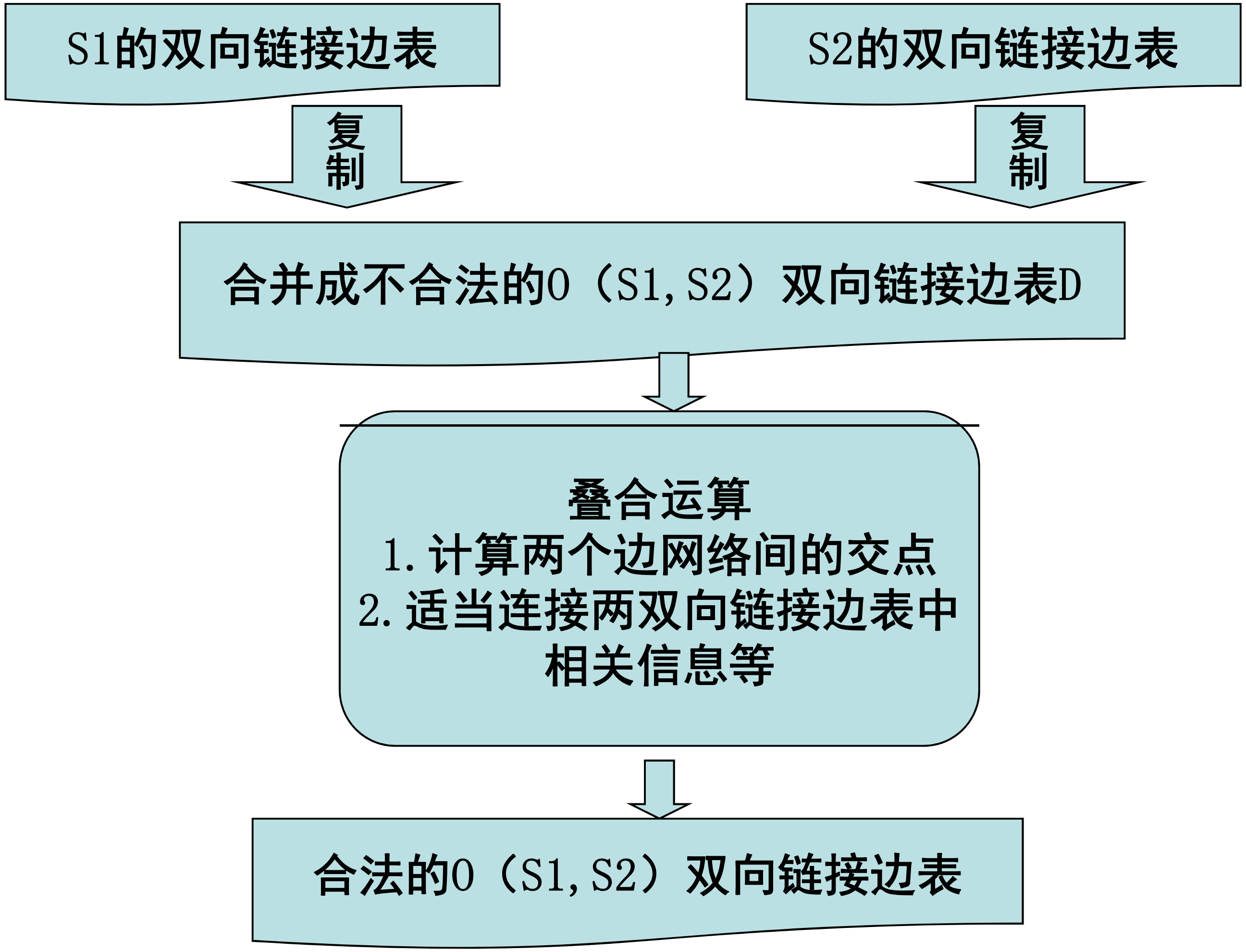
- 双向链接边表由三组记录构成
  - ✓ 一组对应于顶点
  - ✓ 一组对应于面
  - ✓ 一组对应于半边



DECL结构的各组成部分

3、计算子区域划分的叠合

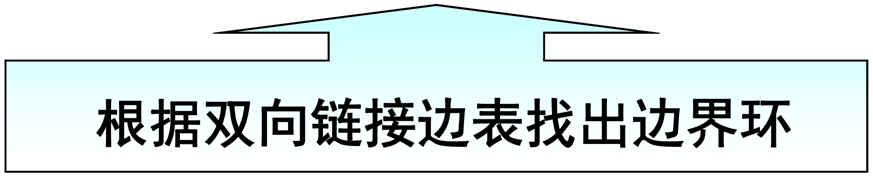
叠合的运算流程



3、计算子区域划分的叠合

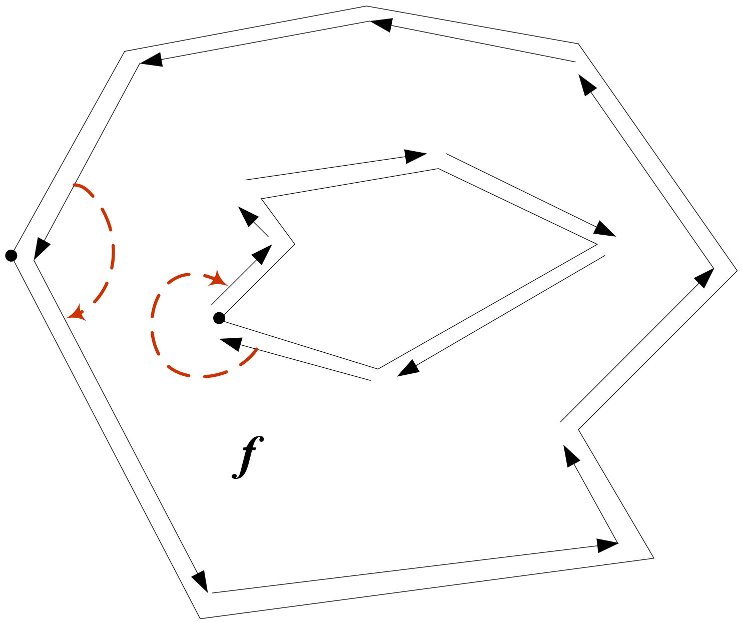
面记录设置的要求

面记录的数目 ＝ 所有外边界的数目+1（无界面）



如何确定边界环是面的外边界还是空洞？

- 外边界与空洞边界的判别：在环中找最左端最低的顶点，判断该点的关联的两条半边的夹角：小于180为外边界；大于180°为空洞边界。（由边界的方向得出）
- 在每个环上，只有最左端的顶点才具有这个性质



3、计算子区域划分的叠合

算法复杂度

【定理2.6】给定任意平面子区域划分S1和S2，其复杂度 分别为n1和n2，令n=n1+n2。则可以在O(nlogn+klogn)时间内计算出S1与S2的叠合，其中k为叠合结果的复杂度

第三讲：多边形三角剖分—画廊看守

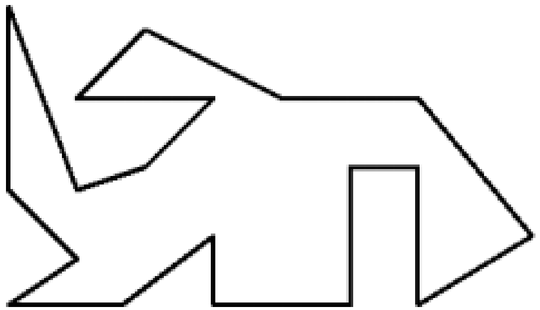
1、看守与三角剖分

2、多边形的单调块划分

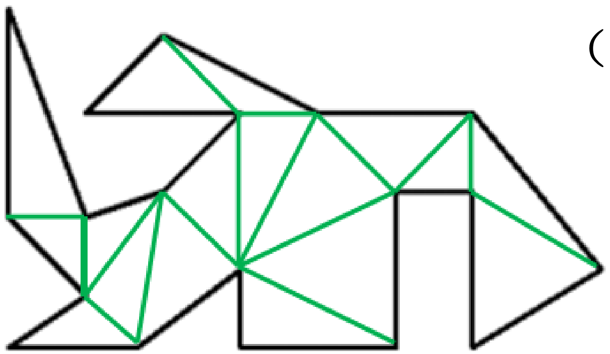
3、单调多边形的三角剖分

3.1 看守与三角剖分

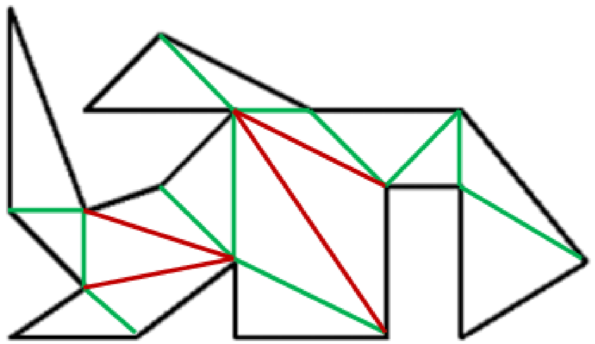
一个简单多边形的三角剖分不是唯一的



（a）



（b）



（c）

还存在很多种

3.1 看守与三角剖分

综上：

- 任何简单多边形都存在（至少）一个三角剖分；若其顶点数目为n，则它的每个三角剖分都恰好包含n-2 个三角形

结论：

- 包含n个顶点的任一简单多边形，为每个三角形配备一台摄像机，就可用n-2 台摄像机来看守

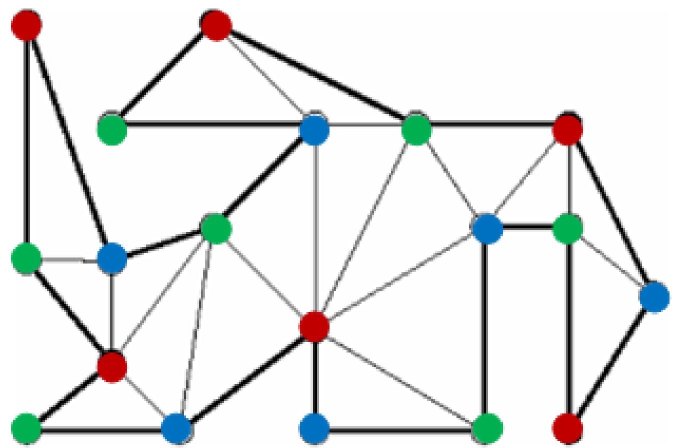
?

如何减少摄像机的数目？

3.1 看守与三角剖分

顶点染色

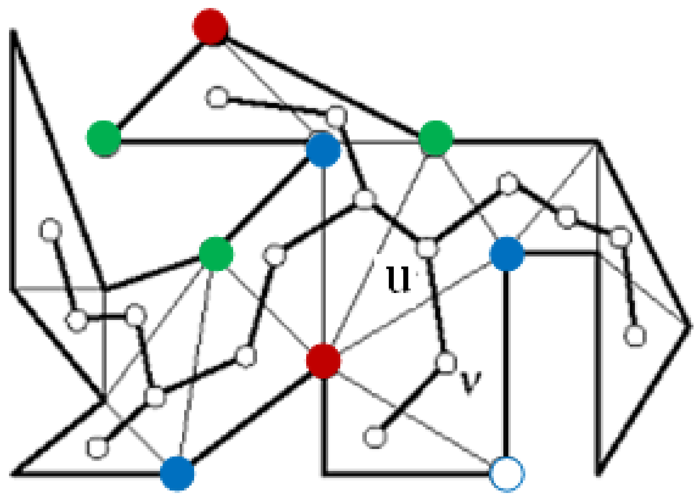
- 关于子集R：可以使用红、绿、蓝三元色，给P的所有顶点染色
- 染色方案须满足：由任何边或对角线联接的两顶点，所染的颜色不能相同，称作“对经过三角剖分后的多边形的3—染色（3-coloring）”



3.1 看守与三角剖分

3—染色方案是否总是存在？

- 将“TP的对偶图” 记之为G(TP)。对应于TP中的每个三角形，G(TP）都有一个顶点。顶点v、u处对应的三角形记作t(v) 、t(u) 。若t(v)与t(u)共用一条对角线，则在v 和u 之间就设置一条弧



- G(TP)中的各条弧与TP中的各条对角线对应。任一对角线都会将P一分为二
- 因此，G(TP)必然是一棵树。只要对该图进行一次（深度优先）遍历（traverse），就可以得到一种3-染色的方案

3.1 看守与三角剖分

- 以上就得出组合几何学（combinatorial geometry ）的一个经典结果——艺术画廊定理（art gallery theorem）：

【定理3.2】包含n 个顶点的任何简单多边形，只需（放置在适当位置的） n/3台摄像机就能保证：其中任何一点都可见于至少一台摄像机。有的时候，的确需要这样多台摄像机

3.2 多边形的单调块划分

“单调块（monotone piece）” 划分

P的单调块划分比凸块划分容易的多！

- 一条简单多边形称为“关于某条直线l单调”，如果对任何一条垂直于l的直线‘， <u>其与多边形的交都是连通的</u>

相交的结果

一条线段

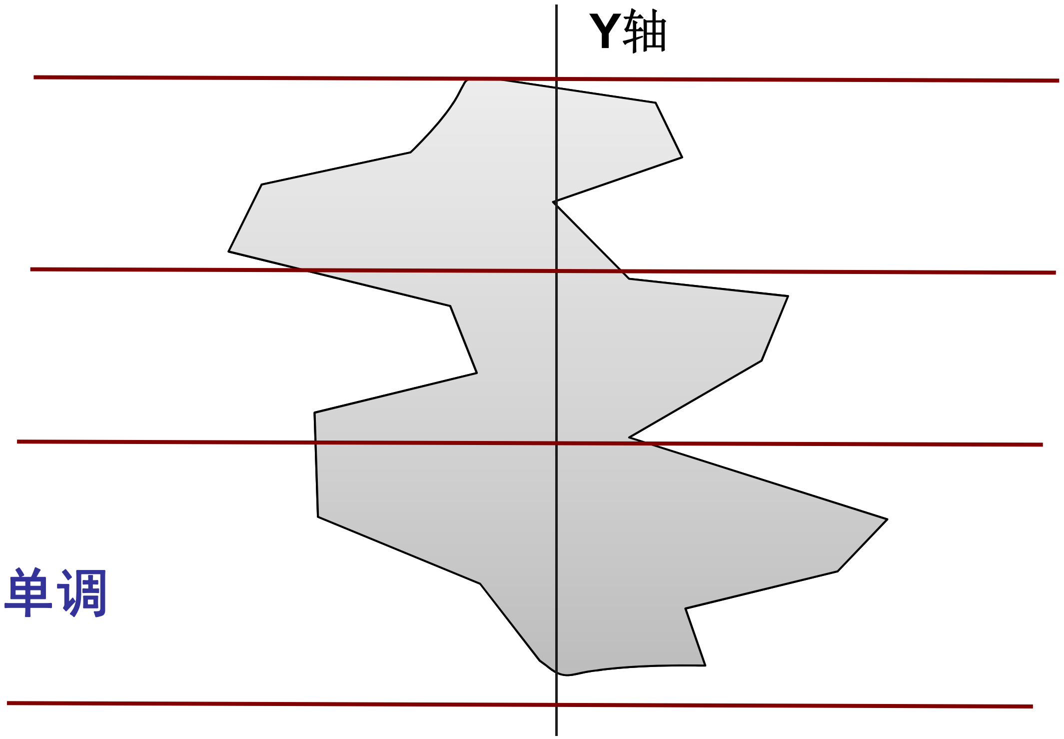
一个点

空集

3.2 多边形的单调块划分

"单调块（monotone piece）" 划分

- Y-单调的多边形的一个特征：在沿着多边形的左（右）边界，从最高顶点走向最低顶点的过程中，我们<u>始终都是朝下方（或者水平）运动</u>，而绝不会向上

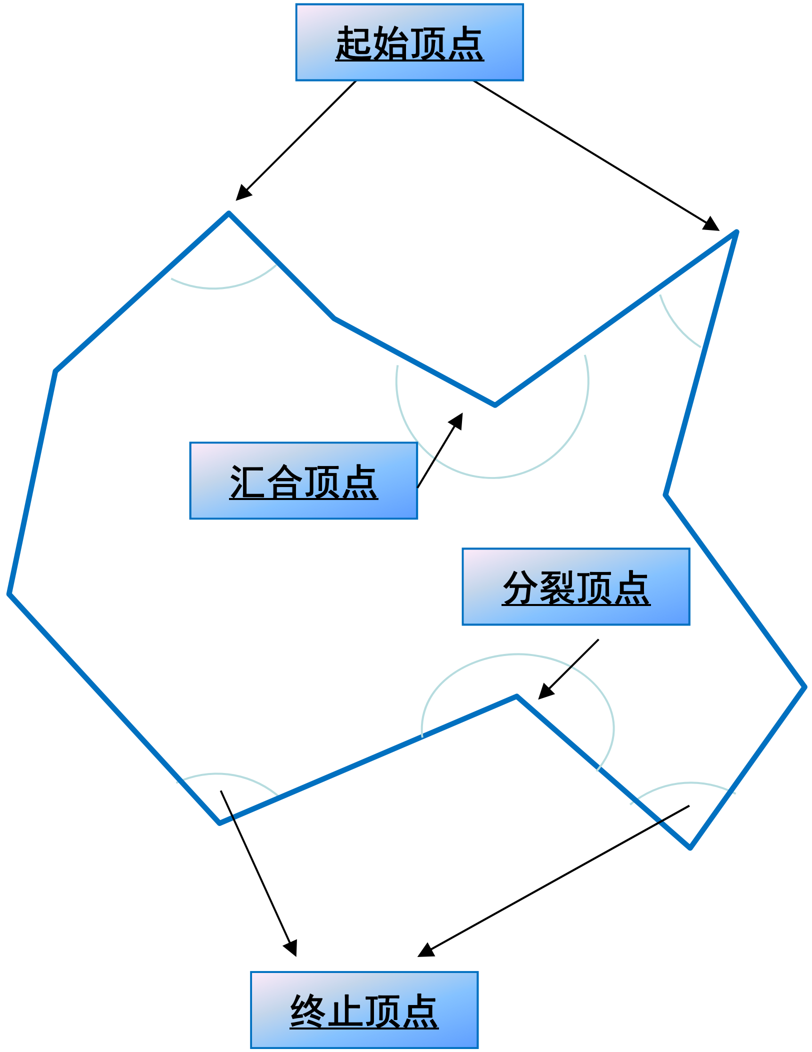


多边形关于Y轴单调

3.2 多边形的单调块划分

P的顶点分类

- **起始顶点（start vertex）：** 与它相邻的两个顶点高度都比它低，而且在v处的内角小于π
- **分裂顶点（split vertex）：** 与它相邻的两个顶点高度都比它低，而且在v处的内角大于π
- **终止顶点（end vertex）：** 与它相邻的两个顶点高度都比它高，而且在v处的内角小于π
- **汇合顶点（merge vertex）：** 与它相邻的两个顶点高度都比它高，而且在v处的内角大于π
- **其它为普通顶点**

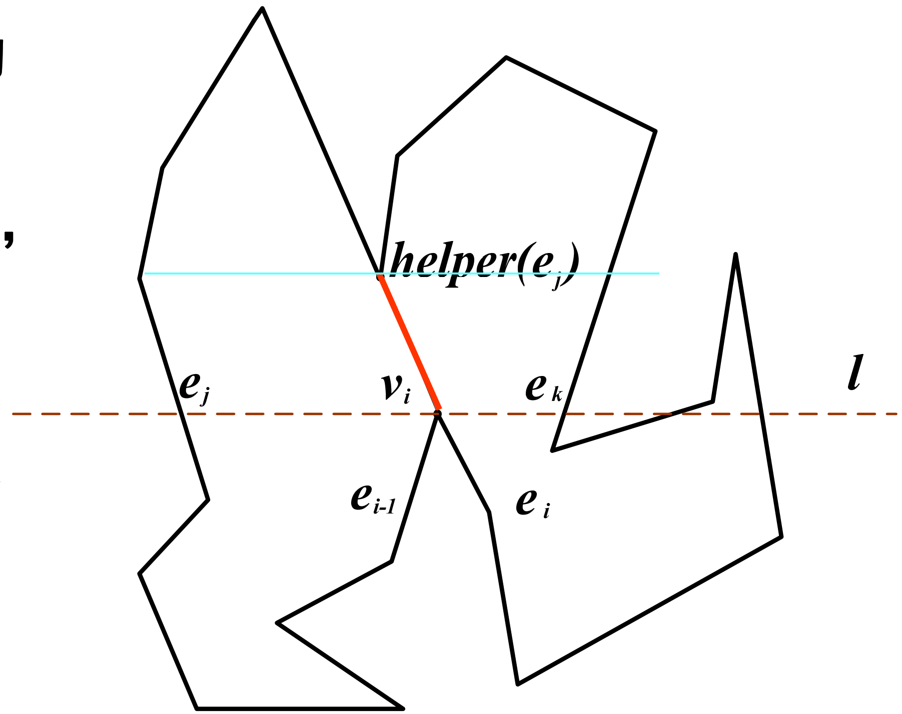


3.2 多边形的单调块划分

分裂顶点的处理

- 沿当前扫描线，令位于v_i的左侧、与之相邻的那条边为e_j；令位于v_i的右侧、与之相邻的那条边为e_k
- 考虑介于e_j和e_k、位于v_i上方的顶点，连接对角线
  - ✓ 若上方至少存在一个顶点，则将其中最低的那个与v_i连接起来
  - ✓ 若上方不存在顶点，则将与v_i与e_j或e_k的上端点连接起来

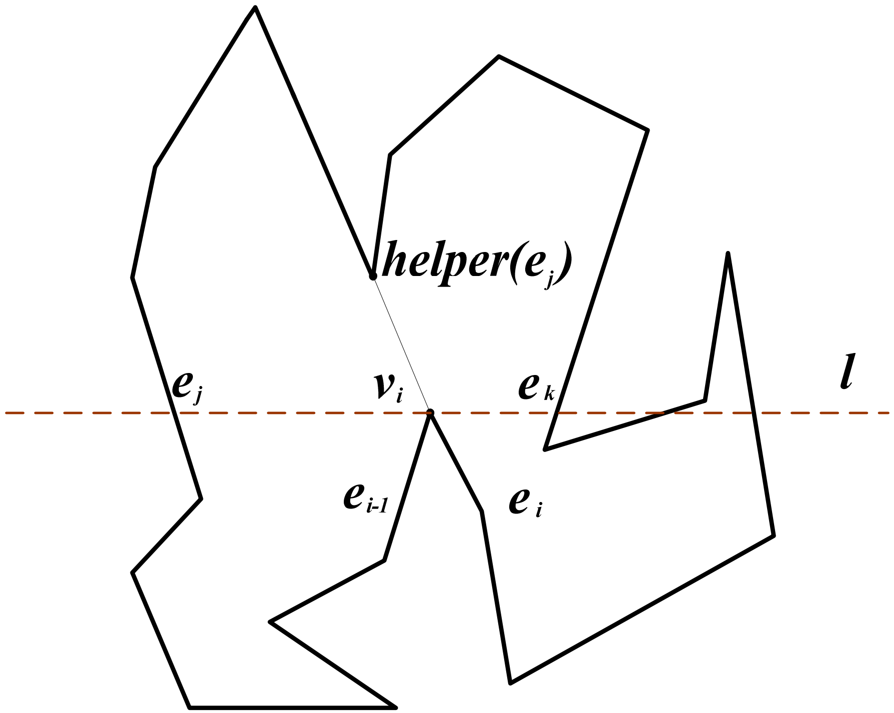
将与v_i连接的顶点称为helper(e_j)



确定事件点 → 找到临边助手 → 连接顶点与助手

3.2 多边形的单调块划分

helper(e_j)应该是"在位于扫描线上方、通过一条完全落在P内部的水平线段与e_j相联的那些顶点中，高度最低的那个顶点



3.2 多边形的单调块划分

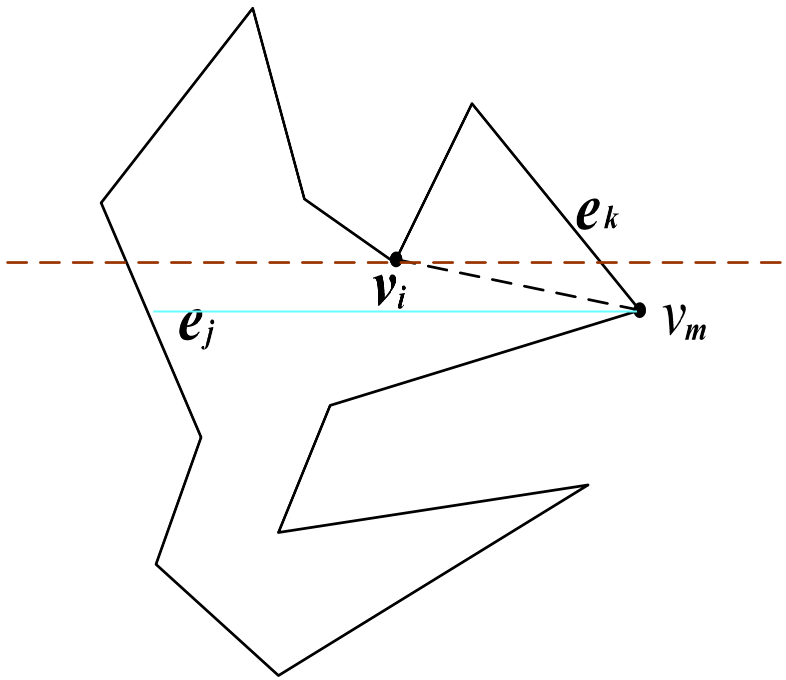
汇合顶点的处理

扫面线下面的顶点未访问到，如何向下引入对角线？

将v_i记做helper(e_j)

向下扫描，找到新的临边助手helper(e_j)，记做v_m

连接v_i与v_m



3.2 多边形的单调块划分

算法复杂度

一棵完全二叉树的存储结构

- 在确定顶点的优先度时，可以采用堆排序，每选出一个顶点的时间复杂度是**O(logn)**，n个点构成优先队列Z的时间复杂度是**O(nlogn)**
- 在扫描过程中，每次处理一个事件点，都只需要对Z操作一次，时间复杂度为**O(1)**；而对二分查找树的查找和更新的操作需时**O(logn)**；由于一共有n个点，所以总费时**O(nlogn)**
- 在Z中，每个顶点至多被存储一次；在状态结构T中，每条边至多存储一次，空间复杂度为**O(n)**

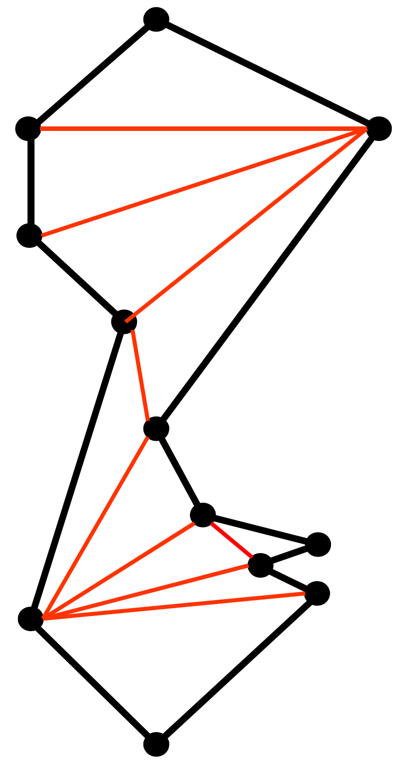
【定理3.6】使用O(n)的存储空间，可以在O(nlogn)时间内将包含n 个顶点的任何简单多边形分解为多个y单调的子块

3.3 单调多边形的三角剖分

三角剖分策略

- **算法策略：**从最高顶点开始，沿着P的（左、右）两条边界链走向最低点，在此过程中引入对角线完成三角剖分

- 按y坐标递减的次序处理各个顶点。若y坐标相等则从左到右处理



贪婪算法（greedy algorithm）

3.3 单调多边形的三角剖分

数据结构

- 初始化一个空栈S作为辅助数据结构，在算法求解过程中存放在P中已被发现、却任然可以生出对角线的那些顶点

顶点处理原则：

尽可能的在当前顶点与栈中的各个顶点之间引入对角线

第四讲：线性规划—铸模制造

1、铸造中的几何

2、半平面求交

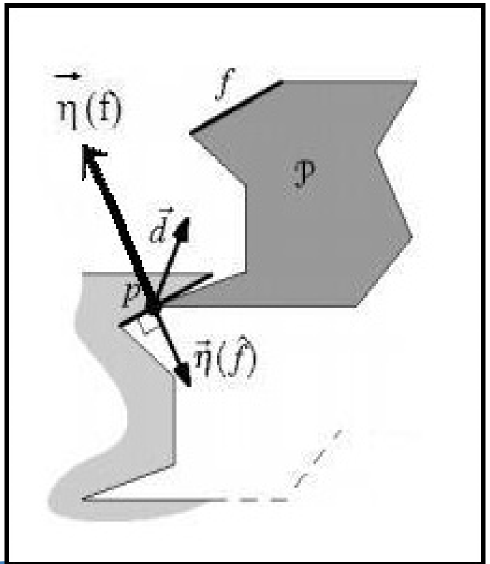
3、递增式线性规划

4、随机线性规划

4.1铸模中的几何

将P从铸模中取出的必要条件

抽取方向$\vec{d}$ 与f 的外法矢 $\vec{\eta}(f)$ 夹角至少为90°



引理

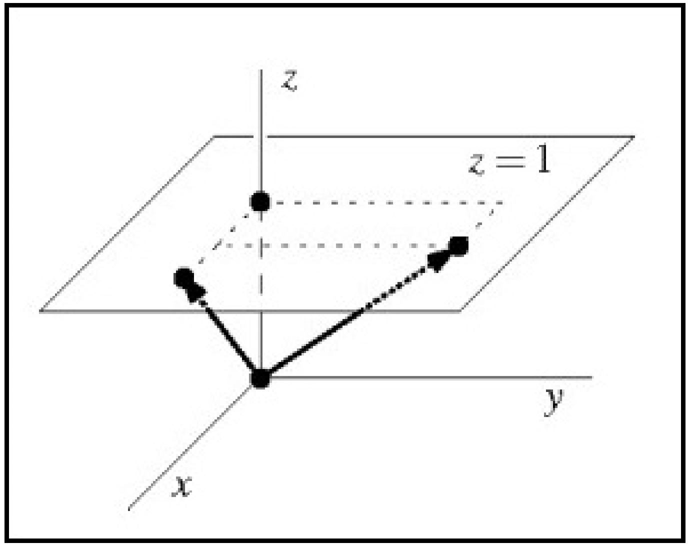
沿着某个方向 通过一次平移，多面体能够从其铸模中取出来，当且仅当相对于P的每一张普通面的外法矢与 $\vec{d}$ 所成的角度都至少为90°

目标：找出一个特定的方向，使之相对于P上任何一张普通平面的外法矢所成的夹角至少为90°

4.1铸模中的几何

Z=1平面的方向表示

- 三维空间中任何一个Z分量为正的方向（起始于原点），都可以表示为平面z=1上的一个点



点（x，y，1）表示与向量（x，y，1）对应的方向

- 反之，每一个Z分量为正的方向，都可以由该平面上的某个点唯一确定

4.1铸模中的几何

充要条件在方向平面上的转换

抽取方向 $\vec{d}$

引入平面z=1

$\vec{d}=\left(d_x,d_y,1\right)$

转换为点

夹角至少为90°

可铸造条件：

$\overrightarrow{\eta_x}d_x+\overrightarrow{\eta_y}d_y+\overrightarrow{\eta_z}\le 0$

（该式表示平面Z=1上的一张半平面）

普通面 f 外法矢

$\vec{\eta}=\left(\overrightarrow{\eta_x},\overrightarrow{\eta_y},\overrightarrow{\eta_z}\right)$

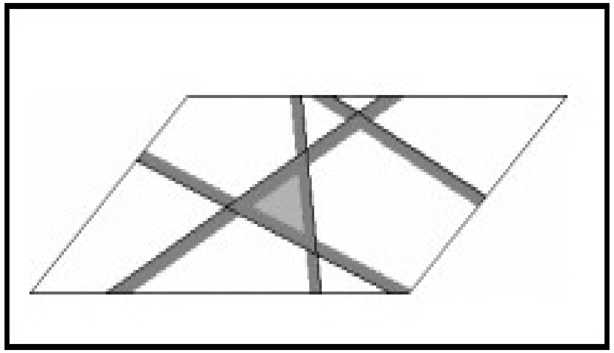
半位面求公共交集

Y

P可以抽取出来（每一交点对应一个可抽取方向）

N

不可铸造



4.2 半平面求交

半平面求交问题描述

- 任给双变量线性约束条件集$H=\{h_1,h_2,\ldots,h_n\}$，其中约束条件式如下：

$$a_i x + b_i y \le c_i$$

- 从几何角度分析，每个约束条件都可理解为R^2空间中的一张闭的半平面，其边界为直线$a_i x+b_i y=c_i$

↓

找出满足n个约束条件的所有点$(x,y) \in R^2$

一般性问题

4.2 半平面求交

一个分治式半平面求交算法

- 分治算法的基本思想是将一个规模为N的问题分解为K个规模较小的子问题，这些<u>子问题相互独立且与原问题性质相同</u>。求出子问题的解，就可得到原问题的解

IntersectHalfPlanes(H)

输入：
由平面上 n 张半平面组成的一个集合 H

```
if (card(H) == 1)
    then C ← H 中唯一的那张半平面 h
    else  将 H 分成两个子集 H1 和 H2，大小分别为⌈n/2⌉和⌊n/2⌋
        C1 ← INTERSECTHALFPLANES(H1)
        C2 ← INTERSECTHALFPLANES(H2)
        C ← IntersectConvexRegions(C1, C2)
```

输出：
凸多边形区域

$$C := \cap_{h \in H} h$$

计算两个凸多边形区域的交集

4.2 半平面求交

算法复杂度

- 已知：可以在O((n+k)logn)时间内计算出叠合部分（第二章）

C1不超过（n/2+1）张半平面

C2不超过（n/2+1）张半平面

交点数k≤n

O(nlogn)

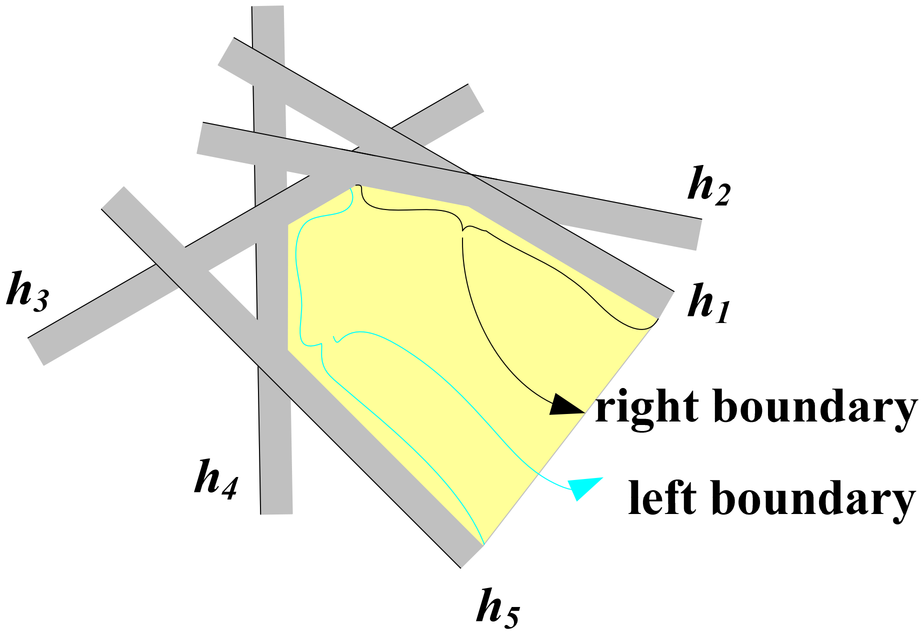
$$T(n)=\begin{cases} O(1), & \text{如果} n=1 \\ O(n\log n)+2T(n/2), & \text{如果} n>1 \end{cases}$$

解为$O(n\log^2 n)$，但算法中多边形区域都是凸的，因此存在进一步优化的可能

4.2 半平面求交

凸多边形区域的表示

- 将集合H构成的凸多边形区域C分为左右两部分，存为两个有序表，分别记作$L_{left}(C)$和$L_{right}(C)$
- 若水平边从上方围住C，则归为左边界
- 若水平边从下方围住C，则归为右边界



$$\begin{cases} L_{left}(C)=h_3,h_4,h_5 \\ L_{right}(C)=h_2,h_1 \end{cases}$$

4.2 半平面求交

算法复杂度

处理每一条边 O(1)	→	两个凸边形求交 O(n)

【定理4.3】平面上任意两个凸多边形的交集，都可以在O(n)时间内计算出来

$$T(n)=\begin{cases} O(1), & 如果 n=1 \\ O(n)+2T(n/2), & 如果 n>1 \end{cases}$$

【推论4.4】给定平面上一共n张半平面，可以使用线性的空间，在O(nlogn)时间内计算出其公共交集

4.3 递增式线性规划

线性规划问题的表述

- 约束条件：

$$\begin{aligned} a_{1,1}x_1 + \cdots + a_{1,d}x_d &\le b_1 \\ a_{2,1}x_1 + \cdots + a_{2,d}x_d &\le b_2 \\ &\vdots \\ a_{n,1}x_1 + \cdots + a_{n,d}x_d &\le b_n \end{aligned}$$

- 目标函数：

$$MAX(c_1x_1 + c_2x_2 + \cdots + c_dx_d)$$

✓ 问题输入：c_i, a_{ij}, b_i

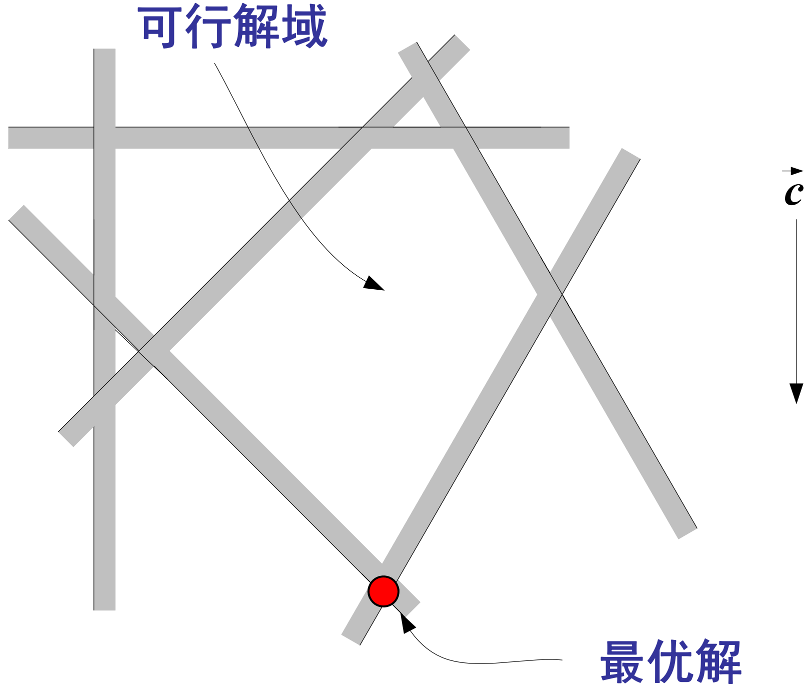
✓ 变量个数d称为线性规划为题的维数（**dimension of the linear program**）

4.3 递增式线性规划

线性规划问题的表述

- 将目标函数视作R^d空间中的某个方向，能够使函数极大化，即延
  $\vec{c}=(c_1,\cdots,c_d)$ 方向 找到极值点

将 $\vec{c}$ 方向上的目标函数记为 $f_{\vec{c}}$



4.3 递增式线性规划

递增式策略处理的线性规划问题

- 将半平面依次编号为h_1，h_2， …， h_n
- 前i个约束条件附加两个约束条件，构成集合H_i，其对应的可行解域为C_i

$$
\begin{cases}
H_i = \{m_1, m_2, h_1, h_2, \cdots, h_n\} \\
C_i = m_1 \cap m_2 \cap h_1 \cap h_2 \cap \cdots \cap h_n
\end{cases}
$$

若对i有 $C_i = \varnothing$ → 则对任何j≥i，都有$C_j = \varnothing$ → 算法终止

4.3 递增式线性规划

递增式策略处理的线性规划问题

? 在引入下一张半平面h_i时，最优解顶点将会如何变化？

- 设$1 \le i \le n$，解域C_i中的最优解为v_i，则有：

① 若$v_{i-1} \in h_i$，则 $v_i = v_{i-1}$

即若最优点v_{i-1}位于新引入的半平面中h_i中，则最优点位置不变

② 若 $v_{i-1} \notin h_i$ ，则要么 $C_i = \varnothing$，要么 $v_i = l_i$（其中l_i 为 h_i 的边界线）

即若最优点v_{i-1}不在新引入的半平面中h_i中，则新的最优点要么不存在，要么位于h_i的边界线上

4.4 随机线性规划

随机次序的引入

- 在问题解决之初，没有任何“好”的方法可以预先确定一个“好”的次序
- 引入半平面的次序，不会改变最终的最优解顶点，但会影响算法运行时间

运气

所以我们应该寻找某种好的次序，以保证运行的时间性能足够好

- 我们可以采取一种非常简单的方法，直接采用H的一种随机次序，这就是本节的研究内容——随机线性规划

4.4 随机线性规划

随机算法复杂度

- n 个对象可能的排列共有n!种，算法也就相应地有n!种执行的方式，而每一种方式也各有其不同的运行时间

期望运行时间 → 全部共n!种可能排列次序的平均运行时间

【引理4.8】任一包含n 个约束条件的二维线性规划问题，都可以在O(n)的期望运行时间内得到解答；而且，所需要的空间在最坏情况下也不会超过线性规模

第五讲：正交区域查找—数据库查询

1、一维区域查找

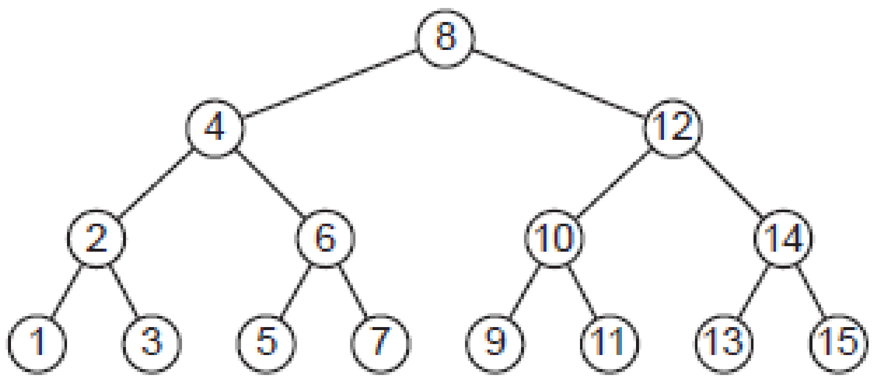
2、*kd*-树

3、区域树

5.1 一维区域查找

二分查找树

- 树中的叶子分别存储P中的各点
- 树中的内部节点存储划分的数值，用来引导查找
- 若内部节点v的划分值为x_v，假设：
  - ✓ v的左子树中存储了坐标不超过x_v的所有点
  - ✓ v的右子树存储了坐标严格大于x_v的所有点



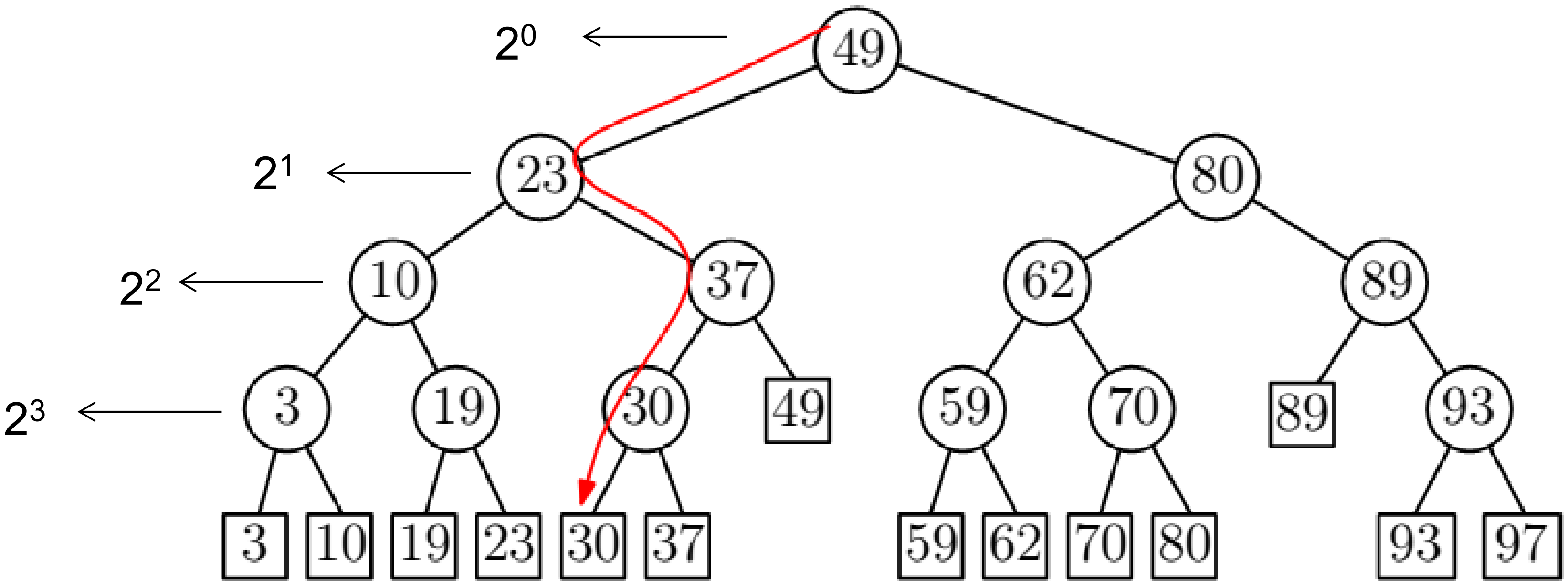
5.1 一维区域查找

二分查找树的形成过程

Suppose $2^k = n$

$\Rightarrow k = \log_2(n)$

$\Rightarrow k = O(\log n)$

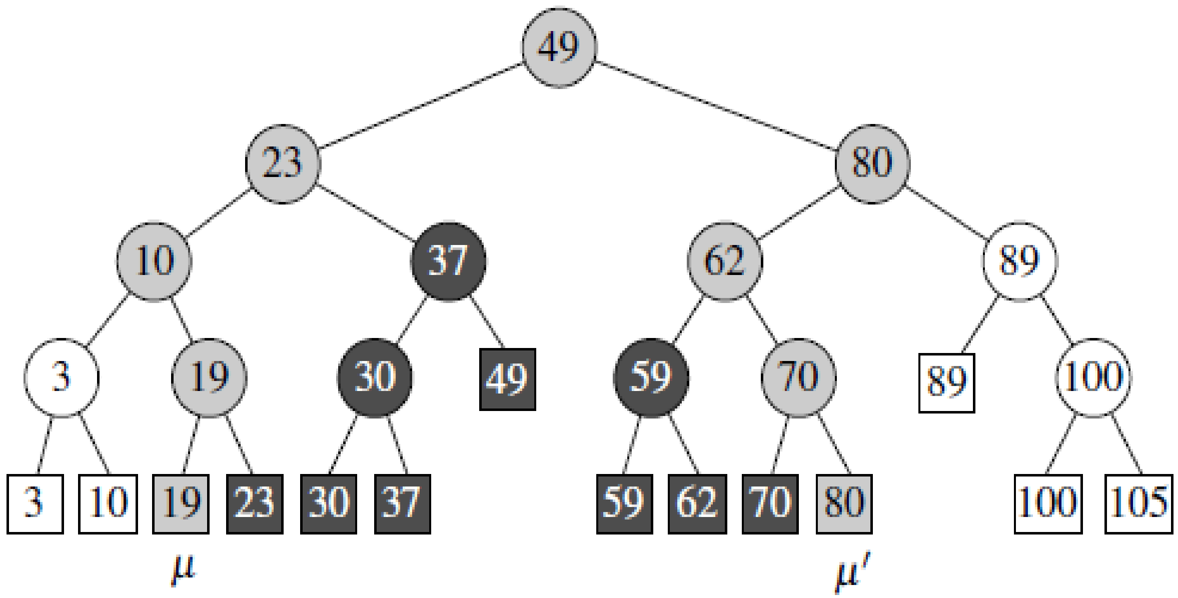


- **The height of a tree is the number of nodes on its longest branch (a path from the root to a leaf).**

5.1 一维区域查找

一维区域查找策略

- 在平衡二叉树中分别查找x和x'，设两次查找分别终止于叶子μ和μ'。于是，位于区间$[x : x']$之内的点，就对应于介于μ和μ'之间的那些叶子，可能还要加上μ或μ'本身

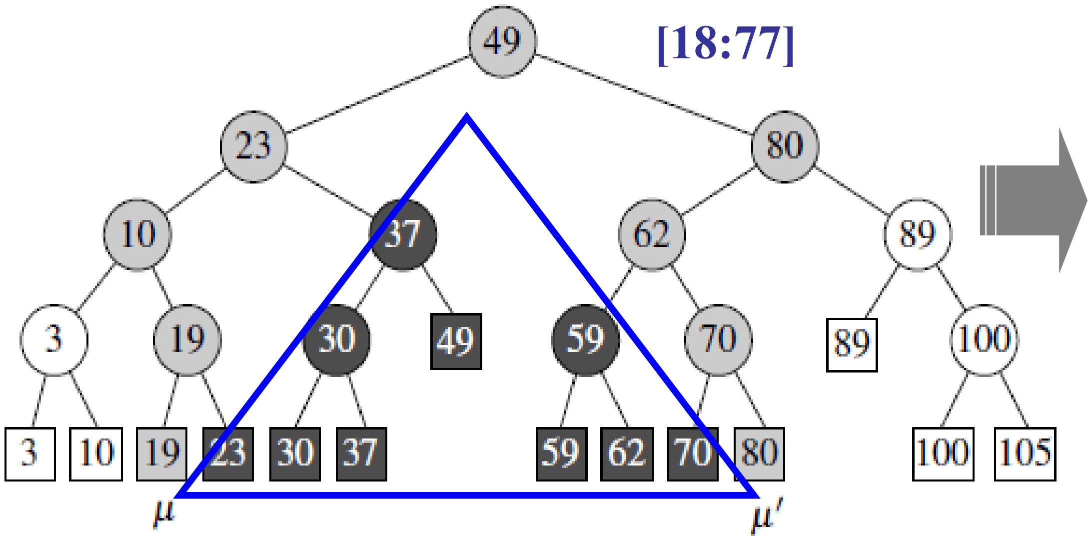


5.1 一维区域查找

一维区域查找策略

如何找出μ或μ‘之间的叶子？

- 观察1：对应于μ或μ‘两条查找路径之间存在一些子树，需要找出的叶子都来自于某棵子树（深灰色为查找子树，查找路径上顶点为深灰色）
- 观察2：查找过程中选出的每一个子树，都以两条查找路径之间的某个节点v为根

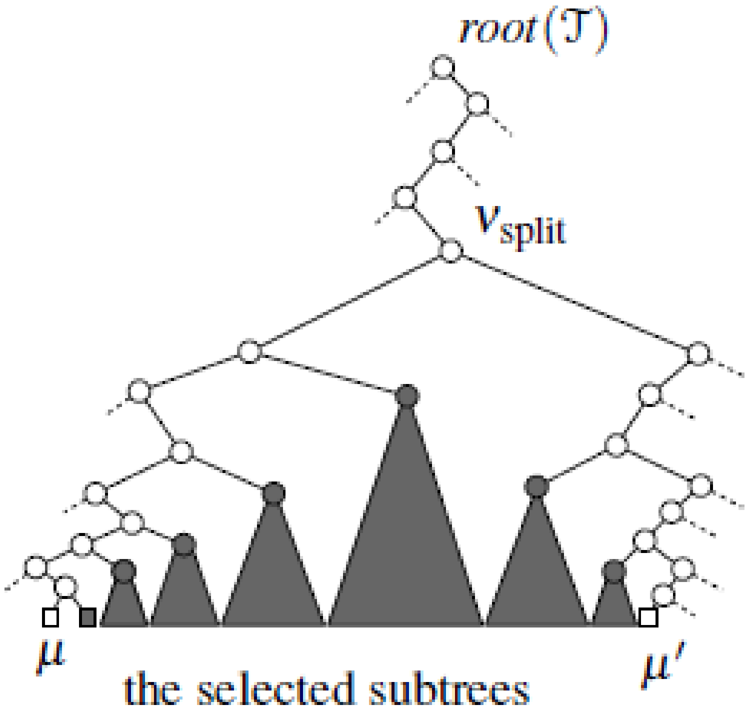


首先需要确定与x和x‘对应对应的两条查找路径开始分叉的位置，记为v_{split}

查找的子树

5.1 一维区域查找

点枚举思路



- 从 v_{split} 的确定出发，分别x和x'对应的查找路径前进
  - ✓ 每向左前进一步，就枚举出该处右子树中的所有叶子
  - ✓ 每向右前进一步，就枚举出该处左子树中的所有叶子
- 最后，检查一下两条查找路径终点处的叶子，它们对应的点有可能位于区间 $[x{:}x']$ 之内，也可能位于其外

REPORTSUBTREE子程序：给定以某个节点为根的一棵子树，该子程序通过遍历，报告出所有叶子对应的点

5.1 一维区域查找

算法复杂度

- 采用平衡二分查找树，占用O(n)空间，在O(nlogn)时间内构造出来
- 查询时间O(k+logn)
  - ✓ ReportSubtree调用所需时间线性正比于报告出来的点数，所需时间为O(k)
  - ✓ *x*和*x*'对应的查找路径长度为O(logn)

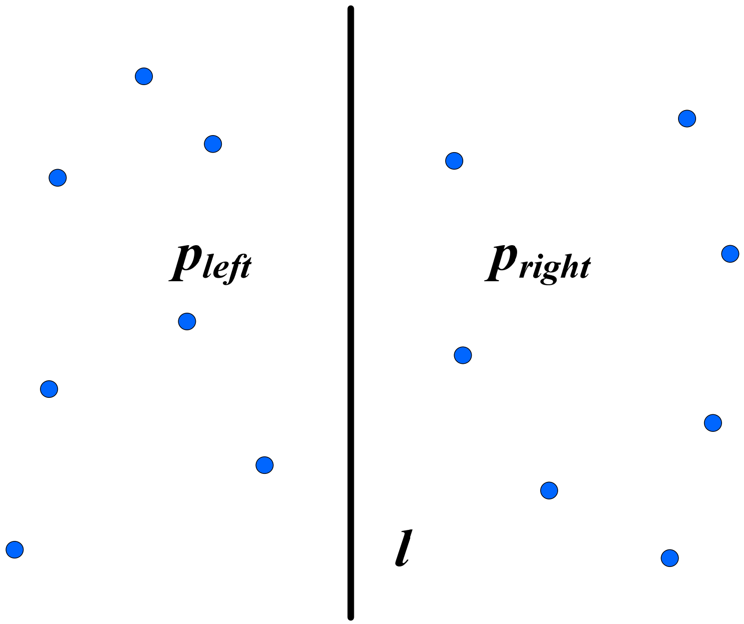
【定理5.2】给定由一维空间中任意n个点构成的集合P。可以使用*O*(n)空间，在*O*(nlogn)时间内构造一棵平衡二分查找树以存储P。这样，可以在*O*(k+logn)时间内查找出任何区间内的所有点（其中，k是实际被查找出来的点数）

5.2 *kd*-树

二分查找树的定义

- 在二维情况下，每个点都拥有两个基本的数值：*x*坐标和*y*坐标。因此，首先沿*x*坐标方向做一次划分，然后沿*y*方向做一次，接着再次沿*x*方向划分，如此下去（递归思想）

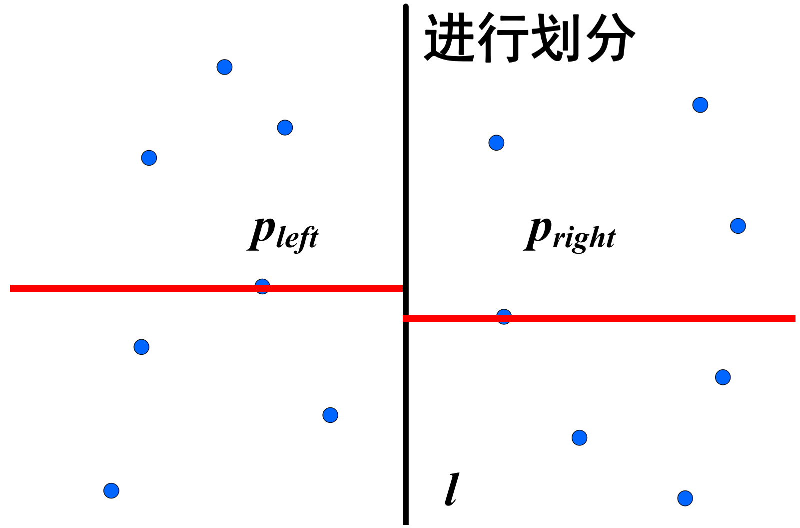
- 这一过程可以定义如下：
  - ✓ 首先，在根节点处，通过一条垂线*l*将集合P划分为大小接近的（左、右）两个子集
  - ✓ 该分割线存储于根节点处。位于分割线左侧的那个子集记作P_{left}，存储于左子树中；位于分割线右侧的那个子集记作P_{right}，存储于右子树中



5.2 *kd*-树

二分查找树的定义

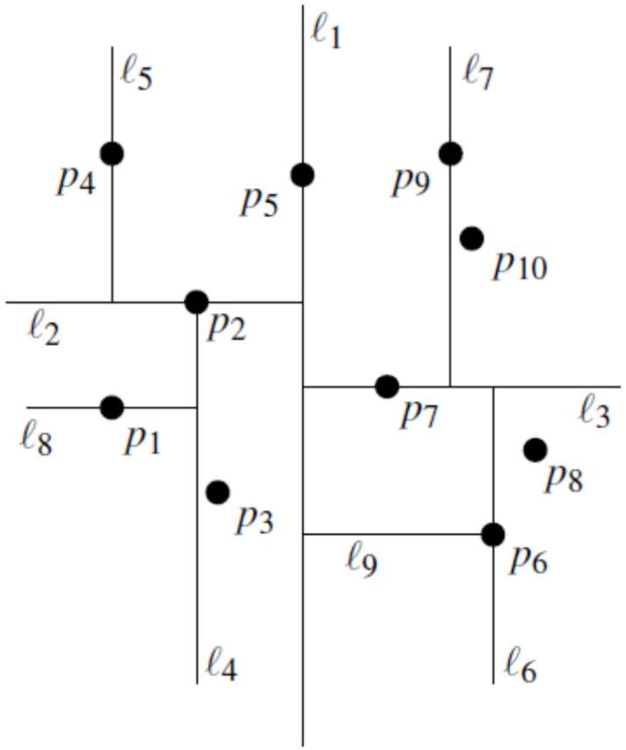
- 在根节点的左孩子处，继续通过一条水平线，将P_{left}划分为（上、下）两个子集：
  - ✓ 位于该分割线以下或者落于其上的那些点，被存储于该左孩子的左子树中；而位于分割线以上的那些点，则被存储于右子树中。该左孩子将记录下水平分割线的位置
- 类似地，P_{right}也将被某条水平线划分成两个子集，它们分别被存储于右孩子的左、右子树中。对于根节点的每个孙子，再次通过一条垂线进行划分

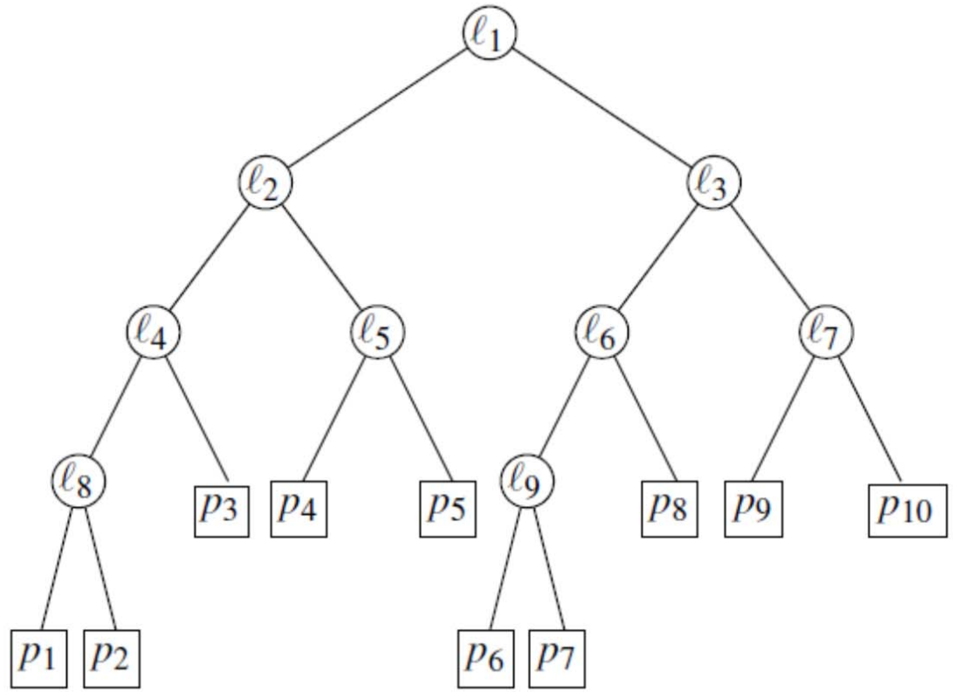


5.2 *kd*-树

二分查找树的定义

- 一般地，对于深度为偶数的节点，使用垂线进行划分；对于深度为奇数的节点，将使用水平线进行划分





5.2 *kd*-树

kd-树（k-dimensional tree）

- kd-树(J.L.Bentley,1975)是k(k>1)维的二分查找树，是二分查找树在多维空间的扩展，主要用于<u>索引多属性的数据或多维点数据</u>

- kd-树或者是一棵空树，或者是一棵具有下列性质的二叉树：
  - ✓ 若它的左子树不空，则左子树上所有结点的第d维的值均小于它根结点的第d维的值（其中d为根结点的分辨器值）
  - ✓ 若它的右子树不空，则右子树的所有结点的第d维的值均大于它根结点的第d维的值（其中d为根结点的分辨器值）
  - ✓ 左右子树也分别为kd-树

5.2 *kd*-树

kd-树的构造算法

- 子程序**BuildKDTree (P, depth)**有两个输入参数：

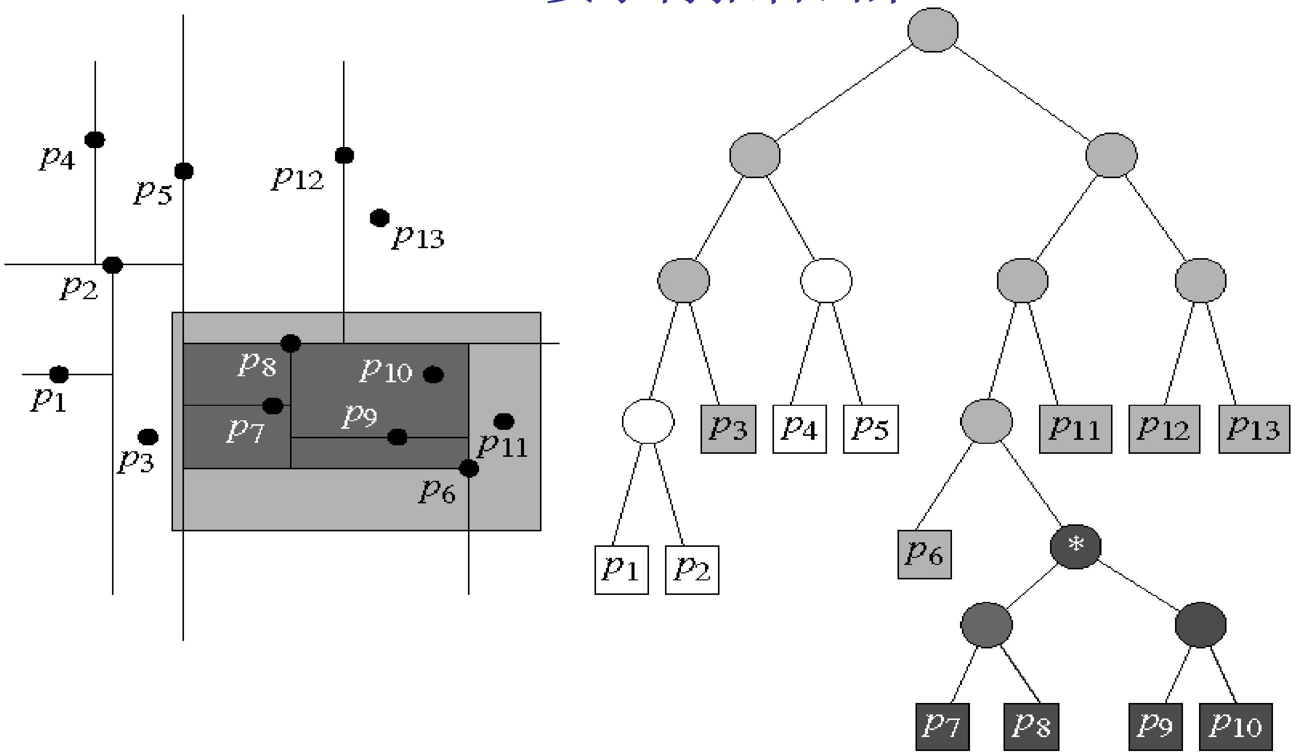
✓ 第一个参数就是我们需要为之建立kd-树的点集；初次调用时，它就是集合P 本身

✓ 第二个参数为递归深度，即在递归调用构造子程序时，对应子树根节点的深度。初次调用时，后一参数设置为零。深度很重要，因为深度决定了我们究竟是用垂直线还是水平线进行划分

5.2 *kd*-树

kd-树的查找策略

- 遍历kd-树，只访问对应子区域与待查找矩形相交的节点
  - ✓ 若某个区域完全落在查找矩形中，就将其中所用的点报告出来
  - ✓ 其它与查找矩形相交的子区域，对其内部的点进行测试，若符合要求再报告出来



5.2 *kd*-树

算法总结：

- **定理5.5：**给定由平面上任意n个点构成的集合P，其对应的kd-树将占用O(n)空间，并且可以在O(nlogn)时间内构造出来，使用这棵kd-树，每次矩形查找所需的时间将不会超过 $O(\sqrt{n}+k)$ ，其中k为实际被报告出来的点数

n	$\log n$	$\sqrt{n}$
4	2	2
16	4	4
64	6	8
256	8	16
1024	10	32
4096	12	64
1.000.000	20	1000

5.2 *kd*-树

小结与分析

- kd-树是为索引空间点或多属性数据而提出的
- kd-树是二叉查找树在多维空间的扩展。对于精确的点匹配查找，它继承了二叉查找树的优点

5.3 区域树

区域树（Range Tree）

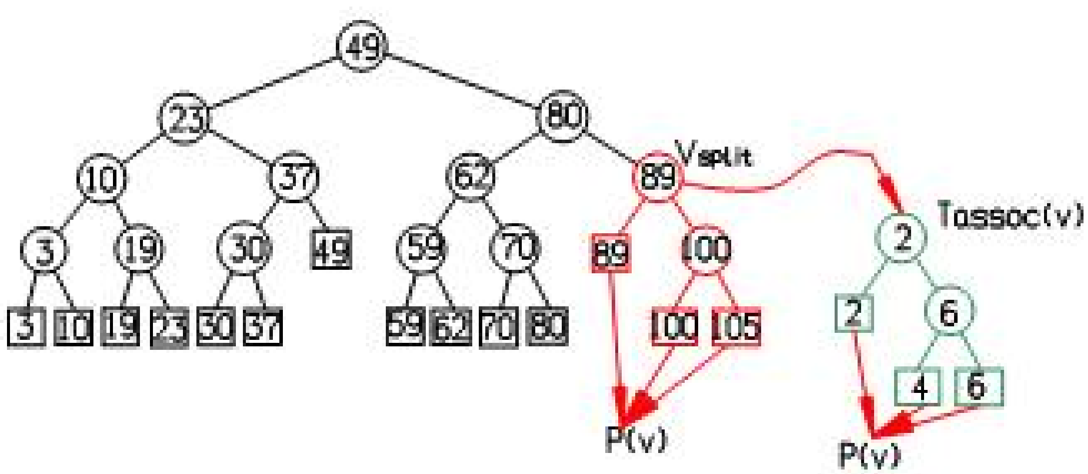
- 主树（main tree）是一棵平衡二分查找树 T，按照P 中各点的x坐标，将它们组织起来。
- 对于 T中的每个内部节点或者叶子v，再将正则子集P(v)中的各点按照y坐标，存储为一棵平衡二分查找树 $T_{assoc(v)}$ 。节点v 拥有一个指针，指向 $T_{assoc(v)}$的根。我们称 $T_{assoc(v)}$为v 的联合结构（associated structure）

区域树结构示意图

5.3 区域树

区域树（Range Tree）

- 如果某一数据结构中配有指向联合结构的指针，往往被称为多层次数据结构（multi-level data structure）。这时，主树 T 被称为第一层的树（first-level tree），而每一联合结构都被称为第二层的树（second-level tree）



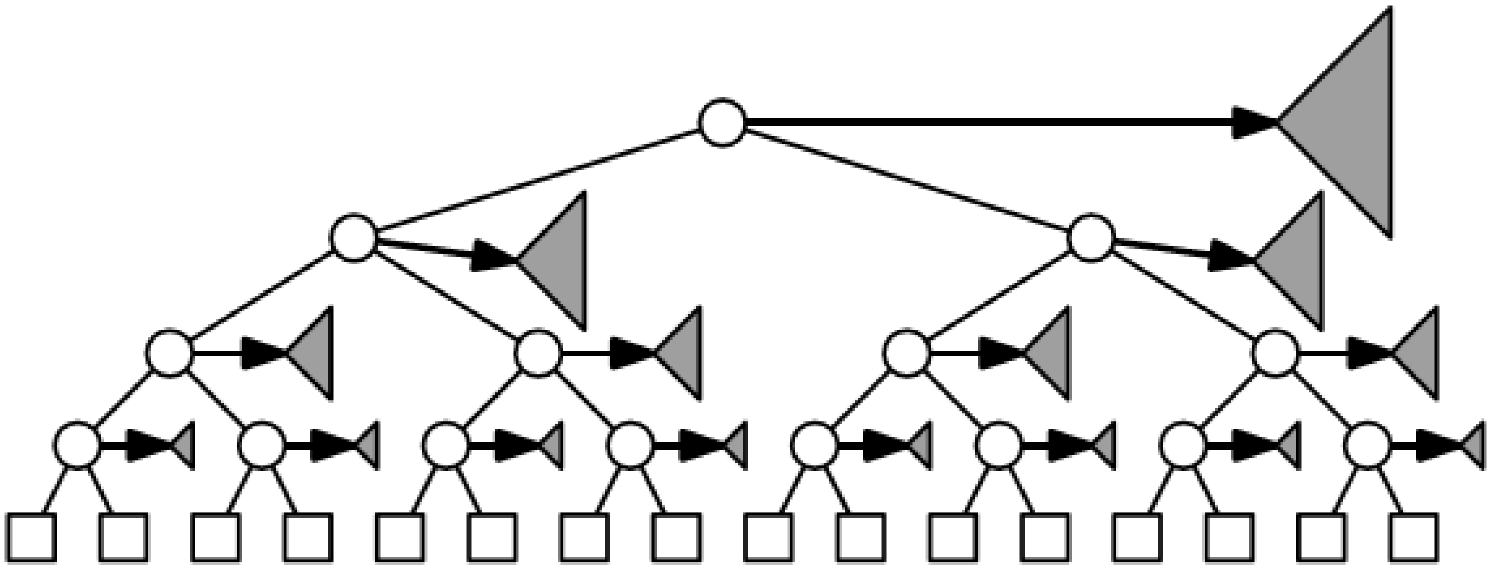
区域树示例图

点	x	y
p_1	89	2
p_2	100	6
p_3	105	4

5.3 区域树

区域树的空间复杂度

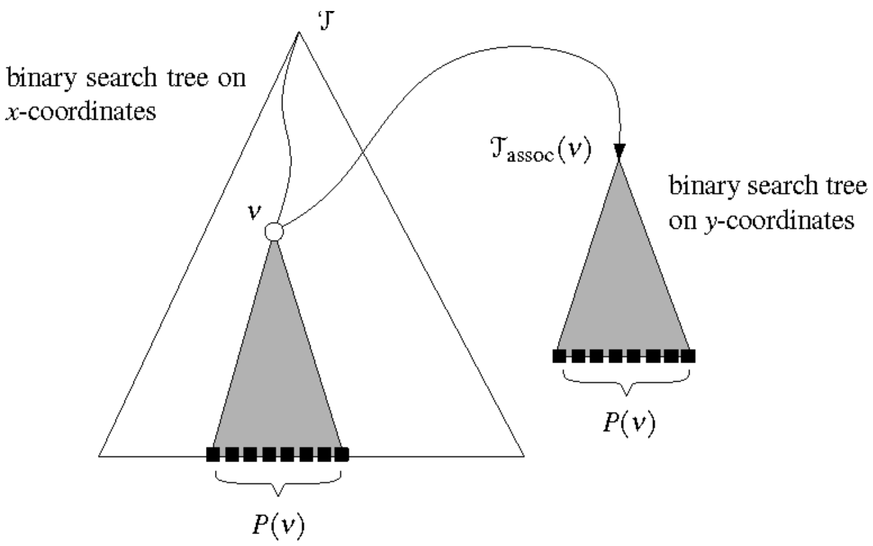
- 既然任何一棵一维区域树只占用线性规模的存储空间，故在T 的每一深度层次上，所有节点对应的联合结构总共只占用O(n)的存储空间。鉴于T的深度为O(logn)，故其所需的总存储空间不会超过O(nlogn)



5.3 区域树

总结

【定理5.8】给定由平面上任意n 个点构成的集合P。对应于P的一棵区域树占用$O(n\log n)$的存储空间，并且可以在O(nlogn)时间内构造出来。对这棵区域树进行查找，可以在$O(\log^2 n+k)$时间内，从P 中报告出落在任一矩形待查询区域之内的所有点，其中k 为实际被报告出来的点数



n	$\log n$	$\log^2 n$	$\sqrt{n}$
16	4	16	4
64	6	36	8
256	8	64	16
1024	10	100	32
4096	12	144	64
16384	14	196	128
65536	16	256	256
1M	20	400	1K
16M	24	576	4K

5.4 高维区域树

高维区域树的构造

- 考虑d维空间上任一点集P。按照其中各点的第一维坐标，构造出一个平衡二分树。第一层次这棵树也就是主树，对于其中的任一节点v，在以v为根节点的子树中，所有的叶子各自对应的点，合起来构成了与v对应的正则子集P(v)。
- 对每个节点v，为其构造一个联合结构 $T_{assoc}(v)$ 。第二层的每棵树 $T_{assoc}(v)$ ，都是对应P(v)中某一点的一棵（d-1）维区域树。——此时各点的坐标都限制在后（d-1）维上。

第六讲：点定位—找到自己的位置

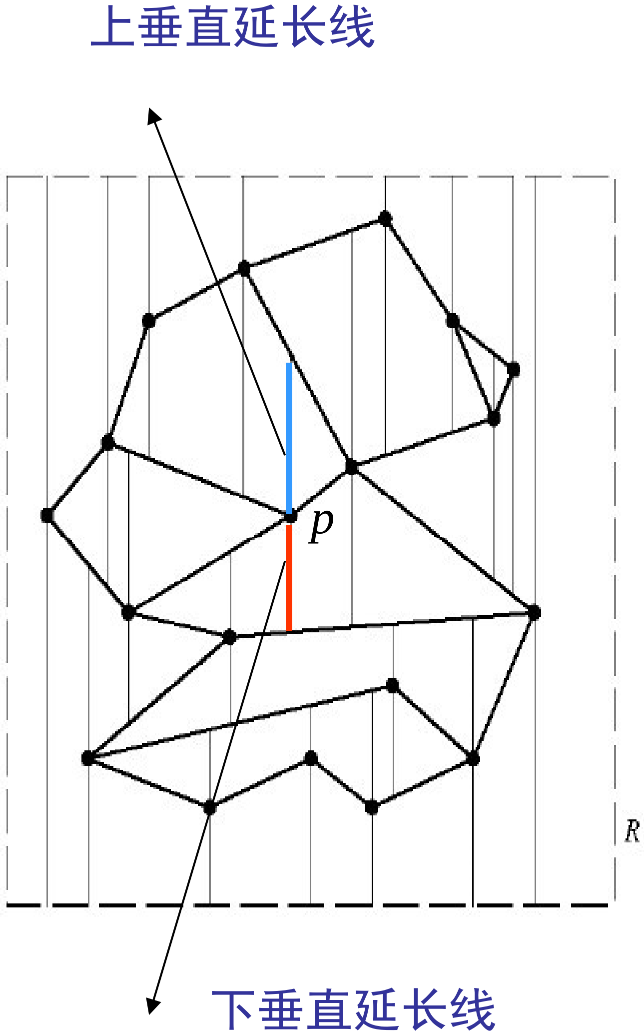
1、点定位与梯形图

2、随机增量式算法

3、退化情况的处理

6.1 点定位及梯形图

梯形分解



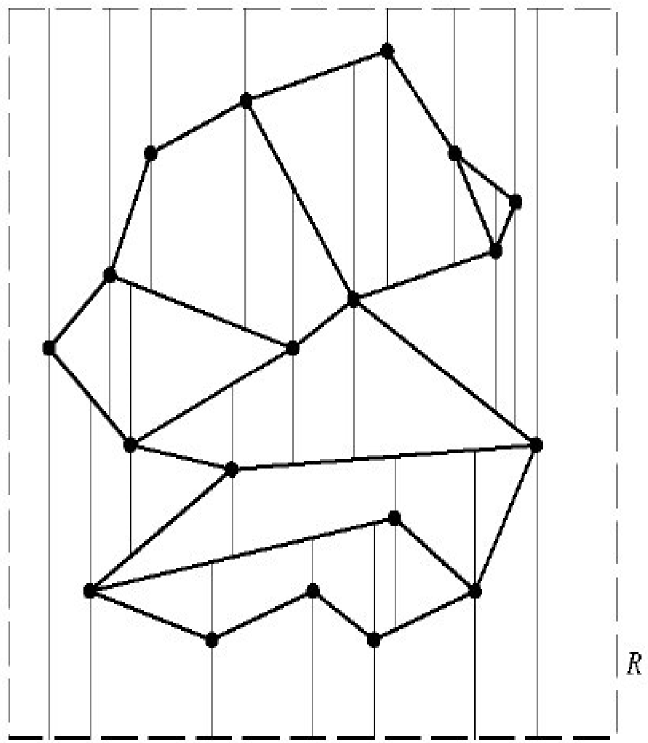
- **S的梯形图T(S)，也称作S的垂直分解或者梯形分解：**
  - ✓ 经过**S**中每条线段的左、右端点，向上方和下方各发出一条垂直射线；在碰到**S**中的另一条线段或**R**的边界后，射线终止

T(S)也是一个S的子区域划分

6.1 点定位及梯形图

为什么在梯形进行点定位更加容易？

- 【引理6.2】由任意n条处于一般性位置的线段组成的集合S，其梯形图T(S)至多含有6n+4个顶点，至多含有3n+1个梯形



欧拉公式：N-M+R=2

其中，N为顶点数，M为边数，R为面数

6.1 点定位及梯形图

梯形图的存储

- top(Δ)
- bottom(Δ)
- leftp(Δ)
- rightp(Δ)

→ 确定唯一的梯形Δ

- 左上方邻居
- 左下方邻居
- 右上方邻居
- 右下方邻居

→ 利用各梯形之间的相邻关系，将整个子区域划分联接成为一个整体

是否可以采用双向链接边表?

6.2 随机增量式算法

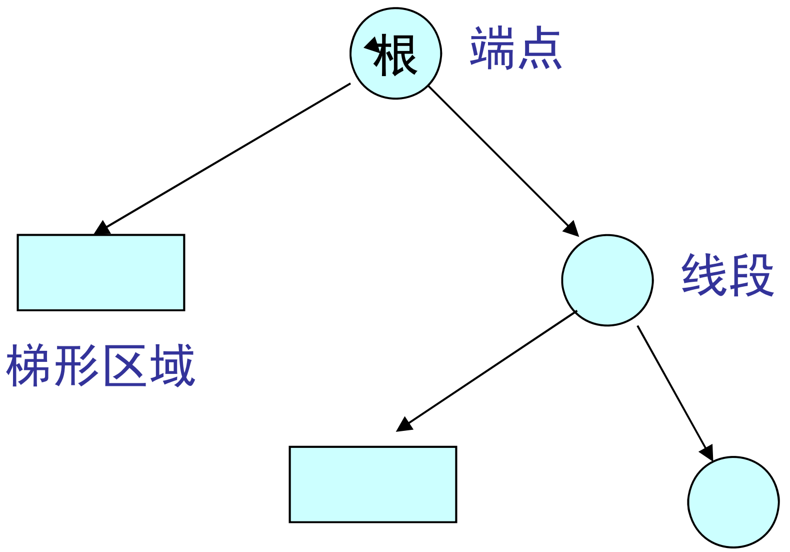
查找结构

- **查找结构（search structure）：一种支持点定位查询的数据结构，是一个有向无环图**
  - ✓ 其中有唯一的根节点。同时，对应于S 的梯形图中的每个梯形，有且仅有一匹叶子
  - ✓ 每个内部节点的出度都是2
  - ✓ 所有内部节点分为两类：x节点和y节点。每个x节点都被标记为S中某条线段的一个端点；而每个y节点都被标记为某条线段

6.2 随机增量式算法

查找策略

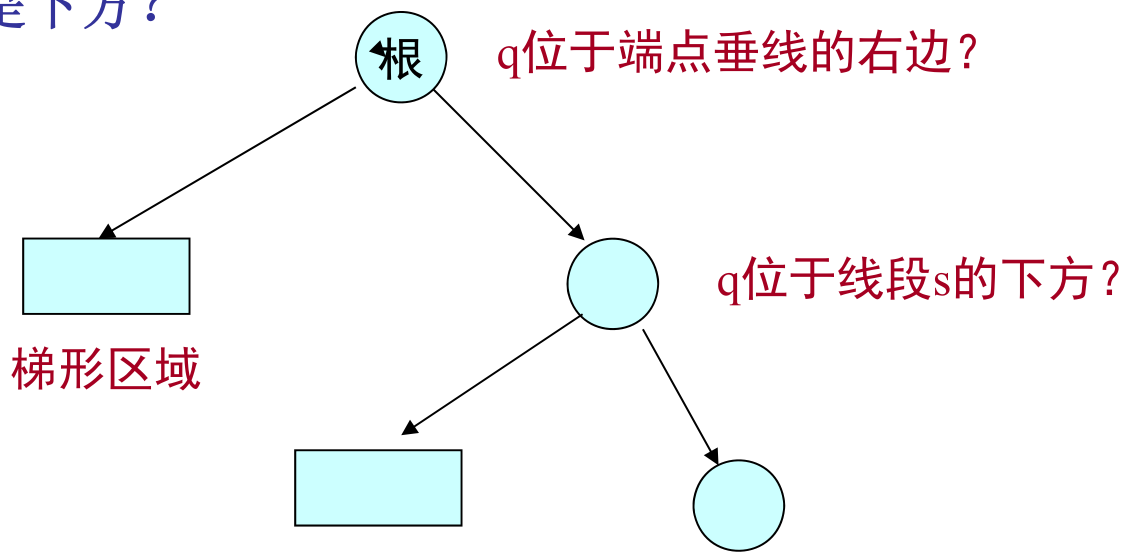
- 在对点q进行查询时，要从根节点出发，沿着某条有向路径到达某匹叶子。最终到达的那匹叶子，就对应于T(S)中包含q的那个梯形Δ
- 在沿查找路径前进时，每遇到一个新的节点，都要将其与q进行对比，以确定应该继续前进到其两个子节点中的哪一个



6.2 随机增量式算法

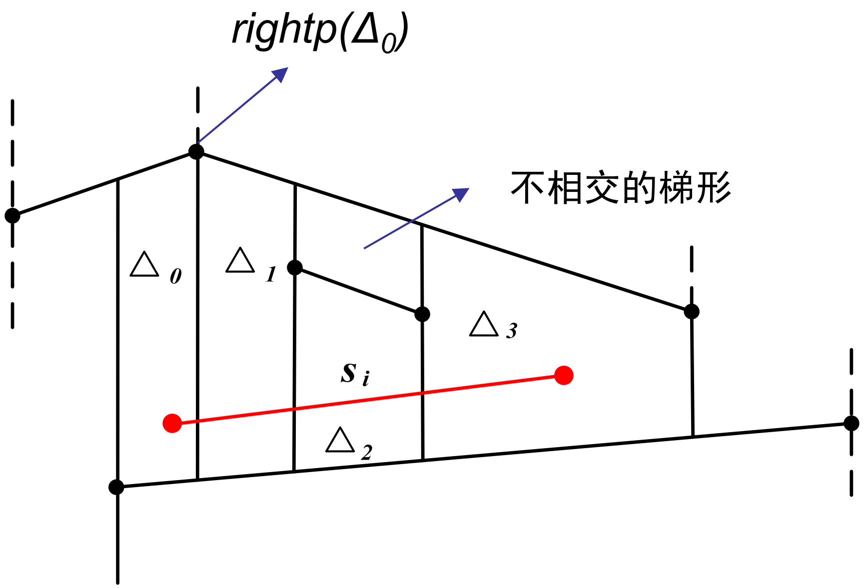
查找策略

- **若是x节点，则按如下形式进行比较：**
  - ✓ 取出存储于该节点处的那个端点；相对于经过该端点的那条垂线，**q**究竟是位于左侧还是右侧？
- **若是y-节点，则比较的方法如下：**
  - ✓ 取出存储于该节点处的那条线段**s**；相对于这条线段，**q**究竟是位于上方还是下方？



6.2 随机增量式算法

插入新线段s_i



- 只有与s_i 相交的那些梯形，才会有所变化

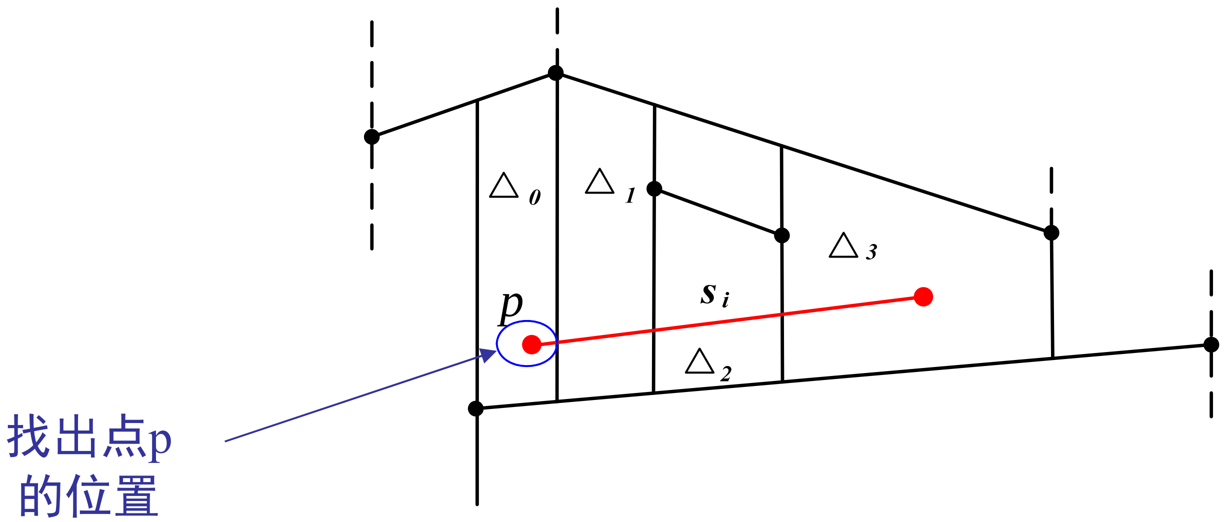
基本策略

- 按其与s_i相交的次序，将这些梯形从左到右记作$\Delta_0, \Delta_1, \ldots, \Delta_k$
- Δ_{j+1} 必然是Δ_j的右邻居之一：
  - ✓ 若rightp(Δ_j)位于s_i的上方，则Δ_{j+1} 必是Δ_j的右下方邻居；否则，就是的右上方邻居。
- 只要找到了Δ_0，就可以通过对梯形图结构的遍历，<u>顺藤摸瓜</u>地找出$\Delta_1, \ldots, \Delta_k$

6.2 随机增量式算法

插入新线段s_i

- 首先，在梯形图$T(S_{i-1})$中查找s_i的左端点p所在的梯形Δ_0



在$T(S_{i-1})$对应的查找结构D上对点p进行一次查询

查找结构的优势

6.2 随机增量式算法

找出Δ0, …, Δk的算法步骤

算法 FOLLOWSEGMENT(T, D, s_i)

输入： 梯形图T，与T 相对应的查找结构D，以及新近引入的一条线段s_i

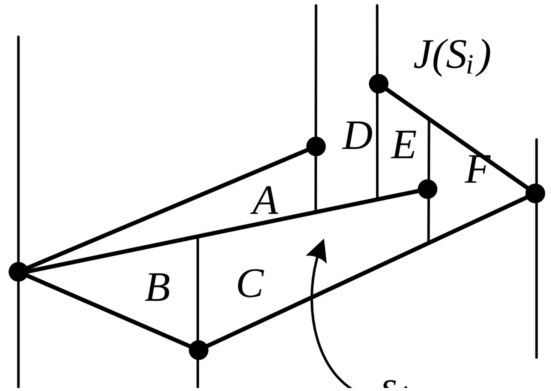
输出： 由所有与s_i 真相交的梯形组成的一个序列：Δ0, …, Δk

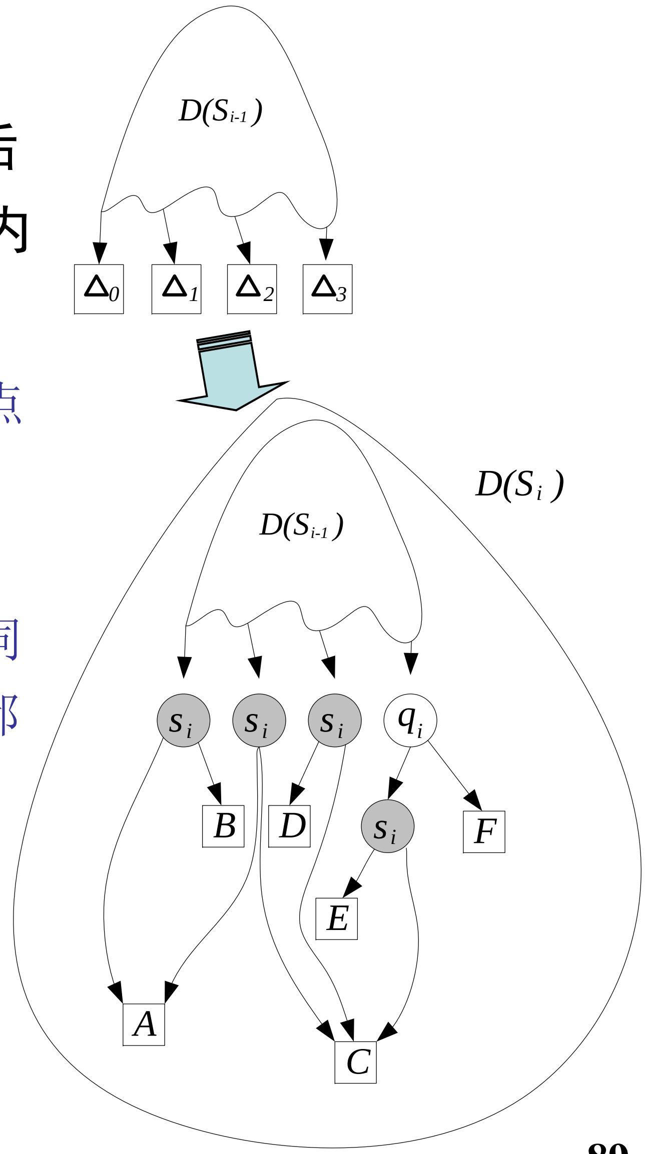
1. 分别令p 和q 为s_i 的左、右端点
2. 在查找结构D 中对p 进行查找，最终找到梯形Δ0
3. j ← 0
4. while (q 位于rightp(Δ_j)的右侧)
5. do if (rightp(Δ_j)位于s_i 的上方)
6. then 令Δ_{j+1} 为Δ_j 的右下方邻居
7. else 令Δ_{j+1} 为Δ_j 的右上方邻居
8. j ← j + 1
9. return (Δ0, Δ1,…, Δj)

6.2 随机增量式算法

复杂情况的讨论——s_i跨越多个梯形

- 对于D，删除对应于Δ0, Δ1, …, Δk的叶子，然后为每个梯形生成一个叶子，还要引入若干新的内部节点
  - ✓ 若s_i的左端落在Δ0内部，则用一个对应于s_i左端点的x节点，以及另一个对应于线段s_i本身的y节点，来替换原先对应于Δ0的那匹叶子
  - ✓ 然后，与{Δ1, …, Δk}对应的所有叶子，都被共同替换为同一个y节点（即s_i），所有节点的出弧都要正确的指向对应的新叶子

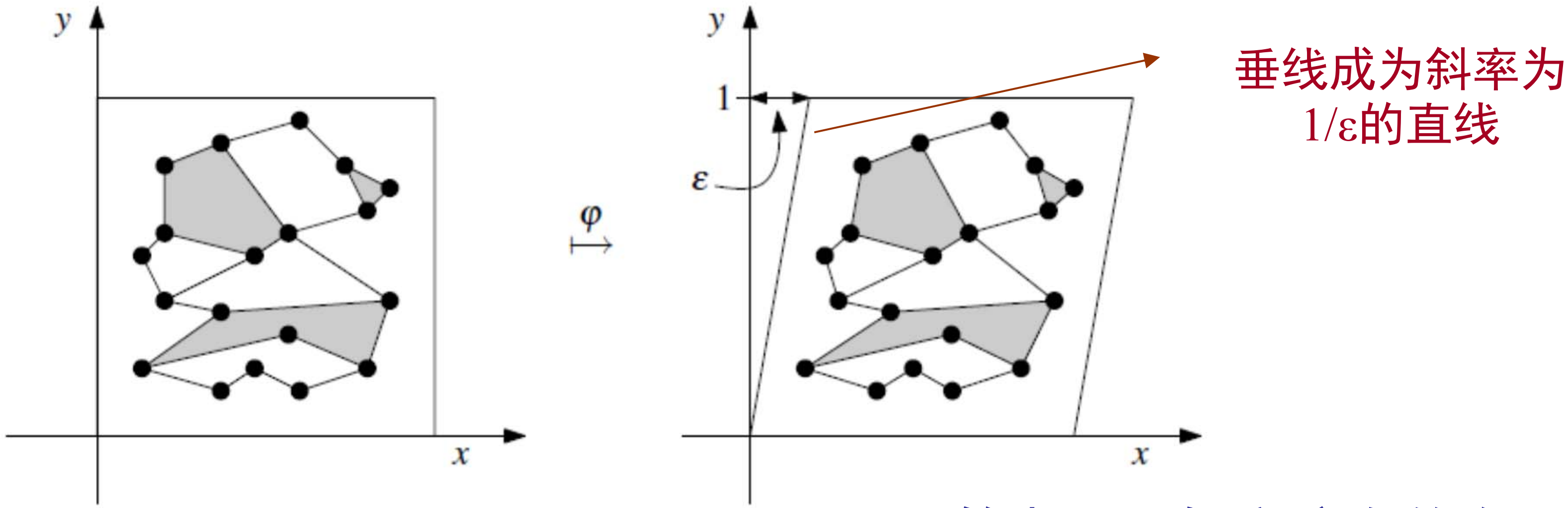




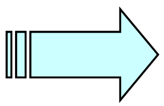
6.3 退化情况的处理

消除“不同端点不会落在同一条垂线上”的假定

- 策略：对坐标系略作旋转，只要角度足够下，就不会有任何两个端点落在同一条垂线上
  - ✓ 沿着x坐标方向做一个偏移量为ε > 0 的一个剪切变换



$$\varphi:\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x+\varepsilon y \\ y \end{pmatrix}$$



- 只要ε足够小，所有延x方向的次序将依然保持原样，因此ε不能超过某一上界

第七讲：Voronoi图—邮局问题

7.1 定义及基本性质

7.2 构造Voronoi图

7.1 定义及基本性质

P对应的Voronoi图

- 设P:={p_1,p_2, … ,p_n}为平面上任意n个互异的点，以这些点为基点做Voronoi图，即将平面划分为n个单元，它们具有这样的性质：

任一点q位于p_i所对应的单元中，当且仅当对于任何$p_j \in P, j \neq i$ 都有：

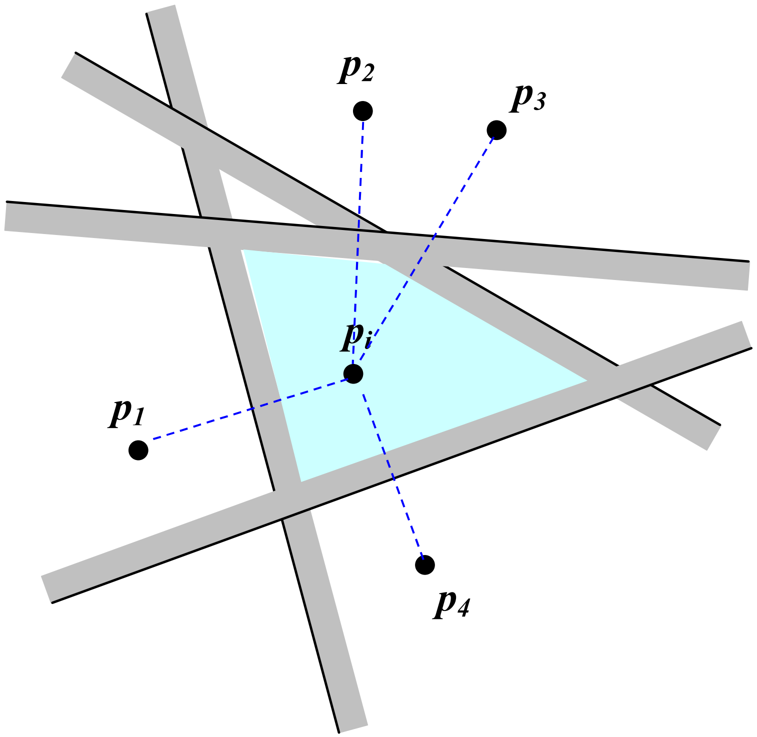
$$dist(q, p_i) < dist(q, p_j)$$

P对应的Voronoi图记做：Vor(P)

与基点p_i对应的单元记做：V(p_i)

7.1 定义及基本性质

$v(p_i)$是(n-1)张半平面的公共交集



一组平分线产生的结果

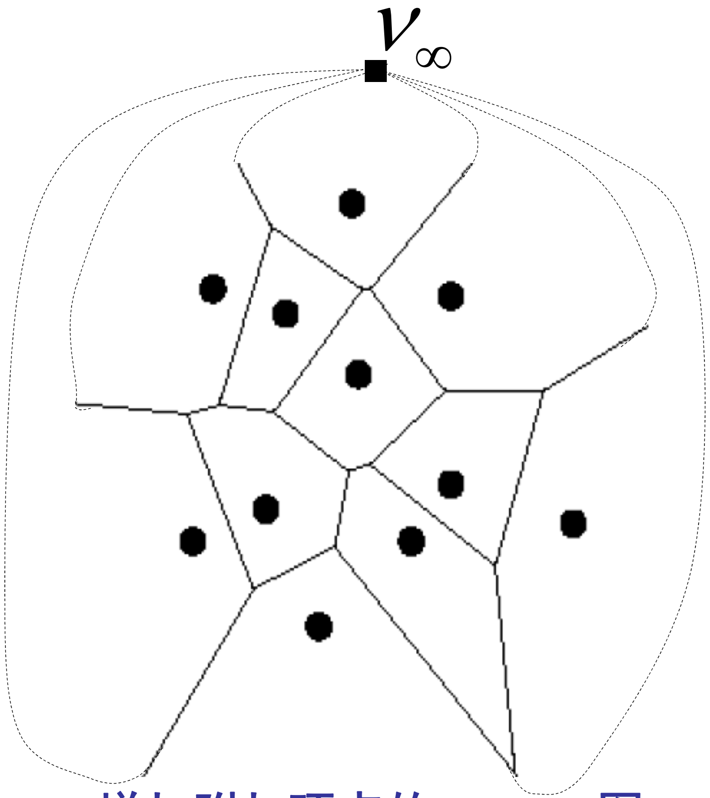
【观察结论7.1

$$v(p_i) = \bigcap_{1 \le j \le n,\, j \ne i} h(p_i, p_j)$$

7.1 定义及基本性质

Voronoi图的性质

- **定理：若n≥3,则在与平面上任意n个基点相对应的Voronoi图中，顶点的数目不会超过2n-5，而且边的数目不会超过3n-6**



增加附加顶点的Voronoi图

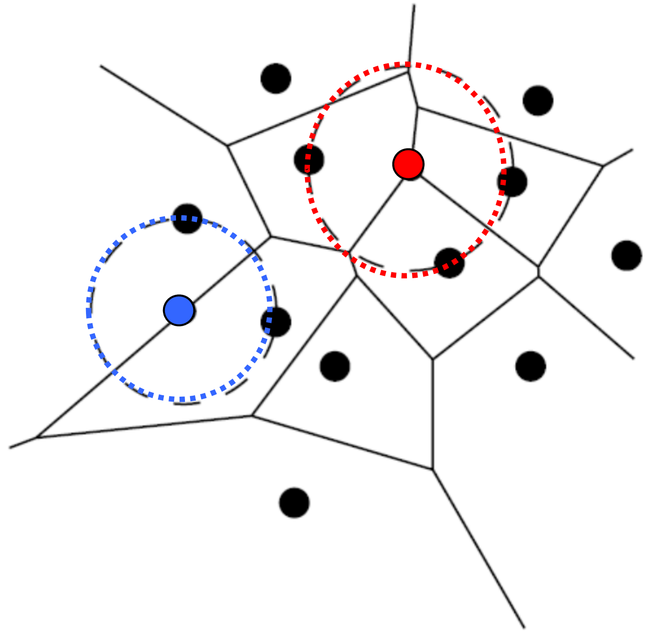
证明：

引入欧拉公式：$\underset{\text{顶点数}}{m_d} - \underset{\text{边数}}{m_e} + \underset{\text{面数}}{m_f} = 2$

$$\underset{\text{顶点}}{(n_v+1)} - \underset{\text{边数}}{n_e} + \underset{\text{基点数}}{n} = 2$$

Voronoi图的性质

- **【定理7.4】** 对于任一点集P所对应的Voronoi图Vor(P)，均有：
  ① 点q是**Vor(P)的一个顶点**，当且仅当在其最大空圆 $C_p(q)$ 的边界上，至少有三个基点
  ② p_i和p_j之间的平分线确定了**Vor(P)的一条边**，当且仅当在这条线上存在一个点q，$C_p(q)$ 的边界过p_i和p_j，但不经过其它基点



7.2 构造Voronoi图

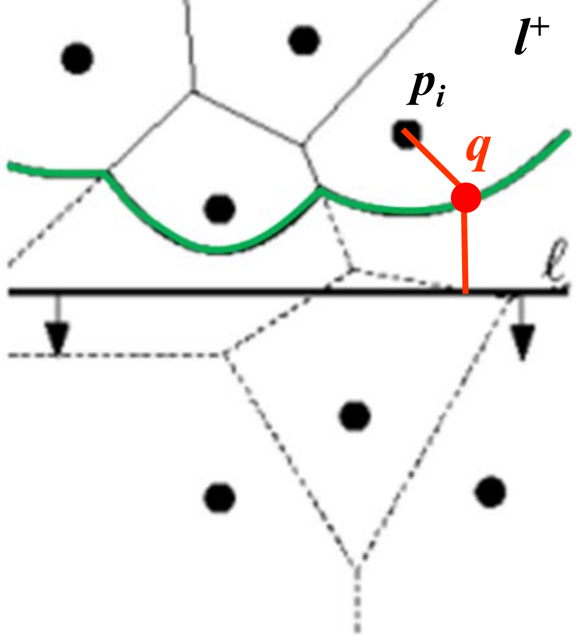
平面扫描策略

- 扫描线之上Voronoi图的哪些部分将不再发生变化？即对于哪些$q \in l^+$，已经可以确定与之最近的基点？

- 若存在一个基点$p_i \in l^+$，使得点q到p_i的距离不超过q到扫描线l的距离，那么q对应的基点不可能出现在l下方

- 距离某个基点比距离l更近的点所构成的集合，其边界是一条抛物线（与一个定点(焦点)距离和一条定直线（准线）的距离）

抛物线以上的点离p_i更近

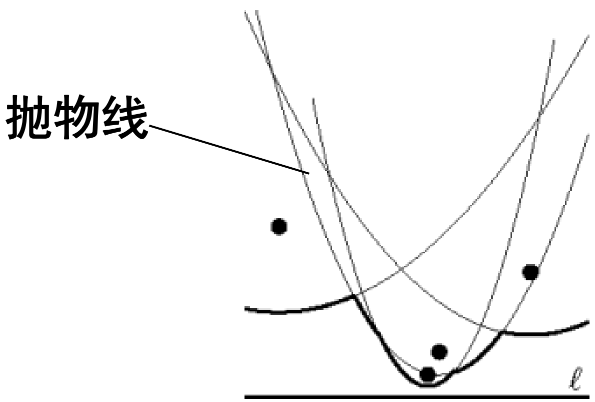


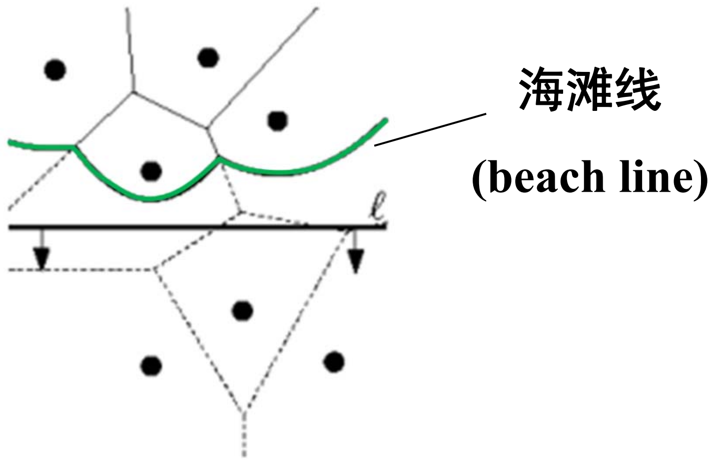
抛物线以上的部分不再发生变化

7.2 构造Voronoi图

海滩线（Beach Line）

- 观察：距离位于l之上的每个基点都要比距离l更近的那些点所构成的集合，其边界必然由若干段抛物线弧确定





所谓的海滩线，就是这样一个函数：对于任一x坐标，该函数的取值都是这些抛物线中的最低者

7.2 构造Voronoi图

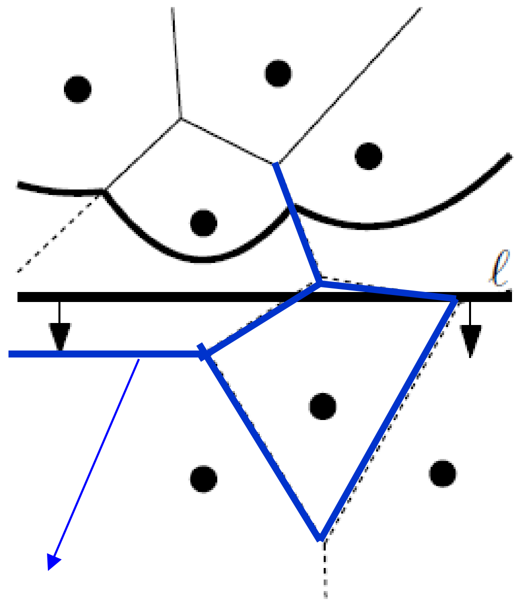
海滩线（Beach Line）

- 【观察结论7.5】：海滩线沿x方向单调，即它与任一垂线相交而且仅相交于一点

- 观察发现：

✓ 组成海滩线的抛物线弧依次首尾相联，其接合点称为断点，随着扫描线自上而下扫过整个平面，所有断点的轨迹合起来恰好就是待构造的Voronoi 图

✓ 有的抛物线可为海滩线贡献多段弧



断点的轨迹

7.2 构造Voronoi图

- **【引理 7.6】**：只有在发生某个基点事件时，海滩线才会有新的弧段出现

<u>**推论：**</u>组成海滩线的抛物线弧，总共不会超过 **2n-1** 段，只有在遇到一个基点时，才会生出一条新的弧，同时最多将原有的某一条弧一分为二；而其它的时候，海滩线上都不可能会有新的弧出现

7.2 构造Voronoi图

弧消失对应的圆点事件

- 【引理 7.7】：海滩线上已有的弧，只有经过某次圆事件后才可能消失

在原有弧段收缩为一个点时，p_i 、p_j 和p_k 在以q为圆心的圆上，扫描线继续下移原有弧段就完全消失了，形成的q点必是Voronoi图的一个顶点

7.2 构造Voronoi图

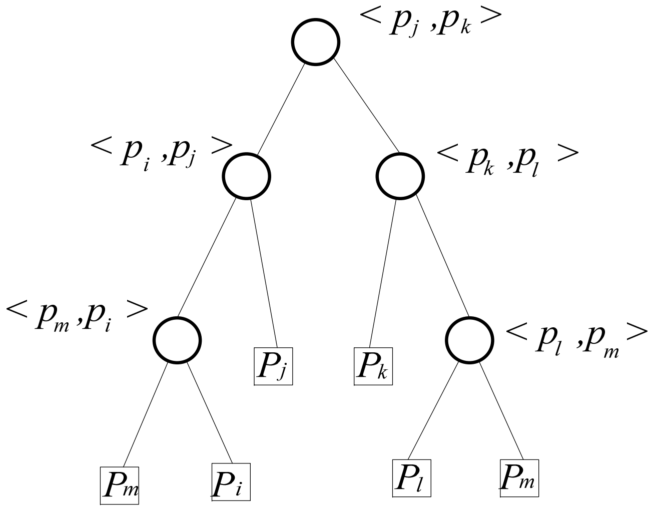
扫描状态的数据结构

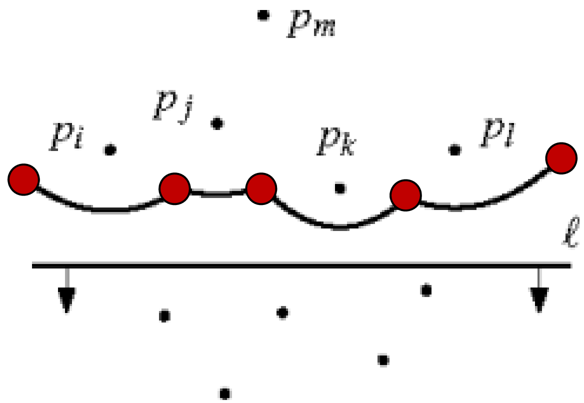
- **平衡二分查找树*T*：**

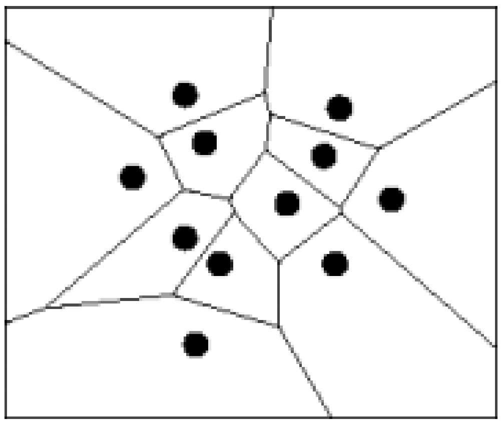
✓ 每片叶子分别对应于海滩线上的某段弧

✓ 每个根节点则分别对应于海滩线上的各断点（<u>不是基点</u>）

✓ 由于海滩线是*x*单调的，各段弧所对应的叶子必然是有序的







在T中，每遇到一个新基点时，可在O(log*n*)时间内找出位于该基点上方的弧

7.2 构造Voronoi图

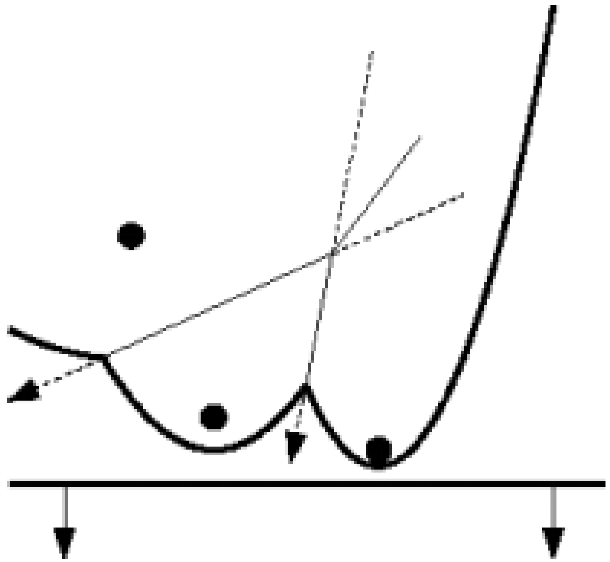
圆事件的确定

所有基点事件都可以事先确定，圆事件无法事先确定

- 为发现圆事件定义**邻接弧三元组**，即沿海滩线依次首尾衔接的任意三段弧，随着扫描线的运动可能出现新的三元组，原有三元组也可能消失，只要其确定了一个圆事件都要保证记录在事件队列中

两点说明：

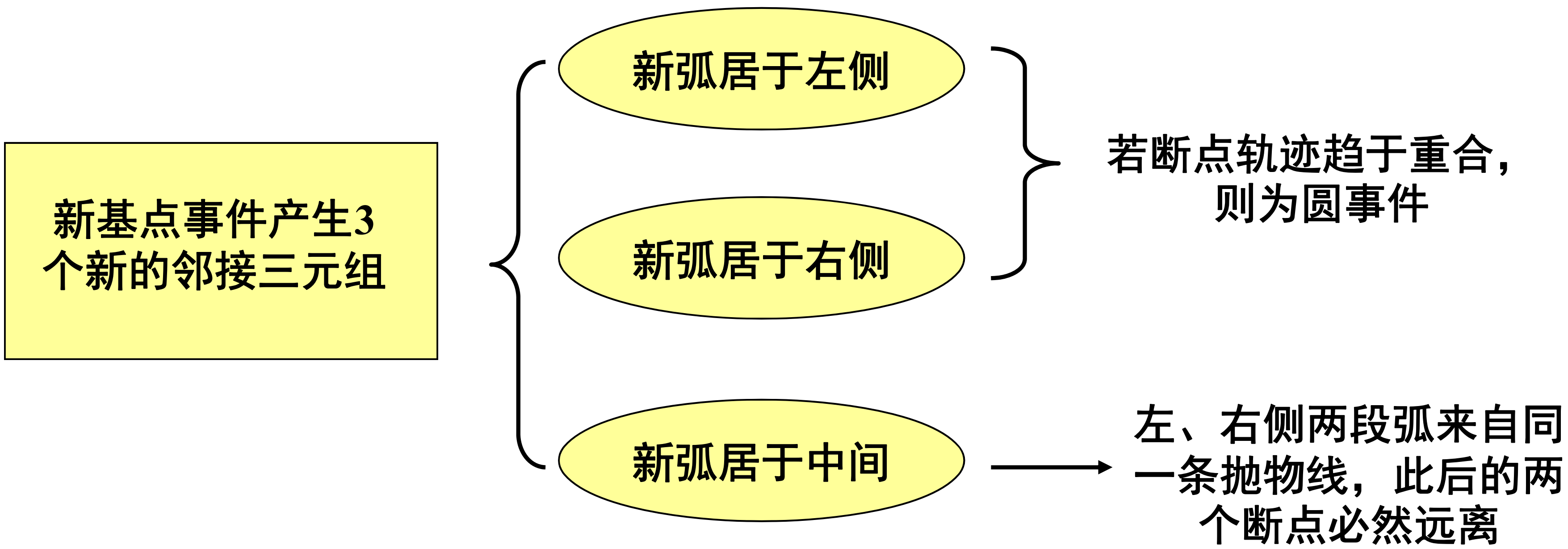
- 某些邻接弧三元组所对应的断点可能<u>不会汇聚到一起</u>
- 即使三元组对应的断点逐渐靠拢，圆事件也可能由于扫描线遇到了新的基点而没有发生，<u>这称为“误警”，要从事件队列删除</u>



7.2 构造Voronoi图

圆事件的处理策略

- 每遇到一个事件，都逐一检查新出现的邻接三元组

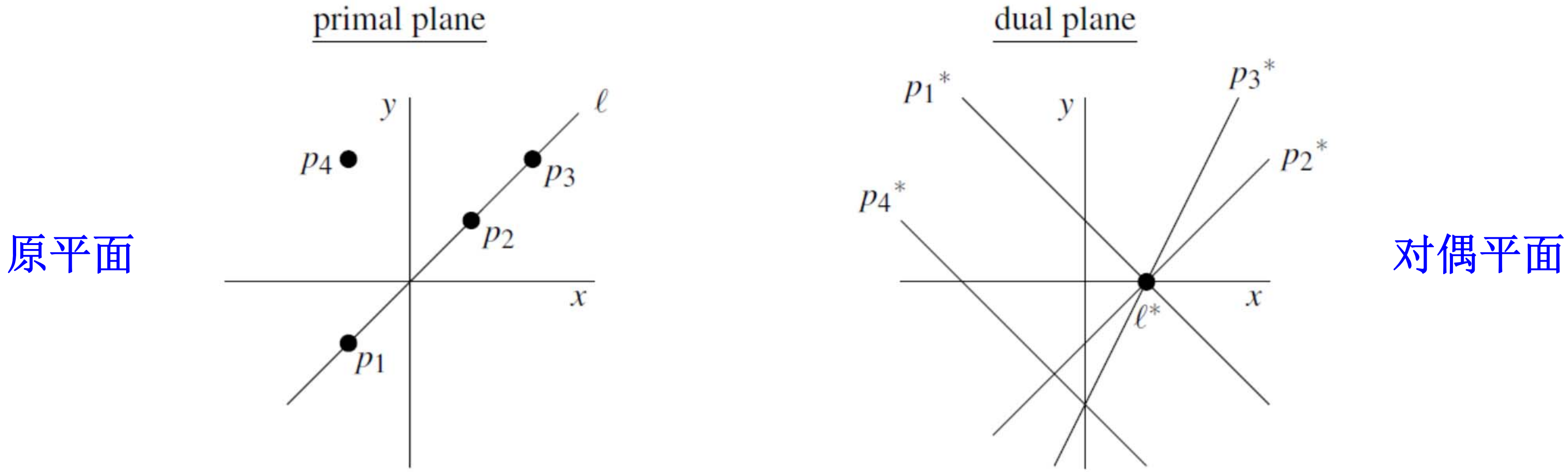


- 【引理 7.8】：每个Voronoi顶点，都会在某次圆事件发生时被发现

8 对偶变换

什么是对偶变换（duality transform）？

- 平面上的任何一点，都拥有两个参数——x坐标和y坐标
- 平面上任何一条（非垂直的）直线，也拥有两个参数——斜率，以及它与y坐标轴的交点



可以通过某种一一对应的方式，将一组点映射为一组直线，反之亦然。如果做得巧妙的话，甚至可以将原先点集所具有的某些性质，转换为直线集所具有的某些性质

8 对偶变换

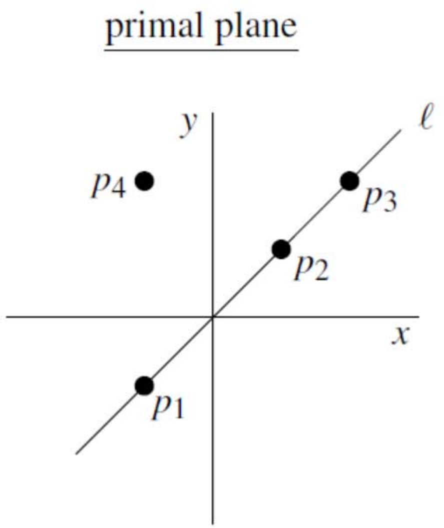
对偶变换的性质

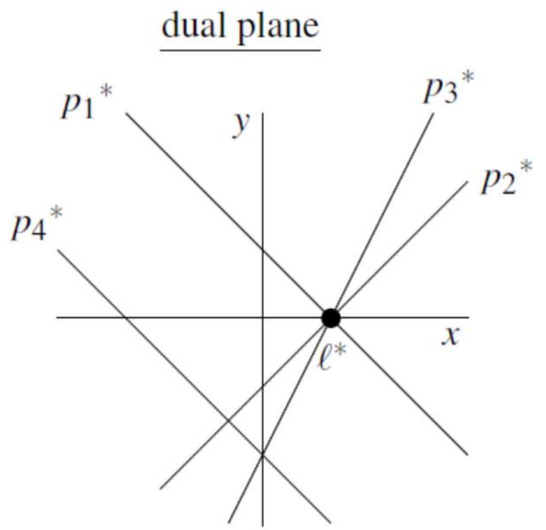
对偶变换将对象从原平面中映射到对偶平面，原来在原平面中具有的某些性质，在对偶平面中依然成立

- 设p 为平面上的一个点，l 为平面上一条非垂直线。则对偶变换$o \mapsto o^*$满足下列性质：

  A) 关联性的保持：$p \in l$ 当且仅当 $l^* \in p^*$；

  B) 位置次序的保持：p 位于l 的上方，当且仅当l^*位于p^*的上方





谢谢各位同学！