课程实验报告

1. 问题说明

集合 S 有 n 个点，凸多边形 P 有 m 个端点，寻找适当的移动使得 P 能够包含 S 中最多的点数。问题中，移动（translation）的含义是指：定义 $\tau(p)=p+\upsilon$, υ 是某个固定的向量，即对多边形 P 的每一个点计算 $\tau(p)=p+\upsilon$，得到新的多边形 $\tau(p)$。

该问题是许多搜索问题的一种变形。例如，找包含平面点集 S 的最小圆周是计算几何中的一个有名的问题。问题可以自然的扩展到寻找包含点最多的最小矩形、正方形、三角形和凸多边形。这个问题的另外一个变形是：给定平面点集 S 和固定整数 K，找到一个区域包含 S 中的 K 个子集，并使得面积或者周长等最小。

目前解决这个问题的方法有两种。第一种方法是利用 Line Sweep，需要的时间 O(nklog(nm)+m)，其中 k 是所包含的最多的点的数目。第二种方法称作 Anchored Sweep,需要 O(nklog(mk)+m)时间，是接近这个问题的解的最快算法。两种算法需要 O(n+m)的空间。算法对这个问题的变形问题能得到同样的运行时间。在该问题的研究中，90 年代中期，已经由平移拓展到旋转，鉴于课程试验时间限制，我们只作平移的讨论，并实现使用 Line Sweep 来实现问题的求解。

2. 算法原理

我们用 q_i 表示输入集 S 中的第 i 个点。假设多边形 P 用点 $p_1 \ldots p_m$ 顺时针表示。因此给定一种平移 τ 用向量 υ 表示，我们可以在常数时间内计算出 $\tau(p)$ 的第 i 个点为 $p_i+\upsilon$。

我们用 ∂p 表示多边形 P 的边界，也就是 P 的边和端点的集合。同样的，$\partial\tau(p)$ 是平移后多边形的边界。

算法的相关定义和定理描述如下。首先，在该算法的实现中，需要用到一个很重要的概念：平移稳定位置（translation stable placement），其定义如下：

定义 1 多边形 P 包含点集 S 中的点，且 $\tau(p)$ 是 P 的平移。如果 S 中至少有两个点在 ∂p 上，

我们就说$\tau(p)$处于平移稳定位置。

定理 1 S 表示平面点集，P 表示凸多边形。如果存在平移τ使得$\tau(p)$包含 S 中的$k \geq 2$个点，那么就存在平移τ^*使得$\tau^*(p)$也能包含至少 S 中的k个点并且处于平移稳定位置。

定理 2 P 表示一个凸多边形，两个点q_1和q_2，τ_1和τ_2是两个平移，把原点分别平移映射到点q_1和q_2。对于任意一个∂p上的点 x，定义$\tau_x = q_2 - x$是将x映射到q_2的平移。那么$\tau_1(x)$是$\partial\tau_1(p)$和$\partial\tau_2(p)$的交点当且仅当q_1在$\partial\tau_x(p)$上。（如图 1 所示）

图 1

定理 3 P 表示一个凸多边形，两个点q_1和q_2，τ_1和τ_2是两个平移，把原点分别平移映射到点q_1和q_2。对于任意一个∂p上的点 x，定义$\tau_x = q_2 - x$是将x映射到q_2的平移。那么$x \in (\tau_1(p) \cap \tau_2(p))$当且仅当$\tau_x(\tau_1(p))$同时包含了$q_1$和$q_2$。

定义 2 交点深度（depth）：交于该点的多边形的数目。

定义 3 区域：两条链之间包围的部分称作区域。

定义 4 区域深度（depth）：包含该区域的多边形的数目。

定义 5 上链、下链：按照字典序找出凸多边形的最左点 L 和最有点 R，这两点把多边形分成两条链，其中处于上方的称之为上链，处于下方的称之为下链。

定理 4 A 表示平面上 n 个多边形的一种放置，其中任何一对多边形的边界都最多相交两次。如果放置的最大深度$\leq k$，那么边界的交点数为$O(nk)$。

定义 6 P 表示一个多边形。定义 PR 为 P 的反射，PR 是将 P 绕着某个端点旋转 180 度（一

般选取最右边的端点)。

定理 5 PR 是 P 的反射，点 q 和 q_R。将 PR 的拷贝放置在 q_R 能够包含 q ，当且仅当将 P 的拷贝放在 q_R 能够包含 q 。

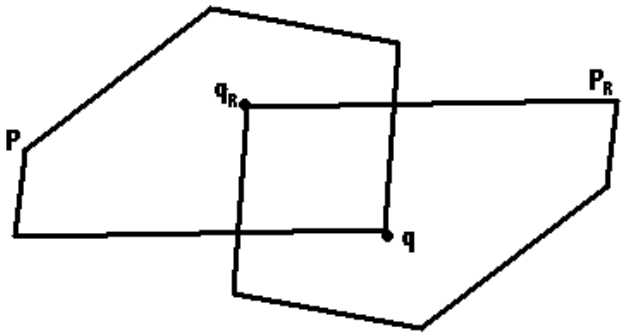

图 2

定理 6 P 表示一个多边形，PR 为 P 的反射，平面上两个点 q_1 和 q_2。如果将 PR 的拷贝分别放置在 q_1 和 q_2，如果它们存在交点 q，那么将 P 放置在 q，P 必然将包含点 q_1 和 q_2。

根据以上定理，我们推广得到以下结论：P 表示一个多边形，PR 为 P 的反射，平面点集 $S=\{q_1\cdots q_n\}$,将 PR 拷贝分别放置在 S 中不同的点上，如果能够得到深度最大的交点，那么将 P 放置在该点上，P 能够包含 S 中最多的点数。我们还可以将求交点的深度，改变为求区域的深度，根据前述定理，从深度最大的区域中任选一点，将 P 放置在该点，可以保证 P 包含 S 中最大的点数。这也是我们整个实验的核心思想。

3.算法及算法分析

3.1 算法的数据结构

与几何结构相关的数据类型:

顶点（Vertex 类)：能够存储点的坐标。

边（Edge 类)：存储 Polygon 的边信息，每条边包括左右两个端点，以及边的方向（向上或向下)。

链（Chain 类）： 存储 Polygon 的上链或者下链。

事件（Event 类）：存储事件点信息，包括顶点和事件类型等。

区域（Region 类）：存储在 y 轴上相邻的两条链围成区域。

平面扫描算法主要利用了的数据结构：

第一个是用于线扫描的事件队列。事件队列用来返回按字典序递增的下一个事件。Equeue::Intert 操作是插入一个事件，如果该事件已经存在，则不作任何操作。Equeue::Delete 操作取出一个事件。这两个操作都需要 O(logq)的时间，其中 q 是当前队列的大小。存储事件点，根据事件点的 x 坐标（没有退化情况）将事件点组织成一个优先队列，保证每次从优先队列中取出的都是当前没有处理的所有事件点中 x 坐标最小的。在具体实现中，采用了 Hight Biased Leftist Tree 实现。

第二个数据结构是当前多边形链树。为了减少求交点的难度，我们使用上下链来表示多边形。将平面扫描过程中与当前扫描线相交的各个链按 y 坐标排序，存储在一个平衡二叉树中。我们把这种结构称为链树。CMyAVLTree::Insert 操作将一个多边形链加入到链树 CT 中并返回新插入的链。CMyAVLTree::Delete 从树中删除一个多边形链。为了方便寻找某条链的相邻的两条链，我们在链数据结构中增加了 parent 指针，直接从事件点得到的链信息来寻找该链的相邻链，并不需要对链在 AVL 树中的位置进行定位，减少查找次数。

对每一对链树中的相邻链之间的区域都会关联一个深度。在这里我们用 Region 来记录区域深度，并记录下形成该区域的上下链。我们的目标就是找到最大深度的区域的。

3.1.1 事件类型

队列中的有三种类型的事件：

1. 起点事件（First Vertex）

P 表示一个多边形，PR 为 P 的反射。First Vertex 是多边形的最左边的顶点，也即初始顶点。在初始阶段将点集 S 的 n 个点作为起点事件加入到事件队列中，这意味着将 PR 放置在点集 S 的 n 个点上，使给定的 n 个点作为 n 个多边形的最左顶点。对起点事件事件，我们计算与之相关联的两个链 C_l 和 C_u，并将两个链加入到链树中（在开始的时候是连续的）。每个链与其相邻链的交点可以通过 Chain::Intersection 计算并加入到事件队列中。按照原文[1]算法，在一般情况下，C_l 和 C_u 之间的新区域的深度比这两条链链所插入的原始区域的深度增加 1，也就是每处理一个 First Vertex，新插入的两链之间的区域深度增加 1。原

始区域就被分成两个区域。但是在图 3 的特殊情况下，原文算法不能正确更新区域及区域深度，对于这种情况，我们提出新的准则，详细分析参见退化情况处理 3.4 节。

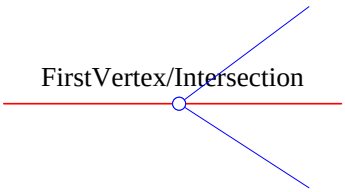

图 3

2. 终点事件（Last Vertex）

Last Vertex 是一个多边形的最后（最右）顶点，也形成一个事件。这个事件结束两个多边形链 C_l 和 C_u。因此，需要将 C_l 和 C_u 从链树中删除。这样，一个新的链对——我们称为 C_{ll} 和 C_{uu} ——在树中变得相邻了。需要检查这两个新的相邻链之间是否存在交点，如果存在，那么要生成交点事件并插入到事件队列中。原文[1]算法中，在该事件发生时 C_l 和 C_u 包围的区域结束，而两边的相邻区域合并成了一个新的区域。但是，在图 4 的特殊情况下，原文算法并不能正确处理，对于这种情况，我们提出了新的准则，详细分析参见退化情况处理 3.4 节。

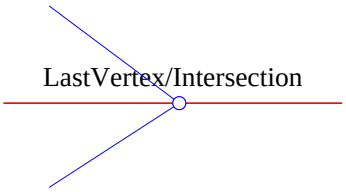

图 4

3. 交点事件（Chain Intersection）

最后一种类型是两个多边形链的交点。在这种情况下，链树中的形成该交点的两个链交换顺序，并且交换顺序后，每条链都要测试其是否与相邻的链有交点，如果有交点的话，要生成交点事件，插入到事件队列中。这两条链围成区域的深度的改变有以下情况：（其余区域的深度不发生变化）

（1）如果两个上链或者两个下链相交，那么区域深度不变。

（2）如果一个上链和一个下联相交，那么区域深度增加或者减少 2。具体情况如下：如果交点之后，处在交点上面的是下链，上面的是上链，那么区域深度减少 2（如图 5-a）；如果交点之后，处在交点上面的是上链，下面的是下链，那么区域深度增加 2(如图 5-b)。

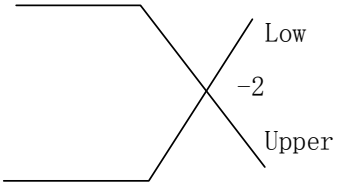

图 5-a

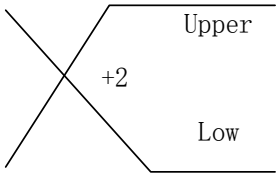

图 5-b

但是当出现多条链交于一点时，上述处理方法不能保证区域深度和链的顺序的正确性。对于这种情况，我们提出了解决方法，详细分析见退化情况处理 3.4 节。

3.1.2 事件队列（Hight Biased Leftist Tree　HBLT）

事件队列为一个优先队列，具体实现时采用 HBLT（Hight Biased Leftist Tree）。HBLT 具有堆的性质，但它是以二叉树的形式存储。在 HBLT 中，每个节点都不大于其左右子树中的任一节点，即 HBLT 相当于一个小根堆。插入删除的基本操作所需要的时间与堆一致。

3.1.3 平衡二叉树（AVL 树）

平衡二叉树（链树）采用 AVL 树来实现，我们的算法中, AVL 树中的每个节点的结构是 Chain。

Chain 的数据域包含：

链的最左最右两个端点（lend, rend）

形成链的边数(size)和边（Edge，为数组结构）

Chain 的类型（上链/下链）

指向与该链相关联的 Region 的指针

为了将链组织成一棵 AVL 树，我们在 Chain 类中定义了指向左右子树的指针。

此外，为了方便查找某条链的相邻链，我们在链的数据域里定义了一个指向父节点的指针。

在 AVL 树中的平衡因子 Balance（-1 表示左高，1 表示右高，0 表示左右高度相等）

平衡二叉树的主要操作是插入和删除，以及寻找某链的相邻链。在链中除了定义指向左右子树的指针外，还定义了父指针。在插入过程中，同时对父指针进行更新，方便其他操作。在删除操作时，我们可以保证要删除的链必然已经存在于 AVL 树中，可以减少链的定位，

直接对该链进行删除操作，提高效率。在寻找相邻链的操作中，利用父指针可以加快查找速度，提高效率。

3.2 算法总过程

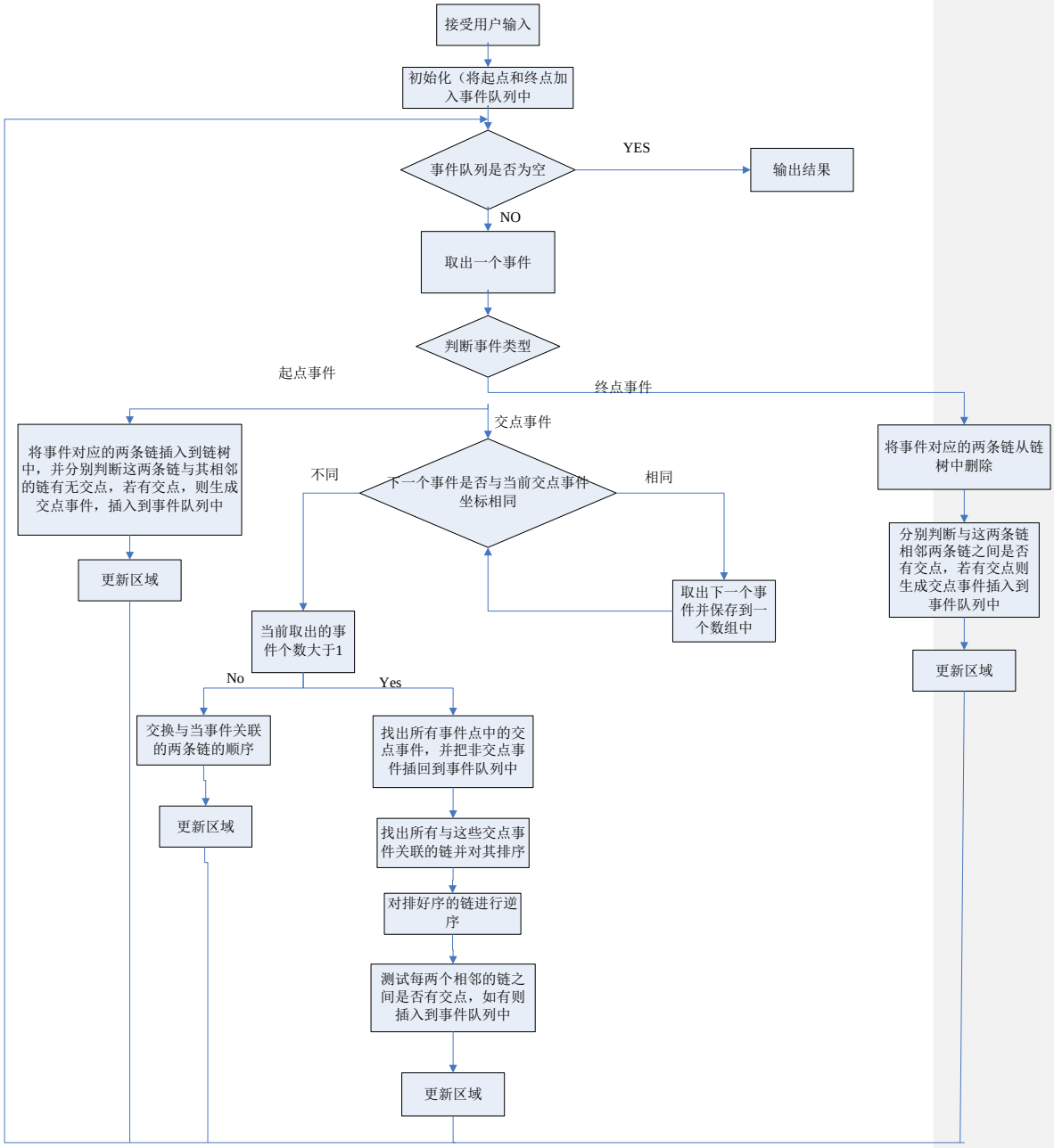

图 6

3.3 算法复杂度分析

假设多边形边数为 m，给定的查询点数为 n，最大深度<=k。

3.3.1 空间复杂度

事件队列需要的空间为 2m+k，AVL 树需要的空间为链数 2n。所以，算法总的空间复杂度为 O(m+k+n)。

3.3.2 时间复杂度

给定的点数为 n，即 FirstVertex 和 LastVertex 的个数为 n。最大深度<=k，则交点数为 O(nk)。处理事件队列的时间为 O(nk)。同样的，任何时刻最多有 2n 个链同时处理，所以链树的处理时间为 O(logn)。求链的交点时我们用追赶法，时间复杂度为 O(m)。对多边形进行预处理的时间为 O(m)。所以，算法总的复杂度是：

$$T = O(nk(\log n + m) + m))$$

需要说明的是，在对链求交时可能有更加好的算法[3]，使得求交的时间复杂度为 logm。鉴于难度所限，我们在这里用的是追赶法。

3. 4 退化情况处理

下面是我们实验调试过程中发现有可能出现的退化情况：

正常（非退化）情况下，处理起点事件，向 AVL 树中插入链时，链之间的大小比较：

如图 7 所示，正常情况下，处理链 b 的起点事件时，需要向链树中插入链 b。由于链 b 的起点不处在链 a 的任何一条边上，此时，要判断链 a 和链 b 之间的大小，只需要判断链 b 的起点和链 a 的关系即可。从上图中很容易可以看出，由于链 b 的起点处于链 a 的下方，所以有 a<b，其中小于号表示链 b 在链 a 的下方。根据链 a 和链 b 之间的偏序关系可以确定插入位置。

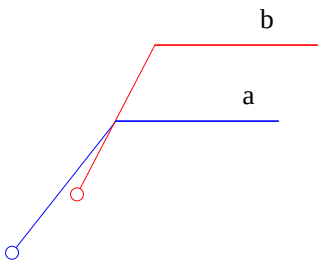

图 7

稍微有一点退化如图 8 所示，即链 a 的起点处于链 b 的某条边上，但是二者没有边重合。此时可以根据链 a 的第一条边的终点相对于链 b 的位置来确定链 b 和链 a 的关系。如下图示，可通过点 v 与链 b 的位置关系（左转还是右转）来确定链 a 与链 b 的位置关系，图中由于 v 处于链 b 的上方（左转），所以链 a 处于链 b 的上方，即 a<b。

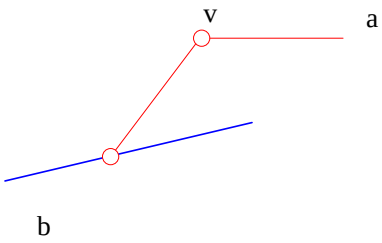

图 8

退化情况：

基本处理策略：

1、 如果某条链的起点恰好落在另外一条链的某条边上，那么在求这两条链的交点时，这个起始点不算作交点。

2、 如果某条链的终点恰好落在另外一条链的某条边上，那么在求这两条链的交点时，这个终点不算作交点。

3、 两条链有部分边重合时，虽然理论上是有无穷多个交点，但处理时根据不同的情况确定交点数目，交点数目最多不会超过两个。而且两条链的相对大小也要根据具体情况来确定。

4、 如果两个待查询的点坐标完全一致，则认为这两个点是一个点。如果这两个点被多边形覆盖，那么输出结果时，认为覆盖的点数为 1，而不是 2。

5、 如果某个事件点既是起点，又是交点，还是终点，那么事件点的处理顺序为：先终点，后交点，最后起点。

6、 多个交点事件，交点的坐标相同，相交的边不同，此时不是对单个交点事件进行处理，而是一次性的对所有相同的交点进行处理。

7、 对两个交点事件，如果交点坐标相同，而且第一个交点对应的上链是第二个交点对应的下链，并且第一个交点对应的下链是第二个交点对应的上链，那么认为这两个交点事件实际上是一个交点事件。

这些只是一些基本的策略，还有很多退化情况不能依靠这些策略完全解决，需要一些新的策略。这些新的策略将在下面具体介绍。

退化情况 1

图 9-a 所示的情况，链 b 的起点处于链 a 的某条边上。在处理链 b 的起点事件时，需要将链 b 插入到链树（实现时采用的是 AVL 树）中，这样就需要比较链 a 和链 b 的大小，从而确定链 b 在链树中的位置。

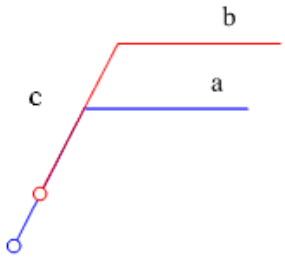

图 9-a

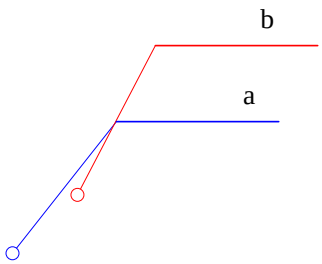

图 9-b

位置关系的处理：此时由于点在 a 的边上，所以不能采用一般的链之间的大小比较算法。对图 9-a 所示情况，我们定义 a<b，其中小于号表示 b 处于 a 的下方。
交点的处理：在求链 a 和链 b 的交点时，由于链 a 和链 b 有一段边重合，所以，交点有无穷多个。但是，我们定义它们的交点只有一个，就是两条链开始分开的那一点（图中为 c 点）。我们对这种退化情况的处理，使得该退化情况相当于图 9-b 的非退化情况。

退化情况 2

图 10-a 所示的情况，链 b 的起点处于链 a 的某条边上。

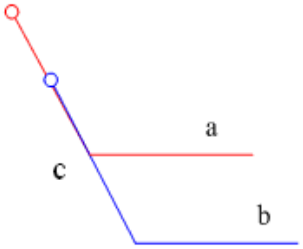

图 10-a

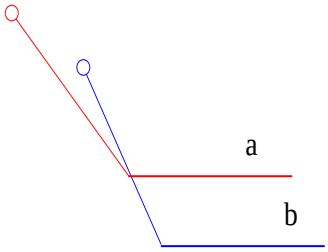

图 10-b

位置关系的处理：我们定义上图中 a>b，即链 b 处于链 a 的上方。
交点的处理：链 a 和链 b 的交点只有一个，就是两条链开始分开的那一点。
这种退化情况经过处理后相当于图 10-b 对应的非退化情况。

退化情况 3

图 11-a 所示的情况，链 b 的起点位于链 a 的某条边上。

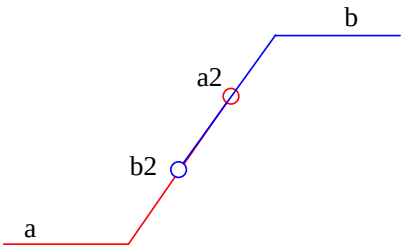

图 11-a

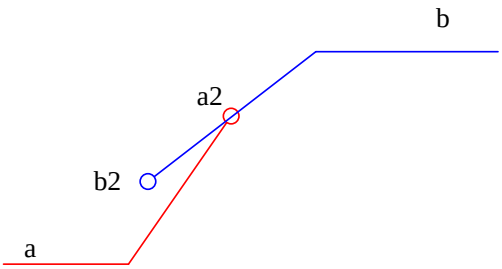

图 11-b

位置关系的处理：我们定义 a>b，即链 b 处于链 a 的上方。
交点的处理：对两条链求交时，根据前面定义的策略，起点和终点均不算作交点，所以如果 a2 是链 a 的终点，那么定义这两条链没有交点。
这种退化情况经过处理后相当于图 11-b 中的非退化情况。

退化情况 4

图 12-a 所示的情况，链 b 的起点位于链 a 的某条边上，并且链 b 的起始边和链 a 的某条边部分重合。

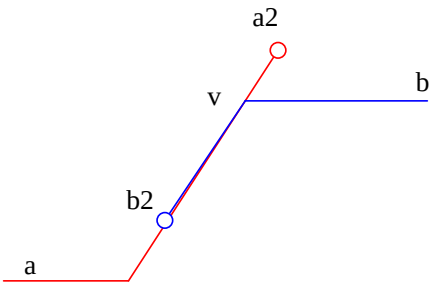

图 12-a

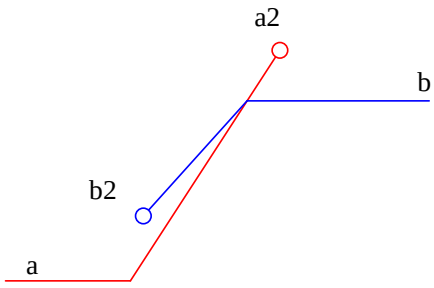

图 12-b

位置关系的处理：我们定义 a>b，即链 b 处于链 a 的上方。

交点的处理：对两条链求交时，由于起始点不算作交点，所以定义两条链只有一个交点，该交点为两条链开始分开时的点（图中为 v）。

这种退化情况经过处理后相当于图 12-b 中的非退化情况。

退化情况 5

图 13-a 所示的情况，链 b 和链 a 有部分边重合。

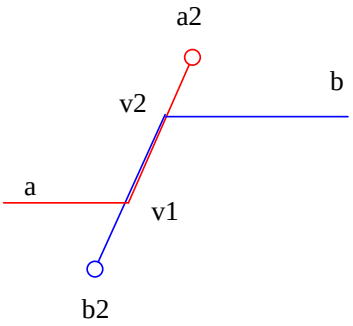

图 13-a

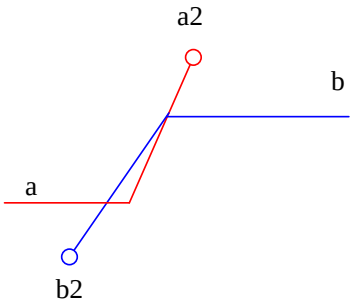

图 13-b

位置关系的处理：可以直接用非退化时的链的比较大小算法来比较大小。本图中显然有 a<b。

交点的处理：在对两条链求交时，虽然两条链有一段边重合，理论上应该有无穷多个交点。但是我们定义它们有两个交点，分别是开始重合的点（v1）和开始分开的点（v2）。

这种退化情况经过处理后相当于图 13-b 所示的非退化情况。

退化情况 6

图 14-a 所示的情况，链 b 和链 a 有部分边重合。

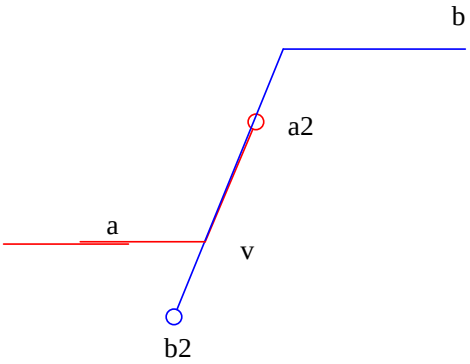

图 14-a

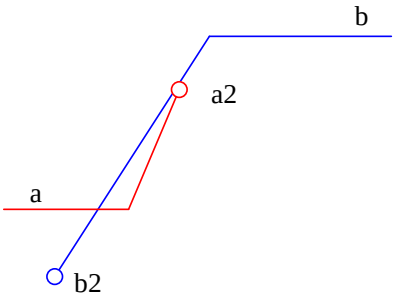

图 14-b

位置关系的处理：可以直接用非退化时的链的比较大小算法来比较大小。本图中显然有 a<b。

交点的处理：在对两条链求交时，如果 a2 为链 a 的终点，由于终点不算作交点，所以定义两条链只有一个交点，该点为这两条链开始重合的第一个点。否则，a2 也是这两条链的交点。

处理后的退化情况相当于图 14-b 所示的非退化情况。

退化情况 7

图 15-a 所示的情况，链 b 的起点位于链 a 的某条边上，并且链 b 的起始边和链 a 的某条边部分重合。

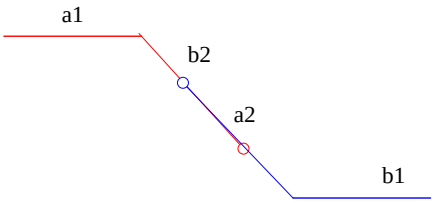

图 15-a

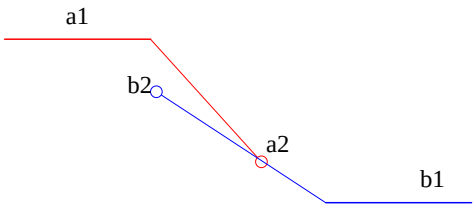

图 15-b

位置关系的处理：我们定义 a<b，即链 a 处于链 b 的上方。

交点的处理：对两条链求交时，如果 a2 是链 a 的终点，那么定义这两条链没有交点。否则定义 a2 就是这两条链的交点。

这种退化情况相当于图 15-b 中的非退化情况。

退化情况 8

图 16-a 所示的情况，链 b 和链 a 有部分边重合。

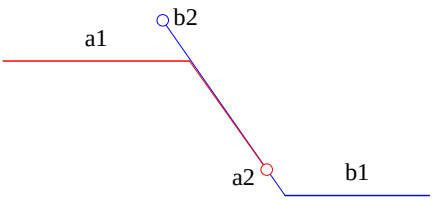

图 16-a

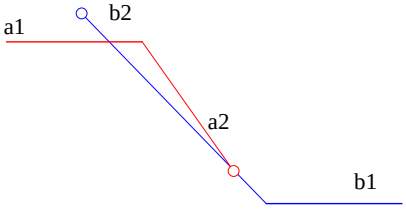

图 16-b

位置关系的处理：可以直接用非退化时的链的比较大小算法来比较大小。本图中显然有 a>b。
交点的处理：在求交时，如果 a2 为链 a 的终点，由于终点不算作交点，所以定义两条链只有一个交点，该点为这两条链开始重合的第一个点。否则，a2 也是这两条链的一个交点。
处理后的退化情况相当于图 16-b 所示的非退化情况。

退化情况 9

图 17-a 所示的情况，链 b 的起点位于链 a 的某条边上，并且链 b 的起始边和链 a 的某条边部分重合。

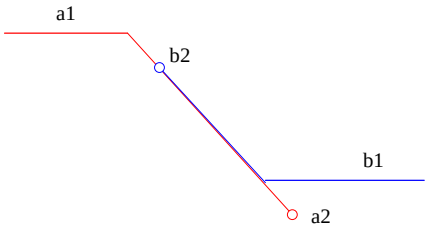

图 17-a

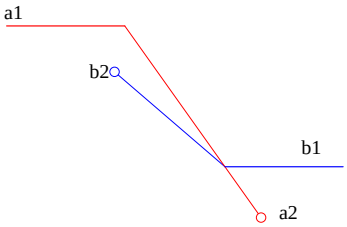

图 17-b

位置关系的处理：我们定义 a<b，即链 a 处于链 b 的上方。
交点的处理：对两条链求交时，因为起始点不算作交点
，所以定义两条链只有一个交点，为两条链开始分开时的点。这种退化情况相当于图 17-b 中的非退化情况。

退化情况 10

图 18-a 所示的情况，链 b 和链 a 有部分边重合。

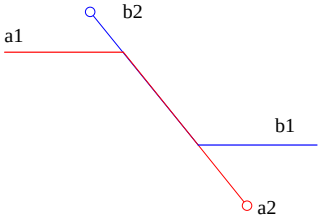

图 18-a

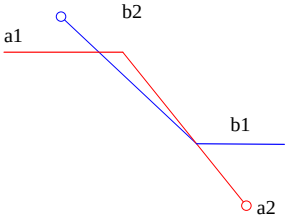

图 18-b

位置关系的处理：可以直接用非退化时的链的比较大小算法来比较大小，此时显然有 a>b。
交点的处理：定义它们有两个交点，分别是开始重合的点和开始分开的点。
通过处理，这种退化情况相当于图 18-b 所示的非退化情况。

退化情况 11

图 19 所示的情况，有一个起点处于某条链的一条边上。

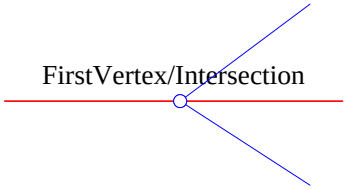

图 19

位置关系的处理：这种情况下，在向链树中插入起点事件关联的两条链（该多边形的上、下链，图中为蓝色的链）时，需要比较这两条蓝色的链和红色链的位置关系。此时可以根据蓝色链的第一条边的终点相对于红色链的位置关系来确定蓝色链和红色链的位置关系。前面已经做了详细说明，此处不再重复。

交点的处理：定义起始点不能作为交点，所以此处不存在交点。

退化情况 12

图 20 所示的情况，有一个终点处于某条链的一条边上。

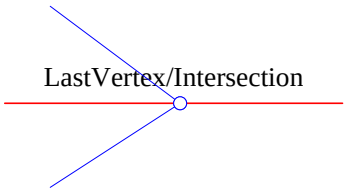

图 20

位置关系的处理：链的位置关系是根据起始点确定的，与终点无关。

交点的处理：终点不能当作交点，所以此处也不存在交点。

注：在 Translating a convex polygon to contain a maximum number of points[1]一文中，作者认为对起始点事件的处理，就是把起始点对应的两条链插入到链树中，然后将起点事件所处的原始区域的深度值加 1，作为**新插入的两链之间的区域**的深度。作者认为新插入的两条链必然围成一个区域，但是实际上这种情况只有在非退化情况下才成立。比如在上面的图 19 中，由于起始点刚好落在某条链的一条边上，显然两条蓝色链之间包含了两个区域，而且显然这两个区域的深度值还不一样，此时如果按照作者在文章中提供的算法进行处理的话，就会出错。

同样，处理端点事件的时候，作者也认为只需要简单的将**两条链围成的区域**删除即可。但从图 20 中可看出，此时两条蓝色的链之间有两个区域。

因此，实际对起点和终点事件进行处理时，我们的做法跟文章[1]中提供的算法不同。如在处理起点事件时，对起点关联的两条链，我们分别进行处理。首先插入一条链到链树中，该链必然会把原来的某个区域一分为二，我们对新生成的区域深度进行调整。然后对另外的一条链，进行类似的操作，这样就能够处理图 19 中的退化情况。处理终点事件也很类似，关键的想法就是将两条链分开来处理。

退化情况 13

图 21 所示的情况，多条链交于一点。

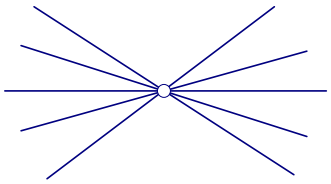

图 21

位置关系的处理：链的位置关系是根据起始点确定的。

交点的处理：终点不能当作交点，所以此处也不存在交点。

在图中显示只有一个点，但是该点会对应事件队列中的多个事件。

在 Translating a convex polygon to contain a maximum number of points[1]一文中，作者只考虑了两个链相交的情况，如果是两个链相交，只需要交换两个链在链树中的位置，并对链关联的区域进行更新即可。但是，多个链交于一点时，会形成多个交点事件。此时不能用文章中提供的方法，对这些交点事件逐个的进行处理。因为逐个进行处理，处理完后，多条链之间的位置可能不正确。如图 22 所示：

1、2、3 三条链交于一点。在交点之前，3 条链的顺序为 1<2<3，交换后，链的顺序应该为 3<2<1。如果链的插入顺序为 1 ，2，3。那么会形成两个交点事件，即 1 和 2 会形成一个交点事件，2 和 3 会形成一个交点事件。处理 1 和 2 形成的交点事件时，交换 1 和 2 的顺序，链的位置关系变为 2<1<3。再处理 2 和 3 的交点事件，继续交换 2 和 3 的顺序，链的位置关系变为 3<1<2，这就是交点事件处理完毕后，链之间的位置关系。显然，这是不正确的。

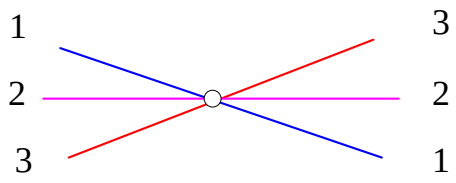

图 22

我们采用的处理方法为：如果有多个链交于一点（退化情况，出现可能性较小），那么我们首先取出所有坐标等于该点的交点事件，并找出与这些交点关联的所有链，并对它们按照从高到低的顺序进行排序，排序完成后，再逆序，此时得到的便是交点之后链之间的位置关系。这样处理能够保证该交点之后的链位置关系是正确的。接下来，每相邻的两条链之间分别求交点，如果有交点的话，生成新的交点事件，并插入到事件队列中。然后再调整链对应的区域。

相应于图 22，我们首先找出所有的交点事件，此处为 1 和 2 形成的交点事件，2 和 3 形成的交点事件。然后找出这些交点事件对应的所有的链，即 1、2、3。然后对这些链进行排序，排序完成后链之间的位置关系为 1<2<3。然后逆序，逆序的结果为 3<2<1，从图中很容易可以看出交点之后链之间的顺序就是 3<2<1。接下来再分别求 1 和 2，2 和 3 之间的交点，如果有交点，则生成交点事件，并插入到事件队列中。最后对区域进行调整。

实际上，这儿的链排序算法也有其特殊之处。因为待排序的链之间并不是两两都可比的，只有与同一事件关联的两条链之间才具备可比性。比如上图中 1 和 2 生成了一个交点事件，因此，1 和 2 可比，且 1<2；2 和 3 生成了一个交点事件，2 和 3 可比，且 2<3；但是 1 和 3 之间就不具可比性。

考虑到链树中实际上已经保存了这些链之间的相对位置关系，因此我们只要找到最上面的那条链，然后经过一次遍历，用线性事件就可以完成排序了。具体实现时，我们通过链树，找到交于该点的所有链中最上面一条链的上一条链，然后从它开始，向下遍历，便完成了排序。

退化情况 14

在图 23 中，中间的那个点表示一个凸多边形的终点和另外一个凸多边形的起点重合，另外，还有几个链的交点也恰好和这个点重合。

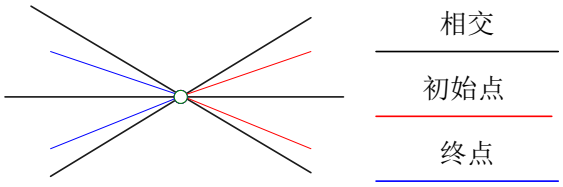

图 23

这种情况实际上是多种退化情况的综合，此时事件的处理顺序为先终点，后交点，最后起点。因为处理完终点、交点后，再处理起点，所以处理起点时并不知道有一个终点和起点重合，所以起始点的深度值会比正确值小 1。为了防止这种情况，我们在处理起点的时候，判断一下是否存在已经处理过的终点，该终点的坐标与起点一样。如果一样，那么就应该新生成一个区域，该区域就是这个点，区域深度就是终点事件对应的深度加 1。

4. 实验结果分析

我们对点数为 100、200、300、400、500 的情况下分别对四边形，五边形和六边形所需要的运行时间进行了测试，得到了下面的结果曲线图（图 24 的横轴表示输入的点数，纵轴表示运行时间，单位为 ms）和柱状图（图 25）。

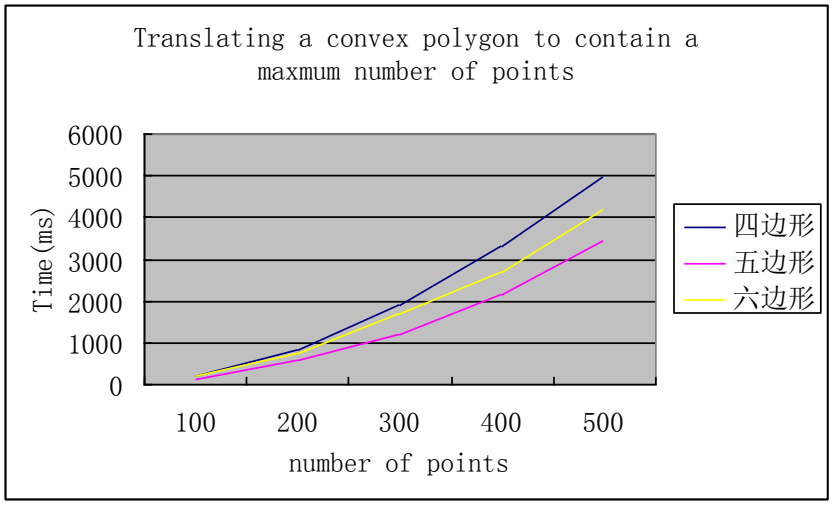

图 24

由于对相同点数的随机不同分布，运行结果不是完全相同的，所以图中的结果取的都是平均值。我们可以看到，对同一个多边形，当输入点数增大的时候，所需要的时间是呈增加趋势的。曲线接近 log 量级，这与我们算法分析的理论值是相符合的。

从图 24 中我们看到，四边形的曲线在五边形和六边形之上，而五边形的曲线在六边形之下，这说明时间和边数没有一定的关系。因为，具体的时间和多边形的具体形状是有关的。就算是相同边数的凸多边形，如果形状不同（包括具体的形状和大小），结果也会相差很大。

总之，得到的多边形包含的最大点数和运行的时间不但和问题的规模和点的分布有关，而且和多边形的具体形状有关。

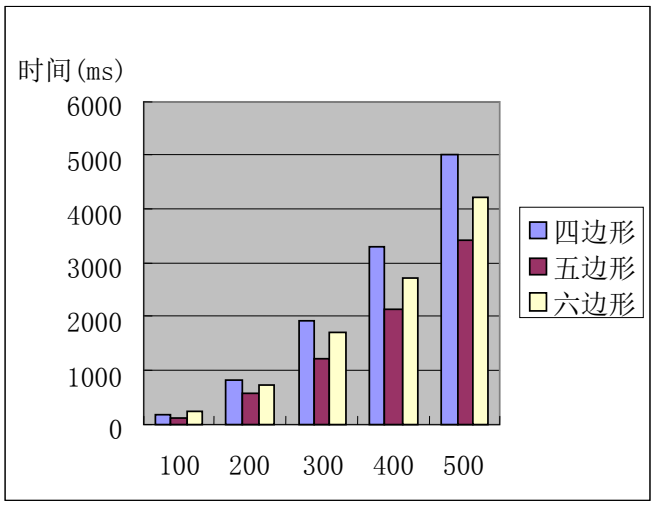

图 25

5. 与上次检查时相比所作的改进

用户界面方面：

1） 增加了随即机输入点的功能，而且可以限制这些随机点的分布范围。这就避免了当用户需要测试较大数目点时，要用鼠标逐个输入的困难。
2） 可以用鼠标拖动一个多边形在屏幕上移动，信息栏窗口会显示该多边形处于当前位置时所覆盖的点数目。
3） 改正了清空时的 bug，每进行一次新的测试只要新建一个空白文档即可，而以前必须关掉程序，然后重新运行。
4） 信息栏可以显示运行时间等信息，方便用户检查算法性能。
5） 增加了数据导入、导出功能

带格式的: 项目符号和编号

算法方面：

1）增加了对退化情况的处理
2）改正了原来的一些 bug

6. 存在的问题

作者在原文中并没有考虑特殊情况的处理，我们在最初设计时，按照原文算法描述实现，在上次检查中，小规模点数（<100），运行结果是正确的。但当点数规模增大到 200 以上时，点密度达到一定条件时，算法执行会出现异常。经过分析，我们发现，当点数增多到一定条件时，会出现很多我们以前没有考虑到的退化情况，造成算法执行错误。为此，我们提出了一系列准则，试图解决这些问题，在调试过程中，发现这些准则的确能够解决我们已经发现

的这些退化情况。但是，在点数比较大，点密度比较大的情况下，程序运行还会出现错误，我们认为这是由于还有一些未发现的退化情况造成的。截止目前，还没有完全解决问题，这是我们继续努力的方向。

对于链求交算法，我们在实现过程中，采用的是追赶法查找两条链的交点，时间复杂性为 O(n+m)（n,m 分别为两条链的边的数目）。利用这种方法，没有充分利用链的形状的相似性，没有利用任意两条链的交点个数<=2 的性质，作者在原文中提出，存在能在 O(log(n+m)) 时间内解决链求交问题的算法。如果能够实现这种链求交算法，将是性能改进的一个方向。

7. 总结

算法的原理总的来说理解起来不是太难，是我们所熟悉的平面扫描算法。实验前期，我们按照原文中的算法描述一步步实现。实现过程虽然比较顺利，但也碰到了一些困难。首先，是链的大小关系的比较。在处理起点事件时，可以根据链的起点相对于其它链的关系来确定两条链的关系。但是，当两条链相交，处理交点事件时，需要将两条链在链树中的位置进行交换。交换后，如果要进行查询操作，此时，应该根据什么来定义两条链的大小关系呢，按照起点是显然不行的。其次是状态结构－－AVL 树的实现。由于算法需要链树能够实现查询与某一条链相邻的两条链，因此，如果采用一般的 AVL 树，不能做到这一点（考虑到上面第一点所说的，要进行查询，就需要比较链的大小，对链进行定位，但出现链交换后，链的定位难以实现）。为此，我们对 AVL 树进行改造，为每个节点增加了一个指向父节点的指针，方便查询。此外，在删除节点时，也不能采用递归的删除方法（链定位无法实现），我们采用的是先直接删除，然后自底向上的调整，确保树的平衡。这样我们就不能直接利用已有的 AVL 树实现代码，只能自己写。

批注 [Jordan1]: 特殊要求

我们自己调试时，测试点数目很少，只有几个到几十个，这种情况下基本不会出现退化情况，因此原文中提出的算法是能够处理的。但是，检查时老师要求输入点为上百个，这个时候很容易就出现退化情况，原文算法就不能正常运行。后期我们的主要工作就是处理这些退化情况。这时候的处理很麻烦，因为点数少的时候基本不出现退化情况，而点数多了以后就根本没办法进行单步跟踪。有些退化情况我们预先能够想到，就先处理了，但对没想到的，我们调试的时候，只能让程序运行，出现 debug assertion failed 时检查异常状态，然后判断是出现什么情况，然后再修改代码。实际上根据程序异常来判断是出现了哪种退化情况是比较困难的，因为这时候链树很大，而且每个节点保存的是一条链，不像数字那么直观。

就这样修修补补，最后我们发现，原文提出的对所有三种类型的事件点的处理方法，在出现退化情况时，都不能正确处理。也就是说现在我们使用的对事件点的处理方法都是我们自己提出的。而平面扫描算法最核心的部分无疑是对事件点的处理，因此相当于我们对原来的算法进行了很大的改造。但是这种改造也并不完备，因为现在我们的程序在点数很多的情况下，运行还是可能会出错，这也说明肯定还有我们没有发现的退化情况存在。

8. 参考文献

[1] Gill Barequet, Matthew Dickerson and Petru Pau, Translating a convex polygon to contain a maximum number of points, Computational Geometry 8(1997) 167-179

[2] 刘卫东，沈官林，严蔚敏，数据结构 C++语言描述，清华大学出版社，1999 年

[3] D.Kirkpatrick and J.Snoeyink, Tentative prune-and-search for computing fixed-points with applications to geometric computing, in: Proc.9th ACM Symp.Comput.Geom.(1993) 133-142.

[4] M. Dickerson and D. Scharstein, Optimal placement of convex polygons to maximize point containment*, Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms,* 114-121, 1996 [5]

[5] J. O'Rourke, *Computational Geometry in C* (Cambridge, 1994).

[6] D.P.Dobkin ,D.G.Krikapatrick , Fast Detection of Polyhedral Intersection , Theoret Comput.Sci.,27:241-253,1983