简单多边形凸包算法实现及演示

总结报告

孙培文 2016213590

肖慎柯 2015311444

朱　翼 2016213629

2017-6-2

目录

一、 选题背景

简单多边形凸包问题，如题名，即为求解给定简单多边形的凸包的问题。因为简单多边形在给定的同时具有边的顺序特征，所以作为一个凸包问题的特例，可以比点集凸包问题的$O(n\log n)$具有更好的时间界，为$O(n)$。

在简单多边形凸包算法的发展过程中，第一个提出该算法的作者是 Sklansky，他提出了后续被许多算法使用的简明的 left-turn 测试，但很可惜，他先后提出的两种算法都是错误的。第一个正确的算法是 McCallum & Avis 于 1979 年提出的，在使用 left-turn 测试的基础上进行了改进。截止目前为止被公认的最优算法是 Melkman 于 1987 年提出的算法，通过维护一个双向队列来优雅的进行判定处理。

本项目在开题报告的 11 种方法之中，选取了共计 9 种方法进行实现，并分别对这些方法在实现的基础上提供演示，方便大家更好的对其进行理解。11 个方法中的 Lee 的算法和 Graham & Yao 的算法如出一辙，因此只实现了流程更为清晰的 Graham & Yao 的算法。而 ElGindy, Avis & Toussaint 的算法更多的是理论上的意义，因为他们将该问题使用求解内部可见多边形的问题的方法，转化为了可以利用 Sklansky 1972 解决的外部可见多边形，所以也没有给予实现。这两个方法和其他所有方法如下列表所示：

1. Sklansky 1972 [1]　孙培文
2. McCallum & Avis 1979 [3]　肖慎柯
3. *Lee 1983[4]　朱翼
4. Graham & Yao 1983 [5]　朱翼
5. *ElGindy, Avis & Toussaint 1983 [6]　孙培文
6. Ghosh & Shyamasundar 1983 [7]　肖慎柯
7. Bhattacharya & ElGindy 1984 [8]　肖慎柯
8. Preparata & Shamos 1985 [9]　朱翼
9. Orlowski 1985 [10]　孙培文
10. Shin & Woo 1986 [11]　孙培文
11. Melkman 1987 [12]　朱翼

同时本项目还使用了 TwinklingStar 开源的简单多边形随机生成算法[15]。

二、 开发环境

使用 VS2015+QT5.8 在 windows10 平台上进行开发。

三、 数据结构

本项目使用的数据结构非常简单，主要使用以下的 Point 类和 SimplePolygon 类来进行计算，使用 Point 类、Line 类以及 Area 类组合成 Display 类进行演示。

```
class Point
{
public:
    double x;
    double y;
};

class Line
{
public:
    Point a;
    Point b;
    LineType type;
};

class Area
{
public:
    Points points;
};

class Display
{
public:
    Points points;
    Colors pointColors;
    Lines lines;
    Colors lineColors;
    Areas areas;
    Colors areaColors;
    void clearAll();
};

class SimplePolygon
{
public:
    Points points;
    vector<int> convexHull;
};
```

四、 算法原理及实现细节

1. Sklansky 1972

(1) 算法原理

该方法是第一个线性时间的简单多边形凸包的算法。它使用了一个非常简单的算法过程来构造凸包。假设简单多边形的点顺序以逆时针方向给出，具体流程如下所示：

1) 每次处理a,b,c共三个点，初始化时凸包队列中即为简单多边形的起始三个点。
2) 若(a,b,c)构成一个 left turn，即$\overrightarrow{bc}$在$\overrightarrow{ab}$的左侧，那么则将c也加入暂存的凸包队列中，并对$(b,c,c.next)$作为下一次计算的对象。
3) 若(a,b,c)构成一个 right turn，那么将点b从凸包队列中删除，并将$(a.pre,a,c)$作为下一次计算的对象。

但很不幸，这个看起来非常简洁的算法是错误的。因为当删除掉的点破坏了原来的简单多边形互相不交叉的条件后，仅仅一个 left-turn 测试并不能保证当前三个点中中间的那个处在当前线段链的凸出的位置上。结合如图 1-1 所示的反例可以看的更清楚：

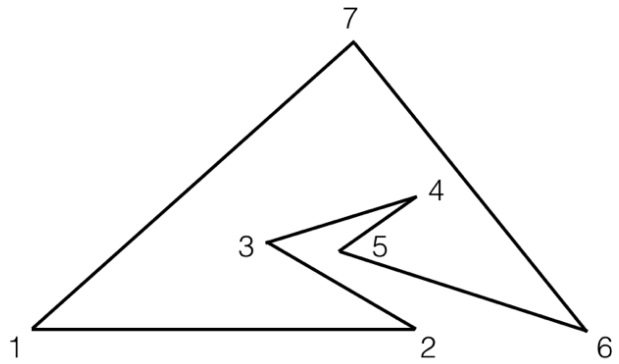

图 1-1

可以看到，在对于(2,3,4)删除 3 后，在处理(1,2,4),(2,4,5)和(4,5,6)时都是 left-turn，所以并没有删除点。最后得到的凸包为(1,2,4,5,6,7)显然是错误的。

虽然这个算法是错误的，但是因为该算法的过程简单高效，具有很强的参考价值，后续有很多方法都使用了 left-turn 测试这一判定方法。并且这个 Sklansky 算法被证明对于外部可见的简单多边形，是正确的。[13]

(2) 实现细节

实际实现中，为了方便的进行循环终结的判断，可以选择最左最下点为第一个加入队列的元素。这样可以确保，在一次遍历到该点重新成为队列中最新元素的时候，凸包的计算已

经完成。

2. McCallum & Avis 1979

第一个正确的算法由 McCallum 和 Avis 于 1979 年提出。算法以坐标最大和最小的点（横坐标和纵坐标均可，这里为说明方便取纵坐标，实际的代码里取的是横坐标。记最上面的点为 A，最下面的点为 B）为分界，分别构造凸包的两半部分。以右半部分为例，算法以 Sklansky 1972 算法为基础，从 B 往 A 按逆时针方向依次处理各顶点。

与 Sklansky 1972 算法的不同之处在于，算法还维护一个栈 T，包含构成当前凸包（当前凸包即当前处理完的顶点加上 A 所生成的凸包）右半部分（为方便，为其命名为支撑链，注意当前处理完的点都在支撑链的左边）的额外顶点。以图 1-1 为例，算法从 1 开始，处理完 4 后，支撑链为 1-2-4-7，而 1,2,4 已被算法加入凸包点集（凸包点集的维护几乎与 Sklansky 1972 算法一致，差异之后叙述），因此 T 包含唯一的点 7。

记 T 的栈顶元素是 t，凸包点集的最后两个点依次是 u,v。还是以图 1-1 为例，处理完 4 后，t,u,v 分别是 7,2,4。直线 vt 和直线 uv 将平面分成四个区域（初始时凸包点集只有一个点，可以认为直线 uv 为水平线），从直线 vt 和直线 uv 的左边区域（即 5 所在的区域）按逆时针方向依次命名为 R_1,R_2,R_3,R_4。注意到若下一个点在 R_1,R_4，即为 Sklansky 1972 算法中 left-turn 的情况，此算法中维护凸包点集的方法与 Sklansky 1972 算法唯一的区别便在于忽略 R_1 中的点，例子中 5 就这样被排除了。当然，根据下一个点在哪个区域算法还需要维护 T，即维护支撑链。具体而言：

1) 如果下一个点在 R_1，直接忽略；
2) 如果下一个点在 R_2，将凸包点集中的最后一个点加入 T；
3) 如果下一个点在 R_3,R_4，回溯 T，使 T 中点从顶往下构成 left-turn。

如上所述，有了 T 的信息，这个算法就能排除掉 Sklansky 1972 算法中错误加进去的顶点，从而得到正确的结果。

3. *Lee 1983

Lee 1983 和 Graham & Yao 1983 两种方法都注意到简单多边形的凸包的一些边和原多边形构成了面积不为 0 的封闭区域。通过合理高效地维护这些区域，就在线性时间内得出最后的凸包。

Lee 算法中的术语定义如下：若凸包上的相邻两点u和v在原简单多边形上不是同一条边的两个端点，称u和v在原多边形之间的一系列边和线段uv构成的封闭区域为 lobe，线段uv称为 handle，lobe 中除去 handle 的部分称为该 lobe 的 body，示意图见图 3-1。

根据凸包的性质，y 坐标最小的点（若有多个点 y 坐标最小，则选取它们之间最右侧的点）一定在凸包上，令其为v_0。预处理输入简单多边形点，使其以v_0为开始，按照逆时针排列。算法利用一个栈维护当前已经确定在凸包上的点。当判断完最后一个点v_{n-1}是否在凸包上后，算法结束。

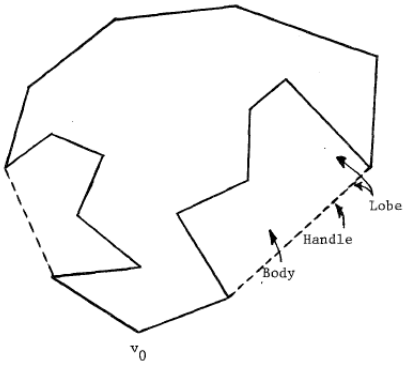

图 3-1

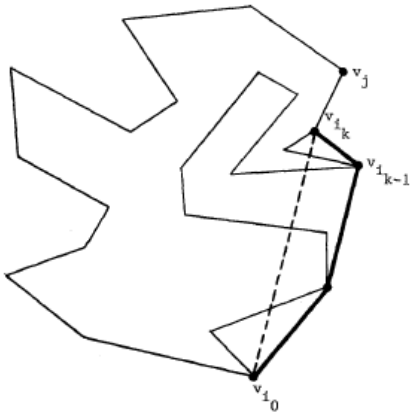

图 3-2

初始时将v_0与v_1加入栈。通常情况下，定义v_{i_0}，v_{i_1}，…，v_{i_k}依次相连构成的多边形为C，点v_{i_k}在栈顶，定义$l(C)$是有向直线$\overrightarrow{v_{i_{k-1}}v_{i_k}}$，设当前处理点为$v_j$。凸包 C 和$v_j$、原多边形的关系见图 3-2 所示。当$v_{i_{k-1}}$，$v_{i_k}$构成了 lobe，定义 C 处在状态 1，简记为$C_1$；否则为$C_2$。

根据v_j所处的不同位置，有不同的处理方法：

1．v_j不在 $l(C)$的严格左侧，见图 3-3。显然，此时v_{i_k}和栈中它下面的一系列点不可能是凸包上的点。所以，我们对栈执行 pop 操作，直到栈中元素为v_0或v_j严格位于$\overrightarrow{v_{i_{j-1}}v_{i_j}}$的左侧。

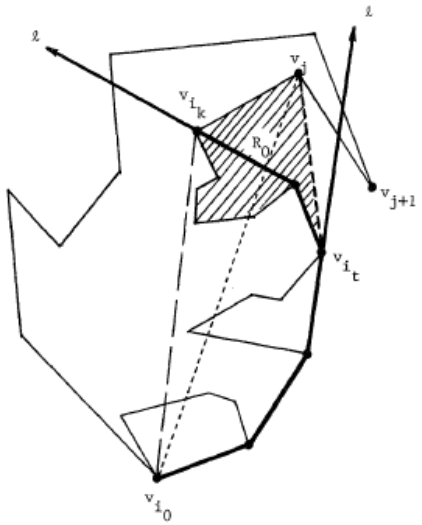

图 3-3

2.若v_j在$l(C)$的严格左侧，根据 C 目前的状态分别做处理。

2.1 C属于类型 1，检查v_j是否在由$v_{i_{k-1}}$，v_{i_k}构成的 lobe 中（定义为R_0，见图 3-4a）。若v_j在R_0中，那么v_{i_k}和v_j一定不是凸包上的点。事实上，在v_j之后在R_0中所有的点均不在凸包中。设v_s是我们找到的第一个不在R_0中的点，此时v_s一定在$l(C)$的严格右侧，执行和 1 类似的操作。若v_j不在R_0中，可以将C视作类型 2。

2.2 C属于类型 2，此时根据v_j的位置做进一步的讨论。

2.2.1 v_j不在$\overrightarrow{v_{i_k}v_0}$的严格右侧，设此时由两个半平面确定的区域为$R_1$（见图 3-4b）。与 2.1 类似，我们遍历$v_j$之后的点，直到点$v_s$不在$R_1$中，将$v_s$作为下次迭代的待处理点继续执行。

2.2.2 v_j在$\overrightarrow{v_{i_k}v_0}$的严格右侧，设此时由两个半平面确定的区域为$R_2$（见图 3-4c）。根据凸包的性质，此时$v_j$在当前凸包上将其加入栈。注意，若之前处理流程为 2.2.1，此时边$\overrightarrow{v_jv_{i_k}}$不是原多边形上的点。

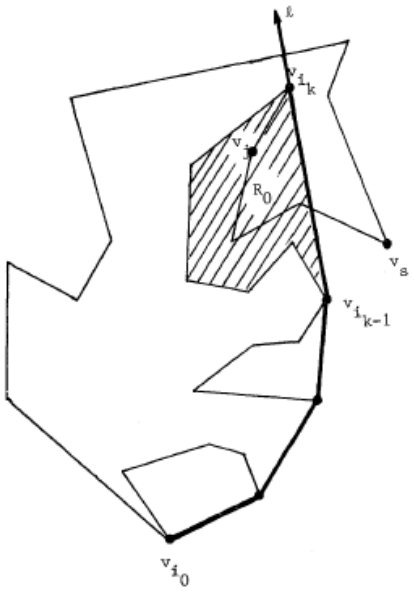

图 3-4a

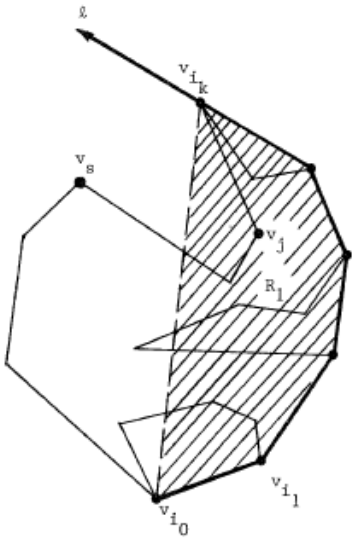

图 3-4b

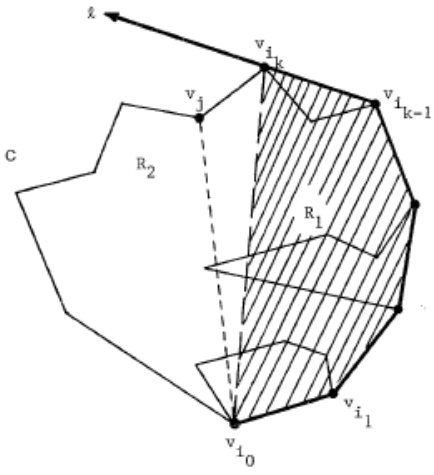

图 3-4c

总的来说，栈中点将下一个点可能出现的位置划分为 3 个区域：1)前一个 lobe。2)栈中点构成的凸包。3)其他区域。当下一个点出现在区域 1)和 2)，则其一定不可能在凸包上，算法遍历其后续点，直到一个点在这个区域之外。当在区域 3)时，根据其在 l(C)的左侧或是右侧，决定是否先删去栈中的一些点，最后将其加入栈。

4. Graham & Yao 1983

(1) 算法原理

Graham 和 Yao 也使用栈存储中间结果。与 Lee 的方法稍有不同，他们的算法先找到确定在凸包上的两个点，这两个点确定的直线将凸包划分为两部分，算法依次确定这两半凸包拼合在一起为最终结果。下面以计算一侧凸包为例阐述算法流程。

设输入的点列为v_1,v_2, …,v_n，按照顺时针排列。根据凸包的性质，最左侧和最右侧（最高和最低）一定在凸包上，记这两个点为v_1,v_m（若v_1不满足要求，在$O(n)$的时间内也能处理多边形使满足要求），则输入点列可划分为v_m，v_1，…，v_{m-1}和v_1,v_m，…,v_n。下面考虑对v_m，v_1，…，v_{m-1}进行处理。

第一步初始化栈中元素。将v_m和v_1压入栈，持续读入后续点x直到x在$\overrightarrow{v_m v_1}$的右侧，并将x压入栈。

主流程依次读入点x，设栈顶元素为 u，次栈顶元素为 v，栈底元素为 w ，u 在原多边形中的前驱点为 y。根据 x 所在不同空间执行不同操作：

1. 若x在$\overrightarrow{vu}$的严格左侧，删去栈顶元素直到x在$\overrightarrow{vu}$的严格左侧，将x压入栈。
2. 若x不在$\overrightarrow{vu}$的严格左侧且x在$\overrightarrow{yu}$的严格左侧，说明x在由$\overrightarrow{vu}$确定的 lobe 中，一定不在凸包上。持续读入后续点并赋值给x，直到x在$\overrightarrow{vu}$的严格左侧（不在 lobe 中），并执行情形 1 的操作流程。
3. 若x不在$\overrightarrow{vu}$的严格左侧且x不在$\overrightarrow{yu}$的严格左侧, 说明x在由当前栈中点确定的凸包中，一定不在凸包上。持续读入后续点并赋值给x，直到x在$\overrightarrow{uw}$的严格左侧（不在由栈中点确定的凸包中），并执行情形 1 的后续操作。

算法的思路与 Lee 类似，不同之处在于预先确定在凸包上的两个点，对下一个凸包候选点在空间中的位置做了更有效的划分，因此算法流程相对 Lee 的更简洁。

(2) 实现细节

先确定输入简单多边形顶点方向，保证为顺时针排列。分别求解上凸包和下凸包，最后合并为完整凸包。在执行算法流程时，可以利用 goto 语句使得对点所在区域判断更方便。

5. *ElGindy, Avis & Toussaint 1983

这个方法和 Sklansky 1982 算法具有相同的思路，即先将给定的简单多边形通过线性时间的算法转化为外部可见多边形，然后在此基础上使用简洁的 Sklansky 1972 算法。他们的转化方法建立在他们之前发表的在线性时间内计算简单多边形内部某一点的可见多边形（Visibility Polygon）的方法[14]基础上。

具体处理过程为，考虑一般情况，先找到当前多边形的四个可以确定的极点（即最高、最低、最左、最右），然后将由这四个点确定的矩形和简单多边形围成的四个区域分别进行处理。以图 4-1 中的区域R_1为例：

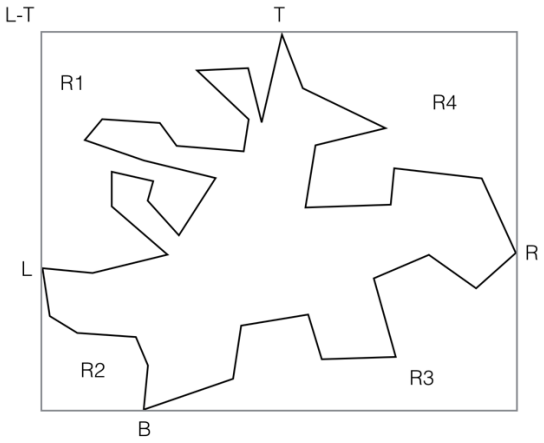

图 4-1

此时，区域R_1可以看作由边界$(\mathrm{L}-\mathrm{T}, \mathrm{T}, \mathrm{chain}(\mathrm{T}, \ldots, \mathrm{L}), \mathrm{L})$组成的一个简单多边形。然后我们计算点$\mathrm{L}-\mathrm{T}$在简单多边形$R_1$中的可见多边形，得到多边形$V_{R_1}$，如图 4-2 所示。

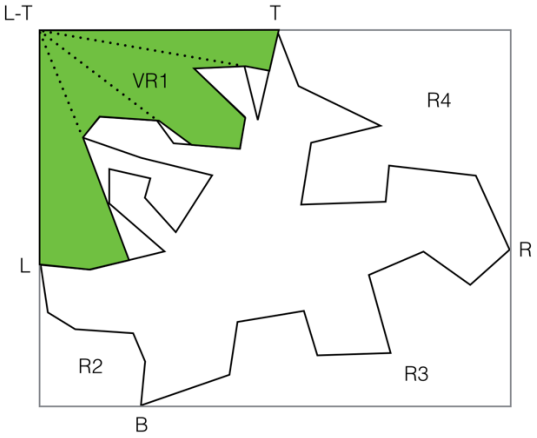

图 4-2

可以证明，得到的可见多边形是一个简单多边形。将该简单多边形的$chain(L, \ldots, T)$提取出，即可代替原简单多边形的$chain(T, \ldots, L)$。进一步还可以证明，这个改变不会改动原

简单多边形的凸包。此时即可采用 Sklansky 1972 的算法对预处理后的简单多边形进行计算得到凸包。

原论文使用的线性时间计算简单多边形内部可见多边形的方法是第一个解决该问题的线性时间方法，但是非常繁琐，一定程度上超过了简单多边形凸包计算本身的复杂程度，更多具有的是理论上的意义。考虑到本项目的复杂程度，最后并没有实现。

6. Ghosh & Shyamasundar 1983

(1) 算法原理

Ghosh 和 Shyamasundar 于 1983 年提出的算法也是以坐标最大和最小的点为分界，分别构造凸包的两半部分。以下半部分为例，参照图 5-1：

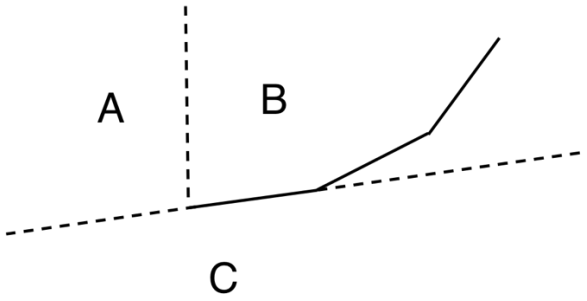

图 5-1

从最右边的点开始开始处理，实线代表目前加入构成最后凸包的点集（以栈的形式维护）形成的链。若下一个点在 A 区域，直接将其加入构成最后凸包的点集；若下一个点在 B 区域，忽略之；若下一个点在 C 区域，则会形成一个 right-turn，处理方式和 Sklansky 1982 算法处理 right-turn 的方式相同。

可惜的是这个算法是错误的，B 里的点可能会出现在最终的凸包上。考虑如图 5-2 反例：

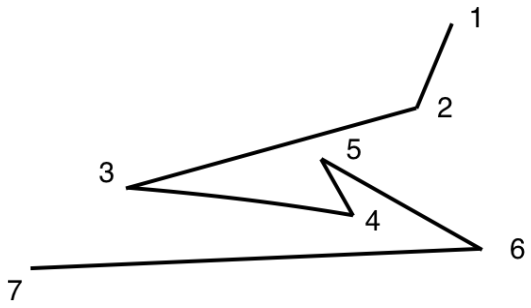

图 5-2

当处理完 5 后，构成最后凸包的点集为 1-2-4-5，接下来的 6 在 B 区域，因此 6 会被忽略，但 6 显然是在最后的凸包里的。

(2) 实现细节

在此算法里始终是 A 区域在左，B 区域在右（考虑上半部分时则反过来）。由于当前凸包的最后一条边可能反过来朝右（比如反例中的 4-5），实现中容易弄混。

7. Bhattacharya & ElGindy 1984

(1) 算法原理

这个算法和 Ghosh & Shyamasundar 1983 算法类似，不同之处在于在考虑下一个点属于哪个区域时考虑了原多边形的结构，如图 6-1：

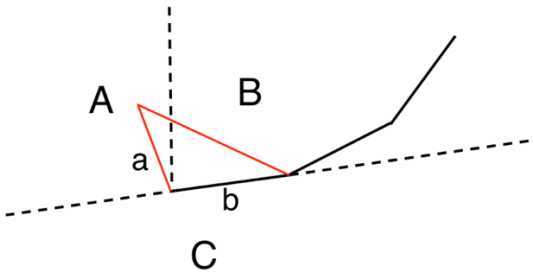

图 6-1

原多边形如红线所绘，如果下一个点在红线圈起来的范围内（只需对直线 a 和直线 b 做 to-left 判断），则忽略之，然后一直忽略接下去的点直到下一个点穿过线段 b 进入区域 C（不可能从红线穿过）。可以证明这个算法是正确的。在 Ghosh & Shyamasundar 1983 算法的反例中，这个算法通过这条额外的规则避免了将 5 加入构成最后凸包的点集，从而避免了 6 被忽略掉。

值得一提的是，原算法并不是以坐标最大和最小的点为分界，而是以坐标最小的点和它关于直线 l 的 antipodal point 为分界（相应地，分割区域 A 和 B 的也由平行于坐标轴的直线变为平行于 l 的直线），其中直线 l 是坐标最小的点（假设是纵坐标）和另一个顶点的连线，使得其与 x 轴的夹角最小。

(2) 实现细节

此算法在考虑点穿过红线框起来的区域时，要尤其注意临界情况。比如线段 b 上有一点 c，其前后线段分别为 x,y，区域里的点通过 x,y 穿出这个区域，如果处理得不好，程序容易把 x,y 判断成与 b 没有交点而认为依旧没有穿出红线框起来的区域。

8. Preparata & Shamos 1985

(1) 算法原理

Preparata 和 Shamos 的算法思想和 Lee 类似，通过对平面划分维护 lobe。略有不同的是，他们的算法能计算上（下）凸包。

下面以计算上凸包为例，说明算法流程。初始时先求取当前多边形的最左侧x_{min}和最右侧x_{max}，这两个点一定在凸包上。将一个虚设的点（在x_{min}下方，且横坐标与x_{min}一致）并按照逆时针的顺序从x_{min}开始依次访问其他点。与 Lee 类似，他们利用一个栈维护当前确定在凸包上的点，并根据这些点将平面划分为四部分（如图 7-1）：

1. 上一个 lobe 内部：由栈顶两元素u和t确定的有向直线$\overrightarrow{ut}$的右侧；设原多边形上t的前驱为s，在$\overrightarrow{st}$的左侧。
2. 当前凸包的上方：有向直线$\overrightarrow{ut}$的左侧。
3. 有向直线$\overrightarrow{tx_{max}}$的左侧
4. 有向直线$\overrightarrow{tx_{max}}$的右侧

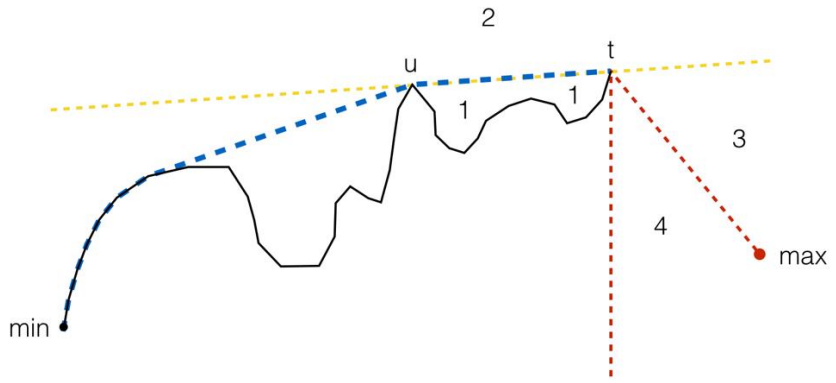

图 7-1

根据所在的不同区域操作如下：

1. 若新加入的点在 1 号区域，忽略并处理下一个点
2. 若新加入的点在 2 号区域，删除栈顶元素直到点新加入的点在栈顶两个元素的右侧
3. 若新加入的点在 3 号区域，将其加入栈中
4. 若新加入的点在 4 号区域，忽略并处理下一个点

(2) 实现细节

设最左点的坐标为(x, y)，虚设点的坐标为$(x + tolerance, y - 20)$，对于通过屏幕输入的数据，能保证虚设点到最右点的连边一定在上凸包上。虚设点可以用-1 特殊标记压入栈。类似地，若最右点的坐标为(x, y)，虚设点的坐标为$(x - tolerance, y + 20)$。

9. Orlowski 1985

(1) 算法原理

Orlowski 的方法在 Sklansky 1972 的方法的基础上，添加了 Lee 1983 方法中的 pocket-test(lobe-test)的思想，从而避免了 Sklansky 1972 方法的错误。具体方法流程如下：

使用一个栈来储存所有的点，记栈中最顶部的三个点为i,j,k，其中k是最顶部的点。假设原简单多边形方向为顺时针，记原多边形中x_{min}的点为点 1，额外增加一个点 0 使得$x_0 = x_1$，并使的(0,1,2)三个点构成一个 right-turn。最初令栈中存储[0,1,2,3]，并进行下列操作：

1) 如果k为点 1，则凸包构造完成；否则转 2)。
2) 如果(i,j,k)不构成一个 left-turn，转 3)；否则删除点j并转 2)。
3) 如果$(j-1,j,k)$不构成一个 left-turn，压点$k+1$进栈并转 1)；否则压点$k+1$进栈，删除点j并转 4)。
4) 如果(i,j,k)不构成一个 left-turn，压点$k+1$进栈，删除点j并转 4)；否则删除点j并转 2)。

注意在每一个步骤中，i,j,k始终代表栈中倒数第三、倒数第二以及最后的三个点。

观察 Sklansky 1972 无法通过的反例，见图 8-1。

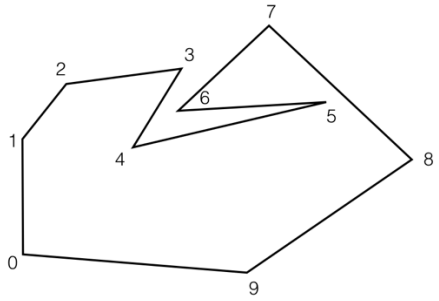

图 8-1

可以看到，在删除了点 4 后，会继续添加 6，然后通过 3)删除 5，再通过步骤 4)删除 6，通过步骤 2)删除 3，最终得到正确的凸包。而 Sklansky 1972 算法在删除点 4 后将不会再删除任何点。

(2) 实现细节

实际实现中，步骤 4)转 2)可以修改为转 1)而并不会对算法产生影响，从而可以只是用循环和判断来实现算法，而不用使用 goto。另外添加初始点 0 时，可以令$x_0 = x_1, y_0 = y_1 - tolerance$，从而避免在点 0、1 中间出现新点而产生问题。

10.Shin & Woo 1986

(1) 算法原理

他们的方法其实与 Preparata & Shamos 1985 非常相似，只不过在其基础上略去了一些计算。以 Preparata & Shamos 1985 的示例为例，如图 7-1。Shin & Woo 仅仅在此基础上省略了其中对区域 4 的考虑，并未对其他计算进行修改。即将根据区域进行的操作修改为：

1. 若新加入的点在 1 号区域，忽略并处理下一个点
2. 若新加入的点在 2 号区域，删除栈顶元素直到点新加入的点在栈顶两个元素的右侧
3. 若新加入的点在 3 号或四号区域，将其加入栈中

而在进行如此修改后，虽然实际上计算仍然没有问题，但是效率在某些情况下会极大的下降。如图 9-1，在计算该简单多边形的凸包时，Lee 1983 以及 Preparata & Shamos 1985 的方法都能快速的忽略螺旋的点，而该方法却低效的一直计算到螺旋最内部的点后再回旋计算出来。

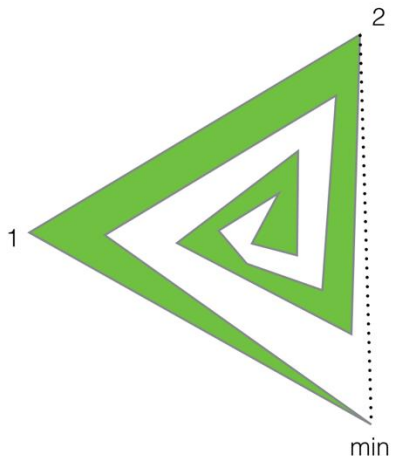

图 9-1

(2) 实现细节

在实际的实现中，本方法和 Preparata & Shamos 1985 的终结条件并不相同。Preparata & Shamos 1985 方法中，因为排除掉了区域 4 的情况，所以只需要遍历搜索到下一个点为min或者max即可。但是本方法中，必须要遍历到min或者max最终压入栈中，才可以停止，否则对于区域 4 中的点的处理将会出现问题。

11.Melkman 1987

(1) 算法原理

Melkman 的算法被认为是计算简单多边形凸包的最好算法。与其他算法不同，它是一个在线算法。前述算法大多要求从一个确定在凸包上点开始顺序处理剩余节点，对输入多边形的顺序也有要求。除了算法输入要求低，其流程也简介明了，整篇文章只占用了两页纸。与 Lee、Graham & Yao 的算法类似，该算法将整个空间划分成了若干区域，根据下一个节点出现在不同区域进行处理。

Melkman 用一个双端队列（deque）来存储当前可能在凸包上的点。先考虑双端队列的初始状态。算法初始时任意取三个点，组成初始凸包，如图 10-1 所示。三个点能构成顺时针顺序，在双端队列中的存储顺序为：（队尾）3-1-2-3（队头）。注意到如果我们从队尾往后读，依次为 3-1-2，是顺时针序；如果我们从队头开始读，依次为 3-2-1，是逆时针序。这也是我们希望双端队列拥有的性质：从队尾往队头读构成顺时针序；从队头往队尾读构成逆时针序。

接着 3 个点的情况，考虑 3 号点之后的节点v。v可能出现在图 10-1 中红、蓝、绿和黄四个区域中，不同情况下的处理方法如下：

(1) 在黄色区域中：忽略当前v，遍历之后的节点，直到它出现在黄色区域外，并根据情况进行后续处理。

(2) 在红色区域中：说明v在有向直线 23 的左侧，删除队头元素直到v在队头两个元素构成的有向直线的右侧。

(3) 在绿色区域中：说明v在有向直线 13 的右侧，删除队尾元素直到v在队尾两个元素的左侧。

(4) 在蓝色区域中：v同时具有红色区域和绿色区域的特征，依次执行两类操作。

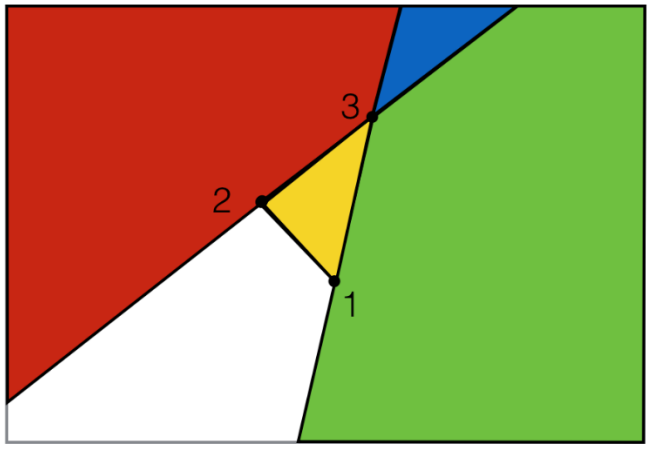

图 10-1

对于一般情况，队头两个元素构成的直线和队尾两个元素构成的直线将平面划分成了四个区域，处理原则与三个点的情形类似。

一言以敝之，Melkman 发现了凸包上的点遵循的基本性质：顺时针相邻的三个点a,b,c满足c一定在有向直线$\overrightarrow{ab}$的右侧；逆时针相邻的三个点a,b,c满足c一定在有向直线$\overrightarrow{ab}$的左侧。

同时 Melkman 利用了双端队列高效维护了这项性质。

(2) 实现细节

可以利用 C++标准库 deque 实现双端队列，使代码更具有鲁棒性。

12.简单多边形随机生成算法

本项目在实际生成随机简单多边形的过程中，实际使用了空间分割法。因为该算法不是本项目的主要内容，所以，在实现中使用了 TwinklingStar 在 GitHub 上公开的源码。具体来源参见参考文献。

五、 实验总结

本次实验中的算法之间没有对比，实际上对于每一个输入的简单多边形，完全可以调用 9 个算法分别计算后，提供各个算法的性能比较情况。

另外在输入点时，并没有提供准确的根据坐标输入的功能，因此在构造测试用例时，仍然需要打开保存的文件单独进行修改。

最终的测试仍然不成体系，只是依靠小组成员头脑风暴式的思考，可能存在遗漏情况。这方面可以进行很大的进一步改善。

参考文献

[1] Sklansky J., "Measuring Concavity on a Rectangular Mosaic", IEEE Transactions on Computing, 21, p1355, 1972.
[2] Sklansky J., "Finding the Convex Hull of a Simple Polygon", Pattern Recognition Letters, v.1 #2, p.79, Dec.1982.
[3] McCallum D., Avis D., "A linear Algorithm for Finding the Convex Hull of a Simple Polygon", Information Processing Letters, v.9 #5, p.201, 1979.
[4] Lee D.T., "On Finding the Convex Hull of a Simple Polygon", International Journal of Computers and Information Sciences, v.12 #2, p.87, 1983.
[5] Graham R.L., Yao F.F., "Finding the Convex Hull of a Simple Polygon", Journal of Algorithms 4, p.324, 1983.
[6] ElGindy H., Avis D., Toussaint G., "Applications of a Two-Dimensional Hidden-Line Algorithm to Other Geometric Problems", Computing, vol.31, p191, 1983.
[7] Ghosh S.K., Shyamasundar R.K., "A linear time algorithm for computing the Convex Hull of a Simple Polygon", Pattern Recognition, v.16 #6, p.587, 1983.
[8] Bhattacharya B.K., ElGindy H., "A New Linear Convex Hull Algorithm for Simple Polygons", IEEE Transactions on Information Theory, vol. IT-30, #1, Jan.1984.
[9] Preparata F.P., Shamos M.I., Computational Geometry, ch.4, Springer-Verlag, New York, 1985.
[10] Orlowski M., "A Convex Hull Algorithm for Planar Simple Polygons", Pattern Recognition, v.18 #5, p.361, 1985.
[11] Shin S.Y., Woo T.C., "Finding the Convex Hull of a Simple Polygon in Linear Time", Pattern Recognition, v.19 #6, p.453, 1986.
[12] Melkman A., "On-line Construction of the Convex Hull of a Simple Polygon", Information Processing Letters 25, p.11, 1987.
[13] Toussaint G.T., Avis D., "On convex hull algorithm for polygons and its application to triangulation problems", Pattern Recognition, vol.15, #1, p.23, 1982.
[14] ElGindy H., Avis D., "A Linear Algorithm for Computing the Visibility Polygon from a Point", Journal of Algorithms, v.2, p186, 1981.
[15] 多边形生成算法, http://www.twinklingstar.cn/2016/2987/6-2-polygon-generation/