成果报告: 最大空元凸子集

张亚娴 软件学院 2016213643　　刘聪颖 软件学院 2016213637
宋正阳 交叉信息院 2016211701

2017 年 6 月 4 日

1 简介

计算几何中的一类重要问题便是寻找给定点集中具有某种性质的子集，而空元凸子集关注的是凸性和空元性。具体来说，给定点集 S 的空元凸子集满足以下性质：该集合中的点全部落在其整体构成的凸包上，即凸包内部为空。而这里的“最大”考虑的是顶点个数，而非凸包面积。我们希望找到满足上述空凸子集中所包含的顶点个数最多的那个。

2 相关工作

Chavatal 等 [CK80] 给出了关于最大凸子集（即不考虑空元性质）时间复杂度为 $O(n^3)$ 的做法。而 Avis 等 [AR85] 在此基础上进行了简单的修改给出了最大空元凸子集的时间复杂度为 $O(n^3)$，空间复杂度为 $O(n^2)$ 的解法。通过使用几何排列（Arrangement）的思想，Herbert 等人 [EG89] 把空间复杂度由 $O(n^2)$ 降至 $O(n)$。而当前已知的最优做法出自 David 等 [DEO90]，其时间复杂度正比于集合的空三角形的个数，该值在点集在单位正方形内随机取点时期望值为 $O(n^2)$ [BF87]；空间复杂度正比于有着相同最左顶点的空三角形的个数最大值。

3 算法描述

该算法 [DEO90] 计算给定点集中的最大凸多边形时大致分为三个部分：

1. 对于 S 的每个顶点 p，舍弃掉其左边的点，将剩余点按照 p 极角排序，从而得到以 p 为核的星形多边形 P_p。使用几何排列中的对偶 [CGL83, EOS86] 该部分时间复杂度可由 $O(n^2 \log n)$ 降至 $O(n^2)$。

2. 计算 P_p 的可见图 VG_p，该可见图包含 P_p 原图中的边，但不包含以 p 为顶点的边。该部分存在时间复杂度为可见图输出大小的线性的算法 [Her87]。

3. 计算 VG_p 中的最长凸链 C_p，对于任意一个凸链将其首尾与 p 相连便可得到一个空凸多边形。该部分时间复杂度为输入可见图的大小的线性。

然后我们对于所有的点 p，保留长度最长的 C_p 及其对应的核，最终将其首尾相连便得到了包含点数最多的空元凸多边形。注意到该算法的前两步与上文中的 [AR85] 是相同的，但采用了更好的算法降低了时间复杂度。

4 具体实现

代码与文档均已被托管至 GitHub [1]。整个工程大致分为三个模块：

- 交互模块：构建用户交互窗口，存储算法处理的点坐标（包括鼠标点击输入、从文件中读入、随机生成），响应按钮点击事件。
- 演示模块：算法过程的动画演示，包括计算结果的显示、所有点的演示、用户选择的单点演示。各个步骤的演示中可以选择显示某些细节。
- 处理模块：按照上文中的描述分为三个部分：星形生成、可见图生成、凸链生成。

接下来我们详细说明算法包含的三个部分。

4.1 极角排序

本部分算法的输入输出如下：

- 输入：二维点集的向量
- 输出：对于该点集中的每一个点，以其为核、其余点按极角大小逆时针排序的向量

我们首先对屏幕上所有点按横坐标进行排序，时间复杂度 $O(n \log n)$。

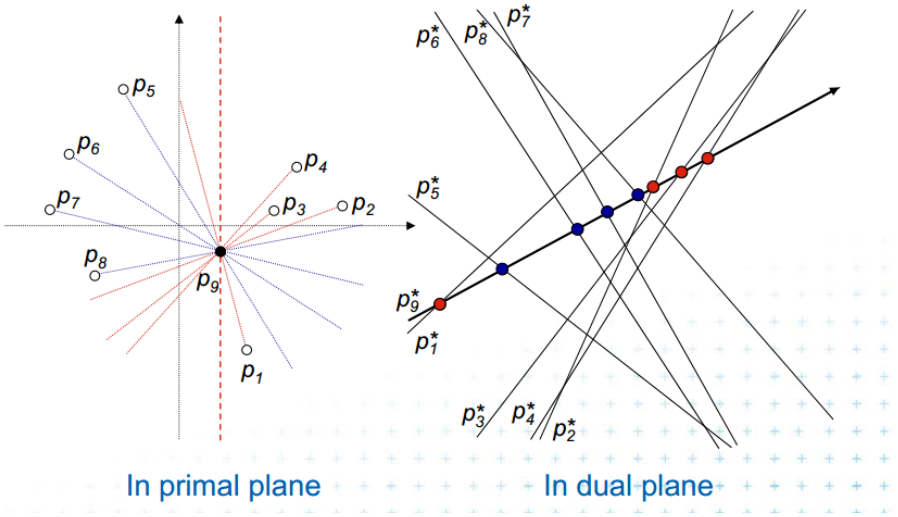
Naïve 的方法是接下来对于每一个点，对其右侧点（横坐标大于当前点）按极角进行时间复杂度为 $O(n \log n)$ 的归并排序。因此总的时复杂度为 $O(n \log n + n \cdot n \log n) = O(n^2 \log n)$.

Arrangement 的方法基于下述几个观察：将原图中坐标为 (a, b) 的点 p 映射为对偶图中的直线 $p^* : y = ax - b$，则原图中的对于某个点与其右侧点

[1] https://github.com/songzy12/LECP

的极角大小关系便映射为对偶图中对于某条直线与其下方直线的交点横坐标的大小关系。具体来说，原图中逆时针极角越小，在对偶图中交点横坐标越小。这样我们只需要从右向左遍历原图中的点，在对偶图中加入相应的直线。每当加入一条直线时，我们使用 $O(n)$ 的时间得到它与对偶图中已有直线从左到右的交点，该顺序即为原图中右侧点极角排序的顺序。总的时间复杂度为 $O(n \log n + n \cdot n) = O(n^2)$.

Figure 1: 原图极角序与对偶图交点序对应关系



如图 4.1（图片来自教材）所示以 p_9 为极点对其它所有点进行极角排序，点 p_9 的对偶直线与其他点对应的对偶直线从左到右依次相交，相交的顺序即为以 p_9 为极点对应的极角排序，而原图中 p_9 右侧点对应对偶图中下侧直线的交点（红色点）。以左图中 p_2 和 p_4 为例，p_9p_2 斜率小于 p_9p_4，因此对偶图中 p_9 与 p_2 交点横坐标小于 p_9 与 p_4 交点横坐标（由于转换关系为 $y = ax - b$）。

4.2 可视图生成

本部分算法的输入输出如下：

- 输入：核及其右侧顶点按极角序给出的向量数组
- 输出：对于右侧的每个顶点，计算出其可视图中的入边和出边

右侧点已经按极角序逆时针排列好，我们便按照逆时针的顺序处理每一个点 p，每次处理 p 的结果是生成在最终的可视图中 p 的所有入边。具体做

法是我们在每个 p 处维护一个队列 Q_p，其中存储了 p 点能看到且在之后的处理中还可以被其他点看到的点。算法如图 4.2 （来自论文）所示。

Figure 2: 可视图生成

```
procedure VISIBILITY;
  for i := 1 to N − 1 do Q_i := ∅ end;
  for i := 1 to N − 2 do PROCEED(i, i + 1) end.

procedure PROCEED(i, j);
  while Q_i ≠ ∅ and TURN(FRONT(Q_i)i, ij) = left do
    PROCEED(FRONT(Q_i), j);
    DEQUEUE(Q_i);
  end;
  ADD(ij);
  ENQUEUE(i, Q_j).
```

注意上图中的 while 处，由于 Q_i 是按照相对点 i 逆时针的顺序存储的，所以在我们遇到第一个 TURN 为 right 的时候，便可以放心地跳出这个循环，因为剩下的所有 TURN 也必定为 right. 而在 Q_i 之前弹出的那些点此时已经被保存在了 Q_j 之中。这样我们保证了每个时刻，所有点的中记录的 Q 的数据之和不会超过当前已处理过的点的个数。从而整体的时间复杂度为 $O(|VG|)$，其中 $|VG|$ 为可见图的大小。

另一方面，这样处理结束后每个点处存储的入边和出边都是按照逆时针排序的，而这一点在之后的处理中也是有用的。

4.3 最长凸链生成

本部分算法的输入输出如下：

- 输入：核及带有入边出边信息的点集，且入边出边均按逆时针排列
- 输出：该有向图中的最长凸链

注意到顶点集依然是按照相对于核按极角大小排序的，每条边带有一个属性 L，其代表以该边为起始边的最长凸链长度。我们这次按照顺时针的方向处理每一个顶点，每一个顶点处理结束后其所有的入边的 L 均被设置为正确值，并记录下该 L 值所对应的上一条边。最终的最长凸链便由带有最大 L 的边及其所有祖先构成。

处理某个顶点 p 时，我们按逆时针处理它的每一条入边。并在处理每一条入边时，我们按照逆时针处理相应的出边。对于每一条入边和出边，我们计算他们的夹角是否为凹角：如果是凹角，我们便根据该出边的 L_o 和入边

当前的 L_i 决定是否将 L_i 更新为 L_o+1；否则 L_i 的更新已完成。这是因为出边已排好序，一旦遇到某个拐角为凸，我们便可以放心地决定剩下的所有拐角也为凸。这样每个顶点处的每条入边和出边只会被考虑一次，从而整体时间复杂度为 $O(|VG|)$. 具体算法细节如图 4.3 （来自论文）所示。

Figure 3: 最长凸链计算

```
procedure MAXCHAIN;
  for i := N-1 downto 1 do TREAT(p_i) end.

procedure TREAT(p);
  Let i_1, ..., i_imax be the incoming edges of p, and let
  o_1, ..., o_omax be the outgoing edges of p, both ordered counterclockwise.
  l := omax; m := 0;
  for j := imax downto 1 do
    L_ij := m+1;
    while l>0 and TURN(i_j, o_l) = left do
      if L_ol > m then
        m := L_ol;
        L_ij := m+1
      end;
      l := l-1
    end
  end.
```

5 边界处理

原论文中假设输入的各点均不相同且无三点共线，我们考虑了在这些情况下的处理方式。

5.1 重复点输入

对于重复点的输入有两种解决方案：

1. 在每个顶点处增加一个域记录它的重度 w_p，算法的前两步保持不变，第三步中的 $L_i = L_o + 1$ 改为 $L_i = L_o + w_p$。

2. 对于每个新的输入点，检查当前点集中是否存在坐标与之相同的点。如果存在，则不进行添加。

由于时间关系我们选择了后一种处理方式。

5.2 横坐标相同情况

对于横坐标相同的情况会出现问题是因为两点生成的对偶图中的两条直线平行，从而不会产生交点，进一步地该点便会在最终的处理结果中遗漏掉。

处理方式是设想我们将画板轻轻顺时针旋转一个小角度，则所有横坐标相同的点便变得不再相同，上方的点会成为右边的点。同时如果上方有多个点横坐标相同时，考虑我们希望生成的星形多边形，将距离该点越近的点视为极角越大的点。

在实现中我们在最初的排序时将横坐标相同但纵坐标不同的点按纵坐标小的排在前面，然后处理时在最后一步中检查与当前点横坐标相同且纵坐标大的点，将其按反序压至极角排序的最后（即纵坐标越大越靠前）。

5.3 多点共线情况

如果用户输入的点集中存在多点共线的情况，极角排序时我们需要考虑的要更复杂一些。想象我们希望生成的星形多边形，该多边形的内部需要为空。于是我们做如下规定：我们记录处理过程中遇到的最后一个多点共线情况，如果该多点共线恰好出于是我们需要处理的点末尾（如上面横坐标相同便是该种情况的特例），那么我们认为离核越近的极角越大；否则，我们认为离核越远极角越大。

6 性能分析

使用对偶图的直线排布进行排序是极其耗费内存空间的，想象我们在 n 个输入点规模的情况下，我们需要存储对偶图中的 n 条直线的所有交点和半边。其中交点的可能个数有

$$1 + 2 + \ldots + (n-1) = n(n-1)/2$$

半边的可能个数有

$$n + 2 \cdot n(n-1)/2 = n^2$$

而 sizeof 给出来的我们实现中的 Vertex 和 HalfEdge 大小均在 ~ 200 这个量级。考虑一个 2G 内存的机器，粗略的计算可以得出理论上所能支持的输入点个数在 ~ 1000 这个量级。

故使用对偶图排序虽然其理论的时间复杂度在渐进意义上较低，但在实际的应用中我们并不能使它充分地发挥威力。

7 总结与展望

我们通过该选题作业，进一步领略了计算几何中算法的精妙。同时了解到在将理论上的算法真正实现成为可供演示的软件时总会遇到设想之外的问题。

在共同完成作业的过程中也体会到团队协作的重要性，在遇到困难时也得到了来自老师的及时点拨。总之我们感谢这段愉快的学习时光，及出现在这段时光里的老师和同学们。

参考文献

[AR85] David Avis and David Rappaport. Computing the largest empty convex subset of a set of points. In *Proceedings of the first annual symposium on Computational geometry*, pages 161–167. ACM, 1985.

[BF87] Imre Bárány and Zoltán Füredi. Empty simplices in euclidean space. *Canad. Math. Bull*, 30(4):436–445, 1987.

[CGL83] Bernard Chazelle, Leo J Guibas, and Der-Tsai Lee. The power of geometric duality. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 217–225. IEEE, 1983.

[CK80] V Chvatal and G Klincsek. Finding largest convex subsets. In *Proc. 11th SE Conf. on Combin., Graph Theory and Comp*, 1980.

[DEO90] David P Dobkin, Herbert Edelsbrunner, and Mark H Overmars. Searching for empty convex polygons. *Algorithmica*, 5(1-4):561–571, 1990.

[EG89] Herbert Edelsbrunner and Leonidas J Guibas. Topologically sweeping an arrangement. *Journal of Computer and System Sciences*, 38(1):165–194, 1989.

[EOS86] Herbert Edelsbrunner, Joseph O' Rourke, and Raimund Seidel. Constructing arrangements of lines and hyperplanes with applications. *SIAM Journal on Computing*, 15(2):341–363, 1986.

[Her87] John Hershberger. Finding the visibility graph of a simple polygon in time proportional to its size. In *Proceedings of the third annual symposium on Computational geometry*, pages 11–20. ACM, 1987.