编号：_____

高级数据结构与算法设计

作业 2：快包算法

学　　院：计算机与通信技术学院

学　　号：G20188751

姓　　名：潘秋实

指 导 老 师：皇甫伟

得　　分：_____

2019 年 03 月 30 日

目录

摘要

分治是算法设计中的一种非常重要的设计思想。学习分治对于算法提升很有帮助，本文使用 Python 语言实现了快速凸包算法（简称快包算法）。在实现任意数量点的快包算法的基础上，本文将快包算法运行过程做了可视化处理，并做了快包算法运行时间关于点数的实验，最后绘制出曲线图。本文观察到，快包算法的时间复杂度是符合$O(n\log n)$的。

1 引言

快速凸包算法（本文中简称快包算法）是一种典型的分治算法，它的思想和快速排序有些类似，都是将一整个“大”问题，通过划分子集的方式，变成许多个“小”问题来解决，体现了经典的分治思想。

求凸包的算法有很多，而快包算法是一种通过分治思想设计的算法。在好的情况下，快包算法的时间复杂度可以达到$O(n\log n)$，而只有在极端情况下才会达到$O(n^2)$，说明快包算法的执行效率是极高的。所以本文即对快包算法展开研究，主要工作如下：

1）使用 Python 语言实现了二维平面上的快包算法。在实现的过程中，本文使用简单的几何知识，简化了快包算法的编程，准确率依然 100%，执行效率不受影响。

2）针对本文中编程实现的快包算法，本文的实验借助 OpenCV 工具，将算法的执行过程图形化展示出来。每次演示脚本执行，都会在二维平面上生成指定数量的点，求凸包的过程会通过实时生成图片展示，最后保存成 avi 格式的视频文件。

3）使用不同长度的数组，分别多次测试快包算法。数组中的元素都是随机生成的，针对每种长度的数组多次运行后，求平均值即可得到快包算法对当前长度执行的大致时间。最后求得数组长度与运行时间的关系，使用 Matplotlib 绘制折线图，并使用 xlwt 保存成 Excel 表格。

2 快包算法理论

快包算法假设二维平面上有 N 个点，它们是按照 x 轴坐标升序排列的，如果 x 轴坐标相同，则按照 y 轴坐标升序排列。这样一来，第一个点p_1（最左边的点）和最后一个点p_n（最右边的点）就一定是属于凸包的顶点，并且它们的连线p_1p_n可以将这 N 个点分为上凸包和下凸包。

而对于上凸包而言，取一个离

p_1p_n最远的点p_{max}，则p_{max}也一定是凸包上的顶点。再给p_{max}和p_1、p_n分别脸上一条线，在这两条线外侧（$p_{max}p_1$的左上方，$p_{max}p_n$的右上方），分别距离两条线最远的两个点，也一定是凸包的顶点。以此类推，一直找到所有的连线，使所有的点都在连线的内侧。对于下凸包也是同理，可以一直找位于两点连线外侧的点。上凸包和下凸包找到的所有的线，除去所有被找到上方或下方存在点的线，即是所求凸包。

在此过程中，如果要计算一个点是否在直线的外侧，可以将直线的左端点$p_i(x_i,\mathrm{y}_i)$、右端点$p_j(x_\mathrm{j},\mathrm{y}_\mathrm{j})$和目标点$p_k(x_k,\mathrm{y}_k)$代入公式(1)求得。

$$\begin{vmatrix} x_i & \mathrm{y}_i & 1 \\ x_\mathrm{j} & \mathrm{y}_\mathrm{j} & 1 \\ x_k & \mathrm{y}_k & 1 \end{vmatrix} = x_i\mathrm{y}_\mathrm{j} + x_k\mathrm{y}_i + x_\mathrm{j}\mathrm{y}_k - x_i\mathrm{y}_k - x_\mathrm{j}\mathrm{y}_i - x_k\mathrm{y}_\mathrm{j} \qquad (1)$$

当p_k位于直线p_ip_j左侧的时候，公式(1)的值为正，而p_k位于直线p_ip_j右侧的时候，公式(1)的值为负。上凸包的每条线，只需要求左上方或右上方是否存在点，而下凸包的线只需要求左下方或右下方是否存在点。即确定了目标的左右方位，也可以确定上下方位，因此可以直接使用公式(1)判断。

以上就是快包算法的基本思路，我在实验之前，详细研究了这一套理论，不禁感叹确实是一种非常聪明的做法。不过本文在接下来的实验中，还是在这个基础上，运用几何知识做了一定的改进，顺利实现了快包算法，并且获得了不错的表现效果。

3 实现思路

3.1 算法简化

对于标准的快包算法而言，其核心的判别式就是第 2 章中的公式(1)，所以这就决定了其寻找的目标点是在直线外侧的点，同时包含上下和左右两个属性。而我认为，这样一次判定需要执行 6 次乘法和 5 次加减法，计算量有点大；并且从几何意义上来讲，寻找外侧的点很抽象，同一个目标点，可能会被属于不同“分包”的多条直线判断是否在外侧，这无疑是不必要的。

对此，我认为可以将寻找直线“外侧”的点，转化为寻找直线上下的点：例如对于上凸包中的一条直线，只判断其左右端点的 x 轴坐标范围内，上方有没有其他点，如果有的话，计算每个点离直线的纵轴截距，截距最大的一个点一定是凸包的顶点。

上边说的比较简洁，具体来说，就是对于一条左端点坐标是$p_i(x_i,\mathrm{y}_i)$、右端点坐标是$p_j(x_\mathrm{j},\mathrm{y}_\mathrm{j})$的直线，如果要判断目标点$p_k(x_k,\mathrm{y}_k)$是不是在其上方，首先要确定$x_i \le x_k \le x_\mathrm{j}$，否则说明$x_k$已经超出 x 轴坐标范围，即目标点不

可能在直线p_ip_j的上方，从而不需要再计算后判断。而p_k的 x 轴坐标符合要求的话，再通过x_i、x_j分别与x_k的差，确定两个比例系数，分别乘以y_i、y_j。这样可以达到直线p_ip_j在 x 轴坐标等于x_k的点的 y 轴坐标d。d的计算公式如公式(2)所示，$d > 0$就代表目标点在直线的上方，d 的绝对值就是目标点距离直线的 y 轴截距。

$$d = y_k - \left(\frac{x_k - x_i}{x_j - x_i} \cdot y_j + \frac{x_j - x_k}{x_j - x_i} \cdot y_i\right) \quad (2)$$

对于下凸包的一条边而言，只要找 x 轴坐标在相应范围内，且$d > 0$的目标点就可以了。

本文提出的这种简化方法，大致进行了两步简化：首先可以看到在判别的时候，进行了横轴坐标的判断，从而缩小了每条边找外侧点的查找范围，如果 x 轴坐标越界，根本不需要带入公式(1)计算，从而减少了计算量；其次，对于每一个符合 x 轴要求的点，带入公式(2)计算，只需要执行 5 次减法、2 次除法和 2 次乘法，明显要比公式 1 的计算量小，同时也可以得到点线位置关系、距离这两个参数。所以，这种简化从理论上来说是有效的。

3.2 算法流程

整个算法包括四个方法，是逐级调用的关系，本章即介绍这四个方法，其中最后一个就是本文要实现的快包算法。

首先介绍的是比较方法，该方法如第 3.2 节中介绍的，输入一条直线和一个被比较的点，可以得知点与直线的位置关系及距离，如算法 1 所示。在后续的实现中，每次做比较的时候，就调用这个方法。

算法 1 Compare 算法

输入：直线左端点 left、右端点 right、目标点 x
输出：位置关系、纵轴截距
1: **if** x 轴坐标越界
2: **return** 越界异常
3: 计算d
4: **if** $d > 0$
5: **return** 上方， 截距
6: **if** $d < 0$
7: **return** 下方， 截距

第二个是为一条线确认上方或下方是否存在点的方法，如算法 2 所示。这个方法思路比较简单，需要为它传入直线的左右端点、候选点集合和目标状态，它就会调用算法 1，为每个候选点与直线作比较。如果没有找到合适的点，就返回“无”的状态。如果有合适的点，就返回截距最大的点和合适的点的集合。因为截距最大的点必然是凸包的一个顶点，会与原直线的两个端点连成两条新的边；而返回的集合，就可以作为查找这两条新边的边外是否有点时的候选点集合。

算法 2 FindNode 算法

输入：直线左端点 left、右端点 right、候选点集合 Candidate、目标状态 sta
输出：新顶点 x，合适点集合 CanNode
1: while 候选点集合不为空
2: 使用算法 1 比较点与直线

```
3:     if 位置关系=目标状态
4:       当前点加入合适点集合
5:       记录截距
6:   if 合适点的集合 == null
7:     return 无，空集
8:   else
9:     find 截距最大的点 x
10:      return 顶点 x，合适点集合 CanNode
```

第三个是处理一条边的方法，其作用是将一条线范围内所有的凸包边找出来，如算法 3 所示。例如上凸包，给该方法输入一条直线，和候选点集合，它可以通过调用算法 2，找出候选点中最远的一个点，为凸包增加一个顶点。而它本身的两个端点和新的顶点，又会产生两条直线，它就会递归调用自身，一只求出凸包所有顶点为止。

算法 3 ProcessLine 算法

输入：直线左端点 left、右端点 right、候选点集 NodeList，目标状态 sta
输出：凸包边界集合 Edge

```
1:  把[Left, Right]加入 Edge
2:   使用算法 2 寻找新顶点 x 和合适点的集合 CanNode
3:  if 找到新顶点
4:    NodeList = CanNode
5:    使用算法 3 自身处理[Left, x]
6:    使用算法 3 自身处理[x, Right]
7:  return Edge
```

最后即是完整的快包算法，其思想很简单，如算法 4 所示：使用最左边的点和最右边的点连一条直线，在分别更改 sta 参数，用算法 3 处理上凸包和下凸包，就可以得到凸包所有的边。

算法 4 QuickHull 算法

输入：点的集合 Nodes
输出：凸包边界集合 Edge

```
1:  找到最左和最右的点 Lift、Right
2:  使用算法 3 处理[Left, Right]的上凸包，得到 Edge1
3:  使用算法 3 处理[Left, Right]的下凸包，得到 Edge2
4:  Edge=Edge1+Edge2
5:  return Edge
```

本文概括得比较笼统，因篇幅限制无法展现细节，所以具体的实现细节，可以查看工程代码。

3.3 可视化处理

本文实验中所构造的工程，依然是对需要调用的方法，进行了了封装，也包括一些专门用来可视化的方法，本节会简单介绍一下。

首先是快包算法过程的可视化方法，我采用 OpenCV 实时生成画面的方案，即每次边有变动的时候，都会把点和当前记录的所有边绘制到一张画面上，并且显示出来产生实时视频的效果。执行快包算法演示脚本，程序会随机生成指定数量的点，并且运行快包算法，在屏幕上显示实时处理进程；程序执行完，会在 output 文件夹中生成一个记录本次过程的 avi 视频文件，方便验证查看。

其次是快包算法性能的测试脚本，我使用了 OpenCV、Matplotlib、xlwt 和 prettytable 来做可视化处理。其中每一次测试结束后，会使用 OpenCV 将当前测试生成的点与凸包绘制出来；多次试验结束后，程序会记录点数与

处理时间的关系，使用 Matplotlib 将其绘制成折线图，在命令行中使用 prettytable 打印出来，最后使用 xlwt 生成一个 Excel 记录文件。

4 实验

本文中，一共做了两个实验：运行工程根目录的 demo_quickhull.py，进行快包算法演示实验；运行工程根目录的 demo_test_perform_time.py，进行快包算法性能测试实验。在本文的试验过程中，使用的平台是搭载 i7 8700k 的 Ubuntu 16.04，参考运行主频为 4.5GHz。在其他平台上运行时，本章的结果可以作为参考。

4.1 快包算法演示

在程序运行前，手动设置点的数量后，程序运行开始会自动生成相应数量、随机坐标的点。一开始，只有 p_1p_n 一条直线（最左到最右），随着程序运行，会慢慢增加边和减少边，如图 4-1 所示。

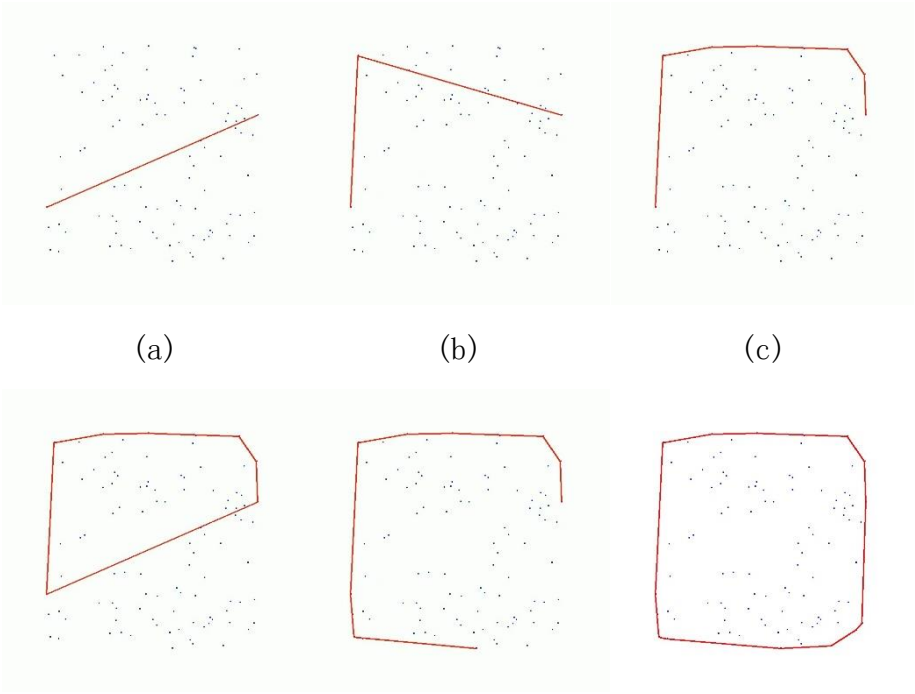

图 4-1 快包算法运行过程图

图 4-1 是 100 个点的凸包求解过程，其中(a)(b)(c)是上凸包的求解过程中的几张图，(d)(e)(f)是下凸包求解过程中的几张图。完整的过程可以从 output 文件夹中的 QuickHull.avi 中查看，求解凸包的结果则保存为 output 文件夹中的 result.jpg，即图 4-1(f)。

4.2 快包算法性能测试

对于快包算法的性能，本文的实验从 10 个点，每次点数乘以 2，一直测试到 327680 个点。对于每种数量的点，程序会随机生成 50 组点，使用快包算法计算凸包，计算 50 次运行时间的平均值，即可估算出快包算法对于该种长度的运行时间。

经过实验，最终得到的点的数量 N 与算法运行时间的对应关系如图 4-2 所示，其具体数值如表 4-1。

表 4-1 快包算法消耗时间 T 关于点数 N 的平均值

Length	Average Time(ms)	Length	Average Time(ms)
10	0.100374	2560	24.76583
20	0.200558	5120	48.42863
40	0.501299	10240	97.45898
80	0.802231	20480	201.4353
160	1.604366	40960	391.9419
320	3.208447	81920	778.9703
640	5.915689	163840	1548.315
1280	12.73398	327680	3109.565

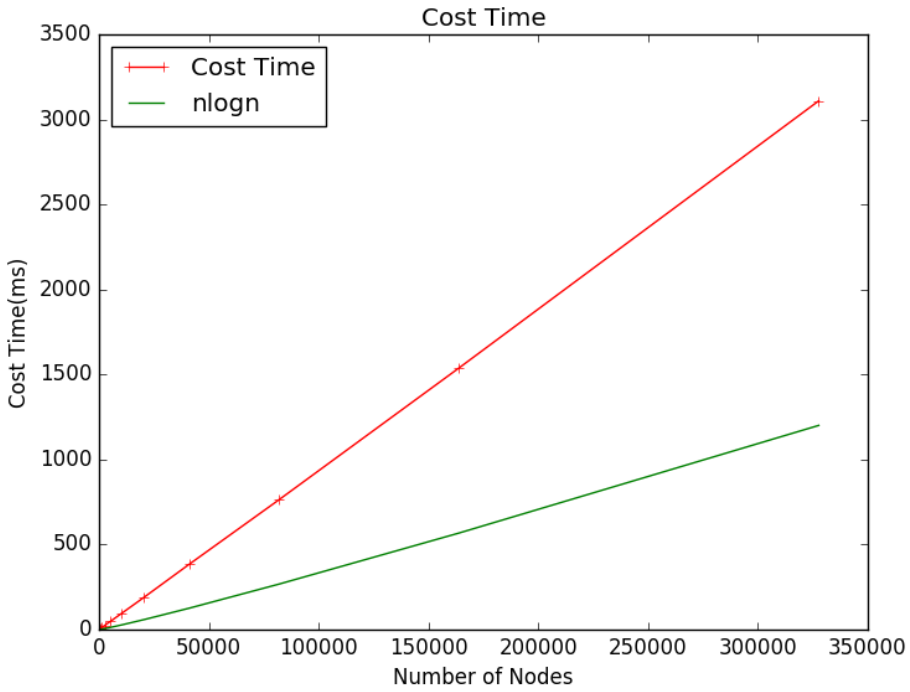

图 4-2 快包算法时间 T 关于点数 N 的变化趋势

值得一提的是，在演示实验中，点的横纵坐标都从 0 到 500 之间生成，而性能实验因为点数较多，0 到 500 的

范围内必然发生重叠，所以性能实验中点的横纵坐标都是从 0 到 10000 随机取值。在图 4-2 中，红色的线代表快包算法运行的时间，是以毫秒为单位的；绿色的线是$O(n\log n)$参考线，是直接使用公式计算得到的。所以由图 4-2 可以看出，快包算法运行的时间复杂度是符合$O(n\log n)$，两者是线性关系，所以快包算法是一种不错的算法。

4.3 验证优化的有效性

在我的实验工程中，lib 文件夹中的 quickhull.py 是我封装快包算法的文件。文件中的前两个同名的方法，分别对应了优化后的比较方法和原比较方法，程序运行时必须注释掉其中的一个，否则会报错。

我依然是使用 4.2 节中的脚本，在比较方法不同的情况下测试快包算法，数组长度从 10 到 327680，每种长度测 50 次，得到的结果如表 4-2 所示。其中 My method 使用的是 3.1 节中的公式(2)作为比较方法，而 Origina method 使用的是第二章中的公式(1)作为比较方法。

表 4-2 优化算法与原方法在不同数组长度上的运行时间对比

Length	My method	Original method	Length	My method	Original method
10	0.100374222	0.15039444	2560	24.76582527	28.72638702
20	0.200557709	0.300693512	5120	48.42863083	55.74817657
40	0.501298904	0.501346588	10240	97.45898247	111.5464687
80	0.802230835	1.002693176	20480	201.4353275	220.2354431
160	1.604366302	1.804804802	40960	391.9418812	442.3255682
320	3.208446503	3.509378433	81920	778.9703369	863.9464378
640	5.915689468	7.319545746	163840	1548.315477	1739.222908
1280	12.73398399	14.28804398	327680	3109.565163	3422.747421

因为是分别进行多次实验得出的平均值，所以该实验没有使用 Matplotlib 作图，而是直接用 excel 文件汇总得到数据。感兴趣的读者可以自行运行实验一下。

从表 4-2 可以看出，本文提出的快包优化方法是确实有效的，可以将算法性能略加提升，而且我的其他实验得出结论：点越散的情况下提升越大，即在随机取值时，设置越大的取值范围，两个算法整体执行效率都会降低，但优化提升的效果却越明显。

5 结论

在本文中，我实现了快速凸包算

法，首先进行了可视化，又根据自己的理解，做了一步优化。面对任何一个算法，思考如何优化是算法爱好者的“通病”，而不应仅仅注重于实现这个算法。本文的优化中，稍微取得了一些效果，并不是很明显，不能为快包算法带来革命性的变革。

本文的优化效果，在点与点之间距离越远，整体分布越松散的情况下，提升效果越好。换句话说，怎样在点十分密集的情况下，提升快包算法性能，仍是一个值得讨论的课题。希望在未来能对快包算法有进一步的研究。