复旦大学计算机科学技术学院

数 据 结 构 课 程 设 计

实 验 报 告

学　　号　17300240035

姓　　名　张 作 柏

课程名称　数 据 结 构

目录

一、 问题描述

本节将介绍课程设计的问题，给出明确的问题定义和数据规模。

（一）问题背景

地理围栏 (Geo-fencing) 是一种形式的 LBS (**L**ocation **B**ased **S**ervices, 基于地理位置的服务)，在现实世界中已经有非常广泛的应用。

当手机进入、离开某个特定地理区域，或在该区域内活动时，手机可以接收自动通知和警告。有了地理围栏技术，位置社交网站就可以帮助用户在进入某一地区时自动登记。

有的时候，我们也需要知道在一个区域内的用户数量，比如统计目前在外滩区域有多少人，这样可以在一定程度上避免拥挤带来的事故；或者统计目前在某个教室有多少人。

如果问题进一步抽象，可以归结为以下两种询问：

- 给定一个点，这个点在哪些多边形内
- 给定一个多边形，有哪些点在这个多边形内

（二）问题定义

本次课程设计中，点和多边形均在二维平面

- **点**：(id, x, y) 其中 x, y 为两个实数，描述了二维平面上的一点，另外还具有正整数 id，代表这个点的标号。
- **多边形**：$(id, n, [(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)])$，其中正整数 id 代表多边形的标号，正整数 n(n>=3)，代表多边形上的点数，另外有 n 个二元组**依次**给出了这 n 个点的实数坐标。应当注意到，多边形总是闭合的。

需要对点和多边形各支持三种操作：增，删，查

1) **增点**：$addPoint\ id\ x\ y$ 增加标号为 id，坐标为 (x,y) 的点
2) **删点**：$deletePoint\ id$ 删除标号为 id 的点

3) **查点**：$queryPoint\ x\ y$ 查询坐标为 (x,y) 的点位于哪些之前插入的多边形内，返回一系列多边形 id

4) **增多边形**：$addPoly\ id\ n\ x_1\ y_1\ x_2\ y_2\ \cdots\ x_n\ y_n$

5) **删多边形**：$deletePoly\ id$

6) **查多边形**：$queryPoly\ n\ x_1\ y_1\ x_2\ y_2\ \cdots\ x_n\ y_n$ 查询给定多边形内有多少之前插入的点

（三）数据规模约定

点和多边形上的点一共的规模在百万级别，算法分析基于此数据规模。

共六个测试点，分别对应不同的问题类型：

	查点	查多边形	混合
静态问题	增多边形之后接查点	增点之后接查多边形	先增后查
动态问题	**顺序随意，有删多边形**	**顺序随意，有删点**	**顺序随意，有删除**

最终测试数据规模如下：

	1	2	3	4	5	6
case_1	0	0	2000	15000	0	0
case_2	150000	0	0	0	0	200
case_3	100000	0	500	5000	0	300
case_4	0	0	2000	15000	1000	0
case_5	150000	20000	0	0	0	200
case_6	100000	10000	300	5000	500	300

点的坐标是 $[-5\times10^6, 5\times10^6]$ 之间的浮点数，点和多边形的标号为 INT 范围内的正整数。

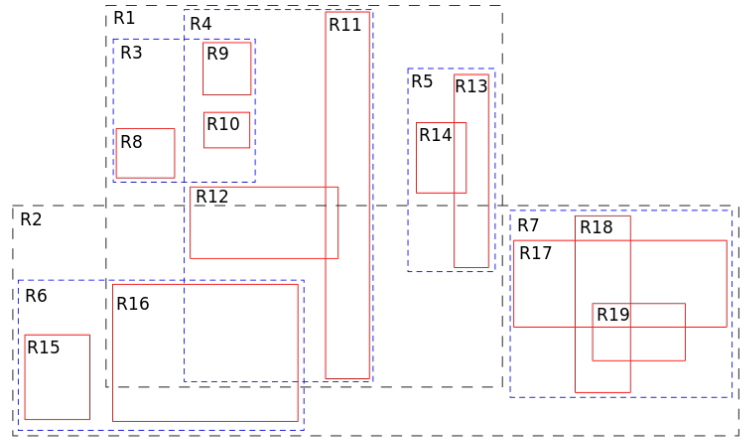
二、 问题分析

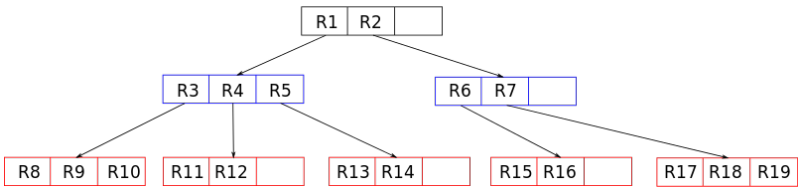
本节给出对问题的具体分析，主要分为两个部分：空间索引数据结构和判断点在多边形内的策略。

（一）Spatial Indexing Data Structure

常用的空间索引数据结构有三种：R 树、k-d 树、四分树，在本问题的探索过程中，我分别在不同算法中使用到了这三种数据结构，以下进行分析。

1. R-tree





R 树是用来做空间数据存储的树状数据结构。例如给地理位置，矩形和多边形这类多维数据创建索引。

R 树的核心思想是聚合距离相近的节点并在树结构的上一层将其表示为这些节点的最小外接矩形，这个最小外接矩形就成为上一层的一个节点。R 树的“R”代表“Rectangle（矩形）”。因为所有节点都在它们的最小外接矩形中，所以跟某个矩形不相交的查询就一定跟这个矩形中的所有节点都不相交。叶子节点上的每个矩形都代表一个对象，节点都是对象的聚合，并且越往上层聚合的对象就越多。也可以把每一层看做是对数据集的近似，叶子节点层是最细粒度的近似，与数据集相似度 100%，越往上层越粗糙。

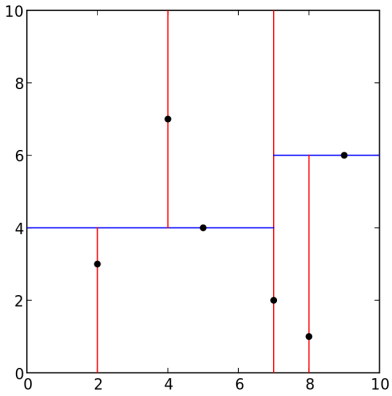
——维基百科

不同于其他两种数据结构，R 树最大的便利之处在于可以处理多边形，将多边形的外界矩形放入 R 树中，查询时可以找到与外界矩形相交的所有多边形。但这也恰恰是它的不

便之处，它规定了只能通过外界矩形来近似多边形，而当遇到不规则的凹多边形时，近似效果往往不佳，会导致许多无用的多边形被选择。

R 树的实现细节比较复杂，因此，我直接使用了 github 上开源的 R 树模板：https://github.com/nushoin/RTree，详见 yourcode 文件夹下的 RTree.h 文件。

2. k-d tree



> k-d 树（k-维树的缩写）是在 k 维欧几里德空间组织点的数据结构。
>
> k-d 树是每个节点都为 k 维点的二叉树。所有非叶子节点可以视作用一个超平面把空间分割成两个半空间。节点左边的子树代表在超平面左边的点，节点右边的子树代表在超平面右边的点。选择超平面的方法如下：每个节点都与 k 维中垂直于超平面的那一维有关。因此，如果选择按照 x 轴划分，所有 x 值小于指定值的节点都会出现在左子树，所有 x 值大于指定值的节点都会出现在右子树。这样，超平面可以用该 x 值来确定，其法线为 x 轴的单位向量。
>
> ——维基百科

k-d 树最大的优势在于其运行效率极高，虽然在二维空间内查找的最坏时间复杂度可达到 $O(\sqrt{n})$，但在实际数据中的表现往往十分出色。

因为之前自己写过 k-d 树，所以就直接在之前的模板上进行了改动。使用自己的模板，也方便于对 k-d 树中具体操作的调整，详见 yourcode 文件夹下的 kd_tree.h 文件。

以下是 kd 树的一些操作的实现细节：

1) **插入**：插入点时，只考虑插入点应属于当前节点的左子树和右子树，类似平衡树的插入操作。若在插入后，树不平衡，则重构。

2) **删除**：删除点时，先找到对应的节点，并标上删除标记，除此之外，不做其他改变。

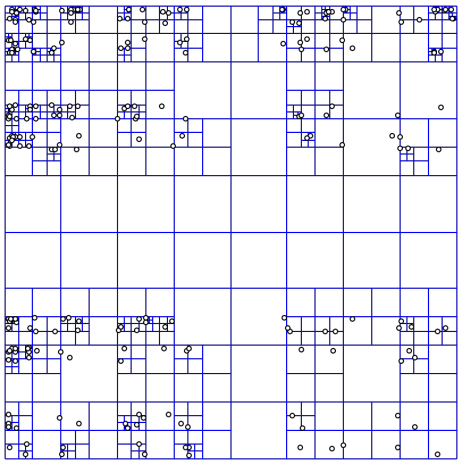
3) **平衡条件**：因为 kd 树本身是不支持旋转操作的，所以无法维持平衡。在插入、删除若干节点后，可能会导致树高严重失衡，此时要通过重构操作，进行平衡的维护。判断的失衡的条件为左子树或右子树大小超过当前子树大小的一半 (可调整)，或子树内删除的节点数占到了一半以上。

4) **重构**：重构一棵子树，只需要先遍历该子树，取出所有未删除的点，并类似线段树的建树方式重构子树。

5) **查询**：查询方式比较灵活，基本思路是若子树中可能是答案的点，就进入子树进行查询，否则就不查。

kd 树维护平衡的思想类似于重量平衡树的维护方式，但是在实际数据集中，因为没有特意构造数据，所以 kd 树的深度基本是平衡的，重构操作和平衡条件的判断反而增加了时间复杂度。

3. Quadtree



> 四叉树是一种树状数据结构，在每一个节点上会有四个子区块。四叉树常应用于二维空间数据的分析与分类。它将数据区分成为四个象限。数据范围可以是方形或矩形或其他任意形状。
>
> ——维基百科

四分树是与 k-d 树十分类似的数据结构，与 k-d 树相比，它的优势在于树的深度有保证，始终不超过 $O(log10^7)$。面对大量的插入删除操作，k-d 树往往会出现不平衡的情况，相比之下，四分树可以避免调整平衡的操作。

然而，在许多实际测试数据中，因为点是随机分布的，k-d 树的效率反而更高。在最终测试的数据中，插入查询操作数量较少，所以更加体现不出四分树的优势，才导致四分树做法的效率远不如 k-d 树。

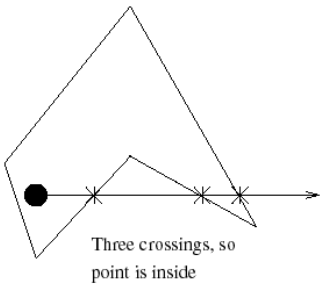
尽管如此，因为四分树对空间划分的模式固定，所以它可以被用来存储矩形或多边形，并且四分树的代码简洁，便于增添功能细节，所以也不失为一种优秀的空间索引数据结构。

（二）Point in Polygon Strategies

对于判断点在多边形内的方法，可以参考 http://erich.realtimerendering.com/ptinpoly/。文中已详细地列出了常用的判断方法，并给出了相应的代码实现和效率对比。

本次课程设计中主要采用了射线法、三角剖分法和格点法三种判断点在多边形内的算法，以下逐一进行介绍。

1. Crossings

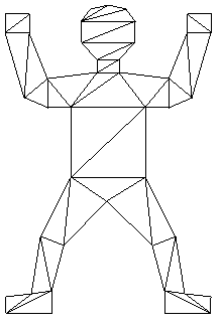


射线法，从询问点随机引出一条射线，计算射线与多边形的边相交的次数。若为奇数，则说明点在多边形内；若为偶数，这说明点在多边形外。若射线通过多边形顶点，则重新进行引射线判断。

显然，射线法的时间复杂度是 $O(n)$，其中 n 表示多边形的边数。射线法的优势在于代码实现简洁，可以通过数据结构加速查找相交线段的过程，而缺点在于对于每一条相交的边都必须要访问，这导致它的最差复杂度与边数同阶。

射线法最大的弊端在于未进行预处理，为具体分析给定多边形的性质，所以每次查询时都需要访问多边形的每一条边。

2. Triangulation



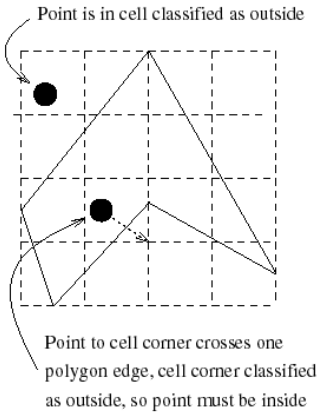
三角剖分法，对于任意一个简单多边形，都可以将其剖分成 $n-2$ 个不重合的三角形。最常见的剖分方法有三种，可以参考 https://github.com/ivanfratric/polypartition。其中分别给出了时间复杂度为 $O(n^3)$，$O(n^2)$ 和 $O(nlogn)$ 的三种算法，基本可适应问题的数据规模。

三角剖分是配合空间索引数据结构使用的算法，通过对多边形进行较为复杂的预处理，将多边形剖分为许多三角形，并通过空间索引数据结构维护三角形。在查询点时，只需要在数据结构中查询包含该点的三角形即可，大大减少了回答询问的复杂度。

然而，算法实际的运行效率还取决于三角剖分的质量，这一点在引文中已说明。$O(n^2)$

的 EC 算法三角剖分质量是可以适应绝大部分情况的，而 $O(nlogn)$ 的 MONO 算法剖分出的三角形质量比较差。这一点会极大地影响三角形在数据结构中的维护和查询。

3. Grid Method



格点法，对多边形所在的区域进行格点剖分，划分成许多小矩形。将小矩形分类，查询点时，直接通过该点所在的矩形的类别，判断该点是否在多边形内。

格点法充分地利用了预处理的过程，剖析了多边形的内在性质，所以在大部分数据集上它的查询复杂度相当快，甚至可以认为是常数级别的，与边数无关。但是预处理的复杂度相当高，可能不适用于多边形边数特别多的情况。

三、 算法设计与分析

在探索 PJ 解法的过程中，共产生了四种算法，本节将分别介绍。

（一）算法 1：R-tree + Brute Force

1. 算法流程

R 树可以用来维护重叠矩形，并支持查询与给定矩形相交的矩形。所以，第一个做法是显然的。

对于操作 1、2、6，加入点和删除点时，将点近似为一个小矩形，放入 R 树中。查询多边形时，先求出多边形的外界矩形，再通过 R 树的 Search 操作查询与该矩形相交的所有点。最后，通过射线法判断点是否在询问多边形内。

对于操作 3、4、5，加入多边形和删除多边形时，用多边形的外界矩形近似多边形，将矩形放入 R 树。查询点时，先将点近似为一个小矩形，在 R 树中查询与该矩形相交的所有矩形。最后，通过射线法判断该点是否在查询到的多边形内。

算法 1 的思路非常显然，因为 R 树维护矩形非常方便，所以用矩形来近似多边形和点，利用 R 树删除不在范围内的点，最后用射线法进行判断。

2. 算法效率分析

时间复杂度分析 R 树的插入、删除操作均为 $O(logn)$，查询操作复杂度与相交的矩形数量有关。射线法的复杂度与多边形边数同阶。所以该算法的时间复杂度瓶颈在于射线法，最差复杂度为 $O($ 查询次数 $\times$ 多边形边数和 $)$，显然是个不可接受的算法。

实际算法效率 算法 1 在随机数据上表现较佳，因为对于随机构造的数据，在查询多边形内的点数量较少，并且 R 树会筛掉许多不在矩形范围内的点，所以用时较少。

算法 1 在测试数据上表现较佳，在开启 O2 优化后，case 1 和 case 4 需要 7s，而对于其他的测试点，3s 内就能运行出结果。之所以这个暴力算法会有比较好的表现，主要原因有：

- 多边形边数较少，均在 150 以内，射线法的时间瓶颈效应被缩小了
- 耗时较大的查询操作较少，耗时较少的插入删除操作非常多
- 数据中的多边形可以很好地用外接矩形近似，因此在空间索引过程中剔除了许多无用点

但是，这个算法显然是错误的，详见下节中的极限数据。

（二）算法 2：k-d tree + Triangulation

1. 算法流程

由算法 1 的分析可知，暴力算法的时间瓶颈主要在于射线法。射线法要求每次都遍历多边形的每一条边，这是非常不划算的。借助三角剖分的预处理，配合空间索引数据结构，我们可以大大地降低判断点在多边形内的复杂度。

算法 2 主要针对操作 1、2、6 进行优化，因为 kd 树比较适合维护点。

对于操作 1、2，直接通过 kd 树的插入和删除操作维护。

对于操作 6，先对多边形进行三角剖分，再把每个三角形放到 kd 树中进行查询。kd 树中每个节点维护一个当前区域的外接矩形，若外接矩形与三角形不相交，则不对该子树进行查询。

2. 算法效率分析

时间复杂度分析 kd 树插入和删除操作的复杂度均为 $O(logn)$，对于每个查询操作，会被拆分为若干个在 kd 树内对三角形的查询。三角形的个数与多边形边数同阶。判断点在三角形内的复杂度是 $O(1)$ 的，且每次查询操作，因为三角形是不重叠的，所以每个答案点只会被访问一次，所以总复杂度是 $O($ 答案点数 $\times$ kd 树查询复杂度 $)$。

实际算法效率 算法 2 在随机数据上表现与算法 1 相近，整体效率略逊于算法 1。主要原因是使用 MONO 算法剖分得到的三角形质量不好，所以会增加在 kd 树中查询的时间。

算法 2 在构造的数据上明显优于算法 1。对于随机够早的 10^5 次插入，一次边数为 10^5 的多边形查询，算法 2 的效率几乎是算法 1 的三倍。因为在多边形边数较多时，射线法的瓶颈效应被放大了，而借助三角剖分和 kd 树可以过滤掉大量的不合法点，所以速率相当快。

算法 2 在测试数据上表现非常好，在开启 O2 优化后，case 2、3、5、6 需要 2s 内就可以运行出结果。因为测试数据中多边形的边数只有 150，所以我使用了 EC 的三角剖分算法，得到质量更好的三角形。实践证明：使用 EC 算法效果远好于 MONO 算法。

算法 2 是唯一能够通过极限数据测试的算法。

（三）算法 3：Quadtree + Triangulation

1. 算法流程

算法 2 只针对了增删点和查询多边形的操作，而多边形的维护仍是一个难题。

针对操作 3、4、5，算法 3 借助四分树对空间的稳定划分，将多边形分成若干个小矩形，放入四分树中维护。

对于操作 4、5，插入删除多边形，可以先将多边形进行三角剖分 (不剖也行)，然后在四分树内完全包含在多边形内的节点上做标记。

对于操作 3，沿四分树路径，访问 $O(logn)$ 个节点，并记录相应的多边形编号。

然而，测试数据中查询操作非常少，插入删除操作反而非常多。为了适应该数据规模，对维护操作进行一定的改进。

因为查询操作非常少，所以我们不妨在查询时在进行多边形的下放，类似线段树的标记维护机制。这样插入和删除操作的复杂度就被大大降低了。

2. 算法效率分析

时间复杂度分析 这个算法的时间复杂度主要取决于每个多边形可被划分成多少个四分树节点对应的正方形。在一维情况下，是 $O(logn)$ 级别的，但是二维情况会更难分析，实

践证明，复杂度相当高！

实际算法效率 算法 3 看上去挺优秀的，但我好像没有正确地实现，导致在测试数据时发生了使用了过量的内存，把我的电脑卡死了。。。因为之后有效率更高的算法，所以就没再仔细实现这个算法。

（四）算法 4：Grid Method + Brute Force

1. 算法流程

格点法是我最后发现的，所以这个算法也是最后才产生的。

格点法对于点在多边形内的判断，是非常非常快的，将一个 $O($ 多边形边数 $)$ 的判断强行变成了 $O(1)$，所以时间上有了质的飞跃。

使用格点法后，即使是暴力枚举每个点和每个多边形，效率都是相当高的。我尝试过使用 R 树索引剔除不在范围内的点，但发现反而不如直接暴力判断。。。

2. 算法效率分析

时间复杂度分析 算法的时间瓶颈主要在于预处理操作，而预处理操作的复杂度在论文中没有给出，所以也不便分析。

对于查询操作，假设每次判断点在多边形内的复杂度是 $O(1)$ 的，那么总复杂度为 $O($ 插入操作数量 $\times$ 查询操作数量 $)$，而插入和查询的次数都可以是百万级的，所以其实也是假算法。

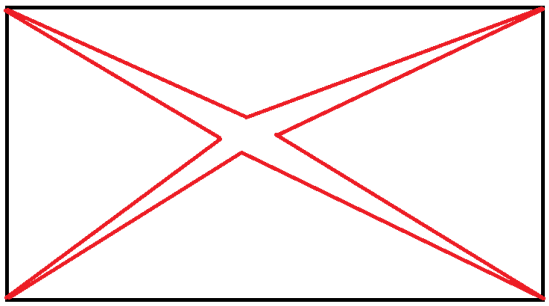
实际算法效率 算法 4 是在测试数据集上运行效率最高的算法，也是我最终提交的版本。因为测试数据中插入和查询操作实在是太少了，实际上这个算法是非常好卡的。只需要增加插入和查询的次数就可以，或者可以减少答案点的个数，这样会对许多无用点进行判断。

四、 构造数据

因为不满于众多假算法水过测试数据的行为，所以特意增加了此节，构造了几种针对性的数据，可以卡掉一部分假算法。

（一）针对外接矩形近似的数据构造

在很多算法中，使用了外接矩形对多边形进行近似，并以此进行索引。然而，这是非常不合理的，因为在大多数多边形中，这往往不能取得很好的近似效果。如下图：

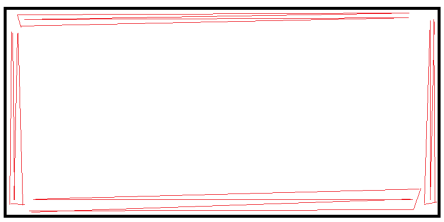


可以看到，在坐标跨度较大时，矩形基本覆盖了整个平面，而多边形只占了矩形的很小一部分。若使用矩形进行近似，那么在索引时几乎所有的点都会被选中，这是非常糟糕的事情，索引数据结构失去了意义！若使用三角剖分算法，就可以很好地筛选出只在多边形内的点。

该多边形的边数可以通过增加锯齿无限增大，而不增大多边形的有效面积，所以也可以同时针对一些不能接受多边形边数增多的算法。

（二）针对射线法的数据构造

基本上，任何基于射线法的算法都可以被卡掉，因为它们将不可避免的访问多边形上与射线相交的每条边。所以只需要构造一个多边形，使得无论点在哪，射线往哪个方向，它都会与多边形的几乎所有边相交。如下图：



在整个平面的四周布满“围墙”，“围墙”是由若干个边数百万级别的宽度无限小的多边形组成。这样，无论射线向哪个方向，都会与一面“围墙”上的所有边相交，这是糟糕的！因为每一次射线的复杂度都是与总边数同阶的，总复杂度会变为 $O($ 射线法次数 $\times$ 多边形总边数 $)$。

五、 讨论与感想

从以上分析可知，只有 k-d 树 + 三角剖分的算法 2 能够很好地适应这些极限数据，同时，算法 2 还可以支持多边形边数为百万级的询问，所以我更愿意将算法 2 列为四个算法中最优秀的算法。

然而，在实际测试数据上，算法 4 运行效率更高。当我发现我的做法被一个暴力算法碾压的时候，我的心里还是很崩溃的。明明之前做了很多努力和尝试，但是最后还是不如一个暴力算法。。。

其实一种更好的询问方法是询问点在多少个多边形内或多边形内有多少个点，这样可以避免访问所有答案节点，也更有利于复杂度的分析。

整体来讲，本次 PJ 主要想让我们了解空间索引数据结构，但因为可以使用开源库，所以对个人的代码能力并没有非常大的提升，整个 PJ 中需要自己写代码的部分也非常少。

在发现格点法后，基本没有正确解法能跑过暴力的格点法，于是就变成了比拼使用搜索引擎的能力。

对于整个测试数据的模式，我更推荐引入 hack 机制，因为每个 case 只有一组测试数据，所以对算法的效率测试并没有足够的说服力。

总体来说，PJ 设计的出发点是很棒的，但是因为种种问题，再加上时间上拖延到了考试周，所以效果不是很好，我在其中的收获也不是很大。

六、 参考文献

http://erich.realtimerendering.com/ptinpoly/

https://github.com/ivanfratric/polypartition

https://github.com/nushoin/RTree