復旦大學

本科生课程论文

课程名称：数据结构　　课程代码：

姓　　名：聂希瑞　　学　　号：16307130133

学　　院：计算机科学技术学院　　专　　业：计算机科学与技术

地理围栏 Project 实验报告

一、情境描述

当我们想知道某个停车场内有多少辆车时，或者我们想知道自己在哪里的时候，我们会使用到地理围栏技术——用虚拟的地理围栏围出一个边界，判断围栏和点的相对位置关系。在大规模的应用上，如何选用合适的数据结构，维护存储地理围栏的数据，达到快速、高效的增删数据、查询数据就显得格外重要。

二、问题分析

我们可以把地理围栏问题简单抽象为判断点和多边形位置关系的算法问题。这又可以划分为六种类型：加点查多边形、加多边形查点、混合查询、加点删点查多边形、加多边形删多边形查点、混合增加删除查询。为了加快查询和删除的效率，有必要建立合适的二维空间索引，以及编写合适的判断点和多边形位置关系的算法。

三、数据结构

在维护空间索引方面，比较合适的数据结构有 R-Tree， KD-Tree 等等。R-Tree 通过维护多边形的外接矩形的包含关系来维持高效索。KD-Tree 则通过维度的不断切换对 n 维空间进行划分。

四、算法分析

这个 project 涉及到的算法部分，除了合适的数据结构之外，主要就是判断点在多边形内外的算法。通过查询资料，我找到了以下算法：

1、 射线法：射线法从点出发沿着 x 轴画一条射线，依次判断该涉嫌雪每条边的交点，统计交点个数，如果交点数为奇数，则点在多边形内部，是偶数则在外部。该方法对凸多边形和非凸多边形均适用，算法复杂度为 O(n)。

2、 多边形面积算法：用待检测点与多边形的各个顶点相连，组成的三角形如果正好能够填充整个多边形，则证明该点在多边形内部。切分三角形时可以用三角剖分的方式进行，然后进行面积计算来判断。该方法的缺点是如果面对凹多边形则需要切割成凸多边形进行判断。算法复杂度也是 O(n)。

3、 角度计算：如果待检测点在多边形内部，则待检测点与多边形每两个相邻的点的夹角之和刚好是 360°，否则点就不在多边形内部。这个算法的复杂度也是 O(n)。

4、 格点法：将多边形切分成若干网格，对每个网格进行标记，标记为多边形内或者外的网格即可。然后待检测点可以用 O(1)的时间确定在哪个网格，此时只需要对该网格进行检测就可以了。

五、实验过程

Version 1 R-Tree+射线法

```
typedef RTree<int, double, 2, double> MyTree;
extern MyTree polygonTree;
extern MyTree pointTree; // each point is a rectangle
extern std::vector<int> hit_polygon_ids;
extern std::vector<int> hit_point_ids;
extern std::unordered_map<int, Rect> polygon_rect;
extern std::unordered_map<int, Rect> point_rect;
```

用外接矩形维护多边形集合和点集的增删，先得到多边形或者点的外接矩形，然后使用插入、删除。涉及到查询时，首先通过 R-Tree 的 Search 返回所有与待检测点或多边形的外接矩形有包含关系的矩形的 id，然后对每个 id 对应的点或者多边形用射线法一一检测。R-

Tree 使用的是开源代码的模板。

射线法：

```cpp
bool pointInPolygon(double px, double py, std::vector<std::pair<double, double>> &polygon) {
    bool flag = false;
    for (int i = 0, l = polygon.size(), j = l - 1; i < l; j = i, i++) {
        double sx = polygon[i].first, sy = polygon[i].second, tx = polygon[j].first, ty = polygon[j].second;
        if ((sx == px && sy == py) || (tx == px && ty == py)) return false;
        if ((sy < py && ty >= py) || (sy >= py && ty < py)) {
            double x = sx + (py - sy) * (tx - sx) / (ty - sy);
            if (x == px) return false;
            if (x > px) flag = !flag;
        }
    }
    return flag;
}
```

算法复杂度分析： R-Tree 的平均查询复杂度为 O(logN)，而通过多边形得到外接矩形的时间复杂度为 O(N)，射线法的时间复杂度也是 O(N)。实际检测的结果（在助教电脑上）如下：算法瓶颈在于通过多边形得到外接矩形以及射线法。

Case	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
Time	15.73	19.59	26.28	15.45	18.92	18.69
Score	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

Version 2 R-Tree + KD-Tree+O2 优化

因为 R-Tree 维护点集时，各个点的外接矩形包含性太低，所以加点查多边形的时候，时间上几乎等同于暴力查询。因此，引进 KD-Tree 用于点集的划分，KD-Tree 是我自己手写的，封装了 Search, Insert, Delete 几个基本的功能。对于搜索来说，给定一个矩形，返回在这个矩形内的所有点的 id，为了加快搜索过程，树中每个节点维护一个外接矩形表示子树中点的范围。而对于插入则是递归地比较坐标，完成插入并回溯更新外接矩形。对于删除来说，仅仅标记当前节点是否被删除，而不调整树的结构（为了保持高效）。这个版本的 R-Tree 部分与 version 1 一致，仅在加点查多边形时用了 KD-Tree 进行索引，然后用射线法一一判定。

KD-Tree:

```cpp
const int k = 2;
extern std::vector<int> hit_ids;
// A structure to represent node of kd tree
struct Node {
    int id;
    double min[k];
    double max[k];
    bool isDeleted;
    double point[k]; // To store k dimensional point
    Node *left, *right;
};

// A method to create a node of K D tree
Node* newNode(double arr[], int& id);
// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *(&root), double point[], int& id, unsigned depth);
// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node *Insert(Node *(&root), double point[], int& id);
// Function to delete a given point 'point[]' from tree with root
// as 'root'. depth is current depth and passed as 0 initially.
// Returns root of the modified tree.
Node* deleteNodeRec(Node *(&root), double point[], int& id, int depth);
// Function to delete a given point from K D Tree with 'root'
Node *Delete(Node *(&root), double point[], int& id);
void searchRangeRec(Node *(&root), double min[], double max[], int depth);
void Search(Node *(&root), double min[], double max[]);
#endif //INC_16307130133_KDTREE_H
```

O2 优化：

```
set(CMAKE_CXX_STANDARD 14)
SET(CMAKE_BUILD_TYPE "Release")
SET(CMAKE_CXX_FLAGS_DEBUG "$ENV{CXXFLAGS} -O2 -Wall -g -ggdb")
SET(CMAKE_CXX_FLAGS_RELEASE "$ENV{CXXFLAGS} -O2 -Wall")
```

算法复杂度分析：KD-Tree 的算法复杂度成迷，因为没有动态维护树的平衡性，所以没法给出具体的分析，但是从实际使用来看，可能是因为我的 Search 写得不好，时间效率上并不比封装的 R-Tree 表现好。另外，O2 优化能够极度提升运行效率。

Case	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
Time	5.19	6.89	6.55	5.30	6.19	5.86
Score	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

Version 3 boost

在尝试过 R-Tree 和 KD-Tree 之后，受限于效率，一度苦于寻找第三方算法库以期提高运行效率，然后尝试了 C++的 boost 库中关于 geometry 部分的库函数，但是由于安装 boost 以及后续发现格点法的原因，这个版本最终未能成型。但是我发现 boost 库中关于 R-Tree 的封装具有良好的兼容性，其不仅能够维护多边形的外接矩形，还包括线段、点等等。实际使用效率比自己写的要快很多。

Version 4 BF+BS+格点法

格点法在实际使用中时间效率远远超过射线法。

```
typedef struct {
    int     xres, yres ;    /* grid size */
    int     tot_cells ; /* xres * yres */
    double  minx, maxx, miny, maxy ;    /* bounding box */
    double  xdelta, ydelta ;
    double  inv_xdelta, inv_ydelta ;
    double  *glx, *gly ;
    GridCell    *gc ;
} GridSet, *pGridSet;

void GridSetup(std::vector<std::pair<double, double> >& pgon, int numverts, int resolution, pGridSet p_gs);
//void GridSetup(double pgon[][2], int numverts, int resolution, pGridSet p_gs);
int AddGridRecAlloc(pGridCell p_gc, double xa, double ya, double xb, double yb, double eps);
int GridTest(pGridSet p_gs, double& x, double& y);

void GridCleanup(pGridSet p_gs);

//bool pointInPolygon(double px, double py, std::vector<std::pair<double, double>> &polygon);

#endif //INC_16307130133_3_GRIDPOINT_H
```

在加多边形时，维护多边形的网格计算结果，在查点时，遍历每个多边形的网格计算结果，快速得到结果。我仍然试图用 R-Tree 索引多边形代替遍历方法，但是由于 R-Tree 插入时计算外接矩形的时间复杂度比较高，所以实际使用时有无索引并没有效率差别。

在加点时，只需要暴力保存点集，然后在查多边形的时候对多边形计算网格，遍历所有点即可。同样也尝试过用 KD-Tree 对点集进行索引，但是仍未取得良好效果。该版本还尝试了每次查多边形的时候，对点集 sort 然后二分地找出最小的在多边形外接矩形的点，但是受外接矩形计算限制，并没有提高运行效率。

```
std::vector<int> QueryPolygonFromAddPointBeforeQueryPolygon(int n, std::vector<std::pair<double, double> > polygon) {
    std::vector<int> hit_point_ids;
    double a_minX = MX, a_maxX = MIN;
    for(int i = 0; i < n; i++){
        double x = polygon[i].first;
        a_minX = a_minX < x ? a_minX : x;
        a_maxX = a_maxX > x ? a_maxX : x;
    }
    GridSet p_gs;
    GridSetup(polygon, n, 20, &p_gs);
    std::sort(points.begin(), points.end(), [](Point a, Point b){
        return a.x == b.x ? a.y < b.y : a.x < b.x;
    });//ascending order
    auto it = std::lower_bound(points.begin(), points.end(), a_minX, [](Point a, double c){
        return  a.x < c;
    });
    if(it != points.end()){
        for(auto i = it; i->x < a_maxX && i != points.end(); i++){
            if(GridTest(&p_gs, i->x, i->y))
                hit_point_ids.push_back(i->id);
        }
    }
    GridCleanup(&p_gs);
    return hit_point_ids;
}
```

实际使用：

Case	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
Time	3.71	1.47	1.91	4.53	2.13	2.43
Score	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

六、总结反思

整个 PJ 写下来，比较有意义的地方在于学习到 R-Tree、KD-Tree 等数据结构，实际上 KD-Tree 还有优化就是用三角剖分辅助判断点在多边形内外，只不过因为时间关系并没有来得及一一试验。在方法论层面，可能更多的是了解了这类问题用什么技术手段（索引结构、判定算法）进行解决和优化。当然也有吃亏的地方，因为这个 PJ 的程序计分是按运行效率计分，而最开始我走的是最普通的路——R-Tree，然后是 KD-Tree，等找到了比射线法更快的模板时，时间已经不允许我对程序进行更多的底层优化了。甚至为了效率放弃了防御式编程的思想和做法，代码风格极度不安全。

总而言之，这是一个数据结构课的课程设计，我觉得自己已经得到了最主要的收获——各种数据结构的应用场景学习，至于更多的奇技淫巧，只能自叹不如了。感谢助教全程的各种帮助，谢谢！

七、参考资料

[1]维基百科 https://en.wikipedia.org/wiki/Point_in_polygon

[2]地理围栏算法解析 https://www.cnblogs.com/mafeng/p/7909344.html

[3] R-Tree https://github.com/nushoin/RTree

[4] point-in-polygon http://alienryderflex.com/polygon/

[5] grid-method https://erich.realtimerendering.com/ptinpoly/