Geofence 实验报告

姓名：罗旭川

学号：17307130162

实验时间：2018.12.15-2019.1.10

一、实验目的

1、深入了解地理围栏（Geofence）在现实世界的应用以及其实现方法
2、学习判断点和多边形位置关系的算法
3、学会使用高效的空间索引结构以及空间划分方法
4、提升优化代码的能力

二、问题定义与问题分析

“地理围栏”问题本质上可归结为：

- 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

>>分析：运用数据结构快速索引插入的点或多边形，用射线法来精确判断点和多边形的位置关系即可。

三、处理流程和各种操作的实现方法

1、选择合适的空间索引结构及空间划分方法

在实验开始前查阅并了解了多种用于空间索引的数据结构，包括四叉树、R 树、R+树、R*树、Hilbert R 树，还了解了 geos 库，考虑到四叉树在数据分布不均匀时没有优势，又考虑到所找到的开源代码的易用性问题，因此最终首先考虑用 R*树来实现。但在之后优化遇到瓶颈后又改用了 R 树实现。

运用 R 树家族进行索引，空间划分方法自然为矩形划分。

2、使用数据结构（R*/R 树）编写代码

6 个 case 实际上归结为插入、询问、删除 3 种问题，因此只要处理好了这 3 种操作的使用，6 个 case 就都能轻易解决了。在开源代码中，Insert、Query、Remove 函数已给出.

1）插入操作的实现

>>对于多边形的插入：（每个叶子为多边形）

a. 对于每个要插入的多边形，遍历一边所有的点（O(n)），求出包围该多边形的最小矩形的边界值，即 BoundingBox

b. 考虑到后续的查询操作需要多边形的信息，因此需要构建 leaf_type 结构体，来储存叶子多边形的信息

c. 将 BoundingBox 和 leaf_type 作为参数 Insert 进 R 树中（调用 Insert 函数）

>>对于点的插入：（每个叶子为点）

a. 对于点（x，y）直接将 [x，x，y，y] 作为边界矩形 BoundingBox（[minX，maxX，minY，maxY]）

b. 同样考虑到后续的查询，需要构建 leaf_type 储存点的信息

c. 将 BoundingBox 和 leaf_type 作为参数 Insert 进 R 树中（调用 Insert 函数）

2）询问操作的实现

a. 按照上述同样的办法求出询问点或询问多边形的最小外接矩形，即为询问矩形
b. 调用 Query 函数得到与该询问矩形有重合的叶子节点的 leaf_type
（即这一步实际上为缩小了询问的范围）
c. 用射线法对待定的叶子进行细致判断，从而得到所询问叶子的 id
d. 将得到的 id 存入 vector 中然后返回

3）删除操作的实现

将需删除叶子的 BoundingBox 和 leaf_type 作为参数直接调用 Remove 函数即可。
（内部实现上实际包括查询匹配+删除+调整树中各结点的操作）

四、优化方面的问题及解决方法

以上的 R*/R 树的使用实际上为能实现 Geofence 功能的未优化版本的基本做法，之后开始对其进行各种可能的优化的尝试。

以下**不分先后**地阐述在代码优化中遇到错误或难题及其解决办法，部分使用的是 R*树的索引结构，另一部分用的是 R 树的索引结构。

由于错误和优化都是对偶的，即发生在多边形上的低效错误同样也会出现在点上，对多边形操作的优化可以同样实施于对点的操作，因此以下论述不妨**以多边形的操作为例**。

Ps：本报告中的数据是不同时期完成的程序的运行结果，再加上计算机性能较差，因此时间较长，但不影响比较，

1、【易错问题】隐式的 vector 拷贝问题

1）结构的拷贝导致 vector 的拷贝

```
struct leaf_type{
    int id;                                  //该叶子的id
    int n;                                   //多边形顶点的个数
    vector<pair<double,double> > vertex;//多边形的顶点集
};
```

>>分析：

当找到命中的叶子时，需要调用叶子中储存的有关多边形的信息，这难免需要拷贝叶子中的信息 leaf_type，若 leaf_type 中将多边形的信息以顶点数组的形式储存，那么在拷贝 leaf_type 时则会发生 vector 的拷贝，大大降低运行速度

>>解决方案：

①方案一：

用数组的指针代替 vector 存于 leaf_type 中，而实际的顶点信息则以全局数组的形式存在

```
struct leaf_type{
    int id;                          //该叶子的id
    int n;                           //多边形顶点的个数
    pair<double,double>* vertex; //顶点集的指针
};
```

（R*树）

进一步的优化，可以有以下方案二

②方案二：

leaf_type 只存放一个编号 cnt，cnt 是我为所有插入的多边形所编的序号（1...150000），同时建立全局的映射数组 cnt_id，cnt_n，顶点集则以全局的二维数组 Vertex[160000][200]存在，Vertex 的每一行存放一个插入多边形的所有顶点，即可以用 Vertex[cnt][200]存放第 cnt 个多边形的顶点集

这样就可以在保证 O（1）时间获得多边形信息的前提下，最小化 leaf_type 的大小，大幅提高了 leaf_type 的拷贝效率

```
typedef int leaf_type;
extern pair<double,double> Vertex[160000][200];  //多边形顶点集，用于储存所添加的所有多边形的点
extern int vk;                                   //vk代表所添加的polygon的cnt

extern int vk_id[160000],vk_n[160000];           //用cnt查id、n
```

（R 树）

2）函数返回值的结构中存在 vector

```
struct Visitor {
    bool ContinueVisiting;
    vector<leaf_type> _hit;
    Visitor() : ContinueVisiting(true) {};

    void operator()(const RTree::Leaf *const leaf)
```

```
Visitor Query(const A
{
    if (m_root)
    {
        QueryFunctor<
        query(m_root)
    }
    return visitor;
}
```

>>分析：

如图所示的 R*树代码中，Query 函数的返回值为一个会询问到命中的叶子的结构，为了收集到这些命中的叶子，我的第一次代码的做法是用一个_hit 数组储存这些叶子，然后顺理成章的通过 Query 返回。在这里我就隐式地拷贝了 vector，造成了效率的低下

>>解决方案：

将收集数组 hit 设置为全局数组，进一步的，不再使用 Query 的值返回方案，而改为用全局变量储存叶子信息

```
extern leaf_type hit[160000];      //用一个全局数组
```

```
struct Visitor {
    bool ContinueVisiting;

    Visitor() : ContinueVis

    void operator()(const R
    {
```

```
void Query(const Acceptor &acc
{
    if (m_root)
    {
        QueryFunctor<Acceptor,
        query(m_root);
    }
}
```

（R*树）

2、【写法分析】Case_3 和 Case_6 中并行操作点和多边形的问题

混合
先增后查
顺序随意，有删除

>>分析：

虽然看似是混合起来，但实际上：询问的点在哪些多边形内只与插入的多边形有关，而与询问的多边形无关；询问的多边形包含了哪些点只与插入的点有关，而与询问点无关。同样删除点是删除插入的点，删除多边形是删除插入的多边形，而询问的点或多边形不会插入到树中。

因此，实际上 Case_3=Case_1+Case_2 且 Case_1∩Case_2= ∅

所以可以建造两颗 R 树/R*树，一个为 point_tree，专门储存插入的点，用于解决 Case_2 类问题，另一个为 polygon_tree，专门储存插入的多边形，用于解决 Case_1 类问题。

>>解决方案：

```
extern Polygon_Tree polygon_tree;                    //专门放多边形的树
```

```
extern Point_Tree point_tree;                        //专门放点的树
```

3、【对比试验】求多边形最小外接矩形的算法优化

```
double x1,x2,y1,y2;
x1=x2=polygon[0].first, y1=y2=polygon[0].second;
for(int i=1;i<n;i++)
{
    x1=min(x1,polygon[i].first);
    x2=max(x2,polygon[i].first);
    y1=min(y1,polygon[i].second);
    y2=max(y2,polygon[i].second);
    Vertex[vk][i]=polygon[i];
}
```

【方案一】

（横向最小为 x1，最大为 x2；纵向最小为 y1，最大为 y2）

>>分析：

要找出最小外接矩形必须要遍历所有的点，因此 O（n）的时间是难免的，但是可以考虑对每次循环的操作次数做优化，如图运用 min()和 max()函数每次循环需要 4 次比较，5 次赋值。考虑到：若 polygon[i].first<x1，则它不可能大于 x2（∵x1<x2），因此这种情况下它

与 x2 的比较是多余的，y1、y2 同理，因此可以考虑用以下方案

```
double x1,x2,y1,y2;
x1=x2=polygon[0].first, y1=y2=polygon[0].second;
for(int i=0;i<n;i++)
{
    if(polygon[i].first<x1) x1=polygon[i].first;
    else if(polygon[i].first>x2) x2=polygon[i].first;
    if(polygon[i].second<y1) y1=polygon[i].second;
    else if(polygon[i].second>y2) y2=polygon[i].second;
    Vertex[vk][i]=polygon[i];
}
```

【方案二】

>>结果：

两种方案运行 Case_1 的时间：（s）

方案一	30.993	30.892	30.874
方案二	30.812	30.621	30.692

>>结论：存在优化，但数据差距较小，说明该优化不大。

4、【对比试验】将 double 数据转化为 int 数据再插入树中

>>分析：

无论是计算还是比较，int 型数据都要比 double 型数据快，又因为本次实验的实验数据都较大，将 double 数据转成 int 再插入树中对结果的正确率的影响不会太大，因此可以考虑对所用 BoundingBox 做转换 int()处理，由于射线法需要比较精确，因此叶子多边形的数据还是用 double 存储

>>结果：

Case_1 运行时间（用含有小数的测试数据）：（s）

double	27.492	27.991	27.292
int	27.051	27.332	27.194

>>结论：存在优化，但数据差距较小，说明该优化不大。

5、关于 Ret 数组实现方式的优化尝试

在程序中，查询操作的返回值被固定为了 vector，在这里我想到有三种方式来得到返回的 vector

①方案一：

设置一个全局的 vector<int> ret，每次将答案 push_back 进去，然后返回 ret 即可．但需要付出的代价是，每次调用该查询函数时要重新 ret . clear()

②方案二：

设置一个全局的 int ret[160000]，每次将答案放进去（通过维护一个 cnt），然后返回时要进行数组转 vector，

③方案三：

设置一个全局的 vector<int> ret，在环境设置时将其 resize 为 160000，查询时每次将答案放进去（同②），然后返回时要将该 ret resize 回 cnt 的大小

>>分析：

由于 ret . clear()和数组转 vector 的时间复杂度均为 O（n），这里 n 为答案的数量，因此前两种方案时间复杂度相同；
对于第三种方案，通过查阅资料发现：
“resize 函数重新分配大小，改变容器的大小，并且创建对象
当 n 小于当前 size()值时候，vector 首先会减少 size()值 保存前 n 个元素，然后将超出 n 的元素删除(remove and destroy)
当 n 大于当前 size()值时候，vector 会插入相应数量的元素 使得 size()值达到 n，并对这些元素进行初始化“
因此 resize()函数的时间复杂度也是 O（N）的，这里的 N 应该是它需要改变的空间的大小，即 160000-n。因此可以看出，第三种方案，看似结合了前两种方案的优点，但实际是耗时最长的，因为当答案数量一般会比较远离最大值 160000.

>>结果：方案三最差，考虑到指针使用起来更加灵活，因此我选择了方案二。

6、关于射线法的相关优化尝试

1)【写法分析】射线法的使用位置优化

方案一：

用全局数组 hit[160000]将命中的叶子收集起来，然后遍历一遍 hit 数组，对所有 leaf_type 进行射线法判定，若判断出其为答案，则装入 ret 数组

方案二：

在命中叶子时，即刻调用射线法函数进行判断，将判断为答案的叶子装入 ret 数组
显然方案二少了遍历数组的步骤，该写法更优

2)【对比试验】射线法写法优化

>>分析：

```
void pointInPolygon(double x,double y,pair<double, double>* polygon,int n,bool& odd)  //射线法
{
    for(int i=0,j=n-1;i<n;j=i,i++)
    {
        if((polygon[i].second<y&&polygon[j].second>=y || polygon[j].second<y && polygon[i].second>=y)
          && (polygon[i].first>=x || polygon[j].first>=x))
            if(polygon[i].first+(y-polygon[i].second)/(polygon[j].second-polygon[i].second)*(polygon[j].first-polygon[i].first)>x)
                odd=!odd;
    }
    return;
}
```

其中关键判断语句为

```
if(  polygon[i].first+(y-polygon[i].second)/(polygon[j].second-polygon[i].second)
                                        *(polygon[j].first-polygon[i].first)    >x )
    odd=!odd;
```

考虑到除法的运算速率最低，因此这里考虑改为乘法的判断语句，但因此必须增加除数正负的判断（来确定移向是否要变号），如下：

```
if((polygon[j].second>polygon[i].second)?
   (y-polygon[i].second)*(polygon[j].first-polygon[i].first)>(x-polygon[i].first)*(polygon[j].second-polygon[i].second):
   (y-polygon[i].second)*(polygon[j].first-polygon[i].first)<(x-polygon[i].first)*(polygon[j].second-polygon[i].second))
    odd=!odd;
```

>>结果：

Case_1 用时：（s）

用除法	26.3	26.371	26.301
加判断语句改乘法	26.311	26.152	26.221

>>结论：基本无优化，说明优化和代价相抵消了

3）对边建 R 树索引，降低复杂度

>>分析：

从网上得知，通过在多边形的边上建立 R 树索引结构，可以将射线法的复杂度由 O（n）降为 O（logn）.实现过程考虑在插入多边形时对每个多边形都建一个 edge_tree. 然后在查询时，对于每一个可能的多边形，都查询它的 edge_tree，命中的叶子即为射线可能穿过的边，对这些边进行射线法判断即可，这样就不用 O（n）遍历每一条边了

```
edge_tree[cnt].Insert(min,max,edge_type(polygon[i],polygon[j]));
```

```
edge_tree[leaf].Search(min,max,edge_visitor);
if(hit_edge%2) ret.push_back(cnt_id[leaf]);
```

```
bool edge_visitor(edge_type leaf)
{
    if(leaf.a.first+(query_point_y-leaf.a.second)/(leaf.b.second-leaf.a.second)
                                        *(leaf.b.first-leaf.a.first)>query_point_x)
        hit_edge++;
    return true;
}
```

>>结果：

边上建树	46.1	41，813	39.183

正常射线法	30.891	27.336	28.801

>>结论：在边上建树反而变慢了，考虑到可能是因为建树时间有所消耗，加上查询多边形边数实际较少，因此复杂度上的优势难以体现

7、【对比试验】关于删除操作的优化尝试

1）方案一：一个一个点进行删除

```
unordered_map<int,Bounding> id_bounding;
```

```
void DeletePolygonFromMixQueryPoint(int id) {
    Bounding bb=id_bounding[id];
    point_tree.Remove(bb.min,bb.max,bb.cnt);
```

①做法：

用一个叫 is_bounding 的 unordered_map 储存 id 到 BoundinBox 的映射关系，然后利用它直接调用 Remove 函数把点一个一个删除

②时间复杂度：

每次删除为 O（logn），设删除节点数为 m，则总复杂度为 O（mlogn）,m=1000

2）方案二：一个点都不删除，仅进行 is_deleted 标记

```
unordered_map<int,int> id_vk;
```

```
bool vk_isdeleted[160000];
```

```
bool has_deleted_vk;
```

①做法：

用一个叫 id_vk 的 unordered_map 储存 id 到 vk（就是 cnt）的映射关系；然后用一个全局数组来保存编号为 vk 的多边形是否被删除；全局变量 has_deleted_vk 表示已经有多边形被删除了.

```
void DeletePolygonFromMixQueryPoint(int id) {
    has_deleted_vk=true;
    vk_isdeleted[id_vk[id]]=true;
```

删除操作即为标记 has_deleted_vk=1，然后将要删除的 id 所映射的 vk 用数组标记为被删除的状态

```
bool Visitor_Point4(leaf_type leaf)
{
    if(has_deleted_vk && vk_isdeleted[leaf]) return true;
```

这样一来，在进行询问操作时，当有多边形被删除并且正在被查询到的这个叶子是已经被删除的状态时，就不用进行判断了，直接跳过这个叶子.

②时间复杂度：

unordered_map 的查询复杂度接近 O（1），且查询次数仅为 1000，因此删除操作耗时少；通过 bool 数组获知删除情况的查询复杂度为 O（1），因此虽然不删除会导致后续查询操作要在一棵较大的树上查询，但由于 O（1）即可判断树上叶子的删除与否，因此查询代价的增加也不多

3）方案三：进行批量删除，一开始用 is_deleted 标记法判断，到删除函数调用最后一次时，将之前所有点一次性删除

```
void DeletePolygonFromMixQueryPoint(int id) {
    deleted_count++;
    cnt_isdeleted[id_cnt[id]]=true;
    if(deleted_count>=1000)      //批量删除，每1000个一批
    {
        Polygon_Tree::Iterator it;
        for( polygon_tree.GetFirst(it); !polygon_tree.IsNull(it); polygon_tree.GetNext(it) )
        {
            if(cnt_isdeleted[*it]) *it=-1;
        }
        deleted_count=0;
    }
}
```

①做法：

对于前 999 个删除的多边形，只进行标记，当要求删除的个数达到 1000 时，利用 RTree.h 中实现的 iterator 类遍历所有叶子节点，将被标记的节点“删除”，这里考虑到 iterator 的删除是 unsafe 的，所以只能通过这次遍历来把被删除点的 leaf_type 置为-1，表示被删除。

②时间复杂度：

由于这里未能按照预想的真正一次性把所有节点删除，后续查询代价的增加依然存在，因此其时间复杂度显然大于方案二

>>测试结果： Case_4（s）

方案一	27.007	26.451	26.571
方案二	26.291	26.021	25.72
方案三	26.381	27.312	27.321

>>结论： 方案二最优，方案一其次，方案三最差（主要原因是不能真正删除）

8、【对比试验】R*还是 R 树？

>>分析：

由于 R*树和 R 树的用法类似，都有 Insert、Query、Remove 操作，因此我能够很轻易地从运用 R*树转化为运用 R 树，我需要从中选一种结构。由网上的资料，R*树是对 R 树的改进，它对结点的插入、分裂算法进行了改进，并采用“强制重新插入”的方法使树的结构得到优化。但我在实际编写代码时，写出的代码并不能比用 R 树的同学快，因此我需要通过对比试验来验证。两份代码除了数据结构和 Tree 的内部实现方面不同，其余外部处理均一

致或类似。

>>测试结果：（s）

————	R	R star	————	R	R star
Case_1	15.992	27.042	Case_4	25.883	27.521
Case_2	19.301	24.001	Case_5	19.001	28.074
Case_3	24.171	27.802	Case_6	22.623	29.901

>>结论：

在我所找到的 R*树和 R 树的开源文件中，R 树的表现要比 R*好.
原因可能是 R*树写的比较复杂，增加了 CPU 的计算量，并且仍然不能有效地降低空间的重叠程度

9、【编译优化】

```
set(CMAKE_BUILD_TYPE Release)
set(CMAKE_CXX_FLAGS_RELEASE "$ENV{CXXFLAGS} -O3 -Wall -Wno-unused-but-set-variable")
```

>>分析：

感谢室友指点，提醒了我可以通过编译优化来提高程序性能

>>测试结果：（s）

Case	Debug	Release –O2	Release –O3
1	30.093	26.851	26.591
2	20.162	15.991	16.032
3	25.061	20.921	20.921
4	29.041	26.521	26.133
5	18.092	14.267	14.158
6	22.321	18.661	18.64

>>结论：通过设置编译选项参数，可以实现编译优化，大幅提高程序性能

五、实验结果

第二次测试提交结果：　　**第八次测试提交结果：**

```
==== case 1 report ====
7.66
1.0000
==== case 2 report ====
8.50
1.0000
==== case 3 report ====
8.03
1.0000
==== case 4 report ====
7.83
1.0000
==== case 5 report ====
10.06
1.0000
==== case 6 report ====
8.75
1.0000
```

```
==== case 1 report ====
3.96
1.0000
==== case 2 report ====
2.40
1.0000
==== case 3 report ====
2.69
1.0000
==== case 4 report ====
3.91
1.0000
==== case 5 report ====
2.40
1.0000
==== case 6 report ====
2.49
1.0000
```

六、实验心得

>>对 R 树变得十分熟悉：

实验的准备阶段时，我们要做的任务是找到一个合适的数据结构来索引空间，当我一开始看到 R*树这类的结构时，由于第一次接触，很难看懂代码的意思，到后来通过阅读许多有关 R 树的博客和论文，逐渐明白了 R 树家族的索引原理以及基本的函数类型，之后我就可以慢慢的看得懂代码，找得到代码的接口了。等到我能够写出一个跑出正确答案的代码时，我已经对 R 树更加熟悉了。在优化阶段，我甚至会尝试修改 R 树中的相关语句，来实现我的一些优化目的。

>>关于 debug：

在做这个实验时，由于测试样例太大，debug 其实挺不方便，大多数时候是通过肉眼观察或者 cout 输出关键信息来辅助肉眼 debug，有时候会自己写一个小的测试样例集来方便测试，当然也还是有打开真正的测试样例进行查点 debug 的经历。

>>关于各种模板的尝试：

在第一次写完 R*树的版本时，实在找不到优化的点了，曾经觉得已经到了程序的极限了，直到看到了其他同学跑出的时间，才坚定了我继续努力的信念。由于找不到优化之处了，我认为其他同学一定是用了更加高效的索引结构，于是开始广泛的尝试，Hilbert RTree、geos 库、线段树等等，甚至还尝试运用“一维”的 R*树，但都以失败告终，最终选择了和 R*类似的 R 树，并取得了进一步的突破。

>>总结：

总的来说，这次实验，虽然代码的效率可能比很多人都差，但是通过了这次实验，我对空间索引有了更深程度的体会，对代码优化的技巧方面也有了许多的收获，收获挺大的。

七、实验参考资料

开源代码：https://github.com/virtuald/r-star-tree
https://github.com/nushoin/RTree

参考博客：https://blog.csdn.net/zhouxuguang236/article/details/7898272
https://www.cnblogs.com/arxive/p/8139109.html
https://blog.csdn.net/chelen_jak/article/details/52650886
参考网站：https://www.jb51.net/article/54792.htm
http://www.cnblogs.com/On1Key/p/5673886.html