HW6-B Spatial Index

> 胡宇森 3200100578 地理信息科学
>
> 魏辰 3200100505 地理信息科学

一、空间数据类型实现

1 Point到LineString和Polygon之间的欧式距离计算

原理

1. Point到LineString的距离计算分解为Point到每个线段的距离计算，Point到所有线段的最短距离即为Point到LineString的距离。求解Point到线段的距离时，首先求解Point在线段上的投影点，判断投影点是否在线段上，然后分情况计算距离：如果在线段上，计算投影点到Point的距离，不在线段上，计算Point到线段端点距离的最小值。需要注意的是，还需要单独考虑点在线段或线段延长线上的情况。
2. Point到Polygon之间欧式距离计算时，首先需要判断Point与Polygon的关系，即Point是否在Polygon内部。此处运用射线法进行判断，即从Point处向左向右各射出一条水平射线，统计Polygon与两射线的交点数。如两侧交点数都为奇数，则Point在Polygon内，如两侧交点数都为偶数，则Point在Polygon外。奇异情况为射线穿过Polygon顶点与射线与Polygon边重合。只需要认为射线穿过的点在射线上方，判断线段端点是否在射线两侧来决定是否计算交点，即可解决奇异情况。对于Point在Polygon外部的情况，计算Point到Polygon的Ring的距离即可。

实现

- Poing到LineString的距离

```
double Point::distance(const LineString *line) const
{
    double mindist = line->getPointN(0).distance(this);
    for (size_t i = 0; i < line->numPoints() - 1; ++i)
    {
        double dist = 0;
        double x1 = line->getPointN(i).getX();
        double y1 = line->getPointN(i).getY();
        double x2 = line->getPointN(i + 1).getX();
        double y2 = line->getPointN(i + 1).getY();
        // Task calculate the distance between Point P(x, y) and Line [P1(x1,
y1), P2(x2, y2)] (less than 10 lines)
        // TODO
        double x = this->getX(), y = this->getY(), x0, y0;

        if (x1 == x2)
        {
            x0 = x1;
            y0 = y;
            if ((y1 <= y0 && y0 <= y2) || (y2 <= y0 && y0 <= y1))
            {
                mindist = 0;
                break;
```

```
            }
            else
            {
                dist = std::min(sqrt(pow((x - x1), 2) + pow((y - y1), 2)),
sqrt(pow((x - x2), 2) + pow((y - y2), 2)));
                mindist = std::min(mindist, dist);
                continue;
            }
        }

        if (y1 == y2)
        {
            x0 = x;
            y0 = y1;
        }
        else if ((x - x1) / (y - y1) == (x - x2) / (y - y2) && ((x1 <= x && x <=
x2) || (x2 <= x && x <= x1)))
        {
            mindist = 0;
            break;
        }
        else if ((x - x1) / (y - y1) == (x - x2) / (y - y2))
        {
            x0 = x;
            y0 = y;
        }
        else
        {
            x0 = x1 + ((x - x1) * (x2 - x1) + (y - y1) * (y2 - y1)) / (sqrt((1 +
(y2 - y1) * (y2 - y1) / ((x2 - x1) * (x2 - x1))) * ((x2 - x1) * (x2 - x1) + (y2
- y1) * (y2 - y1))));
            y0 = y1 + (y2 - y1) / (x2 - x1) * (x0 - x1);
            if (int((y - y0) * (y2 - y1) / ((x - x0) * (x2 - x1))) != -1)
            {
                x0 = x1 - ((x - x1) * (x2 - x1) + (y - y1) * (y2 - y1)) /
(sqrt((1 + (y2 - y1) * (y2 - y1) / ((x2 - x1) * (x2 - x1))) * ((x2 - x1) * (x2 -
x1) + (y2 - y1) * (y2 - y1))));
                y0 = y1 + (y2 - y1) / (x2 - x1) * (x0 - x1);
            }
        }
        if ((x1 <= x0 && x0 <= x2) || (x2 <= x0 && x0 <= x1))
            dist = sqrt(pow((x - x0), 2) + pow((y - y0), 2));
        else
            dist = std::min(sqrt(pow((x - x1), 2) + pow((y - y1), 2)),
sqrt(pow((x - x2), 2) + pow((y - y2), 2)));
        mindist = std::min(mindist, dist);
    }
    return mindist;
}
```

- Point到Polygon的距离一并放在下一节实现Polygon内环后。

测试

```
**********Start**********
测试2: Distance between Point and LineString
Distance between Point and LineString: 10 / 10 tests are passed
**********End**********
```

分析

1. 所有测试点都能通过，可见代码正确性。
2. 由于考虑情况较多，代码较为冗长。后续考虑合并同类情况，进行优化。

2 Polygon的内环几何数据存储，修改Point到Polygon欧氏距离计算

原理

1. Polygon的内环与外环的区别是，Polygon可以有多个内环，因此我们需要一个`std::vector<LineString>`的结构来存储Polygon的内环。
2. Point到带有内环的Polygon的距离计算与不带有内环的Polygon相似，只需要在遍历时读取内环一同遍历即可。

实现

- 带有内环的Polygon类

```
class Polygon : public Geometry
{
    private:
    LineString exteriorRing;
    std::vector<LineString> interiorRings;

    public:
    Polygon() {}
    Polygon(LineString &ering, std::vector<LineString> &iring =
std::vector<LineString>()) : exteriorRing(ering), interiorRings(iring) {
constructEnvelope(); }
    virtual ~Polygon() {}

    LineString getExteriorRing() const { return exteriorRing; }
    std::vector<LineString> getInteriorRings() const { return interiorRings; }

    virtual void constructEnvelope() { envelope = exteriorRing.getEnvelope(); }

    // Euclidean distance
    virtual double distance(const Point *point) const
    {
        return point->distance(this);
    }
    virtual double distance(const LineString *line) const
    {
        return line->distance(this);
    }
    virtual double distance(const Polygon *polygon) const;
```

```
    // intersection test with the envelope for range query
    virtual bool intersects(const Envelope &rect) const;

    virtual void draw() const;

    virtual void print() const;
};
```

- Point到带有内环的Polygon的距离

```
double Point::distance(const Polygon *polygon) const
{
    std::vector<LineString> rings, inRings = polygon->getInteriorRings();
    size_t exNum = polygon->getExteriorRing().numPoints();
    double mindist = 0;

    rings.push_back(polygon->getExteriorRing());
    for (auto iter = inRings.begin(); iter != inRings.end(); iter++)
        if ((*iter).numPoints())
            rings.push_back(*iter);
    // Task whether Point P(x, y) is within Polygon (less than 15 lines)
    // TODO
    double x = this->getX(), y = this->getY();
    int leftcount = 0, rightcount = 0;

    for (auto iter = rings.begin(); iter != rings.end(); iter++)
    {
        if (this->distance(&(*iter)) == 0)
            return 0;
        for (size_t i = 0; i < (*iter).numPoints() - 1; i++)
        {
            double x1 = (*iter).getPointN(i).getX();
            double y1 = (*iter).getPointN(i).getY();
            double x2 = (*iter).getPointN(i + 1).getX();
            double y2 = (*iter).getPointN(i + 1).getY();
            if (y1 > y2)
            {
                std::swap(x1, x2);
                std::swap(y1, y2);
            }
            if ((y1 <= y && y <= y2))
            {
                if (y2 == y && y != y1)
                {
                    if (x > x2)
                        leftcount++;
                    else
                        rightcount++;
                }
                else if (y != y1)
                {
                    double projectX = x1 + (x2 - x1) * (y - y1) / (y2 - y1);
                    if (projectX < x)
                        leftcount++;
                    else
```

```
                        rightcount++;
                }
            }
        }
    }

    if (!((leftcount % 2) && (rightcount % 2)))
    {
        for (auto iter = rings.begin(); iter != rings.end(); iter++)
        {
            double dist = this->distance(&(*iter));
            if (iter == rings.begin())
                mindist = dist;
            else if (dist < mindist)
                mindist = dist;
        }
    }
    return mindist;
}
```

测试

测试5用例详见QuadTreeTest.cpp

```
**********Start**********
测试3: Distance between Point and Polygon
Distance between Point and Polygon: 10 / 10 tests are passed
**********End**********

**********Start**********
测试5: Distance between Point and Polygon with interior ring
Distance between Point and Polygon: 10 / 10 tests are passed
**********End**********
```

分析

1. 所有测试点都能通过，可见代码正确性。
2. 带有内环的Polygon和不带有内环的Polygon在计算到点的距离时实现差别不大。

二、包围盒类函数实现

1 包围盒contain、intersect和unionEnvelope函数实现

原理

1. 总体上来说包围盒类函数的实现逻辑类似，因为包围盒皆为固定的长方形，长方体的四个顶点可以有效地定位包围盒位置。
2. contain关系说明一个包围盒完全位于另一个包围盒之中，从一维直线的角度来看，min1>min2 and max1<max2即为一维线段的contain，二维包围盒的x、y两个维度同时满足即可。值得注意的是，与Post GIS中的contain不同，此处的contain关系适用于相同的包围盒，即四个判断全部取等。
3. intersect关系说明一个包围盒和另一个包围盒存在相交，从一维直线的角度来看，max1>=min2 and min1<=max2即为一维线段的intersect，二维包围盒的x、y两个维度同时满足即可。

4. unionEnvelope将两个包围盒拼接成一个大包围盒，分别选取二者在最左下侧和最右上侧的点生成新长方体即拼接后的包围盒。

实现

- 包围盒contain函数

```
bool Envelope::contain(const Envelope &envelope) const
{
    // Task 测试Envelope是否包含关系
    // TODO
    return envelope.minX >= minX && envelope.maxX <= maxX && envelope.minY >= minY && envelope.maxY <= maxY;
}
```

- 包围盒intersect函数

```
bool Envelope::intersect(const Envelope &envelope) const
{
    // Task 测试Envelope是否相交
    // TODO
    return maxX >= envelope.minX && minX <= envelope.maxX && maxY >= envelope.minY && minY <= envelope.maxY;
}
```

- 包围盒unionEnvelope函数

```
Envelope Envelope::unionEnvelope(const Envelope &envelope) const
{
    // Task 合并两个Envelope生成一个新的Envelope
    // TODO
    return Envelope(std::min(minX, envelope.minX), std::max(maxX, envelope.maxX), std::min(minY, envelope.minY), std::max(maxY, envelope.maxY));
}
```

测试

```
**********Start**********
测试1: Envelope Contain, Intersect, and Union
Envelope Contain: 20 / 20 tests are passed
Envelope Intersect: 20 / 20 tests are passed
Envelope Union: 20 / 20 tests are passed
**********End**********
```

分析

1. 所有测试点都能通过，可见代码正确性。
2. 包围盒类基本函数的原理较为简单，实现难度较低，需要注意等号是否可取。

2 Polygon与Envelope相交判断intersects函数实现

原理

Polygon与Envelope相交的判断可以分为以下顺序三部分，在前一部分不满足的基础上判断下一部分是否满足，存在一部分满足即为相交，全部不满足为不相交，此处的Polygon是包括内环的Polygon

1. 判断外环与包围盒是否相交（直接调用折线与包围盒相交函数）
2. 判断内环与包围盒是否相交（直接调用折线与包围盒相交函数）
3. 分别判断包围盒的四个顶点是否在多边形内部，如果存在一个顶点在多边形内部，则二者必然相交（判断点是否在多边形内部的思路已在点到多边形的距离部分详细阐述）

实现

- 带有内环的Polygon类

```
bool Polygon::intersects(const Envelope &rect) const
{
    // TODO
    bool flag_ExteriorRing = false, flag_InteriorRings = false, flag_Contains =
false;
    std::vector<LineString> rings;
    rings.push_back(this->getExteriorRing());

    if (this->getExteriorRing().intersects(rect))
        flag_ExteriorRing = true;
    for (auto iter = this->interiorRings.begin(); iter != this-
>interiorRings.end(); iter++)
    {
        if ((*iter).intersects(rect))
            flag_InteriorRings = true;
        if ((*iter).numPoints())
            rings.push_back(*iter);
    }

    std::vector<std::vector<double>> envelope_Vertices = {{rect.getMaxX(),
rect.getMaxY()},
                                                          {rect.getMaxX(),
rect.getMinY()},
                                                          {rect.getMinX(),
rect.getMaxY()},
                                                          {rect.getMinX(),
rect.getMinY()}};

    for (auto iter = envelope_Vertices.begin(); iter != envelope_Vertices.end();
iter++)
    {
        double x = (*iter)[0], y = (*iter)[1];
        int leftcount = 0, rightcount = 0;

        for (auto iter = rings.begin(); iter != rings.end(); iter++)
            for (size_t i = 0; i < (*iter).numPoints() - 1; i++)
            {
                double x1 = (*iter).getPointN(i).getX();
                double y1 = (*iter).getPointN(i).getY();
```

```
                double x2 = (*iter).getPointN(i + 1).getX();
                double y2 = (*iter).getPointN(i + 1).getY();
                if (y1 > y2)
                {
                    std::swap(x1, x2);
                    std::swap(y1, y2);
                }
                if ((y1 <= y && y <= y2))
                {
                    if (y2 == y && y != y1)
                    {
                        if (x > x2)
                            leftcount++;
                        else
                            rightcount++;
                    }
                    else if (y != y1)
                    {
                        double projectX = x1 + (x2 - x1) * (y - y1) / (y2 - y1);
                        if (projectX < x)
                            leftcount++;
                        else
                            rightcount++;
                    }
                }
            }
        if ((leftcount % 2) && (rightcount % 2))
            flag_Contains = true;
    }

    return flag_ExteriorRing || flag_InteriorRings || flag_Contains;
}
```

测试

测试6用例详见QuadTreeTest.cpp

```
**********Start**********
测试6: Polygon and Envelope Intersect
Polygon and Envelope Intersect: 10 / 10 tests are passed
**********End**********
```

分析

1. 所有测试点都能通过，可见代码正确性。
2. Polygon和Envelope的intersects函数主要实现难点在于合理的分类讨论，逐步判断。

三、四叉树的创建与查询

1 四叉树的创建

原理

1. 创建四叉树时首先创建一个根节点，并输入一组几何特征。
2. 将节点分裂成四个子节点，将每个特征划分给包围盒重叠的子节点，可能存在一个特征划分给多个子节点。
3. 删除当前节点的几何特征记录。
4. 如果子节点特征数大于指定capacity，重复2、3步递归生成子节点，四叉树创建完毕。

实现

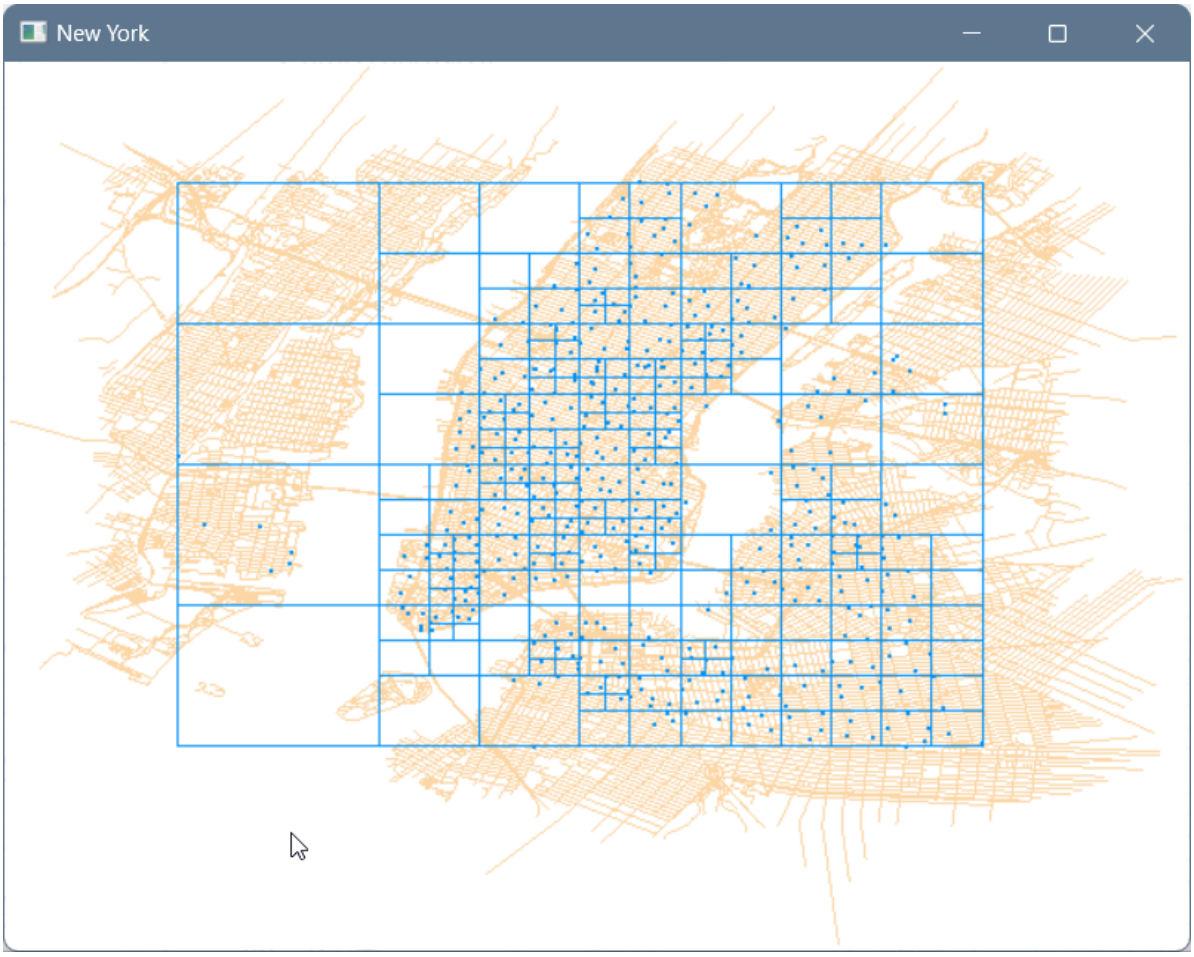
- 四叉树创建

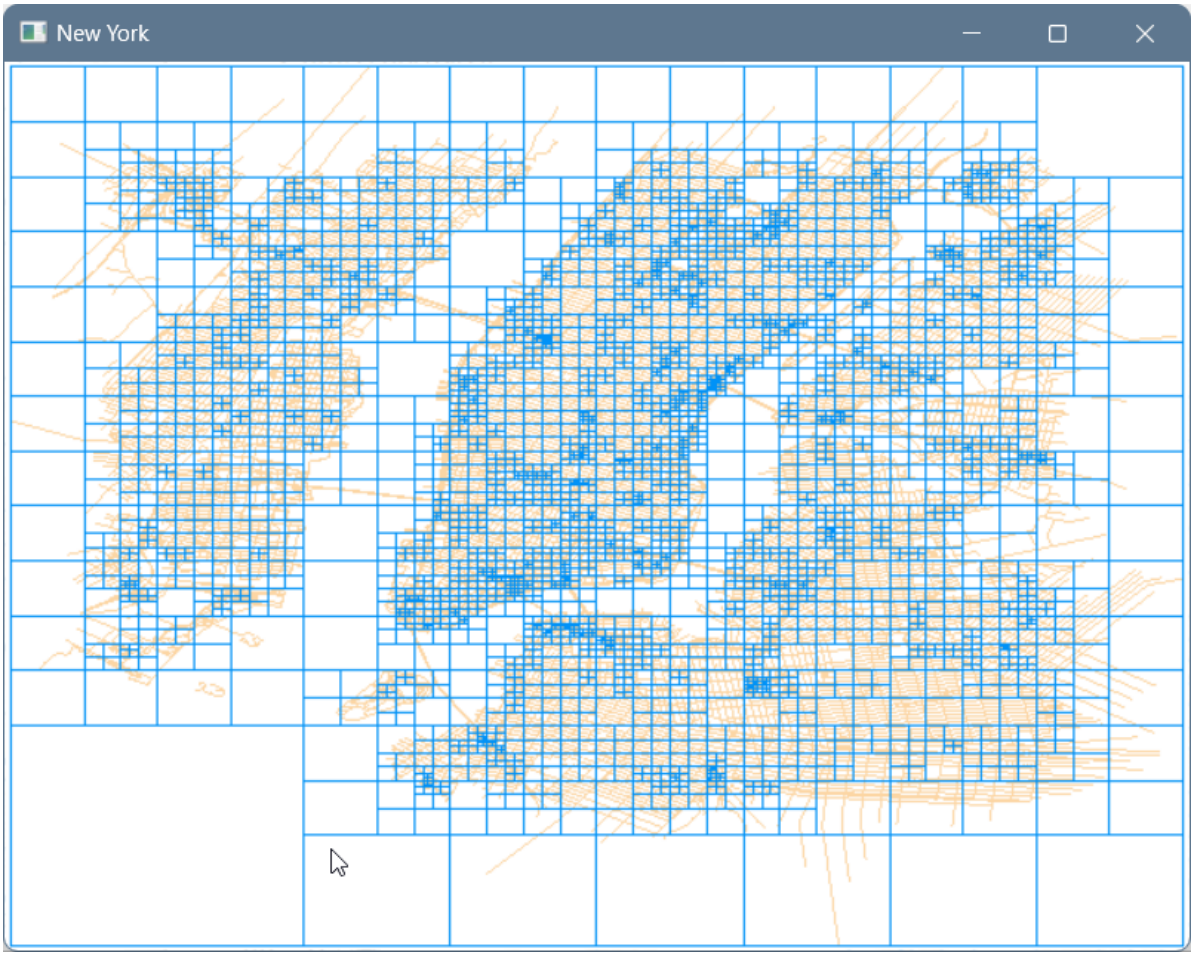
```
bool QuadTree::constructTree(const std::vector<Feature> &features)
{
    if (features.empty())
        return false;

    // Task construction
    // TODO
    bbox = features[0].getEnvelope();
    for (auto i : features)
    {
        bbox = bbox.unionEnvelope(i.getEnvelope());
    }
    root = new QuadNode(bbox);
    root->add(features);
    root->split(capacity);

    return true;
}

void QuadNode::split(size_t capacity)
{
    for (int i = 0; i < 4; ++i)
    {
        delete children[i];
        children[i] = nullptr;
    }

    // Task construction
    // TODO
    // 拆分为4个子节点
    children[0] = new QuadNode(Envelope(bbox.getMinX(), bbox.getMaxX() -
bbox.getWidth() / 2, bbox.getMinY(), bbox.getMaxY() - bbox.getHeight() / 2));
    children[1] = new QuadNode(Envelope(bbox.getMinX() + bbox.getWidth() / 2,
bbox.getMaxX(), bbox.getMinY(), bbox.getMaxY() - bbox.getHeight() / 2));
    children[2] = new QuadNode(Envelope(bbox.getMinX(), bbox.getMaxX() -
bbox.getWidth() / 2, bbox.getMinY() + bbox.getHeight() / 2, bbox.getMaxY()));
    children[3] = new QuadNode(Envelope(bbox.getMinX() + bbox.getWidth() / 2,
bbox.getMaxX(), bbox.getMinY() + bbox.getHeight() / 2, bbox.getMaxY()));

    // 分配要素
    for (auto i : features)
```

```
    {
        for (auto j = 0; j < 4; ++j)
        {
            if (children[j]->getEnvelope().intersect(i.getEnvelope()))
                children[j]->features.push_back(i);
        }
    }

    // 删除要素向量
    features.clear();

    // 继续细分
    for (auto i = 0; i < 4; ++i)
    {
        // std::cout << children[i]->features.size() << std::endl;
        if (children[i]->features.size() > capacity)
            children[i]->split(capacity);
    }
}
```

测试

```
**********Start**********
测试4: QuadTree Construction
QuadTree Construction: 2 / 2 tests are passed
**********End**********
```





分析

1. 所有测试点都能通过，可见代码正确性。
2. 四叉树的创建的重点主要在于判断递归中止条件与几何特征的分配处理。

2 区域查询

原理

通过递归查询，逐层查询与区域rect相交的节点，直到查询到叶节点，获取与rect相交的Feature作为候选Feature。此处相交只使用包围盒判断，作为粗查询。在获取所有候选Feature后，进行精查询。

实现

- rangeQuery粗查询

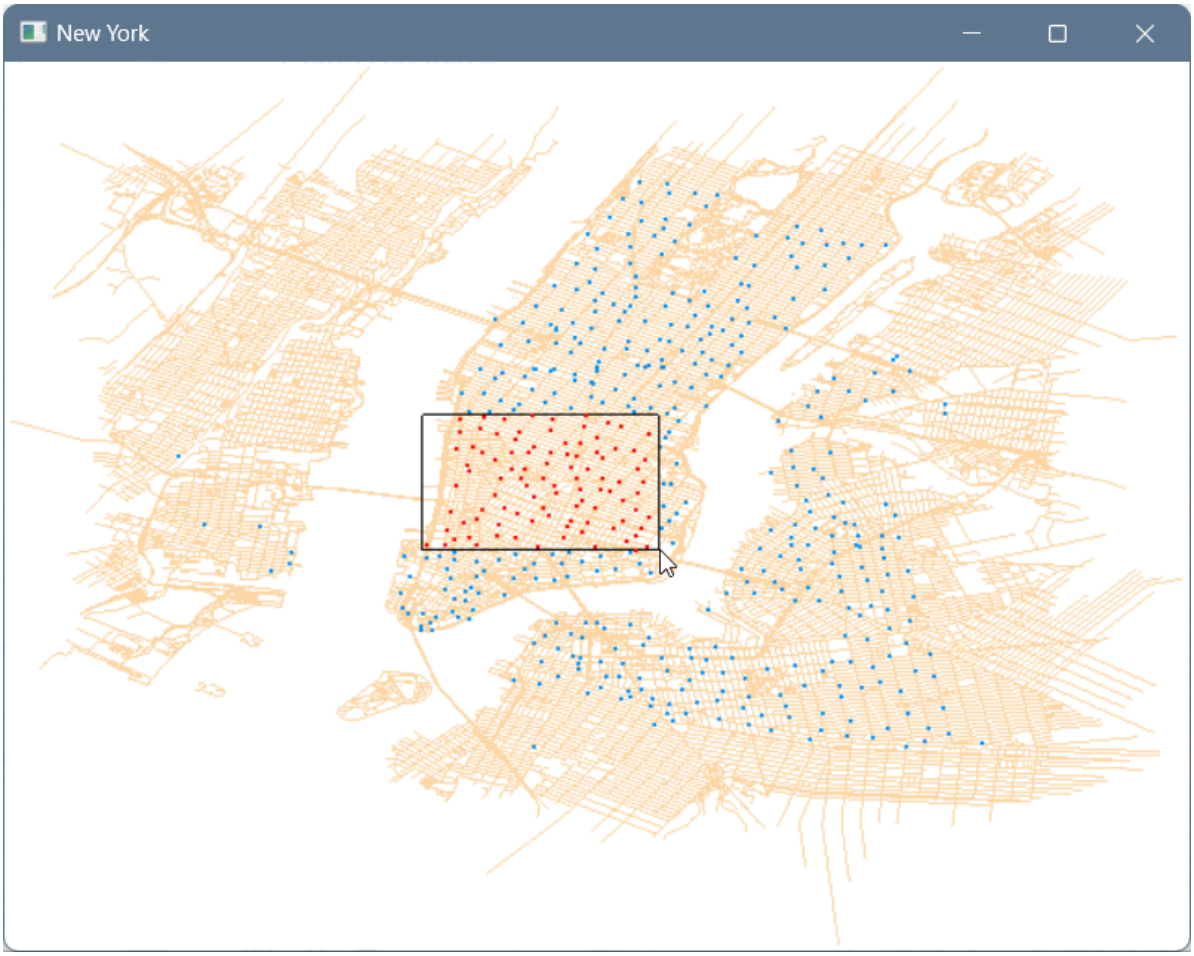
```
void QuadTree::rangeQuery(const Envelope &rect,
                          std::vector<Feature> &features)
{
    features.clear();

    // Task range query
    // TODO
    if (root)
        root->rangeQuery(rect, features);
}

void QuadNode::rangeQuery(const Envelope &rect,
                          std::vector<Feature> &features)
```

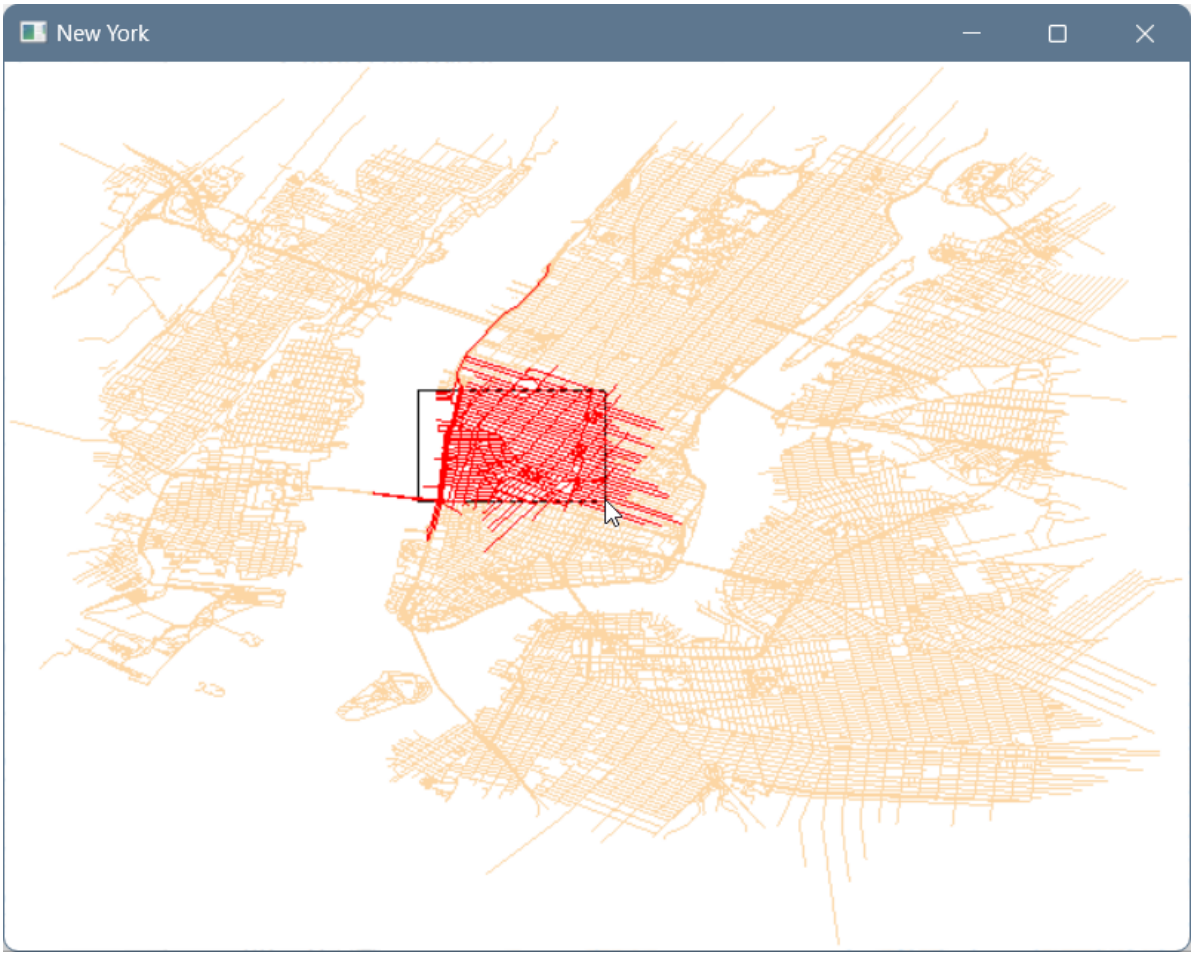
```
{
    // Task range query
    // TODO
    if (!bbox.intersect(rect))
        return;
    else if (!isLeafNode())
    {
        for (auto i = 0; i < 4; ++i)
        {
            children[i]->rangeQuery(rect, features);
        }
    }
    else if (rect.contain(bbox))
        features.insert(features.end(), this->features.begin(), this-
>features.end());
    else
        for (auto i : this->features)
            if (rect.intersect(i.getEnvelope()))
                features.push_back(i);
}
```

- rangeQuery精查询

```
void rangeQuery()
{
    vector<hw6::Feature> candidateFeatures;

    // filter step (使用四叉树获得查询区域和几何特征包围盒相交的候选集)
    if (mode == RANGEPOINT)
        pointTree->rangeQuery(selectedRect, candidateFeatures);
    else if (mode == RANGELINE)
        roadTree->rangeQuery(selectedRect, candidateFeatures);
    // refine step (精确判断时，需要去重，避免查询区域和几何对象的重复计算)
    // TODO
    if (mode == RANGEPOINT)
        selectedFeatures = candidateFeatures;
    else if (mode == RANGELINE)
    {
        for (auto it = candidateFeatures.begin(); it != candidateFeatures.end();
it++)
        {
            if (it->getGeom()->intersects(selectedRect))
                selectedFeatures.push_back(*it);
        }
    }
}
```

测试


New York


New York

分析

1. 根据几何关系可知，rangeQuery结果是正确的。
2. 使用不同大小rect测试，查询时都较为流畅。

3 最邻近几何特征查询

原理

1. 邻近查询查询主要分为两步。首先通过pointInLeafNode查询输入点所在的叶子节点（即输入点在结点的包围盒中），需要注意的是，输入点不一定在某个叶节点中，因此此步找出输入点所在最深的节点。然后计算输入点与该最深节点的各几何特征包围盒的最大距离的最小值，构造查询区域(x – minDist, x + minDist, y – minDist, y + minDist)进行区域查询，得到结果作为候选Feature，而后进行精确判断即可。
2. pointInLeafNode采用深度优先搜索，寻找查询点所在最深的节点。

实现

- NNQuery粗查询

```
bool QuadTree::NNQuery(double x, double y, std::vector<Feature> &features)
{
    if (!root || !(root->getEnvelope().contain(x, y)))
        return false;

    // Task NN query
    // TODO
    QuadNode *leafNode = root->pointInLeafNode(x, y);

    const Envelope &envelope = root->getEnvelope();
    double minDist = std::max(envelope.getWidth(), envelope.getHeight());

    for (auto i = 0; i < leafNode->getFeatureNum(); ++i)
    {
        minDist = std::min(minDist, leafNode-
>getFeature(i).maxDistance2Envelope(x, y));
    }
    root->rangeQuery(Envelope(x - minDist, x + minDist, y - minDist, y +
minDist), features);

    return true;
}
```

- NNQuery精查询

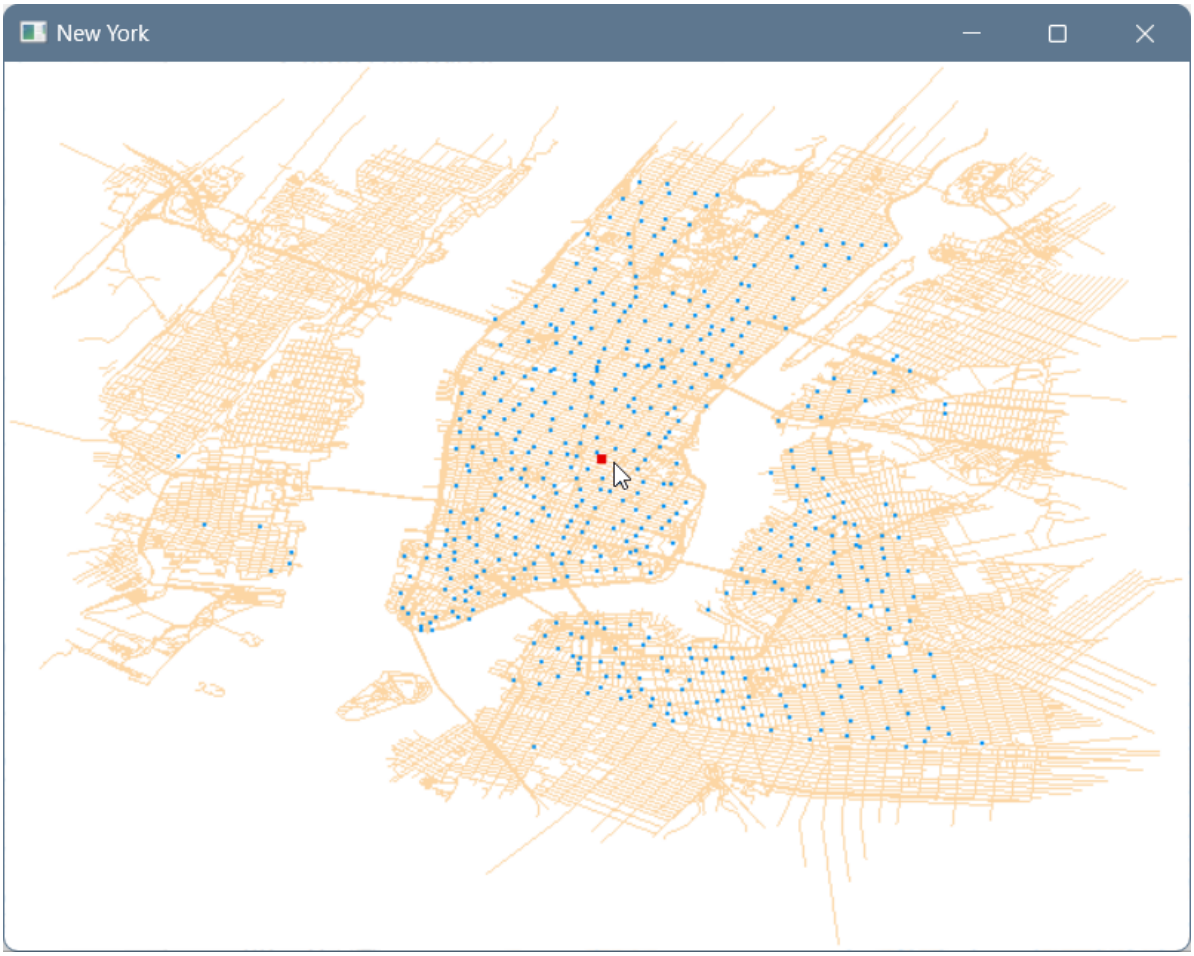
```
void NNQuery(hw6::Point p)
{
    vector<hw6::Feature> candidateFeatures;

    // filter step (使用四叉树获得距离较近的几何特征候选集)
    if (mode == NNPOINT)
        pointTree->NNQuery(p.getX(), p.getY(), candidateFeatures);
    else if (mode == NNLINE)
        roadTree->NNQuery(p.getX(), p.getY(), candidateFeatures);
```

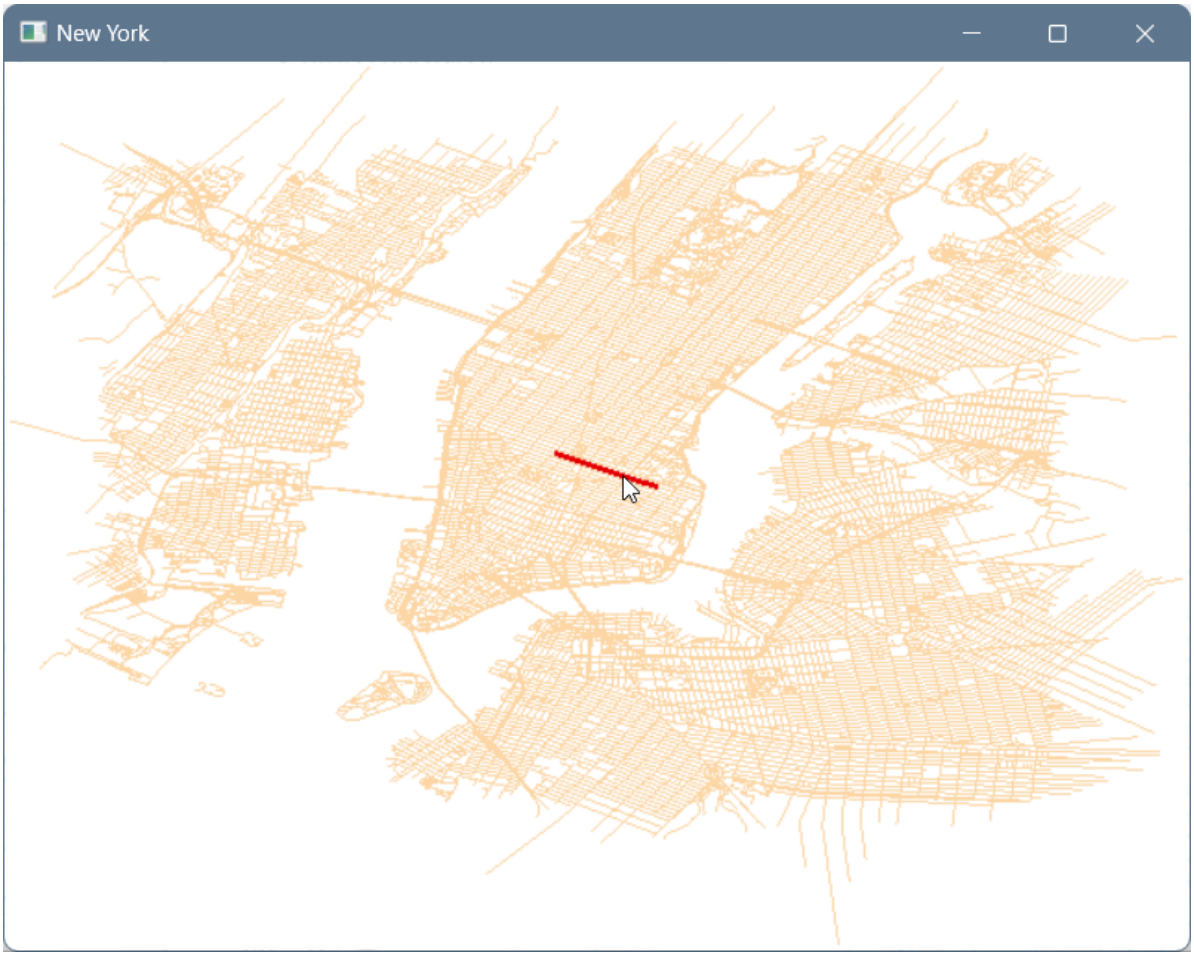
```
    // refine step (精确计算查询点与几何对象的距离)
    // TODO
    double dist = 1000000;
    for (auto it = candidateFeatures.begin(); it != candidateFeatures.end();
++it)
    {
        double tmpDist = it->getGeom()->distance(&p);
        if (tmpDist < dist && tmpDist)
        {
            dist = tmpDist;
            nearestFeature = *it;
        }
    }
}
```

- pointInLeafNode实现

```
QuadNode *QuadNode::pointInLeafNode(double x, double y)
{
    // Task NN query
    // TODO
    if (!this->bbox.contain(Point(x, y).getEnvelope()))
        return nullptr;
    else if (isLeafNode())
    {
        return this;
    }
    else
    {
        for (auto i = 0; i < 4; ++i)
            if (children[i]->pointInLeafNode(x, y) != nullptr)
                return children[i]->pointInLeafNode(x, y);
    }

    return nullptr;
}
```

测试


New York


New York

分析

1. 根据几何关系可知，NNQuery结果是正确的。
2. 在不同区域进行测试，查询时都较为流畅。

四、R-树的创建与查询

1 R-树的创建

原理

1. 创建R-树时首先创建叶子节点，叶子节点存储指向Feature的指针和节点的最小包围盒。
2. 创建叶子节点时将Feature逐个插入，基于节点新增面积越小越好的原则选择节点，当超过节点所能存储的最大几何特征时，基于二次分裂(quadratic split)算法，选择最左和最右两个几何特征作为种子点，对几何特征进行分组。
3. 创建完叶子节点后，基于创建的叶子节点创建上层内部节点，同理采用节点新增面积越小越好的原则选择节点，超过节点存储上限时进行二次分裂。
4. 当某一层的节点数量小于M时，创建根节点，R-树创建完毕。

实现

- R-树创建

```
virtual bool constructTree(const std::vector<Feature> &features) override
{
    // TODO
    std::vector<Feature> existFeatures;
    std::vector<RNode<M> *> leafNodes;

    // 叶子节点获取
    for (auto it = features.begin(); it != features.end(); ++it)
    {
        double minAreaAdd = 1000000;
        Envelope featureEnvelope = (*it).getEnvelope(), markEnvelope;
        RNode<M> *mark;
        std::vector<RNode<M> *>::iterator markIter;
        int markPos;
        if (leafNodes.size() == 0)
        {
            RNode<M> *firstNode = new RNode<M>(featureEnvelope);
            firstNode->add((*it));
            leafNodes.push_back(firstNode);
            continue;
        }

        // 与每个叶子节点比
        for (auto rIter = leafNodes.begin(); rIter != leafNodes.end(); ++rIter)
        {
            Envelope nodeEnvelope = (*rIter)->getEnvelope();
            double minX = std::min(featureEnvelope.getMinX(),
nodeEnvelope.getMinX());
            double minY = std::min(featureEnvelope.getMinY(),
nodeEnvelope.getMinY());
            double maxX = std::max(featureEnvelope.getMaxX(),
nodeEnvelope.getMaxX());
```

```
            double maxY = std::max(featureEnvelope.getMaxY(),
nodeEnvelope.getMaxY());
            Envelope newEnvelope(minX, maxX, minY, maxY);
            double areaAdd = newEnvelope.getArea() - nodeEnvelope.getArea();
            if (areaAdd < minAreaAdd)
            {
                minAreaAdd = areaAdd;
                mark = *rIter;
                markEnvelope = newEnvelope;
                markIter = rIter;
            }
        }

        if (mark->getFeatureNum() < M)
        {
            mark->add(*it);
            mark->setEnvelope(markEnvelope);
        }
        // 二次分裂
        else
        {
            std::vector<Feature> allFeatures = mark->getFeatures();
            Feature seed1, seed2, insertFeature = *it;
            std::vector<Feature>::iterator seed1Iter, seed2Iter;
            int seed1pos, seed2pos;
            allFeatures.push_back(insertFeature);
            double left = 1000000, right = -1000000;
            for (auto fIter = allFeatures.begin(); fIter != allFeatures.end();
++fIter)
            {
                if (left > fIter->getEnvelope().getMinX())
                {
                    left = fIter->getEnvelope().getMinX();
                    seed1 = *fIter;
                    seed1Iter = fIter;
                }
                if (right < fIter->getEnvelope().getMaxX())
                {
                    right = fIter->getEnvelope().getMaxX();
                    seed2 = *fIter;
                    seed2Iter = fIter;
                }
            }
            Envelope env1 = seed1.getEnvelope(), env2 = seed2.getEnvelope();
            std::vector<Feature> seed1Features, seed2Features;
            seed1Features.push_back(seed1);
            seed2Features.push_back(seed2);

            for (auto fIter = allFeatures.begin(); fIter != allFeatures.end();)
            {
                if (fIter->getName() == seed1.getName())
                {
                    fIter = allFeatures.erase(fIter);
                    break;
                }
```

```
                else
                    fIter++;
            }
            for (auto fIter = allFeatures.begin(); fIter != allFeatures.end();)
            {
                if (fIter->getName() == seed2.getName())
                {
                    fIter = allFeatures.erase(fIter);
                    break;
                }
                else
                    fIter++;
            }
            for (auto fIter = allFeatures.begin(); fIter != allFeatures.end();
++fIter)
            {
                double addArea1, addArea2;
                double minX = std::min(fIter->getEnvelope().getMinX(),
env1.getMinX());
                double minY = std::min(fIter->getEnvelope().getMinY(),
env1.getMinY());
                double maxX = std::max(fIter->getEnvelope().getMaxX(),
env1.getMaxX());
                double maxY = std::max(fIter->getEnvelope().getMaxY(),
env1.getMaxY());
                Envelope newEnvelope1, newEnvelope2;
                newEnvelope1 = Envelope(minX, maxX, minY, maxY);
                addArea1 = newEnvelope1.getArea() - env1.getArea();
                minX = std::min(fIter->getEnvelope().getMinX(), env2.getMinX());
                minY = std::min(fIter->getEnvelope().getMinY(), env2.getMinY());
                maxX = std::max(fIter->getEnvelope().getMaxX(), env2.getMaxX());
                maxY = std::max(fIter->getEnvelope().getMaxY(), env2.getMaxY());
                newEnvelope2 = Envelope(minX, maxX, minY, maxY);
                addArea2 = newEnvelope2.getArea() - env2.getArea();
                if (addArea1 < addArea2)
                {
                    seed1Features.push_back(*fIter);
                    env1 = newEnvelope1;
                }
                else
                {
                    seed2Features.push_back(*fIter);
                    env2 = newEnvelope2;
                }
            }
            RNode<M> *seedNode1 = new RNode<M>(env1);
            RNode<M> *seedNode2 = new RNode<M>(env2);
            for (auto fIter = seed1Features.begin(); fIter !=
seed1Features.end(); ++fIter)
                seedNode1->add(*fIter);
            for (auto fIter = seed2Features.begin(); fIter !=
seed2Features.end(); ++fIter)
                seedNode2->add(*fIter);
            leafNodes.erase(markIter);
            leafNodes.push_back(seedNode1);
```

```
            leafNodes.push_back(seedNode2);
        }
    }

    std::vector<RNode<M> *> childNodes;
    std::vector<RNode<M> *> parentNodes;
    RNode<M> *firstNode;
    std::vector<std::vector<RNode<M>>> treeNodes;
    for (int i = 0; i < leafNodes.size(); i++)
        childNodes.push_back(leafNodes[i]);
    bool flag = true;
    while (flag)
    {
        // 其余节点获取
        std::vector<RNode<M> *>::iterator markIter;
        firstNode = new RNode<M>(childNodes[0]->getEnvelope());
        // RNode<M> firstNode(childNodes[0].getEnvelope());
        firstNode->add(childNodes[0]);
        parentNodes.push_back(firstNode);
        for (auto childIter = childNodes.begin() + 1; childIter !=
childNodes.end(); ++childIter)
        {
            Envelope leafEnvelope = (*childIter)->getEnvelope(), markEnvelope;
            double minAddArea = 1000000;
            for (auto parentIter = parentNodes.begin(); parentIter !=
parentNodes.end(); ++parentIter)
            {
                Envelope parentEnvelope = (*parentIter)->getEnvelope();
                double minX = std::min(leafEnvelope.getMinX(),
parentEnvelope.getMinX());
                double minY = std::min(leafEnvelope.getMinY(),
parentEnvelope.getMinY());
                double maxX = std::max(leafEnvelope.getMaxX(),
parentEnvelope.getMaxX());
                double maxY = std::max(leafEnvelope.getMaxY(),
parentEnvelope.getMaxY());
                Envelope newEnvelope = Envelope(minX, maxX, minY, maxY);
                double addArea = newEnvelope.getArea() -
parentEnvelope.getArea();
                if (minAddArea > addArea)
                {
                    minAddArea = addArea;
                    markIter = parentIter;
                    markEnvelope = newEnvelope;
                }
            }
            if ((*markIter)->getChildNum() < M)
            {
                (*markIter)->add(*childIter);
                (*markIter)->setEnvelope(markEnvelope);
            }
            // 二次分裂
            else
            {
                std::vector<RNode<M>> allNodes;
```

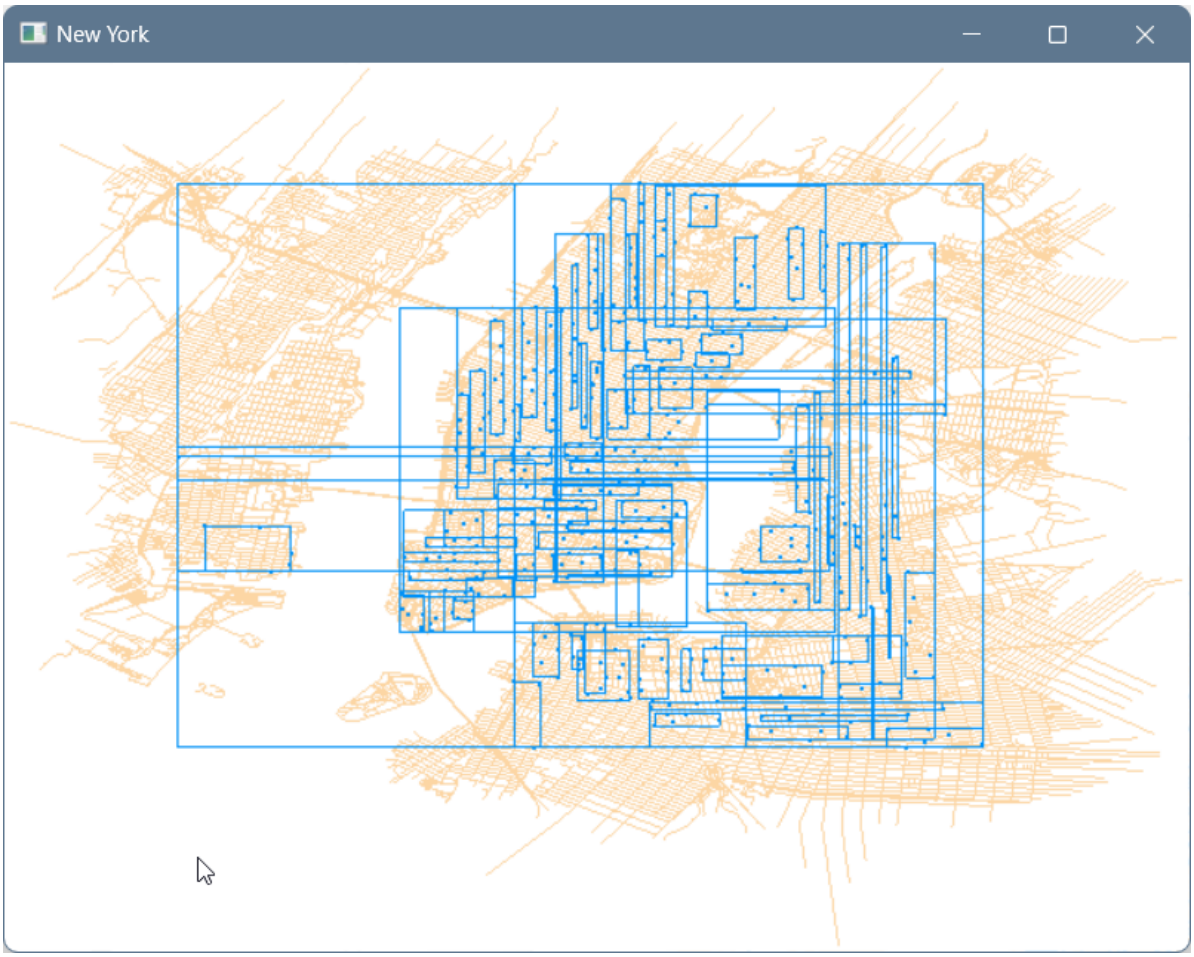
```
                std::vector<RNode<M> *> allNodesPointers;
                for (int i = 0; i < (*markIter)->getChildNum(); i++)
                    allNodesPointers.push_back((*markIter)->getChildNode(i));
                allNodesPointers.push_back(*childIter);
                std::vector<RNode<M>>::iterator seed1Iter, seed2Iter;
                RNode<M> *seed1Pointer(nullptr), *seed2Pointer(nullptr);

                double left = 1000000, right = -1000000;
                for (int i = 0; i < allNodesPointers.size(); i++)
                {
                    if (left > allNodesPointers[i]->getEnvelope().getMinX() &&
allNodesPointers[i] != seed2Pointer)
                    {
                        left = allNodesPointers[i]->getEnvelope().getMinX();
                        seed1Pointer = allNodesPointers[i];
                    }
                    if (right < allNodesPointers[i]->getEnvelope().getMaxX() &&
allNodesPointers[i] != seed1Pointer)
                    {
                        right = allNodesPointers[i]->getEnvelope().getMaxX();
                        seed2Pointer = allNodesPointers[i];
                    }
                }
                RNode<M> *seed1 = new RNode<M>(seed1Pointer->getEnvelope()),
*seed2 = new RNode<M>(seed2Pointer->getEnvelope());
                seed1->add(seed1Pointer);
                seed2->add(seed2Pointer);
                Envelope env1 = seed1->getEnvelope(), env2 = seed2-
>getEnvelope();

                for (int i = 0; i < allNodesPointers.size(); i++)
                {
                    RNode<M> *nodePointer = allNodesPointers[i];
                    if (nodePointer == seed1Pointer)
                        continue;
                    if (nodePointer == seed2Pointer)
                        continue;
                    double addArea1, addArea2;
                    double minX = std::min(nodePointer->getEnvelope().getMinX(),
env1.getMinX());
                    double minY = std::min(nodePointer->getEnvelope().getMinY(),
env1.getMinY());
                    double maxX = std::max(nodePointer->getEnvelope().getMaxX(),
env1.getMaxX());
                    double maxY = std::max(nodePointer->getEnvelope().getMaxY(),
env1.getMaxY());
                    Envelope newEnvelope1, newEnvelope2;
                    newEnvelope1 = Envelope(minX, maxX, minY, maxY);
                    addArea1 = newEnvelope1.getArea() - env1.getArea();
                    minX = std::min(nodePointer->getEnvelope().getMinX(),
env2.getMinX());
                    minY = std::min(nodePointer->getEnvelope().getMinY(),
env2.getMinY());
                    maxX = std::max(nodePointer->getEnvelope().getMaxX(),
env2.getMaxX());
```

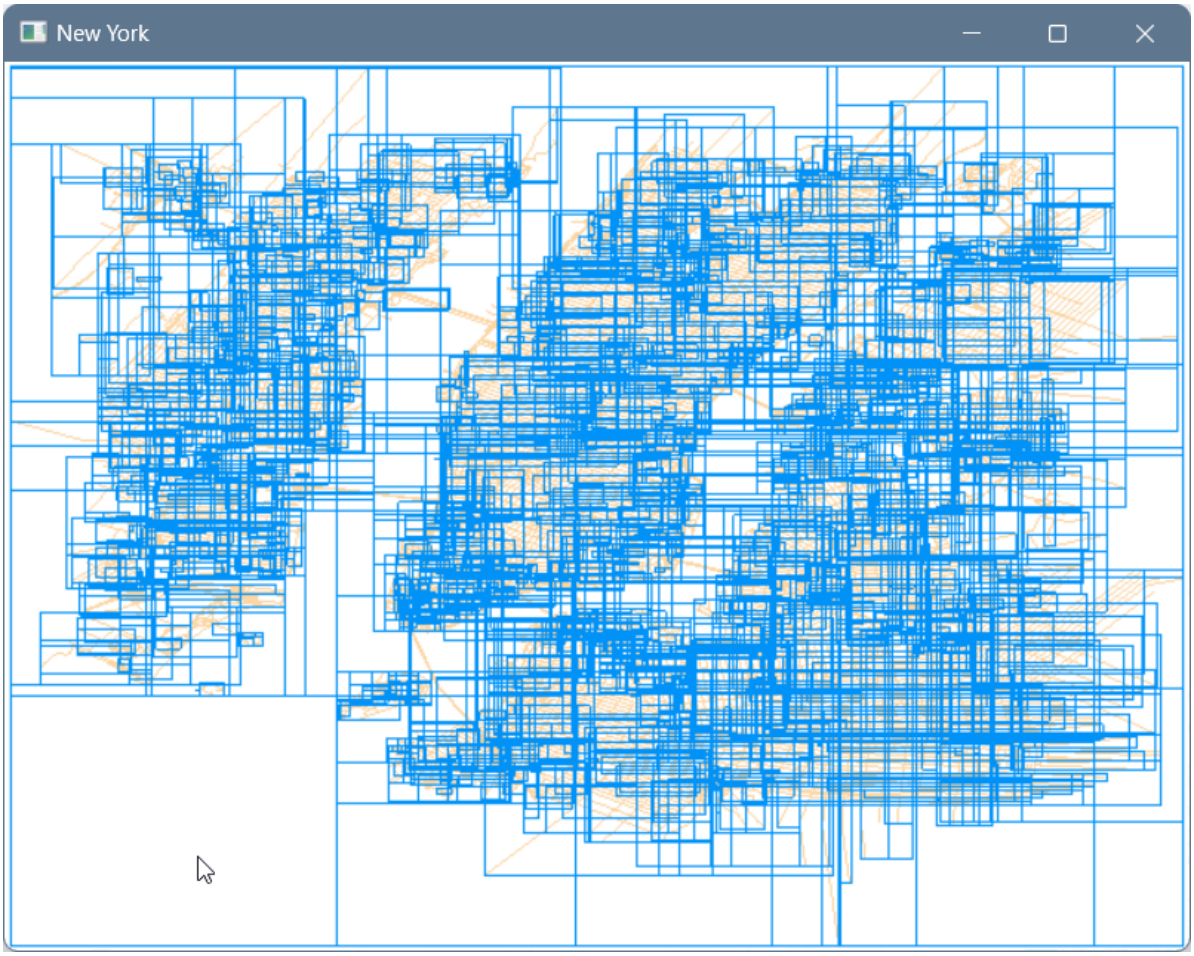
```
                    maxY = std::max(nodePointer->getEnvelope().getMaxY(),
env2.getMaxY());
                    newEnvelope2 = Envelope(minX, maxX, minY, maxY);
                    addArea2 = newEnvelope2.getArea() - env2.getArea();
                    if (addArea1 < addArea2)
                    {
                        seed1->add(nodePointer);
                        env1 = newEnvelope1;
                        seed1->setEnvelope(env1);
                    }
                    else
                    {
                        seed2->add(nodePointer);
                        env2 = newEnvelope2;
                        seed2->setEnvelope(env2);
                    }
                }
                parentNodes.erase(markIter);
                parentNodes.push_back(seed1);
                parentNodes.push_back(seed2);
            }
        }
        if (parentNodes.size() <= M)
        {
            flag = false;
            Envelope env = parentNodes[0]->getEnvelope();
            for (int i = 1; i < parentNodes.size(); i++)
            {
                double minX = std::min(parentNodes[i]->getEnvelope().getMinX(),
env.getMinX());
                double minY = std::min(parentNodes[i]->getEnvelope().getMinY(),
env.getMinY());
                double maxX = std::max(parentNodes[i]->getEnvelope().getMaxX(),
env.getMaxX());
                double maxY = std::max(parentNodes[i]->getEnvelope().getMaxY(),
env.getMaxY());
                env = Envelope(minX, maxX, minY, maxY);
            }
            root = new RNode<M>(env);
            bbox = env;
            for (int i = 0; i < parentNodes.size(); i++)
                root->add(parentNodes[i]);
        }
        else
        {
            childNodes = parentNodes;
            parentNodes.clear();
        }
    }
    return true;
}
```

测试

```
**********Start**********
测试4: RTree Construction
Case 1: Your answer is height: 4, interiorNum: 20, leafNum: 90. One possible
answer is height: 4, interiorNum: 18, leafNum: 86
Case 2: Your answer is height: 6, interiorNum: 856, leafNum: 2897. One possible
answer is height: 6, interiorNum: 484, leafNum: 2305
**********End**********
```

此处修改了R-树包围盒显示方式显示出了所有节点的包围盒，而非只显示叶节点的包围盒。





分析

可以看到虽然创建的结果与给出的答案并不完全一致，但大体上深度和节点数量相似。我创建的R-树节点数量较possible answer多，推测是只使用增加面积最小这一约束条件进行插入，没有考虑节点中储存的子节点数量。

2 区域查询

原理

通过递归查询，逐层查询与区域rect相交的节点，直到查询到叶节点，获取与rect相交的Feature作为候选Feature。此处相交只使用包围盒判断，作为粗查询。在获取所有候选Feature后，进行精查询。

实现

- rangeQuery粗查询

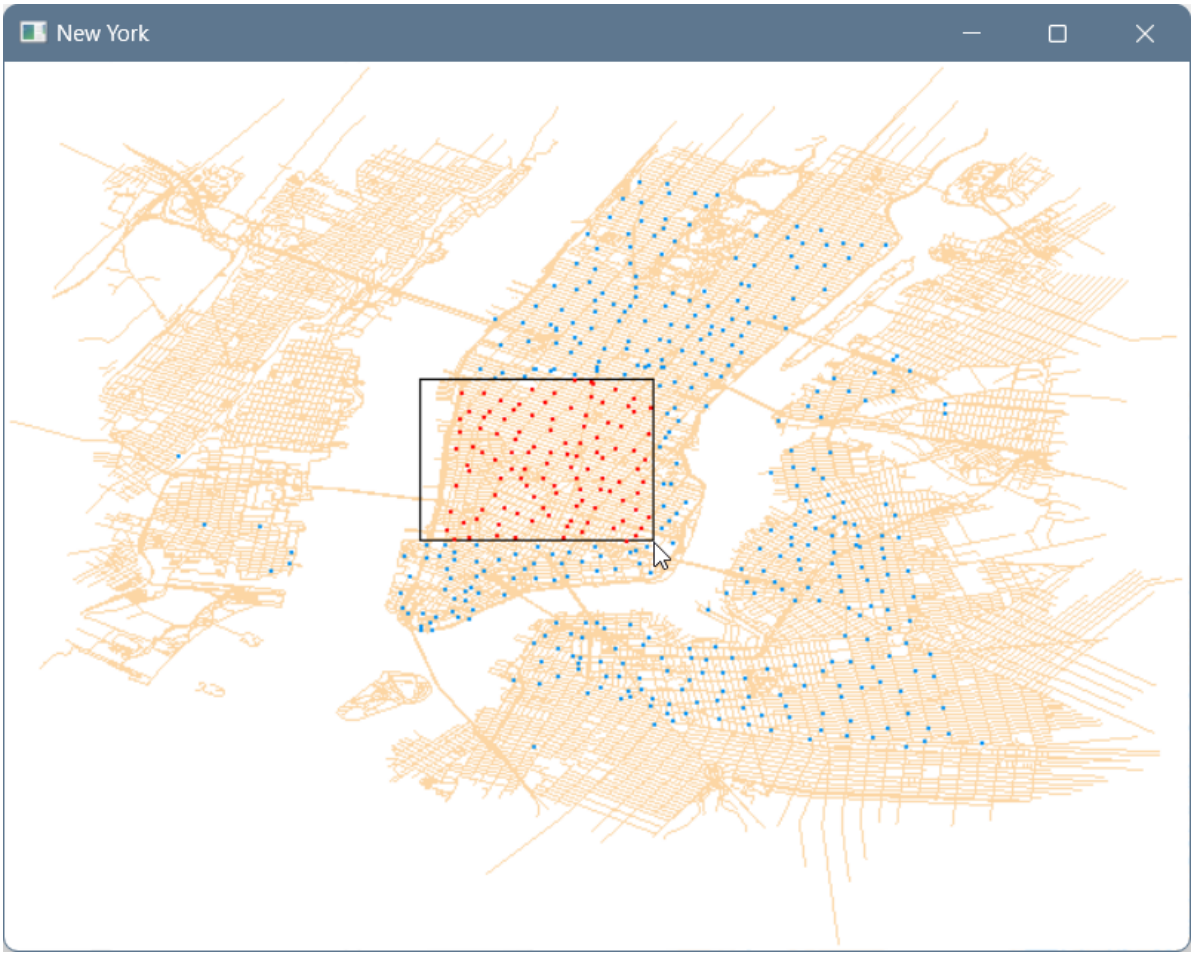
```
void rangeQuery(const Envelope &rect, std::vector<Feature> &features)
{
    // TODO
    int h = countHeight(0);
    std::vector<RNode<M> *> nodes, tmpNodes;
    for (int i = 0; i < getChildNum(); i++)
        nodes.push_back(getChildNode(i));
    for (int i = 0; i < h - 2; i++)
    {
        for (int j = 0; j < nodes.size(); j++)
        {
            if ((nodes[j]->getEnvelope()).intersect(rect))
```

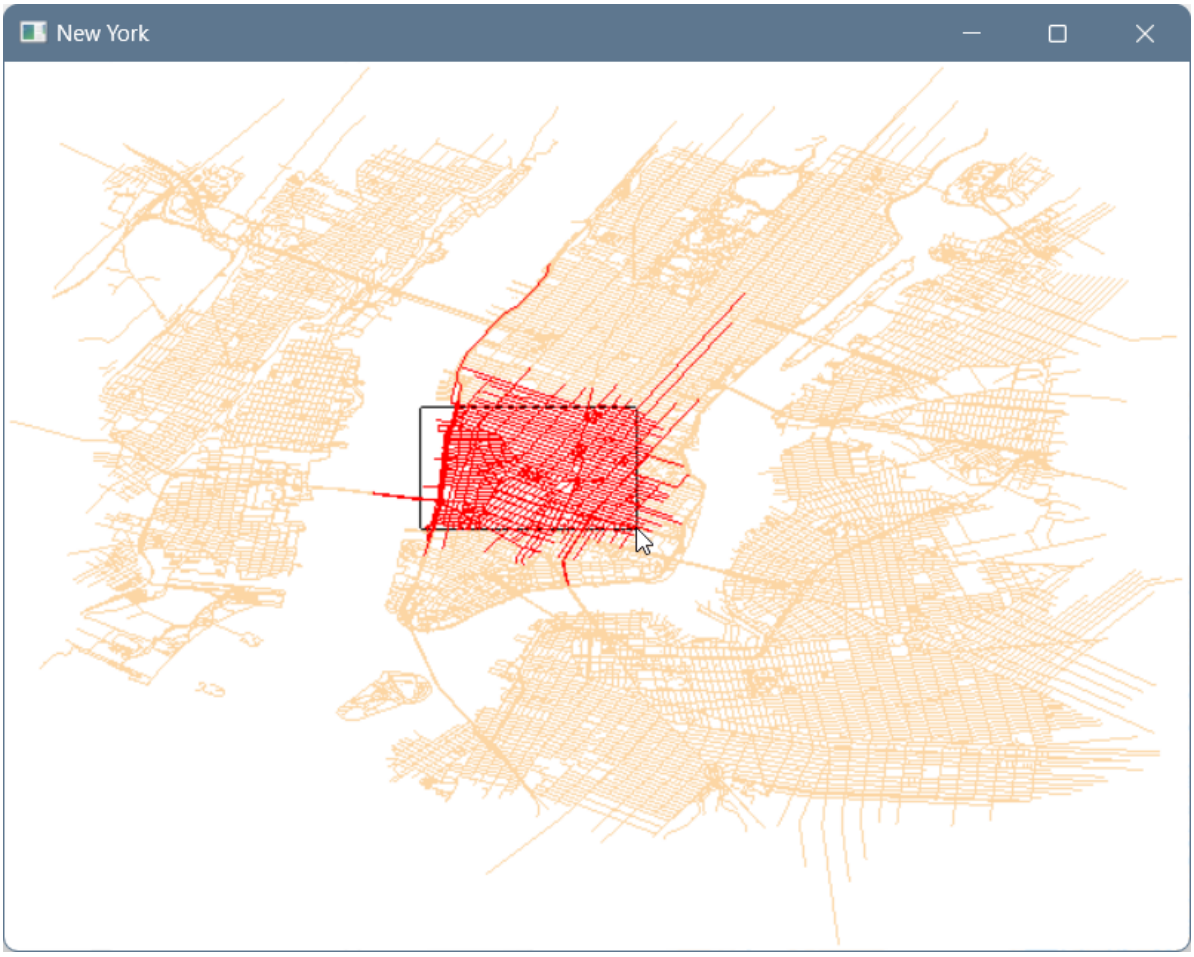
```
                for (int k = 0; k < nodes[j]->getChildNum(); k++)
                    tmpNodes.push_back(nodes[j]->getChildNode(k));
        }
        nodes = tmpNodes;
        tmpNodes.clear();
    }
    for (int i = 0; i < nodes.size(); i++)
    {
        const std::vector<Feature> &allFeatures = nodes[i]->getFeatures();
        for (int j = 0; j < allFeatures.size(); j++)
            if (allFeatures[j].getEnvelope().intersect(rect))
                features.push_back(allFeatures[j]);
    }
}
```

- rangeQuery精查询

```
void rangeQuery()
{
    vector<hw6::Feature> candidateFeatures;

    // filter step (使用四叉树获得查询区域和几何特征包围盒相交的候选集)
    if (mode == RANGEPOINT)
        pointTree->rangeQuery(selectedRect, candidateFeatures);
    else if (mode == RANGELINE)
        roadTree->rangeQuery(selectedRect, candidateFeatures);
    // refine step (精确判断时，需要去重，避免查询区域和几何对象的重复计算)
    // TODO
    if (mode == RANGEPOINT)
        selectedFeatures = candidateFeatures;
    else if (mode == RANGELINE)
    {
        for (auto it = candidateFeatures.begin(); it != candidateFeatures.end();
it++)
        {
            if (it->getGeom()->intersects(selectedRect))
                selectedFeatures.push_back(*it);
        }
    }
}
```

测试


New York


New York

分析

1. 根据几何关系可知，rangeQuery结果是正确的。
2. 使用不同大小rect测试，查询时都较为流畅。

3 最邻近几何特征查询

原理

1. 邻近查询查询主要分为两步。首先通过pointInLeafNode查询输入点所在的叶子节点（即输入点在结点的包围盒中），需要注意的是，输入点不一定在某个叶节点中，因此此步找出输入点所在最深的节点。然后计算输入点与该最深节点的各几何特征包围盒的最大距离的最小值，构造查询区域(x – minDist, x + minDist, y – minDist, y + minDist)进行区域查询，得到结果作为候选Feature，而后进行精确判断即可。
2. pointInLeafNode采用深度优先搜索，寻找查询点所在最深的节点。

实现

- NNQuery粗查询

```
virtual bool NNQuery(double x, double y,
                     std::vector<Feature> &features) override
{
    features.clear();
    // TODO
    RNode<M> *node = pointInLeafNode(x, y);
    if (node->isLeafNode())
    {
        std::vector<Feature> newFeatures = node->getFeatures();
        double dist = 1000000;
        for (auto it = newFeatures.begin(); it != newFeatures.end(); ++it)
        {
            Envelope env = it->getEnvelope();
            double d1 = env.getMinX() - x, d2 = env.getMinY() - y, d3 =
env.getMaxX() - x, d4 = env.getMaxY() - y;
            double dist1 = sqrt(d1 * d1 + d2 * d2);
            double dist2 = sqrt(d1 * d1 + d4 * d4);
            double dist3 = sqrt(d3 * d3 + d2 * d2);
            double dist4 = sqrt(d3 * d3 + d4 * d4);
            dist = std::min(std::max(std::max(dist1, dist2), std::max(dist3,
dist4)), dist);
        }
        const Envelope rect = Envelope(x - dist, x + dist, y - dist, y + dist);
        rangeQuery(rect, features);
    }
    else
    {
        std::vector<RNode<M> *> childs;
        for (int i = 0; i < node->getChildNum(); i++)
            childs.push_back(node->getChildNode(i));
        double dist = 1000000;
        for (auto it = childs.begin(); it != childs.end(); ++it)
        {
            RNode<M> *tmpNode = *it;
            Envelope env = tmpNode->getEnvelope();
```

```
            double d1 = env.getMinX() - x, d2 = env.getMinY() - y, d3 =
env.getMaxX() - x, d4 = env.getMaxY() - y;
            double dist1 = sqrt(d1 * d1 + d2 * d2);
            double dist2 = sqrt(d1 * d1 + d4 * d4);
            double dist3 = sqrt(d3 * d3 + d2 * d2);
            double dist4 = sqrt(d3 * d3 + d4 * d4);
            dist = std::min(std::max(std::max(dist1, dist2), std::max(dist3,
dist4)), dist);
        }
        const Envelope rect = Envelope(x - dist, x + dist, y - dist, y + dist);
        rangeQuery(rect, features);
    }
    return true;
}
```

- NNQuery精查询

```
void NNQuery(hw6::Point p)
{
    vector<hw6::Feature> candidateFeatures;

    // filter step (使用四叉树获得距离较近的几何特征候选集)
    if (mode == NNPOINT)
        pointTree->NNQuery(p.getX(), p.getY(), candidateFeatures);
    else if (mode == NNLINE)
        roadTree->NNQuery(p.getX(), p.getY(), candidateFeatures);

    // refine step (精确计算查询点与几何对象的距离)
    // TODO
    double dist = 1000000;
    for (auto it = candidateFeatures.begin(); it != candidateFeatures.end();
++it)
    {
        double tmpDist = it->getGeom()->distance(&p);
        if (tmpDist < dist && tmpDist)
        {
            dist = tmpDist;
            nearestFeature = *it;
        }
    }
}
```

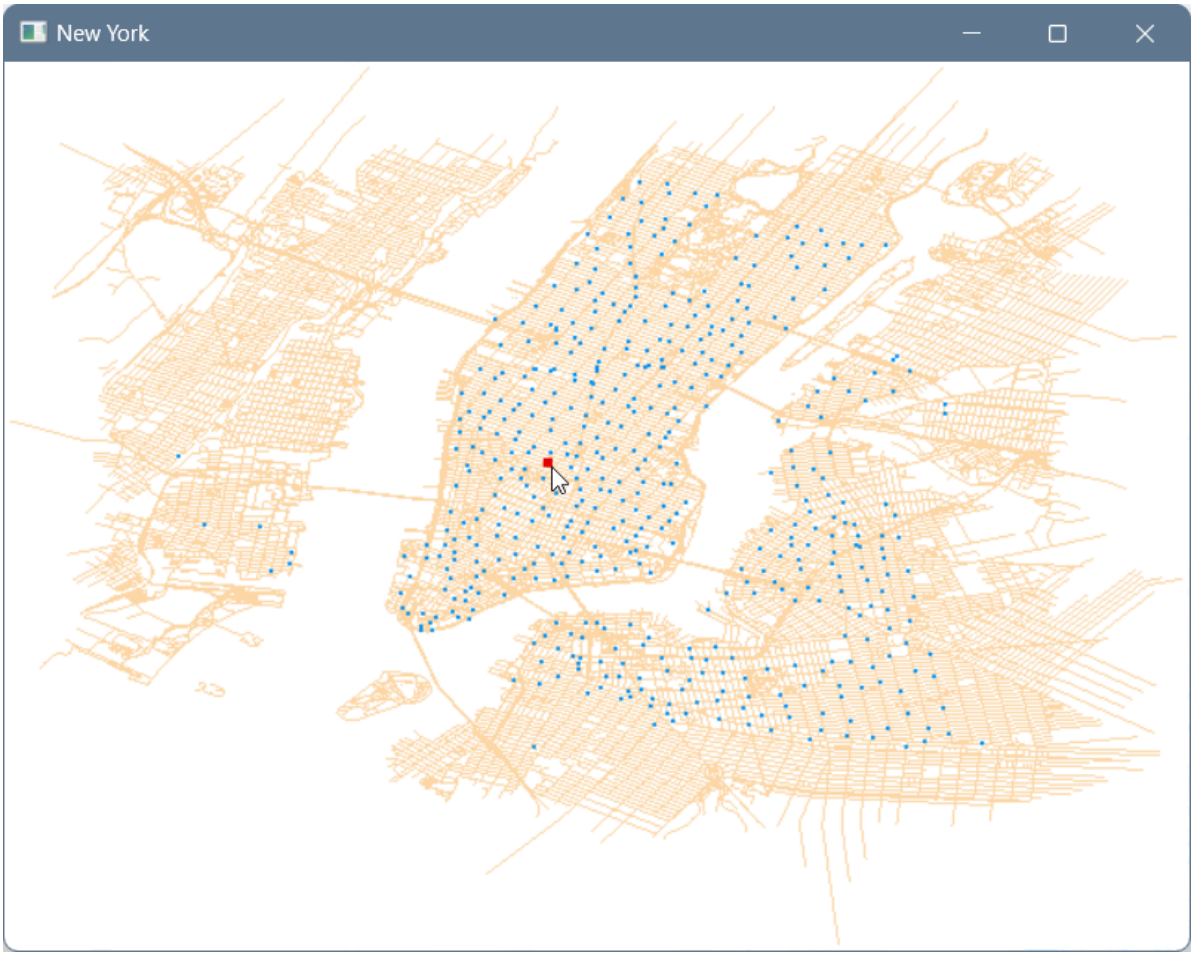
- pointInLeafNode实现

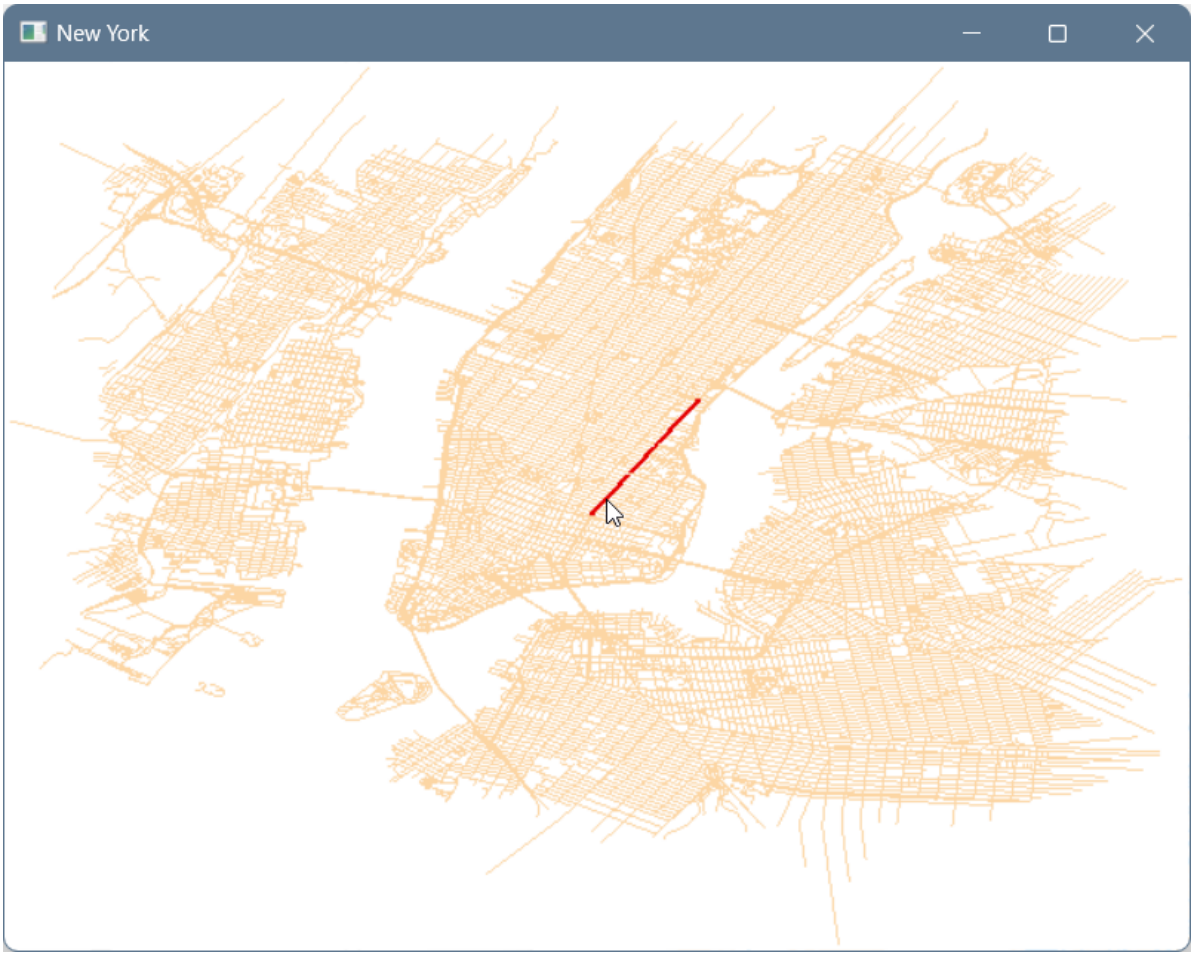
```
RNode<M> *pointInLeafNode(double x, double y)
{
    // TODO
    RNode<M> *node = this;
    int h = countHeight(0);
    depth = 0;
    res = nullptr;
    dfs(x, y, node, 1);
    return res;
}
```

- 深度优先搜索实现

```
int depth;
RNode<M>* res;
void dfs(double x, double y, RNode<M>* node, int d)
{
    if (node->isLeafNode())
        res = node;
    bool flag = true;
    for (int i = 0; i < node->getChildNum(); i++)
    {
        RNode<M>* tmpNode = node->getChildNode(i);
        if(tmpNode->getEnvelope().contain(x, y))
        {
            flag = false;
            dfs(x, y, tmpNode, d + 1);
        }
    }
    if (flag && d >= depth)
    {
        depth = d;
        res = node;
    }
}
```

测试



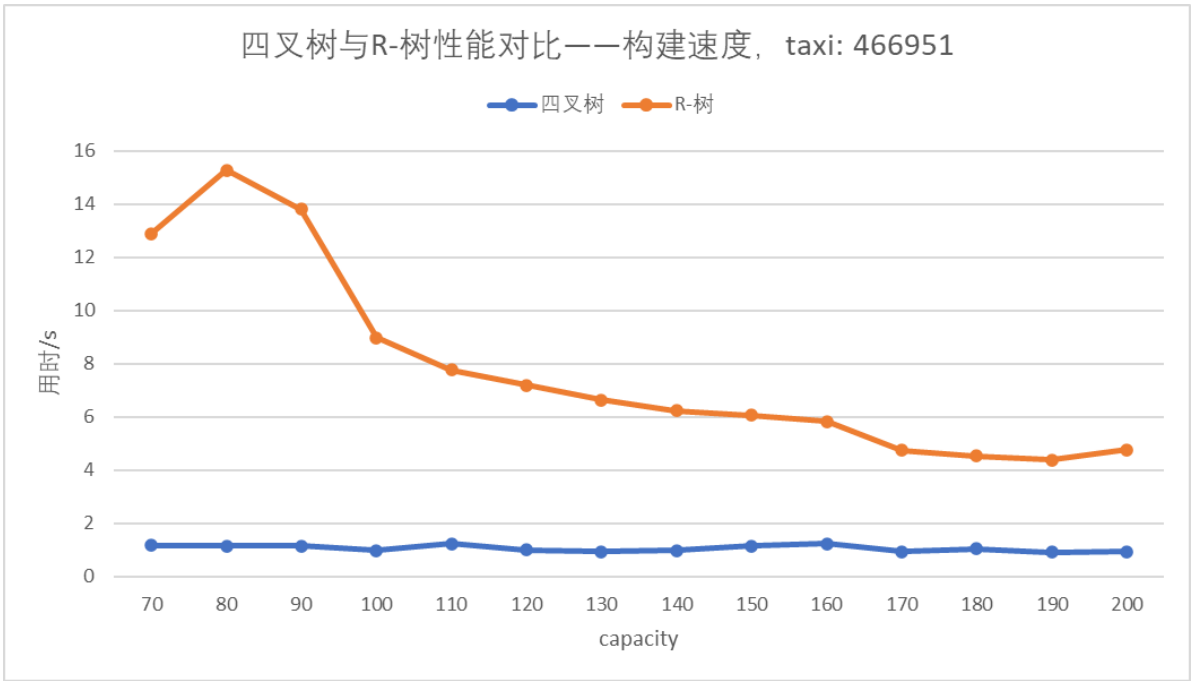


分析

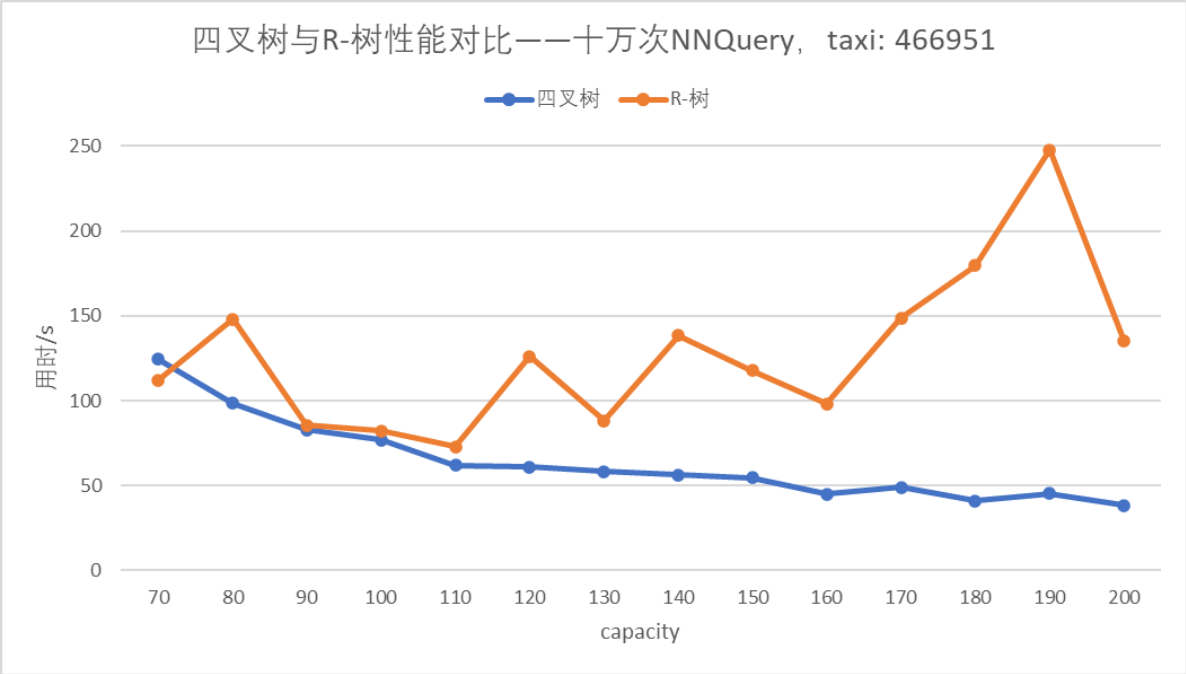
1. 根据几何关系可知，NNQuery结果是正确的。
2. 在不同区域进行测试，查询时都较为流畅。

五、R树与四叉树性能分析

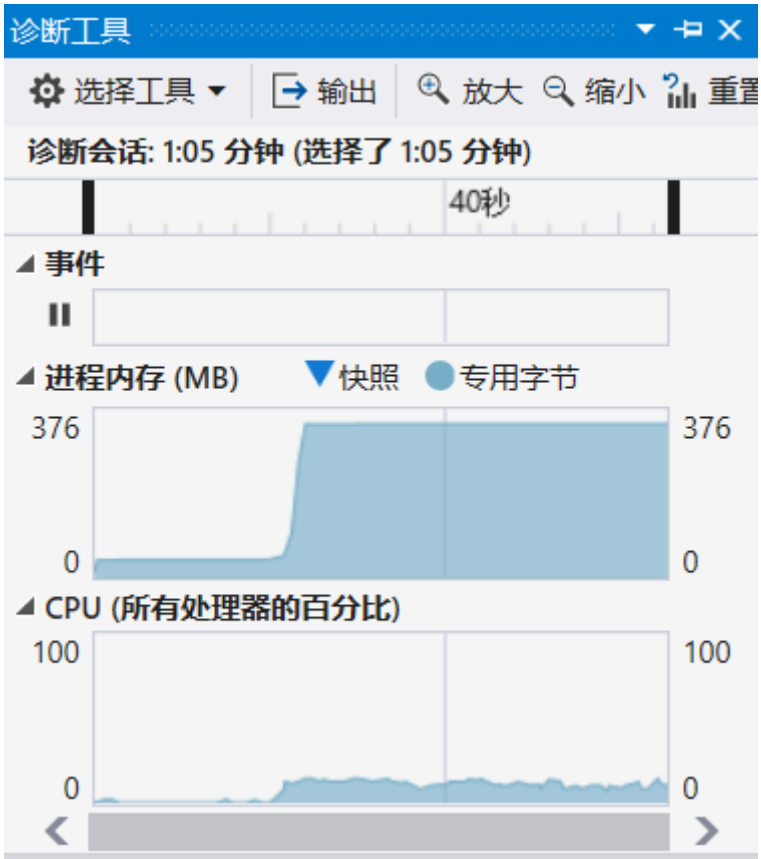
四叉树和R-树构建速度分析，taxi数据

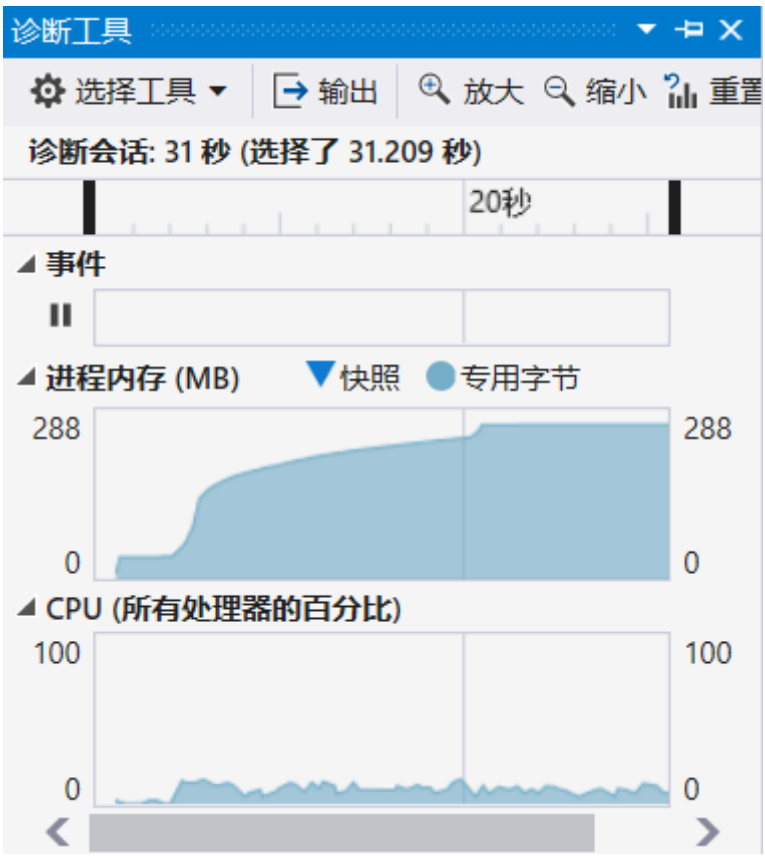


四叉树和R-树NNQuery速度分析，taxi数据

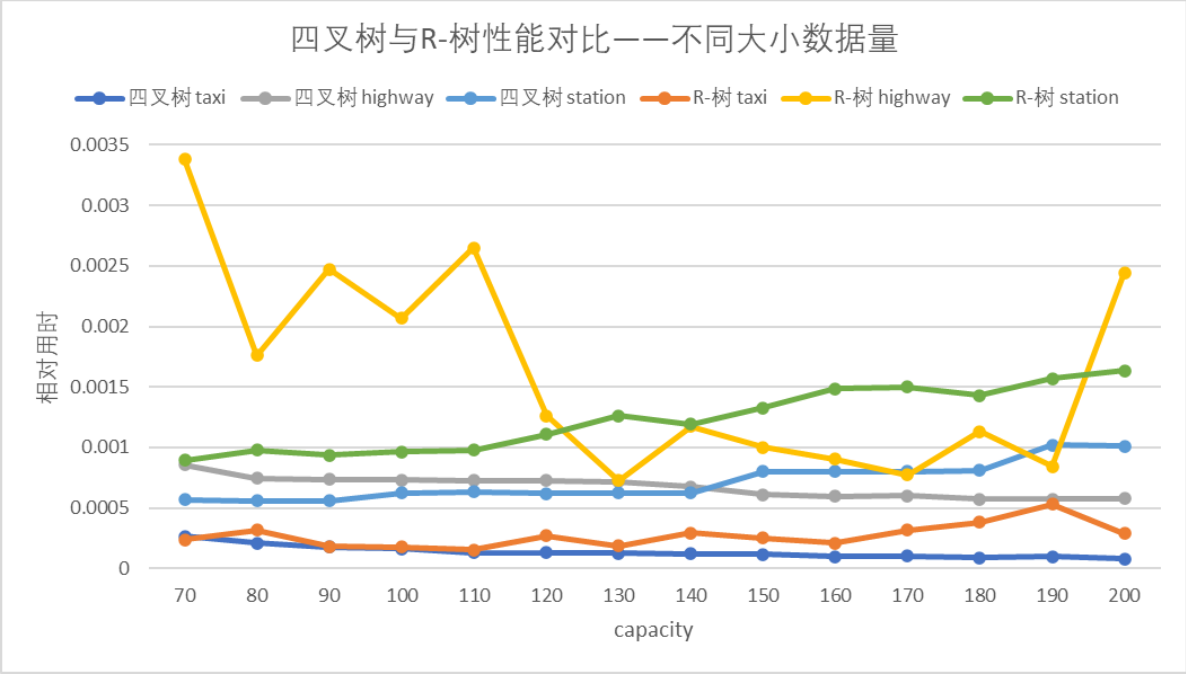


四叉树和R-树运行内存变化





四叉树和R-树在不同数据下的性能差异



分析

根据上述结果，可得两点结论：

1. R-树具有不稳定性，从其构建和NNQuery时间就可以看出。
2. R-树深度较四叉树小，构建时间较四叉树长，R-树内存增长的斜率较小就佐证这一点。
3. 在查询点数据时，R-树性能较为稳定，且其性能普遍低于四叉树。在查询线数据时，R-树性能较为不稳定。

六、成员分工情况

胡宇森：空间数据类型实现、R-树创建与查询、R-树性能分析、报告相应部分

魏辰：包围盒类函数实现、四叉树创建与查询、四叉树性能分析、报告相应部分、PPT展示