Voronoi 图的分治法实现与演示

一、问题描述：

分治法，顾名思义，分而治之，将一个大问题分解为若干个相似的子问题，再递归对子问题分别求解，再把子问题分成若干个更小的子问题，再将子问题的解合并，得到最终问题的解。分治法在计算几何领域中应用广泛，例如求平面上的最近点对，构造凸包等等。分治法的思想十分朴素，其中最重要的部分是分解和合并部分，这两部分是理解分治法的核心。

Voronoi 图在计算几何学科中有重要地位，理解用分治法求 Voronoi 图的分解和合并部分对我们理解计算几何算法有很大帮助。为此，我们计划构建一个演示系统来展示用分治法求 Voronoi 图的过程，系统界面上可以提供干净且易于理解的信息，增强演示的效果，从而帮助同学们理解这部分算法。

二、算法说明：

算法中最关键的一步是如何对两个被竖直分离的 Voronoi 图进行合并：

（1）计算 L 和 R 的凸包，记为 conv(L) 和 conv(R)。计算两个凸包的公共切线，记相应的上切点为 l 和 r。

（2）计算 l 和 r 中垂线，记为 b。

（3）分别计算 b 与 Cell(l)和 Cell(r)的交点。

若先与 Cell(l)相交，更新 l 为 conv(L)中顺时针顺序的下一个点，并删除 Vor(L)在 b 右侧的线段。

若先与 Cell(r)相交，更新 r 为 conv(R)中逆时针顺序的下一个点，并删除 Vor(R)在 b 左侧的线段。

（4）将 b 与 Vor(L)，Vor(R)进行关联。

（5）重复步骤 2-4，直到 l，r 分别为下切点。

整个算法以递归的方式进行：

（1）将点集 P 按 x 轴进行排序，并按 x 轴中值点将 P 分成 L 和 R 两个子点集。

（2）分别计算 L 和 R 的 Voronoi 图，记为 Vor(L)和 Vor(R)。

（3）将 Vor(L)和 Vor(R)进行融合得到 Vor(P)。

由于细节太多，详情请见代码。

三、系统演示说明：

（1）程序设计思路：

希望实现一个带界面交互的 Voronoi 图分治算法演示程序，由用户输入点集，将计算的结果在主界面的左边表示。特别的是，主界面的右边展示一棵满二叉树，对应分治法的每个步骤，分治算法的核心是合并部分，通过合并左边的 Voronoi 图和右边的 Voronoi 图得到整个 Voronoi 图，对应满二叉树中一个中间节点的结果是合并左孩子节点和右孩子节点的结果。但是由于时间原因，这一版只实现了部分，最后并没有采用。

整个系统分为前端和后端，前端主要输入多个点，后端运行分治算法求 Voronoi 图，系统有两个模式，一个模式可以直接输出 Voronoi 图的结果，另一个模式可以一步步执行展示分治算法求 Voronoi 图的过程，可以在过程中切换。

（2）主界面展示：

主界面主要以一个大窗口进行呈现，将会展示分治法的执行流程，包括每一

层处理和合并的流程。整个界面以左右两端为开端进行处理，并最终在界面的中部完成最后的合并。

（3）程序功能说明：

主界面部分即画布，可以在界面上通过鼠标点击某一位置的点，在界面最底端的状态栏显示对应的位置，点击下一步菜单项展示 Voronoi 图的合并过程。详细来说，程序提供了这些功能：

①点的生成方式。为使用者提供手动输入和随机生成点的功能。

②手动输入点时，可以在系统界面的状态栏中显示点的位置。

③可以把点的位置保存成文件，下次仅需通过打开文件即可读入对应的点。

④算法演示过程中的下一步功能。可以通过点击“下一步”菜单，使得构造 Voronoi 图的过程一步步展示出来。

⑤重置功能。取消目前算法的播放，并清除界面内所有点。

⑥重新播放功能。取消目前算法的播放，并从当前界面内已有的点进行算法的演示。

⑦展示结果功能。可以直接展示 Voronoi 图的结果。

四、数据结构与模块设计：

1. 算法：

（1）数据结构：

```
class vertex
{
public:
    double x;
    double y;
    edge *inc;

    vertex (double init_x, double init_y)
    {
        x = init_x;
        y = init_y;
        inc = NULL;
    }
};
```

```
class edge
{
public:
    edge *twin; //对偶边
    edge *succ;
    edge *prev;
    face *inc;
    vertex *root; //起点vertex指针
    vertex *tgt;  //终点vertex指针
    int finite; // 0为线段，1表示终点为无穷远，2表示起点为无穷远，3表示直线
    vector<double> abc; //直线方程，ax+by+c=0

    edge(face *init_inc)
    {
        inc = init_inc;
        succ = NULL;
        prev = NULL;
        root = NULL;
        tgt = NULL;
        finite = 3;
    }
};
```

```
class face
{
public:
    double x;
    double y;
    edge *inc;  //围成面的所有边中的任何一个的指针

    face (double init_x, double init_y)
    {
        x = init_x;
        y = init_y;
        inc = NULL;
    }
};
```

（2）核心步骤：

```
DCEL VD_dac(int begin, int end)
{
    DCEL result;
    if (end - begin == 1)
    {
        result.faces.push_back(&faces[begin]);
        vector<output> output_dac = result.record_results();
        outputs.push_back(output_dac);
        return result;
    }
    if (end - begin == 2)
    {
        vector<double> line = calculate_vertical_line(faces[begin], faces[begin + 1]);
        vector<vertex*> vertices = calculate_vertical_line_vertex(faces[begin], faces[begin + 1]);
        result.faces.push_back(&faces[begin]);
        result.faces.push_back(&faces[begin + 1]);

        edge *e1 = new edge(result.faces[0]);
        edge *e2 = new edge(result.faces[1]);

        e1->abc = line;
        e1->root = vertices[0];
        e1->tgt = vertices[1];
        e1->prev = e1;
        e1->succ = e1;
        e1->twin = e2;
        e1->finite = 3;

        e2->abc = line;
        e2->root = vertices[1];
        e2->tgt = vertices[0];
        e2->prev = e2;
        e2->succ = e2;
        e2->twin = e1;
        e2->finite = 3;

        result.faces[0]->inc = e1;
        result.faces[1]->inc = e2;
        vector<output> output_dac = result.record_results();
        outputs.push_back(output_dac);
        return result;
    }
    int middle = (begin + end) / 2;
    DCEL left = VD_dac(begin, middle);
    DCEL right = VD_dac(middle, end);
    result = merge(left, right);
    vector<output> output_dac = result.record_results();
    outputs.push_back(output_dac);
    return result;
};
```

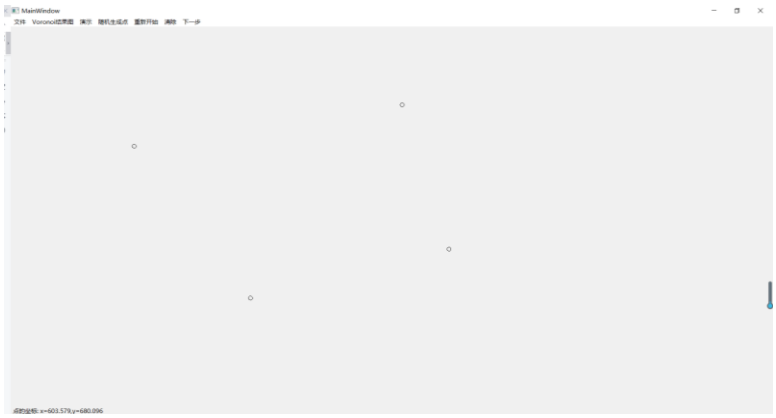
2. 系统界面：

（1）数据结构

变量	含义
left_faces_global_pf	左边 VD 包含点
right_faces_global_pf	右边 VD 包含点
last_points_global_lf	保存已经显示过的点
last_edges_global_lf	保存已经显示过的边
internal_chains_global_lf	分割链
left_edges_global_lf	左边 VD 的边
right_edges_global_lf	右边 VD 的边

（3）系统展示：

①任意选择点集。



②点击“演示”。

可以看到主界面第一步的结果，其中，红点表示分治算法中当前合并的左边 Voronoi 图涉及的点集，绿点表示分治算法中当前合并的右边 Voronoi 图涉及的点集，虚线表示二者的分割线。

③点击“下一步”，灰线表示已经合并了的 Voronoi 图，红线表示当前合并的左边 Voronoi 图分割好的边，绿线表示当前合并的右边 Voronoi 图分割好的边。

五、性能测试和结果分析：

我们对分治法进行了性能测试与复杂度分析。我们随机生成了点集大小分别为 16,32,64,128,256,512,1024 的 7 种数据对程序进行测试。并且对于每种数据我们测试了多次取平均值作为最终测试结果。结果如下表所示：

输入点数	算法运行时间(ms)
16	16
32	47
64	125
128	312
256	703
512	1593
1024	3610

分治法的理论时间复杂度为 O(nlgn)，该实验结果与理论相符合。

六、实验总结：

一开始低估了分治法的难度，没想到比 Delaunay 三角剖分还要麻烦。这个算法主要的难点在于如何得到分割线后如何将其以 DCEL 数据结构的方式与左右 VD 图联系起来以及如何进行删边操作。这其中涉及到大量非常细节的指针操作，并且存在许多的 corner cases。在 debug 的时候，我们借助了 python scipy.spatial 库中的 Voronoi 算法生成测试样例。（韦祎）

在系统演示部分，我们使用了 Qt Creator 实现。在实现的过程中发现 Qt QPainter 在绘图的时候消耗过多 CPU 资源，我们中间使用了多种方法来改进它，例如多线程，但是都不能真正解决问题，系统主界面容易卡死，最后采用了 QFuture 与 QtConcurrent 的方式解决。但是由于 QPainter 中每次都是重新构图，drawLine()方法消耗了不少 CPU 资源，所以最后系统直接输出 Voronoi 图的速度还可以，但是系统一步步演示的速度会比较慢。（宋彦晶）

本次实验我主要负责部分主要辅助功能的开发，以及协助一部分主界面的演示开发，在开发的过程中还是遇到不少问题，比如文件的存储方式以及如何改进界面的效果和效率，还是有了一定工作量的，但确实感觉到一开始选的组件可能就不太合适，导致了后续开发中出现较多的问题，希望在后续的工作中能更好地对项目进行改进。（谢康培）

七、参考资料：

[1] 计算几何课程讲义，邓俊辉

[2] Aurenhammer F . Voronoi diagrams—a survey of a fundamental geometric data structure[J]. ACM Computing Surveys, 1991, 23(3):345-405.

[3] Oishi Y , Sugihara K . Topology-Oriented Divide-and-Conquer Algorithm for Voronoi Diagrams[J]. Graphical Models and Image Processing, 1995, 57(4):303-314.

八、小组成员及分工：

宋彦晶 2020213967 系统界面部分

谢康培 2020214019 系统界面部分

韦祎 2019310973 实现 Voronoi 图算法部分