简单多边形的生成与内部最短路

贾越凯 徐瑞翔 许怡文

1. 问题描述

在计算几何中，求一个几何区域内的最优路径是一个基础问题，在机器人路径规划、地理信息系统 (GIS)、电路布线等领域上有广泛应用。另一方面，为了测试各类计算几何算法的正确性、鲁棒性和实际运行效率，需要生成大规模的满足一定约束的几何体，例如生成一个随机的简单多边形。因此，本文将针对以下两个问题分别进行研究：

1. 简单多边形生成(或平面点集的简单多边形化)：给定平面内一组点，构造一个简单多边形，使得多边形的顶点恰好就是点集中的点。
2. 简单多边形内部最短路：给定一个简单多边形，和多边形内部的两个点 s 和 t，求 s 到 t 的最短路，要求路径也在多边形内。

2. 算法说明

2.1 简单多边形生成

对于简单多边形生成，本项目主要实现了 [1] 中的几种多项式时间的启发式简单多边形生成算法：

1. 空间划分 (Space Partitioning)
2. 2-Opt Moves

此外，本项目还实现了几种非多项式时间的算法，用于比较与测试：

1. 排列法 (Permute & Reject)
2. 回溯法 (Backtracking)

排列法

排列法非常简单：给定一个初始的多边形顶点排列，判断是否是简单多边形，然后随机打乱顶点顺序，直到得到一个简单多边形。

对于简单多边形的判断，即判断多边形的 n 条边是否有交(如果只在端点处相连则不算)，可使用 $O(n \log n)$ 的 Bentley & Ottmann's 算法。但排列法本身支持 n 就比较小，使用 $O(n^2)$ 枚举即可。

本算法的期望时间复杂度为 $O(n!/T \cdot n^2)$，其中 T 是以给定点集为顶点能形成的简单多边形总数。

回溯法

为了测试几种多边形生成算法的随机性，本项目需要生成给定点集为顶点的所有简单多边形。相比于最简单的枚举顶点全排列的方法，本项目使用了基于回溯的搜索算法，一般情况能够更快。

算法假设已得到了一条简单多边形链，当前需要再加入一个顶点。此时需要选择一个未在多边形链中的点，且它与前一个顶点构成的边不与其他边相交。找到所有符合要求的下一个点，逐个进行尝试，并延长链的长度，直到最后链的长度达到了 n，再判断将链首尾相接后是否与其他边有交。

该算法能求出以给定点集为顶点的所有简单多边形，时间复杂度最坏为 $O(n^2 \cdot n!)$。但是由于每次尝试延长多边形链时，符合要求的下一个点不会很多，所以情况数会大大减少。

2-Opt Moves

2-opt Moves 算法是一种基于随机排列解除相交线的算法，该算法首先基于输入的点随机生成一种排列方式，将这些点按顺序连接成为一个多边形。然后该算法两两枚举当前多边形的边，判断这对边是否为相交边，并记录所有相交的边。之后，算法会随机选择一对相交的边，并采用 2-opt move 操作来解除相交边 (relink)，即对于每一对相交边 $e_1 = (v_i, v_{i+1})$ 和 $e_2 = (v_j, v_{j+1})$，2-opt move操作会将其进行重连接，如下图所示。

当完成解相交之后，需要重新判断新形成的边与其他边的相交关系，更新相交关系数据结构。如此反复直到最终形成的多边形不再存在相交边，即得到一个简单多边形。根据Van Leeuwen 和 Schoone 等人[1]的证明，该算法最多需要 $O(n^3)$ 次 2-opt moves 操作就会形成一个简单多边形，所以算法复杂度为 $O(n^4)$。

空间划分

空间划分算法是一种基于分治的算法。首先对于给定点集，先选取点集中两点，用经过这两点的直线将点集分为两部分(这两部分点集的凸包一定不相交)。然后算法会分别对两点集求一条以初始两点为起点终点的简单多边形链，最后将两简单多边形链拼起来即得到简单多边形(由于它们的凸包不相交，直接连起来后一定是还简单的)。

然后，我们考虑给定起点 F 和终点 L，如何构造一条经过所有点的多边形链。还是采用分割法，我们随机选择点集中一个点 T，并随机选一条经过 T 且与线段FL 相交的直线，以此为依据继续分割点集。由于分出的两点集的凸包还是不相交，我们可以再递归求经过两点集的多边形链，并将它们拼起来。如果点集只剩两个点，则直接将它们相连，结束递归。

在具体实现时，我们无需维护具体的多边形链，而是将该算法看做一个对多边形顶点的排序过程，每次递归是对一段区间进行操作，来得到一个合法的多边形顶点次序。以某一直线分割点集的过程非常类似快速排序，只需将在直线左侧的点放到点 T 的左边，在直线右侧的点放到点 T 的右边，这样分割的过程就是 $O(n)$ 的。而且这样一来也自然地实现了多边形链的合并。

此外，该算法也需要注意多点共线的情况，我们只需保证在递归开始前，选择的初始两点不与其他点共线即可。

虽然在最差情况下，算法会调用 $O(n)$ 次递归，导致 $O(n^2)$ 的时间复杂度。然而，期望的意义上来讲，复杂度可以大致达到 $O(n \log n)$。

2.2 简单多边形三角剖分

为了解决简单多边形内部最短路问题，目前的绝大多数算法都需要先将多边形进行三角剖分。我们实现了 $O(n \log n)$ 的基于单调多边形分解的三角剖分算法。该算法的基本原理是先用平面扫描算法，用 $O(n \log n)$ 的时间将简单多边形分解为若干个单调多边形。然后可用 $O(n)$ 的时间对每个单调多边形进行三角剖分。

单调多边形分解

如果一个简单多边形没有内部尖顶 (interior cusp)，它就是单调多边形，内部尖顶又可分为 stalactite 和 stalagmite 两类，分别如下所示：

本算法通过在每个内部尖顶处引入对角线，来消除内部尖顶，从而将简单多边形分解为若干个单调多边形。为了保证对角线之间不会相交，新引入的对角线有个限制：如果我们从多边形每个顶点触发向左右发射水平线，直到碰到多边形的边，从而将原多边形分为若干个梯形，则对角线必须在一个梯形的内部，且一个梯形内最多只能有一条对角线。

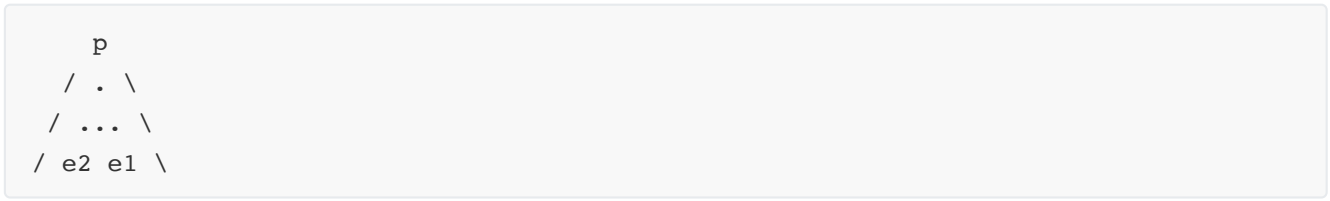
因此算法首先要对简单多边形进行梯形剖分，可用以下基于平面扫描的算法在 $O(n \log n)$ 的时间内完成：首先对所有顶点按纵坐标从大到小排序，简单起见，先假设没有两个顶点有相同的纵坐标(在下文会专门讨论此类情况)。然后依次枚举每个顶点，以当前顶点纵坐标作为当前扫描线的纵坐标，并维护当前扫描线穿过的所有梯形，作为扫描线状态。我们不需要关心梯形的上下边，它们一定与某时刻的扫描线重合，所以只需记录梯形的左右边。同时我们需要让这些梯形保持有序，可将当前扫描线与梯形左侧边交点的横坐标作为排序的依据。然后需要根据当前顶点的与相邻顶点的位置关系，往扫描线状态中插入、删除梯形，或查找扫描线上某个点所在的梯形，因此可用平衡二叉树维护这些梯形，每次以 $O(\log n)$ 的时间完成操作。

在进行梯形剖分的同时，我们还需要引入对角线来消除 stalactite 和 stalagmite。当遇到一个 stalagmite 时，其一定会在某个梯形的底边上，而相应的，该梯形的顶边上也必定有一个多边形的顶点，我们只需将它们相连即可。因此，我们还需要对每个梯形记录其顶边上的属于原多边形的顶点，称其为一个 "helper"。但是此方法只能消除所有 stalagmite，而不能消除所有 stalactite，因为 stalactite 相当于反过来的 stalagmite，有些时候可能需要作为梯形顶边的点，向下找梯形底边上的非 stalagmite 点连边。对于这种情况，我们只需在处理多边形顶点时，对当前梯形判断其 helper 是否是一个 stalactite，且还未被连过边，如果是就将其和当前顶点连边。

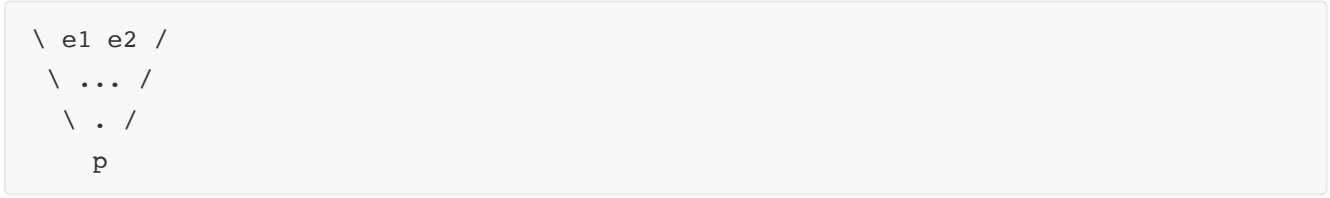
最后，对于添加对角线操作，可使用 DCEL 结构，但为了维护边的顺序，需要为每个 helper 同时记录它在 DCEL 结构中的上一条边是什么。这样就可以用 O(1) 的时间维护了。

以下列举了正在处理多边形顶点为 p，其上一条、下一条边分别为 e1，e2 时，所有可能遇到的情况：

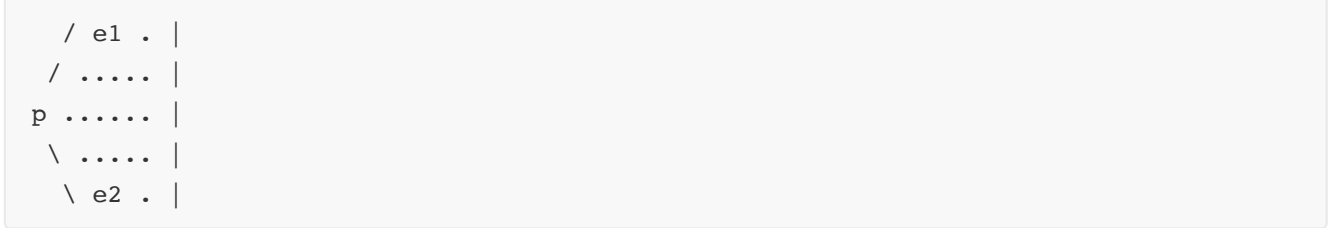
1. 起始点：插入一个新的梯形，将 helper 设为 p。



2. 结束点：删除梯形。



3. 左相邻：查找 p 所在的梯形，修改其左侧边，并将 p 设为 helper。



4. 右相邻：查找 p 所在的梯形，修改其右侧边，并将 p 设为 helper。

```
| . e2 \
| ..... \
| ...... p
| ..... /
| . e1 /
```

5. Stalagmite：查找 p 所在的梯形，将其分裂为左右两个新梯形，它们的 helper 都是 p。

```
| .................. |
| ....... p ....... |
| ..... /   \ ..... |
| . e1 /     \ e2 . |
```

6. Stalactite：查找 p 所在的左右两个梯形，将它们合并为一个大梯形，将其 helper 设为 p。

```
| . e2 \     / e1 . |
| ..... \   / ..... |
| ....... p ....... |
| .................. |
```

单调多边形三角剖分

在上一步添加完对角线后，可通过遍历 DCEL 来得到分出的所有单调多边形。然后也需要对单调多边形顶点按纵坐标从大到小排序，但由于两条单调链已经有序，可以用 O(n) 的时间合并。

然后，按顺序依次处理每个顶点，同时用一个栈维护属于同一单调链的内凸链。每加入一个顶点，分为以下几种情况：

1. 与栈上的点同属一条单调链，则弹出栈上所有当前点可见的点，同时可直接向它们连对角线，最后将当前点加入栈。
2. 与栈上的点位于不同的单调链，则当前可见栈上所有点，需要从栈上弹出除了栈顶外的所有点，同时也直接向它们连对角线，最后将当前点加入栈。

由于每个顶点最多入栈、出栈一次，所以时间复杂度为 O(n)。

处理水平边

在考虑以上算法时，我们都假设没有两个顶点的纵坐标相同。但在实际实现时这种情况不容忽视，需要进行特殊处理。

首先，如果两个纵坐标相同的顶点不是多边形的相邻顶点，即没有水平边，则只需在排序时将纵坐标相同的顶点再按横坐标从小到大排，以上算法仍然适用。

对于梯形剖分时遇到的水平边，我们可以将对顶点 p 的处理推广到对水平边 pq 的处理，也分为以下 6 种情况：

1. 起始：p 和 q 均可作为 helper。

```
  q -- p
 / .... \
/ e2  e1 \
```

2. 结束： 如果当前梯形有一个 helper 是 stalactile，p 和 q 均可向其连边。

```
\ e1  e2 /
 \ .... /
  p -- q
```

3. 左相邻：我们需要选择 p 和 q 中更靠右的作为 helper，以防 helper 向右侧连对角线时与边 pq 重合。

```
       Case A              Case B
  / e1 ...... |           / e1 . |
 / .......... |          / ..... |
p -- q ...... |    q -- p ...... |
      \ ..... |     \ .......... |
       \ e2 . |      \ e2 ...... |
                  (need to add diagonal)
```

4. 右相邻：选 p 作为 helper 即可，因为之后与 p 和 q 同一纵坐标的点不会选 p 连对角线。

```
  Case C              Case D
| . e2 \           | ...... e2 \
| ..... \          | .......... \
| ...... q -- p    | ...... p -- q
| .......... /     | ..... /
| ...... e1 /      | . e1 /
                   (need to add diagonal)
```

5. Stalagmite：分裂出的左梯形选 p 作为 helper，右梯形选 q 作为 helper。

```
| ..................... |
| ...... p -- q ...... |
| ..... /      \ ..... |
| . e1 /        \ e2 . |
```

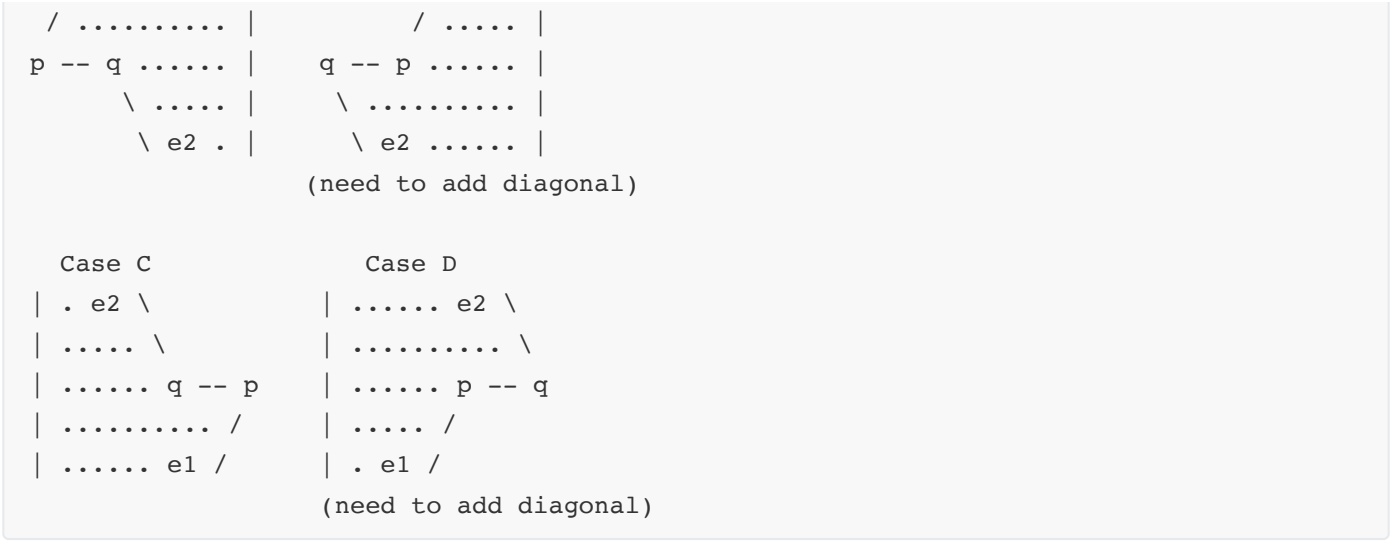
6. Stalactite：根据 helper 靠右原则，选 p 作为 helper。

```
| . e2 \         / e1 . |
| ..... \      / ..... |
| ...... q -- p ...... |
| ..................... |
```

这样就能处理有水平边的单调多边形分解了。

不过这样得到的单调多边形也可能有水平边，以上的单调多边形三角剖分算法有时会遇到麻烦。我们考虑以下四种可能的情况：

```
      Case A              Case B
  / e1 ...... |           / e1 . |
```



如果我们在遇到纵坐标相同的点时，先按点属于左侧链还是右侧链排，然后再按原多边形链上的顺序排(即 Case A 中 p 在 q 之前，Case C 中 q 在 p 之前)，则之前的算法在处理这些情况时：

1. Case A, Case C：仍能正确运行；
2. Case B：如果右侧链中有一个与 p 和 q 纵坐标相同的点，它就会向 p 和 q 都连对角线，造成对角线与 边 pq 重合；
3. Case D：如果左侧链中有一个与 p 和 q 纵坐标相同的点，且还未出栈，则之后按 q、p 的顺序处理时，会让 p 先向其连对角线，造成对角线与 边 pq 重合。

对此，我们的解决方法是，通过在梯形剖分中引入额外的对角线，来消除 Case B 和 Case D，使得不会在单调多边形三角剖分中遇到：

1. 对于 Case B，需要从 p 处引出对角线，可直接将 p 视为一个 stalactile，让之后的点自己向其连边；
2. 对于 Case D，需要从 p 处引出对角线，直接将 p 和当前梯形的 helper 连边即可。

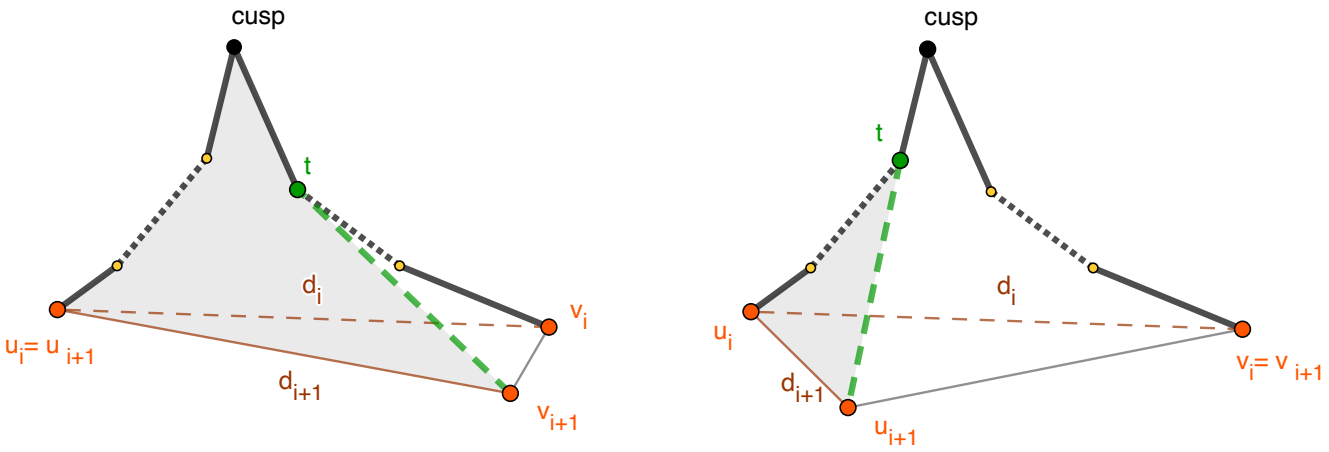
至此，我们实现的算法可以完美处理水平边问题，下文第 4.1 节中给出了几个例子，说明了我们算法的鲁棒性。

2.3 简单多边形内最短路

在对简单多边形三角剖分后，我们使用 [1] 中的算法求多边形内两点的最短路。

算法首先根据三角剖分的结果，求出对偶图，并求出 s 和 t 所在的三角形在对偶图中的路径。由于无洞多边形三角剖分的对偶图是树，所以该路径是唯一的。然后，路径上每个点对应的三角形合起来形成原多边形的一个子集，被称为一个袖形多边形 (sleeve)，即其三角剖分的对偶图是一条链的多边形。则原多边形中起点到终点的最短路只能经过袖形多边形上的顶点。

在得到袖形多边形后，我们依次处理它的每条对角线。该算法会维护一个漏斗 (funnel)，漏斗由一个尖顶 (cusp) 和两条向内凸的多边形链组成。初始的尖顶即为起点。之后每遇到一条新对角线和新的点，需要求出新的点与漏斗某一分支的切线。此时有两种情况，如下图所示：



1. 与新点 (v_{i+1}) 相切的分支和新对角线 (d_{i+1}) 在异侧(左图)：按切线切去小部分漏斗，缩小漏斗范围。
2. 与新点 (u_{i+1}) 相切的分支和新对角线 (d_{i+1}) 在同侧(右图)：按切线切去大部分漏斗，同时需要更新漏斗尖顶，并将对应分支被切去的部分加入最短路径。

对每条对角线重复以上过程，直到遇到终点，即可得到最终最短路径。

在实现时，使用两个双端队列维护漏斗的两个分支，其中队首是 cusp 节点，因为由图可知我们需要支持从队列两端弹出被切去的点。在求切线的同时可确定被切去的点，将它们从队列中弹出。由于点被切去后不会再被加入漏斗，所以该算法的时间复杂度为 $O(n)$。

3. 系统架构

本次项目的核心算法使用 Rust 语言实现，可编译生成多个平台的目标代码，因此我们为算法提供了两种运行模式：

1. 编译为本机上的可执行程序
2. 编译为 WebAssembly，直接在浏览器里运行

以下是本项目的源代码组织结构：

- `core/`：
  - `src/`：核心算法 (Rust)
  - `tests/`：单元测试 (Rust)
- `cli/`：命令行工具 (Rust)
- `web/`：网页端
  - `src/`：将核心算法导出为 JavaScript 函数的胶水代码 (Rust)
  - `site/`：网页前端代码 (HTML, JavaScript)

3.1 模块设计

3.1.1 核心算法

本项目核心算法使用 Rust 语言编写，源码位于 `core/` 目录。该部分被封装为一个库 (Rust crate)，可供命令行工具和网页端调用。

核心算法可分为以下几个模块，分别对应了 `core/src/` 下的子目录：

- `geo/`：基础几何模块，包括点、多边形的定义与相关函数，以及 DCEL (Doubly Connected Edge List structures) 结构的实现。
- `gen/`：多边形生成模块，包括排列法、空间划分算法、2-opt moves 算法与回溯法的实现。
- `tri/`：三角剖分模块，包括单调多边形分解算法和割耳法的实现。
- `shortest/`：最短路模块，实现了上文所述的基于三角剖分的最短路算法。
- `graph/`：图论模块，定义了图结构，用于定义三角剖分的对偶图和求图上两点的路径。

3.1.2 命令行工具

本项目提供一个命令行工具，也用 Rust 语言编写，源码位于 `cli/` 目录，支持以命令行方式运行简单多边形生成、简单三角剖分、简单多边形内部最短路算法。由于是直接在本机执行二进制代码，所以拥有最高的执行效率。

关于命令行工具的使用方法详见《用户手册》。

3.1.3 网页端

由于 Rust 语言支持编译到 WebAssembly，且能和 JavaScript 相互调用，这为在网页端实现复杂的算法提供了极大的便利。

网页端的源码分为两部分：`web/src/` 中包含一些 Rust 胶水代码，为几个核心算法加了一个包装函数，以供在 JavaScript 中调用；`web/site` 中的是网页前端的代码，包括一些 HTML 和 JavaScript 文件，并可使用 webpack 打包以方便部署。

关于网页端应用的使用方法详见《用户手册》。

3.2 数据结构设计

以下是本项目中几个重要数据结构的设计与实现。

3.2.1 DCEL

对于三角剖分算法，我们使用 DCEL 结构 (Doubly Connected Edge List structures) 维护平面图，其中边和面结构的定义如下(Rust)：

```
pub struct Edge {
    pub id: usize,
    pub start: usize,
    pub end: usize,

    pub twin: WeakEdge,
    pub prev: WeakEdge,
    pub next: WeakEdge,
    pub face: Option<RcFace>,
}

pub struct Face {
    pub id: usize,
    pub first_edge: WeakEdge,
}

pub type RcEdge = Rc<RefCell<Edge>>;
```

```rust
pub type WeakEdge = Weak<RefCell<Edge>>;

pub type RcFace = Rc<Face>;
```

其中 `RcEdge`、`WeakEdge` 和 `RcFace` 可认为分别是 `Edge` 和 `Face` 的指针。

以此为基础，我们可以方便地遍历平面图的所有面和面上的所有边(或者说是三角剖分的所有三角形和三角形的所有边)。为了方便，我们定义了两个迭代器 `FaceIter` `EdgeIter`，用于实现这两类遍历。

3.2.2 扫描线状态

在使用平面扫描进行简单多边形单调剖分时，需要维护一个扫描线状态，记录一个有序的梯形序列。为了支持快速插入、删除、查找，需要使用平衡二叉树进行维护，在 Rust 中提供了 `BTreeMap` 结构，因此扫描线状态的定义如下(Rust)：

```rust
struct Helper {
    /// The vertex index of the helper.
    idx: usize,
    /// The next edge of the diagonal if it was added to connect some vertex and the
helper.
    next_edge: RcEdge,
    /// Is the helper vertex a stalactile?
    is_stalactile: bool,
}

struct TrapezoidKey {
    k: f64,
    b: f64,
}

struct Trapezoid {
    /// The left edge of the trapezoid.
    left: RcEdge,
    /// The right edge of the trapezoid.
    right: RcEdge,
    /// The helper vertex used to add a diagonal.
    helper: Helper,
}

struct SweepLineStatus<'a> {
    points: &'a [Point],
    status: BTreeMap<TrapezoidKey, Trapezoid>,
}
```

其中，`TrapezoidKey` 为排序关键字，这里可用梯形左侧边所在的直线方程表示，在比较时根据当前扫描线的纵坐标，计算与梯形左侧边交点的横坐标即可。而 `Trapezoid` 描述了一个梯形，其中记录了梯形的左右两边，还有一个 helper，而 helper 的相关信息包括 helper 顶点的编号、DCEL 结构的下一条边、还有是否是 stalactile。

4. 实验与评估

我们对算法进行的评估包含三类：算法鲁棒性测试、多边形生成的效果评估，以及算法的性能评估。

4.1 鲁棒性测试

为了验证算法在各种畸形和极端情况下的正确性，我们构造了一些具有丰富性、多样性和代表性的测试用例，具有共线，水平，曲折的特点，以评估我们实现的各种算法的鲁棒性。

多边形生成算法

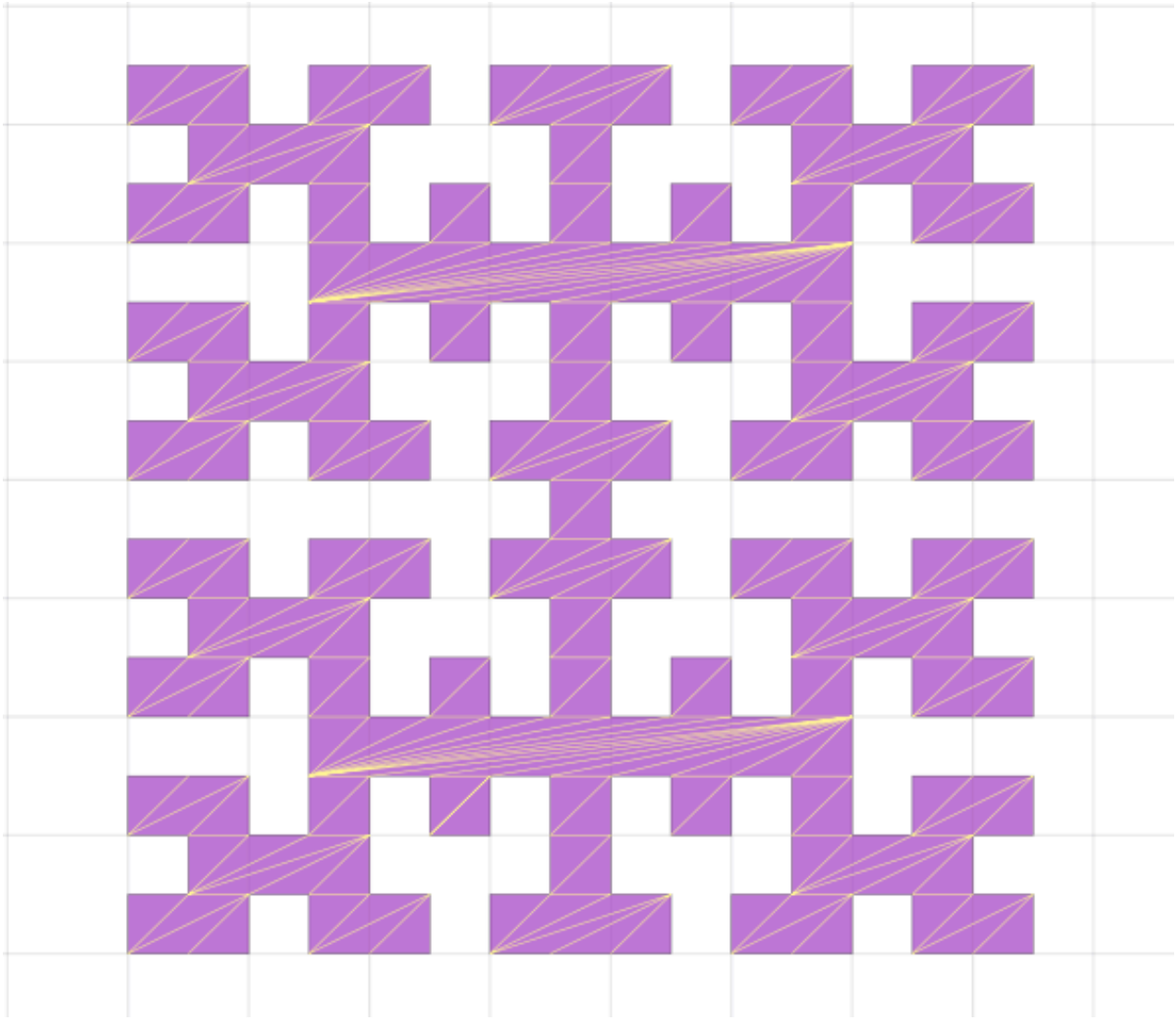
对于多边形生成算法，我们首先会确保给定点集没有重复的点。然后我们考虑了以下两类可能的畸形数据：

1. 存在大量点共线
2. 所有点都在点集的凸包上，使得简单多边形只有一个

针对这两类数据，我们分别生成了测试用例，分别为 $n \times m$ 的网格点，以及所有点都在圆上，并对我们实现的几种简单多边形生成算法进行测试，结果算法仍能正确执行，如下图所示：

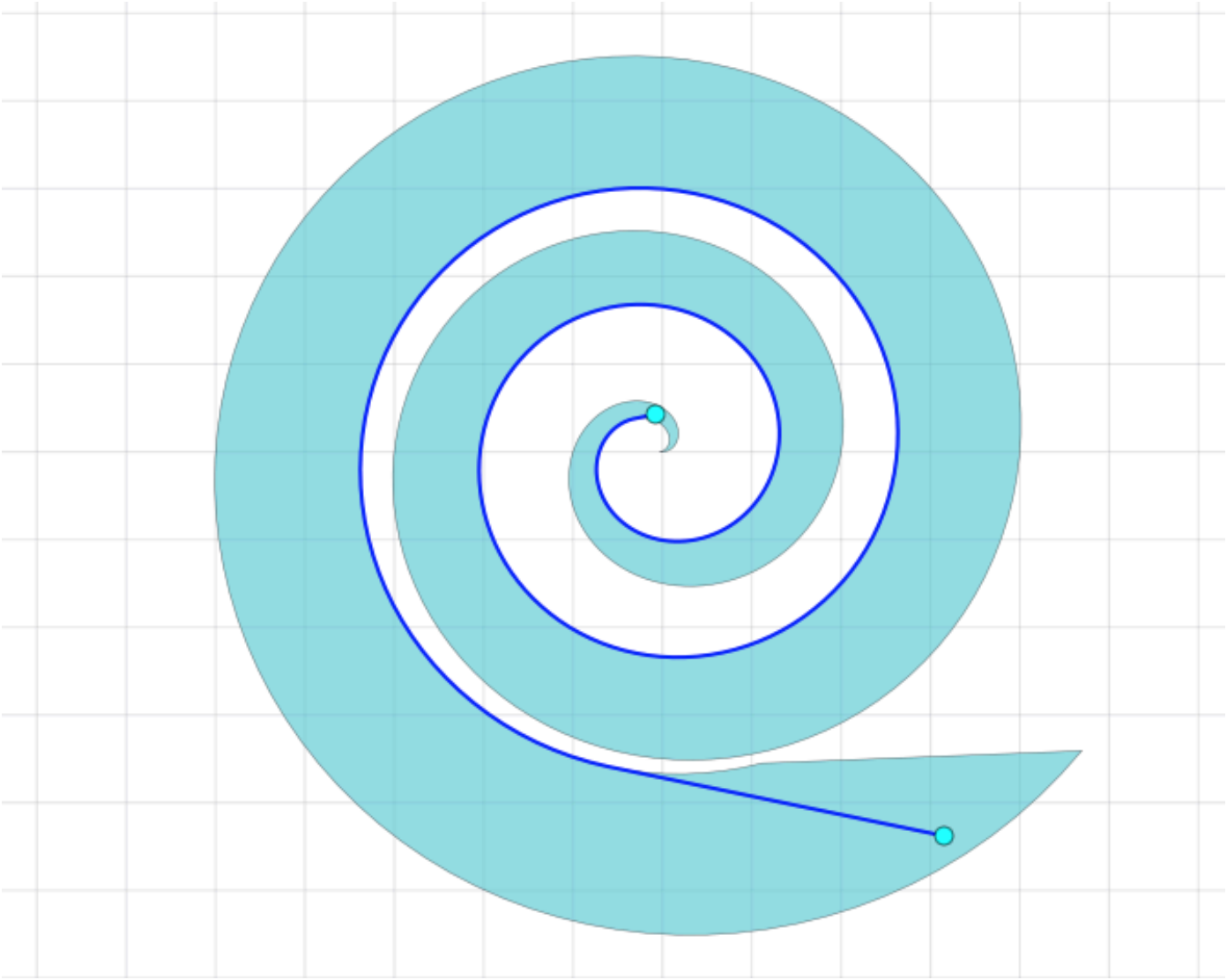
三角剖分算法

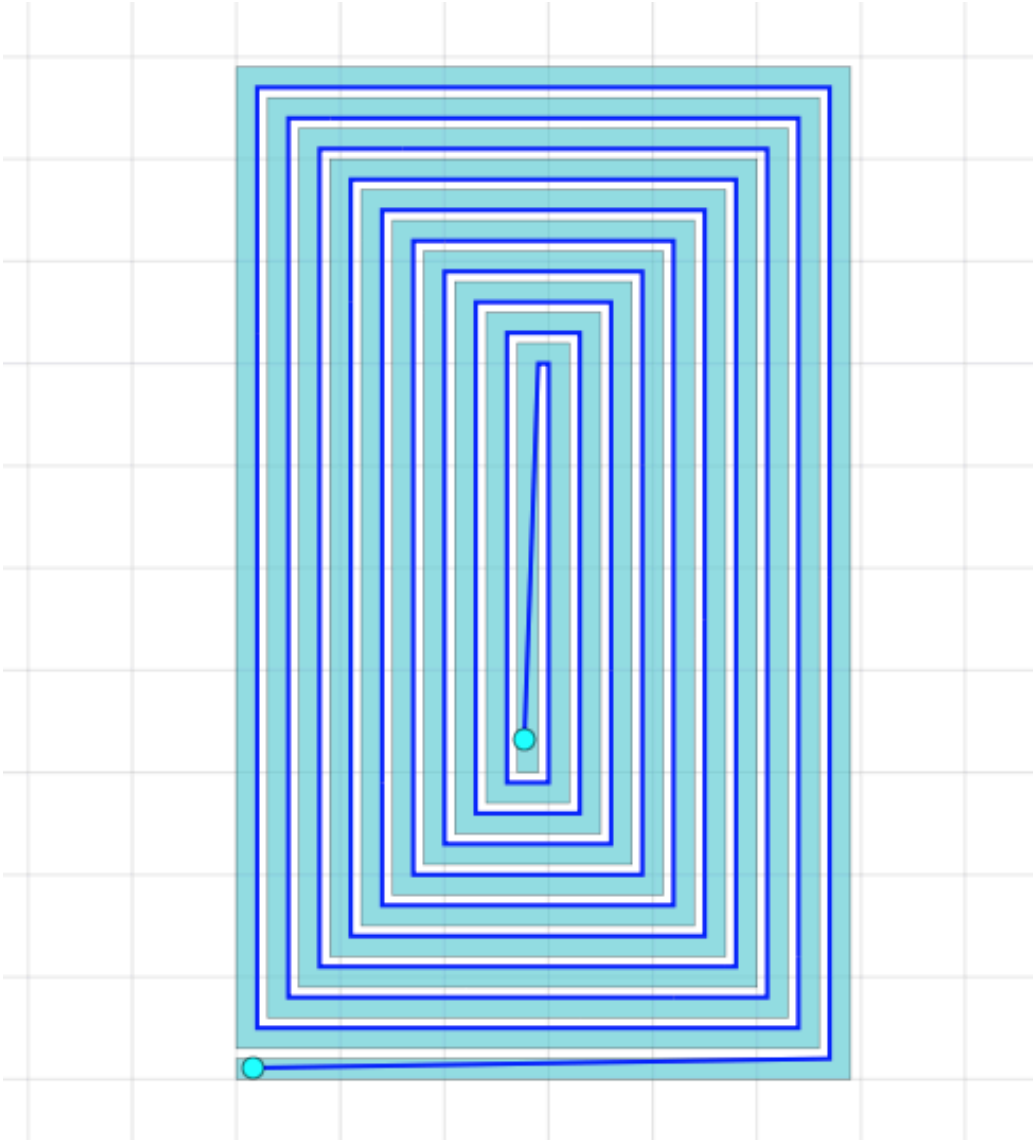
如上所述，对于三角剖分，最难处理的是有水平边的情况。为此，我们专门使用正交多边形(所有边平行与坐标轴)进行测试。如下图所示，测试用例是一个由希尔伯特曲线围成的正交多边形，在这种情况下，我们的系统仍能对其进行正确的三角剖分：



最短路算法

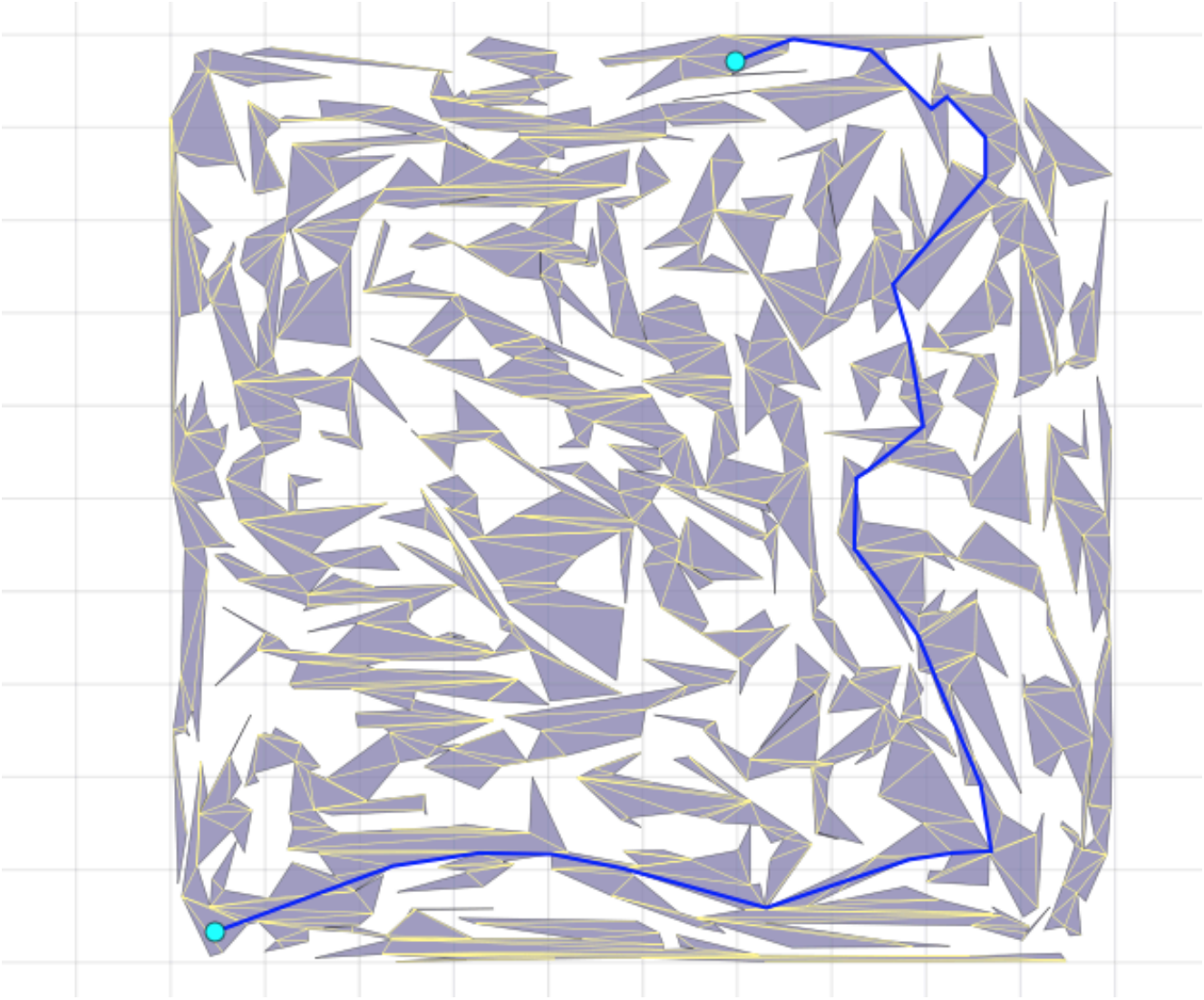
我们通过构造螺旋形的多边形，使得两点的最短路尽可能长，以此测试我们系统在极端情况下的正确性，结果如下图所示：

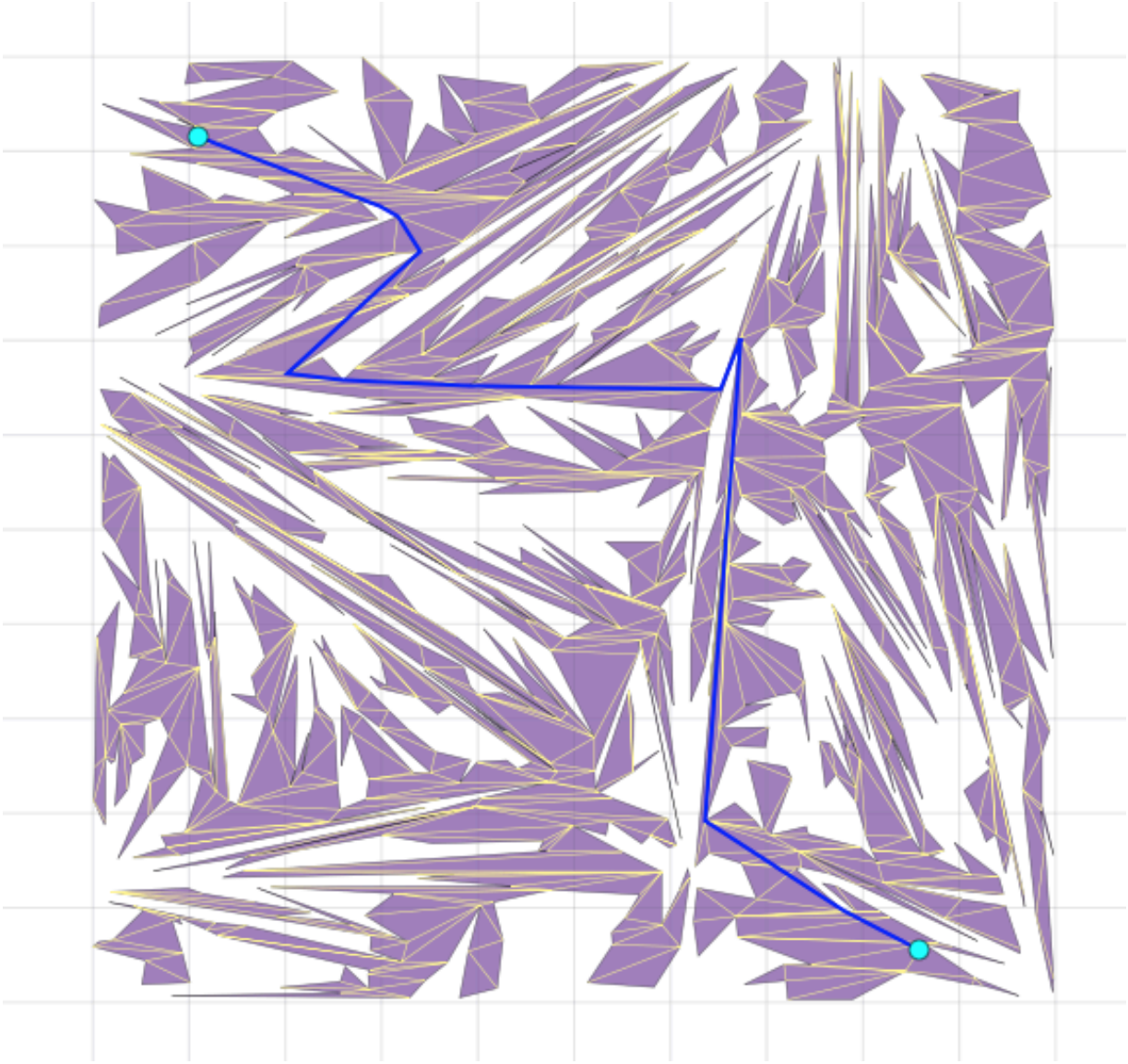




随机数据

此外，基于实现的多种随机简单多边形生成算法产生的“畸形数据”，也可以进行较好的鲁棒性测试，且是对三类算法的一个综合性测试。以下是一些测试用例：





用户界面鲁棒性

在本项目提供的 Web 应用中，在运行算法之前，会先检验一遍用户输入的多边形，如判断是否有重合点、是否是简单多边形、顶点是否按逆时针顺序等。

此外，对于最短路算法，如果用户输入的点在多边形外，也能够被检测出来。

4.2 多边形生成评估

对于不同的多边形生成算法，其生成的多边形可能具有某些特征。或具有特定的分布。因此，测试某一生成算法所生成简单多边形的分布情况，以及锁生成的简单多边形的多样性，是一个重要的评价指标。

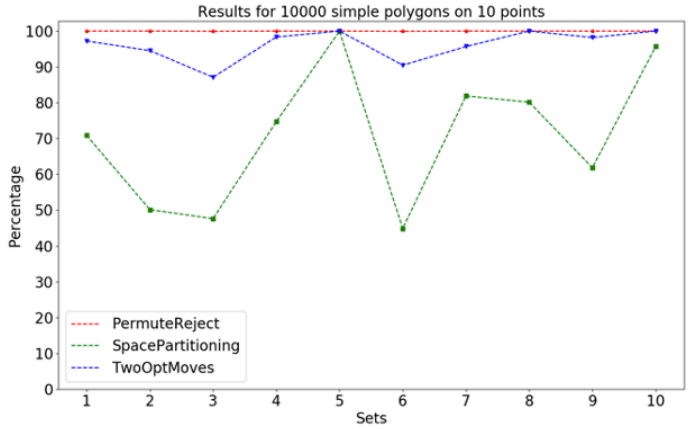
为此，我们对于一个点集，先使用回溯法，求出该点集能构成的所有可行简单多边形。然后，我们调用某一生成算法多次，并记录每次得到的简单多边形是哪一个，这样我们就能得到每个可能的简单多边形，使用生成算法能命中的次数。之后，通过统计生成简单多边形在所有可行的简单多边形中所占的比率，来评估多边形生成算法的多样性；通过统计每个可能的简单多边形命中情况的概率分布，来评估生成的简单多边形的均匀性。

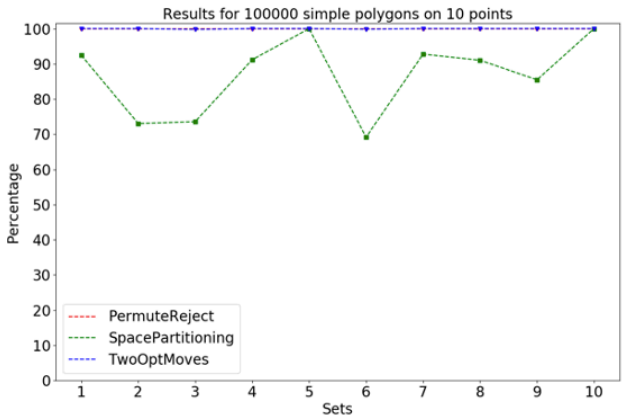
我们对于顶点数为 10 和 15 的情况，分别随机构造了 10 个不同的点集，以下是每个点集能生成的所有不同简单多边形的总数(固定起始点，顶点的方向都为逆时针)：

点数	编号	简单多边形总数	点数	编号	简单多边形总数
10	1	357	15	1	97,193
10	2	768	15	2	312,905
10	3	1,347	15	3	163,737
10	4	419	15	4	153,697
10	5	43	15	5	51,317
10	6	1,245	15	6	178,757
10	7	138	15	7	74,602
10	8	156	15	8	115,156
10	9	385	15	9	509,301
10	10	46	15	10	24,322

多边形多样性评估

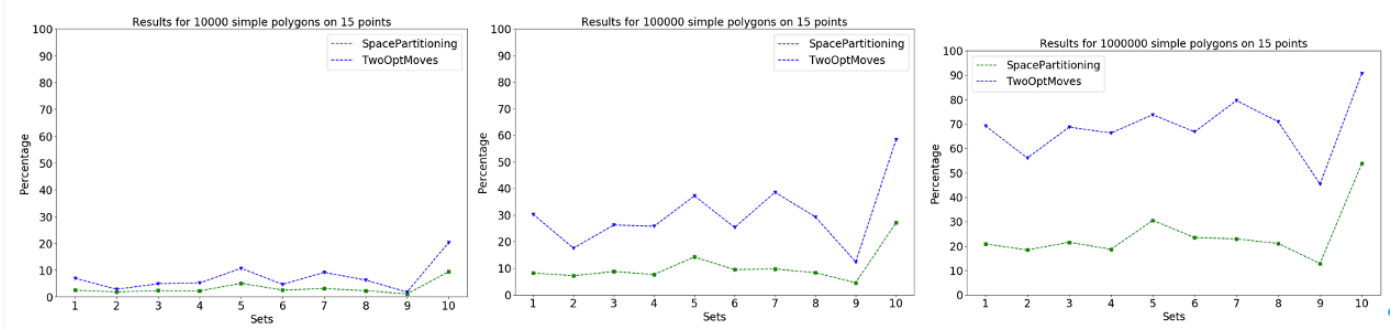
以下是对三种多边形生成算法(2-Opt Moves、Space Partitioning、Permute & Reject) 分别调用 10,000、10,0000 次后生成的简单多边形，在所有可行的简单多边形中所占的比率：





对于 10 个点，10,000 个简单多边形的生成要求，Permute & Reject 由于采用的是随机全排列的方式，且生成次数足够多，因此命中了所有可行的简单多边形。而 2-Opt Moves 相比 Space Partitioning 具备更好的多样性，基本上可以达到 90% 以上的简单多边形覆盖率。Space Partitioning 的覆盖率随着测试输入点集的不同波动较大，简单多边形覆盖率接近 70% 左右。对于 100,000 个简单多边形的生成要求，覆盖率比 10,000 个点更高了，2-Opt Moves 和 Space Partitioning 算法分别达到了 100% 和 90% 左右的覆盖率。

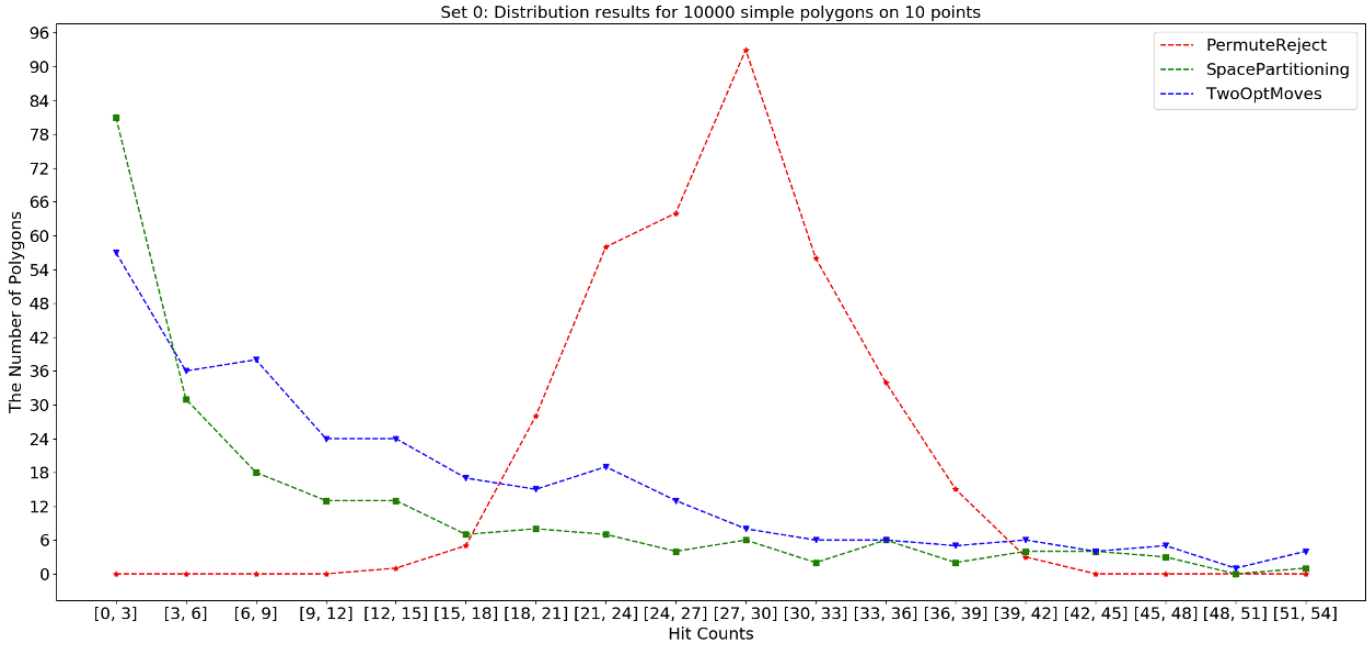
而对于每一组包含 15 个随机点的点集，我们测试两种算法 (2-Opt-Moves、Space Partitioning) 生成的 10,000、100,000、1,000,000 个简单多边形在所有可行的简单多边形中所占的比率：



在这组实验中，随着点集点数的增加，2-Opt Moves 和 Space Partitioning 的覆盖率总体都有所下降。2-Opt-Moves 保持在 70% 左右的覆盖率，Space Partitioning 保持在 20% 左右的覆盖率。总体而言，2-Opt-Moves 生成的简单多边形多样性依旧好于 Space Partitioning。并且我们可以观察到，不同的输入点集对于两种算法生成的多边形多样性是有影响的，比如可能的简单多边形总数少的点集会在两种算法上都产生较好的简单多边形多样性。

多边形数量分布评估

为了评估各算法生成的简单多边形的均匀性，根据每种简单多边形的命中次数情况，我们以命中次数为横坐标，命中次数在某一区间内的简单多边形个数为纵坐标，得到以下的分布图(点数为 10，1 号数据集)：



由于 Permute & Reject 算法的思想是随机一个全排列的方式，其产生每种简单多边形的概率是均等的，因此其调用产生简单多边形的分布规律符合正态分布曲线。我们以 Permute & Reject 算法的曲线作为标准，对比处理其他两种随机生成算法 (2-Opt Moves 和 Space Partitioning)。首先，我们统计并计算 Permute & Reject算法生成的简单多边形的期望命中次数 E，而后将其我们将命中次数区间化处理，以 $[0, 2E]$ 为 x 轴总体区间进行分割，统计其余两种随机生成算法所生成的每种简单多边形的命中次数，对同一区间内的命中次数的简单多边形进行计数统计。

可以明显观察到两种算法生成的大多数简单多边形不符合正态分布，是非均匀的“命中”，且大多数简单多边形的命中次数都较少，集中在值较低的命中区间内。但是也有较少数量的简单多边形会被多次命中，超过 Permute & Reject 算法的期望值。

此外，从图中也能看出，2-Opt Moves 算法的曲线比 Space Partitioning 算法的曲线更高，也就是更加集中在 $[0, 2E]$ 范围内，因此可以认为 2-Opt Moves 算法比 Space Partitioning 算法的均匀性更好。

4.3 性能评估

测试数据集编号	测试数据集大小	测试数据集坐标范围
1	100	100
2	1000	100
3	10000	1000
4	100000	1000
5	1000000	10000
6	10000000	100000

简单的多边形生成

测试数据集编号	算法	用时(s)	2-Opt-Move次数
1	2-Opt Moves	0.003	184
2	2-Opt Moves	0.203	3717
3	2-Opt Moves	1:23.07	53757

测试数据集编号	算法	用时(s)
1	Space Partitioning	0.002
2	Space Partitioning	0.002
3	Space Partitioning	0.006
4	Space Partitioning	0.042
5	Space Partitioning	0.525
6	Space Partitioning	7.380

简单的多边形三角剖分

使用Space Partitioning生成的数据进行测试：

测试数据集编号	算法	用时(s)
1	mono_partition	0.002
2	mono_partition	0.003
3	mono_partition	0.012
4	mono_partition	0.134
5	mono_partition	1.497
6	mono_partition	17.552

测试数据集编号	算法	用时(s)
1	ear_cutting	0.002
2	ear_cutting	0.010
3	ear_cutting	0.515
4	ear_cutting	46.113

多边形内最短路

测试数据集编号	算法	用时(s)
1	mono_partition	0.002
2	mono_partition	0.003
3	mono_partition	0.015
4	mono_partition	0.171
5	mono_partition	1.815
6	mono_partition	18.816

测试数据集编号	算法	用时(s)
1	ear_cutting	0.002
2	ear_cutting	0.008
3	ear_cutting	0.515
4	ear_cutting	46.277

5. 小组分工

- 贾越凯：系统框架搭建、简单多边形三角剖分与最短路算法的实现、前端单步演示的实现
- 徐瑞翔：Steady Growth 和 Space Partitioning 算法的实现，用户界面和图标设计、视频剪辑
- 许怡文：2-opt Moves 算法的实现、网页前端实现、多边形生成评估、性能评估

6. 参考文献

1. T. Auer and M. Held. Heuristics for the generation of random polygons. In Proc. 8th Canad. Conf. Comput. Geom., pages 38–43, 1996.
2. D.T. Lee and F.P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. Networks, 14:393–410, 1984.
3. M.R. Garey, D.S. Johnson, F.P. Preparata, and R.E. Tarjan. Triangulating a simple polygon. Inform. Process. Lett., 7:175–179, 1978.
4. J.E. Goodman, J. O'Rourke and C.D. Tóth, Handbook of Discrete and Computational Geometry, third edition. CRC Press LLC, Boca Raton, FL, 2017, ISBN 978-1498711395.
5. J. O'Rourke. Computational Geometry in C, second edition. Cambridge University Press, 1998.
6. C. Zhu, G. Sundaram, J. Snoeyink, and J. S. B. Mitchell. Generating random polygons with given vertices. Comput. Geom. Theory Appl., 6:277–290, 1996.
7. B. Chazelle. A theorem on polygon cutting with applications. In Proc. 23rd IEEE Sympos. Found. Comput. Sci., pages 339–349, 1982.
8. S. Hertel and K. Mehlhorn. Fast triangulation of the plane with respect to simple polygons. Inform. Control, 64:52–76, 1985.
9. R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. Comput. Geom., 1:51–64, 1991.
10. R. E. Tarjan and C. J. Van Wyk. An O(nloglogn)-time algorithm for triangulating a simple polygon. SIAM J. Comput., 17:143-178, 1988. Erratum in 17:1061, 1988.
11. B. Chazelle. Triangulating a simple polygon in linear time. Discrete Comput. Geom., 6:485–524, 1991.
12. L.J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R.E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. Algorithmica, 2:209–233, 1987.
13. L.J. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. J. Comput. Syst. Sci., 39:126–152, 1989.
14. D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. SIAM J. Comput., 6:594-606, 1977.