# 组队大作业设计文档

2018/6/30

本文档主要介绍了该程序解决的问题，和程序的使用方法、运行结果与分析。

## 实验说明

Paper: New Algorithms for the Rectilinear Steiner Tree Problem

需要求解的问题是将电路板上的一些点用最少的垂直电路长度全部连成一棵树的最优方案。

论文提出的算法是基于平面曼哈顿距离的一些性质，利用了平面上斯坦纳树的结构，可用最小生成树算法加一些枚举，来获得一个效果不错的电路图。

## Reference

Ho, J. M., Vijayan, G., & Wong, C. K. (1990). New algorithms for the rectilinear steiner tree problem. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 9(2), 185-193.

## 程序设计说明

该程序使用论文中的算法来计算电路排布方案，程序内容在 namespace RECTILINEAR_STEINER_TREE 下，调用 RSTSolver 类来解决这个问题。

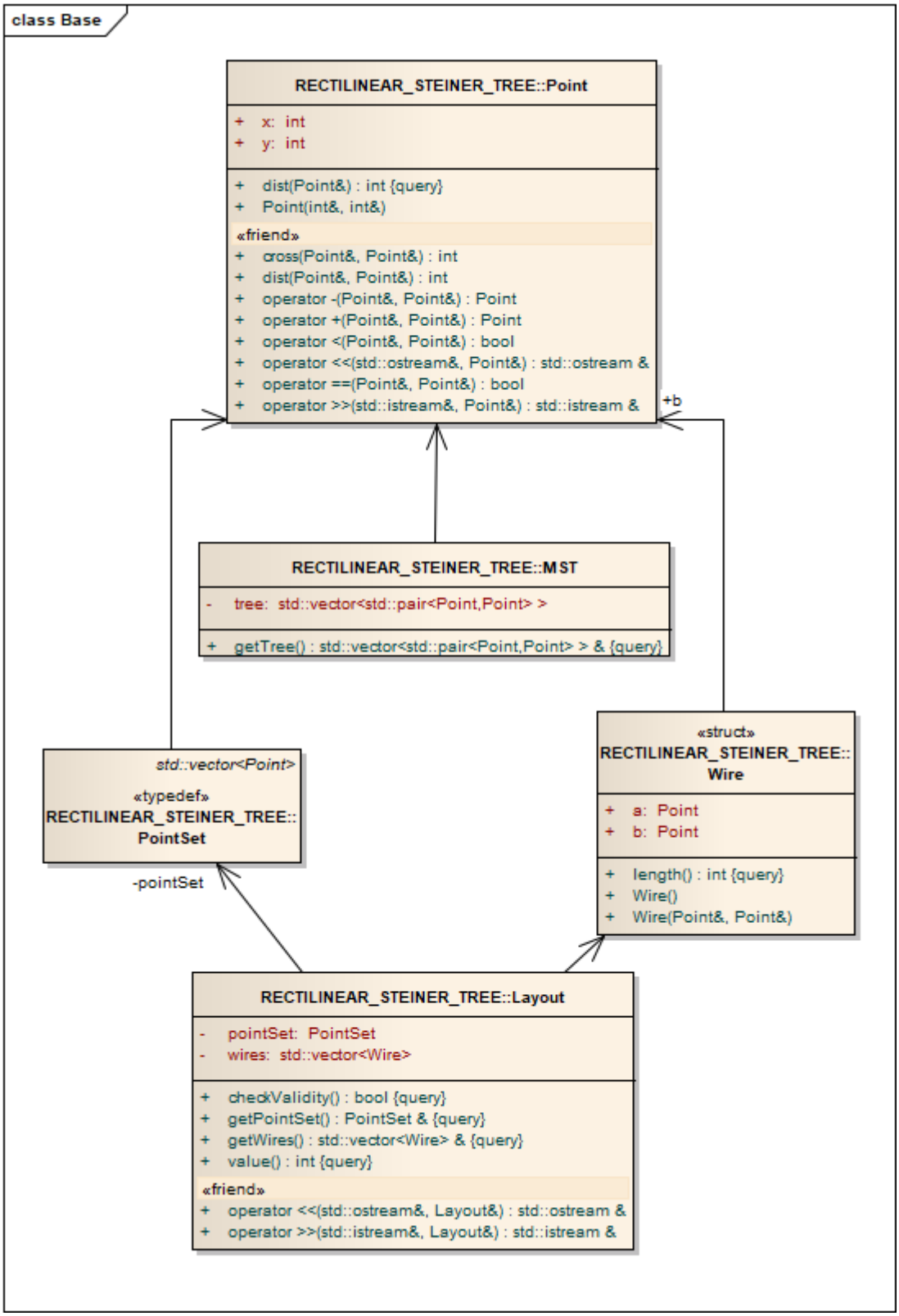
第一步是计算一个三元组意义下的平面最小生成树，这一步可以用 Prim 算法来完成，也可以用一种计算平面曼哈顿距离最小生成树的技巧来实现，时间复杂度分别是 $O(n^2)$和 $O(n \log n)$。这两个算法分别用 PrimSMST 和 OptSMST 实现，它们共同的基类 SMST 提供了接受点集的接口和输出最小生成树类 MST 的接口。

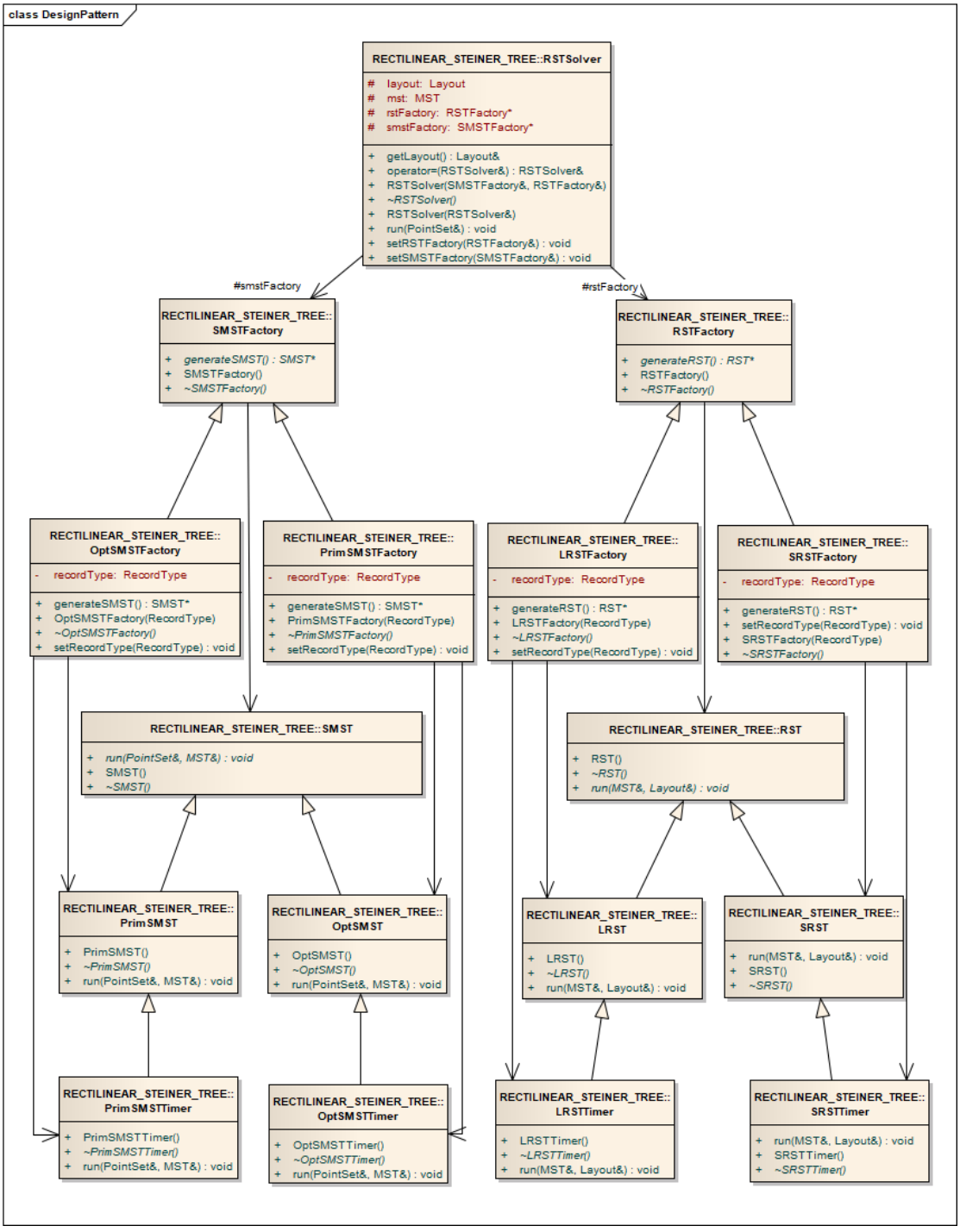
第二步是根据第一步计算的最小生成树来排布点与点之间的电路。论文中证明了两次转折和多次转折的最优解是同样优的，因此可以只考虑两次转折来计算结果。但由于两次转折的计算方式复杂度过高，也可以考虑用一次转折来计算较优的方案。多次转折的求解程序类是 SRST，单次转折的求解程序类是 LRST，它们共同的基类 RST 提供了接受 MST 的接口和输出最小生成树类 MST 的接口。

那么整个解决问题的流程如下：


除了提到的这些类以外，程序中还有一些辅助性的类，如二维点，并查集，无向树，计时器，计时代理等类，在程序执行过程中被这些类使用。

# UML 图


class Base
RECTILINEAR_STEINER_TREE::Point
+ x: int
+ y: int
+ dist(Point&) : int {query}
+ Point(int&, int&)
«friend»
+ cross(Point&, Point&) : int
+ dist(Point&, Point&) : int
+ operator -(Point&, Point&) : Point
+ operator +(Point&, Point&) : Point
+ operator <(Point&, Point&) : bool
+ operator <<(std::ostream&, Point&) : std::ostream &
+ operator ==(Point&, Point&) : bool
+ operator >>(std::istream&, Point&) : std::istream &
+b
RECTILINEAR_STEINER_TREE::MST
- tree: std::vector<std::pair<Point,Point> >
+ getTree() : std::vector<std::pair<Point,Point> > & {query}
std::vector<Point>
«typedef»
RECTILINEAR_STEINER_TREE::
PointSet
-pointSet
«struct»
RECTILINEAR_STEINER_TREE::
Wire
+ a: Point
+ b: Point
+ length() : int {query}
+ Wire()
+ Wire(Point&, Point&)
RECTILINEAR_STEINER_TREE::Layout
- pointSet: PointSet
- wires: std::vector<Wire>
+ checkValidity() : bool {query}
+ getPointSet() : PointSet & {query}
+ getWires() : std::vector<Wire> & {query}
+ value() : int {query}
«friend»
+ operator <<(std::ostream&, Layout&) : std::ostream &
+ operator >>(std::istream&, Layout&) : std::istream &




# 用户接口

为用户提供了 RSTSolver 类作为解决器的类，用 run()接口计算并用 getLayout()接口获取储存答案的 Layout 类。

为了在内部实现不同的算法，RSTSolver 可以通过设置 SMSTFactory、

RSTFactory 这两个工厂类来确定运行时使用的算法。RSTSolver 的 run()接口需要传入 const PointSet&，表示电路板上点的集合。在执行 run()之后，由 getLayout()接口返回 const Layout&为答案信息。

Layout 内存储有电路板的结果信息，有 getPointSet()和 getWires()接口可以获取电路板的点集和所有的电线集（用 Layout::Wire 中的坐标对表示）。Layout 重载了输入输出，提供了 checkValidity()和 value()接口来验证答案的正确性和获取答案的电路长度。

我们还使用了 QT 对结果进行可视化输出。

## 设计模式

### (1) 代理

为测量每个算法的运行时间，使用了代理模式，为每个算法类创建了一个代理，代理中依次进行了计时器启动，代码运行，计时器暂停 3 个操作。

### (2) 单件

计时器为了精确统计运行时间，方便在各处调用计时函数统计时间，采用了单件的设计模式来实现。可以支持在运行时的任何时候开始或结束对时间的统计。

### (3) 参数化工厂

使用工厂产生每个算法的具体实现类。每个算法类有原类和代理类两个版本，为避免工厂数目过多，每个算法类对应了一个工厂，工厂内使用参数确定

产生原类或是产生代理类。

### (4) 模板方法

本问题分为由点集求出 SMST 和由 SMST 求出 RST 两部分，每个部分分别需要产生算法类和运行算法类两个步骤。因此，使用模板方法，将算法骨架定义为产生 SMST 算法类，运行 SMST 算法类，产生 RST 算法类，运行 RST 算法类四个步骤。

### (5) 策略模式

SMST、RST 分别有两种实现方法，这两种实现方法分别被封装到了两个对应的算法类中，并定义了相应的产生算法类的工厂。为了较为灵活地选择各个步骤中使用的算法，我们将算法类的产生工厂视为对应步骤的策略，并定义了改变使用策略的接口。

## 实验数据与结果

数据 1、2 为 20 个点，3、4、5 为 100 个点，6 位 10000 个点。

对比 PrimSMST 和 OptSMST：

| | **PRIMSMST** | **OPTSMST** |
|---|---|---|
| **CASE 1** | $4.9\times10^{-5}$ | 0.0001 |
| **CASE 2** | 0.001 | 0.0007 |
| **CASE 3** | 0.001 | 0.0006 |
| **CASE 4** | 0.001 | 0.0008 |

| | | |
|---|---|---|
| **CASE 5** | 0.001 | 0.0011 |
| **CASE 6** | 10.99 | 0.1246 |

可以发现数据大小越大，OptSMST 的效率越优秀。

对比 LRST 和 SRST：

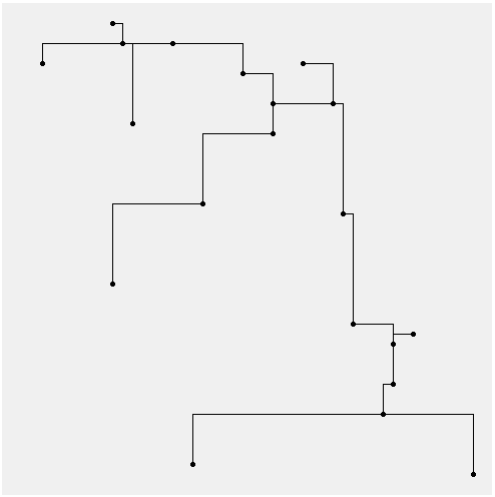
| | **LRST** | | **SRST** | |
|---|---|---|---|---|
| | 时间 | 总长度 | 时间 | 总长度 |
| **CASE 1** | 0.0002 | 159 | 0.0084 | 159 |
| **CASE 2** | 0.0011 | 161 | 0.2166 | 161 |
| **CASE 3** | 0.0014 | 381 | 0.1416 | 379 |
| **CASE 4** | 0.0010 | 159 | 0.0874 | 159 |
| **CASE 5** | 0.0011 | 780 | 0.4192 | 778 |
| **CASE 6** | 0.1342 | 15277 | 复杂度过高无法计算 | |

可以发现 LRST 速度很快。效果上，在随机数据中二者表现差距不明显，但 SRST 必然比 LRST 效果好。

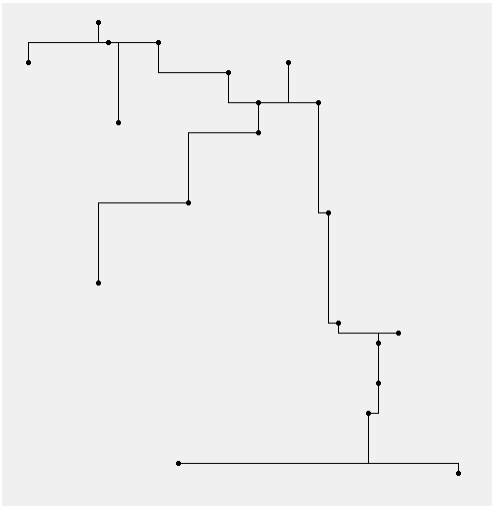
## 最终生成电路版

CASE 1：

LRST：

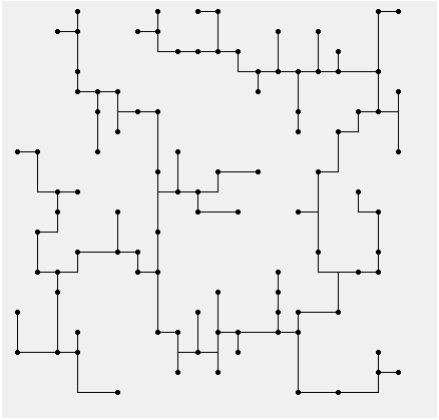


SRST：

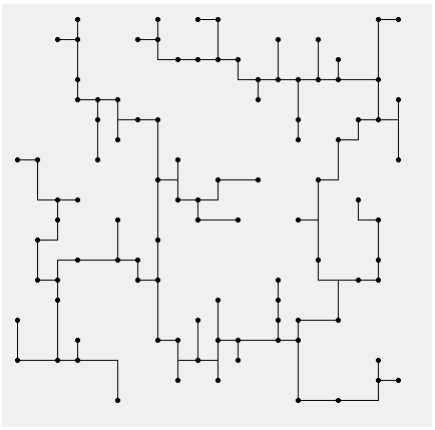


CASE 2：

LRST：

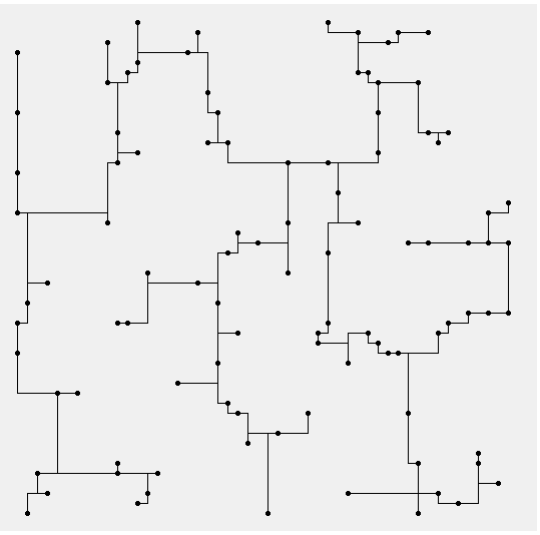


SRST：

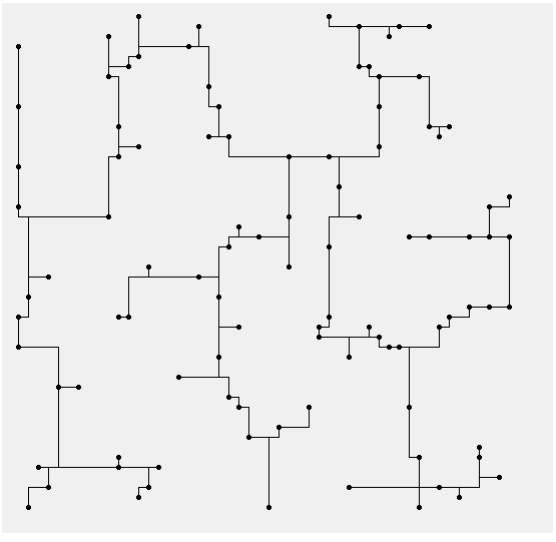


CASE 3：

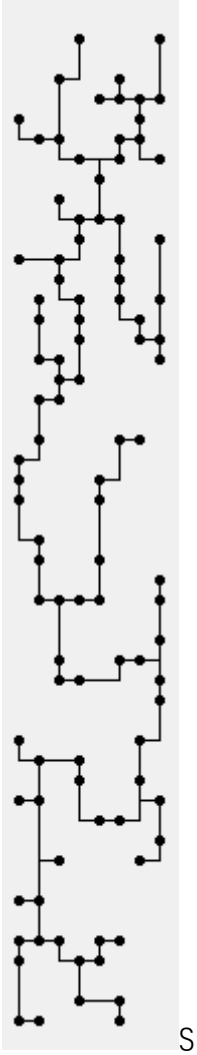
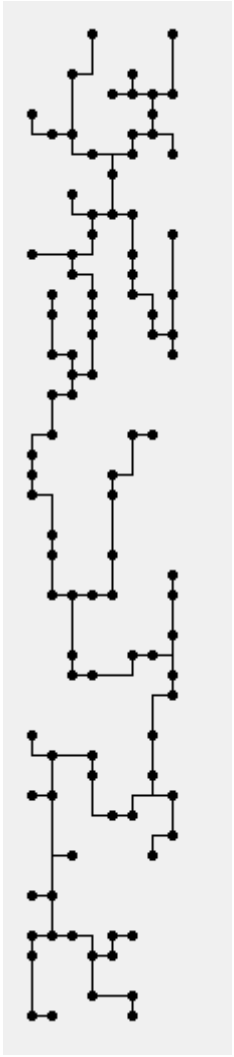
LRST：



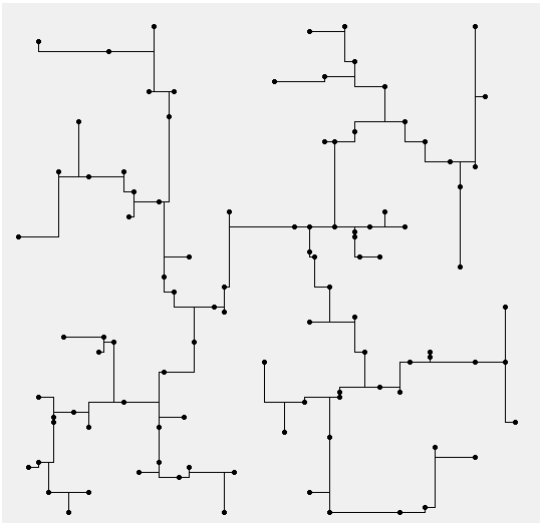
SRST：



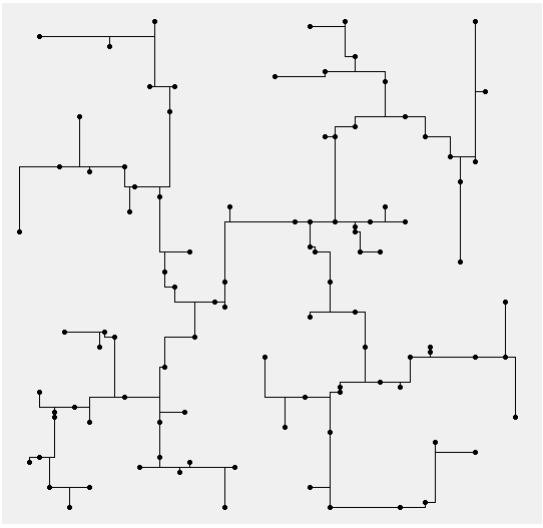
CASE 4：

LRST：  SRST： 

CASE 5：

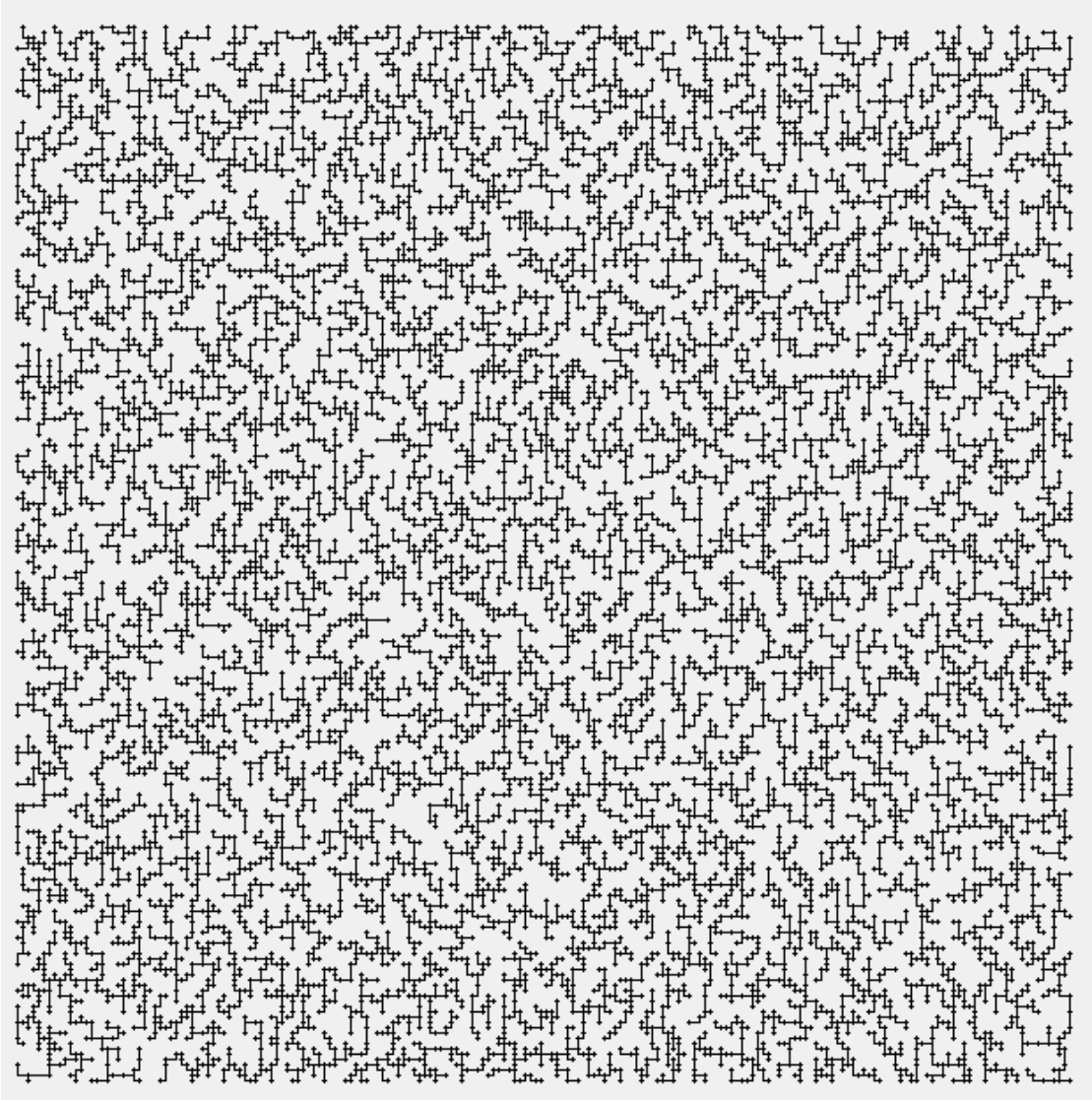
LRST： 

SRST：



CASE 6：

LRST：



SRST：无法计算