防雷暗敷引下线算法开发文档

1 背景介绍

在建筑设计过程中，必然要考虑到防雷系统（Lightning Protection System）的设计。当前的防雷系统主要包含接闪带、接闪网、引下线和接地网。在防雷系统当中，**引下线系统**（Down Conductor System）是一个重要的组成部分，它负责将电从楼高处的接闪带逐层引导到地下。引下线分为明敷和暗敷两类，明敷引下线主要应用在屋顶突出构件上，线路直接暴露在外表；暗敷引下线主要应用在楼层之间的传输上，布置位点取于楼内的竖向构件当中，无法直接从楼房外表看到。

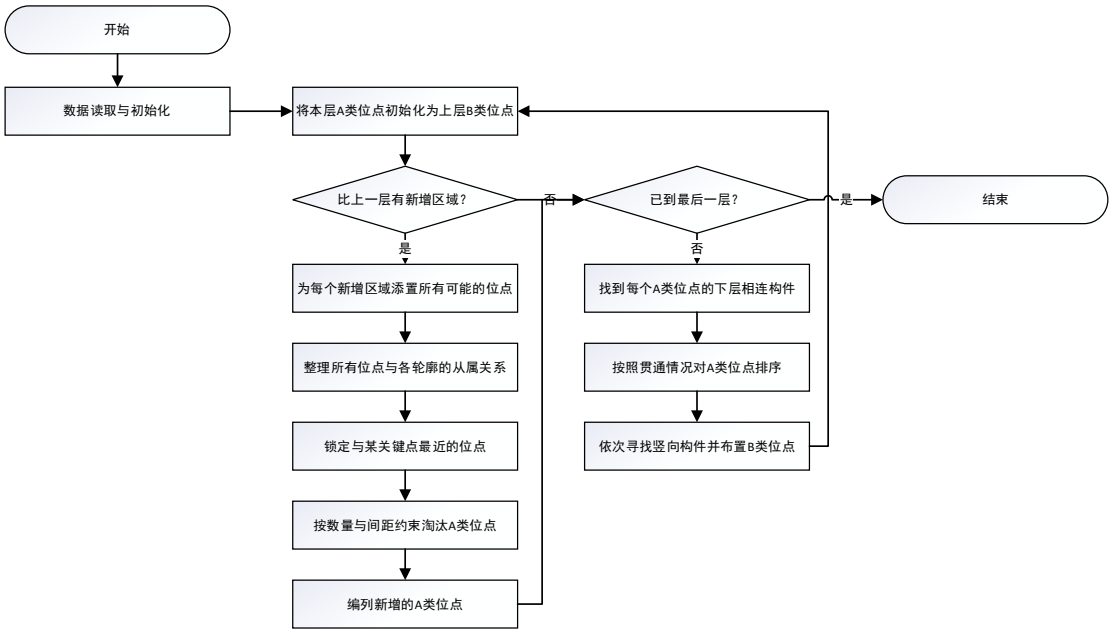
本算法的目标是：根据自大屋面而下各楼层的平面图纸，解析出每一层的建筑外轮廓、竖向构件和横梁，实现建筑单体各楼层（暗敷）引下线的自动布置。

2 算法思路

2.1 核心思想

从上到下逐层布置；每层引下线分 A、B 两类依次布置；对于每个楼层求出所有可能的位点，全部布置之后再根据相关约束逐步过滤掉可以不布置的位点。

2.2 流程图



算法从顶层开始布置。对于每一层来说，引下线可以分为两类：一类从本层天花板引到本层地面，记为 A 类；一类从本层地面引到下层，记为 B 类。布置时先 A 类后 B 类。

如果本层区域与上层区域完全相同，或者多出一部分区域且该区域未布置接闪带，则本层 A 类可以直接照抄上层 B 类；如果本层区域多出的区域（可能有多个不连通的部分）装有接闪带，则需要按照约束添加 A 类位点，同时更新所有位点与本层各轮廓的从属对应关系。添加 A 类位点是本项目的核心部分。首先找到本楼层轮廓的关键点备用；对于每个外

围的竖向构件，找到它与哪些新增区域有交集；对于那些与一个或多个新增区域有交集的竖向构件，找到可以布置引下线的位点并维护从属的区域列表（可能不止一个）；将找到的所有位点都布置上引下线，并连同非新增的位点一起维护从属的轮廓列表（也可能不止一个）及分别对应的投影边、投影点；最后按照需求中的约束条件逐点检查，对于删去后整体仍满足约束条件的点，则舍弃掉不用。

接下来布置本层 B 类位点。对于某 A 类位点，探究它与下层竖向构件的贯通关系，一共有三种情况：贯通（through）、半贯通（shift）和断层（disrupt）。不管是上述三种贯通情况当中的哪一种，都有可能找到多个与本层竖向构件相连的下层竖向构件。从中选取未布置引下线且离外轮廓最近的构件布置位点；若找不到未布置引下线的竖向构件，则舍弃该点，该点对应的引下线也中断于此。

对于每一层数据都执行上述操作，一直到最后一层。最后的结果是每一层的 A 类位点列表和 B 类位点列表，且每个位点除了基本的坐标信息以外，还包括对应的引下线 id、楼层号、类别（A 类还是 B 类）等等。在本项目当中，结果会以 geojson 的格式呈现出来。

2.3 计算新增区域并添置 A 类位点

各层轮廓（含外轮廓与内轮廓两种）由数据读入，每个轮廓用不带洞的多边形表示。在预处理阶段，用 CGAL 自带的多边形集合操作得到每个楼层的区域。在计算新增区域时，直接用本层区域减去上层区域得到带洞多边形列表，作为新增区域的集合。由于图纸的精度与 CGAL 的 EPECK 核精度不一致，在图纸上相同的两个区域在 CGAL 上可能会减出来毛刺，严重影响程序的正常运行。为了解决这个问题，同时提升程序的鲁棒性，**在集合相减之前先把上层区域外扩一个极小的宽度（暂定 1.0mm）**，实践证明这样做能很好地杜绝毛刺的产生。

对于每一个连通的新增区域（即一个带洞的多边形），判断其是否有安装接闪带。接闪带本来是由线段列表来表示，但是为了与 CGAL 的集合操作统一接口，故**把每条线段转化成带有某个小宽度（暂定 1.0mm）的矩形**。判断新增区域是否安装接闪带的方法是：先把新增区域外扩内缩一个给定的宽度得到新的带洞多边形，如存在某接闪带矩形与之相交，则视为整个新增区域需要安装接闪带。随后程序会重点关注那些安装了接闪带的新增区域。

为了进一步提升程序的运行效率，尽量避开耗时的多边形布尔运算，在代码中添加了一个缓存字典，该字典存储了程序遇到过的两类楼层之间的安装了接闪带的新增区域列表。等到下次出现同样的情况时，就可以直接从该字典中拿到新增区域列表即可，无需重新计算一遍。

2.4 整理 A 类位点与各新增区域的从属关系

在该项目的场景中，一个竖向构件可能会横跨多个新增区域。基于此，程序会对所有外围竖向构件进行遍历，找到每个竖向构件所从属的所有新增区域。为了提高运算效率，实际的代码**没有使用多边形与多边形的交集操作，取而代之的是验证竖向构件各顶点是否被某新增区域所包含**。在本项目的需求当中，这两种操作是等价的。对于从属的新增区域大于零的竖向构件，**要检查它是否已经布置了引下线位点，只有在没有布置的情况下才能计算并添加新位点**，否则需要沿用原有位点，以避免重复。计算竖向构件的新位点，统一采用 CGAL 提供的 2D Straight Skeleton and Polygon Offsetting 模块来处理。具体地，对于一个竖向构件

（本质上是不带洞的多边形），利用 create_interior_straight_skeleton_2()函数找到它的骨架（straight skeleton），然后找到骨架当中距离楼层外轮廓的那个骨架顶点（skeleton vertex）作为最终的引下线位点。

最后，不管沿用还是新增，引下线位点与它所从属的所有新增区域需要做到互可索引，以备后用。

2.5 整理 A 类位点与各轮廓的从属关系

如果 2.3 节中存在新增位点，则为了准备后续的淘汰环节，需要先整理所有 A 类引下线位点与各轮廓的从属关系。与某引下线位点具有从属关系的轮廓（可以没有，也可以不止一个），也就是与该引下线位点所在竖向构件具有从属关系的轮廓，这部分关系由数据直接给到，不需再做额外的计算。引下线位点与它所从属的所有轮廓需要做到互可索引；引下线位点和在它所属的轮廓上左右相邻的两个引下线位点也需要做到互可索引。

一个引下线位点，对于某个它所从属的轮廓而言，还需要包含更多的信息，比如该位点到轮廓的投影点所对应的里程。投影点里程的计算方法如下：首先找到与该位点最短距离最小的轮廓边，再计算出该边上与该点距离取到最小值的点（可能是两个端点或者垂线段交点）作为投影点，最后沿边累加得到从轮廓起点开始到该投影点的里程数。这样在同一个轮廓上相邻的两个引下线位点的里程之差，就是这两个位点在这条轮廓上的间距。

2.6 锁定、淘汰与编列 A 类位点

有 2 种 A 类位点需要提前锁定（即不允许被淘汰）：一是从上面楼层引下来的位点；二是存在轮廓上一个关键顶点与之构成最短距离的位点。对于外轮廓上的顶点，满足以下 3 点即为关键顶点：

①对于一个多边形轮廓的顶点，它的起始边、顶点、终边需要构成逆时针转动关系；

②它的起始边和终边的边长均大于某一给定阈值；

③在轮廓所在二维平面里，轮廓可以与一给定半径的圆外接于该顶点。

其中条件③可以直接调用 CGAL 的 Alpha_shape 模块实现。对于内轮廓的某顶点，只需要满足上述条件②时就算作关键顶点。

执行淘汰时，遍历位点列表中的点并逐一做判定，若同时满足以下条件即可将该位点淘汰掉：

①在该点所从属的所有轮廓上，该点以左和以右两个点间距均不大于某给定值；

②在删除该点后，在该点所从属的所有轮廓上，各点间平均间距均不大于某给定值；

③在删除该点后，全局位点数仍不小于 10；

④在删除该点后，该点所在的所有新增区域内，剩余位点数均仍不小于 2；

⑤该点尚未上锁。

淘汰完毕以后，对于还未有引下线编号的位点，赋予一个新的引下线编号，也就是从本层往下会多出一条引下线。

2.7 找到 A 类位点的下层相连构件

确定了 A 类位点之后，如果不是最底层，则需要继续布置 B 类位点。遍历 A 类位点，求出每个位点与下层竖向构件的贯通关系。如果本层 A 类位点位于下一层某竖向构件多边形内部，则说明是贯通的（through）；如果下一层中不存在竖向构件能包含本层的 A 类位点，

但存在某竖向构件与 A 类位点对应的本层竖向构件有交集，则是半贯通的，需要移动位点坐标（shift）；若仍不能找到楼层间竖向构件的相交关系，则需要以横梁为中介去寻找与本层位点相连接的下层竖向构件，这被称为断层（disrupt）。不管属于哪一种贯通关系，每个 A 类位点都会产生一个列表，这个列表包含了与该位点相连接的所有下层竖向构件：对于贯通的（through）位点，就是所有包含该位点的竖向构件；对于半贯通的（shift）位点，就是所有与该位点所在竖向构件有交集的竖向构件；对于断层的（disrupt）位点，就是所有能够通过横梁与该位点所在竖向构件有交集的竖向构件。

考虑到多边形与多边形的取交集操作比起点与多边形的位置关系判断要耗时间得多，而且在测试数据中绝大多数情况下位点都是可以向下贯通的（through）的，所以在实际代码中，总是先查询是否有下层竖向构件可以包含某位点，如果已经找到了，就不必再计算竖向构件之间、竖向构件与横梁的集合关系，从而避开了大量的多边形与多边形的取交集操作。

2.8 依次寻找竖向构件并布置 B 类位点

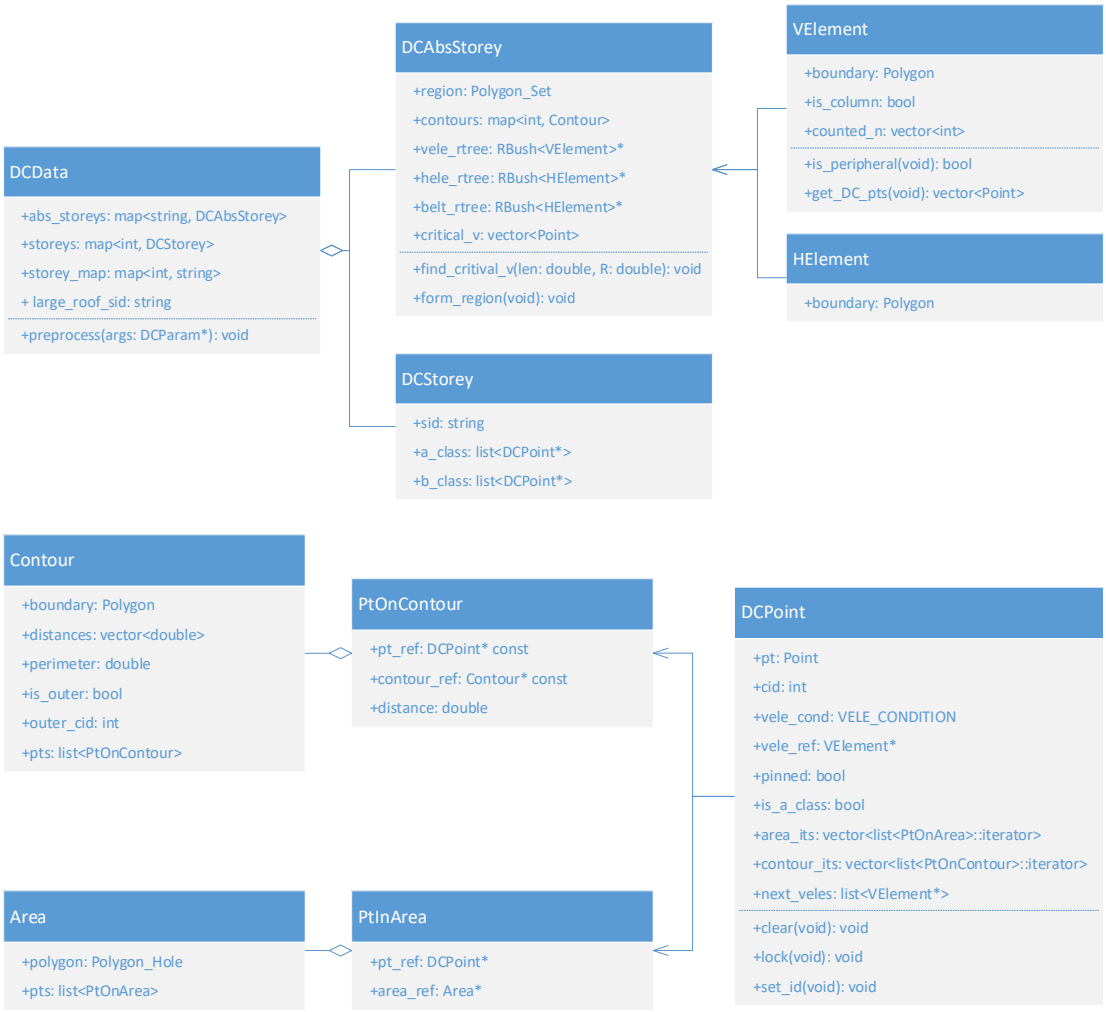
为了确保贯通构件中引下线的贯通性，程序会先处理贯通的（through）位点，半贯通（shift）的次之，断层的（disrupt）最后处理。对于每个 A 类位点的相连竖向构件列表，先按与外轮廓的距离由小到大排序，再按次序逐一检查，如果没有被布置过引下线位点，则选择此竖向构件并新建 B 类位点，并沿用对应 A 类位点的引下线编号。对于贯通的（through）位点，其坐标可以照抄；对于半贯通的（shift）和断层的（disrupt）位点，则需要根据选择的竖向构件重新计算位点坐标，计算方式与 2.3 节所述相同。若相连竖向构件列表为空，或者列表中找不到尚未布置的竖向构件，则放弃布置，也就是位点对应的引下线在此处终止。整个过程不再新增引下线编号。

3 数据结构与算法接口

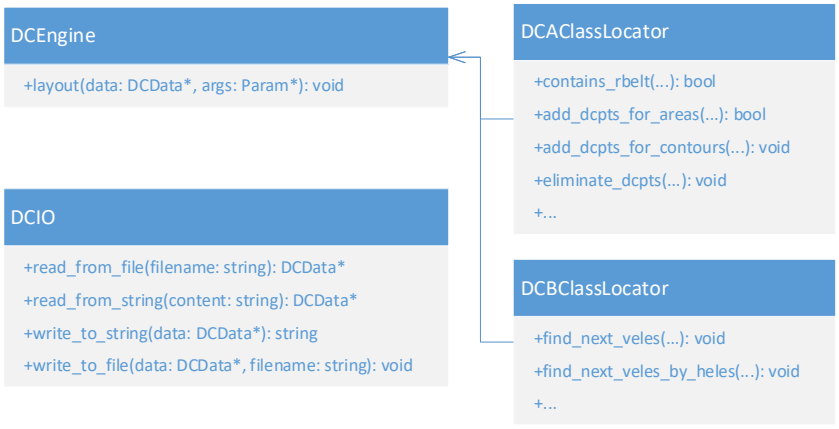
该项目采用以下一些类或结构来存储空间数据。DCData 是最顶层的数据结构，也是算法类调用数据时接收的数据类型。每套数据中用楼层 Id 来区分不同类别的楼层，结构 DCAbsStorey 封装了同一类楼层的信息，包含楼层所占平面区域、轮廓字典、所有竖向构件、所有横梁、所有接闪带、关键顶点列表等等。结构 DCStorey 则以楼层数为单位，主要封装了单个楼层的引下线布置结果。所有的柱和剪力墙实体用 VElement 结构来封装，该结构主要表达了竖向构件的外轮廓与从属轮廓信息，也包含了计算潜在引下线位点的方法。所有的横梁和接闪带实体用 HElement 结构来封装。同一类楼层的所有竖向构件、横梁和接闪带分别用三个 RBush 树来维护，运用这个数据结构可以加快空间搜索的效率。

一个基本的引下线位点用 DCPoint 类去表示，里面包含了位点坐标、引下线 id、类别（A 类还是 B 类）、是否上锁、所属竖向构件、到下层的贯通性等关键信息。在执行 A 类位点淘汰算法时，DCPoint 类需要与轮廓类和新增区域类紧密互动。Contour 封装了单个轮廓的重要信息，包括轮廓本身、各顶点里程数、周长、是否为外轮廓、对应外轮廓的编号以及从属于该轮廓的所有引下线列表。Area 则封装了一个区域的重要信息（本层轮廓比上一层轮廓多出的一个连通部分称为一个区域，相邻楼层之间可能会出现零个、一个或多个区域），包含区域对应的多边形、是否装有接闪带以及该区域内的所有引下线位点列表。DCPoint 结果分别通过 PtOnContour 和 PtInArea 两种结构与 Contour 和 Area 互为索引：PtOnContour 和

PtInArea 可以通过指针找到原本的 DCPoint 对象；DCPoint 也可以通过迭代器直接访问它对应的 PtOnContour 和 PtInArea 对象。除了必要的指针信息以外，PtOnContour 结构还封装了一个里程属性，来表征该引下线在该轮廓上的位置。



该项目的算法实现部分也用类进行了封装，实现了与数据分离的目的，并且按照算法的不同步骤和部分拆成若干的子模块，尽量满足了高内聚和低耦合。以下是本项目用到的所有算法类的类图。



为了将原始数据封装成 DCData 以供算法处理，本项目设计了 IO 类 DCIO，该类负责调用 read_to_*方法从文件或者字符流中读取数据，然后封装成 DCData*指针返回。

为了使算法的具体实现与数据结构分离，本项目设计了多个算法类，其中 DCEngine 是实际应用中的顶层类，充当了枢纽的角色，它的 layout 方法接收 DCData 指针 data 和参数结构指针 args，自动布置的结果保存在更新后的 DCData 指针当中，用 DCStorey 结构来封装。在 DCEngine 的 layout 方法中，需要调用一些重要的算法子模块，这些子模块也用类的方式各自封装起来，其中包括 DCAClassLocator 和 DCBClassLocator，分别负责自动布置单个楼层的 A 类引下线位点和 B 类引下线位点。

为了能够生成可视化结果，DCIO 还提供了 write_to_*方法，该方法负责将 DCEngine 的处理结果转换成 geojson 字符串，或者直接写入文件当中，以便测试端进行分析和调试。

类图中只列出了主干方法，一些辅助性的方法没有列出。

4.潜在问题

暂无。

5.开发配置

编程语言：C++

编程环境：Visul Studio 2010

所需软件包：boost_1_65_0-msvc-10.0-64、CGAL 4.9.1 64 位

撰写人：郑友怡团队 殷俊麟

撰写时间：2021 年 6 月 28 日星期一