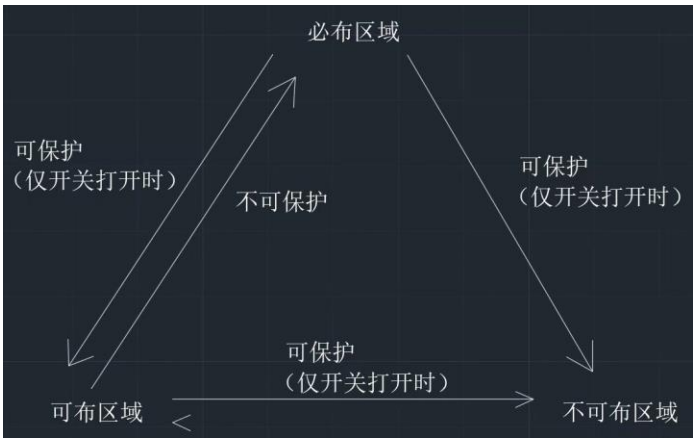
# 地下冲洗点位自动布置算法开发文档

## 1 背景介绍

本算法的目标是：根据一定的规则，为地下车库自动生成一套冲洗点位方案，为用户的后续加工提供基础。

地下车库由三类空间（Space）构成：建筑空间（隔油池、水泵房、垃圾房等）、停车空间和其他空间。其中**建筑空间等价于必布空间（即每个建筑空间必须布置冲洗点），停车空间等价于可布空间，其他空间等价于不可布空间。以上每组等价概念在下文和程序中可相互替代。**在必布空间中，冲洗点根据参数决定是否能保护可布空间和不可布空间；可布空间中，冲洗点无法保护必布空间，但可根据参数决定是否能够保护不可布空间。每个空间实例用封闭的多边形表示，但是可能会有“洞”的情况。下面是三类空间相互的保护关系图示。




每个空间实例当中存在四类元素：墙（ShearWall）、柱（Column）、障碍（Obstacle）和排水设施（DrainageFacility）。其中墙、柱、障碍和排水设施中的集水坑与地漏都用封闭多边形来表示，排水设施中的排水沟用多段线表示。一个墙、柱和障碍实例不会跨越两个或多个空间。所有冲洗点只能布置在墙和柱的边缘，或者整个空间的边缘，且要避开所有障碍。在停车空间当中，冲洗点要优先布置在集水坑或地漏附近，其次则是排水沟附近，再次是孤立墙和孤立柱的边缘，最后是空间轮廓上。同一个集水坑或地漏附近视情况可布置零个或一个冲洗点。除停车空间以外的其他空间则不考虑排水设施。

在上述要求和约束的背景下，设计算法自动完成布置冲洗点位，使得在给定的冲洗半径之下没有盲区（除非盲区不可避免），且布置的冲洗点位尽量要少。注意冲洗点提供保护时可以穿透所有的墙、柱、障碍和排水设施，不需要考虑遮挡效应。

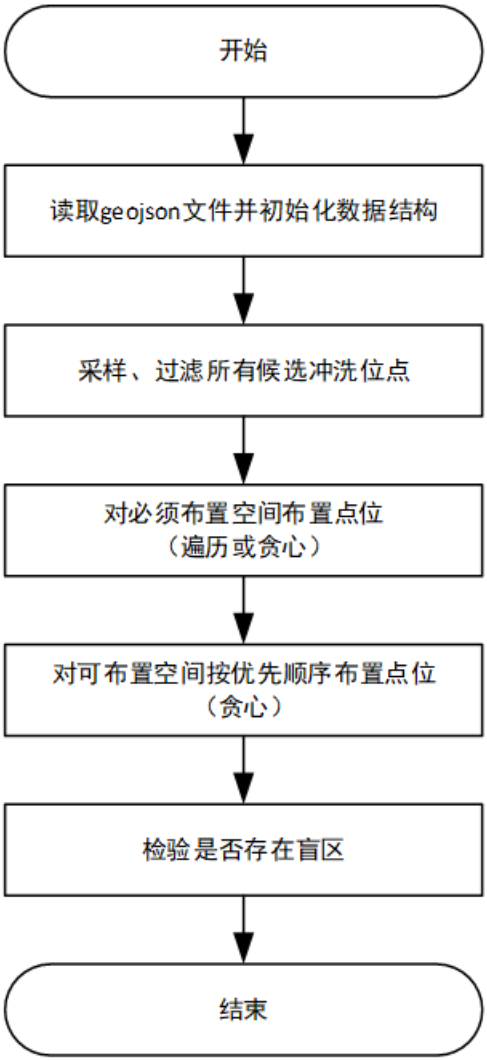
## 2 算法思路

### 2.1 核心思想

1. 变连续为离散，通过采样得到有限个候选位点，这样可以大大简化选择布置位点的过程；
2. 按照空间类型和优先级梯次布点，并与贪心算法相结合，尽量减少各个冲洗点保护区域的重叠；

3. 利用平面图形的集合操作来引导布点和检验效果，显著降低了编程和调试的难度。

## 2.2 流程图




## 2.3 读取文件与归并空间

算法的第一步就是读取 geojson 文件，然后对数据结构进行初始化。**注意在运行程序之前需要先将文件转换成 UTF-8 编码，否则中文方面的处理难以统一**。总共需要遍历两次 geojson 文件。这一步通过调用 GeojsonIO 类的方法来实现。

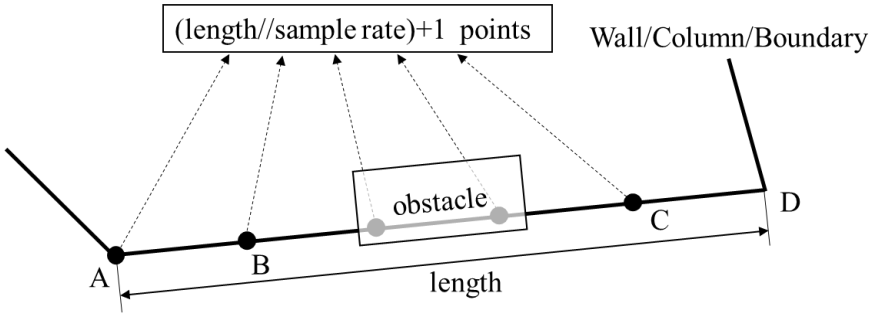
第一遍遍历文件时，将所有的空间按照空间名称添加进对应的向量里。对于所有的排水设施，通过比较第一个点与最后一个点是否相等，来判断是按照集水坑还是排水沟来处理，然后添加到对应列表里。第二遍遍历文件时，对于每一个墙、柱和障碍，找到其所在的空间，更新对应的空间数据。具体做法是：用均值求出墙、柱和障碍的中心，再遍历所有空间检验该中心点是否在某一空间以内（CGAL 库提供了相关的 API），若是，则代表整个墙、柱和障碍都严格在该空间内部。该逻辑由数据端保证。

为了**加快后续集合运算的速度，也为了算法的简化**，在处理之前要先进行空间的归并。归并的核心操作是要**将同一类的所有空间的对应区域集合取并集保存起来，包括当前未被**

**保护的区域、所有墙柱所占区域，以及所有障碍物所占区域**。注意只对可布空间和不可布空间进行归并，必布空间不归并，以便后续逐一单个处理。假设原来整个地下车库有 x 个必布空间，y 个可布空间，z 个不可布空间，那么归并以后就只有 x+2 个空间了。更新后的未保护区域，等于空间轮廓围成的区域，减去全局障碍的区域，再减去该空间中所有墙和柱的区域，区域的减法用平面多边形的差集实现。

## 2.4 候选点的采样与过滤

需求规定，所有冲洗位点只能依附在墙柱表面，或者位于空间的边缘上，但不能落在障碍物中。无论是墙柱表面还是空间边缘，都属于连续的空间要素，如果不对其进行离散化，那么就只能用线段集合来表示潜在位点的可能范围，这是极其繁琐和复杂的，也难以设计算法进行处理。所以，我们需要在布点之前先对空间中（这里的空间不包含不可布空间）所有的墙柱表面与空间边缘采样，并在采样的同时过滤掉不符合需求的点。




以上图为例，假设 AD 是一个孤立墙/柱或者空间边界的一条边，边的中间有一个障碍物。首先计算这条边的长度 length，将这条边**均匀地**分成 length//sample rate+1 段（符号“//”表示整除），然后从边的起点 A 开始，采得 length//sample rate+1 个等分点。对于每个等分点，判断它是否位于障碍物之内，如果是则舍弃掉。最后，AD 边经过采样和过滤，就得到了 3 个位点，即 A、B 和 C。注意 D 点是边的终点，它会在下一条边的采样中作为起点被取到，所以为了不产生重复样点，不能采到边的终点。无论是对必布空间还是可布空间采样，都需要对空间内所有可能的边进行上述操作，最后得到隶属于一个空间的候选点列表。

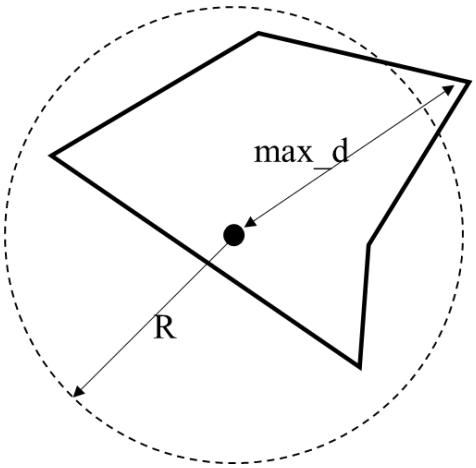
按照上述方式进行采样，不同边上相邻候选点之间的距离不是恒定的，而是依赖于所在边的长度在一个范围内波动。下表展示了边的长度与相邻候选点之间距离的关系：

| 边长 length | 候选点数量 N | 相邻候选点距离 Δ distance |
|---|---|---|
| [0*sample rate, 1*sample rate) | 1（即起点） | — |
| [1*sample rate, 2*sample rate) | 2 | [0.50*sample rate, 1*sample rate) |
| [2*sample rate, 3*sample rate) | 3 | [0.67*sample rate, 1*sample rate) |
| [3*sample rate, 4*sample rate) | 4 | [0.75*sample rate, 1*sample rate) |
| …… | …… | …… |

**可见相邻候选点距离总体上被控制在了采样率的一半到采样率之间，最稀疏时也不会达到采样率的水平。过低的采样率会导致无法对所有区域提供保护，但过高的采样率会加重计算负担，也会产生大量的无用点，显得没有必要。**所以，使用合理的采样率非常关键。

对于必布空间的候选点，在过滤时还需要增加一个判断条件，就是它与空间各顶点的最大距离不能超过冲洗点的保护半径。因为一个必布空间内只布一个点，而且至少要保证布置

之后能够完全覆盖该空间，所以一旦这个最大距离超过了保护半径，就至少存在一个点不能被保护到，下图就是一个例子，图中的这个候选点需要被舍弃掉。




对于可布空间的候选点，则还需要遍历全局所有的集水坑和排水沟，计算该候选点是否位于它们附近。如果是，则更新候选点对应的属性。另外，该点原本位于孤立墙柱上，还是位于空间轮廓上，也要明确。等到采样与过滤工作处理完毕之后，为了方便后续算法的操作，先拉通所有候选点进行排序，排序的标准是：首先按附近集水坑数降序排列，若相等则按附近是否有排水沟排列（有排在没有前面），若仍相等则按候选点的来源排列（孤立墙柱排在空间边界前面）。

**之后所有布点的步骤，都只会从这一步选出来的位点当中进行选择。**

## 2.5 对必布空间布置点位

通过 2.4 节的论述可以知道，对必布空间经过采样与过滤，得到的所有候选点都能覆盖它所在的空间。所以从理论上说，从每个必布空间的候选点中分别随机选取一个点，就能保证所有必布空间都被保护到。但是，实际的策略会因为用户设定的参数不同而有所调整。

如果参数设定，必布空间不能保护其他两类空间，那么确实只用管好自己就行。在实际编码中，为了提升程序的鲁棒性，算法会**遍历**每个空间，然后选取那个**与空间各顶点的最大距离最小**的那个点（以下简称 maxmin 点），来布置那个空间的冲洗点。算法伪代码见下图中的 MustLocateSpace 函数。

如果参数设定，必布空间可保护其他两类空间，尤其是当可布空间不能保护不可布空间时，就只能让必布空间来负责保护不可布空间。这种情况下，按照 maxmin 点来布置就很可能照顾不到不可布空间。取而代之地，本项目会采用一种**基于面积的贪心策略**来布点。具体地说，每次迭代拉通遍历所有必布空间的候选点，找到那个**圆形保护区域与不可布空间的交集面积最大**的点（以下简称 maxS 点）。之后选取并布置该点，更新未保护区域，再删去它所在空间的所有候选点（保证一个空间只布一个点）。循环上述操作，直到每个空间都已经布置了一个点为止，最后检查不可布空间是否被完全保护。算法伪代码见下图中的 MustLocateSpace_Extend 函数。

**1** Locating in a must-locate space.

**Input:**

List of the space boundary vertices, $V$;
List of candidate points, $C_{in}$;
Protection radius, $R$;
Uncovered regions, $U$;

**Output:**

List of chosen points, $C_{out}$;
**function** MUSTLOCATESPACE($V, C_{in}, R, U$)
    **for** each must-locate space $s$ **do**
        $c^* \leftarrow \min_{c \in C_{in}[s]}$ MAXDIST($c, V[s]$);
        update uncovered regions $U$ using $c^*$;
        add $c^*$ into $C_{out}$;
    **end for**
    **return** $C_{out}$;
**end function**

**function** MUSTLOCATESPACE_EXTEND($V, C_{in}, R, U$)
    **while** $C_{in}$ is not empty **do**
        $c^* \leftarrow \max_{c \in C_{in}} |U \cap Circle(c, R)|$;
        update uncovered regions $U$ using $c^*$;
        delete all candidate points that share the same space as $c^*$;
        add $c^*$ into $C_{out}$;
    **end while**
    **return** $C_{out}$;
**end function**

**function** MAXDIST($c, V$)
    **return** $\max_{v \in V} ||c - v||_2$;
**end function**

MustLocateSpace_Extend 函数的实现难点和性能瓶颈就在于计算带圆弧的平面图形交集的面积。CGAL 库没有提供这一步的 API，只提供了不带圆弧的版本。为了解决这一问题，目前有两种替代性方案：一是将圆近似成多边形，来迎合 CGAL 的 API 接口；二是借用蒙特卡洛采样的思想，在完全包含交集的区域中均匀采样，然后统计落入交集区域的比例，从而估算出交集面积。通过对比试验，暴力采样的方法能够在妥协一部分结果精确度的情况下，达到可以接受的时空复杂度；近似多边形的方法则不仅减慢了必布空间的处理速度，也极大减慢了可布空间的处理速度，这对于主要工作量在于可布空间的任务来说将是致命的。权衡以上几点，本算法将**采用暴力采样的方法来估算集合的面积**。

除了布点算法会根据参数不同而不同以外，更新策略和成功条件也会有所差异。下表详细展示了这几项在不同参数配置下的具体选择。

| 参数配置 | FFF | TTF | TFT | TTT |
|---|---|---|---|---|
| 布点算法 | 遍历<br>选取 minmax 点 | 贪心<br>选取 maxS 点 | 遍历<br>选取 minmax 点 | 贪心<br>选取 maxS 点 |
| 更新策略 | 仅更新本空间 | 更新本空间、可布空间和不可布空间 | 仅更新本空间 | 更新本空间、可布空间和不可布空间 |
| 成功条件 | 完全覆盖必布空间 | 完全覆盖必布空间和不可布空间 | 完全覆盖必布空间 | **完全覆盖必布空间** |

表中参数配置的三位布尔值分别依次代表：是否保护不可布空间、必布空间是否保护其他两类空间、可布空间是否保护不可布空间。注意当参数配置为 TTT 时，虽然采取贪心策略，但也只需要保证完全覆盖必布空间即可，因为可布空间也有机会去弥补不可布空间的缺漏。

## 2.6 对可布空间布置点位

在可布空间中，点位需要按照集水坑附近、排水沟附近、孤立墙柱位点和空间轮廓位点的优先顺序来选取与布置，可以看做**算法大框架的四个小阶段**。这四个阶段都采用贪心算法来处理，**算法的核心框架统一简洁，且不受参数配置的影响**，但是在初始化和某些具体策略方面略有不同。

统一而简洁的核心框架：代码的输入包含待布置的候选点列表 Cin，已经布置的点列表 Cout 和一个阈值$\varepsilon$。**设置阈值的目的是防止位点的布置过于密集，从而让程序不拘泥于某个阶段，去考虑视野以外的候选点。阈值的本质是优先级与最优解的权衡。**首先初始化 Cin 候选点对 Cout 中各点的最大最短距离 d。如果初始的 Cout 列表为空，则需要按照某种首点生成策略，从 Cin 中选一个点布置，并更新 Cout 和未保护区域。随后每次迭代中，首先进行判定，若未保护区域为空，或者 d 小于阈值$\varepsilon$，或者 Cin 为空，则跳出循环；若不跳出循环，则从 Cin 里选择出与 Cout 中各点的最短距离最大的点（以下简称为 maxmin 点）并将其移出 Cin、更新 d，然后判断是否布置该点。若该点不能更新未保护区域，或者 d 小于阈值$\varepsilon$，则舍弃不要；否则更新 Cout 列表和未保护区域。跳出循环之后，检验此时的未保护区域是否为空，并返回该布尔值。

四个阶段的区别：具体体现在代码框架中的 Cin 初始值、Cout 初始值、首点生成策略、阈值$\varepsilon$的取值上面。详见下表。

| | 第一阶段 | 第二阶段 | 第三阶段 | 第四阶段 |
|---|---|---|---|---|
| Cin 初始值 | 所有位于集水坑附近的候选点 | 所有不在集水坑附近，且位于排水沟附近的候选点 | 所有不在集水坑附近，且位于孤立墙柱上的未选的候选点 | 所有不在集水坑附近的未选的候选点 |
| Cout 初始值 | 空或沿用必布空间的 Cout（根据参数确定） | 沿用第一阶段的结果 | 沿用第二阶段的结果 | 沿用第三阶段的结果 |
| 首点生成策略 | 随机选取附近的集水坑数量最多的候选点 | 随机选取候选点 | 随机选取候选点 | 随机选取候选点 |
| 阈值$\varepsilon$的取值 | 大于集水坑的附近直径即可 | 大于采样率的一半即可 | 大于采样率的一半即可 | 等于 0 |

第一阶段中，Cin 只选取位于集水坑附近的候选点，Cout 根据具体的参数配置有不同的选择策略，详见后文；为了让布置位点尽量少，所以让第一个点就能涵盖尽可能多的集水坑；阈值$\varepsilon$大于集水坑的附近直径，保证了一个集水坑附近最多一个点，若第一阶段不能覆盖所有区域，则转至第二阶段；第二阶段中，Cin 只选取那些位于排水沟附近，且不在集水坑附近的所有候选点，Cout 则沿用前一阶段的结果，从第二阶段开始，首点生成均采用随机选取的策略，阈值$\varepsilon$理论上只需大于采样率的一半即可，实际情况可设置较大的值，这样可以接

近布点数量的最优解，若第二阶段不能覆盖所有区域，则转至第三阶段；第三阶段与第二阶段基本一致，唯一的不同在于 Cin 初始时包含上一步 Cin 剩下的，再加上所有位于孤立墙柱上的、不在集水坑附近的候选点，也就是说，排水沟区域里剩下的候选点还有机会被布置上，若第三阶段不能覆盖所有区域，则转至第四阶段；第四阶段的 Cin 在上阶段所剩的基础上，再加上所有位于空间边界上的、不在集水坑附近的候选点，由于没有下一个阶段，所以不再有阈值的限制，从而让程序尽可能地消灭所有未保护区域，直到所有候选点用光为止。单个阶段的算法伪代码见下图中的 GreedyKernel 函数。

**2** Locating in a could-locate space, each phase.

**Input:**

List of candidate points, $C_{in}$;
List of chosen points, $C_{out}$;
Uncovered regions (with spaces which can be protected), $U$;
Distance threshold, $\epsilon$;
(For phase 0 to 2, $\epsilon >$ candidate sample rate; for phase 3, $\epsilon \leftarrow 0$)

**Output:**

$C_{in}, C_{out}, U$;
Whether all the regions are covered, *succ*;

```
function GreedyKernel(C_in, C_out, U, ε)
    d ← +inf;
    if C_out is empty then
        generate c* according to First-Point-Generating-Policy;
        assert U ∩ Circle(c*, R) ≠ ∅
        remove c* from C_in;
        add c* into C_out;
        update uncovered regions U using c*;
    end if
    initialize MinDist(c_i, C_out) for all c_i ∈ C_in
    while (U is not empty) ∧ (d ≥ ε) ∧ (C_in is not empty) do

        /* find the next candidate point */
        c* ← arg max_{c_i ∈ C_in} MinDist(c_i, C_out);
        d ← MinDist(c*, C_out);
        remove c* from C_in;

        /* check whether to choose the point */
        if (U ∩ Circle(c, R) ≠ ∅) ∧ (d ≥ ε) then
            add c* into C_out;
            update uncovered regions U using c*;
            update MinDist(c_i, C_out) for all c_i ∈ C_in
        end if

    end while
    succ ← U is empty;
    return C_in, C_out, U, succ;
end function

function MinDist(c, C_out)
    return min_{c_j ∈ C_out} ||c − c_j||_2;
end function
```

和处理必布空间类似，可布空间的处理细节也会受到参数配置的影响，具体包含 Cout 初始值、更新策略和成功条件三项，下表详细展示了这几项在不同参数配置下的具体选择。

| 参数配置 | FFF | TTF | TFT | TTT |
|---|---|---|---|---|
| Cout 初始值 | 空 | **沿用必布空间的 Cout** | 空 | **沿用必布空间的 Cout** |
| 更新策略 | 仅更新可布空间 | 仅更新可布空间 | 更新可布空间和不可布空间 | 更新可布空间和不可布空间 |
| 成功条件 | 完全覆盖可布空间 | 完全覆盖可布空间 | 完全覆盖可布空间和不可布空间 | **完全覆盖可布空间和不可布空间** |

表中参数配置的三位布尔值分别依次代表：是否保护不可布空间、必布空间是否保护其他两类空间、可布空间是否保护不可布空间。注意当必布空间可以保护其他两类空间时，在计算 maxmin 点时需要考虑必布空间中已经布置的位点，这样处理可以**约束可布空间的冲洗点与必布空间的冲洗点保持一定间距**，从而可以进一步节省冲洗点的用量。另外，当可布空间能保护不可布空间时，不管必布空间能否延伸保护范围，**都由可布空间最终负责不可布空间的完全保护问题**。

## 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。

GeoData 是描述一个完整的地下车库所有数据的结构，它维护了建筑空间列表、停车空间列表和不可布空间列表，也存储了整个空间中所有的集水坑和排水沟。本项目的布点算法就主要接收一个 GeoData 结构的指针进行处理。

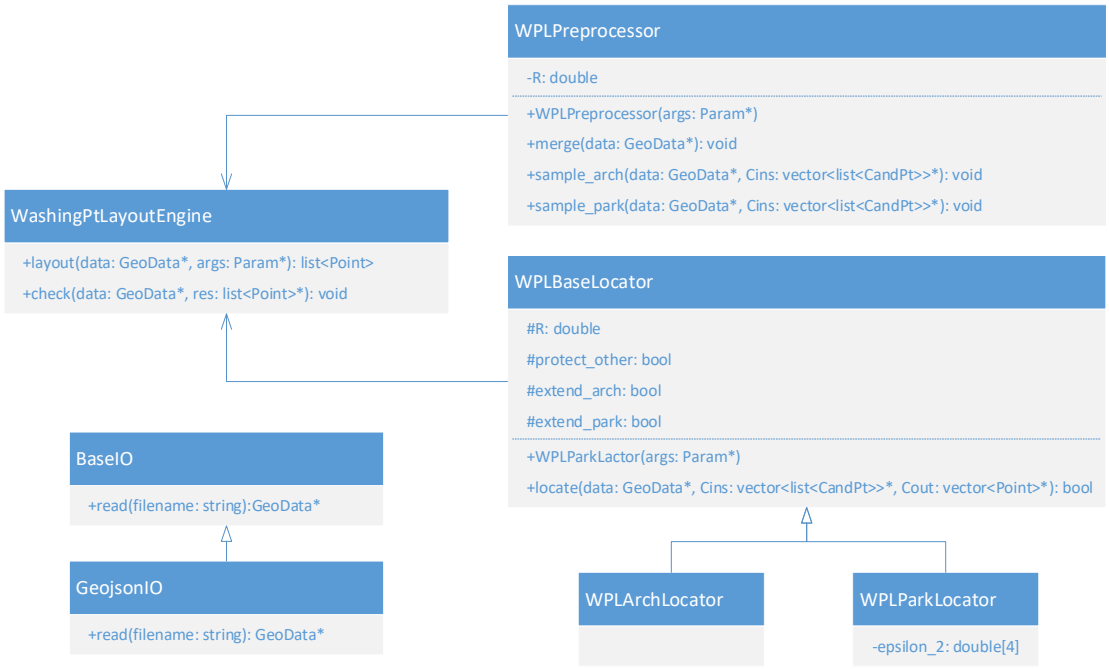
Space 是 GeoData 中与算法关联度最紧密，也是具有众多操作的结构，不同类型的空间共享同一种 Space 结构。它存储了该空间的边界轮廓列表（如果空间连通，则列表里只有一个多边形），以及在该空间以内的墙柱列表与障碍列表。为了后续的算法实现方便，Space 结构还维护了该空间中所有墙柱的区域集合、所有障碍的区域集合，以及当前未被冲洗点保护的区域集合。此外，Space 结构还封装了许多针对单个空间的操作，使得算法主体看起来更简洁易读，例如在读取文件加载数据时需要用到 add_col_wall 和 add_obstacle 方法，在归并空间时需要用到 merge 方法，在候选点采样与过滤时需要用到 test_candpt_in_obstacle 和 test_candpt_protect_all 方法，在更新未保护区域集合时需要用到 update_uncovered 方法。

集水坑 Sump 和排水沟 Drain 都属于排水设施，区分的标准是：集水坑的边会形成闭环但排水沟不会。考虑到具体操作的差异，集水坑和排水沟各用一个类来表示，里面主要存放各自的关键信息，以及判断一个点是否靠近它们的 is_near 方法。

CandPt 是描述一个候选的冲洗位点的结构，为了方便算法的实现，除了保存坐标以外，此结构还封装了一些关键属性，包括是否靠近集水坑与排水沟，靠近集水坑的数目，以及是否位于空间轮廓上。在可布空间中，所有候选点将根据这些属性按照优先顺序逐步选取和布置。为了方便贪心算法最优化目标函数，CandPt 还特意预留了一个属性来保存该点的目标函数值。

该算法采用结构 Param 来封装所有的参数，包括设置的保护半径（规定为 1 到 99 的整数）、是否保护建筑空间即必须保护空间（必须为真）、是否保护停车空间即可保护空间（必须为真）、是否保护不可布空间、建筑空间是否能保护停车空间和不可布空间、停车空间是否能保护到不可布空间（前者必须等于后两者的或运算）。本项目的布点算法需要接收一个 Param 结构的指针进行处理。


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 GeoData 以供算法处理，本项目设计了 IO 基类 BaseIO，在开发过程中，采用该基类的一个继承类 GeojsonIO，该类负责调用 read 方法从 geojson 文件中读取数据，然后封装成 GeoData 返回。后续的应用可以通过再继承 BaseIO 类，实现以 dwg 文件为数据源的封装操作。

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 WashingPtLayoutEngine 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 GeoData 的指针 data 和参数结构的指针 args，返回一个点位的列表，表示所有布置了冲洗点的坐标。用代码的方式呈现出来写法见下：

```
string filename;
GeojsonIO io;
WashingPtLayoutEngine engine;
Param args = { 30, true, true, true, false, true };

cin >> filename;
GeoData* data = io.read(filename);
list<Point> results = engine.layout(data, &args);
```

在 WashingPtLayoutEngine 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 WPLPreprocessor 与 WPLBaseLocator 两大类。WPLPreprocessor 类负责布点之前的一系列预处理操作，包括归并所有可布空间和所有不可布空间（调用 merge 方法）、对必布空间和可布空间采样过滤候选点（分别调用 sample_arch 和 sample_park 方法）。WPLBaseLocator 的 locate 方法负责调用遍历或贪心算法选取并布置，考虑到该方法的具体实现依赖于空间类型，所以又设计了两个继承类 WPLArchLocator 和 WPLParkLocator，对 locate 方法分别进行定义。

WashingPtLayoutEngine 还提供了 check 方法，可以帮助验证自动布置的冲洗位点能否完成用户指定的保护任务，并给出必要的信息作为反馈。

类图中的方法只列出了主干方法，一些辅助性的方法没有列出。

## 4.潜在问题

建筑空间（即必布空间）的布点必须满足的条件：一个建筑空间中必然存在一个合法的候选点，它可以覆盖到整个空间。如果建筑空间太大，或者保护半径太小，都会影响该条件的成立。

候选点的采样率会对布点结果产生重要的影响，如果最后的布点策略无法覆盖整个空间，则有可能是采样率过低导致的。

在集水坑附近布点时，一个点布置好，其他跟它位于同一个集水坑附近的所有候选点都不能再布置。另外，阈值的限制也可能将两个位于不同集水坑附近、但彼此相距较近的两个点排除掉其中一个。这些情况都有可能会导致最后保护区域无法覆盖整个空间。

在处理可布空间时，生成第一个布置点的随机性会给结果带来不确定性。

在估计集合的面积时，暴力采样带来的随机性误差会导致估算结果的精确度受损，严重时会干扰程序的贪心策略，从而不能完全保护到用户指定的所有区域。

**（以上所有与随机性有关的问题，可以用多次尝试的办法解决）**

## 5.开发配置

编程语言：C++

编程环境：Visul Studio 2019

所需软件包：CGAL5.2（用 vcpkg 安装到本地，生成 NuGet 包）

## 6.初步测试结果

本项目存在一些非用户指定的参数，在测试过程中取值情况如下：

| 名称 | 解释 | 取值 | 单位 |
|---|---|---|---|
| SUMP_NEIGHBOR | 集水坑的附近半径 | 5000 | 毫米 |
| DRAIN_NEIGHBOR | 排水沟的附近半径 | 5000 | |
| SAMPLE_RATE | 候选点的采样率 | 500 | |
| EPSILON_0 | 第一阶段的阈值$\varepsilon$ | 10000 | |
| EPSILON_1 | 第二阶段的阈值$\varepsilon$ | 5000 | |
| EPSILON_2 | 第三阶段的阈值$\varepsilon$ | 5000 | |
| N | 估算面积的采样点数 | 500 | 无 |

其他的参数组合，一是由于时间仓促，二是由于没有经验不知道微调的方向，所以没有尝试过，这部分可以后续跟进，以求达到最佳的功能和性能。

### 6.1 功能测试

对给到的 8 个测试样例分别做读取、归并和采样操作，分别在必布空间和可布空间采得候选点总数如下。

| 序号 | 文件名 | 必布空间候选点总数 | 可布空间候选点总数 |
|---|---|---|---|
| 1 | FL232CER | 276 | 9823 |
| 2 | FL2324OY | 356 | 8661 |
| 3 | FL2334X5 | 0 | 21136 |

| | | | |
|---|---|---|---|
| 4 | FL2340LH | 618 | 9656 |
| 5 | FL2341AP | 100 | 6003 |
| 6 | FL23269N | 242 | 4623 |
| 7 | FL23343J | 828 | 2175 |
| 8 | [无标识] | 208 | 2304 |

对这 8 个样例使用自动布点算法进行测试。首先固定保护半径 R=30 m，改变三位布尔值（即是否保护不可布空间、必布空间是否保护其他两类空间、可布空间是否保护不可布空间），统计必布空间和可布空间分别的冲洗点数量，以及检查是否完成覆盖任务。结果见下表。标红处为任务失败的部分。

| 第 1 组 参数配置：R=30 m，FFF | | | | | |
|---|---|---|---|---|---|
| 序号 | 文件名 | 必布空间冲洗点数 | 可布空间冲洗点数 | 总数 | 任务成功 |
| 1 | FL232CER | 3 | 26 | 29 | 是 |
| 2 | FL2324OY | 7 | 25 | 32 | 是 |
| 3 | FL2334X5 | 0 | 64 | 64 | 是 |
| 4 | FL2340LH | 12 | 0 | 12 | 是 |
| 5 | FL2341AP | 1 | 20 | 21 | 是 |
| 6 | FL23269N | 2 | 24 | 26 | 是 |
| 7 | FL23343J | 8 | 9 | 17 | 是 |
| 8 | [无标识] | 2 | 11 | 13 | 是 |
| 第 2 组 参数配置：R=30 m，TFT | | | | | |
| 序号 | 文件名 | 必布空间冲洗点数 | 可布空间冲洗点数 | 总数 | 任务成功 |
| 1 | FL232CER | 3 | 26 | 29 | 是 |
| 2 | FL2324OY | 7 | 746 | 753 | 否 |
| 3 | FL2334X5 | 0 | 64 | 64 | 是 |
| 4 | FL2340LH | 12 | 25 | 37 | 是 |
| 5 | FL2341AP | 1 | 22 | 23 | 是 |
| 6 | FL23269N | 2 | 28 | 30 | 否 |
| 7 | FL23343J | 8 | 65 | 73 | 否 |
| 8 | [无标识] | 2 | 13 | 15 | 是 |
| 第 3 组 参数配置：R=30 m，TTF | | | | | |
| 序号 | 文件名 | 必布空间冲洗点数 | 可布空间冲洗点数 | 总数 | 任务成功 |
| 1 | FL232CER | 3 | 21 | 24 | 否 |
| 2 | FL2324OY | 7 | 23 | 30 | 否 |
| 3 | FL2334X5 | 0 | 64 | 64 | 否 |
| 4 | FL2340LH | 12 | 0 | 12 | 否 |
| 5 | FL2341AP | 1 | 19 | 20 | 否 |
| 6 | FL23269N | 2 | 22 | 24 | 否 |
| 7 | FL23343J | 8 | 5 | 13 | 否 |
| 8 | [无标识] | 2 | 9 | 11 | 否 |
| 第 4 组 参数配置：R=30 m，TTT | | | | | |
| 序号 | 文件名 | 必布空间冲洗点数 | 可布空间冲洗点数 | 总数 | 任务成功 |
| 1 | FL232CER | 3 | 19 | 22 | 是 |

| | | | | | |
|---|---|---|---|---|---|
| 2 | FL2324OY | 7 | 742 | 749 | 否 |
| 3 | FL2334X5 | 0 | 64 | 64 | 是 |
| 4 | FL2340LH | 12 | 16 | 28 | 是 |
| 5 | FL2341AP | 1 | 20 | 21 | 是 |
| 6 | FL23269N | 2 | 32 | 34 | 否 |
| 7 | FL23343J | 8 | 22 | 30 | 否 |
| 8 | [无标识] | 2 | 10 | 12 | 是 |

根据这 4 组测试结果，有以下观察：

- 只要样本数据本身合理，本项目提出的算法完全可以保证必布空间和可布空间的自我覆盖。问题的关键在于，如何确保不可布空间被完全保护；
- 要求可布空间保护不可布空间之后，少数样本可以不增加冲洗点就能完成任务（如样本 1、3），一些样本可以增加少量冲洗点也能完成任务（如样本 4、5、8），但一些样本无论如何增加冲洗点都无法完成任务（如样本 2、6、7）；
- 要求必布空间保护不可布空间之后，所有样本都不能完成任务，但是几乎所有样本（样本 3 除外）的可布空间冲洗点数量会下降一些，这是由于必布空间的冲洗点也能保护到部分可布空间；
- 要求必布空间和可布空间共同保护不可布空间之后，一些样本通过增设可布空间的冲洗点能够帮助完成对不可布空间的全面保护（如样本 3、4、5、8），但也有些样本仍不能成功（如样本 2、6、7）。

为了探究不可布空间被保护的问题，我特意编程计算了每个样本中，不可布空间的所有顶点与两类空间的所有候选点可能出现的最长距离，统计结果如下表所示，单位为米。标红处为距离大于保护半径的部分。

| 序号 | 文件名 | 与必布空间候选点的最长距离 | 与可布空间候选点的最长距离 |
|---|---|---|---|
| 1 | FL232CER | 122.5 | 10.8 |
| 2 | FL2324OY | 208.0 | 185.8 |
| 3 | FL2334X5 | — | 2.85 |
| 4 | FL2340LH | 64.3 | 19.3 |
| 5 | FL2341AP | 188.3 | 19.1 |
| 6 | FL23269N | 103.9 | 33.2 |
| 7 | FL23343J | 66.9 | 61.0 |
| 8 | [无标识] | 65.5 | 23.1 |

可见除了样本 3 没有必布空间以外，所有样本的必布空间候选点都无法保护到不可布空间的所有顶点。此外，样本 2、6、7 的可布空间也存在这种情况。这也很好地解释了为什么只有样本 2、6、7 在两类空间都参与保护不可布空间的情况下，仍不能完成任务。

为了进一步算法的有效性，再做两组实验。单独抽出样本 4、7、8，将保护半径增大到 70 米，只用必布空间保护不可布空间，测试能否成功；另外单独抽出样本 6、7，将保护半径增大到 65，只用可布空间保护不可布空间，测试能否成功。两组实验的结果如下。

| 第 5 组　参数配置：R=70 m，TTF | | | | | |
|---|---|---|---|---|---|
| 序号 | 文件名 | 必布空间冲洗点数 | 可布空间冲洗点数 | 总数 | 任务成功 |
| 4 | FL2340LH | 12 | 0 | 12 | 是 |

| | | | | | |
|---|---|---|---|---|---|
| 7 | FL23343J | 8 | 1 | 9 | 是 |
| 8 | [无标识] | 2 | 0 | 2 | 是 |
| 第 6 组 参数配置：R=65 m，TFT | | | | | |
| 序号 | 文件名 | 必布空间冲洗点数 | 可布空间冲洗点数 | 总数 | 任务成功 |
| 6 | FL23269N | 2 | 4 | 6 | 是 |
| 7 | FL23343J | 8 | 8 | 16 | 是 |

可见只要保护半径大于可能出现的最长距离，算法就能够成功地实现全面保护，这进一步证明了本算法在功能方面的强大。不过到目前为止，**由于面积估算这一部分存在随机性，不能保证每一次尝试都会成功，但是重复两到三次通常都能试出成功的结果。**

## 6.2 性能测试

目前采用以下硬件和系统配置进行测试：

- CPU：Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz；
- 内存：16GB；
- 操作系统：Windows10 64 位。

分别选取 R=30 时，参数配置为 FFF 和 TTT 时测试给到的 8 个文件，分阶段统计所耗时间，其中 R&M 代表读取文件、初始化数据结构和归并空间三个步骤，S_A 和 S_P 分别代表对可布空间和必布空间进行候选点采样，L_A 和 L_P 分别代表对可布空间和必布空间进行自动布点算法，SUM 代表以上五个步骤加总起来的耗时，单位为毫秒。

| 第 1 组 参数配置：R=30 m，FFF | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 序号 | 文件名 | R&M | S_A | S_P | L_A | L_P | SUM | 任务完成 |
| 1 | FL232CER | 3059 | 23 | 6895 | 8 | 967 | 10952 | 是 |
| 2 | FL2324OY | 2848 | 26 | 14786 | 11 | 557 | 18228 | 是 |
| 3 | FL2334X5 | 9441 | 3 | 18347 | 2 | 27585 | 55378 | 是 |
| 4 | FL2340LH | 3113 | 40 | 8810 | 21 | 2 | 11986 | 是 |
| 5 | FL2341AP | 1205 | 8 | 713 | 4 | 308 | 2238 | 是 |
| 6 | FL23269N | 2055 | 23 | 3931 | 5 | 964 | 6978 | 是 |
| 7 | FL23343J | 1990 | 80 | 657 | 28 | 66 | 2731 | 是 |
| 8 | [无标识] | 2462 | 15 | 681 | 8 | 127 | 3293 | 是 |
| 第 4 组 参数配置：R=30 m，TTT | | | | | | | | |
| 序号 | 文件名 | R&M | S_A | S_P | L_A | L_P | SUM | 任务完成 |
| 1 | FL232CER | 3030 | 23 | 6771 | 5627 | 363 | 15814 | 是 |
| 2 | FL2324OY | 2743 | 30 | 14898 | 19340 | 39817 | 76828 | 否 |
| 3 | FL2334X5 | 9863 | 3 | 18773 | 1 | 28292 | 56752 | 是 |
| 4 | FL2340LH | 3012 | 36 | 8727 | 68063 | 3193 | 83031 | 是 |
| 5 | FL2341AP | 1190 | 7 | 725 | 1297 | 395 | 3614 | 是 |
| 6 | FL23269N | 2111 | 16 | 3975 | 3533 | 1593 | 11588 | 否 |
| 7 | FL23343J | 1881 | 67 | 625 | 47702 | 1048 | 51323 | 否 |
| 8 | [无标识] | 2519 | 17 | 714 | 5388 | 154 | 8792 | 是 |

根据这 2 组测试结果，有以下观察：

- 从采样方面考虑，可布空间的采样操作明显占据主要地位，这是由可布空间面积大，候选点数量多天然决定的。当然，降低采样率能明显提升这一部分的速度，但是需要权衡好与结果最优性的矛盾；
- 在必布空间无需向其他两类空间提供保护时，算法对必布空间的处理非常快；但反之则较慢，甚至耗时会超过可布空间的处理（如样本 1、4、5、6、7、8），这也体现了用暴力采样估计面积的美中不足之处。减少暴力的采样点数是一种缓解问题的权宜之计，但是相应地也会带来面积估算更加不精确的副作用；
- 当可布空间需要保护不可布空间，而且任务无法完成时，算法对可布空间的处理会明显减慢（如样本 2、6、7）；当任务可以完成时，大多数样本也会稍微变慢（如样本 3、4、5、8），也有极少数样本会更快（如样本 1），这可能跟必布空间也能保护到可布空间有关系。

综合上述所有分析，该项目设计的算法能够根据用户给定的参数，自适应地完成自动布置冲洗点的任务；在性能方面，对于绝大多数样本和应用场景能够达到可以接受的时空复杂度。不过局部功能模块的优化将会给功能和性能的提升带来更多的可能。

撰写人：郑友怡团队 殷俊麟

撰写时间：2021 年 3 月 28 日星期日