清华大学软件学院

Art-Gallery Problem 近似算法实现与验证

作者: 张泉

学号: 2020312636

日期: June 18, 2021

1 问题介绍

1.1 Art Gallery Problem

1.1.1 问题介绍和定义

Art Gallery Problem（艺术画廊问题）的现实情景是在一个艺术馆内，希望用最少的监控摄像头监控整个博物馆。它在现实生活中有大量的应用场景，包括摄像头或无线天线的放置，城市规划等。具体来说，我们定义一个简单多边形为 P，由点集 $S_p = \{p_1, p_2, p_3, ..., p_n\}$ 个点按照逆时针方向连接而成。首先我们定义可见图，对于任意在 P 内或者 P 边界上的点 G，设其可见图为 $V_P(G)$，则对于任意 $x \in V_P(G)$，其与离散点 G 连线应全部位于多边形 P 内部或者多边形 P 边界上。而对于 Art Gallery Problem，我们希望找到一个最小的点集 $G = \{g1, g2, g3, ..., g4\}$，其中 g_i 都在多边形 P 上或者或 P 的边界上，我们计算这些点的可见图的并，并希望并出来的多边形等于简单多边形 P。

Art Gallery Problem 还有大量的变种，其中研究较多的包括正交多边形艺术画廊问题（Orthogonal Art Gallery Problem），端点艺术画廊问题 (Vertex Art Gallery Problem)，边艺术画廊问题（Edge Art Gallery Problem）。同时，根据处理的简单多边形是否带洞，问题的解决难度和解决方案都有较多的不同。在本文中，我们主要关注点在不带洞简单多边型上的 Vertex Art Gallery Problem。

1.2 相关工作

关于 Art Gallery Problem 的研究最早由 Victor Klee 于 1973 年提出，在两年后就由 Chvátal 证明在简单多边型内其需要点个数的上界为 $\lfloor (n/3) \rfloor$ [1]。而 Fisk 也在 1978 年借助图染色算法证明了这个上限 [2]。虽然确定了上限，但 Art Gallery Problem 也被证明为了 NP-hard 问题 [3]，这意味着想要真的寻找到最优解需要极大地代价。

在被证明为 NP-hard 问题之后，研究者们的研究方向也转变为研究与现实更为接近的 Vertex Art Gallery Problem 等变种问题的最优解范围界定和更高效精确的近似解计算算法。其中 Ghosh 提出了一个近似度为 $log(n)$ 的近似算法，算法的复杂度为 $O(n^4)$ [4]。在 2007 年，由 Couto 提出将 Art Gallery Problem 问题转化为集合覆盖问题 [5]，具体的转化思路是通过设置离散点，寻找能够覆盖所有离散点的最小的守卫点点集。但是在 2007 年，作者只在正交多边形中进行了 Vertex Art Gallery Problem 的研究。在 2011 年，作者再次将他推广到了一般普通多边形中 [6]，同时保持了高效的性能和精确的结果。而我们复现的主要部分也是这篇论文的计算过程。

2 算法说明

算法的整体思路如下：（1）首先选择初始的离散点和初始的候选守卫点，离散点用于评估整个多边形的覆盖情况，候选守卫点中将会选取出最终的守卫点，我们假设选取候选点集 $C = \{c_0, c_1, ..., c_n\}$ 和离散点集 $W_0 = \{w_1, w_2, ..., w_3\}$。（2）完成初始化后，我们尝试寻找能覆盖当前离散点所需的最少守卫点集 G_i，这个步骤需要将问题转化为集合覆盖问题并使用整数规划进行

求解。（3）获得当前所需最少的守卫点后，由于覆盖离散点并不能保证覆盖了整个多边形，我们检查当前未覆盖的部分，添加新的离散点来代表这些部分，构成新的离散点集 W_i，然后回到步骤（2）尝试计算新的最小守卫集。(2)、(3) 步需要持续多次直到计算的守卫点集能够覆盖整个多边形。

在算法中，第二步的求解需要进行转化，首先我们需要计算每个守卫点和离散点的可见关系，构建可见矩阵：

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \tag{1}$$

在矩阵中，每列表示了每个候选守卫点的可见性，每行代表每个离散点的可见性。每个元素 $a_{i,j} \in \{0, 1\}$, 其代表离散点 w_i 和候选守卫点 c_j 是否相互可见，可见为 1，不可见为 0。我们希望利用最少的守卫点能够覆盖所有现有的离散点，这个问题可以抽象以下的整数规划（Integer Programming) 问题：

$$\begin{aligned} z = & min \sum_{j \in C} x_j \\ & s.t. \sum_{j \in C} a_{i,j} x_j \geq 1, \ for\ all\ w_i \in W \\ & x_j \in 0, 1, \ for\ all\ j \in C \end{aligned} \tag{2}$$

在公式中，每个 x_j 代表每个候选离散点是否选择，若选择则为 1，未选择则为 0。此时的最优化目标就是希望 x_j 中为 1 的点数尽量的少。最优化问题中的约束是要求对于一个离散点 w_i，要求至少有一个与它相互的见的离散点被选为最终的离散点。

但由于覆盖离散点集 W 并不代表当前的守卫点能够覆盖整个多边形，单次求解的结果并不一定为正确的解，故当我们发现有未覆盖的区域时，我们需要添加新的离散点，约束最优化过程考虑新添加的离散点并求新的守卫点集，即持续算法的（2）、（3）步直到当前的守卫点集能够覆盖整个多边形。

在整个算法中，需要大量的多边形操作，其中开始初始化需要计算所有守卫点，此时计算所有多边形可见图复杂度为 $O(n^2)$。随后在求解可见矩阵中，需要判断点是否在可见图内，整体复杂度为 $O(mnlongn)$。整个算法最为复杂的部分仍为整数规划求解，特别是有较多的点数时，其计算速度较慢。但论文中并没有给出相关复杂度的分析，整体优化算法复杂度在最差情况下可能会为为 $O(n^4)$。

3 数据结构与模块设计

我们将从算法实现和前端交互来详细介绍我们的项目。

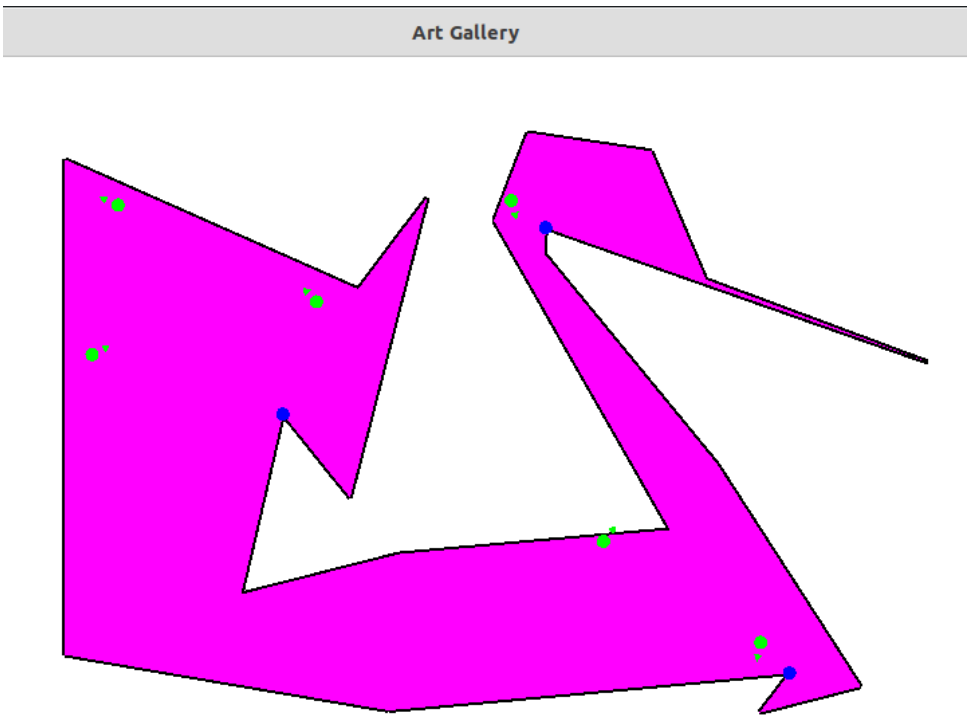

图 1: 运行示例

3.1 整体

本次大作业整体被实现为系统原生应用，支持在 Linux 平台跨平台编译运行。系统运行时核心包含三类对象：Gallery、守卫点、入侵者，如图 3所示。其中 Gallery 为整个场馆，场馆形状可外部导入与交互变换；守卫点依附于 Gallery 且具有视野，由核心算法基于 Gallery 当前形状计算获得，视角为每个守卫点在场馆内的可见范围，在图中体现为紫色的区域，当前示例中场馆的所有区域均可见因此整个场馆的颜色为紫色，可见性核心依赖于 visibility 算法；入侵者为场馆内随机移动的一些点，其移动方式基于游客建模，其颜色根据所在位置不同会有所不同，当入侵者在守卫点视角内时，颜色为绿色，代表其不具有危险性，否则为红色，代表当前未侦测到该点，由于当前守卫点可以观测真个场馆，所以所有的入侵者均为绿色。为充分展示算法能力，本次大作业实现了一定的交互性，包括画面移动、基于点边拓展的场馆形状构造、守卫点移动 删除、入侵者移动删除等，与交互动作对应的状态更新逻辑也有充分的支持，从而实现任意输入数据的构造，此外本次大作业还实现了历史状态记录，支持操作回退。

本次大作业整体采用 Rust、C++ 语言实现，最终的实现包含 2500 行 C++ 代码、1200 行 Rust 代码。其中 C++ 的部分用于实现整个系统的核心算法，Rust 部分用于实现交互逻辑、状态维护更新与 UI 展示，两个部分代码通过特定的 FFI 代码结合。本项目主要关注于近似算法中建模和求解部分的实现，故我们部分借助了 CGAL 库提供的基础类型和相关操作辅助开发，利用 CGAL 提供的功能还能大大的提高本次大作业实现的算法的精度，此外，由于本次大作业所采用的算法需要求解线性方程组，因此系统还依赖了 GLPK 库进行线性方程求解。为实现灵活的交互操作，本次大作业使用了较为 low-level 的图形操作渲染 API，从而可以实现较为细节的界面控制。

3.2 近似算法设计与实现

近似算法的整体流程包括离散点和候选守卫点的初始化，最优守卫点求解和未覆盖区域计算，我们将在本章中介绍这些步骤。同时在这些步骤中，我们都需要借助多边形可见性算法，计算某一点其在一个多边形内或者边界上的可见区域，故我们会额外介绍可见性算法相关实现。

3.2.1 初始化

在初始化时，我们需要设置所有的初始离散点和候选守卫点。在实现中，我们将 P 中所有的端点作为候选守卫点，同时将他们全部选为离散点。在此时看似方法比较暴力，实际在后继进行求解之前，会对候选守卫点和离散点进行去重，去除功能相同的点，故此时我们选择直接添加所有的端点。在初始化过程中，我们使用 vector 直接存储每个点的指针，因为不会删除离散点和候选守卫点，故使用 vector 效率足够。在完成点的选择后，为了方便后继的计算，我们会计算所有离散点的可见图并进行暂存，每个可见图都使用 Polygon 进行存储。存储每个离散点和候选守卫点的可视图，我们使用 unordered_map 数据结构进行维护，为此我们将点转为字符串以方便进行哈希。

3.2.2 整数规划求解

在这一部分，算法需要将各离散点与候选守卫点的可见关系转换为可视矩阵，并利用求解整数规划问题来求解最优守卫点。整体的过程可以大致分为四步：基础可视矩阵的构造、守卫点约减、离散点约减和矩阵求解。

首先，我们需要确定每一个候选守卫点和离散点的可视关系，构造基础的可视矩阵。具体地，算法将迭代每一个离散点 i，确定可以看到该离散点的可视多边形，并判断各候选守卫点是否在多边形内。如果某观候选守卫点 j 在可视多边形内，则说明 j 可以看到当前迭代的离散点 i，相应矩阵元素 $a_{i,j}$ 即设定为 1；如果不在多边形内，则相应候选守卫点看不到当前迭代的离散点，相应矩阵元素值即设定为 0。

在构造完毕基础可视矩阵后，随即算法将进行守卫点和离散点的约减。在整数规划求解问题中，矩阵的维数将极大影响求解的效率。而在可视矩阵中，行数即为离散点的个数，列数即为守卫点的个数。如果可以将重复的守卫点与离散点进行约减，也就能够压缩求解矩阵的维数，进而缩短求解最优守卫点的时间。在可视矩阵中，每一行的各元素代表着**该离散点能否被各守卫点看到**的情况。算法将枚举每两个离散点 i,j 所对应的基础可视矩阵中的两行 $a_{i,*}$ 和 $a_{j,*}$。将枚举出的两行元素一一对应做或运算，如果运算结果与 $a_{i,*}$ 完全相同，则说明如果守卫点能看到 i，那么必然守卫点一样可以看到 j，因此 j 离散点为多余的离散点，可以进行约减；反之亦然。同时在可视矩阵中，每一列代表着**该守卫点能否看到各离散点**的情况。算法可以枚举每两个守卫点 i,j 所对应的基础可视矩阵中的两列 $a_{*,i}$ 与 $a_{*,j}$。将枚举出的两列元素一一对应地做或运算。如果运算结果与 $a_{*,i}$ 相同，则说明守卫点 j 能看到的离散点必然也能被守卫点 i 看到，此时守卫点 j 即为多余的候选守卫点，可以进行约减；反之亦然。这样的过程可以有效压缩需要计算的可视矩阵的维数，提高后续矩阵求解的效率。在运算过程中，我们使用 bitset 压缩可是矩阵并进行二进制运算的加速。

随后，算法即进行最终的矩阵求解工作，利用公式2所给出的条件来优化求解出最少的守卫点个数及最优的守卫点位置。具体的矩阵求解可以使用各类整数规划求解算法，在本算法实现中使用的是 GLPK 线性规划求解器进行计算，最终的结果将会以一个单列矩阵的形式给出。矩阵中的每个元素代表着一个候选守卫点。如果元素值为 1，则说明该候选守卫点被选中，应当作为最终的守卫点来使用；如果元素值为 0，则说明该候选守卫点未被选中，不在最终守卫点结果的列表中。

3.2.3 未覆盖区域计算

由于覆盖离散点并不能确定整体多边形已经被覆盖，我们需要利用当前守卫点的可视图确定没有覆盖的区域。我们首先借助可见性算法寻找可见多边形，具体计算算法在下章进行介绍。获得可见多边形后，每个可见多边形都为简单多边形，我们可以进行简单多边形求并的计算来获得所有选中守卫点的可视区域。简单多边形交需要维护一个多边形集合，每次插入都需要进行大量的多边形求并的运算，我们本次复现的核心是关于近似算法的实现，故多边形求交的算法我们借助直接使用 CGAL 中的 PolygonSet 数据结构和其实现的 join 操作进行维护。获得所有选中守卫点的可视区域后，我们只需要与整体多边形进行 difference 操作，即可获得未覆盖的所有区域。

获得每个未覆盖区域后，如果未覆盖区域为空，则代表我们已经找到符合要求的最少点，此时我们可以结束迭代，返回一个数组代表我们需要的守卫点集。若未覆盖区域不为空，则针对每个未覆盖区域，我们需要寻找一个点代表这块区域加入离散点，以再下一轮迭代中确保覆盖此多边形。此时的未覆盖区域为多个简单多边形。此时我们需要分为两个情况进行讨论，首先如果当前多边形为凸包，我们只需要将此多边形的重心作为新的离散点；但当当前多边形为非凸简单多边形，我们需要寻找非凸多边形内部最大圆的圆心作为新的离散点。为了方便实现，我们并没有找到真正最大圆圆心，而是选择在某个点开始直接寻找当前点能够构成的的凸包，并在凸包内寻找圆心。其算法复杂度为 $O(n)$，实现较为简单。对于每个未覆盖区域，我们添加一个离散点，我们希望下次迭代在考虑了这些点后，能够获得对整个多边形完全覆盖的守卫点集。

在获取了新的离散点后，我们需要构建新的可见性矩阵中以进行下一次计算。由于我们只是添加了部分新的离散点，并没有修改候选守卫点，故我们只需要在原矩阵中添加部分新增加的离散点的数据即可。其添加过程与节3.2.2中相同，随后我们也会重新执行压缩算法降低数据维度。

3.2.4 可见区域计算

可见性算法 1 该算法 [7] 主要计算简单多边形内一个点的可视范围。通过迭代每个顶点，适当地插入新的“可视点”，利用栈来维护当前迭代可以被看到的顶点及可视点，在迭代完毕后即可确定出构成可视范围多边形的各个顶点。算法简述如下。首先，记需要计算可视范围的点为 z，将多边形的各顶点标做 $u_0, u_1, ..., u_n$，考虑在一次迭代结束后，下次迭代的点为 u_i，此时栈中存在一些点 $s_0, s_1, ..., s_j$，我们将这样的状态记作 $(u_i; s_0, s_1, ..., s_j)$。判断 u_{i-2} 在向量 $\overrightarrow{z, s_j}$ 的哪一侧。将其在左侧的情况记为 C1，在右侧的情况记为 C2。

在 C1 情况中，$z, s_0, s_1, ..., s_j$ 会围成一个多边形 Z，将平面区域分为多边形 Z 以内与 Z 以外两个区域。记 Z 以内的区域为 $R1$。在 Z 以外的区域中，平面又进一步被 z 与 s_j 所构成的线段分为左右两个区域，分别记作 $R2$ 和 $R3$。考虑 u_i 在哪个区域。(1) 如果 u_i 在区域 R1，则 z 无法看到 u_i。我们继续从 u_i 开始扫描各顶点直至出现 u_k 点，线段 $\overline{u_{k-1}, u_k}$ 与从 z 点与 s_j 点无限延伸出的线在逆时针相交，记交点为 v。下一组迭代状态即为 $(u_{k+1}; s_0, s_1, ..., s_j)$。(2) 如果 u_i 位于 R2 区域，下一组迭代状态即为 $(u_i; s_0, s_1, ..., s_j)$。(3) 如果 u_i 位于 R3 区域，则 $\overline{u_{i-1}, u_i}$ 将会挡住当前栈中的一些点，使其变得不可见。算法将循环弹出栈中的元素，直到元素 s_m 使得 $\overline{u_{i-1}, u_i}$ 与 $\overline{s_{m-1}, s_{m+1}}$ 交于点 v，或者 u_i 位于 $\overrightarrow{z, s_m}$ 的左侧。在前者的情况中，u_i 本身是不可见的，需要继续扫描多边形端点直至 u_k 使得 $\overline{u_{k-1}, u_k}$ 与 $\overline{s_m, v}$ 逆时针相交于 w。在这种情况下，下一组迭代的配置即为 $(u_{k+1}; s_0, s_1, ..., s_m, w, u_k)$。在后者的情况时，记 $\overline{s_m, s_{m+1}}$ 与 $\overrightarrow{z, u_i}$ 交于点 v，下一组配置即为 $(u_{i+1}; s_0, s_1, ..., s_m, v, u_i)$。

在 C2 情况中，u_{i-2} 位于 $\overrightarrow{z, s_j}$ 的左侧。我们依然将此时的平面划分为 3 个区域，R1 指的是 s_{j-1} 点与 s_j 点所夹的多边形边与 $\overline{s_j, s_{j-1}}$ 所构成的区域；R3 指的是 $\overline{z, s_0}$, 点 s_0 与 s_{j-1} 所夹的多边形边与 $\overline{s_{j-1}, z}$ 所围成的区域；R2 指平面剩余的区域。根据 u_i 点位于哪个区域中，我们将这一部分划分成三种情况来阐述。（1）u_i 位于 R1 区域。此时 u_i 将变得不可见。算法将继续扫描多边形顶点直至出现顶点 u_k，使得 $\overline{u_{k-1}, u_k}$ 与 $\overline{s_{j-1}, s_j}$ 交于 v 点。后续的处理与 C1 情况的 (3) 完全相同，只需将 v 看作是 s_{j-1} 即可。（2）u_i 位于 R2 区域。此时下一组的迭代状态即为 $(u_{i+1}; s_0, s_1, ..., s_j, u_i)$。（3）$u_i$ 位于 R3 区域。后续的处理步骤与 C1 情况的（3）相同。通过这样的迭代过程，我们即可以确定组成 z 点的可视多边形各顶点的位置。

可见性算法 2　经过查阅资料，我们发现可见性问题有较多其他解决算法，且复杂度各不相同。我们又实现了一种基于扫描射线的可见性算法 [8]，整个算法通过将点按照极角序排列，随后逐步扫过每个点，添加新遇到的点上开始的边并计算当前在最前方的边，同时删除已经扫描完的边。在这个过程中，我们需要动态维护现有扫描线段的可见顺序，同时确保在边界上的点不会扫描到简单多边形外部的点。具体步骤如下，我们利用一个 set 来维护当前的可见边，首先将所有边进行极角排序，遇到一个边，首先在 set 中去除已插入的线段，并更新当前可见的边，同时插入从此边开始的边并更新可见线段。在执行两者时，根据可见线段是否发生变化，确定最新的可见区域的边界。最后返回一个多边形代表可见区域。此算法在处理点在边界和边上时需要较为细致的处理。

3.3 交互

本次大作业实现的核心交互操作为画面移动、基于点和边拓展的场馆形状构造、守卫点移动和删除、入侵者移动和删除、状态记录和操作回退。由于系统需要实现的 UI 展示需要较为细节的控制，例如入侵者移动于不同区域时不同颜色的展示，本次大作业选择了较为底层的 API ——Piston 来实现。Piston 是 Rust 语言支持的一个跨平台游戏引擎，利用该库即可以灵活地捕捉外部事件、图形渲染，但随之带来的一个坏处是使用较为繁琐。为了解决该问题，我们在实现交互时采用 MVC 的设计思路，尽管 Piston 库本身并不是 MVC 风格的框架。具体来说，我们把整个交互模块划分为上层事件监听分发模块（C）、状态更新模块（M）、状态渲染模块 (V)，事

件监控分发模块负责监听各类交互事件，例如鼠标点击、键盘按下松开等，并将对应的事件转换为更新逻辑输入，使用更新逻辑跳转整个系统的状态，最后渲染模块负责更具当前状态，在前端展示界面。

在事件监听分发的处理中，实现最为复杂的部分为鼠标事件的响应。本次大作业支持用鼠标拖拽来改变场馆形状，拖拽时根据点击的不同对象产生有不同的响应。拖拽场馆的边框将为场馆新增点和边，拖拽场馆的点，将实现场馆点的移动，拖拽守卫点或入侵者将实现相应元素的位移操作。此外，双击鼠标左键可删除特定对象，例如场馆边点、守卫点、入侵者，上述功能的实现都依赖于对鼠标事件的处理响应。特别地，在选择对象判断的代码逻辑实现中，我们曾错误地将理论直接套用，忽略了现实本身的一些物理特性。例如，在判断是否选中边时，我们初始通过简单的点与边的位置关系（ToLeft 测试）来实现，导致实际交互时“拖拽不灵敏”。在实际的交互过程中，很难选中只有几个像素点宽的边。因此上述所有交互都需要考虑实际交互需求，我们通过定义“直觉上友好”的一些超参数来近似所有的相对关系的判定。例如判定是否选中点时，如果距离已在 S 范围内，则已经选中了响应的点。通过判定两次鼠标点击是否超过 200 毫秒实现双击事件判定，通过判断第一次鼠标点击到鼠标释放时鼠标移动的距离是否超过 15 个像素点，结合相对位置来判定实现拖拽事件。

状态更新部分核心需要实现整个状态的更新，状态渲染则展示更新后状态。视觉中的敌人移动、颜色变化等，均需要响应的更新逻辑实现，这部分最为核心的部分为当场馆形状变换后重新计算所有守卫点，当守卫点位置移动后，重新计算其视角，以及一定频率的日常状态更新（入侵者移动等）。守卫点计算与守卫点视角计算通过调用 C++ 语言实现的算法完成，日常更新核心完成三部分：删除场外入侵者（交互移动道中）、移动现有入侵者、新增入侵者，其中移动通过建模普通游客行为实现，例如范围内随机频率方向变换、位置移动。在每次状态更新后，状态渲染部分利用 Piston 提供的几何图形绘制 API 将模型基于当前状态进行展示，这部分需要确认一定的渲染顺序，例如先渲染视角而不是先渲染守卫点，否则将发生守卫点被覆盖的情况，状态也需要被正确的考虑，例如入侵者当前是否暴露。

4 鲁棒性测试

鲁棒性测试的核心目的为确认当前算法实现在任意“奇怪形状”的输入下依然可以输入的正确结果，确认程序是否会在这些极端输入下发生越界访问、段错误等常见的 C++ 程序漏洞。为做到这一点，我们设计了两部分实验进行鲁棒性测试，第一部分为代表性极端样例测试，第二部分为模糊测试（Fuzz）。在进行两部分实现时，我们为 C++ 代码编写了对应的测试驱动，在编译时开启了多种检查机制，例如 address sanitizer、leak sanitizer、thread sanitizer 等，从而保证算法的对应类型错误可以被及时的捕获。此外，由于本次大作业实现的是近似算法，因此这两部分鲁棒性测试的实验关注与程序是否会崩溃而不是输出是否完全符合预期，因为确定近似算法的每个测试的输出较为困难。

实验设计

实验的第一部分为代表性极端样例测试，核心思想是利用等价类将输入空间进行划分。在每一个等价类中，选择较为代表性的样例对算法进行测试，以此获得程序在这些等价类上表现

表 1: 性能评估数据集各组样例数

	随机简单多边形	随机正交多边形	完全科赫曲线	随机科赫曲线
20	2	2	1	2
40	2	2	-	2
60	2	2	-	2
80	2	2	-	2
100	2	2	1	2
200	2	2	-	2
300	2	2	-	2
400	2	2	-	2
500	2	2	1	2
600	2	2	-	2
700	2	2	-	2
800	2	2	-	2
1000	2	2	-	2
2500	-	-	1	-

的鲁棒性情况。我们将输入划分为随机简单多边形、随机正交多边形、完全科赫曲线、随机科赫曲线这四类。对于其中每一类，我们通过参考开源样例、手动构造等方法，设置了 20 到 1000 等不同数量级别边数的对应类型的多边形。在每个类别的每个数量级，我们保证均有两个样例，最终构造了 82 个样例输入，相应信息可参见表 1。

实验的第二部分为模糊测试（Fuzz）。模糊测试通过向算法不断的输入随机数据，监控算法执行过程中是否会崩溃。若崩溃则说明算法存在漏洞，在某些输入下表现较差，鲁棒性需要提升。我们首先编写了随机简单多边形生成器，这一部分能生成多边形，不断向算法生成输入，随后构造上层算法调用的驱动程序并插入 Sanitizer，最后利用脚本不断调用生成器测试算法，监控运行情况，捕获程序异常。此外，为了确保整个过程覆盖了足够多的输入空间，这一部分实验持续进行了 24 小时。

数据集

表一列出了我们设计的各个类型不同边数量的输入集的信息，图 3展示了数据集中的三个例子。

结果分析

整体而言，我们的最终算法实现在两部分实验表现均较好，算法并没有在极端样例下崩溃或长时间无法获得结果，也没有因为 Fuzz 的随机输入导致段错误、越界等，在 82 个极端样例输入、24 小时的模糊测试过程中，算法均正常退出。因此，通过这部分实现，我们可以初步得出当前算法实现较为鲁棒的结论，算法在一定程度可以实现“奇怪形状”输入下依然稳定运行的目标。

正交随机 1000 多边形　　随机 koch von1000 多边形　　700 简单多边形

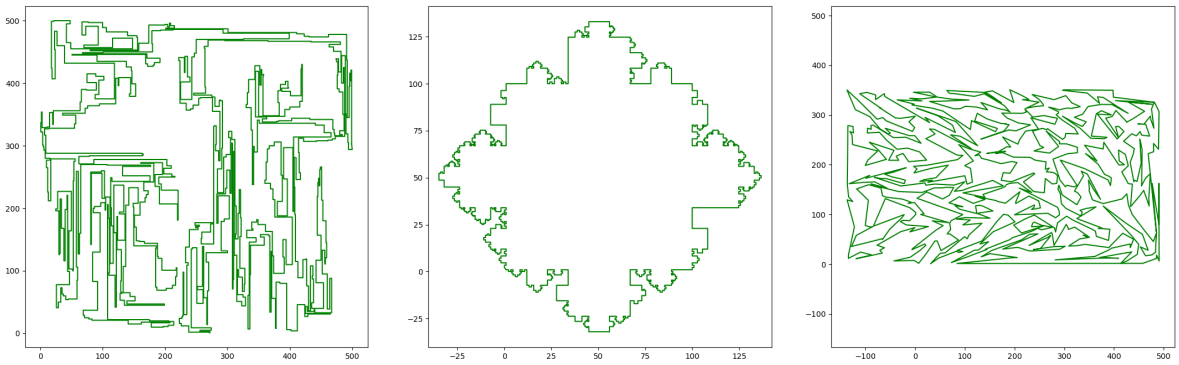

图 2: 代表性极端样例数据集中的三个样例

表 2: 性能评估数据集各组样例数

	随机简单多边形	随机正交多边形	随机科赫曲线
20	30	30	30
40	30	30	30
60	30	30	30
80	30	30	30
100	30	30	30
200	30	30	30
300	30	30	30
400	30	30	30
500	10	10	10
600	10	10	10
700	10	10	10
800	10	10	10
1000	10	10	10

5　吞吐能力与性能评估

5.1　实验目的

性能评估主要需评估当前算法的执行效率及吞吐能力。Art Gallery Problem 本身属于一个 NP Hard 问题，输入的多边形形状、多边形的复杂程度不同，求解的速度也会有很大的差异。在学术界提出了各种优化的方法来提升效率，争取在可接受的时间内对用户的输入给出一个较优的解。

本项目实现算法是基于整数规划的求解策略，因此我们将实现算法与同属整数规划策略的算法 [7] 进行对比，在相同整数规划求解器、相同运行环境的前提下，统计不同点数、不同形态的多边形求解所需要的时间，评估对比算法的性能。

5.2 实验设计

实验数据集以两个维度进行划分。按照多边形形态的不同，我们分别构造了随机简单多边形、随机正交多边形、完全科赫曲线和随机科赫曲线四种数据集，每种数据集按照点数的不同划分为若干组，每组有多个样例重复试验，以保证实验结果的准确性。取每组中单个样例平均求解耗时来评估多边形形态、点数对算法性能的影响，并与 [7] 形成横向对比，以确定本算法与对比算法在性能上的差异。详细的数据集信息参照表2，图 2 给出了几个数据集的样例。

实验环境是一台运行 Ubuntu 20.04 系统的主机，处理器为 Intel i7-10700，内存为 16G。

正交随机多边形 (200 顶点)　　完全科赫曲线 (500 顶点)　　简单多边形 (40 顶点)

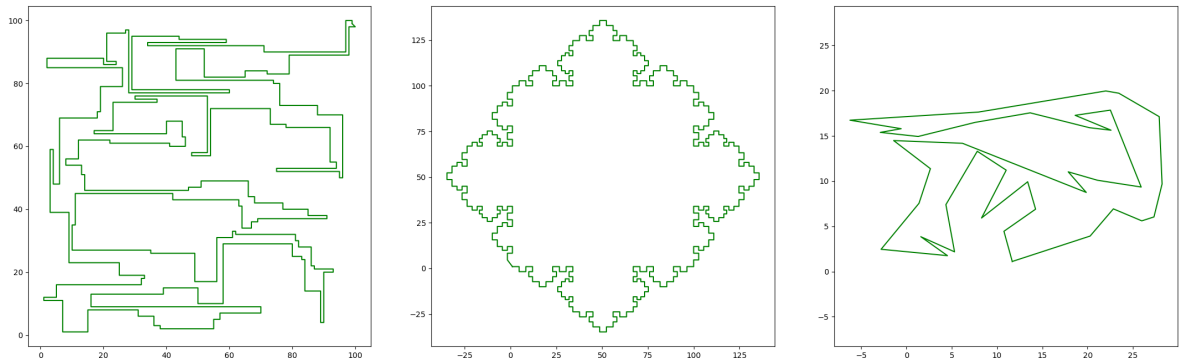

图 3: 性能评估数据集中的三个代表样例

本项目实验结果　　对比项目实验结果

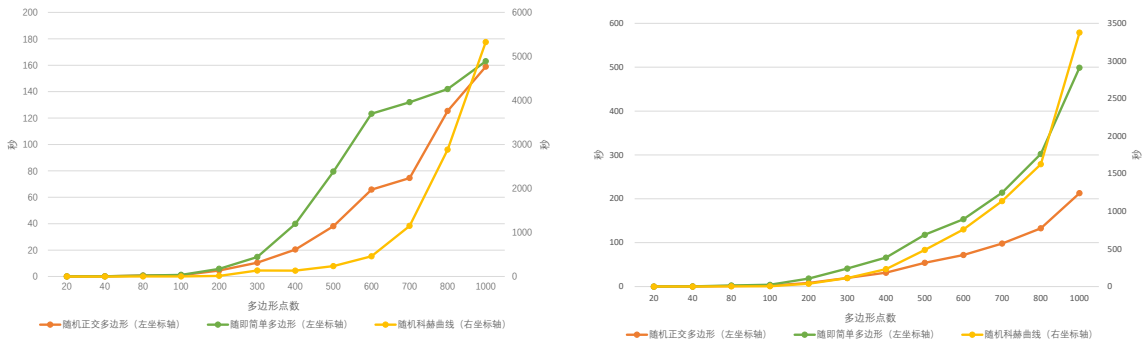

图 4: 计算所有本项目实验结果和对比项目实验结果折线图

5.3 结果分析

在不同顶点数、不同形态的多边形上，本算法和对比算法平均在每组样例上的求解时间如图4所示。

从图中可以看出，多边形形态、点数均对执行时间有较大的影响。其中，多边形形态相对越复杂，平均求解耗时越长。在点数相同的情况下，相对比较复杂的科赫曲线平均求解时间要比一般简单多边形的求解时间高出 300%。而点数越多，求解时间相应也就越长，并在各形状上随

点数增加呈近似指数形态的增长。例如，在求解简单多边形的测试实验中，点数从 100 增长到 200 时，平均每组样例的耗时从 1.25 秒增加到了 5.80 秒，增加了 4.5 倍。

相较对比算法而言，本算法在求解时间上表现不佳。根据我们的分析，主要原因可能有以下两个方面。（1）对比算法在一开始初始化时采用了 Shadow AVP 的方法来更合理地初始离散点，这样的方法可以显著减少后续迭代的次数，进而减少求解整数规划问题的次数。（2）对比算法更精准地控制了迭代的次数，根据输入多边形点数的不同，给迭代尝试次数设定了一个很灵活的上界。因此，不便求解或已求出可接受的次优解情况下，程序即会自动报告出结果，终止迭代。（3）对比工作在调用求解器时会采用更高效的优化算法，这部分与计算几何并无太大关联，我们未来得及实现，也带来了部分性能损失。

在今后，我们的算法也可以在这两个方向上继续精进，优化初始离散点和初始守卫点的选择策略，以提高整体程序的效率。考虑到此论文作者 COUTO 最早在 2007 年即发表了使用集合覆盖问题解决正交多边形端点艺术画廊守卫问题的第一篇论文 [5]，在 Art Gallery 问题上进行了十余年的研究，其近些年更是不断改进尝试将其推进至解决带洞简单多边形的艺术画廊守卫问题（点可以位于多边形内），其算法不断迭代优化，我们还应多多学习，将来会进一步提高其性能和准确性。

6 小组分工

殷梓敬：阅读论文，编写算法主流程，联合调试代码，进行性能测试，撰写报告

孙浩：阅读论文，编写可交互前端，联合调试代码，进行鲁棒性测试，撰写报告

张泉：阅读论文，研究可见性算法，联合调试代码，进行性能测试，撰写报告。

7 实验总结

在本次实验中，小组尝试解决端点艺术画廊问题。首先，我们完成了一个可用的求解算法，且完成了具有较丰富互动功能的前端界面，实时展示算法计算的结果。在这个过程中，我们学习到了很多计算几何知识，进一步实践了算法的过程，学习到了算法在面对复杂输入时，应该怎样保证算法的稳定性。特别地，我们的选题是一个 NP-Hard 的问题，只能追求速度和近似解的平衡。在这过程中我们处理了大量的工程优化问题，极大地锻炼了我们的实践能力。

参考文献

[1] CHVáTAL V. A combinatorial theorem in plane geometry[J/OL]. Journal of Combinatorial Theory, Series B, 1975, 18(1):39-41. https://www.sciencedirect.com/science/article/pii/0095895675900611. DOI: https://doi.org/10.1016/0095-8956(75)90061-1.

[2] FISK S. A short proof of chvátal's watchman theorem[J]. J. Comb. Theory, Ser. B, 1978, 24:374.

[3] LEE D, LIN A. Computational complexity of art gallery problems[J/OL]. IEEE Transactions on Information Theory, 1986, 32(2):276-282. DOI: 10.1109/TIT.1986.1057165.

[4] GHOSH S K. Approximation algorithms for art gallery problems in polygons[J/OL]. Discrete Applied Mathematics, 2010, 158(6):718-722. DOI: https://doi.org/10.1016/j.dam.2009.12.004.

[5] COUTO M C, DE SOUZA C C, DER EZENDE P J. An exact and efficient algorithm for the orthogonal art gallery problem [C/OL]//XX Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI 2007). 2007: 87-94. DOI: 10.1109/SIBGRAPI.2007.15.

[6] COUTO M C, REZENDE P J, SOUZA C D. An exact algorithm for minimizing vertex guards on art galleries[J]. Int. Trans. Oper. Res., 2011, 18:425-448.

[7] LEE D. Visibility of a simple polygon[J/OL]. Computer Vision, Graphics, and Image Processing, 1983, 22(2):207-221. https://www.sciencedirect.com/science/article/pii/0734189X83900658. DOI: https://doi.org/10.1016/0734-189X(83)90065-8.

[8] LEE D T. Proximity and reachability in the plane.[D]. USA: University of Illinois at Urbana-Champaign, 1978.