AN APPLICATION OF THE ART GALLERY PROBLEM TO DETERMINE THE NUMBER OF CAMERAS PLACED FOR ROADWAY MONITORING

AMPHON KLIARAM

A Thesis Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Applied Mathematics
Suranaree University of Technology
Academic Year 2022

การประยุกต์ใช้ปัญหาหอศิลป์เพื่อกำหนดจำนวนกล้องสำหรับตรวจสอบถนน

นายอำพล เกลียรัมย์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต

สาขาวิชาคณิตศาสตร์ประยุกต์

มหาวิทยาลัยเทคโนโลยีสุรนารี

ปีการศึกษา 2565

AN APPLICATION OF THE ART GALLERY PROBLEM TO DETERMINE THE NUMBER OF CAMERAS PLACED FOR ROADWAY MONITORING

Suranaree University of Technology has approved this thesis submitted in partial fulfillment of the requirements for a Master's Degree.

Thesis Examining Committee

J. Tanthanuch

(Asst. Prof. Dr. Jessada Tanthanuch)
Chairperson

[illegible]

(Dr. Akanat Wetayawanich)
Member (Thesis Advisor)

[illegible]

(Asst. Prof. Dr. Wipawee Tangjai)
Member

Yupaporn [illegible]

(Assoc. Prof. Dr. Yupaporn Ruksakulpiwat)
Vice Rector for Academic Affairs
and Quality Assurance

[illegible]

(Prof. Dr. Santi Maensiri)
Dean of Institute of Science

อำพล เกลียรัมย์ : การประยุกต์ใช้ปัญหาหอศิลป์เพื่อกำหนดจำนวนกล้องสำหรับตรวจสอบถนน (AN APPLICATION OF THE ART GALLERY PROBLEM TO DETERMINE THE NUMBER OF CAMERAS PLACED FOR ROADWAY MONITORING) อาจารย์ที่ปรึกษา : อาจารย์ ดร.เอกณัฐ เวทยะวานิช, 58 หน้า.

คำสำคัญ: ปัญหาหอศิลป์/ รูปหลายเหลี่ยมเชิงตั้งฉากหนึ่งหน่วย/ ขั้นตอนวิธีค้นหาทั้งหมด

วิทยานิพนธ์นี้มีจุดประสงค์เพื่อที่จะหาขั้นตอนวิธีสำหรับหาจำนวนกล้องที่เหมาะสมในการตรวจสอบถนน ที่มีลักษณะเป็นรูปหลายเหลี่ยมเชิงตั้งฉากหนึ่งหน่วย ซึ่งเป็นปัญหาที่เกี่ยวข้องกับปัญหาหอศิลป์ในทางคณิตศาสตร์ ภายใต้ข้อสมมติที่ว่า กล้องมองเห็นเป็นมุมกว้าง 90 องศา และมีระยะมองเห็นได้อย่างไม่จำกัด เมื่อสร้างขั้นตอนวิธีใหม่ขึ้นมา และเปรียบเทียบผลลัพธ์ที่ได้จากขั้นตอนวิธีค้นหาทั้งหมด พบว่า ผลลัพธ์จากการใช้ขั้นตอนวิธีใหม่นี้สามารถหาจำนวนกล้องที่เหมาะสมได้ โดยมีความซับซ้อนของการประมวลผลเป็นแบบเชิงเส้น เพียงแต่ไม่สามารถสรุปได้ว่าผลลัพธ์ที่ได้มีจำนวนกล้องที่น้อยที่สุด สำหรับการใช้ขั้นตอนวิธีค้นหาทั้งหมด จะให้ผลลัพธ์ที่ได้มีจำนวนกล้องน้อยที่สุดแน่นอน แต่เมื่อรูปหลายเหลี่ยมเชิงตั้งฉากหนึ่งหน่วยที่พิจารณามีขนาดใหญ่จะต้องใช้เวลาประมวลผลนานมาก เนื่องจากมีความซับซ้อนของการประมวลผลเป็นแบบฟังก์ชันเลขชี้กำลัง อย่างไรก็ตาม สำหรับรูปหลายเหลี่ยมเชิงตั้งฉากหนึ่งหน่วยขนาดเล็ก มีหลายตัวอย่างที่ขั้นตอนวิธีค้นหาทั้งสองแบบให้ผลลัพธ์เป็นจำนวนกล้องเท่ากัน

สาขาวิชาคณิตศาสตร์ ลายมือชื่อนักศึกษา อำพล

ปีการศึกษา 2565 ลายมือชื่ออาจารย์ที่ปรึกษา [illegible]

AMPHON KLIARAM : AN APPLICATION OF THE ART GALLERY PROBLEM TO DETERMINE THE NUMBER OF CAMERAS PLACED FOR ROADWAY MONITORING. THESIS ADVISOR : AKANAT WETAYAWANICH, Ph.D. 58 PP.

Keyword: ART GALLERY PROBLEM/ UNIT ORTHOGONAL POLYGON/ BRUTE FORCE ALGORITHM

This study aims to find an algorithm for determining the suitable number of cameras required to monitor all points on a road system that takes the shape of a unit orthogonal polygon. This problem is related with the arts gallery problem in mathematics, and it is studied under the assumption that the cameras have a 90-degree field of view and an infinite range. After creating the new algorithm and comparing it with the results of the brute force algorithm, it is found that the new algorithm does indeed provide an appropriate number of cameras, with only linear time complexity. However, its results do not guarantee minimality of the number of cameras. As for the brute force algorithm, it can determine the minimum number of cameras with certainly, but significantly longer processing time for large unit orthogonal polygons, as it requires exponential time complexity. Nevertheless, for small unit orthogonal polygons, there are many examples where both algorithms yield the same number of cameras.

School of Mathematics Student's Signature อำพล

Academic Year 2022 Advisor's Signature [illegible]

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Dr.Akanat Wetayawanich for his unwavering support and guidance throughout my research. I am grateful to Asst. Prof. Dr. Jessada Tanthanuch for providing the initial idea for this work, formatting the code in LaTeX, and assisting in prototyping the C++ code, which played a crucial role in developing the algorithms presented in this thesis. I am immensely thankful to Asst. Prof. Dr. Poj Lertchoosakul for his invaluable guidance in improving this study. I am deeply grateful to my family and friends for their emotional support and encouragement throughout my studies. Finally, I would like to mention the financial support provided by the DPST scholarship, which made this research possible.

Amphon Kliaram

CONTENTS

CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

CHAPTER I

INTRODUCTION

Camera placement on roads is beneficial for modern traffic management. With advancements in technology, cameras have become an important part of monitoring and ensuring road safety, managing traffic flow, and assisting in law enforcement activities. Moreover, cameras can provide real-time information, so authorities can respond and implement measures to improve traffic flow and ensure smooth transportation. To determine camera placement on roads, there are several factors need to be considered. These factors include traffic volume, accident-prone areas, crime hotspots, road geometry, visibility, and legal requirements. Strategic placement ensures that the camera system provides optimal coverage and cover the most important areas. It helps minimizing the number of cameras required for road surveillance. Then, we can reduce infrastructure costs, maintenance efforts, and resource allocation while maintaining the visible area of monitoring and security. This raises an important question that what is the minimum number of cameras required to monitor every point on the road? and this problem is related to one of the problem in mathematics which known as "art gallery problem".

The concept of art gallery problem was proposed in 1973 by Victor Klee (Stewart, 2015). He presented a problem to Václav Chvátal "What is the minimum number of stationary guards required to protect an art gallery?" Geometrically, the problem can be formulated as follows, given an n vertices simple polygon "what is the minimum number of guards needed to see every point within the interior of the polygon?" Chvátal (1975) was able to prove that for simple polygons, $\left\lfloor \frac{n}{3} \right\rfloor$ guards are both necessary and sufficient to protect the gallery when there are n vertices in the polygon. However, his proof was complicated and relied on induction. Fisk (1978) developed a much simpler proof using triangulation, a method of dividing a polygon into triangles and vertex coloring. This problem has many real-life applications that have not only motivated the mathematics community to find better solutions due to real-life constraints, but also inspired various versions of the problem that model real-world scenarios.

Therefore, in this research the approach involves examining the unit orthogonal polygon which is an orthogonal polygon with a width of one unit. We focus on determining a suitable cameras placement for monitoring every point on roads and we will compare the number of cameras utilized by main algorithm with the brute force algorithm to evaluate its effectiveness.

1.1 Research objective

To find an algorithm that can provide a suitable number and position of cameras needed to guard every point on the roads of the unit orthogonal polygon, assuming that the cameras have a 90-degree field of view and an effective range that extends to infinity.

1.2 Scope and limitations

1. We study the unit orthogonal polygon, which is an orthogonal polygon with a width of one unit.
2. We assume that cameras have a 90-degree field of view and an effective range that extends to infinity.

1.3 Expected results

We can find an algorithm that can provide a suitable number and position of cameras needed to guard every point on the roads of the unit orthogonal polygon, assuming that cameras have a 90-degree field of view and an effective range that extends to infinity.

CHAPTER II

LITERATURE REVIEW

This chapter presents the background of the art gallery problem and the concept of vertex cover.

2.1 Definitions

Definition 2.1 (Vector space $\mathbb{R}^2$) The vector space $\mathbb{R}^2$ is defined as the set of all ordered pairs of real numbers, represented as two-dimensional vectors (x, y), where x and y are real numbers. The operations of addition and scalar multiplication on vectors in $\mathbb{R}^2$ are defined as follows:

Addition of vectors:

$$(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2).$$

Scalar multiplication of vectors:

$$c \cdot (x, y) = (cx, cy)$$

where c is a real number.

Definition 2.2 (Distance function) Given two points $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ represented as vectors (x_1, y_1) and (x_2, y_2) in $\mathbb{R}^2$, respectively. The distance function d can be defined by using the Euclidean distance formula:

$$d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

This study we investigate the problem by using analytic geometry on $\mathbb{R}^2$, encompassing considerations of length, angles and area. For example, the straight line between point P_1 and P_2 is

$$\overline{P_1P_2} = \{\ tP_1 + (1-t)\,P_2 \mid\ 0 \le t \le 1\ \}.$$

Definition 2.3 (Simple polygon) Given $\{v_1, v_2, v_3, \ldots, v_n\} \subset \mathbb{R}^2$ and $E = \{e_1, e_2, e_3, \ldots, e_n\}$ where $e_i = \overline{v_i v_{i+1}}$ for all $i = 1, 2, 3, \ldots, n$ and $v_{n+1} = v_1$. A simple polygon $P = (V, E)$ is a closed curve consisting of the set of vertices V and the set of edges E such that there are no intersecting consecutive edges and there is no hole inside (figure 2.1).

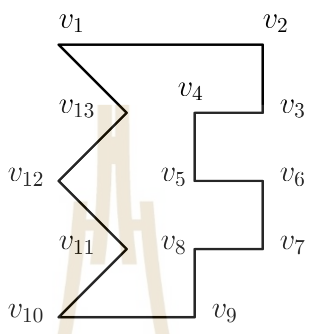


Figure 2.1 13-vertex simple polygon.

Definition 2.4 (Diagonal) A diagonal is a straight line that connects two non-adjacent vertices of a simple polygon. It is a line segment that joins two vertices of a simple polygon that are not connected by an edge.

Definition 2.5 (Polygon with holes) Given a simple polygon P and a set of m disjoint simple polygons $P_1, P_2, \ldots, P_m$ contained in the interior of P. We call $P| \{P_1, P_2, \ldots, P_m\}$ a polygon with m holes (figure 2.2).

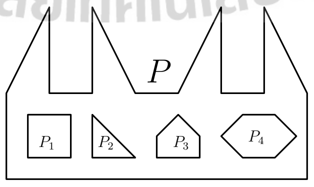


Figure 2.2 Polygon with holes.

Definition 2.6 (Orthogonal polygon) An orthogonal polygon is a polygon whose all sides meet at right angles and the interior angle is the angle formed by two adjacent sides inside the polygon. Thus the interior angle at each vertex is either 90° or 270° (figure 2.3).

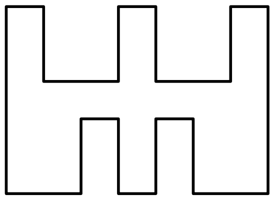
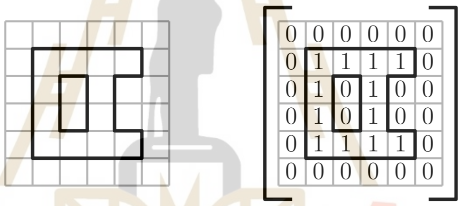


Figure 2.3 Orthogonal polygon.

Definition 2.7 (Unit orthogonal polygon) An orthogonal polygon P is a unit orthogonal polygon such that every vertex of P has integer coordinates, and no integer coordinate point lies in the interior of P.

According to this definition, any unit orthogonal polygon can be represented by a matrix. For example, in the figure 2.4, every exterior unit square is represented by the number 0, and every interior unit square is represented by the number 1.



$$\begin{bmatrix} 0&0&0&0&0&0\\ 0&1&1&1&1&0\\ 0&1&0&1&0&0\\ 0&1&0&1&0&0\\ 0&1&1&1&1&0\\ 0&0&0&0&0&0 \end{bmatrix}$$

Figure 2.4 Unit orthogonal polygon.

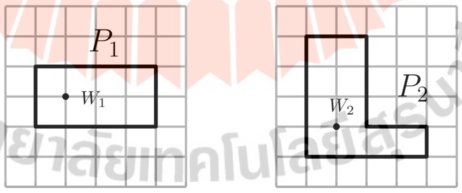


Figure 2.5 Orthogonal polygons P_1 and P_2 are not unit orthogonal polygons because there are integer coordinate points W_1 and W_2 lying in the interior of P_1 and P_2 respectively.

Definition 2.8 (Visibility in a polygon) Two points p and q of polygon P are said to be visible from each other if the straight line from p to q lies completely in P (figure 2.6).

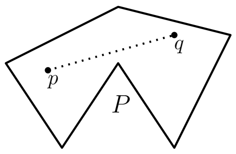


Figure 2.6 Visibility in a polygon.

Definition 2.9 (Guards in polygon)

- A guard is a point inside or on the boundary of the polygon that can see other points in the polygon.
- A vertex guard refers to a guard that is placed anywhere on a vertex (figure 2.7).
- An edge guard refers to a guard that is placed anywhere along an edge (figure 2.8).
- A point guard refers to a guard that is placed anywhere in the polygon (figure 2.9).
- A mobile guard refers to a guard that is allowed to patrol along a line segment lying in the polygon.

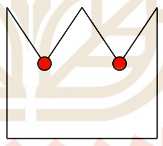


Figure 2.7 Vertex guard positions are represented by two circle position.

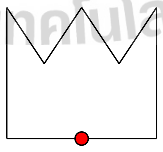


Figure 2.8 Edge guard position is represented by a circle position.

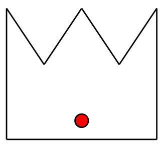


Figure 2.9 Point guard position is represented by a circle positions.

2.2 Art gallery problem

Art gallery problem begins with the question that “how many guards are needed to ensure the security of an art gallery?” The objective is to position the guards in such a way that every point inside the polygon is visible from at least one guard. The original art gallery problem aimed to determine the minimum number of guards required to have visibility over every point within an n-vertex simple polygon. In other words, the question was how many guards are needed to ensure the security of the entire museum. Victor Klee (Stewart, 2015) first presented this problem to Vaclav Chvátal in 1973. Chvátal (1975) was able to prove that $\left\lfloor \frac{n}{3} \right\rfloor$ guards are both necessary and sufficient to cover the entire gallery when the polygon has n vertices. However, Chvátal’s proof was quite complicated, relying on the method of induction. Fisk (1978) provided a much simpler proof using triangulation, a technique that involves decomposing a polygon into triangles, and the coloring of vertices. Fisk’s approach offered a more straightforward demonstration of the minimum number of guards required for complete coverage.

For the number of guards that are mentioned above, the meaning of $\left\lfloor \frac{n}{3} \right\rfloor$ guards are both necessary and sufficient to cover the entire gallery when the simple polygon has n vertices is that $\left\lfloor \frac{n}{3} \right\rfloor$ is the upper bound of the number of elements of a set of guards that can guard simple polygon, so for simple polygons with the same number of vertices but different shape may use a number of guards less than $\left\lfloor \frac{n}{3} \right\rfloor$. However, $\left\lfloor \frac{n}{3} \right\rfloor$ guards can guarantee that they can guard simple polygon as shown in figure 2.10.

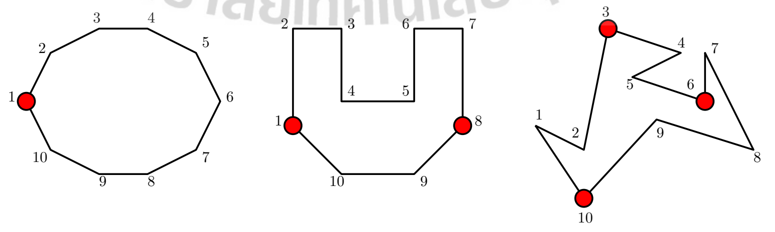


Figure 2.10 Decagon, which is a different shape, may not have the same minimum number of guards. A circle position is used to indicate the position of the guard.

2.2.1 Polygon triangulation

Polygon triangulation is a technique used to divide a simple polygon into a collection of triangles. There are various methods for solving the triangulation problem, and one of the method that is called “ear trimming.” In this approach, an “ear” is defined as a vertex and its two adjacent neighbors, forming a triangle that can be drawn without intersecting any other edges inside the polygon where the added side of the triangle is contained in the polygon. The algorithm identifies these ears and removes them from the polygon. For a vertex that is not an ear, the algorithm proceeds to the next vertex. The process continues until only one triangle remains, indicating that the polygon has been fully triangulated.

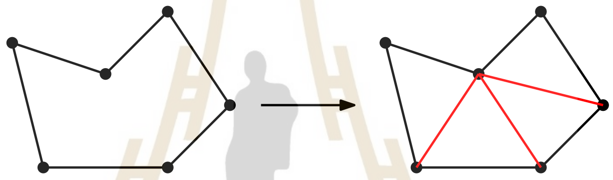


Figure 2.11 Polygon triangulation.

Triangulating a polygon with holes is achievable by extending lines from the holes to the main polygon. This process considers the holes as part of the overall polygon, allowing us to apply the same triangulation method as before. It is important to note that triangulation for a polygon is not a unique. The order in which the vertices of the polygon are processed can lead to different sets of diagonals generated by the triangulation algorithm. As a result, there can be multiple valid triangulations for the same polygon, depending on the chosen vertex handling order.

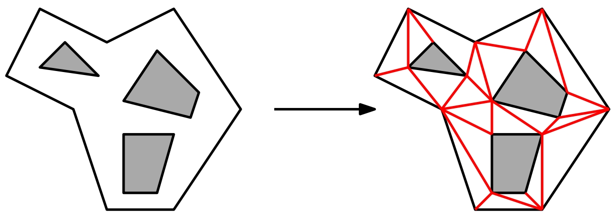


Figure 2.12 Polygon triangulation with holes.

2.2.2 Vertex coloring

Vertex coloring is a fundamental concept in graph theory. It involves giving different colors to each of a graph's vertices so that no two adjacent vertices have the same color. The objective of vertex coloring is to determine the minimum number of colors set of colors needed to color a graph's vertices so that no two adjacent vertices share the same color. This minimum number is called the chromatic number of the graph.

Here are some important theorems and properties related to vertex coloring.

Theorem 2.1 (The Four-Color Theorem (Appel and Haken, 1975)) *For any planar graph, a graph that can be drawn on a plane without any edge crossings, can be colored with at most four colors in such a way that no two adjacent vertices have the same color.*

Theorem 2.2 (West, 1996) *For any triangulation graph G, its chromatic number $\chi(G)$ satisfies the inequality $\chi(G) \leq \Delta(G)+1$, where $\Delta(G)$ is the maximum degree of a vertex in G.*

In the art gallery problem, the idea of Fisk's proof is using the triangulation method for a simple polygon. After that, it can be colored by using three different colors: red, green, and blue, as shown in figure 2.13. The coloring algorithm is relatively simple. One random triangle is initially colored. Then, adjacent triangles can be colored since they share two vertices with the first triangle and have only one missing color. This process continues until all vertices are colored. After all vertices are colored, the color that has been used the least is then used to place guards at the vertices with the least used color. It is obtained that the number of vertices that are placed guard is at most $\left\lfloor \frac{n}{3} \right\rfloor$.

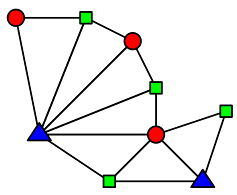


Figure 2.13 The color with the least number of appearances is represented by two "small triangles" at the vertices. These are the positions to place the guards.

2.2.3 Polygon with holes

First, we present Chvátal's Art Gallery theorem (Chvátal, 1975). This theorem is known as an upper bound and there is no theorem that can provide the number of guards less than this theorem. Therefore, many researchers are interested in the study of specific polygon or changing the problem, such as considering the hole inside the polygon.

Theorem 2.3 *Any polygon with n vertices can always be guarded with $\left\lfloor \frac{n}{3} \right\rfloor$ guards, and these guards are always sufficient and necessary to cover a polygon.*

The concept of the art gallery problem was extended to include cases where there are holes in the polygon. O'Rourke (1987) made significant contributions by presenting the first results on guarding polygon with holes.

Theorem 2.4 *Any polygon with n vertices and h holes can always be guarded with $\left\lfloor \frac{n+2h}{3} \right\rfloor$ vertex guards.*

Shermer (1982) proposed a conjecture suggesting that there exists a lower number of guard.

Conjecture 2.1 Any polygon with n vertices and h holes can always be guarded with $\left\lfloor \frac{n+h}{3} \right\rfloor$ vertex guards.

In 1982, Shermer provided a proof for his conjecture in the case of $h = 1$. The conjecture remains open for $h > 1$, and then it is extended to the case of point guards. In the case of point guards, many researchers provide the result, such as Bjorling-Sachs and Souvaine (1991), as well as Hoffmann, Kaufman, and Kriegel (1991). They have independently provided proof that any polygon with n vertices and h holes can always be guarded with $\left\lfloor \frac{n+h}{3} \right\rfloor$ point guards.

Theorem 2.5 *Any polygon with n vertices and h holes can always be guarded with $\left\lfloor \frac{n+h}{3} \right\rfloor$ point guards.*

The concept of art gallery problem has been studied by many researchers. So various cases of polygons have been investigated. One particularly famous case is the orthogonal polygon, which will be discussed in the next section.

2.2.4 Orthogonal polygon

In the art gallery problem, orthogonal polygons have received much attention. This is perhaps because most real buildings are orthogonal, and thus the result will be useful to apply to real problems. However, the study of orthogonal allows us to obtain very interesting results in mathematics. The first major result here was presented by Kahn, Klawe and Kleitman (1983).

Theorem 2.6 *Any orthogonal polygon with n vertices can always be illuminated with $\left\lfloor \frac{n}{4} \right\rfloor$ vertex guards.*

This proof was based on a similar technique to used by Fisk (1978). The main idea of their proof is to partition an orthogonal polygon into convex quadrilaterals. By adding internal diagonals to each of these quadrilaterals and the graph, thus obtained four-vertex colored.

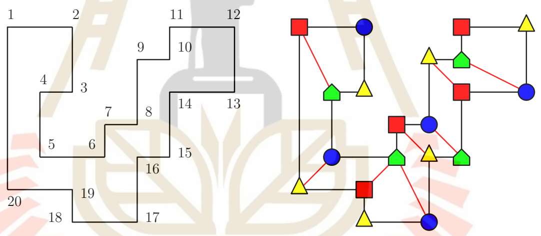


Figure 2.14 Quadrilaterals of orthogonal polygon and vertex coloring.

2.2.5 Orthogonal polygons with holes

In 1987, O'Rourke proved that any orthogonal polygon with n vertices and h holes can always be guarded with $\left\lfloor \frac{n+2h}{4} \right\rfloor$ vertex guards and provide the conjecture that $\left\lfloor \frac{n}{4} \right\rfloor$ point guards are always sufficiennt to guard any orthogonal polygon with holes. Aggarwal (1984) was able to verify this conjecture for $h = 1, 2$. It then remained open until 1990, Hoffmann (1990) provided a proof.

Theorem 2.7 *Any orthogonal polygon with n vertices and h holes can always be guarded with $\left\lfloor \frac{n}{4} \right\rfloor$ point guards.*

According to theorem 2.7, any orthogonal polygon can always be guarded by $\left\lfloor \frac{n}{4} \right\rfloor$ point guards regardless of whether an orthogonal polygon with or without holes. However for vertex guards, the best-known upper bound is $\left\lfloor \frac{n+2h}{4} \right\rfloor$, as proposed by O'Rourke. However, it has been recognized for sometime that $\left\lfloor \frac{n}{4} \right\rfloor$ vertex guards are not always enough to guard orthogonal polygons with holes. For instance, the polygon depicted in figure 2.15, featuring 44 vertices and 4 holes, requires 12 vertex guards.

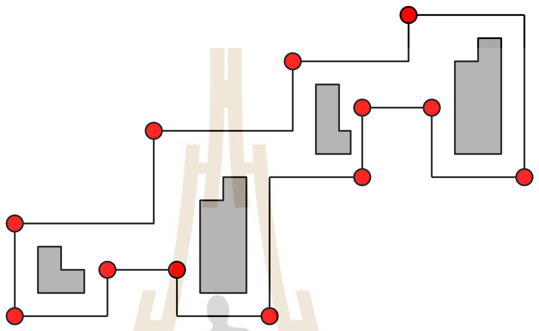


Figure 2.15 An orthogonal polygon with 44 vertices and 4 holes that requires 12 vertex guards. A circle position is used to indicate the location of guard.

In 1982, Shermer made the following conjecture for orthogonal polygons with holes.

Conjecture 2.2 Let P be an orthogonal polygon with n vertices and h holes, then $\left\lfloor \frac{n+h}{4} \right\rfloor$ vertex guards are sufficient to cover orthogonal polygon P.

For this conjecture Kahn, Klawe and Kleitman (1983) proved Shermer's conjecture for $h = 0$. Aggarwal (1984) proved Shermer's conjecture for $h \leq 2$.

The orthogonal art gallery problem interests many researchers in finding more conditions and the minimum number of guards.

Michael and Pinciu (2016) study the orthogonal art gallery theorem with constrained guards, where the guards are constrained to only move along certain paths.

Theorem 2.8 *Let V^* and E^* be specified sets of vertices and edges of P. Then P has a guard set of cardinality at most $\left\lfloor \frac{n+3|V^*|+2|E^*|}{4} \right\rfloor$ that includes each vertex in V^* and at least one point of each edge in E^*.*

2.2.6 Vertex cover

In graph theory, a *vertex cover* of a graph is a set of vertices that contains at least one endpoint of every edge in the graph. The *minimum vertex cover problem* is the optimization problem of finding a vertex cover of minimum cardinality. It is NP-hard, so it cannot be solved by a polynomial-time algorithm if $P \neq NP$ (Garey and Johnson, 1979).

A vertex cover can be used to solve a variety of problems in computer science and operation research. For example, it can be used to find the minimum number of guards needed to patrol a security area or the minimum number of servers needed to provide a certain level of service.

There is a number of different algorithms for finding vertex covers. Some of the most common algorithms include (Cormen, Leiserson, Rivest, and Stein, 2009).

- The greedy algorithm: this algorithm starts with an empty set of vertices and then adds a vertex to the set one by one, by choosing the vertex that covers the most edges.
- The branch-and-bound algorithm: this algorithm starts by considering all possible vertex covers and then recursively eliminates vertex covers that cannot be optimal.
- The local search algorithm: this algorithm starts with a random vertex cover and then iteratively improves the cover by swapping vertices in and out of the cover.

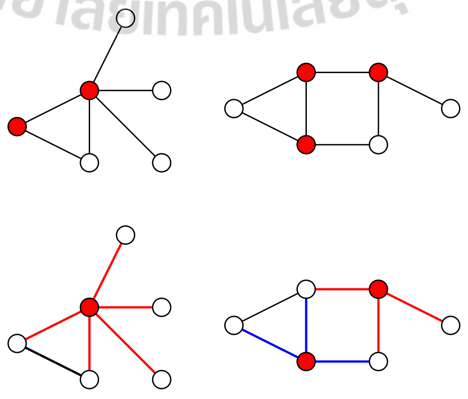


Figure 2.16 Vertex cover.

CHAPTER III

RESEARCH METHODOLOGY

In this chapter, we present the theorem and the algorithms needed in this work. First, we provide the idea of this thesis.

- The traditional art gallery problem and this thesis differ mostly in their methodologies. The classical art gallery problem utilizes graph theory and focuses on visibility without considering length, but this thesis investigates the problem by using analytic geometry on $\mathbb{R}^2$, encompassing considerations of length, angles, and area.
- In the case of the orthogonal polygon, as shown in figure 3.1. If we place a guard at a circled position, we notice that the guard cannot cover the entire figure. For simplification, we study the case of a unit orthogonal polygon.

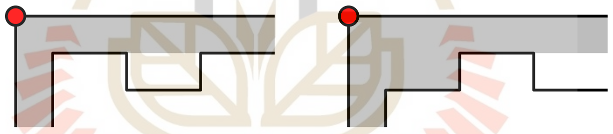


Figure 3.1 Camera placement in the case of orthogonal polygon.

- The camera placement problem in this thesis is studied in the case of two dimensions for a unit orthogonal polygon.
- We add two important limitations, including that cameras have a 90-degree field of view, which is different from the classical problem that cameras have a 360-degree field of view and an effective range that extends to infinity.

3.1 Number of vertices in a unit orthogonal polygon

For a unit orthogonal polygon without hole with n vertices, the number of interior angles of 90° and 270° is a and b, respectively, as shown in figure 3.2. We consider the total number of angles and the sum of all interior angles in the unit orthogonal polygon, we obtain a system of equations:

$$a + b = n$$
$$90a + 270b = 180(n - 2).$$

The solution of system of equation is

$$a = \frac{n+4}{2}$$
$$b = \frac{n-4}{2}.$$

Therefore, the number of vertices must be even, and the vertices of a unit orthogonal polygon can be determined by counting either the total number of 90° interior angles or the total number of 270° interior angles of the polygon.

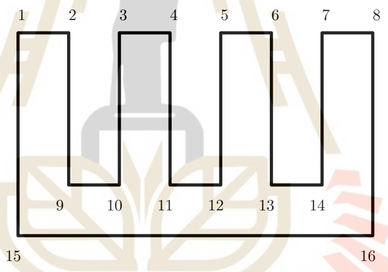


Figure 3.2 16-vertex unit orthogonal polygon without hole with $a = 10$ and $b = 6$.

In the context of a unit orthogonal polygon with holes, which has n vertices, it is necessary to consider the vertices of holes within the shape. So, for finding the number of vertices in term of the number of 90° interior angles and the number of 270° interior angles is complicated. Therefore, the unit orthogonal polygon is divided into five little forms. A straight way, an edge position, a corner position, a triple junction position, and a crossroad position are the five main parts of the shape (see figure 3.3) as follow:

- there is no vertex for any straight way;
- there are two vertices for any edge position;
- there are two vertices for any corner position;

- there are two vertices for any triple junction position;
- there are two vertices for any crossroad position.

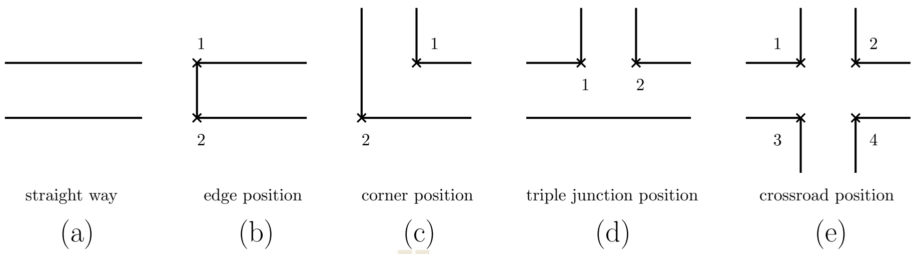


Figure 3.3 “x” refers to vertex position. (a) A straight way. (b) An edge position. (c) A corner position. (d) A triple junction position. (e) A crossroad position.

Given the number of edge positions, corner positions, triple junction positions, and crossroad positions in a unit orthogonal polygon with holes as $E, C, T,$ and F respectively, we can determine the number of vertices as follows:

$$n = 2E + 2C + 2T + 4F.$$

Moreover, this equation remains valid for any unit orthogonal polygon without hole.

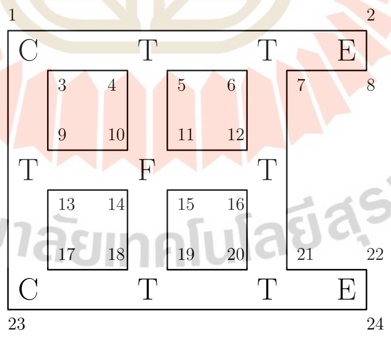


Figure 3.4 24 vertices unit orthogonal polygon with holes where $E = 2, C = 2, T = 6,$ $F = 1$ and $24 = 2(2) + 2(2) + 2(6) + 4(1)$.

3.2 Position of cameras

When placing the cameras in positions such as straight way, edge position, corner position, triple junction position, and crossroad position, we can agree to the following rule: place the camera at a border or a vertex, aligning its view direction parallel to or perpendicular to the border only (see figure 3.5). If the camera is not placed at the border or vertex, we can adjust its position according to the predetermined rule without changing the number of cameras. Consequently, there are a total of nine proper camera placement positions (see figure 3.6).

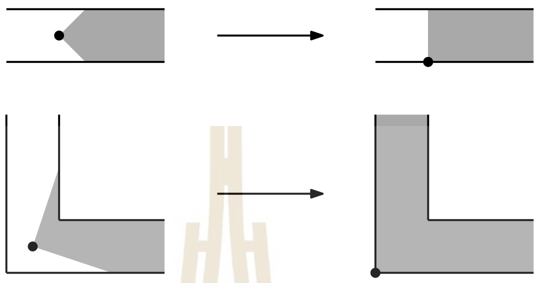


Figure 3.5 Adjust the camera position so that it has a parallel or perpendicular line of sight with the border.

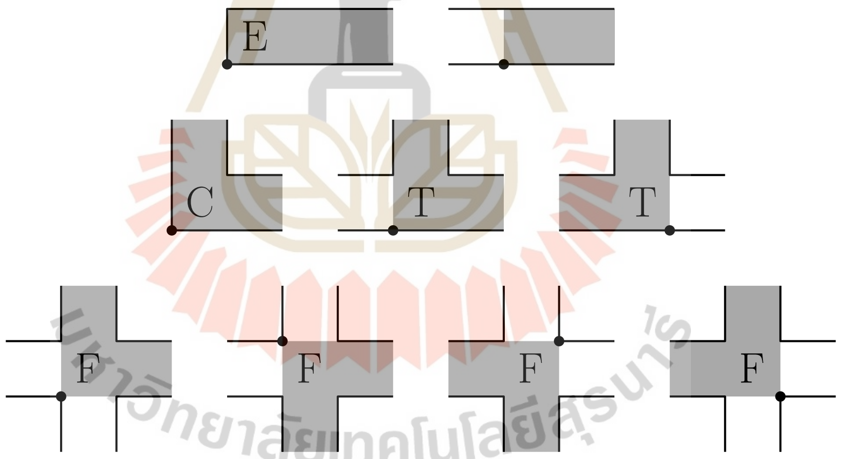


Figure 3.6 Nine proper camera placement positions for this thesis.

For any straight way position, we can move the camera to the nearest position in the opposite direction of the visible area of the camera, such as the edge position, corner position, triple junction position, or crossroad position, and the new camera position can still cover the visible area of the camera at the straight way position (see figure 3.7). After moving cameras from a straight way position, they remain in eight proper camera placement positions.

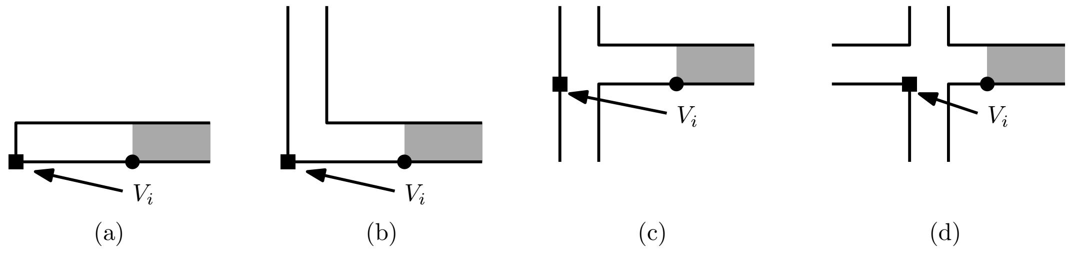


Figure 3.7 Moving the camera V_i from the straight way position to the nearest position. (a) moving to the edge position. (b) moving to the corner position. (c) moving to the triple junction position. (d) moving to the crossroad position.

For any crossroad, it can be covered by at least two cameras (see figure 3.8). If there exists a camera at the crossroad position, we can remove or move this camera to the nearest position in the opposite direction of the visible area of the camera, such as the edge position, the corner position, or the triple junction position, and the new camera position can still cover the visible area of the camera at the crossroad positions.

- In figure 3.8(a), we move V_1 in the down direction and move V_2 in the right direction.
- In figure 3.8(b), we move V_2 in the right direction, move V_3 in the up direction, and remove V_1.
- In figure 3.8(c), we remove V_3.

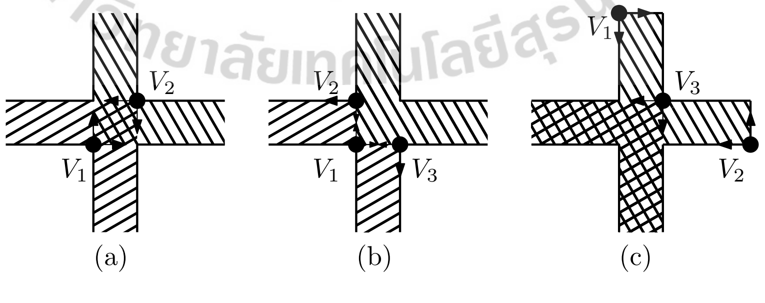


Figure 3.8 (a) Two cameras V_1 and V_2 can cover the crossroad. (b) Three cameras V_1, V_2 and V_3 can cover the crossroad. (c) Three cameras V_1, V_2 and V_3 can cover the crossroad.

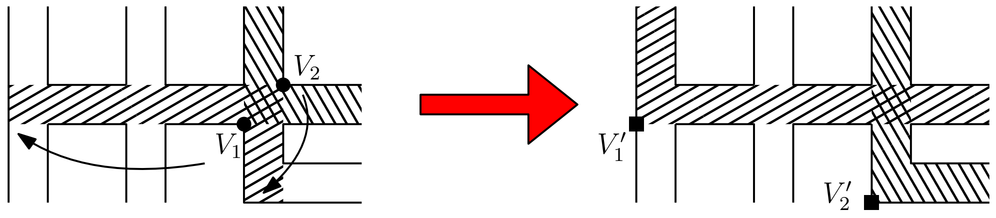


Figure 3.9 Moving the cameras V_1 and V_2 from the crossroad position to the triple junction position and the corner position respectively.

From the above conclusion, it follows that for a unit orthogonal polygon, whether it has holes or not, the camera positions are limited to only four positions (see figure 3.10).

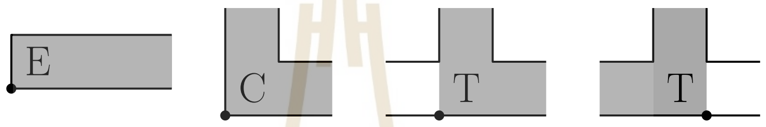


Figure 3.10 The camera positions are limited to only four positions.

Let P be a unit orthogonal polygon with or without holes and $\widetilde{V} = \{V_1, V_2, V_3, \ldots, V_k\}$ be the set of camera positions that can guard the area inside the unit orthogonal polygon P.

Theorem 3.1 *Let $\widetilde{V}$ be an arbitrary set of camera positions that can guard the area inside the unit orthogonal polygon P. If there exists a corner position W_1 and an edge position W_2 satisfying the following properties:*

1. *W_2 lies within the visible area of W_1, and*
2. *there are no triple junctions or crossroads within the intersecting visible area of W_1 and W_2.*

Then $\left(\widetilde{V} - \{W_2\}\right) \cup \{W_1\}$ can still effectively guard the area inside the unit orthogonal polygon P.

Proof Since positions W_1 and W_2 can be either members or non-members of $\widetilde{V}$. We have the following two cases.

- $W_1 \in \widetilde{V}$. Since W_2 lies within the visible area of W_1, we can remove the edge position W_2 from the set $\widetilde{V}$, if possible, while keeping the visible area the same.

- $W_1 \notin \widetilde{V}$. This implies that $W_2 \in \widetilde{V}$, so we can use the camera at corner position W_1 instead of W_2 while keeping the visible area the same.

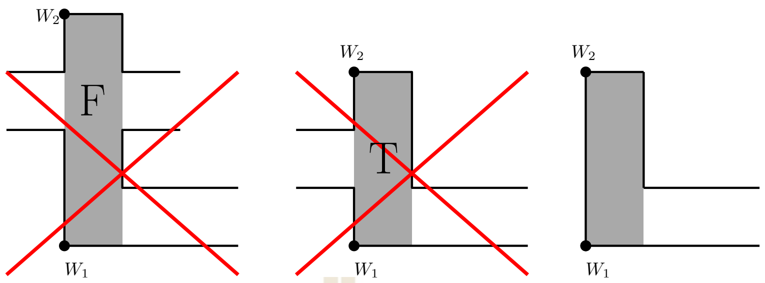


Figure 3.11 Visible area of position W_1 and W_2.

From the above conclusion, we search for the specific corner positions where we must place cameras. This is beneficial in reducing the number of executions of program when using a brute force algorithm.

We use C++ code programming to find the number of cameras for the unit orthogonal polygon. The device specifications and Windows specifications are shown in the table 3.1.

Table 3.1 Device specifications and Windows specifications.

Device Specification	
Processor	Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
Installed RAM	16.0 GB
System type	64-bit operating system, x64-based processor
Windows specifications	
Edition	Windows 10 Home Single Language
Version	22H2
OS build	19045.3208
Experience	Windows Feature Experience Pack 1000.19041.1000.0

3.3 Main algorithm

The main algorithm starts from one edge position and then recursively moves and places cameras. The concept of recursively moving is walking in the unit orthogonal polygon, when reaching an intersection, it will choose one of the available paths. Upon reaching a dead end, it will backtrack to the last intersection and then choose a new path at the intersection. This algorithm is called the depth-first search (DFS) algorithm. During the walk, it will continuously check the conditions for placing cameras. After completing this process, it will recheck the entire unit orthogonal polygon. If there are still positions not covered by cameras, additional cameras will be placed until all positions in the unit orthogonal polygon are covered. Then, we select a new starting edge position to process the algorithm and consider all edge starting positions. Different starting positions may obtain different results, so we choose the result with the minimum number of cameras for every edge starting position.

The concept of using the main algorithm to solve the camera placement problem can be outlined with the following steps:

1. Display the unit orthogonal polygon and store data of straight way, edge position, corner position, triple junction, and crossroads.

2. Choose the starting position from an edge position. If there is no edge position in the unit orthogonal polygon, then choose a corner position to be the starting position.

3. Move from the starting position to the next corner position. If the corner position and two adjacent positions around this corner position are not covered (figure 3.12):

    - if so, place a camera at this corner position.
    - If not, move to the next corner position.

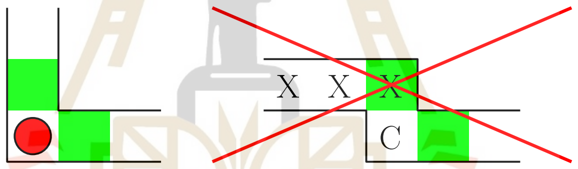


Figure 3.12 Check if the corner position is not covered and two adjacent positions around this corner position are not covered, then place a camera at this corner position.

4. Move to the next corner position and repeat step 3 until one has already moved to every corner position in the unit orthogonal polygon.

5. After placing the cameras at a possible position in step 4, if there is a position that is yet covered, then it is a triple junction position or edge position. For remaining triple junction positions, if one adjacent positions around these triple junction positions is covered (figure 3.13):

    - if so, place a camera at this triple junction position,
    - If not, move to the next triple junction position.

In this step, we will check all triple junction positions that satisfy above condition.

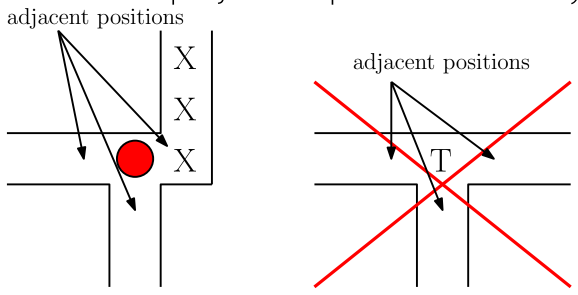


Figure 3.13 Consider triple junction positions that are still not covered and check if one of the three adjacent positions around these triple junction positions is covered, then place the camera at those triple junction positions.

6. Check the remaining positions to see which positions are still not covered (figure 3.14):

    - if so, place a camera at this position;
    - if not, move to the next position.

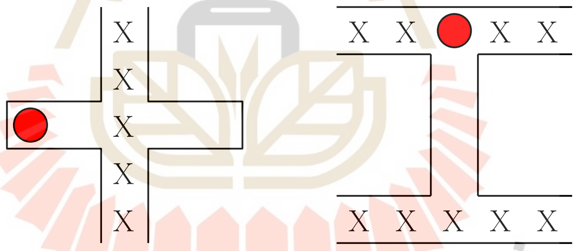


Figure 3.14 Check which positions are still not covered, then place cameras at those positions.

7. Change the starting position and repeat steps 2–6 to find the minimum number of cameras for the main algorithm.

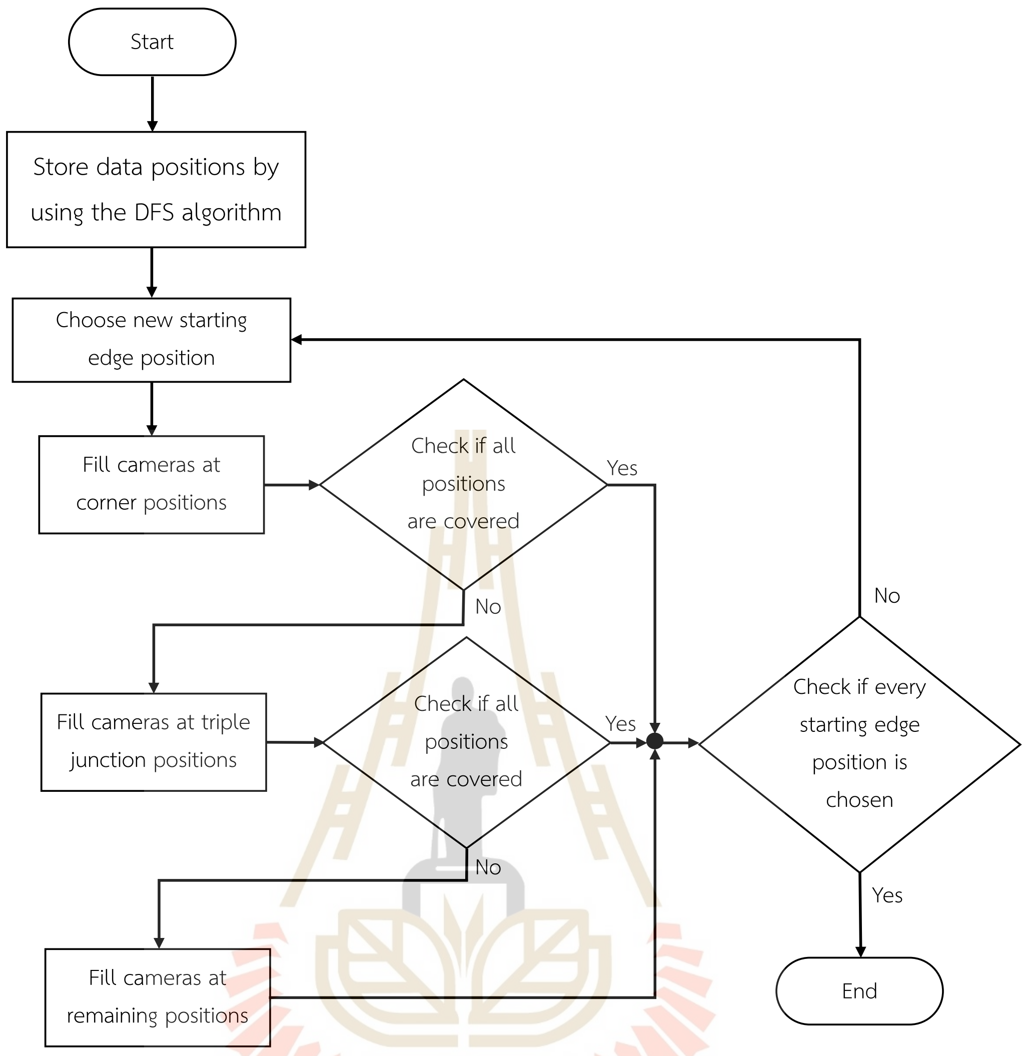

Start
Store data positions by using the DFS algorithm
Choose new starting edge position
Fill cameras at corner positions
Check if all positions are covered
Yes
No
Fill cameras at triple junction positions
Check if all positions are covered
Yes
No
Fill cameras at remaining positions
Check if every starting edge position is chosen
No
Yes
End

Figure 3.15 Flowchart of the main algorithm.

In figure 3.16, we denote that S is a starting position, E is an edge position, C is a corner position, 3 is a triple position, 4 is a crossroad position, 1 is a camera position, and X is an area that is covered by cameras.

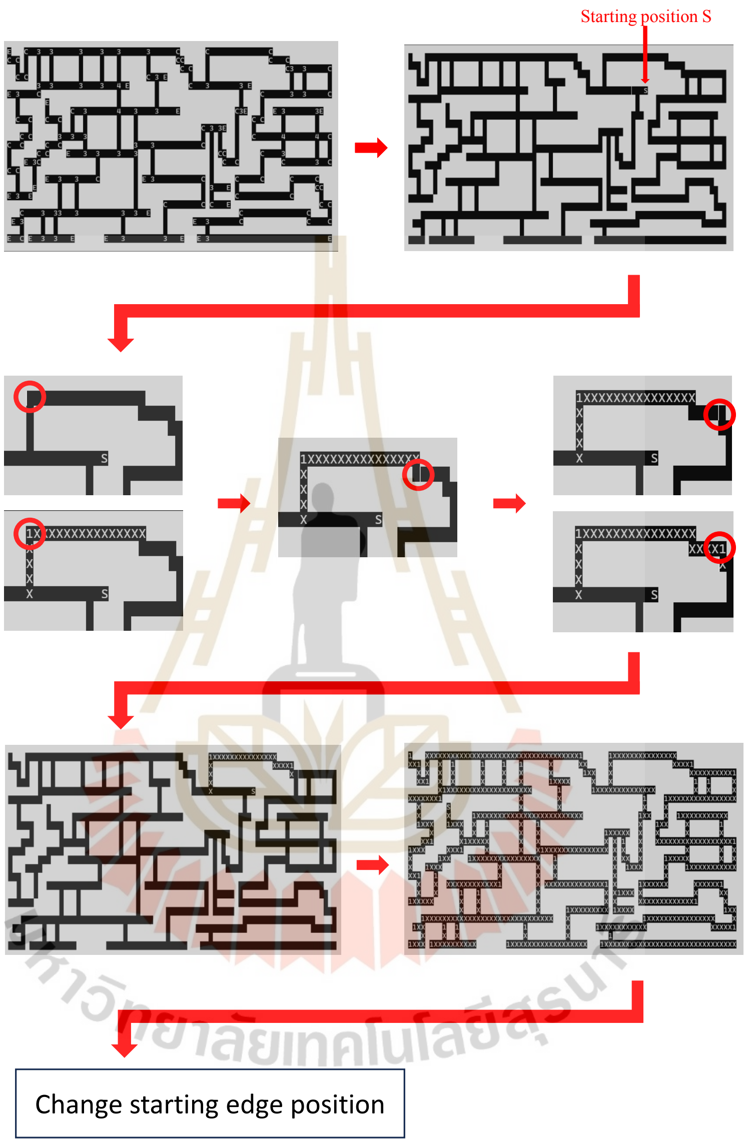


Figure 3.16 Step of the main algorithm.

3.4 Brute force algorithm

In the main algorithm, we choose the result with the minimum number of cameras for every starting position. However, it does not guarantee that it is the minimum number of cameras because, in some cases, the result obtained by the main algorithm can reduce some camera positions. To compare the efficiency of main algorithm, it is necessary to use the brute force algorithm to find the minimum number of cameras. The algorithm starts by placing one camera at every possible position and then checks if these cameras can cover every area within the unit orthogonal polygon. If not, we add more cameras, starting with 2, 3, and so on, until we find the minimum of cameras that can cover the entire area.

Suppose that unit orthogonal polygon is P, the concept of Brute force algorithm for solving the camera placement problem can be outlined with the following steps:

1. Display an image of a unit orthogonal polygon along with counting the number of edges, corners and triple junctions.
2. Store the data of all possible of camera positions such as edge positions, corner positions, and triple junction positions in set $\widetilde{V}$. We obtain that
$$\left|\widetilde{V}\right| = \ell = E + C + 2T.$$
3. Let K be a set of camera positions such that $K \subset \widetilde{V}$.
   - Suppose that $|K| = 1$ so there are $\binom{\ell}{1}$ possible scenario. Then, checks if these cameras can cover every area within the unit orthogonal polygon. If not, go to next step.
   - Suppose that $|K| = 2$ so there are $\binom{\ell}{2}$ possible scenario. Then, checks if these cameras can cover every area within P. If not, go to next step.
   - Increase the number of element in set K, until we find the set K that can cover every area within P.

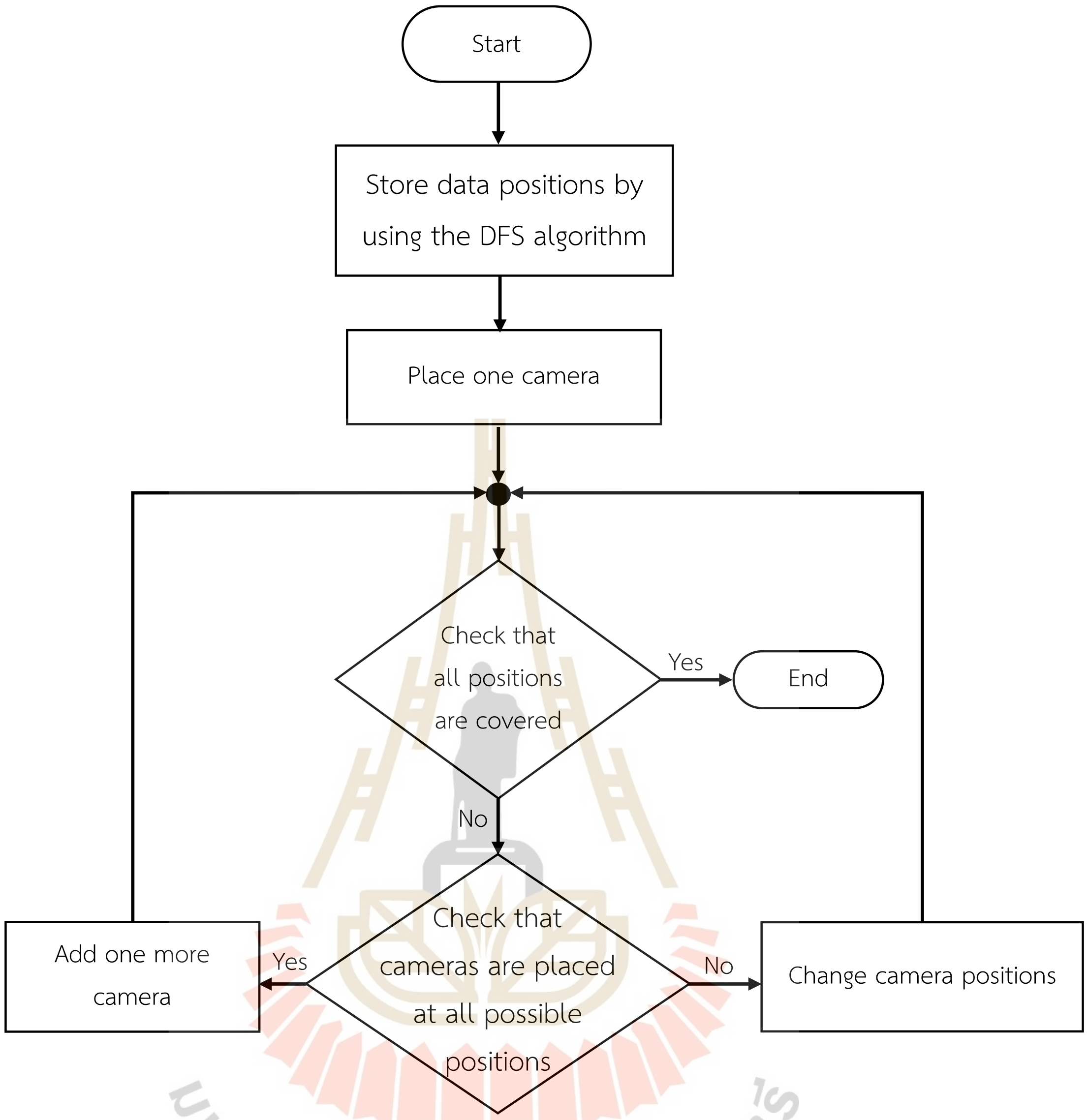

Start
Store data positions by using the DFS algorithm
Place one camera
Check that all positions are covered
Yes
End
No
Check that cameras are placed at all possible positions
Yes
Add one more camera
No
Change camera positions

Figure 3.17 Flowchart of the brute force algorithm.

In figure 3.18, we denote that E is an edge position, C is a corner position, 3 is a triple position, 4 is a crossroad position, 1 is a camera position and X is a position that is covered by cameras.

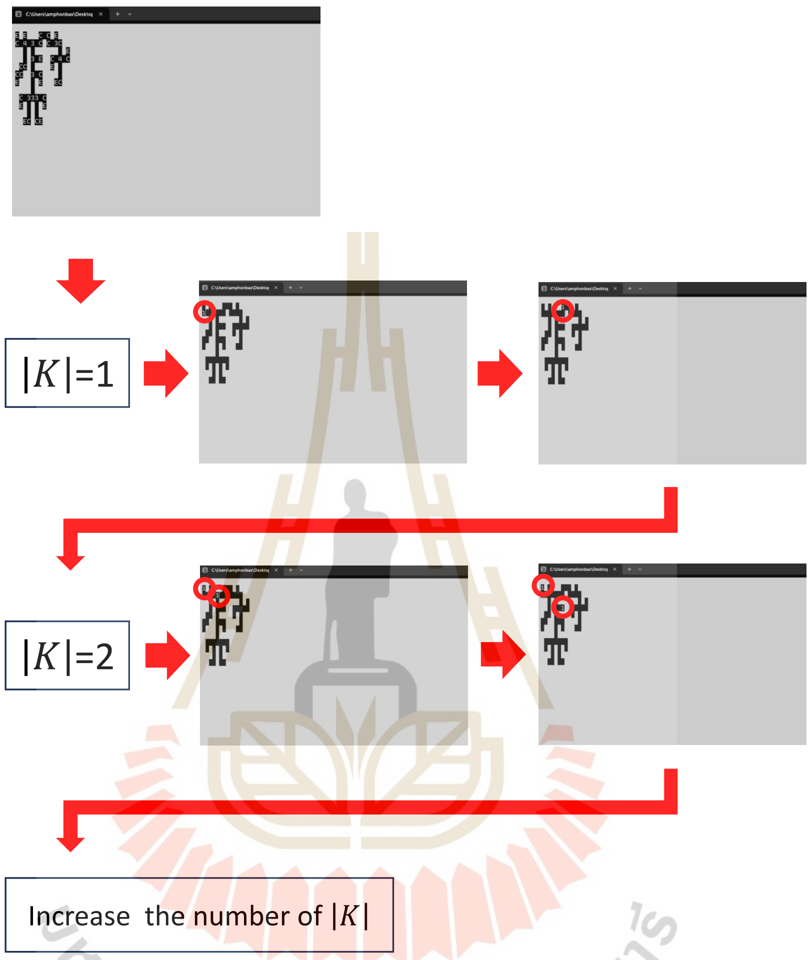


Figure 3.18 Step of the brute force algorithm.

CHAPTER IV

RESULTS

In this thesis, we develop the main algorithm and the brute force algorithm (see more in the appendix), and this chapter presents the results of both algorithms for the unit orthogonal.

4.1 Main algorithm and brute force algorithm

The result of unit orthogonal polygon with or without holes by using Main algorithm and Brute force algorithm are shown in table 4.1 and table 4.2

Table 4.1 The result of small unit orthogonal polygon by using main algorithm and brute force algorithm.

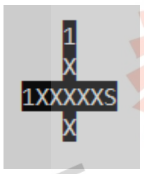
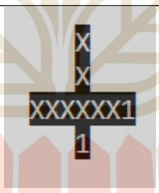
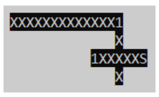
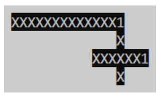
No.	Main algorithm	Brute force algorithm	Information
1	 Number of cameras=2 Time=0.717 s	 Number of cameras=2 Time=0.170 s	Vertices=12 Blocks=10 Edge=4, Corner=0 Triple=0, Crossroad=1 All camera positions=4
2	 Number of cameras=2 Time=0.768 s	 Number of cameras=2 Time=0.215 s	Vertices=14 Blocks=23 Edge=4, Corner=1 Triple=0, Crossroad=1 All camera position=5

Table 4.1 (Continued) The result of small unit orthogonal polygon by using main algorithm and brute force algorithm.

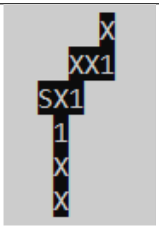
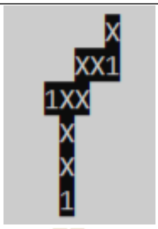
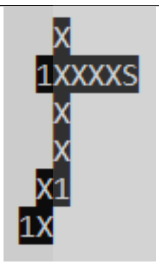
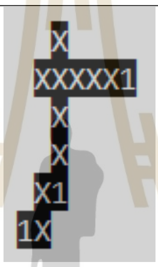
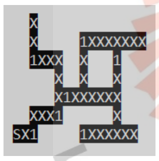
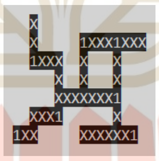
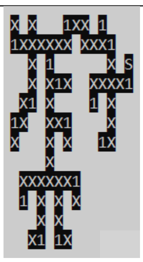
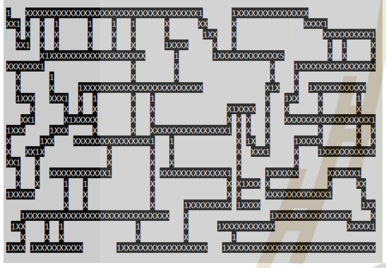
No.	Main algorithm	Brute force algorithm	Information
3	 Number of cameras=3 Time=0.653 s	 Number of cameras=3 Time=0.238 s	Vertices=14 Blocks=10 Edge=3, Corner=3 Triple=1, Crossroad=0 All camera position=8
4	 Number of cameras=3 Time=0.719 s	 Number of cameras=3 Time=0.250 s	Vertices=18 Blocks=13 Edge=4, Corner=3 Triple=0, Crossroad=1 All camera position=7
5	 Number of cameras=7 Time=0.939 s	 Number of cameras=7 Time=715.483 s	Vertices=32 Blocks=42 Edge=5, Corner=6 Triple=5, Crossroad=0 All camera position=21

Table 4.2 The result of large unit orthogonal polygon by using main algorithm and brute force algorithm.

No.	Main algorithm	Brute force algorithm	Information
1	 Number of cameras=16 Time=2.114 s	No data	Vertices=84 Blocks=64 Edge=13, Corner=18 Triple=7, Crossroad=2 All camera position=45
2	 Number of cameras=71 Time=6.790 s	No data	Vertices=334 Blocks=659 Edge=30, Corner=73 Triple=56, Crossroad=4 All camera position=215

When considering the result of some small unit orthogonal polygon (see table 4.1), it will be found that the number of cameras, when using the main algorithm, is minimum, but the execution time of brute force algorithm will increase when the camera positions increase. When observing example No.5 in the table 4.1, it can be found that the execution time of both algorithms differs significantly.

When considering the large unit orthogonal polygon with a large number of camera positions (see table 4.2), it found that the execution time of the brute force algorithm becomes excessively high to wait for results. However, upon using the main algorithm, results can be generated within a few seconds. Although the obtained results may not represent the minimum number of cameras, considering the execution time, the results fall within an acceptable range.

CHAPTER V

CONCLUSION DISCUSSION AND RECOMMENDATION

5.1 Conclusion

Using the main algorithm to find the number of cameras that can monitor all areas of a unit orthogonal polygon with or without holes cannot guarantee that it provides the minimum number. However, the brute force algorithm is one algorithm that can find the minimum number of cameras. Therefore, we utilize the brute force algorithm to assess the effectiveness of the main algorithm, leading us to the following conclusion:

1. When considering some small unit orthogonal polygon, both algorithms can find the same number of cameras. However, when comparing their execution times, it is evident that the execution time of the brute force algorithm is lower than that of the main algorithm only when the number of camera positions is very small. But, as the number of camera positions increases, the execution time of the brute force algorithm significantly increases.

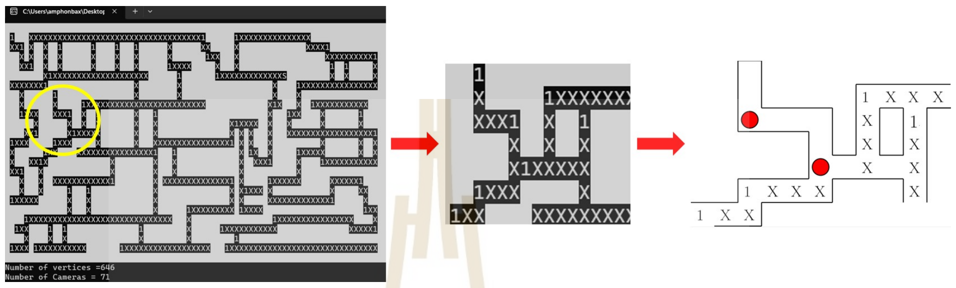
2. The difference in execution time between both algorithms becomes evident when using them to find the number of cameras in a large unit orthogonal polygon. The execution time of the brute force algorithm becomes excessively high, making it impractical to wait for results. However, upon using the main algorithm, results can be generated within a few seconds.

3. When comparing the time complexity of both algorithms, it can be observed that the time complexity of the main algorithm and the brute force algorithm are $O(n)$ and $O(2^n)$, respectively, where n is the number of vertices of the polygon.

Therefore, when considering the results of the number of cameras and time complexity, the main algorithm is one algorithm that can find a suitable number of cameras while also using a few execution times for processing.

5.2 Discussion

Once we have placed cameras to cover all positions using the main algorithm, we can further improve the results by finding ways to reduce the number of cameras. For example, we can consider if it is possible to move a camera from a corner position to a triple junction position, as shown in the figure 5.1.



```
Number of vertices =646
Number of Cameras = 71
```

Figure 5.1 The number “1” are refer to camera positions and the two circle positions are the improvement of cameras positions.

5.3 Recommendation

Since using the brute force algorithm to find the minimum number of cameras required to cover the entire area in a unit orthogonal polygon with or without holes, takes a substantial amount of time for program execution. To improve the efficiency of the brute force algorithm, we can consider the following approaches.

1. Find the exact camera placement positions or adjust positions to allow cameras to be placed in those specific locations. For instance, according to Theorem 3.1, we can position cameras at specific corner positions. This reduces 2 positions needed to run the brute force algorithm.

2. In some cases, it is unnecessary to run the brute force algorithm starting from $|K| = 1$. If we can determine a lower bound on the number of cameras needed in terms of the count of edge positions (E), corner positions (C), triple junction positions (T), and crossroad positions (F), we can reduce unnecessary steps in the brute force algorithm.

REFERENCES

REFERENCES

Aggarwal, A. (1984). The Art Gallery Theorem: *Its variations, applications, and algorithmic aspects*. PhD thesis, Johns Hopkins University.

Appel, K., and Haken, W. (1975). Every planar map is four colorable. *Bulletin of the American Mathematical Society, 81*(5), 105-110.

Bjorling-Sachs, I., and Souvaine, D. (1991). *A tight bound for guarding general polygons with holes*. Technical Report LCSR-TR-165, Rutgers University, Department of Computer Science.

Chvátal, V. (1975). A combinatorial theorem in plane geometry. *Journal of Combinatorial Theory, Series B, 18*, 39–41.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Fisk, S. (1978). A short proof of Chvátal's watchman theorem. *Journal of Combinatorial Theory, Series B, 24*(3), 374.

Garey, M. R., and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY: W. H. Freeman.

Hoffmann, F., Kaufman, M., and Kriegel, K. (1991). The art gallery problem for polygons with holes. *Proceedings of the 32nd Symposium on Foundations of Computer Science*, 39-48.

Kahn, J., Klawe, M., and Kleitman, D. (1983). Traditional galleries require fewer watchmen. *SIAM Journal on Algebraic and Discrete Methods, 4*(2), 194–206.

Michael, T. S., and Pinciu, V. (2016). The orthogonal art gallery theorem with constrained guards. *Electric notes in discrete mathematics, 54*, 27-32.

O'Rourke, J. (1987). *Art Gallery Theorems and Algorithms*. Oxford University Press.

Stewart, I. (2015). *The Math Book: From Pythagoras to the 57th Dimension* (4th ed.). Basic Books.

West, D. B. (1996). *Introduction to graph theory.* Prentice Hall.

APPENDICES

APPENDIX A

APPLICATION OF C++ CODE FOR MAIN ALGORITHM AND BRUTE FORCE ALGORITHM

This chapter presents some C++ code using in this thesis.

A.1 C++ code for the main algorithm

The main algorithm starts from one edge position then recursively moves and places cameras. We ensure that these cameras can see every area within the unit orthogonal polygon. Then, we select a new starting edge position and consider all possible starting points. By different start positions may obtain different results. We choose the result with the optimal number of cameras for every starting positions. However, we can not guarantee that the results is minimum number of cameras.

```
#include <iostream>
using namespace std;
#include <iomanip>
#include <conio.h>
#include <windows.h>
#include <vector>
#define maxx 80
#define maxy 25
#define SleepT 0
#define SleepT2 0
int unused=0;
void gotoxy(short x, short y) {
COORD pos = {x, y};
SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), pos);
}

char scr[maxy][maxx]=
{
"###############################################################################",
"######    #########     #######################################################",
"###### ################ #######################################################",
"####                    #######################################################",
"###### ################ #######################################################",
"###### ################ #######################################################",
"###                             ############################################################",
"####### ################### ################################################################",
"####### ################### ################################################################",
"############################################################################################",
"############################################################################################",
"############################################################################################",
"############################################################################################",
"############################################################################################",
"############################################################################################",
"############################################################################################",
"############################################################################################",
"############################################################################################",
"############################################################################################",
"############################################################################################",
"############################################################################################",
"############################################################################################",
"############################################################################################",
"############################################################################################",
"############################################################################################"
};

int E[maxy][maxx],countmaze=0,countfld=0,fld[maxy][maxx],weight[maxy][maxx],
filltable[maxy][maxx],reduce=0,countC=0,countE=0,countT=0,count4=0,countCam=0,
minCamera,minX,minY;
char showC[6] = "XEC34";
vector<int> pointx,pointy,Ex,Ey;
// Initialize the variables and arrays used in the program
void init_var()
{
    // Create a vector of integers from 0 to maxy-1 and store it in pointy
    for (int l=0;l<maxy;l++) pointy.push_back(l);

    // Create a vector of integers from 0 to maxx-1 and store it in pointx
    for (int k=0;k<maxx;k++) pointx.push_back(k);

    // Set all elements in weight and fld arrays to 0,
    // and all elements in E array to 0
    memset(weight,0,sizeof(weight));
    memset(fld,0,sizeof(fld));

    // Loop through each row and column of the maze
    for (int i=0;i<maxy;i++)
    {
        for (int j=0;j<maxx;j++)
        {
            // If the current character is a space, print a space and set
            weight and E arrays accordingly
            if (scr[i][j]==' ') { cout << ' '; weight[i][j] = 1;E[i][j]=0;countmaze++; }
            // If the current character is not a space, print a block
            character and set weight and fld arrays accordingly
            else { cout << char(219); weight[i][j] = 0; fld[i][j]=0;}
        }
        cout<<endl;
    }
}


void printmaze ()         // Print the maze
{
    // Loop through each row and column of the maze
    for (int i=0;i<maxy;i++)
    {
        for (int j=0;j<maxx;j++)
        {
            // If the current character is a space, print a space and set
            weight, fld, and E arrays accordingly
            if (scr[i][j]==' ') { cout << ' '; weight[i][j] = 1; fld[i][j]=1;E[i][j]=0;}
            // If the current character is not a space, print a block
            character and set weight and fld arrays accordingly
            else { cout << char(219); weight[i][j] = 0; fld[i][j]=0;}
        }
        cout<<endl;
    }
}

void flood1(int x, int y) {
    int c, d, j = x + 1;
    do {
        c = pointy[y];
        d = pointx[j];
        if (fld[c][d] == 1 )
        {   gotoxy(d,c) ; cout << 'X' ;
            fld[c][d] = 2;
            countfld ++ ;
        }
        j++;
    } while (j < maxx && weight[c][pointx[j]] !=0);
}

void flood2(int x, int y) {
    int a, b, i = y - 1;
    do {
        a = pointy[i];
        b = pointx[x];
        if (fld[a][b] == 1)
            {gotoxy(b,a); cout << 'X';
            fld[a][b] = 2;countfld ++ ;
        }
        i--;
    } while (i >= 0 && weight[pointy[i]][b] != 0);
}

void flood3(int x, int y) {
    int c, d, j = x - 1;
    do {
        c = pointy[y];
        d = pointx[j];
        if (fld[c][d] == 1)
```

```
            {gotoxy(d,c) ; cout << 'X' ;
            fld[c][d] = 2;countfld ++ ;
        }
        j--;
    } while (j >= 0 && weight[c][pointx[j]] != 0);
}

void flood4(int x, int y) {
    int a, b, i = y + 1;
    do {
        a = pointy[i];
        b = pointx[x];
        if (fld[a][b] == 1)
            {gotoxy(b,a); cout << 'X';
            fld[a][b] = 2; countfld ++ ;
        }
        i++;
    } while (i < maxy && weight[pointy[i]][b] != 0);
}

void walk(int x, int y)
{
     if (x>=0 && x<maxx && y>=0 && y<maxy && scr[y][x]==' ')
     {
         gotoxy(x,y);
         cout << '#';
         scr[y][x]='+';
         walk(x+1,y);
         walk(x,y-1);
         walk(x-1,y);
         walk(x,y+1);
         gotoxy(x,y);
         int w1,w2,w3,w4,w5,w6,w7;
         w1=                      weight[y-1][x]+
             weight[y][x-1]+                        weight[y][x+1]+
                                 +weight[y+1][x];
         w2= weight[y][x-1]+weight[y][x+1];
         w3= weight[y-1][x]+weight[y+1][x];

         w4= weight[y-1][x]+weight[y][x-1];
         w5= weight[y-1][x]+weight[y][x+1];
         w6= weight[y+1][x]+weight[y][x-1];
         w7= weight[y+1][x]+weight[y][x+1];
         if (w2!=2 && w3!=2 || w1==3 || w1==4)
            {
                cout << showC[w1];
                if  (w1==1)  {countE++; Ex.push_back(x);Ey.push_back(y);}
                if  (w1==2)  {countC++;}
                if  (w1==3)  {countT++;}
                if  (w1==4)  {count4++;}
            } else  {cout << ' ';}
     }
}
void fillcam(int x,int y,int a,int b)
{
    if (x>=0 && x<maxx && y>=0 && y<maxy)
    {

     if (scr[y][x]==' ' )
     {
         scr[y][x]='#';
         Sleep(SleepT);
         gotoxy(x,y);
         int w1,w2,w3,w4,w5,w6,w7;
         w1=                       weight[y-1][x]+
             weight[y][x-1]+                        weight[y][x+1]+
                                 +weight[y+1][x];
         w2= weight[y][x-1]+weight[y][x+1];
         w3= weight[y-1][x]+weight[y+1][x];

         w4= weight[y-1][x]+weight[y][x-1];
         w5= weight[y-1][x]+weight[y][x+1];
```

```
w6= weight[y+1][x]+weight[y][x-1];
w7= weight[y+1][x]+weight[y][x+1];

if (w2!=2 && w3!=2 || w1==3 || w1==4)
 {
  if (x!=a || y!=b)
   {
    if(w1==2 && w4==1 && w5==2 )
     {
   if(fld[y][x]==1&&(fld[y-1][x]+fld[y+1][x]+fld[y][x-1]+fld[y][x+1])==2)
                {
                    cout << '1';fld[y][x]=2;countfld ++ ;countCam++;
                    scr[y][x] = showC[w1];
                    flood1(x,y) ;
                    flood2(x,y) ;
                }
            }
       if (w1==2 && w4==1 && w5==0 )
       {
   if(fld[y][x]==1&&(fld[y-1][x]+fld[y+1][x]+fld[y][x-1]+fld[y][x+1])==2)
                    {
                        cout << '1';fld[y][x]=2;countfld ++ ;countCam++;
                        scr[y][x] = showC[w1];
                        flood3(x,y) ;
                        flood4(x,y) ;
                    }
            }
        if (w1==2 && w4==2 )
         {
    if(fld[y][x]==1&&(fld[y-1][x]+fld[y+1][x]+fld[y][x-1]+fld[y][x+1])==2)
                    {
                    cout << '1';fld[y][x]=2;countfld ++ ;countCam++;
                    scr[y][x] = showC[w1];
                    flood2(x,y) ;
                    flood3(x,y) ;
                    }
            }
         if  (w1==2 && w4==0 )
         {
     if(fld[y][x]==1&&(fld[y-1][x]+fld[y+1][x]+fld[y][x-1]+fld[y][x+1])==2)
                {
                    cout << '1';fld[y][x]=2;countfld ++ ;countCam++;
                    scr[y][x] = showC[w1];
                    flood1(x,y) ;
                    flood4(x,y) ;
                }
            }
         if  (w1==1 && w2==0 && w4==0)
         {
       if(fld[y][x]==1&&(fld[y-1][x]+fld[y+1][x]+fld[y][x-1]+fld[y][x+1])==2)
                {
                    cout << '1';fld[y][x]=2;countfld ++;countCam++;
                    scr[y][x] = showC[w1];
                    flood4(x,y) ;
                }
            }
         if  (w1==1 && w2==1 && w4==0)
         {
        if(fld[y][x]==1&&(fld[y-1][x]+fld[y+1][x]+fld[y][x-1]+fld[y][x+1])==2)
                {
                cout << '1';fld[y][x]=2;countfld ++;countCam++;
                scr[y][x] = showC[w1];
                flood1(x,y) ;
                }
            }
         if  (w1==1 && w2==0 && w4==1)
         {
         if(fld[y][x]==1&&(fld[y-1][x]+fld[y+1][x]+fld[y][x-1]+fld[y][x+1])==2)
                {
                cout << '1';fld[y][x]=2;countfld ++;countCam++;
                scr[y][x] = showC[w1];
                flood2(x,y) ;
```

```
                              }
                          }
                        if (w1==1 && w2==1 && w4==1)
                        {
                        if(fld[y][x]==1&&(fld[y-1][x]+fld[y+1][x]+fld[y][x-1]+fld[y][x+1])==2)
                              {
                              cout << '1';fld[y][x]=2;countfld ++;countCam++;
                              scr[y][x] = showC[w1];
                              flood3(x,y) ;
                              }
                          }
                          if (w1==3)
                          {
                              scr[y][x] = showC[w1];
                          }
                          if (w1==4)
                          {
                              scr[y][x] = showC[w1];
                          }
                      } else cout << 'S' ;
                  }
            fillcam(x+1,y,a,b);
            fillcam(x,y-1,a,b);
            fillcam(x-1,y,a,b);
            fillcam(x,y+1,a,b);
     }
    }
  }

void check()
{
    int m,n ;
        for (m=0;m<maxx;m++)
            for (n=0;n<maxy;n++)
                    if (weight[n][m]!=0 && (weight[n-1][m] + weight[n+1][m]
                    +weight[n][m-1] + weight[n][m+1]==3)  )
                    {
            if (fld[n][m]==1 && (fld[n-1][m] + fld[n+1][m] +fld[n][m-1] + fld[n][m+1]==4))
                    {
                        gotoxy(m,n);cout << '1';fld[n][m]=2;countfld ++;countCam++;
                        if (fld[n-1][m]==1) {flood2(m,n);}
                        if (fld[n+1][m]==1) {flood4(m,n);}
                        if (fld[n][m-1]==1) {flood3(m,n);}
                        if (fld[n][m+1]==1) {flood1(m,n);}
                        else ;
                        }
                    }
}

void check1()
{
    int i,j ;
        for (i=0;i<maxx;i++)
            for (j=0;j<maxy;j++)
            {
                    if (fld[j][i]==1)
                    {
                        gotoxy(i,j);cout << '1';fld[j][i]=2;countfld ++;countCam++;
                        flood1(i,j) ;flood2(i,j) ;flood3(i,j) ;flood4(i,j) ;
                    }

            }

}

int main()
{
    int m,n,startX,startY,state=0;
    minCamera=maxx*maxy;
    init_var();
    for (int k=0;k<maxy&& state==0;k++)
    {
```

```
    for (int j=0;j<maxx && state==0;j++)
    {
        if (scr[k][j]==' ')
        {
            startY=k;
            startX=j;
            state=1;
        }
    }
}
walk(startX,startY);
for (int i=0;i<Ex.size(); i++)
{
    gotoxy(0,0);
    for (m=0;m<maxx;m++)
        for (n=0;n<maxy;n++)
            if (weight[n][m]==1){scr[n][m]=' ';}
    printmaze ();
    countCam=0;countfld=0;
    fillcam(Ex[i],Ey[i],Ex[i],Ey[i]);
    if (countmaze != countfld) {check() ; check1() ;}
    if (minCamera>countCam)
    {
        minCamera=countCam;
        minX=Ex[i];
        minY=Ey[i];
    }
}
gotoxy(0,0);
for (m=0;m<maxx;m++)
    for (n=0;n<maxy;n++)
        if (weight[n][m]==1){scr[n][m]=' ';}
printmaze ();
countCam=0;countfld=0;
fillcam(minX,minY,minX,minY);
gotoxy(minX,minY);
```

```
    cout <<'S';
    if (countmaze != countfld)
    {check() ;
    check1() ;}
    gotoxy(minX,minY);
    cout <<'S';
    gotoxy(0,maxy);
    cout << "Number of vertices =" << 2*(countE+countC+countT)+4*count4 << endl ;
    cout << "Number of Cameras = " << minCamera << endl;
    cout << "Number of Edge = " << countE << endl;
    cout << "Number of Coner = " << countC << endl;
    cout << "Number of Three = " << countT << endl;
    cout << "Number of Four = " << count4 << endl;
    cout << "Start point Ex= " << minX << endl;
    cout << "Start point Ey= " << minY << endl;
    return 0;
}
```

A.2 C++ code for brute force algorithm

The algorithm starts by placing one camera at every possible position and then checks if these cameras can cover every area within the unit orthogonal polygon. If not, we add more cameras, starting with 2, 3, and so on, until we find the minimum of cameras that can cover the entire area.

```
#include <iostream>
using namespace std;
#include <iomanip>
#include <conio.h>
#include <windows.h>
#include <vector>
#define maxx 80
#define maxy 25
#define SleepT 0
#define SleepT2 0
```

```
int unused=0;
int Area=0;
void gotoxy(short x, short y) {
COORD pos = {x, y};
SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), pos);
}

char scr[maxy][maxx]=
{
"###############################################################################",
"######    #########     #######################################################",
"###### ################ #######################################################",
"####                    #######################################################",
"###### ################ #######################################################",
"###### ################ #######################################################",
"###                         ###################################################",
"###### ################ #######################################################",
"###### ################ #######################################################",
"###############################################################################",
"###############################################################################",
"###############################################################################",
"###############################################################################",
"###############################################################################",
"###############################################################################",
"###############################################################################",
"###############################################################################",
"###############################################################################",
"###############################################################################",
"###############################################################################",
"###############################################################################",
"###############################################################################",
"###############################################################################",
"###############################################################################",
"###############################################################################"
};
```

```
int weight[maxy][maxx],filltable[maxy][maxx],countC[4]={0,0,0,0},
numFill=0,camera=0,status=0,lowerbound;
char showC[6] = "XEC34";
vector<int> pointx,pointy,type,Allpoint,position,position_type;

void walk(int x,int y)
{
    if (x>=0 && x<maxx && y>=0 && y<maxy && scr[y][x]==' ')
    {
        if (x==maxx-2 && y==maxy-1) {cout << 'X'; getch();}
        else
        {
        gotoxy(x,y);
        cout << '#';
        scr[y][x]='#';
        walk(x+1,y); //right
        walk(x,y-1); //up
        walk(x-1,y); //left
        walk(x,y+1); //down

        gotoxy(x,y);
        int w1,w2,w3,w4,w5,w6,w7;
        w1=             weight[y-1][x]+
            weight[y][x-1]+             weight[y][x+1]+
                        +weight[y+1][x];
        w2= weight[y][x-1]+weight[y][x+1];
        w3= weight[y-1][x]+weight[y+1][x];

        w4= weight[y-1][x]+weight[y][x-1];
        w5= weight[y-1][x]+weight[y][x+1];
        w6= weight[y+1][x]+weight[y][x-1];
        w7= weight[y+1][x]+weight[y][x+1];

        if (w2!=2 && w3!=2 || w1==3 || w1==4)
          {
              cout << showC[w1];
```

```
                    scr[y][x] = showC[w1];
                    countC[w1-1]++;
                    if(w1==1)
                    {    pointx.push_back(x);
                         pointy.push_back(y);
                         type.push_back(0);
                    }
                    if(w4==2&& w1!=4)
                    {    pointx.push_back(x);
                         pointy.push_back(y);
                         type.push_back(1);
                    }
                    if(w5==2&& w1!=4)
                    {    pointx.push_back(x);
                         pointy.push_back(y);
                         type.push_back(2);
                    }
                    if(w6==2&& w1!=4)
                    {    pointx.push_back(x);
                         pointy.push_back(y);
                         type.push_back(3);
                    }
                    if(w7==2&& w1!=4)
                    {    pointx.push_back(x);
                         pointy.push_back(y);
                         type.push_back(4);
                    }
               }
          else  { cout << ' ';  scr[y][x]=' '; }
          }
     }
}

void clear_maze()
{
     int i,j;
```

```
    for(j=0;j<maxy;j++)
        for(i=0;i<maxx;i++)
        {
             if(weight[j][i]!=0)
             {
                    gotoxy(i,j);
                    cout << ' ';
                    scr[j][i] =' ';
                    filltable[j][i]=0;
             }
             else
             {
                    filltable[j][i]=9;
             }
        }
}

int direction[4][2]={{0,-1},{0,1},{-1,0},{1,0}},
    UP=0,DW=1,LF=2,RT=3;

void filldirectway(int x,int y,int dir)  //UP=0,DW=1,LF=2,RT=3;
{
    if (weight[y][x]!=0)
    {
        if (scr[y][x]==' ')
        {
            gotoxy(x,y);
            cout << 'X';
            scr[y][x]='X';
            numFill++;
        }

        filldirectway(x+direction[dir][0],y+direction[dir][1],dir);
    }
}
```

```
void fillcorner(int x,int y,int cornertype)
{
    if(cornertype==0)
    {
        filldirectway(x,y,0);
        filldirectway(x,y,1);
        filldirectway(x,y,2);
        filldirectway(x,y,3);
    }
    if(cornertype==1)
    {
        filldirectway(x,y,0);
        filldirectway(x,y,2);
    }

    if(cornertype==2)
    {
        filldirectway(x,y,0);
        filldirectway(x,y,3);
    }
    if(cornertype==3)
    {
        filldirectway(x,y,1);
        filldirectway(x,y,2);
    }
    if(cornertype==4)
    {
        filldirectway(x,y,1);
        filldirectway(x,y,3);
    }
}

void init_var()          //show maze
{
    memset(weight,0,sizeof(weight));
    for (int i=0;i<maxy;i++)
    {
        for (int j=0;j<maxx;j++)
            if (scr[i][j]==' ') { cout << ' '; weight[i][j] = 1; Area++ ;} else
                                  { cout << char(219); weight[i][j] = 0;}
        cout<<endl;
    }
}

void generateCombinations(vector<int>& Allpoint,vector<int>& combination,int start,int k)
{ if (k == 0)
    {
        for (int n : combination)
        {
            fillcorner(pointx[n],pointy[n],type[n]);
        }
        if(numFill>=Area && combination.size()<camera)
        {       position=combination;
                camera=combination.size();
                status=1;
        }
        clear_maze();
        numFill=0;
        return;
    }
    for (int i = start; i < Allpoint.size()&& status==0; i++)
    {
        combination.push_back(Allpoint[i]);
        generateCombinations(Allpoint, combination, i + 1, k - 1);
        combination.pop_back();
    }
}

void displayCombinations(vector<int>& Allpoint)
{   int k;
    for (k = lowerbound; k <= Allpoint.size(); k++)
    {   vector<int> combination;
```

```
            gotoxy(0,maxy+2);
            cout<<"k= "<< k << endl;
            generateCombinations(Allpoint, combination, 0, k);
            if(status==1)
                return;
        }
}

int main()
{   int k,j,c,startX,startY,check=0;
    init_var();
    for (int k=0;k<maxy&& check==0;k++)
    {
        for (int j=0;j<maxx && check==0;j++)
        {
             if (scr[k][j]==' ')
             {
                   startY=k;
                   startX=j;
                   check=1;
             }
        }
    }
    walk(startX,startY);
    lowerbound=1;
    for(k=0;k<type.size();k++)
    {
        Allpoint.push_back(k);
    }
    clear_maze();
    camera=pointx.size();
    displayCombinations(Allpoint);
    clear_maze();
    for(k=0;k<position.size();k++)
    {
        fillcorner(pointx[position[k]],pointy[position[k]],type[position[k]]);
    }
    for(k=0;k<position.size();k=k+c+1)
    {    c=0;
        for(j=1;j<=3&& k+j<position.size();j++)
        {
          if(pointx[position[k]]==pointx[position[k+j]]&&
              pointy[position[k]]==pointy[position[k+j]])
                c++;
        }
        gotoxy(pointx[position[k]],pointy[position[k]]);
        cout << c+1;
    }
    gotoxy(0,maxy+1);
    int sum=0;
    for (int  i=0;i<4;i++) sum+=countC[i];
    cout << "All Number of position =" << pointx.size() << endl;
    cout << "Minimum Number of Cameras = " << camera << endl;
    cout << "Area = " << Area << endl;
    cout << "Number of end points = " << countC[0]<< endl;
    cout << "Number of corners = " << countC[1]<< endl;
    cout << "Number of T-junctions = " << countC[2]<< endl;
    cout << "Number of crossed-points = " << countC[3]<< endl;
    return  0;
}
```

CURRICULUM VITAE

NAME : Amphon Kliaram **GENDER :** Male

EDUCATION BACKGROUND:

- Bachelor of Science (Mathematics), Suranaree University of Technology, Thailand, 2020

SCHOLARSHIP:

- Development and promotion of Science and Technology Talents Project (DPST), Thai government scholarship for graduate honors student of Suranaree University of Technology in Bachelor degree and Master degree.

CONFERENCE:

- The 26th Annual Meeting in Mathematics and The 1st International Annual Meeting in Mathematics 2022 (AMM 2022), Online Conference, Thailand, May 18-20, 2022 “The study of envelope of family of lines that divide areas of triangle and quadrilateral”

EXPERIENCE:

- Teaching in Calculus I, Calculus II, and Calculus III for SUT dormitory.

- Teaching in SUT camp.