Nanyang Technological University, Singapore

India Connect@NTU Research Internship Program

School of Mechanical & Aerospace Engineering (MAE)

Internship Supervisor: Assoc Prof Cai Yiyu

Name: Rutwik Rajesh Bonde

Home College: Vishwakarma Institute of Technology, Pune

Project Topic: Art Gallery Problem and Algorithms

Project Area: Computational Geometry

Acknowledgement: I would like thank India Connect@NTU Research Program for giving me the opportunity to do this five months long research internship. I learnt a lot from this internship and got the idea of how the actual research is conducted. I would also like to thank Assoc Prof Cai Yiyu for guiding me throughout the project and clearing all my doubts during the project.

Objective: The main objective of this project is to develop an algorithm to solve the art gallery problem and to implement the same in python, to create a code which can directly give the exact number and position of the minimum guards required to guard an art gallery, a museum, a hall-way, a building-shell etc.

Introduction to the Art Gallery Problem and the Project

The art gallery problem is a problem to determine the number of guards that are required or are sufficient to cover or see every point in the interior of an art gallery. An art gallery can be viewed as a polygon with or without holes with a total of n vertices; and guards as points in polygon, or on vertex, or on the edge of the polygon. Any point P in the polygon is said to be visible from a guard G if the line segment joining P and G does not intersect the exterior of the polygon. If the stationary guards are placed anywhere inside the polygon, then they are referred as point guards. If the stationary guards are placed on vertex of the polygon, then those are called vertex guards. Whereas if the guards move inside the polygon, then they are referred as mobile guards and further if these mobile guards are restricted to edges of polygon, then they are called edge guards.

Organization: The flow of the report is given in the organization, in terms of sections starting from **Section 1** till **Section 14**.

- In **Section 1**, Literature review is explained and in **Section 1.1**, the different triangulation methods are explained.
- In **Section 2**, one of the triangulation methods (Ear- Clipping Method) is discussed and implemented. **Section 2.1** gives the algorithm to find guards using the above triangulation method and **Section 2.2** shows the implementation of the Algorithm in python. In **Section 2.3** Limitation of the above algorithm is discussed.
- In **Section 3**, a new method to solve the Art Gallery Problem is discussed, that is the diagonalisation Method. **Section 3.1** gives the new algorithm using diagonalisation for vertex cover, whereas **Section 3.2** shows the implementation of the Algorithm in Python. In **Section 3.3** some more examples of the implementation of the algorithm are shown.
- In **Section 4**, the solutions two methods mentioned in **Section 2** and **Section 3** are compared.
- **Section 5** introduces us to the concept of Shrink Polygon. **Section 5.1** gives the new algorithm using diagonalisation method for shrink polygon – vertex cover, whereas **Section 5.2** shows the implementation of the Algorithm in python. **Section 5.3** discusses the limitation of the new algorithm.
- **Section 6** introduces us to the concept of Edge Cover. **Section 6.1** gives the new algorithm using diagonalisation method for Edge cover, whereas **Section 6.2** shows the implementation of the Algorithm in python. In, **Section 6.3** some more examples of the

implementation of the algorithm are shown. **Section 6.4** discusses the limitation of shrink polygon.

- In **Section 7**, Optimization of the solution of the Art Gallery Problem is discussed. **Section 7.1, Section 7.1.1, and Section 7.1.2** discuss about the dual tree, method to draw dual tree, and the implementation of the dual tree with the new algorithm in python respectively. In **Section 7.1.3**, the Limitation of the dual tree is discussed. **Section 7.2, Section 7.2.1, and Section 7.2.2** discuss the Delaunay triangulation, method to draw Delaunay triangulation and the implementation of the Delaunay triangulation in python respectively. **Section 7.3, Section 7.3.1, and Section 7.3.2** discuss the Voronoi Diagram, method to draw Voronoi diagram and the implementation of the Voronoi Diagram in python respectively. **Section 7.4** discusses the Medial Axis. And finally **Section 7.5 and Section 7.6** shows us the implementation of Delaunay triangulation with new algorithm of diagonalisation in python and the implementation of Voronoi diagram with new algorithm of diagonalisation in python respectively.
- In **Section 8**, Selective edge cover is discusses. **Section 8.1** shows us the implementation of the selective edge cover in python.
- **Section 9** shows us the results of the implementation of all the above mentioned algorithms in python on the Building Shell Example.
- **Section 10** provides us the table of the performance time of each algorithm and code in python, implemented on the building shell of 65 edges.
- **Section 11** includes the conclusion which is divided into contributions and limitations.
- **Section 12** gives the future scope of the project.
- Finally **Section 13** show all the references used for the project

(Section 1)

Literature Review

The Art Gallery Problem was first posed by Victor Klee (1973) for stationary guards. Vasek Chvatal **[1]** (1975) gave a proof for the same, and came up with a theorem, "[n/3] guards are occasionally necessary and always sufficient to cover a polygon of n vertices". Later, in 1978, Fisk **[2]** proved the same result using coloring method. After triangulating the polygons, by adding internal diagonals between vertices until no more can be added, the graph of triangulated polygon can be 3-colored. A coloring of a graph is an assignment of colors to nodes (vertices of triangles), one color per node, using no more than 3 colors, such that no two adjacent nodes are assigned the same color. Then, place the guards at those colored nodes which have fewer occurrences.

Further, based on Fisk's proof, Avis and Toussaint (1981) **[3]** developed an algorithm for positioning guards in the polygon. They propose to 3-color the polygon by a divide and conquer strategy. Their divide step partitions the polygon into two pieces, each of atleast [n]/4 vertices.

For the orthogonal polygons J. Kahn, M. Klawe and D. Kleitman (1983) **[4]**, gave a theorem, "For a simple orthogonal polygon, i.e., the edges of polygon are horizontal or vertical, needs at most [n]/4 stationary guards".

For Art Gallery Problem with holes, J. O'Rourke (1987) **[9]**, gave a perfect theorem for the same with upper bound solution, his theorem was, "Any polygon with n vertices and h holes can always be guarded with [n+2h]/3 **vertex guards**". Furthermore, F.Hoffmann, M.Kaufmann and K.Kriegel (1991) **[7]**; and I. Bjorling-Sachs and D. L. Souvaine (1995) **[8]**, proved independently that to guard a polygon with n vertices and h holes, [n+h]/3 **point guards** are always sufficient and occasionally necessary.

J. Czyzowicz, E. Rivera-Campo, N. Santoro, J. Urrutia and J.Zaks (1994) **[12]**, gave a theorem for rectangular art galleries proving that any rectangular art gallery with n rooms can be guarded with exactly [n/2] guards.

In 1981, Godfried Toussaint asked that how many edge/mobile guards are always sufficient to guard any polygon of n vertices. To this J. O'Rourke **[9]** gave a theorem, "To guard a simple polygon with n vertices, [n/4] mobile guards are always sufficient and occasionally necessary. In 1984 Aggarwal **[10]** showed that to guard an orthogonal polygon with n vertices, [3n+4]/16 mobile or edge guards are always sufficient and occasionally necessary. In 1996 E.Györi **[11]** proved that to guard an orthogonal polygon with n vertices and h holes, [3n+4h+4]/16 mobile guards are always sufficient and occasionally necessary.

For survey of art gallery problem, see O'Rourke **[9]**

Subir Kumar Ghosh **[5]**: Approximation algorithms for art gallery problems in polygons:

He presented the approximation algorithms for minimum vertex and edge guard problems for polygons with or without holes. The approximation algorithms partition the polygonal region into convex components and construct sets consisting of these convex components. Then the algorithms use an approximation algorithm for the minimum set covering problem on these constructed sets to compute the solution for the minimum vertex and edge guard problems. For simple polygons, approximation algorithms for both problems run in O(n^4) time and yield solutions that can be at most O(log n) times the optimal solution. For polygons with holes, approximation algorithms for both problems give the same approximation ratio of O(log n), but the algorithms take O(n^5) time.

Chromatic Art Gallery Problem **[6]**: One of the extensions of the art gallery problem is the chromatic art gallery problem, which aims to determine the minimum number of colors required to color a guard set, a set of vertices in an n-vertex polygon. A guard set is colored such that no two conflicting guards have the same color, where two conflicting guards are those whose areas of visibility overlap. A lower bound for this problem was found by Erickson and LaValle in 2010 **[14]**, which stated that for any value k there exists a polygon with (3k^2 +2) vertices such that the minimum number of guards needed is k.

Watchman route problem **[6]**: The watchman route problem is an optimization problem in geometry where guards are now mobile and the aim is to determine the shortest route that a watchman should take such that all points are visible from this route. The first polynomial time algorithm for this problem was determined by Carlsson, Johnsson and Nilsson in 1993 **[13]**.

Until now many people were focusing on applications, extensions of the Art Gallery Problem, and on improvement of the running time and time complexity of the algorithms for solving the art gallery problem by converting the polygons in small convex structures or triangles, but I haven't seen any work on the art gallery problem done just by considering the use of diagonals independently without thinking of any formation of triangulation or convex structures.

Hence in this project I am trying to solve the problem by only considering the use of diagonal in the polygon without thinking about any triangulation or convex component.

[But before I start working on the diagonalising method, I have worked on the polygon triangulation method itself (ear clipping method), to get better understanding of the flaws in the triangulation method, so that I can correct them in the further diagonalising algorithm.]

(Section 1.1)

Methods to Triangulate Polygons:

1) **Fan Center Method:**
   Basically, triangulation with one vertex (the fan center) is done which is shared by all triangles. It is trivial to triangulate any convex polygon in linear time into a fan triangulation, by adding diagonals from one vertex to all other vertices. Every triangulation of a simple polygon P of n vertices uses (n – 3) diagonals and has (n – 2) triangles.
2) **Ear Clipping Method**:
   Ears are defined as triangles with 2 sides being the edges of the polygon and 3rd inside it. A simple polygon with atleast 4 vertices without holes has atleast 2 ears. The ear clipping algorithm consists of finding an ear, removing it from the polygon and repeating until there is only one triangle left. This algorithm only works on polygon without holes.
   The time complexity of this algorithm is O(n^2).
3) **Monotone Polygon Triangulation**:
   We can consider a simple polygon as monotone with respect to a line L if any line orthogonal to L intersects the polygon at most twice. A monotone polygon can be split into two monotone chains. A polygon that is monotone with respect to the y-axis is called y-monotone. This method follows a greedy algorithm; it begins by walking on top to bottom (for y-monotone) while adding diagonals wherever it is possible. The time complexity of this algorithm is O(n) time.
4) **Triangulating a non – monotone polygon:**
   If a polygon is non monotone; it can be partitioned into monotone sub-polygons using sweep line approach; starting from top to bottom or vice-versa. The algorithm does not require polygon to be simple, thus it can be applied to polygon with holes. The time complexity to convert non-monotone to monotone polygons is O(nlogn).
5) **Seidel's Algorithm to triangulate the polygon:**
   Seidel's Algorithm basically had three main steps:
   - Decompose the polygon into trapezoids.
   - Decompose the polygon into Monotone polygon.
   - Triangulate the monotone polygon. (time complexity – O(n))

6) **Delaunay Triangulation:**

Delaunay triangulation of a vertex set (polygon vertices) is a triangulation of the vertex set with the property that no vertex in the vertex set falls in the interior of the circumcircle (circle that passes through all three vertices) of any triangle in the triangulation. To find the Delaunay triangulation, just join the points in vertex set such that they form a triangle whose circumcircle does not contain any other point of the vertex set.

(Section 2)

Ear Clipping Method of Triangulation:

Ears are defined as triangles with 2 sides being the edges of the polygon and 3rd inside it. A simple polygon with atleast 4 vertices without holes has atleast 2 ears. The ear clipping algorithm consists of finding an ear, removing it from the polygon and repeating until there is only one triangle left. This way the ear clipping triangulation is done.

(Section 2.1)

Algorithm Using Ear Clipping Method of Triangulation to find Guards:

Step-1: Create a Polygon by importing the co-ordinates.

Step-2: Triangulate the polygon using tripy module (ear clipping algorithm)
(Triangles = list of triangles formed after triangulation)

Step-3: Create a list (lst) and store all the occurrence of each vertex in the triangulation.

Step-4: Create a list (Flst) for appending final guards' vertices to it.

Step-5: Find the most frequent vertex in the list (lst). (Vertex with most occurrences)

Step-6: Append the most frequent vertex of list (lst) to the list (Flst).

Step-7: Create another list (X) and append all those elements (triangles) in list (Triangles) to list (X), which do not contain the most frequent vertex that is present in the list (Flst).

Step-8: Triangles:= X, now the list triangles is updated. (Basically replacing all elements of lst (Triangles) by lst (X)).

Step-9: While X != [ ], continue the loop from step-3 with updated list (Triangles) and when X==[], Output the (Flst), stop.

The final list (Flst) gives us the exact number and position of the vertex guards to guard the polygon.

(Section 2.2)

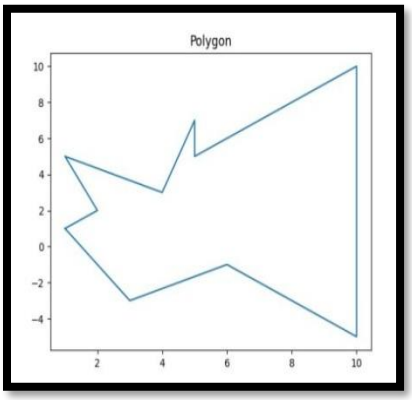
Implementation of Algorithm in Python:

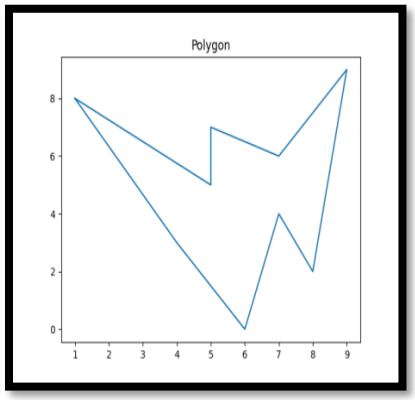
(Refer Appendix 1)

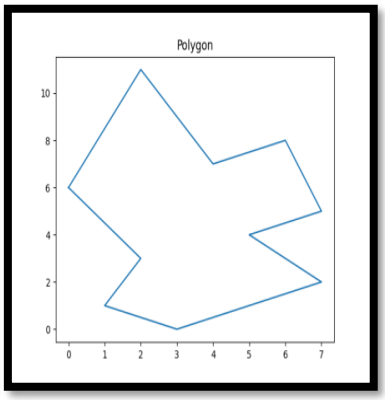
1) Creating a Polygon:

```
#Creating a polygon
import matplotlib.pyplot as plt
Dx = list()
Dy = list()
def Plot_Polygon(xlst,ylst):
    plt.title('Polygon')
    plt.plot(xlst,ylst)
    return  plt.show()
xlst = list()
ylst = list()
while  True:
    Vx  = input ("Enter the X vertices of the polygon:")
    if Vx == "done":
        break
    try:
        Vx = float(Vx)
    except:
        print("Invalid  Input")
        continue
    xlst.append(Vx)
    Vy = input ("Enter the Y vertices of the polygon:")
    if Vy == "done":
        break
    try:
        Vy = float(Vy)
    except:
        print("Invalid  Input")
        continue
    ylst.append(Vy)
Plot_Polygon(xlst,ylst)
```

Outputs:



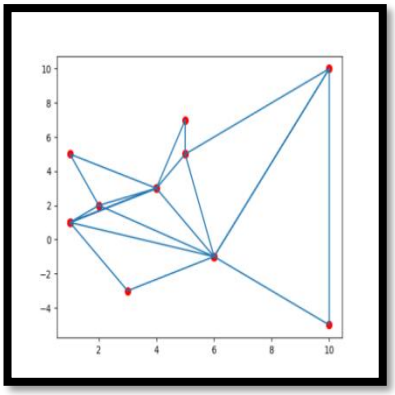
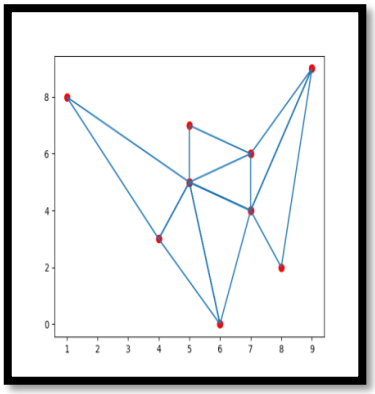
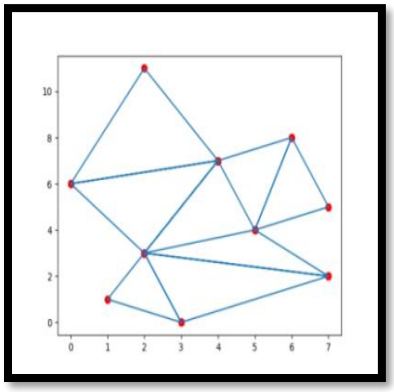




2) Triangulating a Polygon:

```
import matplotlib.pyplot as plt
import tripy
def triangulate_poly(polygon):
    import tripy
    triangles = tripy.earclip(polygon)
    plot_lstx = list()
    plot_lsty = list()
    for i in triangles:
        for j in i:
            plot_lstx.append(j[0])
            plot_lsty.append(j[1])
    plt.plot(plot_lstx,plot_lsty)
    plt.scatter(plot_lstx,plot_lsty,\
    s = 200, marker = '.',color = 'r')
    return triangles, plt.show()
polygon = list()
X = list()
Y = list()
while True:
    Vx = input("Enter the x coordinates:")
    if Vx == "done":break
    try: Vx = float(Vx)
    except: print("Invalid Input");continue
    X.append(Vx)
    Vy = input("Enter the y coordinates:")
    if Vy == "done":break
    try: Vy = float(Vy)
    except: print("Invalid Input");continue
    Y.append(Vy)
polygon = [(X[i],Y[i]) for i in range(0,len(X))]
triangulate_poly(polygon)
```

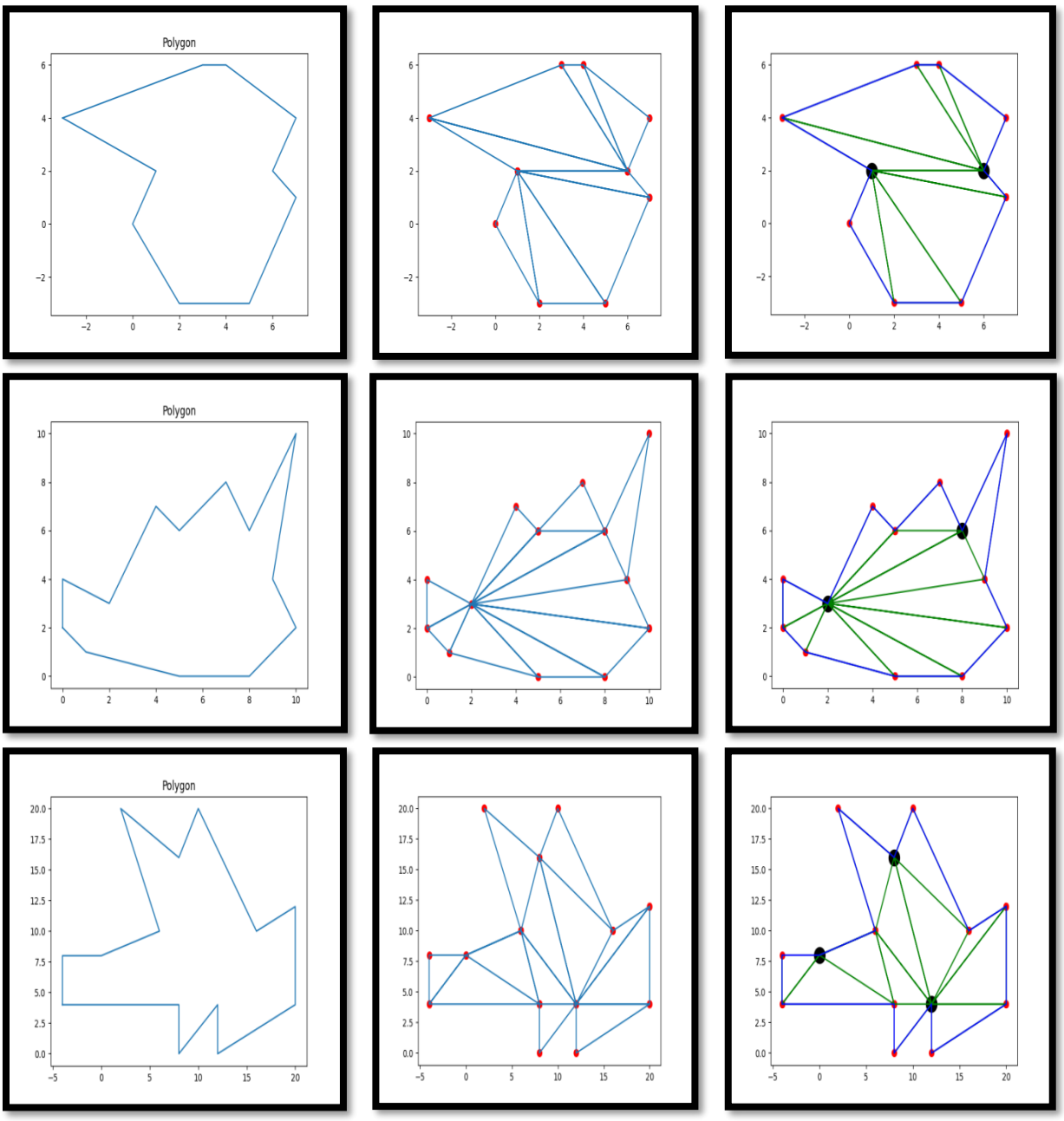
Outputs:

3) Final Guard Finding: (Combination of all the sub-code to find the final guards)

Outputs: (Implementation of the Algorithm in Python) (Refer Appendix 1)

Polygon	Polygon Triangulation	Final Guards Position

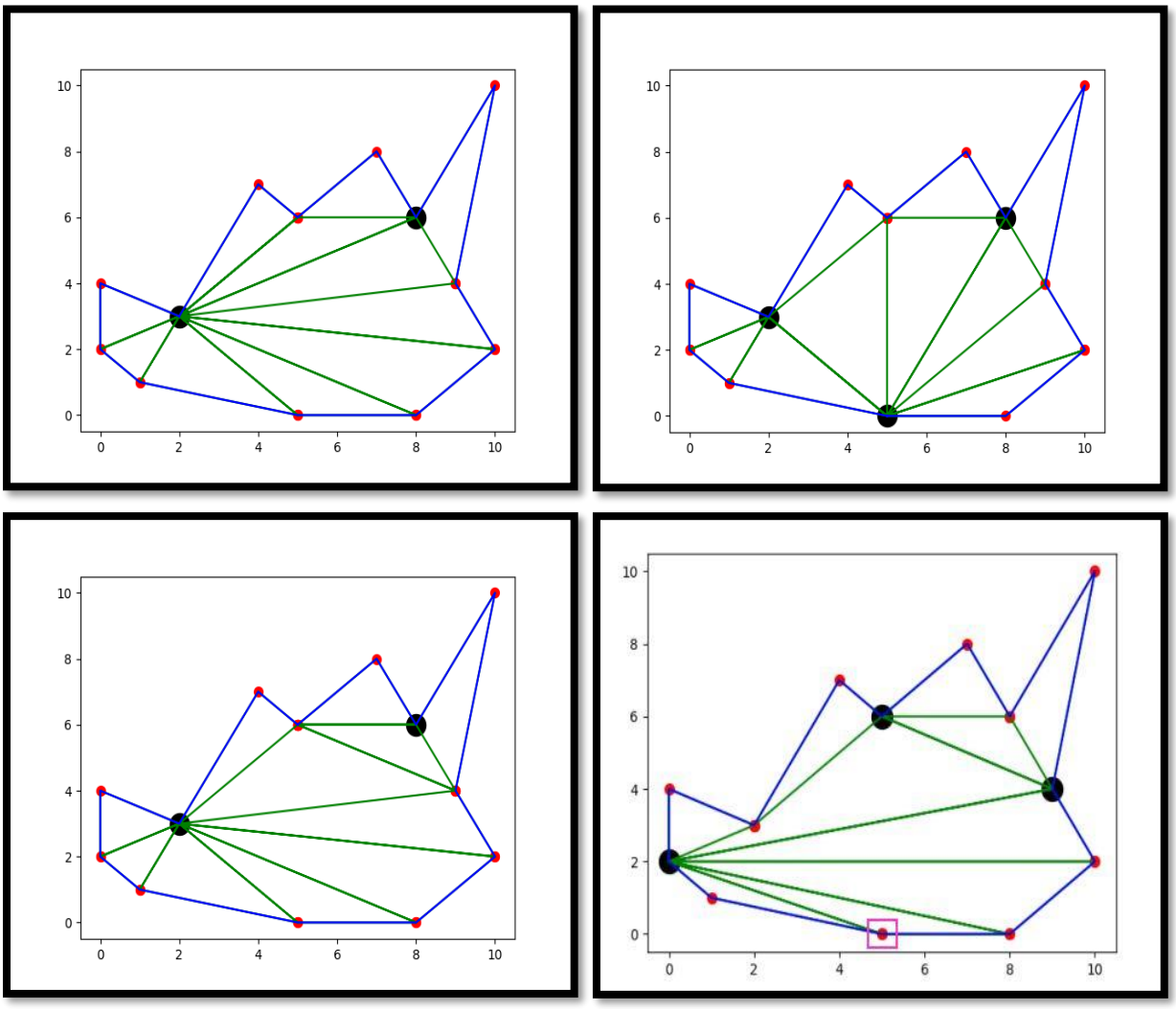


The above examples show the creation of the polygon, followed by the triangulation of the polygon, and then finally the position of the minimum number of guards required to cover the polygon.

(Section 2.3)

Limitation of the Algorithm using Ear-clipping Method:

The problem is that, if we change the order of inserting the data of co-ordinates, then the triangulation changes and hence the position and number of guards changes.



As we can see above that one polygon have four different solutions, which mean triangulation depends on the way we enter the co-ordinates. Therefore it is hard for us to predict which way is the best way to enter the co-ordinate and which point to start from.

Moreover, if we observe the highlighted point in pink box in the 4th figure under the varying solutions, we can manually find out that the point (5, 0) can completely guard the whole polygon and so only one guard is sufficient to guard it. Hence we can also conclude that the above algorithm fails to give a perfect solution. Therefore, I need to think of another and efficient way to approach the Art Gallery Problem.

So, one of the efficient ways to approach the art gallery problem is the diagonalisation of polygon instead of triangulation of polygon. Diagonalisation of a polygon method will be discusses in the next section.

(Section 3)

Diagonalisation:

In this method the main focus will be on covering the complete polygon by drawing diagonals throughout the polygon, wherever needed. I think this method will work because in this method I will try to find that point in the polygon from which maximum diagonals emerge and cover the area. And the area which is left uncovered will be covered by another such point. The advantage of this method is that, the frequency of varying solutions, in the previous method of triangulation, will be reduced and the algorithm will try to give the best solution with minimum number of guards.

(Section 3.1)

New Algorithm Using Diagonalisation Method (Vertex Cover):

Step-1: Create a polygon.

Step-2: Create a list (lst1) whose elements are the lists of all the lines drawn from each vertex of the polygon to other vertices.

Step-3: After getting all the line, create a list (lst2) whose elements are lists of diagonals in the polygon from each vertex. Sort the list (lst2) in descending order of length.

Condition for diagonal:
1) Diagonals are the lines which are completely inside the polygon.
2) Diagonals are lines that must not intersect with the sides of the polygon

Step-4: Now from the list (lst2), find the vertex which has maximum diagonals (which is present in the 0^{th} sub-list of (lst2), as list is sorted in descending order) and then append it to list (lst3)

Step-5: Now remove the sub-list of the diagonals of the vertex present in list (lst3); from the list (lst2) and update list (lst2)

Step-6: Find all those vertices in the polygon, which are not processed or used, and append them to list (F).

Step-7: Find sub-list in list (lst2) which contains maximum of the non used co-ordinate present in list (F). Then append the main vertex (vertex from which the diagonals are drawn) from that sub-list, to the list (lst3).

Step-8: Now remove all the processed vertices, if any, from list (F).

Step-9: If list (F) ! = [ ]: then go back to Step-5 and continue.

Step-10 When list (F) == [ ], stop and return list (lst3).

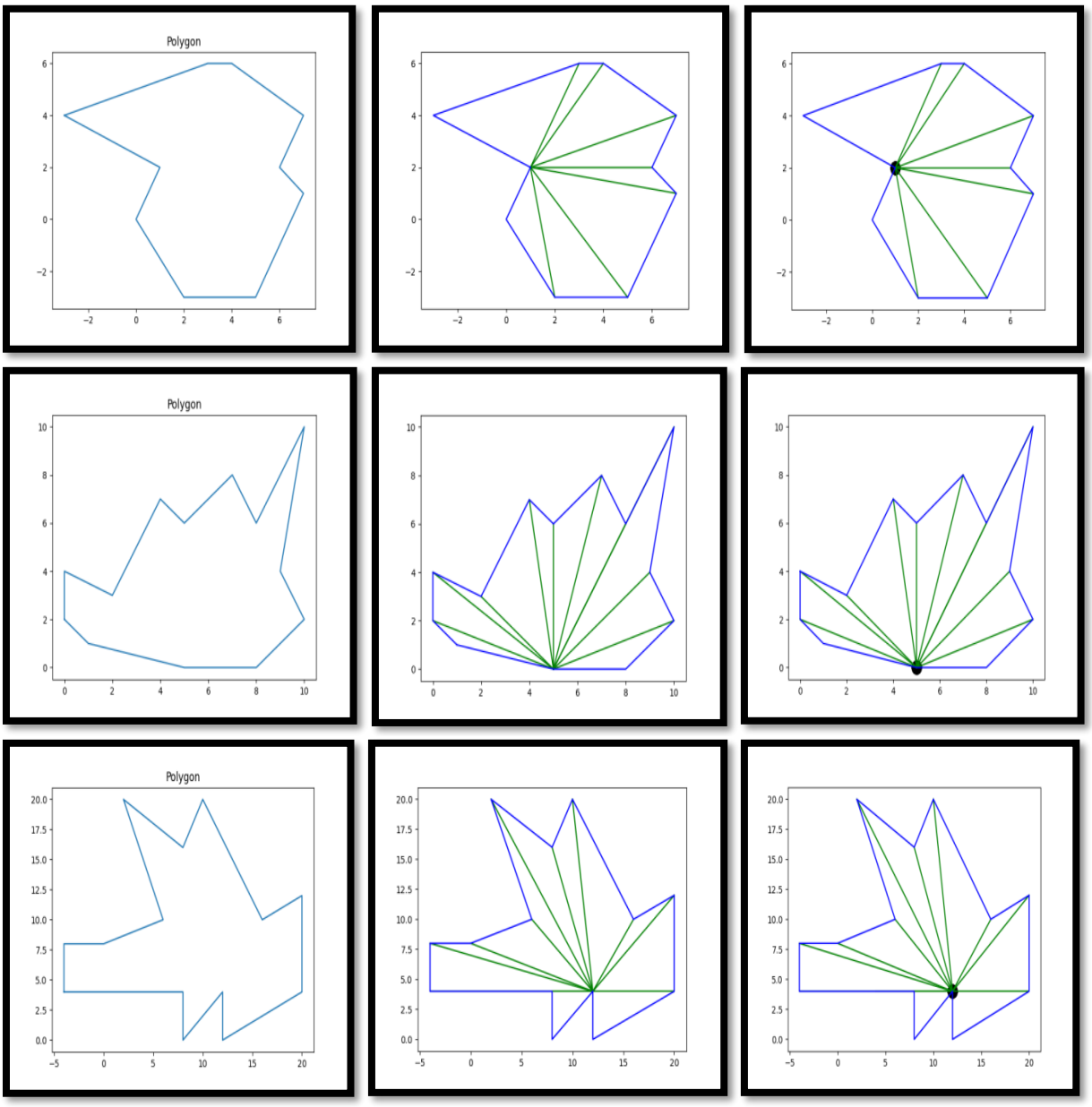
List (lst3) gives all the required vertex guards and their positions.
(Section 3.2)

Implementation of the New Algorithm Using Diagonalisation Method (Vertex Cover) in Python:

(Refer Appendix 2)

OUTPUTS: (Implementation of the Algorithm in Python)

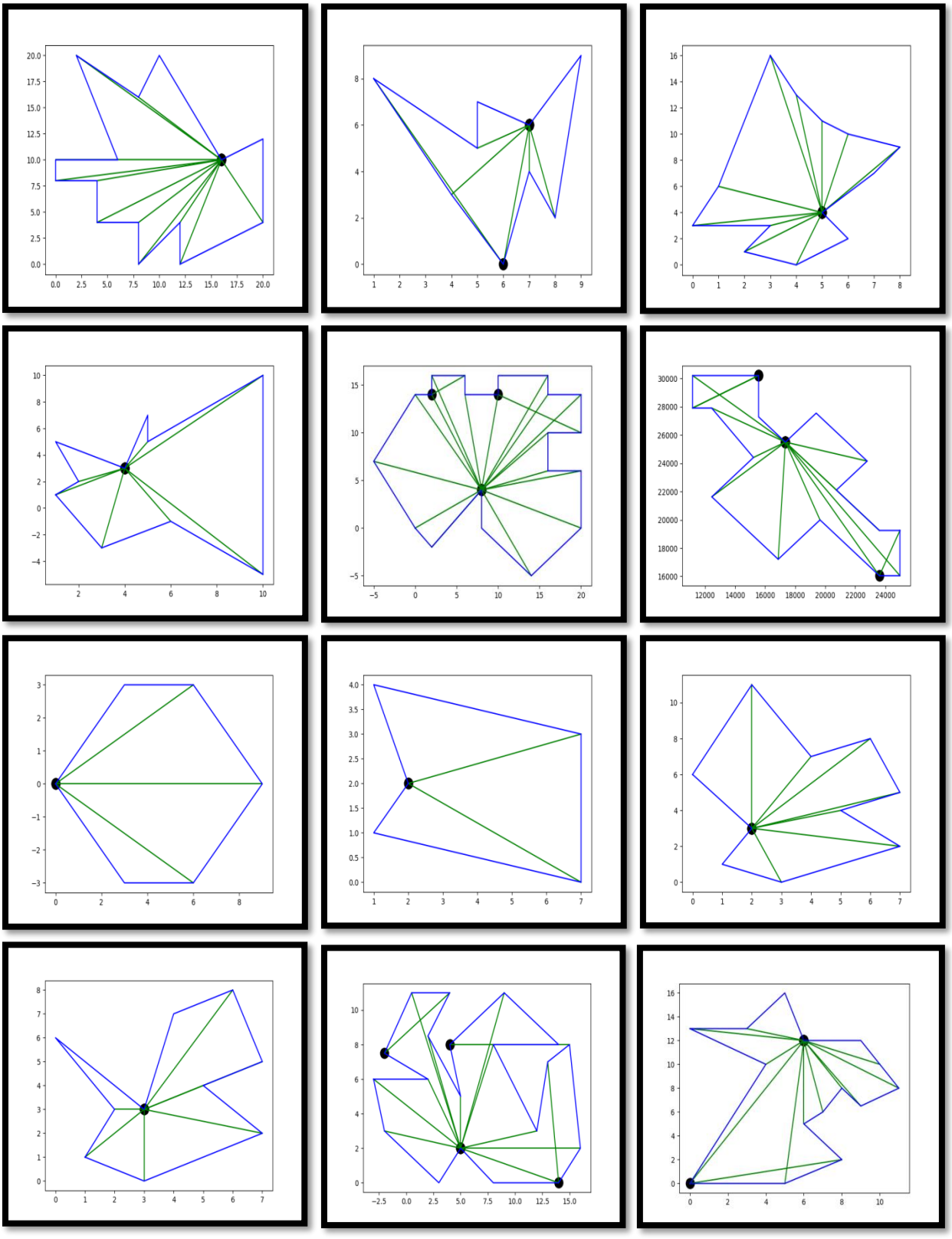
Polygon	Polygon Diagonalisation	Final Guards Position



The above examples show the creation of the polygon, followed by the diagonalisation of the polygon, and then finally the position of the minimum number of guards required to guard the polygon.

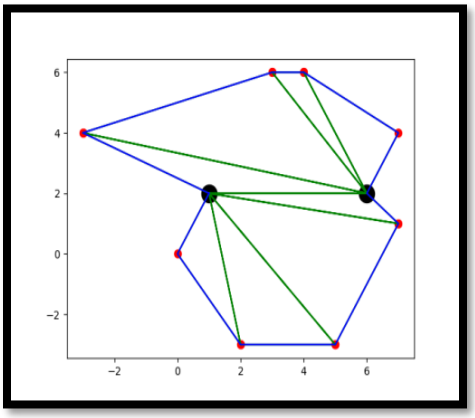
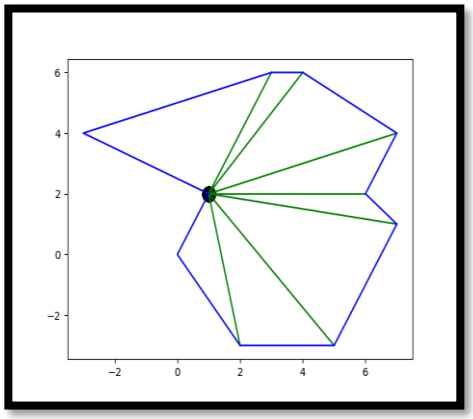
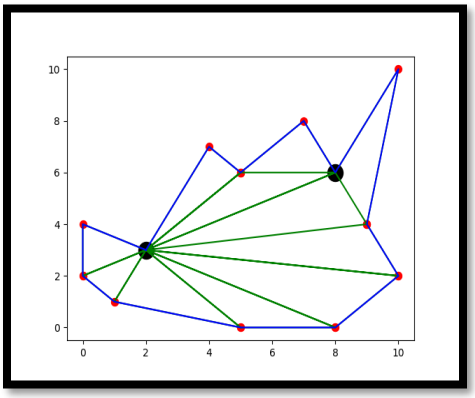
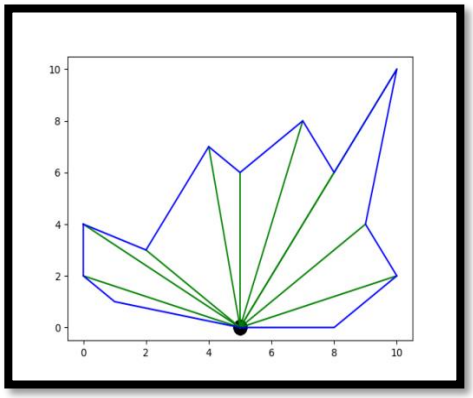
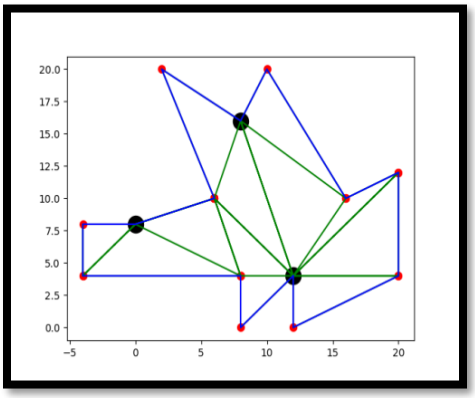
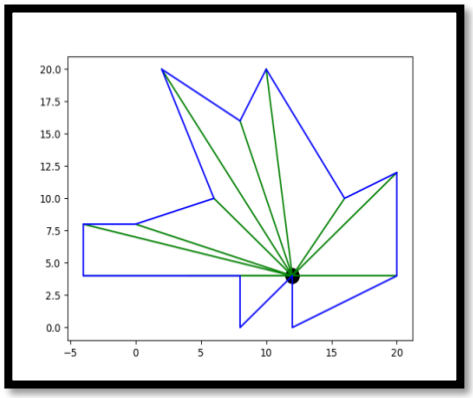
(Section 3.3)

Some More Outputs of the above New-Algorithm using Diagonalisation (Vertex Cover):



(Section 4)

Comparison of two Algorithms:

Algorithm using Ear-clipping Triangulation Method	New Algorithm using Diagonalisation Method
 2 Guards	 1 Guard
 2 Guards	 1 Guard
 3 Guards	 1 Guard

From the above comparison it is clear that the diagonalisation method is better than the triangulation method and also cost effective as the number of guards required in the diagonalisation method is less as compared to that of the triangulation method.

(Section 5)

Shrink Polygon:

A polygon, whose scale factor reduces from the original is called shrink polygon. As the guards are 360 degree cameras, it is better to place the guards on the shrink polygon so that we can make better use of them.
One step that I had to add in the new algorithm was to shrink the polygon and use the co-ordinates of shrink polygon to cover the actual polygon.

(Section 5.1)

New Algorithm using Diagonalisation Method (Shrink Polygon – Vertex Cover)

Step-1: Create a polygon.

Step-2: Shrink the polygon. **(The change)**

Step-3: Create a list (lst1) whose elements are the lists of all the lines drawn from each vertex of the shrink polygon to the vertices of the actual polygon. **(The change)**

Step-4: After getting all the line, create a list (lst2) whose elements are lists of diagonals in the polygon from each vertex of the shrink polygon. Sort the list (lst2) in descending order of length.

Condition for diagonal:

1) Diagonals are lines which are completely inside the polygon.
2) Diagonals are lines that must not intersect with the sides of the polygon.

Step-5: Now from the list (lst2), find the vertex of shrink polygon which has maximum diagonals (which is present in the 0^{th} sub-list of (lst2), as list is sorted in descending order) and append it to list (lst3)

Step-6: Now remove the sub-list of the diagonals of the vertex of shrink polygon present in list (lst3), from the list (lst2) and update list (lst2)

Step-7: Find all those vertices in the polygon, which are not processed or used, and append them to list (F).

Step-8: Find the sub-list in list (lst2) which contain maximum of the non used co-ordinate present in list (F). Then append the main vertex (vertex of the shrink polygon from which the diagonals are drawn) from that sub-list, to the list (lst3).

Step-9: Now remove all the processed vertices, if any, from list (F).

Step-10: If list (F) ! = [ ]: then go back to Step-6 and continue.

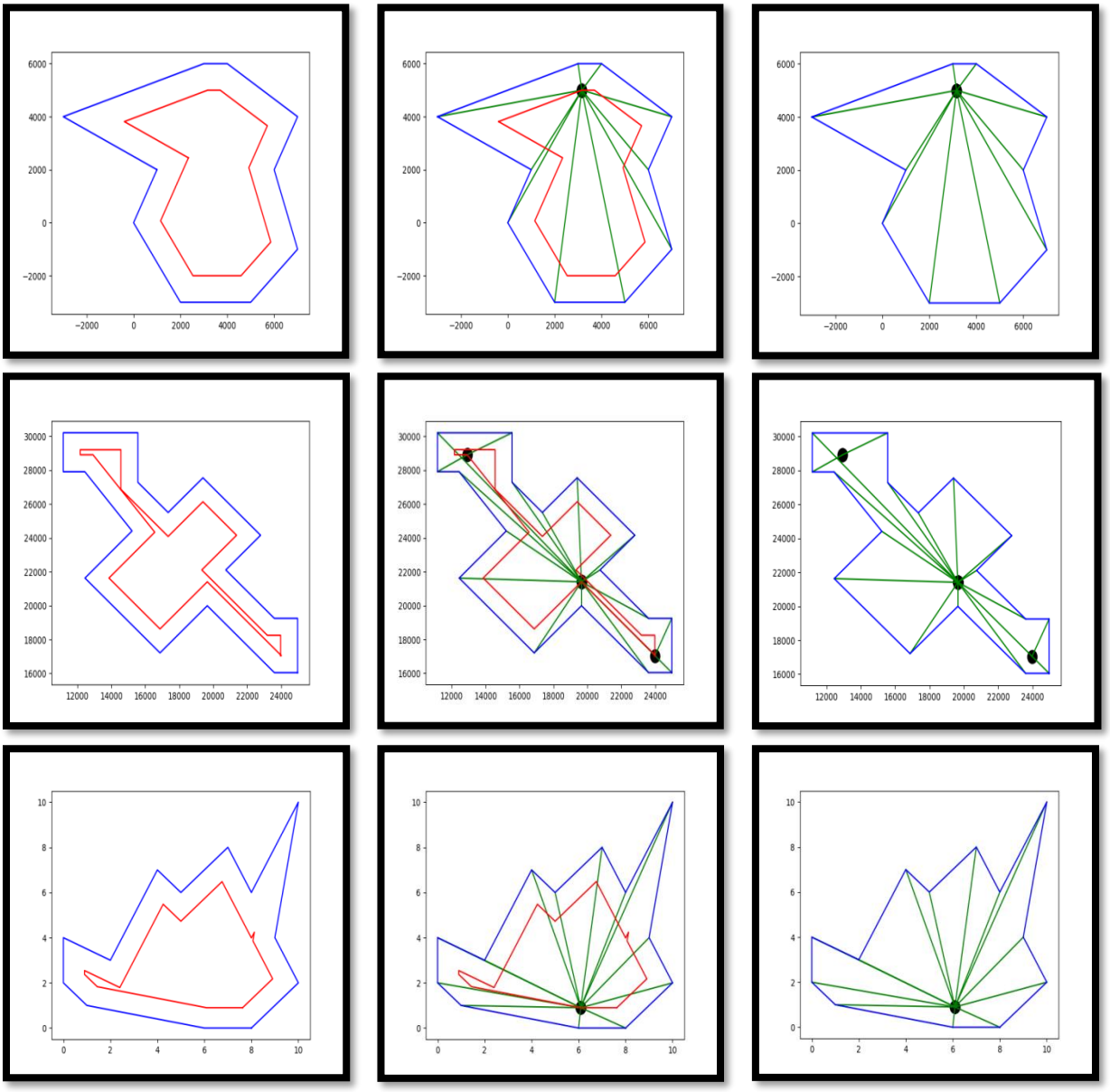
Step-11: When list (F) == [ ], stop and return list (lst3).
The lst3 gives the exact number of the guards required to guard the polygon.
(Section 5.2)

Implementation of the New Algorithm using Diagonalisation Method (Shrink Polygon – Vertex Cover)

(Refer Appendix 3)

Outputs: (Implementing the Algorithm in Python)

Shrink Polygon	Actual Polygon Covering	Final Result

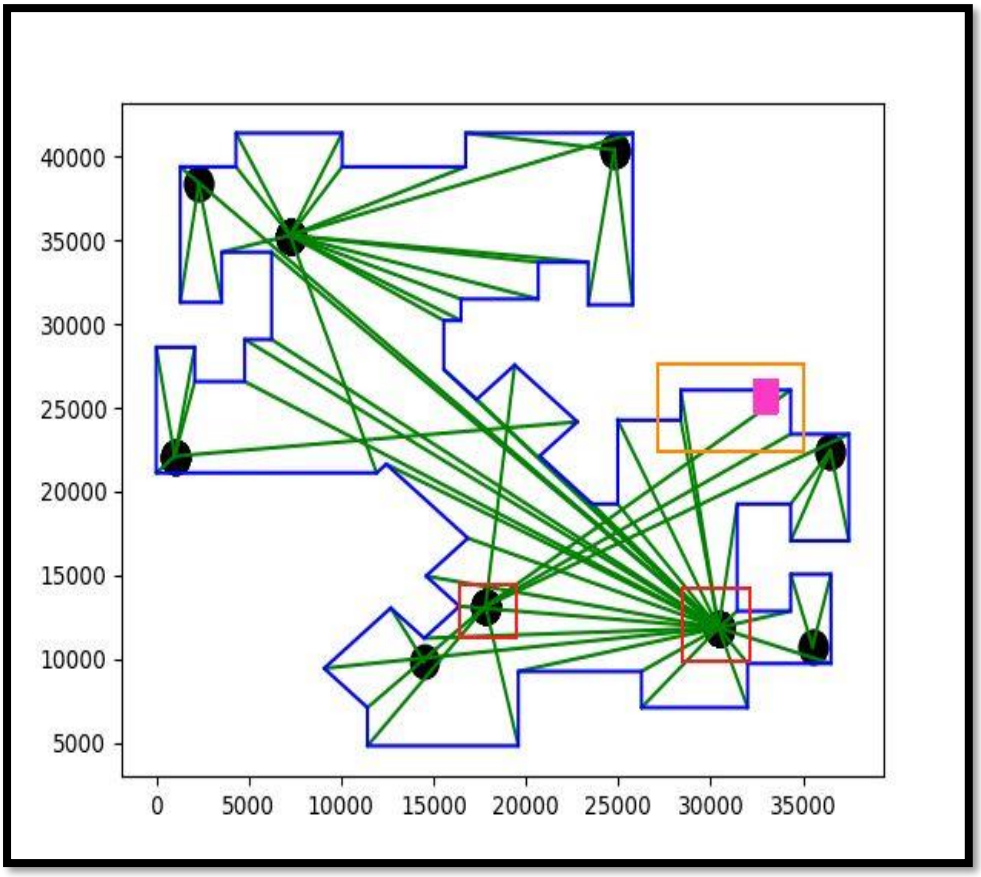


The above examples show the creation of the actual polygon (blue) and the shrink polygon (red), followed by the diagonalisation of the actual polygon, vertex cover, from the vertices of the shrink polygon, and then finally the position of the minimum guards (black) required to guard the polygon.

(Section 5.3)

Limitations of the above New Algorithm:

As I tested many polygons with the above Algorithm, I found that some regions in the polygon were partially covered as seen below. The edge (highlighted in orange box), of the polygon shows us the partial coverage. We can see that the highlighted edge is guarded by two different guards in the polygon (highlighted in red box). Those two guards fail to cover the complete edge, there is some region on the edge left unguarded (highlighted in pink).



To overcome this problem, I decided to cover each edge, instead of covering the vertex, to guard the polygon.

(Section 6)

Edge Cover

Till now I was focusing on vertex cover using diagonalisation method, but as discussed above some area remains uncovered in the polygon. Hence, to overcome this issue I have introduced another concept of edge cover. In edge cover, instead of covering the vertices of the polygon, the edges of the polygon are covered. Due to edge cover, no area in the polygon is left uncovered.

(Section 6.1)

New Algorithm Using Diagonalisation Method (Edge Cover):

Step-1: Create a polygon.

Step-2: Shrink the polygon.

Step-3: Create pair of vertices of actual polygon, to form edges. Append those edges to list (Pb).

Step-4: Create a list (lst1) whose elements are the lists of all the lines drawn from each vertex of the shrink polygon to the vertices of the actual polygon.

Step-5: After getting all the line, create a list (lst2) whose elements are lists of diagonals in the polygon from each vertex of shrink polygon. Sort the list (lst2) in descending order of length.
Condition for diagonal:

1) Diagonals are lines which are completely inside the polygon.
2) Diagonals are lines that must not intersect with the sides of the polygon.

Step-6: Using the sub-lists of list (lst2) make pairs of two adjacent diagonals (diagonals whose one vertex, present on shrink polygon, is same and the other vertices, present on actual polygon, are adjacent to each other on the polygon).

Step-7: Make a sub-list of the list of pairs of two adjacent diagonals which have same first vertex present on the shrink polygon. Append that sub-list to a list (lst3). Sort the list (lst3) in descending order of length.

Step-8: Now from the list (lst3), find the vertex which has maximum pair of two adjacent diagonals (which is in the 0^{th} sub-list of (lst3) as list is sorted in descending order) and append it to list (lst4)

Step-9: Now remove the sub-list, of the pair of diagonals of vertex present in list (lst4), from the list (lst3) and update list (lst3).

Step-10: Find all those edges in list (Pb) or in the polygon, which are not processed or used, and append them to list (F).

Step-11: Find the sub-list in list (lst3), which contains maximum pairs of two adjacent diagonals, to cover these non used edges present in list (F). Then append the main vertex (vertex from which pair of two adjacent diagonals are drawn) from that sub-list, to the list (lst4).

Step-12: Now remove all the processed edges from list (F).

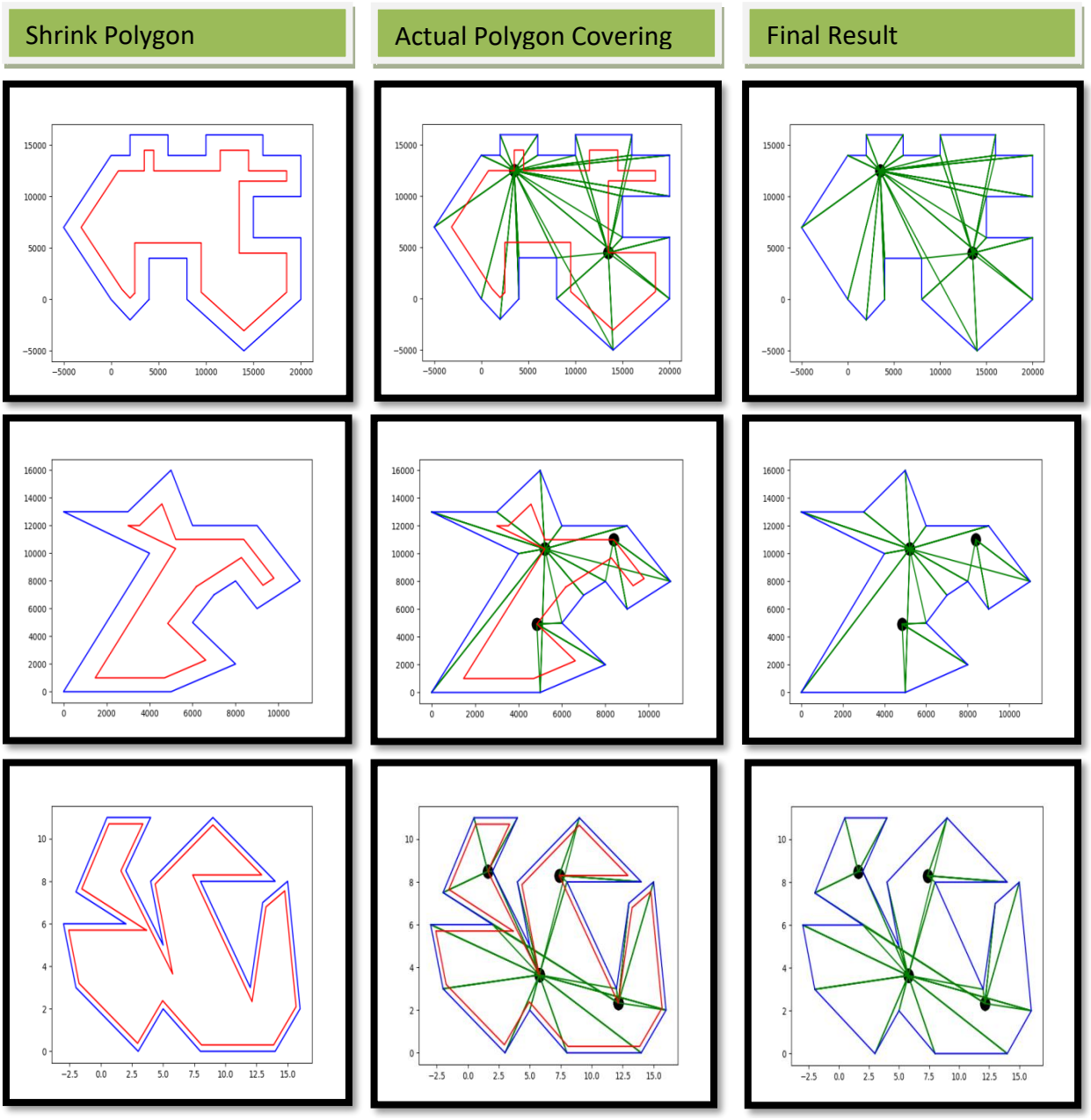
Step-13: If list (F) ! = [ ]: then go back to Step-9 and continue.
Step-14: When list (F) == [ ], then stop and return lst4.
The lst4 gives the exact number of the guards and their position, required to guard the polygon.

(Section 6.2)

Implementation of the New Algorithm using Diagonalisation Method (Edge Cover)

(Refer Appendix 4)

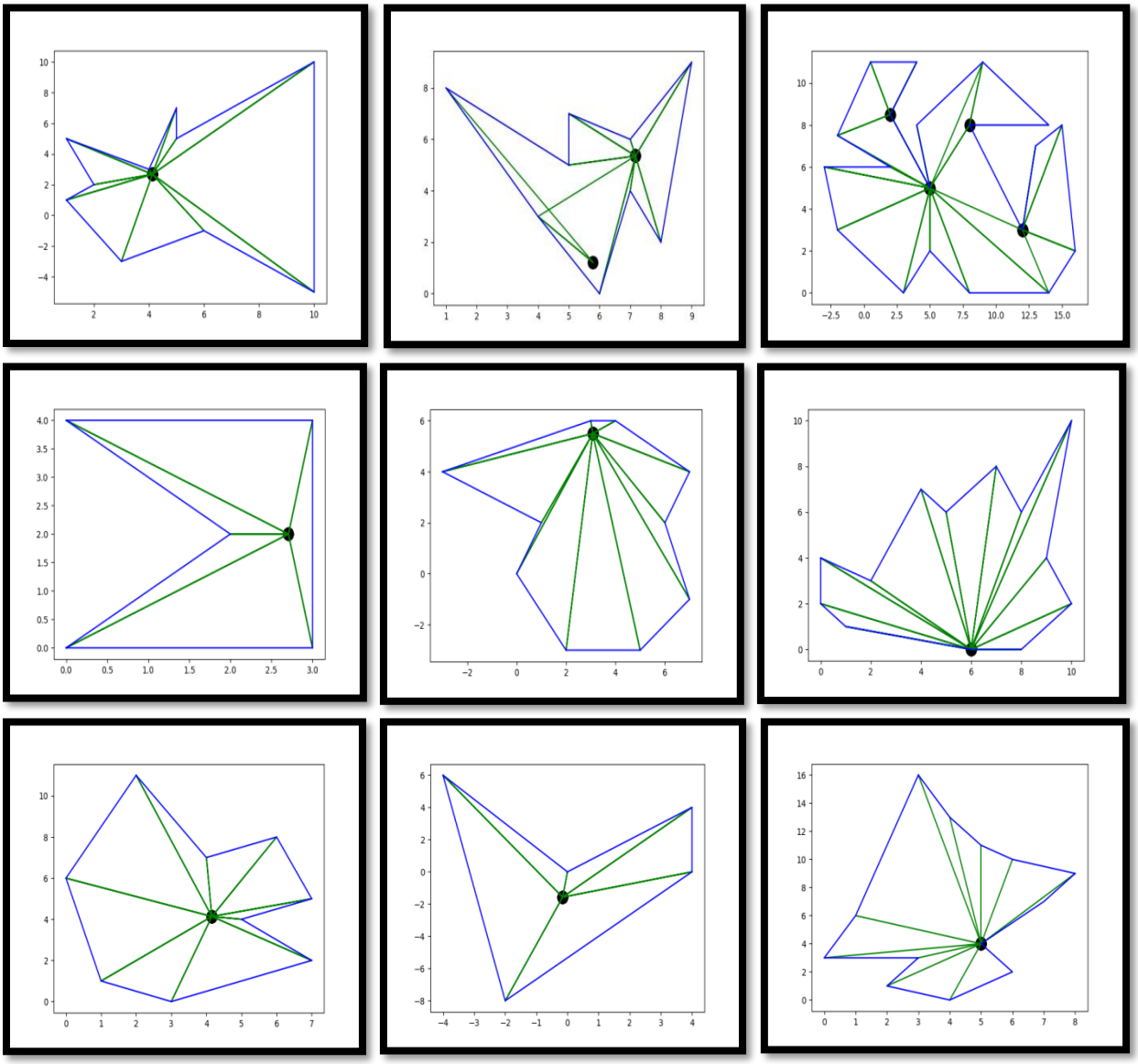
Outputs: (Implementing the Algorithm in Python)



The above examples show the creation of the actual polygon (blue) and the shrink polygon (red), followed by the diagonalisation of the actual polygon, edge cover, from the vertices of the shrink polygon, and then finally the position of the minimum guards (black) required to guard the polygon.

(Section 6.3)

Some more outputs of the New Algorithm using Diagonalisation (Edge Cover - with or without shrink polygon)



(Section 6.4)

Limitation of Shrink Polygon:

One limitation of the shrink polygon is that, when we go on shrinking the polygon, there will be a situation when the edges of the shrink polygon will start overlapping and the code will give an error. It is always better to get maximum shrinkage, but at the same time overlapping must also be avoided. Hence, this problem can be solved by introducing the concept of the Medial axis and the Voronoi diagram. The Medial Axis of a polygon is similar to the skeleton of a human body. In the next section of optimization, dual tree; Delaunay triangulation; Voronoi diagram; and Medial axis will be discussed in detail.

(Section 7)

Optimization:

One of the main aims during the optimization of the problem was to create a centerline of a polygon or a medial axis of a polygon, which is similar to the skeleton of a human body. After creating the medial axis the guards were supposed to be placed on it, to check if the number of guards reduces and to check if the scan quality improves.

There is a relation between the Voronoi diagram, Delaunay triangulation, and Medial axis. Hence, to find and understand the medial axis I studied the Voronoi Diagram and the Delaunay Triangulation. But before moving to the Voronoi Diagram and Delaunay triangulation, I tried and implemented the concept of Dual Tree, which in some cases is similar to the medial axis.

(Section 7.1)

Dual Tree: Given a triangulated simple polygon, the dual-tree is the graph generated by plotting a vertex at each triangle and edges joining vertices in adjacent triangles (triangles which share a diagonal).

(Section 7.1.1)

Basic Method to draw a Dual Tree or Dual of a Graph:

(G = Planar Graph; G* = Dual of a Graph; Fi = triangles in Graph; Pi = points in Graph)

1) Inside each triangle Fi of graph G, we take a point Pi. These points Pi are the vertices of the G*.

2) We join these points Pi as follows:

    - If two regions Fi and Fj are adjacent, draw a line joining points Pi and Pj that crosses the common edge between Fi and Fj exactly one.

    - If there is more than one edge common between Fi and Fj, draw one line between Pi and Pj for each of the common edge.

3) For an edge e lying entirely in one region, say Fk, draw a self loop at point Pk of Fk intersecting e exactly once.

The graph G* obtained by this procedure is called dual of the given graph or the dual tree.

Below, in the next section, we will see the implementation of the dual tree or dual graph with the diagonalisation method (Edge Cover).

(Section 7.1.2)

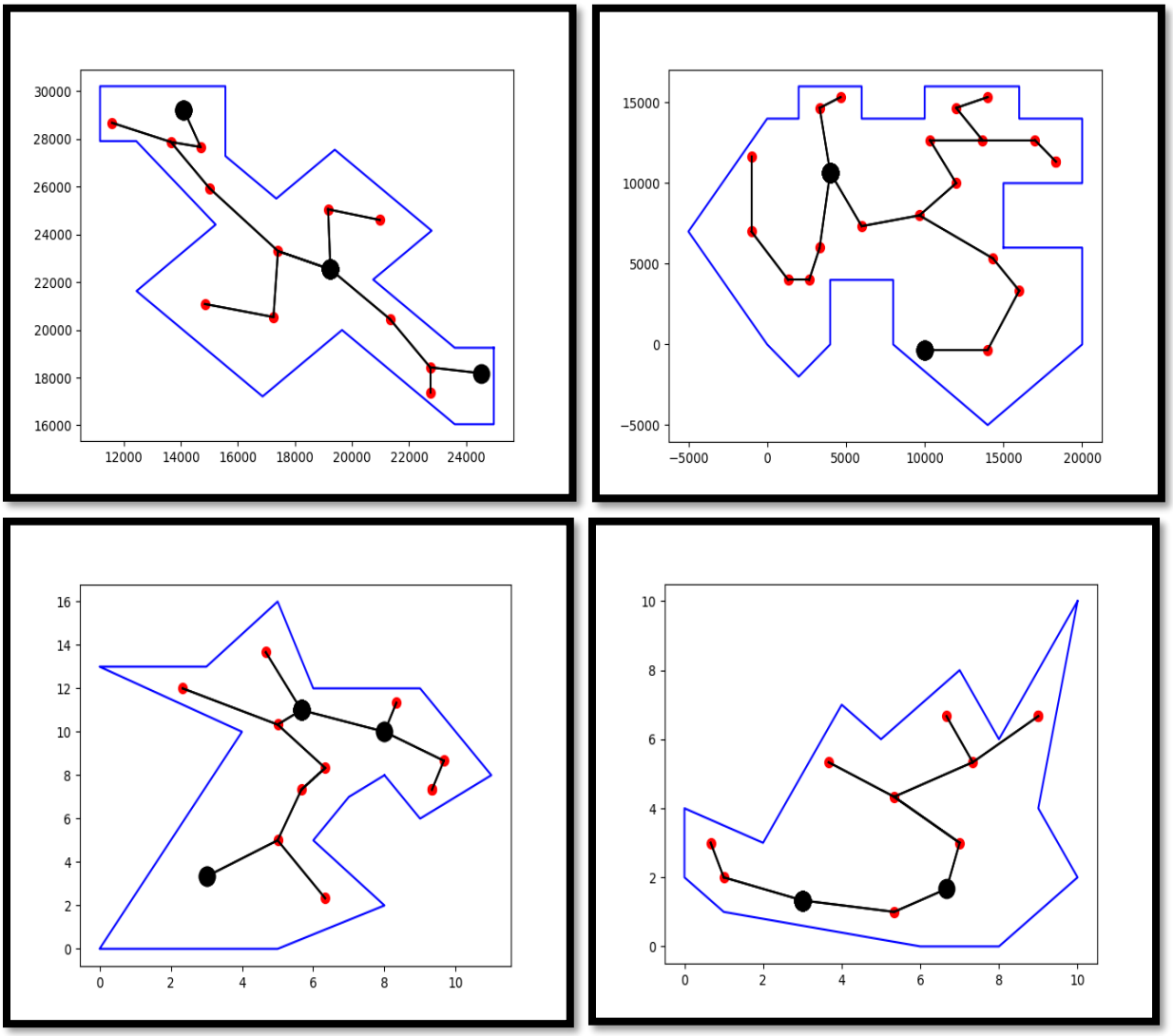
Implementation of Dual Tree in Python:

(Refer Appendix 5)

For the implementation of dual graph with the New Algorithm (given in the October report) in python, I had to make a change in the new algorithm, which was to create a dual tree according to the above procedure and then draw diagonals from the vertices of the dual tree to the vertices of the actual polygon.

(In the below examples I haven't shown the diagonals, to prevent a mess in the solutions)

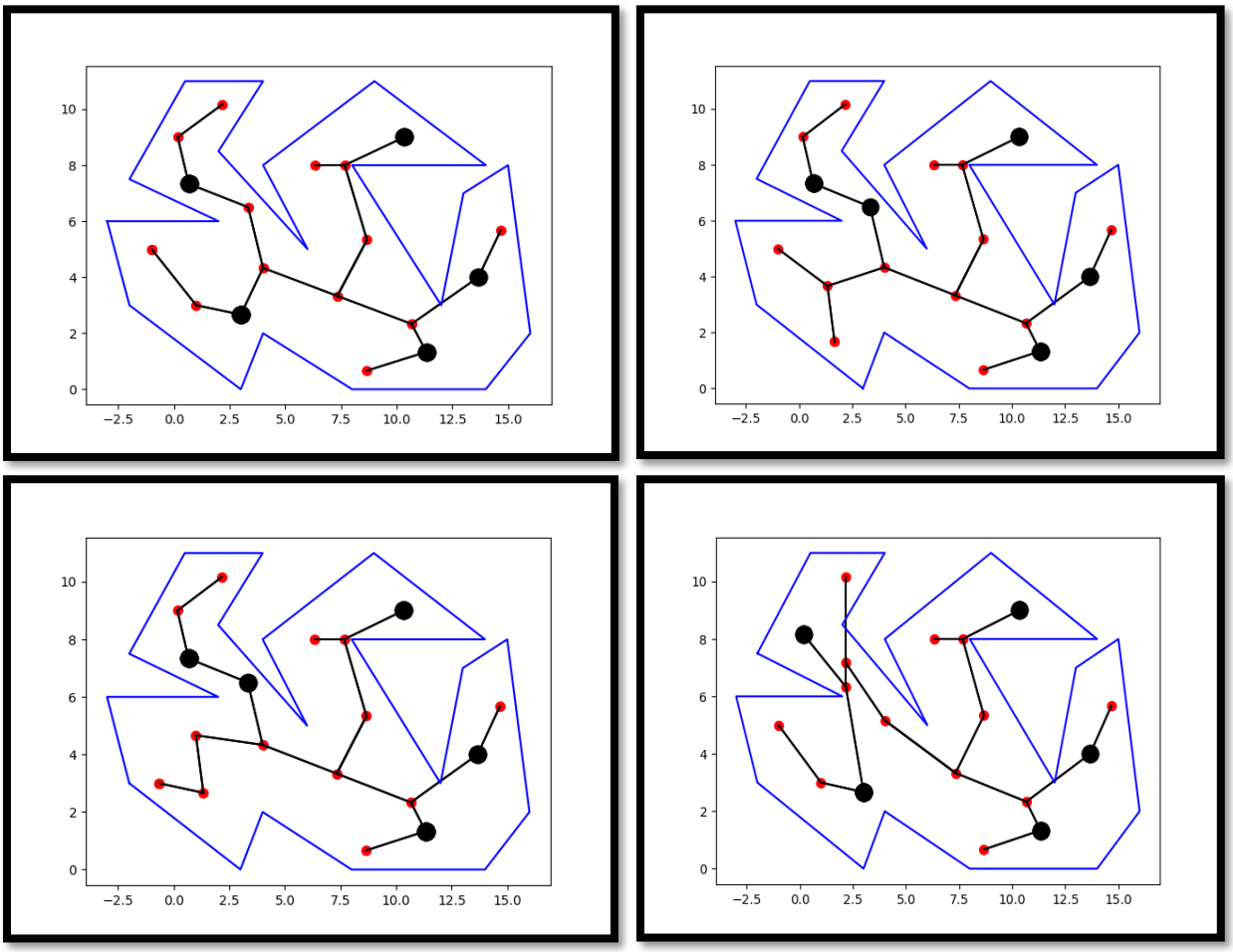
Polygon = Blue Edges;
Dual Tree = Black Edges;
Vertices of Dual Tree = Red Circles;
Final Minimum Required Guards = Black Circles



(Section 7.1.3)

Limitation and Varying Solution of the Dual Tree:

The dual tree varies a lot, when the order of the input co-ordinates changes. As the order of input changes the triangulation changes, which further changes the dual tree and the solution of the problem. Hence I believe that it is not reliable.



(Section 7.2)

Delaunay Triangulation: A Delaunay triangulation of a vertex set (polygon vertices) is a triangulation of the vertex set with the property that no vertex in the vertex set falls in the interior of the circumcircle (circle that passes through all three vertices) of any triangle in the triangulation.

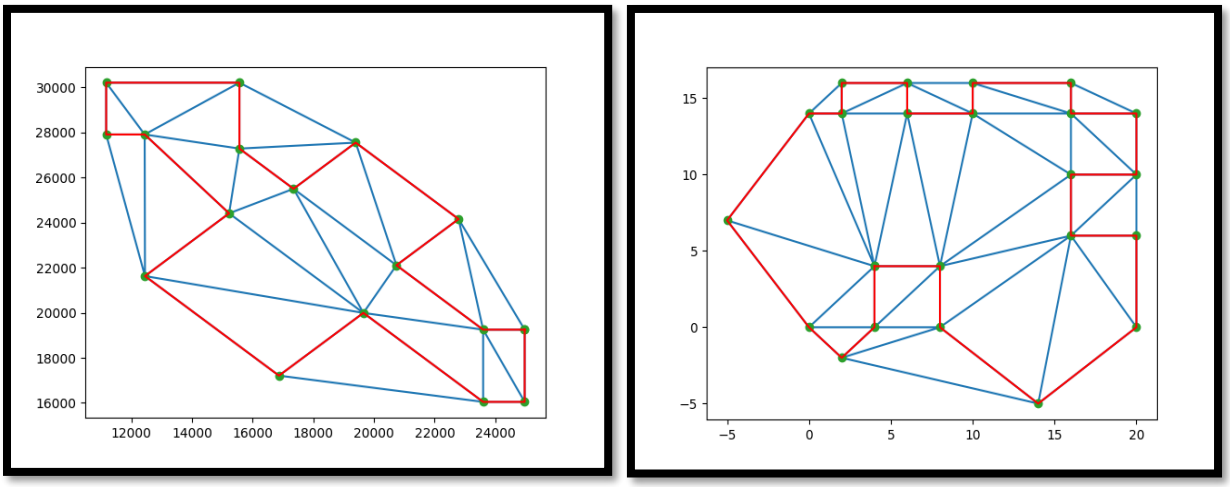
(Section 7.2.1)

Basic Method to draw a Delaunay triangulation: To find the Delaunay triangulation, just join the points in vertex set such that they form a triangle whose circumcircle does not contain any other point of the vertex set.

(Section 7.2.2)

Implementation of Delaunay Triangulation in Python:

(Delaunay triangles: Blue; Polygon: Red)



(Section 7.3)

Voronoi Diagram: A Voronoi Diagram of a vertex set is a subdivision of the plane into polygonal regions (some of which may be infinite), where each region is the set of point in the plane that are closer to some input vertex than to any other input vertex.

(Section 7.3.1)

Basic Method to draw a Voronoi Diagram:

1) Input Sites (Vertices of the Polygon)
2) Connect Nearest Neighbors
3) Find the Midpoints of these neighbors
4) Draw a perpendicular Bisector
5) Trace cells by joining the perpendicular bisectors with each other

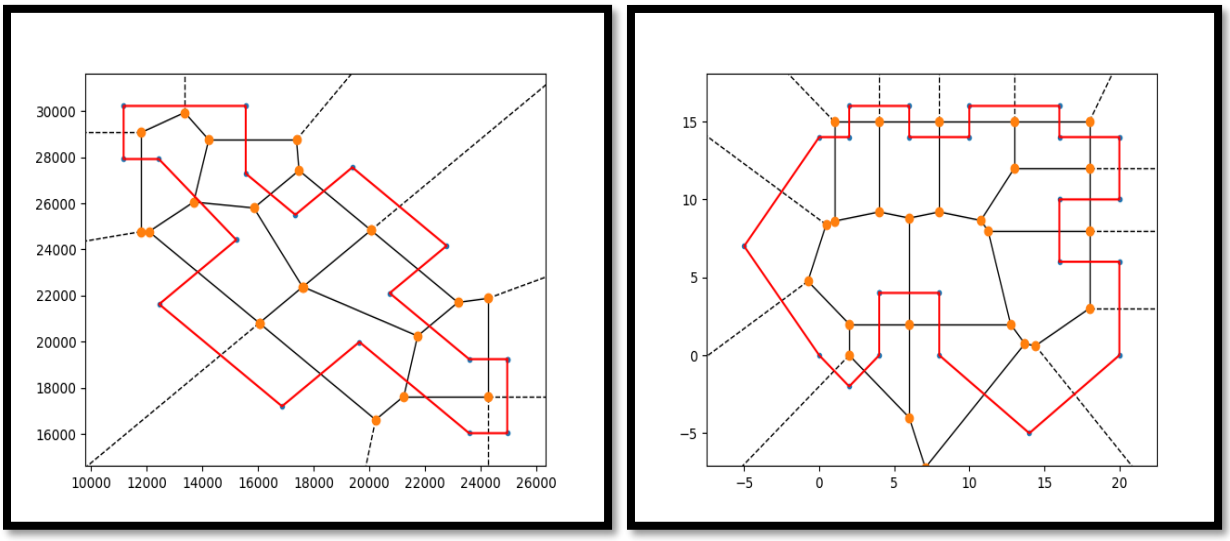
Note:

- Voronoi Diagram is the dual of the Delaunay Triangulation.
- The Circumcenters of the Delaunay Triangles are the vertices of the Voronoi Diagram. Hence using the Voronoi Vertices as guards to guard the polygon is the same thing as using the circumcenters of Delaunay Triangles as guards to guard the polygon.

(Section 7.3.2)

Implementation of Voronoi Diagram in Python:

(Voronoi Diagram: Black; Voronoi Vertices: Orange; Polygon: Red)

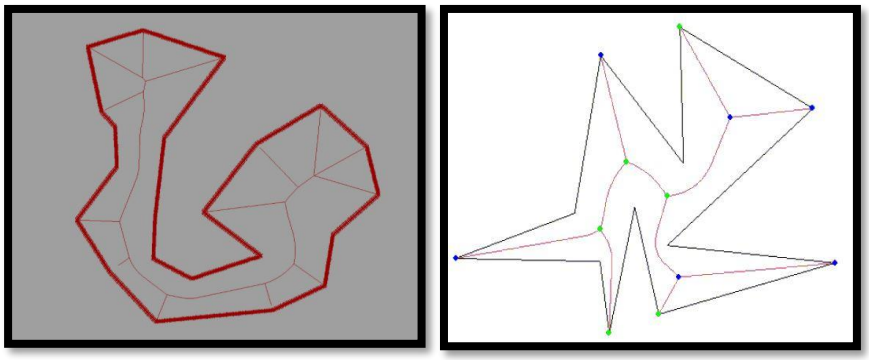


(Section 7.4)

Medial Axis:

A Medial Axis is defined as the collection of points within the polygon that are closest to more than one of the edges. It can also be viewed (equivalently) as the points that can be the center of a circle that is entirely within the polygon and touches the polygon in atleast two places.
The medial axis transform is a complete shape descriptor, meaning that it can be used to reconstruct the shape of the original domain.

To a computational geometer**, the medial axis of an n-gon is a Voronoi Diagram, whose sites are the open edges and the vertices of the boundary.**
The medial axis is a subset of the Voronoi diagram of the edges and vertices of the polygon. Voronoi edges that meet the reflex vertices are not part of the medial axis.
Hence, we need to concentrate on computing the Voronoi diagram and then obtain the medial axis by removing these Voronoi edges that meet reflex vertices.



So Voronoi diagram of the vertices of the polygon, inside the polygon, is the Medial Axis and the Voronoi Diagram is the Dual of the Delaunay Triangulation. Now the relation between the Delaunay Triangulation; Voronoi Diagram; and Medial Axis is clear. So let's implement them.

(Section 7.5)

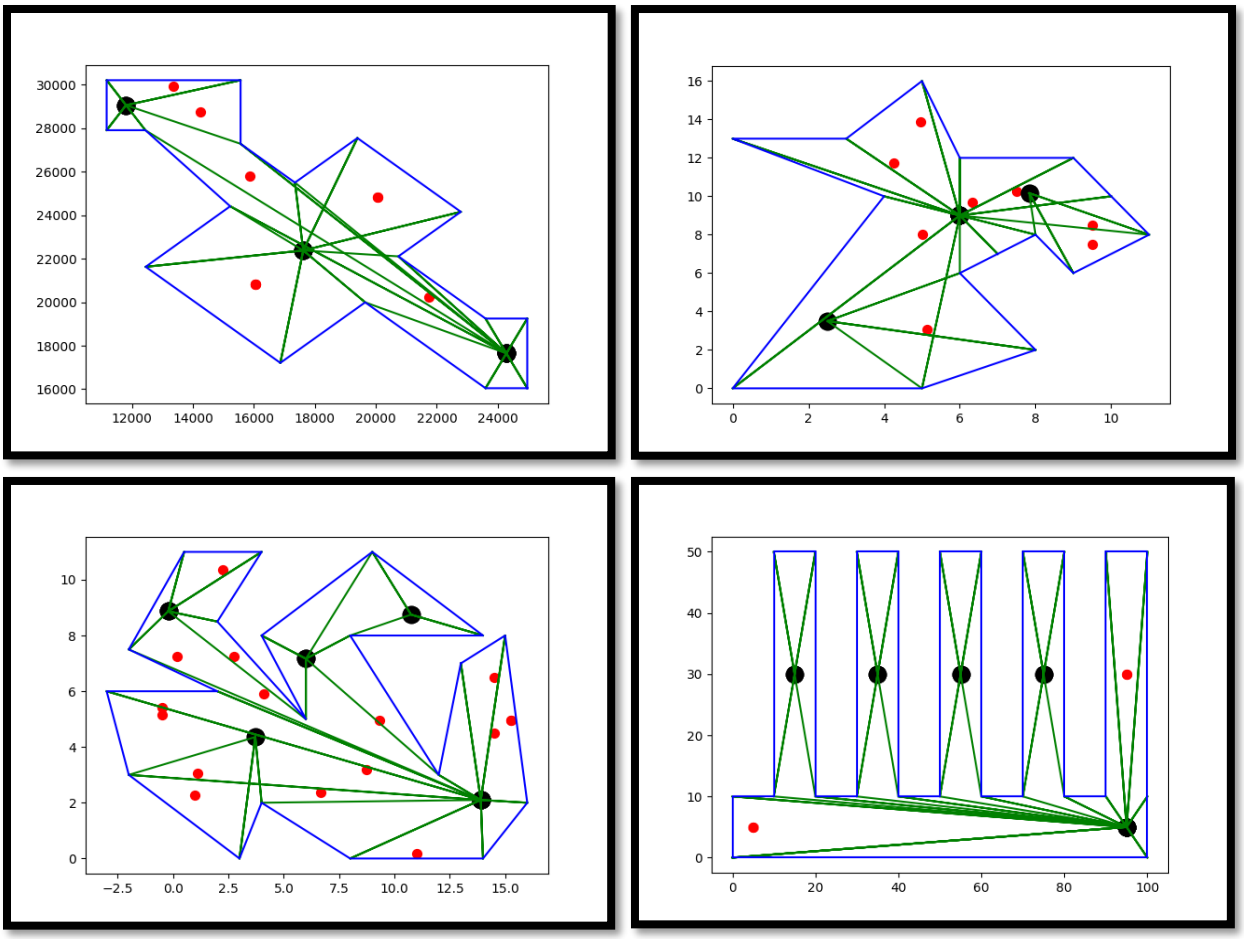
Implementation of Delaunay Triangulation with New Algorithm (Diagonalisation – Edge Cover) in Python:

(Refer Appendix 6)

The change that I made in the New Algorithm was to create a Delaunay Triangulation of the polygon vertices and then found the circumcenters of those triangles; and then finally drew the diagonals from the circumcenters to the vertices of the actual polygon.

(To avoid mess in the solutions Delaunay triangulations are not shown, but their circumcenters are shown)

Polygon: Blue; Diagonals: Green lines
Delaunay Triangle Circumcenters: Red Circles
Final Minimum Required Guards = Black Circles



So, in the above figures the final guards (Black) are placed on the circumcenters (Red) of the Delaunay triangles, to guard the polygon (Blue).

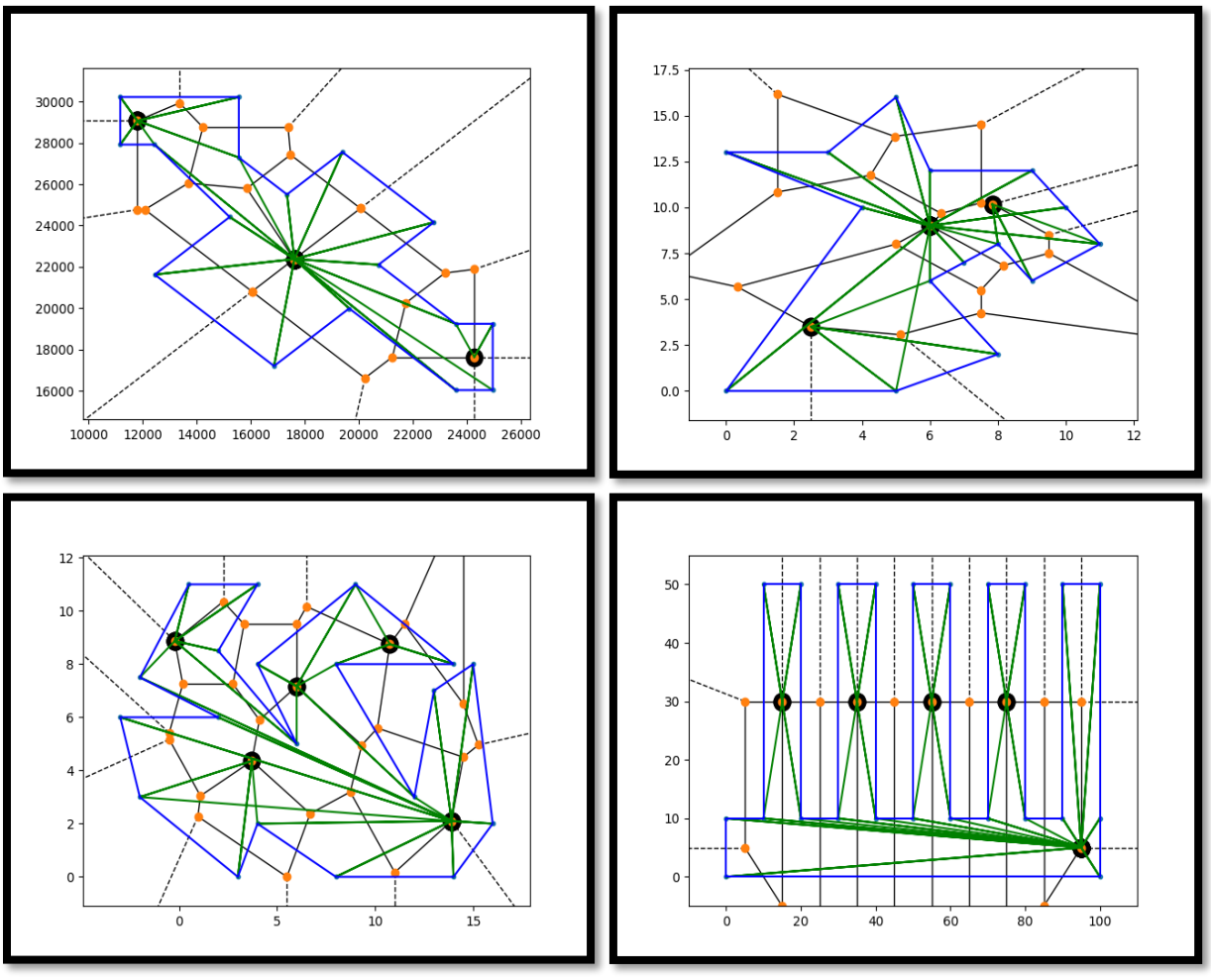
(Section 7.6)

Implementation of Voronoi Diagram with New Algorithm (Diagonalisation Edge Cover) in Python:

(Refer Appendix 7)

The change that I made in the New Algorithm was to create a Voronoi Diagram of the polygon vertices and then drew the diagonals from the vertices of the Voronoi diagram to the actual polygon.

Polygon: Blue; Diagonals: Green lines
Voronoi Diagram: Light Black Lines
Voronoi Vertices: Orange Circles
Final Minimum Required Guards = Black Circles



If we carefully see the solution of Delaunay triangulation and Voronoi diagram then, we notice that the position of the guards (black) is almost the same and also the total number of guards is same. Hence, we get the same solution from both irrespective to the position of the diagonals (green).

(Section 8)

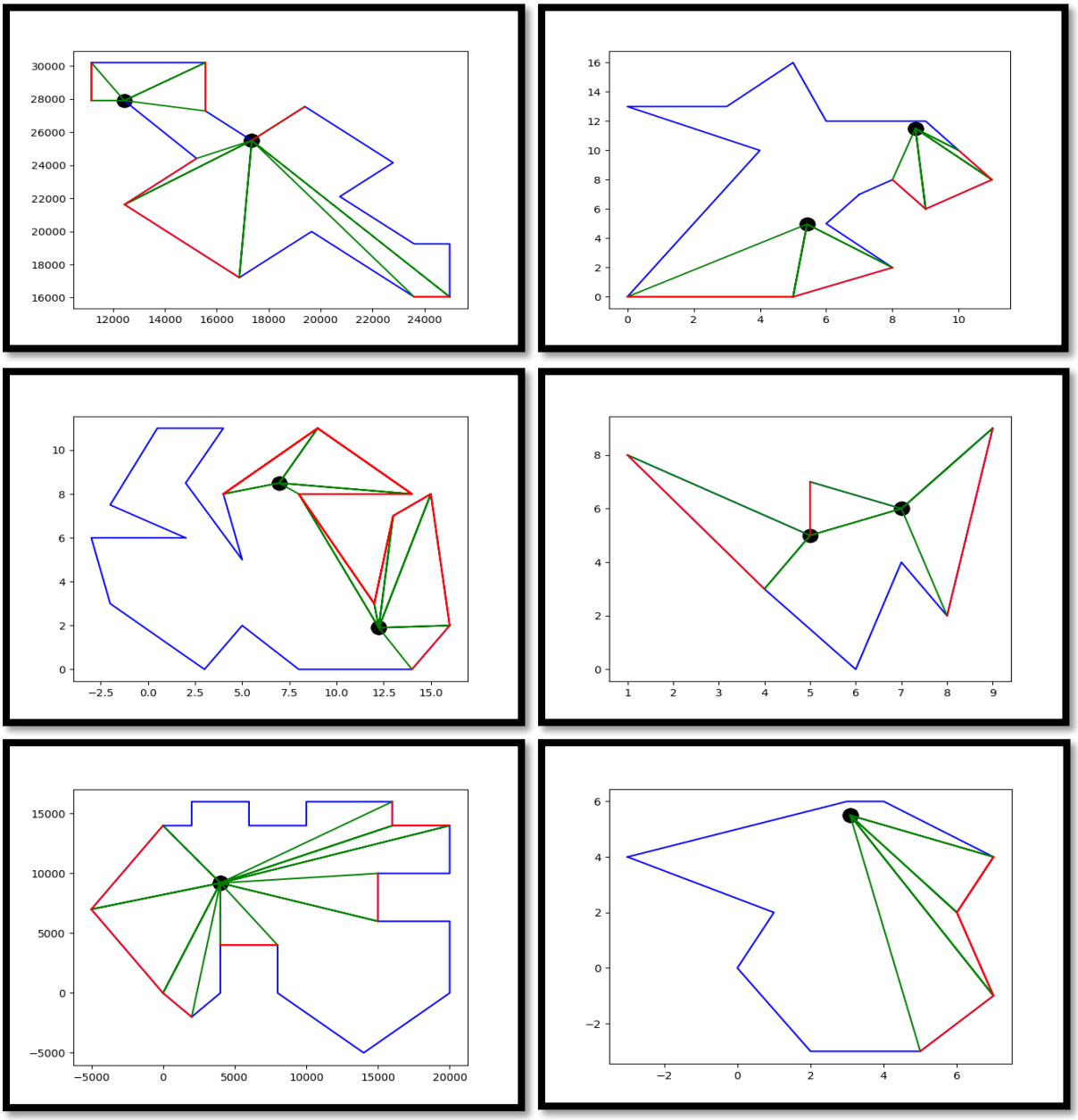
Selective Edge Cover:

In some cases not all the edges of the polygon are required to be covered by the guards; hence I made some change in the New Algorithm (Diagonalisation Edge Cover) for selective edge cover. The changes were to separately add those edges, along with complete polygon, which were required to be covered.

(Section 8.1)

Implementation of Selective Edge Cover in Python:

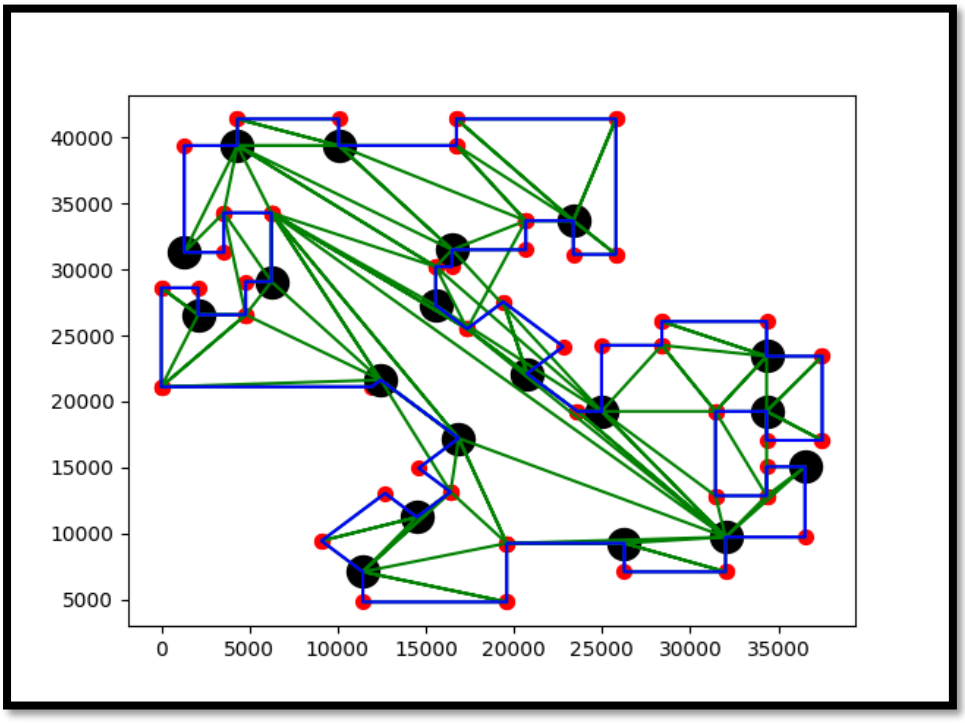
(Refer Appendix 8 and 9)



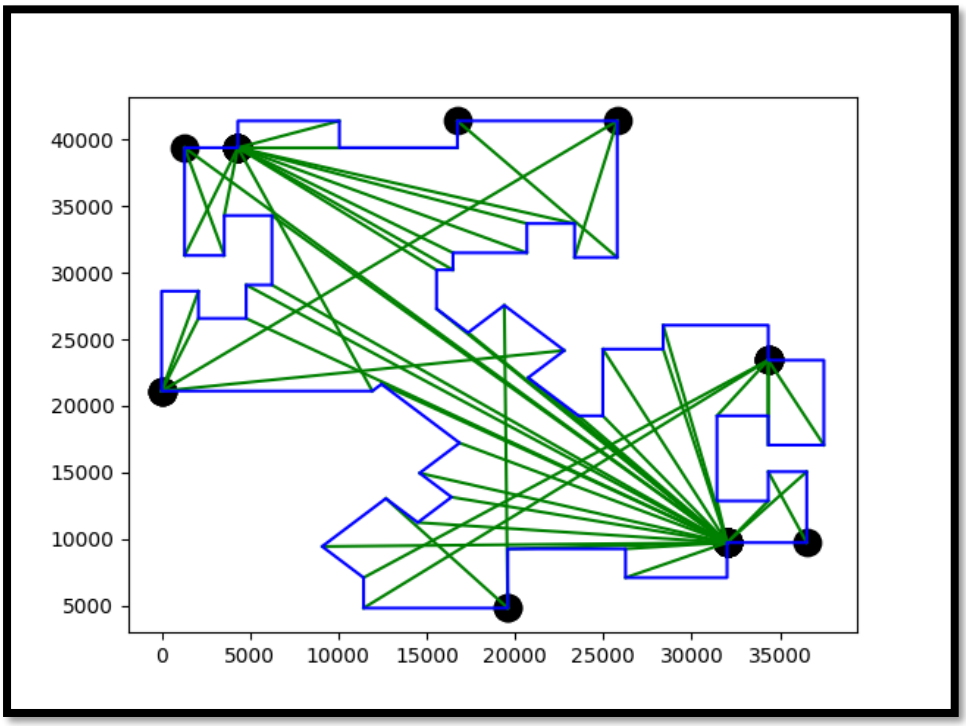
(Section 9)

Building Shell Example Results

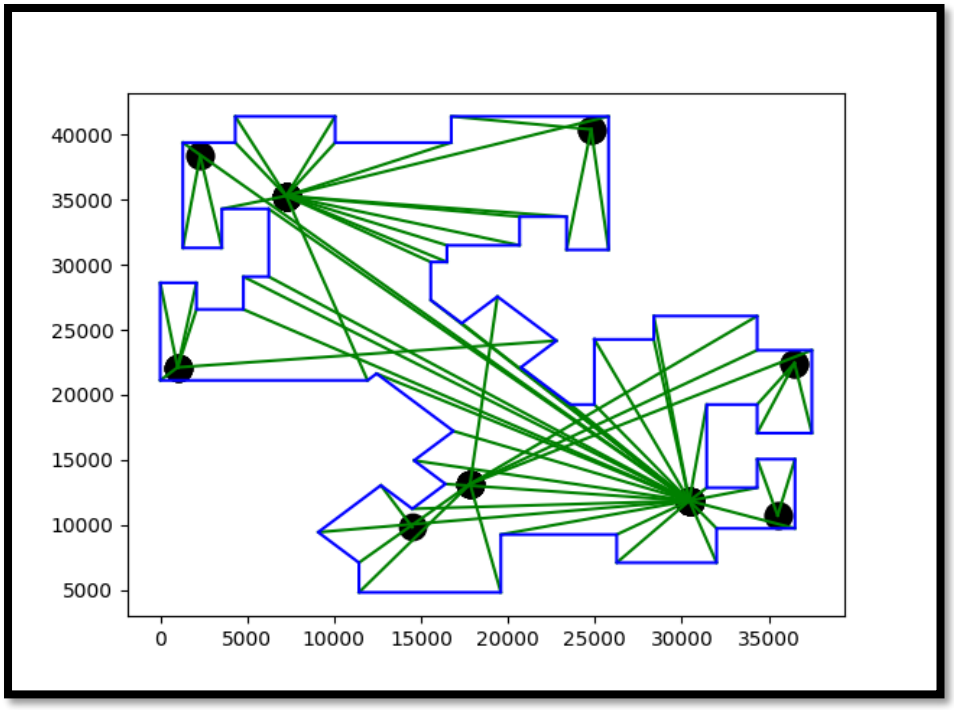
1) Algorithm Using Ear-clipping Method of Triangulation:



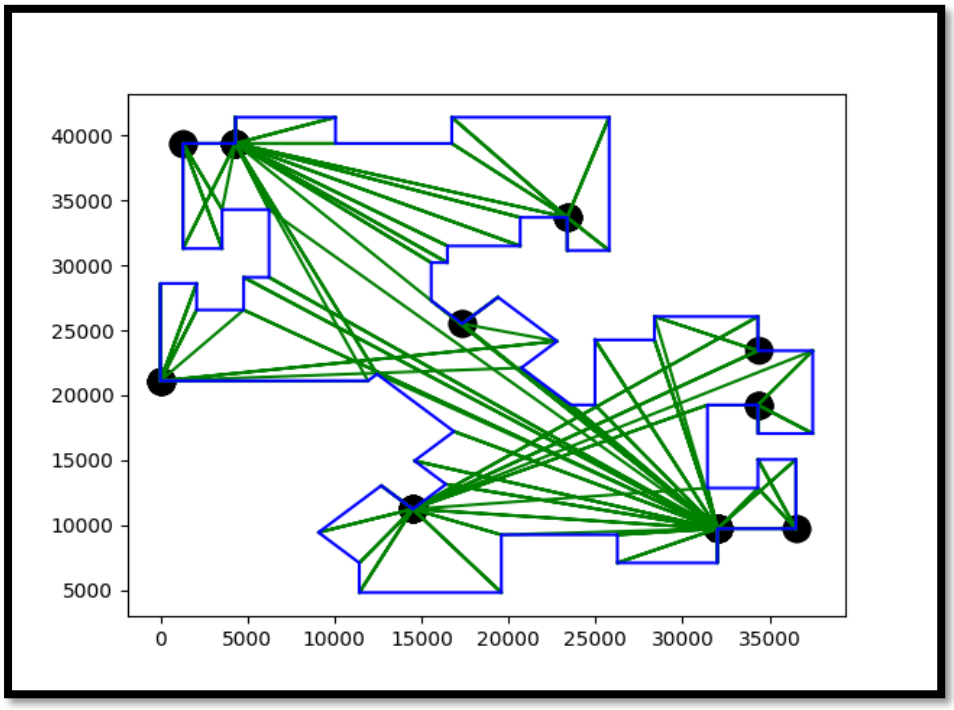
2) New Algorithm Using Diagonalisation Method (Vertex Cover)



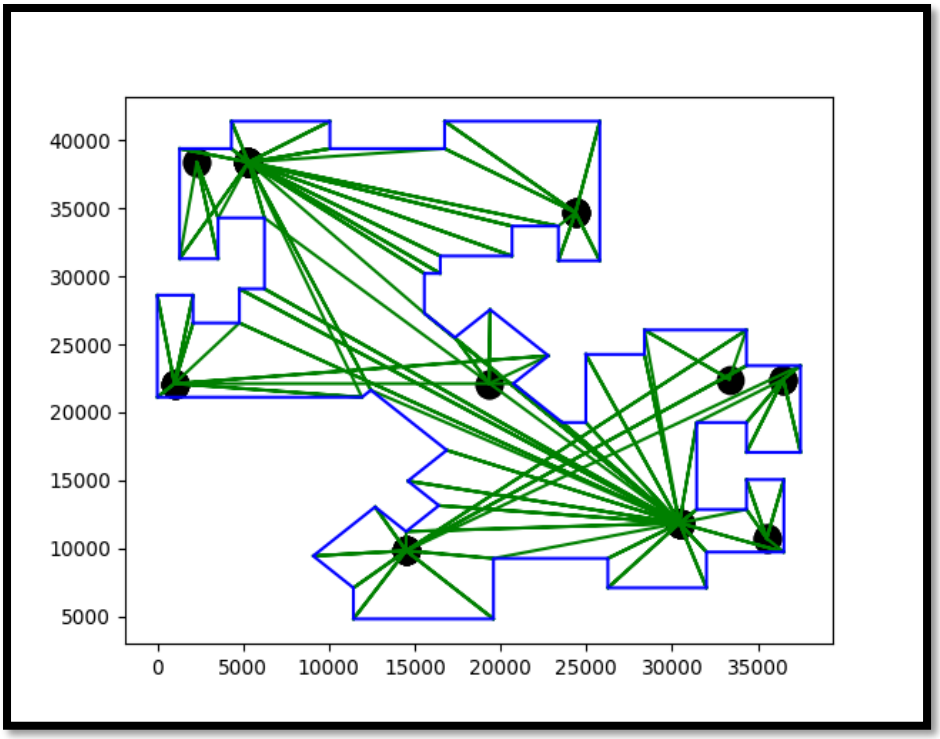
3) New Algorithm Using Diagonalisation Method (Vertex Cover – Shrink Polygon)



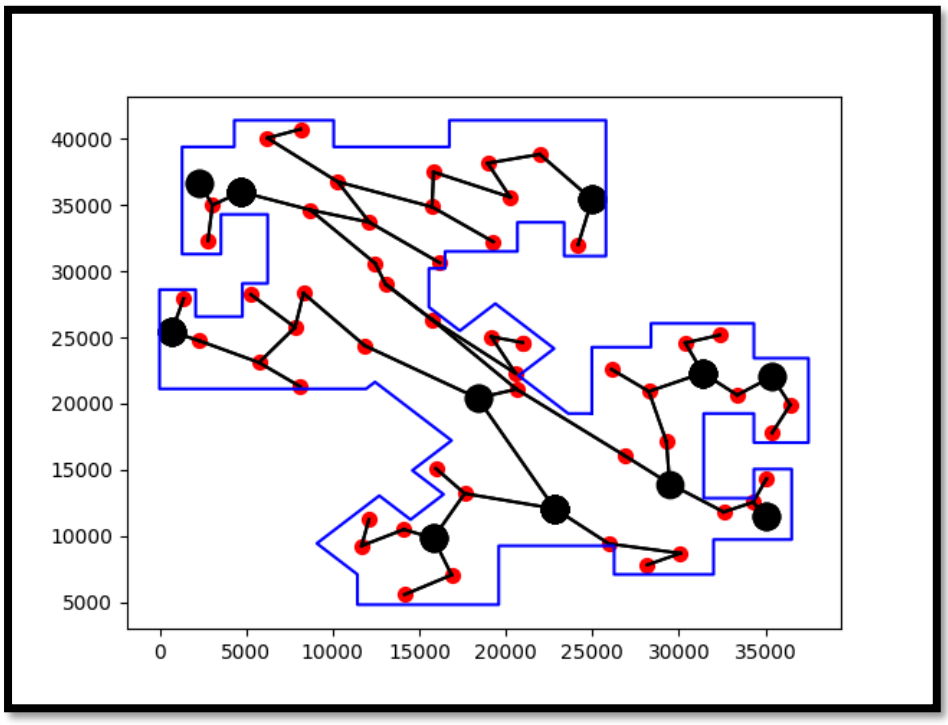
4) New Algorithm Using Diagonalisation Method (Edge Cover)



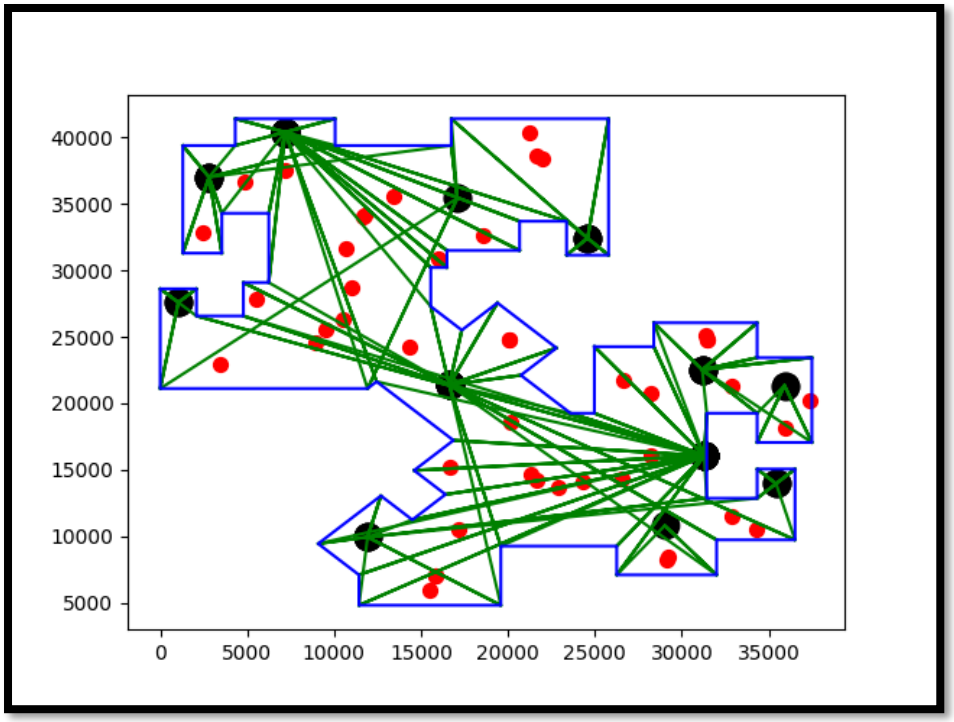
5) New Algorithm Using Diagonalisation Method (Edge Cover – Shrink Polygon)



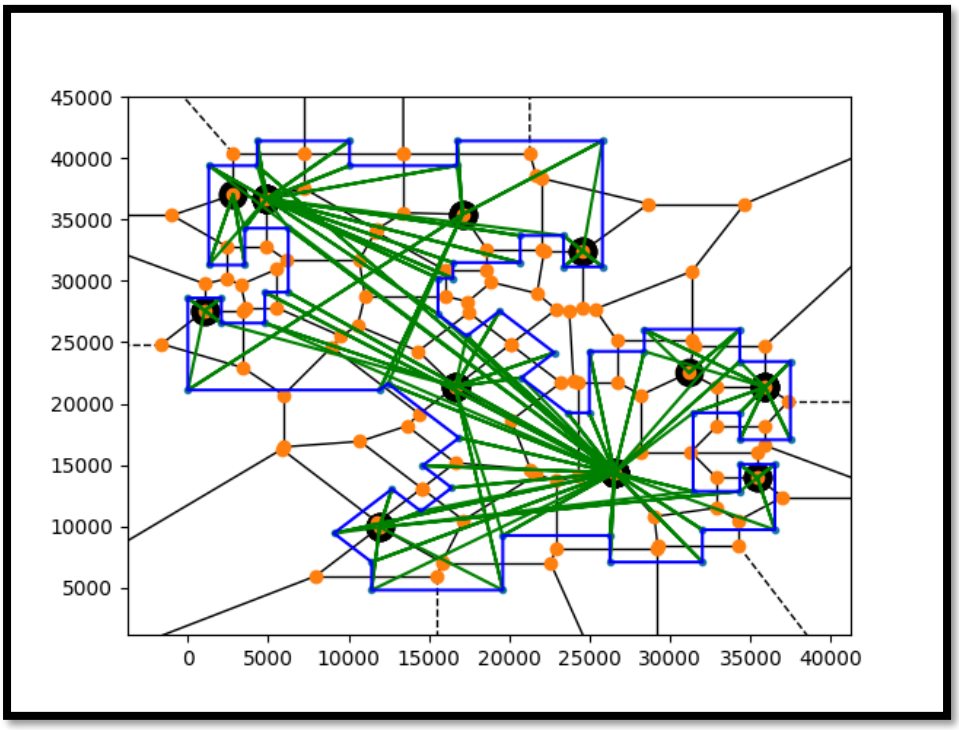
6) New Algorithm Using Diagonalisation Method (Edge Cover – Dual Tree)



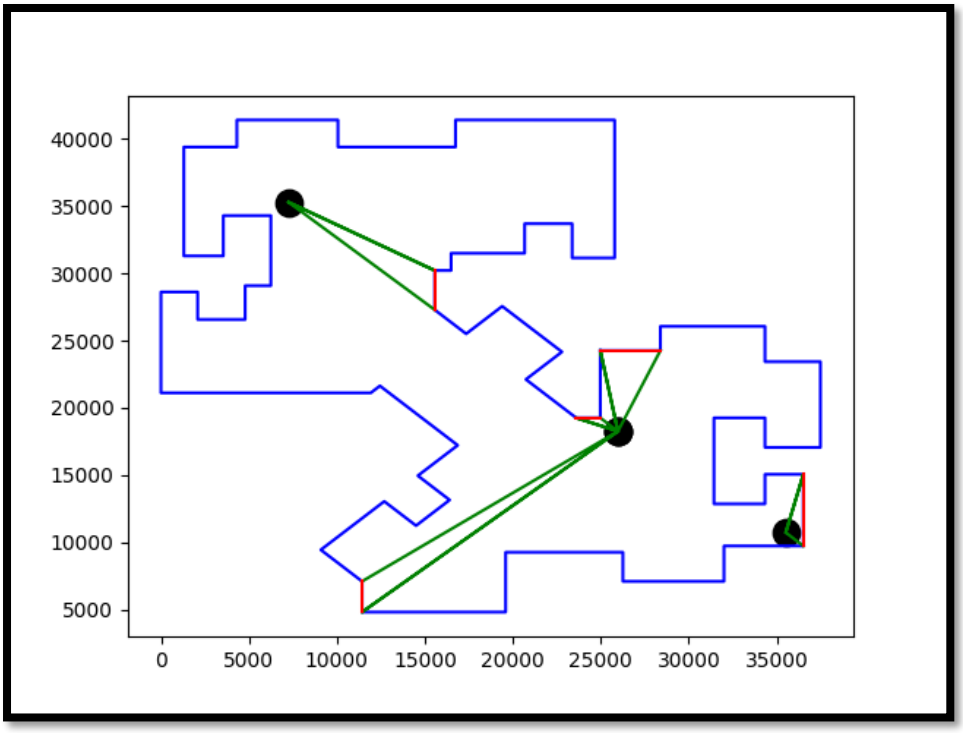
7) New Algorithm Using Diagonalisation Method (Edge Cover – Delaunay Triangulation)



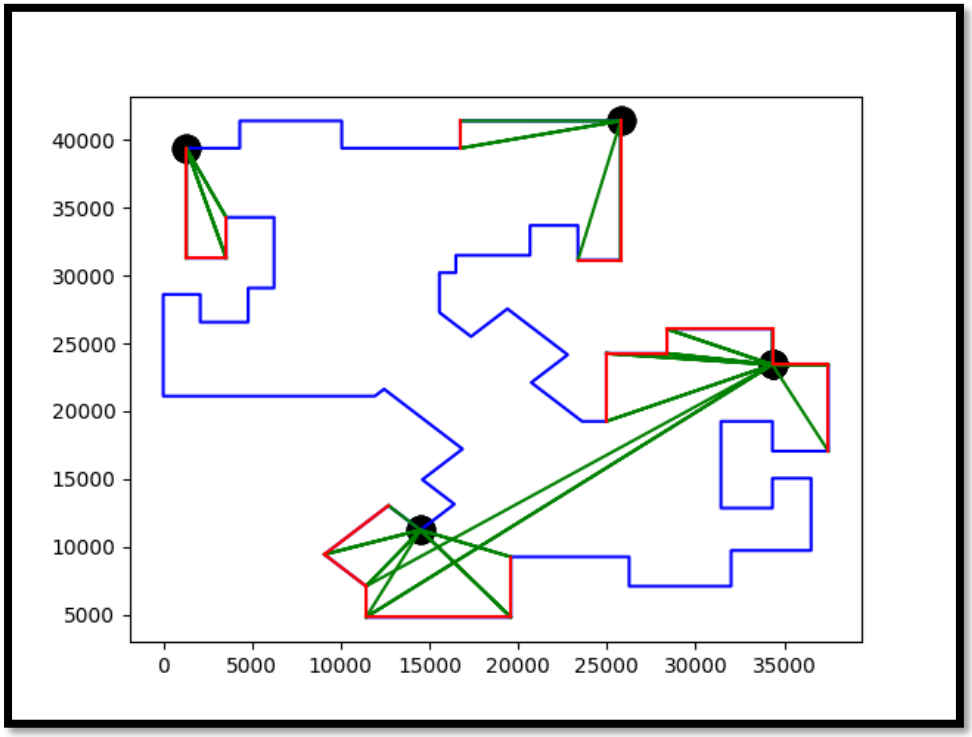
8) New Algorithm Using Diagonalisation Method (Edge Cover – Voronoi Diagram)



9) New Algorithm Using Diagonalisation Method (Edge Cover – Selective Edge Cover)



10) New Algorithm Using Diagonalisation Method (Edge Cover – Selective Edge Cover)



(Section 10)

Performance Time for Building Shell of 65 edges:

Type of Algorithm and Code	Performance time in Python in Seconds (Complete Code)	Performance Time in Python After the Input is Given
Solution for Algorithm using Ear-Clipping Triangulation Method	1.060958 sec	N/A
Solution for New Algorithm Using Diagonalisation Method (Vertex Cover)	1.580784 sec	N/A
Solution for New Algorithm Using Diagonalisation Method (Vertex Cover – Shrink Polygon)	1.529716 sec	1.518773 sec
Solution for New Algorithm Using Diagonalisation Method (Edge Cover)	1.543774 sec	1.540648 sec
Solution for New Algorithm Using Diagonalisation Method (Edge Cover – Shrink Polygon)	1.586331 sec	1.576901 sec
Solution for New Algorithm Using Diagonalisation Method (Edge Cover – Dual Tree)	1.568779 sec	1.553146 sec
Solution for New Algorithm Using Diagonalisation Method (Edge Cover – Circumcenters of Delaunay Triangulation)	2.630174 sec	2.598583 sec
Solution for New Algorithm Using Diagonalisation Method (Edge Cover – Vertices of Voronoi Diagram)	5.555641 sec	5.426995 sec

(Section 11)

Conclusions:

Contributions:

1) Developed the Diagonalisation Method, to overcome the disadvantages of triangulation method, to solve the art gallery problem. Diagonalisation Method gives better and efficient results than the triangulation method, in terms of minimum guard solutions and also the frequency of varying solutions.
2) Implemented the method of Edge Cover to completely cover the polygon, without leaving any space uncovered.
3) Implemented the Voronoi Diagram with the Diagonalisation Method to get a better scan of edges from the guards.

So finally my Diagonalisation Method can be successfully used with the vertex guards, shrink polygon guards, and Voronoi diagram vertex guards to efficiently solve the real life Art Gallery Problems.

Limitations:

1) The specific scan quality is still not defined and needs to be worked upon.
2) The scanning range is also not defined and needs to be worked upon.
3) The time taken by the Voronoi diagram implementation with the diagonalisation method is much larger as compared to the other methods.

(Section 12)

Future Scope:

1) A specific scan quality can be defined and can be worked upon;
2) Work on Art Gallery Problem with Holes can also be done;
3) Art Gallery Problem for Non – Planar surfaces can also be tried, to find a solution; etc

(Section 13)

References:

1) V. Chvatal, A combinatorial theorem in plane geometry, Journal of Combinatorial Theory, Series B 18 (1975) 39-41.
2) S. Fisk, A short proof of Chvatal's watchman theorem, Journal of Combinatorial Theory, Series B 24 (1978) 374
3) D. Avis, G.T. Toussaint, An efficient algorithm for decomposing a polygon into star-shaped polygons, Pattern Recognition 13 (1981) 395-398
4) J. Kahn, M. Klawe, and D. Kleitman. Traditional galleries require fewer watchmen. SIAM Journal of Algebraic and Discrete Methods, 4:194–206, 1983.
5) Subir Kumar Ghosh ,Approximation algorithms for art gallery problems in polygons, Discrete Applied Mathematics, Volume 158, Issue 6, (2010) 718-722
6) The Art Gallery Problem: An Overview and Extension to Chromatic Coloring and Mobile Guards - Nicole Chesnokov, 2018.
7) F. Hoffmann, M. Kaufmann, and K. Kriegel. The art gallery theorem for polygons with holes, In Proceedings of the 32nd IEEE Symposium on the Foundation of Computer Science, pages 39–48, 1991.
8) I. Bjorling-Sachs and D. L. Souvaine. An efficient algorithm for guard placement in polygons with holes. Discrete & Computational Geometry, 13:77–109, 1995.
9) J. O'Rourke. Art Gallery Theorems and Algorithms. Oxford University Press, New York, 1987.
10) A. Aggarwal. The art gallery theorem: its variations, applications, and algorithmic aspects.
Ph. D. Thesis, Johns Hopkins University, 1984
11) E. Györi, F. Hoffmann, K. Kriegel, and T. Shermer. Generalized guarding and partitioning

for rectilinear polygons. Computational Geometry: Theory and Applications, 6:21–44, 1996.

12) J. Czyzowicz, E. Rivera-Campo, N. Santoro, J. Urrutia and J.Zaks, *Guarding rectangular art galleries*, Discrete Applied Mathematics, 50 (1994), 149-157.
13) [CJN] Carlsson, S., Jonsson, H., Nilsson, B. J. (1993). Finding the shortest watchman route in a simple polygon. Algorithms and Computation Lecture Notes in Computer Science, 22(3), 58-67. doi:10.1007/3-540-57568-5 235
14) [EL] Erickson, L.H., LaValle, S.M. (2010). A chromatic art gallery problem.