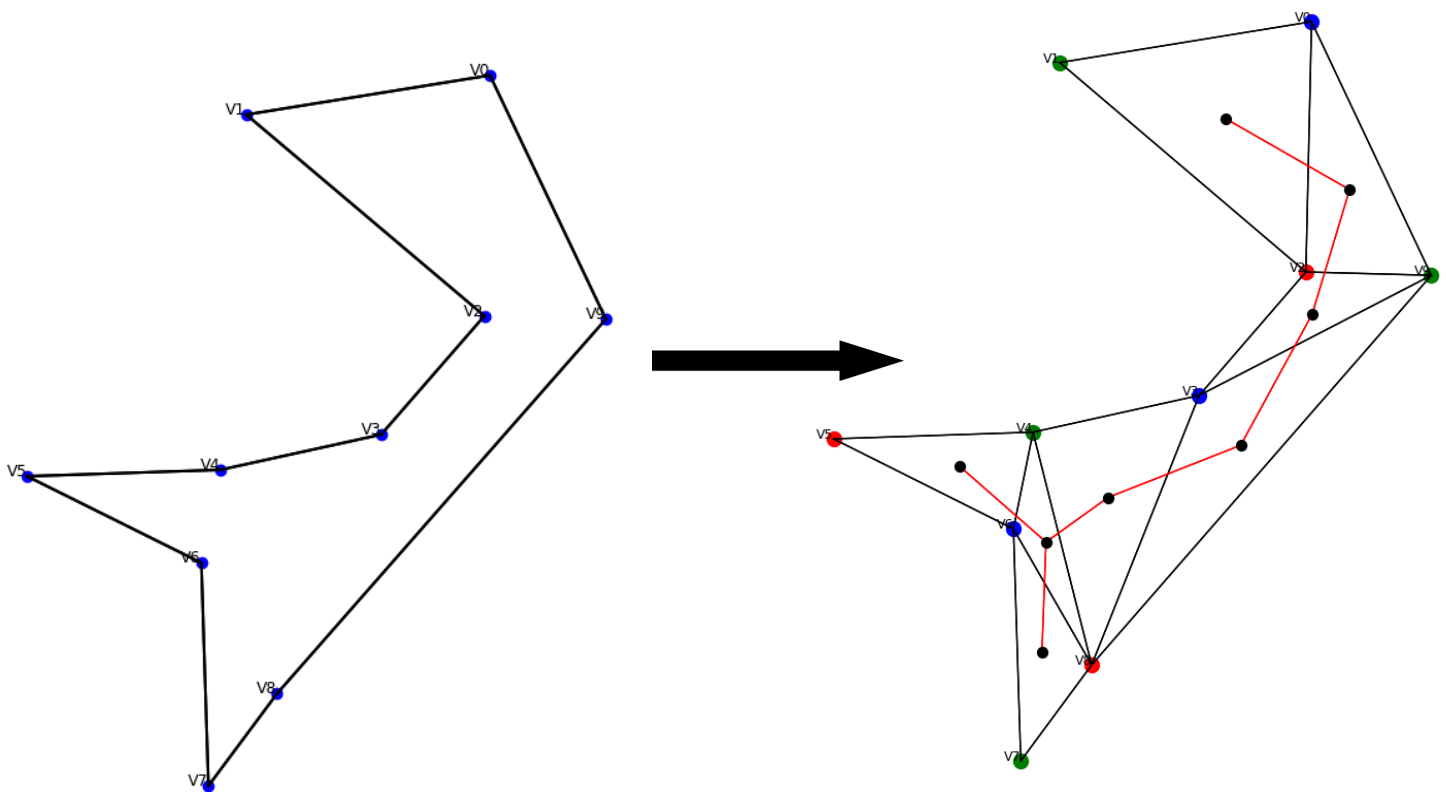


CSN-523

Computational Geometry

Coding Project 1: Report

Implementation of different Computational Geometry algorithms for solving the ArtGallery Problem.



Group ID: 14 ([Github](#), [Demonstration Video](#))

Name	Enrollment Number	E-Mail Id	Mobile Number
Ashutosh Kumar	21114021	a_kumar1@cs.iitr.ac.in	7061339843
Raiwat Bapat	21114078	r_nbapat@cs.iitr.ac.in	7666191528

1. Introduction

This report presents the implementation of computational geometry algorithms to solve the Art-Gallery Problem. The main objective is to analyze the algorithms for constructing a simple polygon, obtaining monotone partitions, and performing triangulation. We then explore the application of the triangulated polygon in the context of the Art-Gallery Problem, determining the minimum number of guards necessary to cover the polygon, using graph-theoretic coloring methods.

2. Problem Statement

The Art-Gallery Problem involves placing a minimum number of guards within a polygon (representing an art gallery) such that every point within the polygon can be viewed by at least one guard. The solution is based on the following computational geometry algorithms:

1. Constructing a simple polygon with a user-defined number of vertices.
2. Performing Sweep Line Monotone Partitioning of the polygon.
3. Triangulating each monotone partition.
4. Constructing Dual Graph of the Triangulated Polygon
5. Three Coloring of the Dual Graph

3. Algorithms and Methods

3.1 Constructing a Simple Polygon

A **simple polygon** is a closed shape in the plane formed by straight, non-intersecting lines. The first task is to generate a random polygon based on n vertices, where n is a user-defined input. The polygon must ensure the **DCEL** (Doubly Connected Edge List) data structure is used to store geometric information, including vertices, half-edges, and faces.

Code Snippet:

```
Python

class DCEL:

    def __init__(self, n=None, vertices=None, half_edges=None, faces=None):

        self.n = n if n else len(vertices)

        if n is not None:
```

```

        self.vertices = []
        self.half_edges = []
        self.faces = []
        self.images = []
        self.random_simple_polygon()

    else:
        self.vertices = vertices if vertices else []
        self.half_edges = half_edges if half_edges else []
        self.faces = faces if faces else []
        self.images = []

    self.create_polygon()

```

The DCEL structure is used to handle the polygon's vertices, edges, and faces efficiently.

3.2 Sweep Line Monotone Partitioning

Once the simple polygon is constructed, the next step is Sweep line monotone partitioning—a computational geometry algorithm used to divide a polygon into simpler monotone polygons. A polygon is monotone if any vertical line intersects it at most twice. The algorithm employs a sweep line that moves from top to bottom, splitting the polygon by adding diagonals between vertices to ensure monotonicity. By using a priority queue to manage events (vertex additions) and a balanced tree for edge management, it identifies and handles critical vertices (split, merge) to insert necessary diagonals. This partitioning simplifies subsequent polygon triangulation.

MonotonePartitioner Class:

In this class, the sweep line algorithm is implemented to partition the polygon into monotone components. The vertices are classified as start, split, end, merge, or regular vertices.

```

Python

class MonotonePartitioner:

```

```

def __init__(self, dcel):
    self.dcel = dcel # The DCEL representation of the original polygon
    self.status_tree = StatusTree() # Status structure (active edges)
    for the sweep line
    self.vertex_types = {} # To store classified vertices
    self.new_diagonals = []

def classify_vertices(self):
    """ Classifies vertices into start, end, split, merge, and regular """

def perform_sweep_line_partition(self):
    """ Perform the sweep line algorithm to partition the polygon into
    monotone pieces """

def handle_regular_vertex(self, vertex):
    """ Handle regular vertex during the sweep """

def handle_merge_vertex(self, vertex):
    """ Handle merge vertex by adding a diagonal """

def handle_end_vertex(self, vertex):
    """ Handle end vertex during the sweep """

def handle_start_vertex(self, vertex):
    """ Handle start vertex during the sweep """

def handle_split_vertex(self, vertex):
    """ Handle split vertex by adding a diagonal """

```

The above class handles vertex classification and inserts diagonals to divide the polygon into monotone pieces.

After monotone partitioning, the polygon is subdivided into monotone polygons—polygons that have a property where any vertical line intersects the polygon at most twice. Monotone partitioning is a crucial step because triangulation can be efficiently performed on monotone polygons.

3.4 Triangulation of Monotone Polygons

Each monotone polygon can be further divided into triangles through triangulation. This is important because solving the Art-Gallery Problem becomes easier on

triangulated polygons. For a polygon with n vertices, triangulation adds exactly $n-3$ diagonals to form $n-2$ triangles.

The MonotoneTriangulation class is implemented to handle this step:

```
Python

class MonotoneTriangulation:

    def __init__(self, dcel: DCEL):

        self.dcel = dcel # The DCEL that contains monotone polygons

        self.new_diagonals = [] # List of added diagonals

    def triangulate_monotone_polygon(self, face: Face):

        # Triangulate a monotone polygon represented by a face in the DCEL.

    def triangulate(self):

        # Main triangulation method. Triangulates each monotone polygon
        stored as faces in the DCEL.
```

The vertices of each monotone face are sorted, and diagonals are added to divide the face into triangles. This ensures that the original polygon is now fully triangulated, and the information is stored in the DCEL structure.

4. Solving the Art-Gallery Problem

4.1 Triangulated Dual Graph

With the triangulated polygon, we construct the dual graph of the triangulated faces. In this dual graph, each node represents a face (triangle), and edges connect nodes if the corresponding triangles share a common edge. The goal is to perform 3-coloring on the dual graph, as it has been proven that the vertices of any triangulated polygon can be colored with three colors such that no two adjacent vertices share the same color.

The **DualGraph** class is used to build and color the dual graph:

```
Python

class DualGraph:

    def __init__(self, dcel):

        # Initialize the dual graph with the given DCEL.

        self.dcel = dcel
```

```

        self.dual_graph = {}

    def build_dual_graph(self):
        """Build the dual graph by iterating through each face and connecting
        adjacent faces that share a common edge. """

```

4.2 Three-Coloring and Vertex Guard Placement

Using DFS (Depth-First Search), the vertices of the triangulated polygon are colored with three distinct colors. The 3-coloring ensures that the minimum number of guards required to secure the entire gallery is the smallest color set (i.e., the color assigned to the fewest vertices).

```

Python

class DualGraph:


    def dfs_color_faces(self, face, available_colors):
        """
        DFS Algo for 3-coloring
        """

    def three_coloring(self):
        """
        Perform three-coloring of the vertices of a triangulated polygon
        using the dual graph.

        The graph must be represented as an adjacency list of faces, where
        each face contains three vertices.
        """

    def plot_colored_dcel_with_dual_graph(self):
        """
        Plots both the colored DCEL (vertices and edges) and the dual graph
        (faces and their adjacency)
        in the same image.
        """

```



This function visually demonstrates the solution, plotting the polygon, the guards, and the coloring of vertices.

The number of guards corresponds to the vertices with the least-used color in the 3-coloring solution. The guards are placed at these vertices to ensure coverage of the entire gallery.

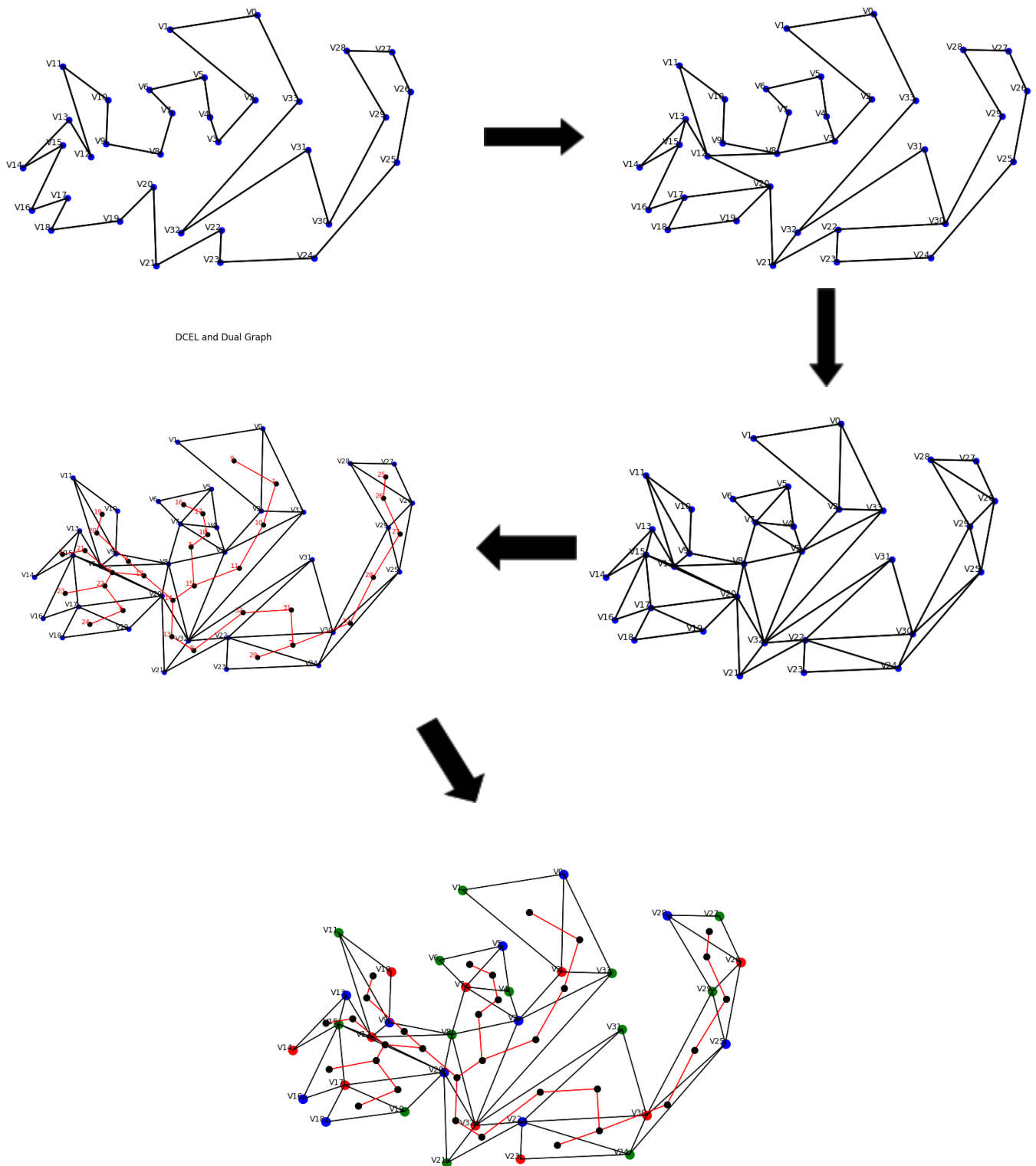
5. Results and Demonstration

We tested the implementation with six different polygon sizes, three with less than 20 vertices and another three with more than 20 vertices. Both cases involved the following steps:

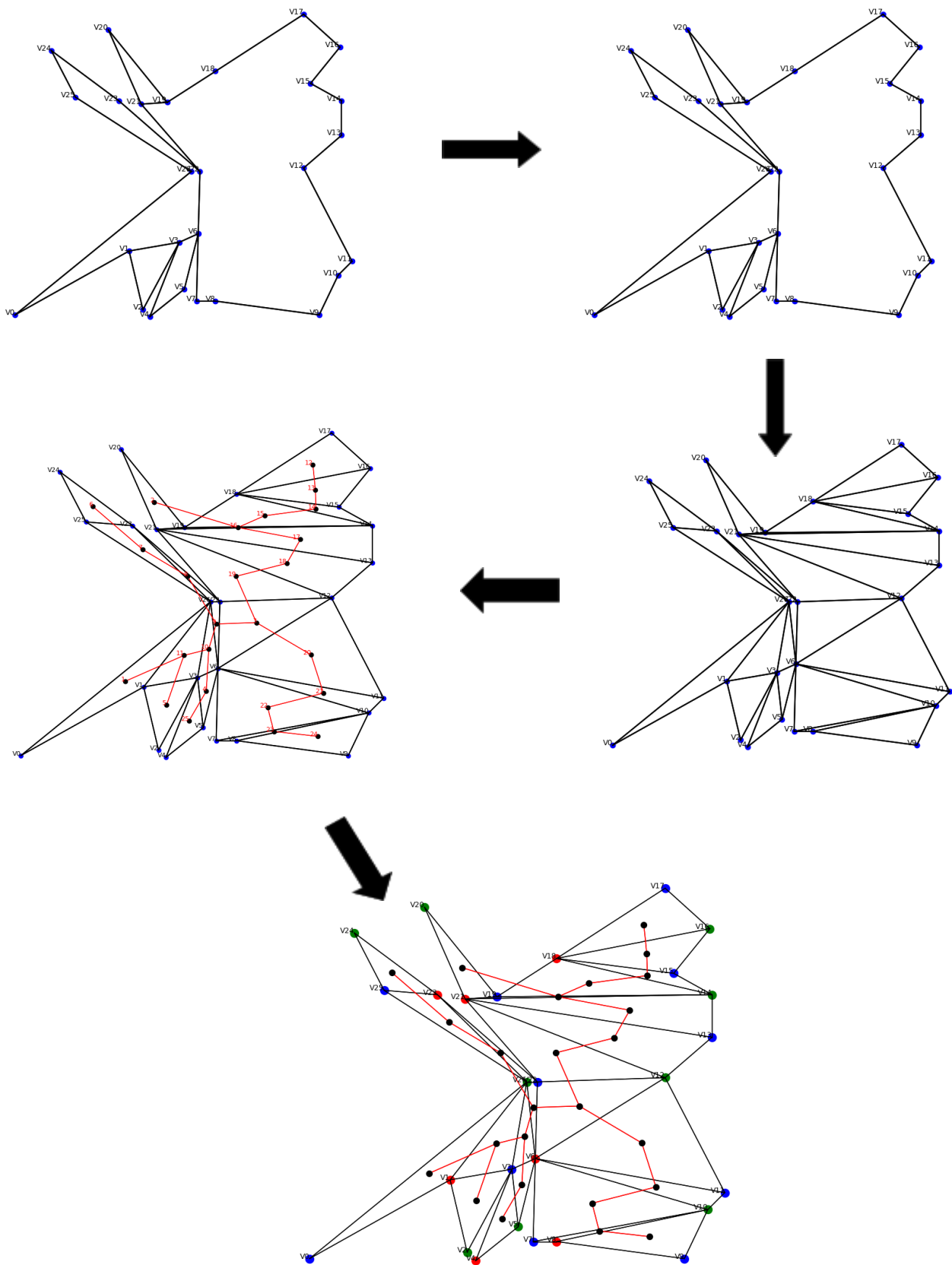
- Polygon generation.
- Monotone partitioning.
- Triangulation of monotone polygons.
- Dual graph creation and 3-coloring.
- Determination of the number of guards and their placement.

Screenshots of the outputs of different intermediate steps are depicted from the next page.

Polygon with 34 Vertices

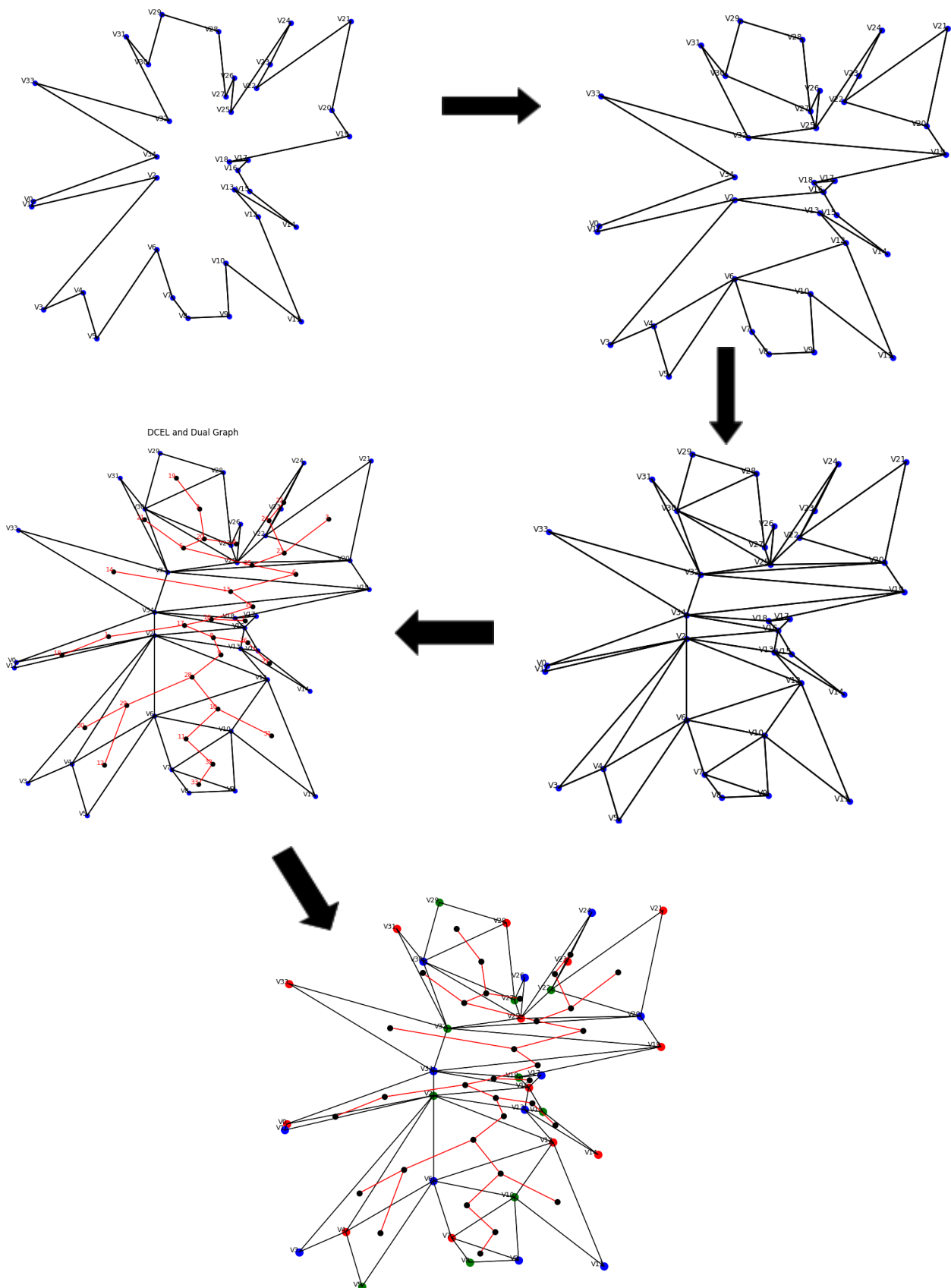


Polygon with 27 Vertices



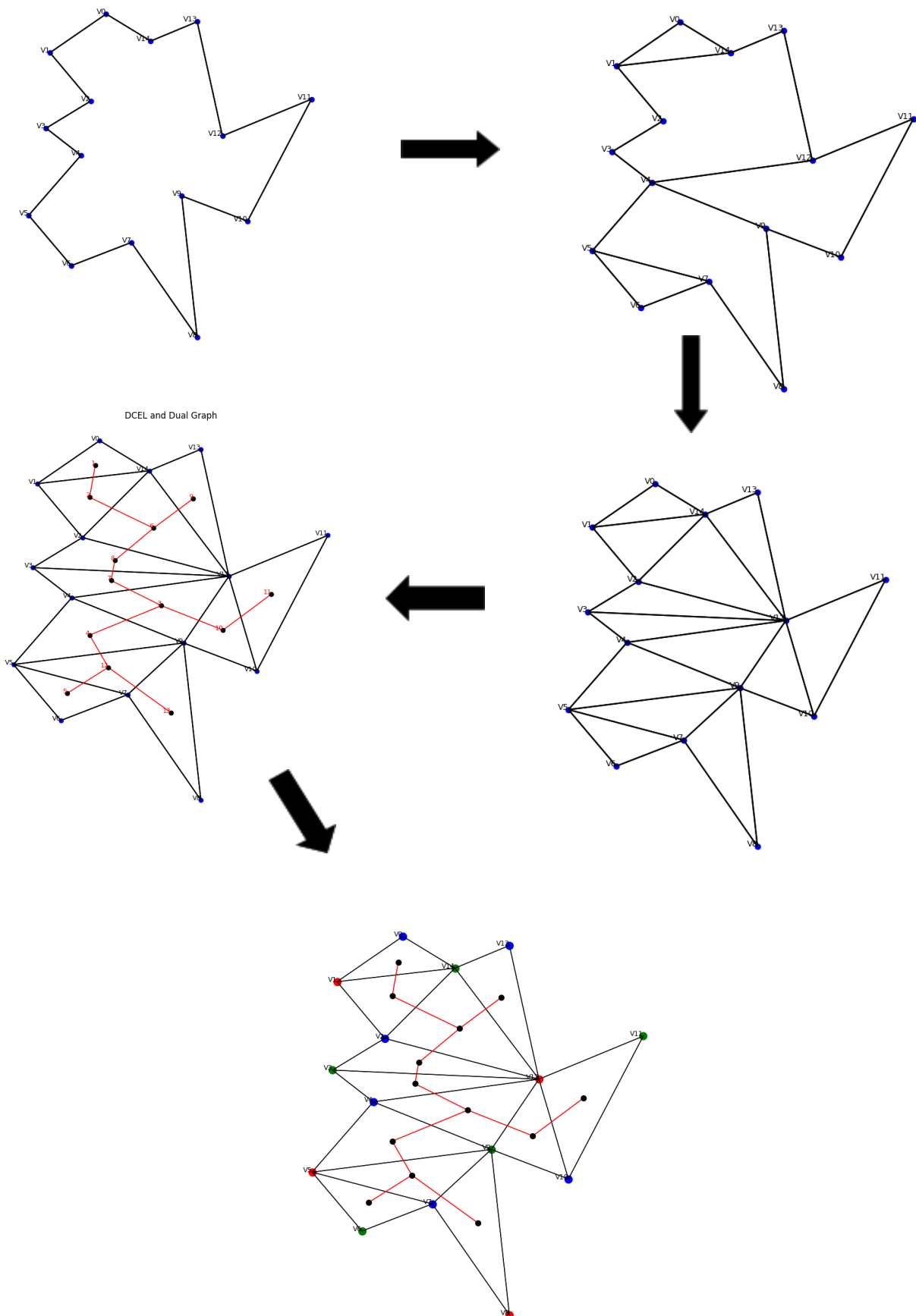
The Sufficient number of Vertex Guards required will be 7, Colored in Red

Polygon with 35 Vertices

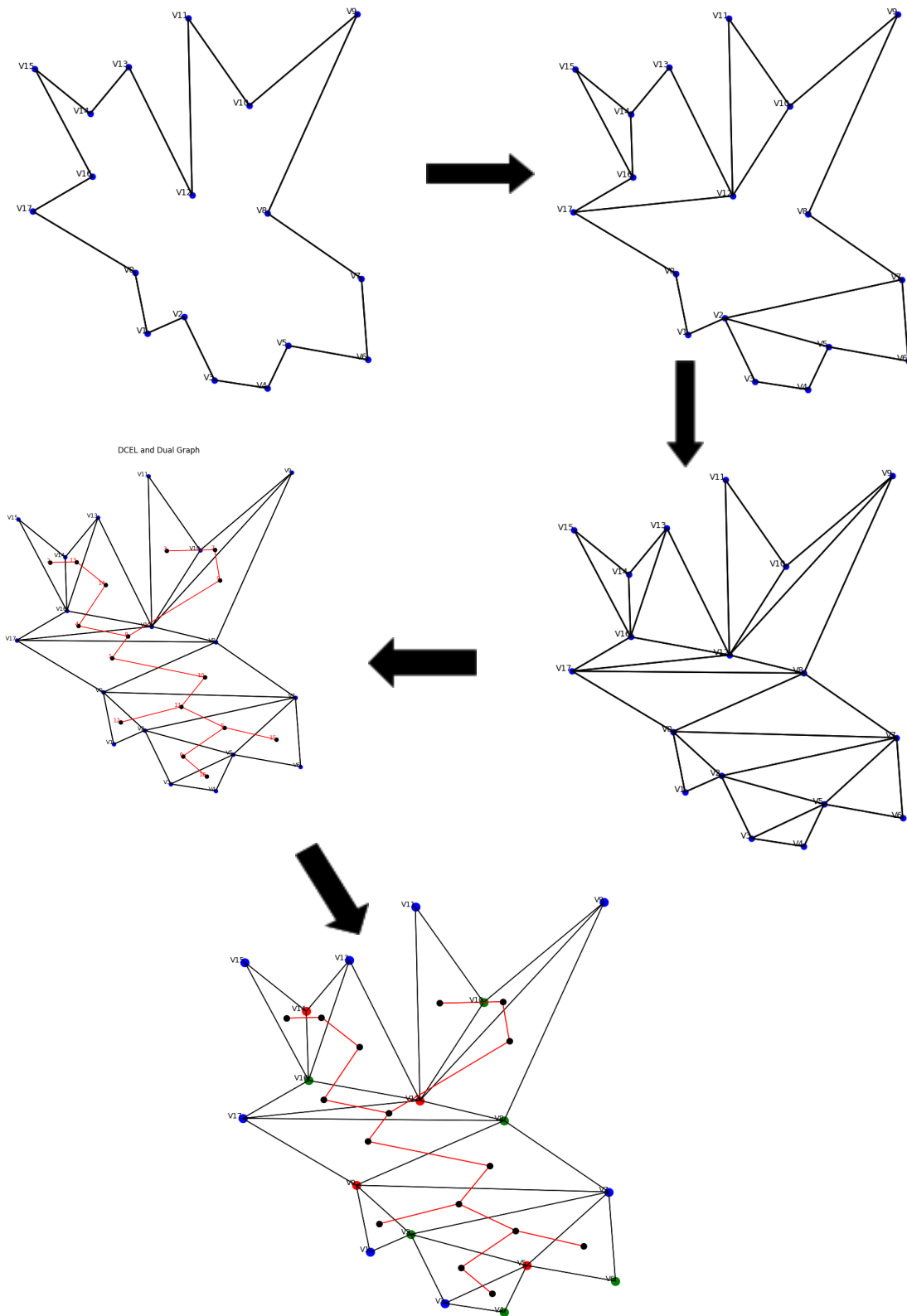


The Sufficient number of Vertex Guards required will be 10, Colored in Green

Polygon with 15 vertices

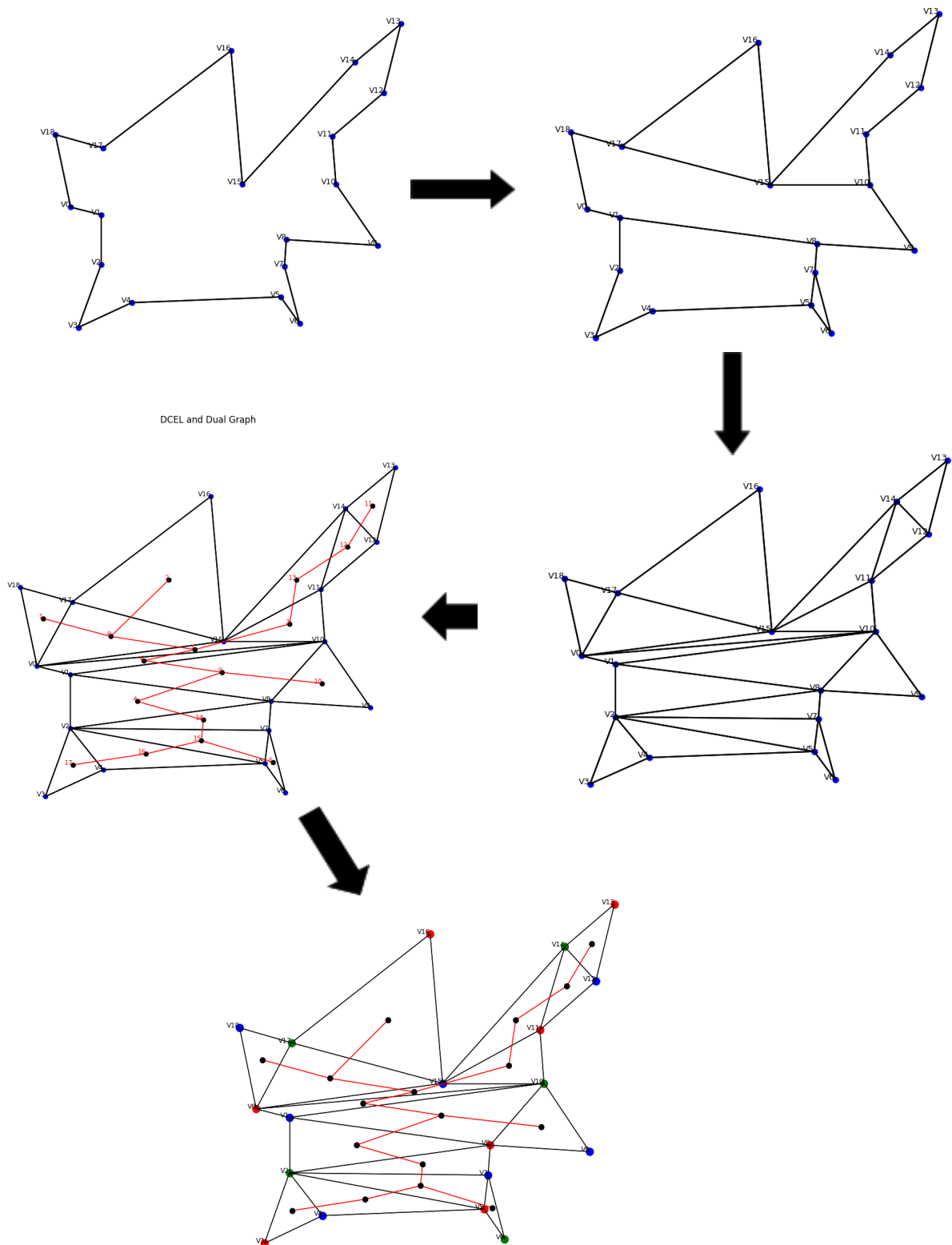


Polygon with 18 Vertices



The Sufficient number of Vertex Guards required will be 4, Colored in Red

Polygon with 19 Vertices



The Sufficient number of Vertex Guards required will be 5, Colored in Green

