Jacob Gayban
NSHE ID: 5006017024
CS302

Art Gallery Problem Bonus Assignment

**Table of Contents**

## Overview of Algorithm

This project makes use of the monotone (specifically y-monotone) decomposition method of triangulating a polygon, which breaks the polygon up into several smaller pieces that are easier to triangulate, and from there we can reconstruct the original polygon. More detail is provided in the later sections, but this is the general concept.

The entire solution is heavily based on this lecture series (with a few modifications to accommodate non-strict y-monotone polygons).

### Data Structures

The majority of this project was made during the first few months of class, when we didn't go over graphs, meaning that many of the data structures used in this project are not "explicitly" graphs in the way that we have learned them, but they do share similarities with them. Throughout this document I will make note of the graph properties I would have used to accomplish certain tasks had my data structures provided the functionality.

There are two main data structures involved with the triangulation process: the Mesh, which stores the original polygon and any changes made to it, and the DCEL (doubly-connected edge list), which stores the monotone pieces created by the decomposition, and then triangulates them, will all updates being reflected in the original Mesh.

#### Mesh

This project could have been done with only a DCEL, however the Mesh simplifies things by providing the edges and vertices of the polygon in a more explicit manner, making input and output easier.

The Mesh class is basically a container for 2 sets of data (implemented as std::vector's): the vertices and edges of the polygon. Any new edges added through the triangulation process are added to the edge vector.

The Edge and Vertex data structures are pretty straightforward; the Edges contain references to their endpoint vertices, and each Vertex contains its positional data and a few other values used to describe it.

In my implementation, each Vertex is connected to 2 others in a doubly linked list, where the "next" pointers point to the next vertex in clockwise order, and the "previous" vertex is the one in counter-clockwise order.

- Instead of having each vertex storing references to its neighbors, I could have also used an adjacency table to note the relations between each vertex. The adjacency matrix could also take the place of the Edge record of the Mesh, as they are implicitly defined within the matrix.

DCEL

The DCEL makes it easy to locate and define any new faces made from the monotone decomposition. A brief overview of the DCEL data structure: each “normal” edge is replaced with two directed edges called HalfEdges, where all the HalfEdges on the outside of a face flow clockwise (in my implementation), and all the HalfEdges on the inside flow counter-clockwise:

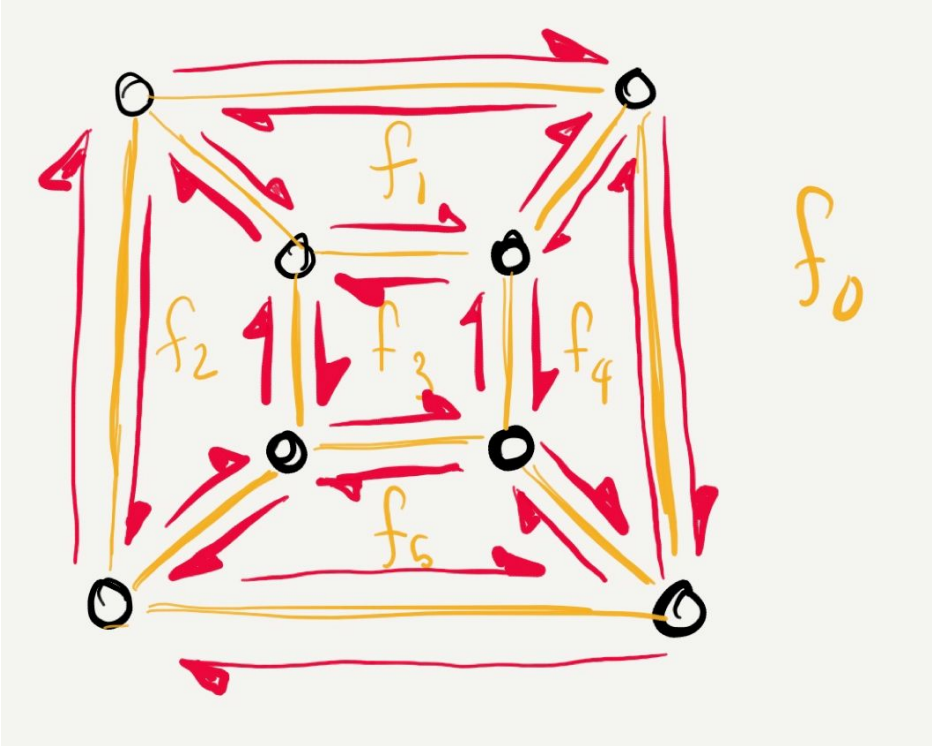



Figure 1: HalfEdge orientations in the DCEL [source]

Any diagonals added to the Mesh through the decomposition process will create 2 new HalfEdges which may or may not define a new “face”. Therefore, we just need to keep track of the new HalfEdges that are added in order to find all the new faces of the polygon. From there, we can grab the vertices of each face and begin the triangulation process.

**Polygon Creation and Storage**

From the formatted input (see Program Usage), we can create a doubly-linked list of vertices. The vertices may or may not be in clockwise order, so a checking function was made. The “setClockwise” algorithm is as follows:

1. Find the highest (y-value) vertex in the set
2. Create vectors from that vertex to its neighbors
3. Normalize each vertex and determine which one has a greater x-component
4. Set the “next” Vertex as the one with the greater x-component, and update the rest of the linked list accordingly

The reason why we’re setting the vertices in clockwise order is to make classifying each vertex easier (see Monotone Decomposition), and in preparation for the DCEL.

- Because each vertex “flows” clockwise, the interior of the polygon will always be to the right of the vector from a vertex V to its “next” neighbor

Once all the vertices are created, we can make all the edges. The edges also store their endpoints in clockwise order, also in preparation for the DCEL.

Any new edges that get added from the decomposition or triangulation get added to the end of the vector.

**Monotone Decomposition**

In a y-monotone polygon, any horizontal line passing through the polygon will intersect the boundary of the polygon at most twice. Parts of the polygon that are horizontal are only counted as one intersection. We can identify a y-monotone polygon by identifying the “types” of vertices that make up the polygon. There are 5 ways we can categorize a vertex:

| **Type** | **Relation to Neighbors** | **Location of Interior** |
|---|---|---|
| Start | Above both | Below vertex |
| Split | Above both | Above vertex |
| End | Below both | Below vertex |
| Merge | Below both | Above vertex |
| Regular | Between both | Left or right of vertex |
| **If two vertices are on the same y-level, the one with the lower x-value is considered “above” the other one. | | |

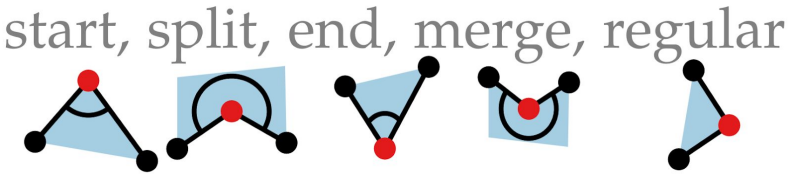



Figure 2: Types of vertices [source]

Two of these types of vertices will break the y-monotone property: the split and merge types. To “treat” these vertices, we have to add diagonals inside the polygon. The algorithm to do this is as follows (proof omitted):

1. Sort polygon vertices by height (prioritizing higher x-values first)
2. Visit each vertex in height order, and perform the following events depending on the vertex type:
    a. **Start** - no events happen

b. **Merge** - project a horizontal line to the left of the vertex and identify the first edge it hits; mark this merge vertex as the “helper” of that edge
    i. Do not consider the edge that has this merge vertex as one of its endpoints
    ii. Identify the edge clockwise from this vertex, and check if it has a helper
        1. If the helper is a merge or regular vertex, connect it to the bottom (lower y-value vertex) of the edge
        2. If the helper is a split vertex, connect it to the top (higher y-value vertex) of the edge
c. **Split** - project a horizontal line to the left of the vertex and identify the first edge it hits; if this edge has a “helper”, draw a diagonal from this vertex to the helper
    i. If there is no helper, this vertex becomes the helper
    ii. Do not consider the edge that has this merge vertex as one of its endpoints
d. **Regular** - check if this vertex can help an edge
    i. If it can, check the helper of that edge, then become the new helper
        1. If a merge vertex is helping that edge, connect to it
    ii. If the vertex cannot help an edge (the left side of the vertex is outside of the polygon), then perform the same operation as the merge vertex, part (ii)
e. **End** - perform the same operation as the merge vertex, part (ii)

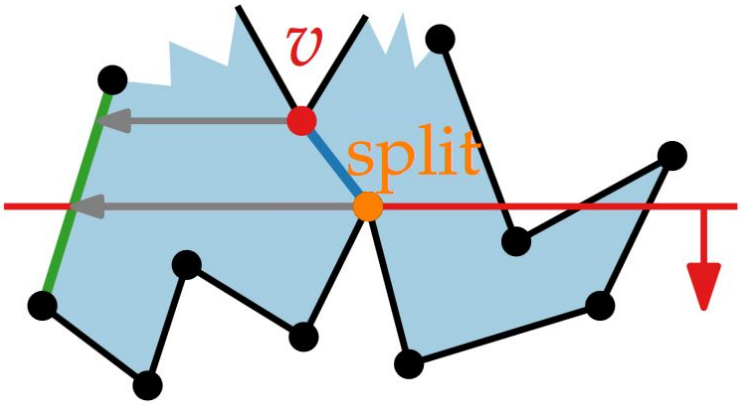


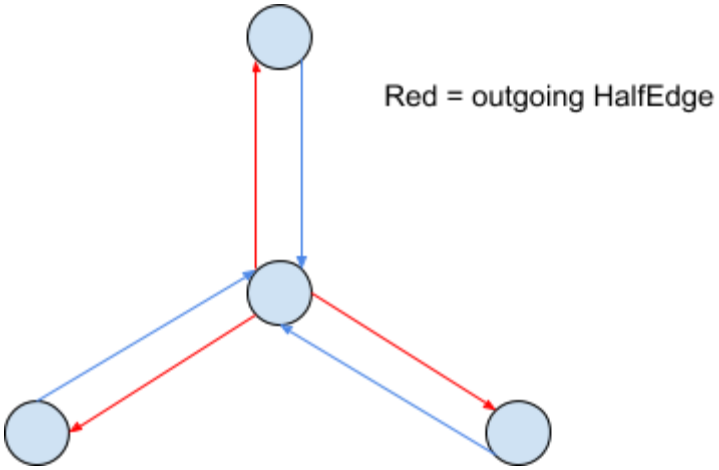

Figure 3. Example connection; merge vertex V is helper of green, split vertex becomes new helper, then connects to the old one

**DCEL Conversion**

Once the decomposition is finished, our polygon will be divided into y-monotone sections. Right now, there is no easy way to identify each of these pieces by itself, so we will create a DCEL out of our polygon, which will naturally isolate the faces. The process for creating a DCEL from our mesh is relatively straightforward. The specifics of the DCEL structure will be omitted for brevity, however the basis of my implementation will be included in the references.

The DCEL conversion looks more or less like this:

1. Create HalfEdges from every edge in the polygon
2. Have each vertex store a reference to all HalfEdges that originate from it
3. Once all the HalfEdges are created, we will connect them to each other following this algorithm:
    a. Order all outgoing HalfEdges by their angle from the positive x-axis
    b. Perform a “local solve” around each vertex:




- Each of the blue HalfEdges can be reached from the central vertex
- This algorithm basically connects each blue HalfEdge to the first red (outgoing) HalfEdge that is clockwise to it

4. Do the above for all vertices

Once all of the HalfEdges are linked correctly, we can begin identifying each of the y-monotone pieces we made earlier. To do this, we must:

1. Identify the HalfEdges associated with the added diagonals
2. For each HalfEdge, consider it as a potential “face”, and travel along its path
    a. If, as you go through the path, you find another HalfEdge that was created from a diagonal, remove this HalfEdge from consideration
    b. If you come back to where you started, this HalfEdge defines a unique face
        i. A face can be defined from just one of the HalfEdges that make up its border

Note: The DCEL could possibly work as a graph-based implementation, however the structure is easier to traverse in a linked-list form.

**Monotone Triangulation**

Once all of the “faces”/monotone pieces have been identified, we can begin triangulating them. This algorithm first assigns each vertex (except the “top” and “bottom” ones) to a left or right “chain” depending on if that vertex is on the left or right side of the monotone piece. We then visit each vertex in height order (prioritizing lower x-values), and perform the following actions:

1. Add the first two vertices into an “unconnected” stack
2. For every new vertex you visit:
    a. If the vertex is not on the same chain as the one on top of the stack, then connect the current vertex to all vertices in the stack (if possible) until the stack is empty
        i. Pop after every connection
        ii. Afterwards, push this vertex and the vertex “before” it (in height order) onto the stack
    b. If the vertex is on the same chain as the one on the top of the stack, connect the current vertex to all vertices in the stack until either the stack is empty or until you can’t make a valid connection
        i. Afterwards, push this vertex and the vertex you last connected to (if the stack gets emptied) or the vertex you couldn’t see (no valid connection)
3. Once you reach the last vertex, look at all the other vertices and make connections wherever possible

**Vertex Coloring**

The vertex coloring algorithm uses recursion to solve for all the vertices. First, color 2 vertices (v1 and v2) of any edge and then perform the following:

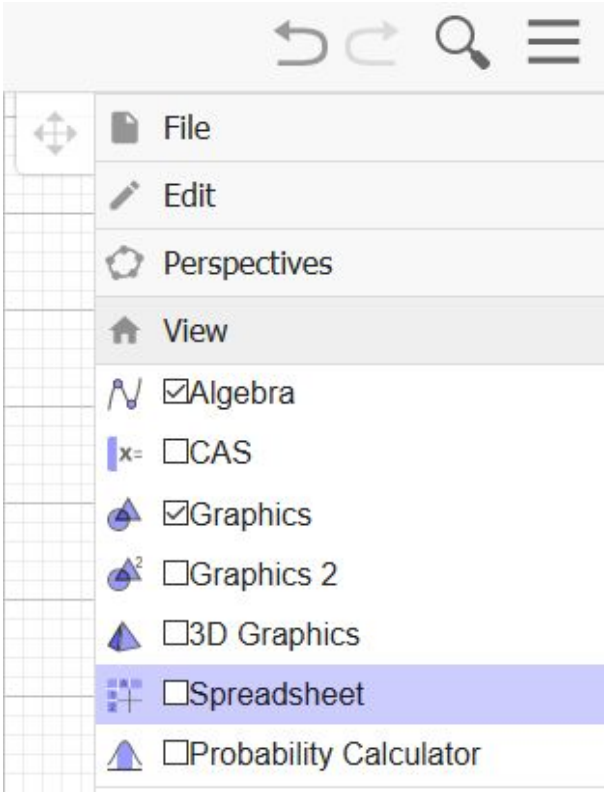
1. Find the common vertices of v1 and v2
    a. There are either 1 or 2 common vertices that they are adjacent to
2. Color all the common vertices with the correct third color, if they aren’t already colored
    a. If all the common vertices are colored, nothing more needs to be done
3. For every vertex v that you colored, perform 2 recursive calls with arguments:
    a. v and v1
    b. v and v2

Every “colored” edge can be used to “solve” up to 4 more edges, until all the edges of the polygon are solved.

**Program Usage**

This program makes use of Geogebra’s polygon tool and Execute commands to handle input and output. Note that my implementation **will not check** if the polygon is non-self-intersecting or closed (aka simple). This solution also doesn’t consider the case where the polygon has holes.

1. Go to https://www.geogebra.org/classic
2. Use the polygon tool to create your polygon
    a. 

    b. Due to the nature of floating-point arithmetic, this solution may not work for shapes that are very small or thin
3. Click top right menu > View > Spreadsheet
    a. 

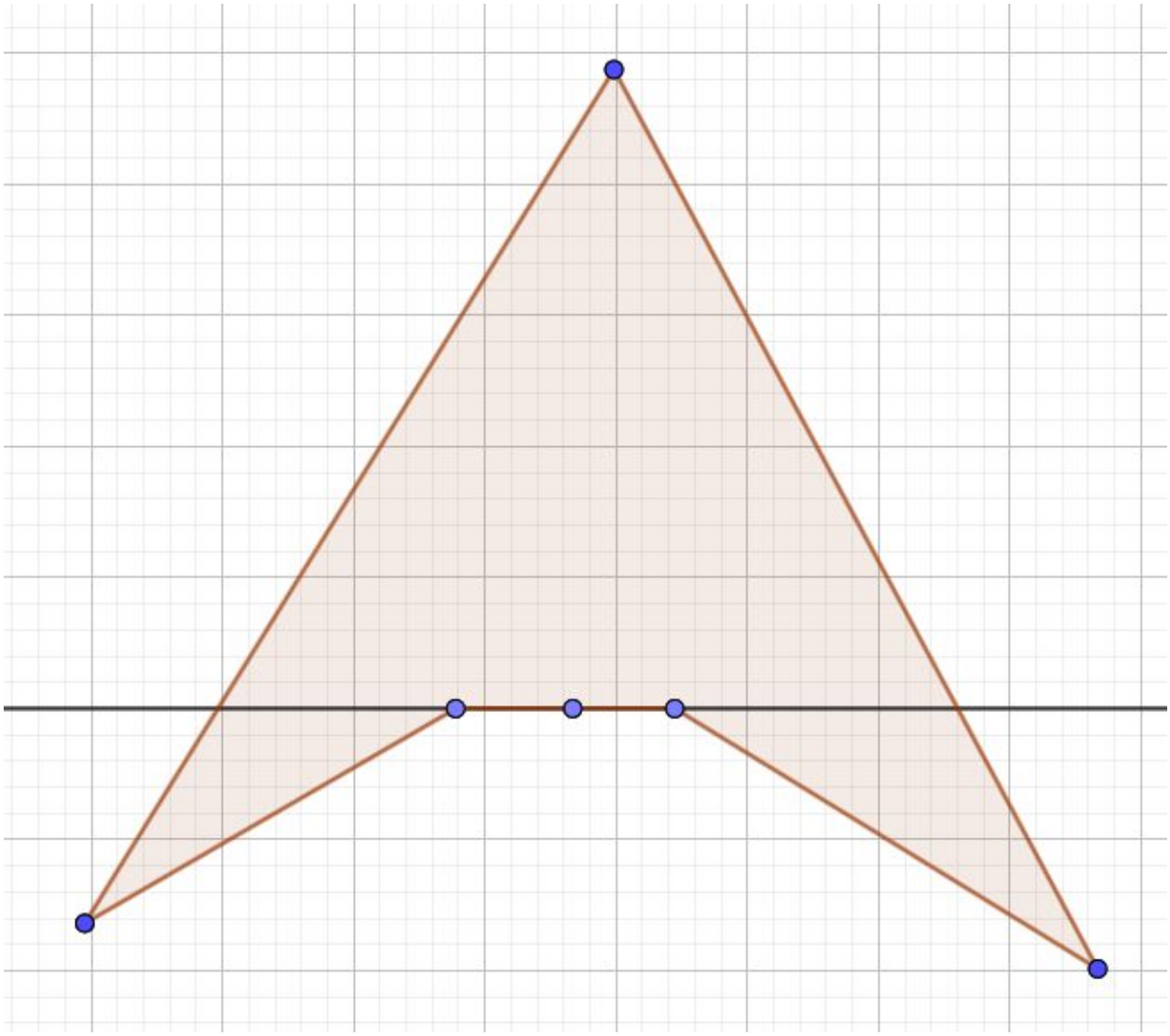
4. In one of the cells, type: **={Vertex(<polygon name>)}**
    a. The first polygon you make should be called “poly1” if the shape has >= 5 sides
        i. If this is the case, then you would type **={Vertex(poly1)}**
    b. Press enter or click off the cell, and you should see a bracket-enclosed list of vertices
5. Run the solving program, and copy the list into it
    a. Using Ctrl+C to copy the list will result in numbers with very high precision, which may be lost as this solution makes calculations with the float type
    b. It is advised to **Right click > Copy** your vertex list instead
        i. You can change the decimal precision of Geogebra by going to the top right menu (where the spreadsheet was) > Settings
6. The program will generate an “output.txt” file which will describe several Geogebra commands to be copied onto the site
    a. Scroll down to the bottom of the sidebar (may sometimes be located on the bottom) where the vertices and polygon are stored and copy each line into the “**Input**” box, and pressing enter:

b. 


c. Note that if there are many vertices in the polygon the command may take a bit to show up

**Solution Gallery**

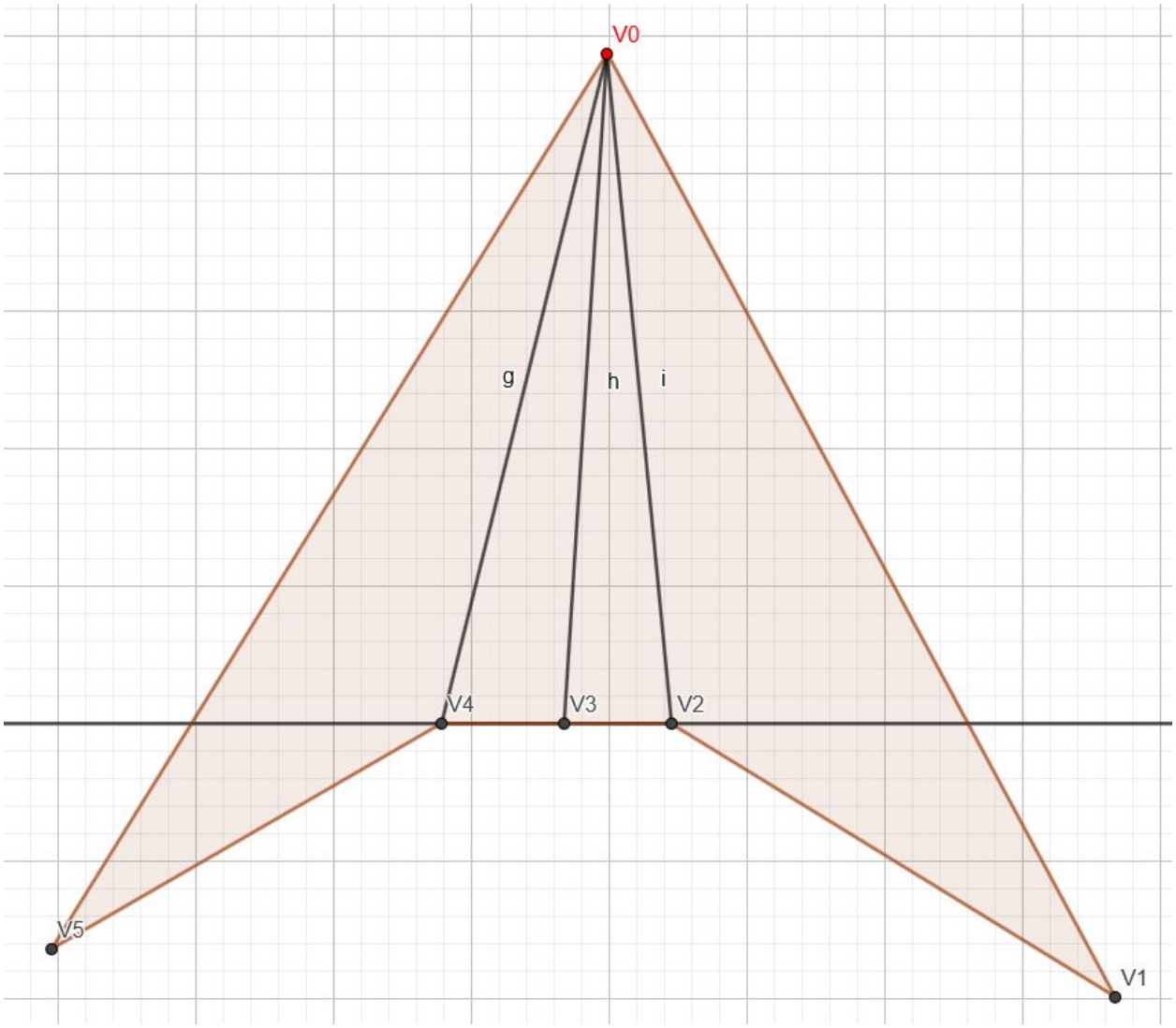
Note that this program will only highlight the solution vertices.



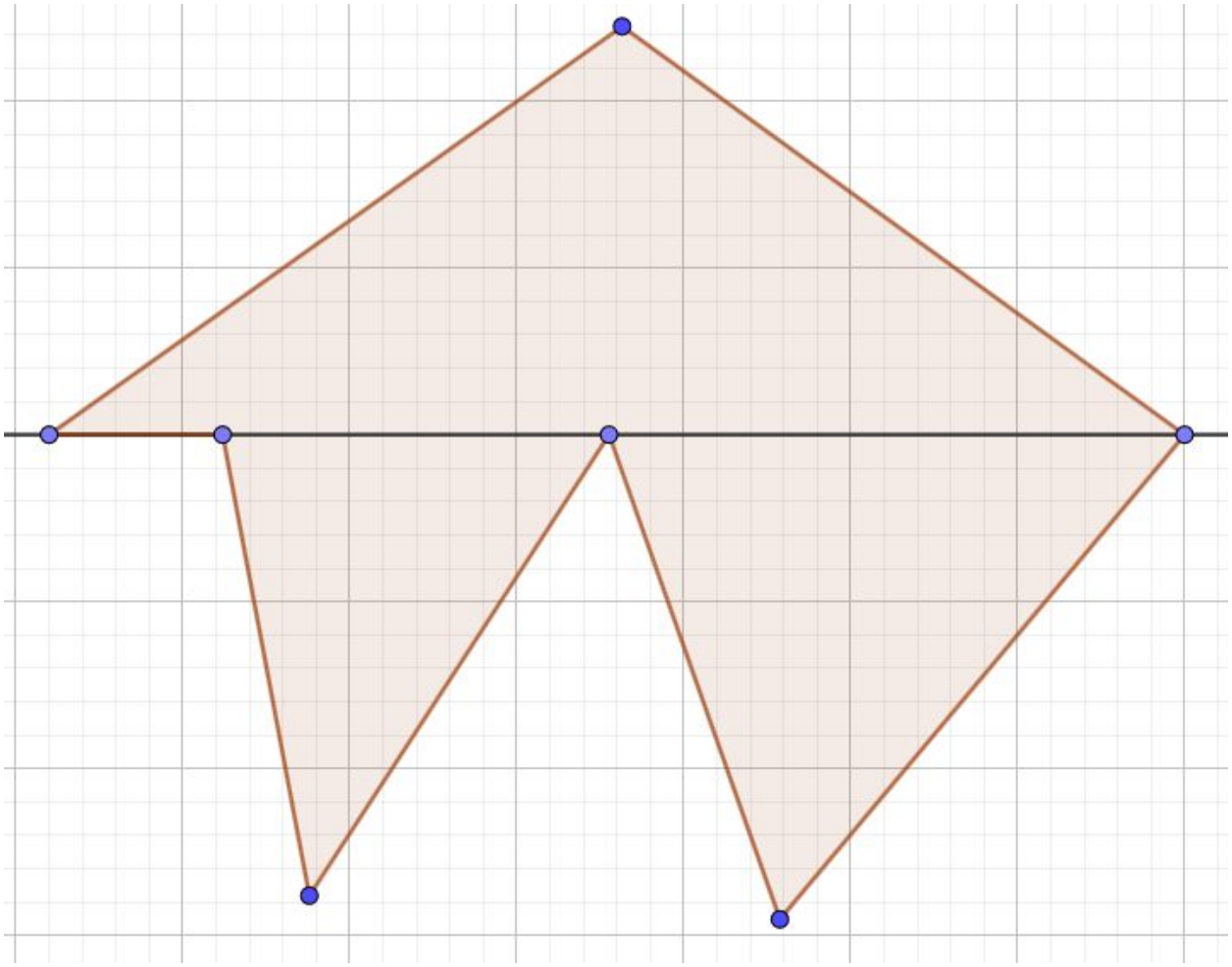
Coordinates list:

{(6.98, 7.87), (10.67, 1.01), (7.45, 3), (6.67, 3), (5.78, 3), (2.95, 1.36)}

Triangulation and solution vertices:

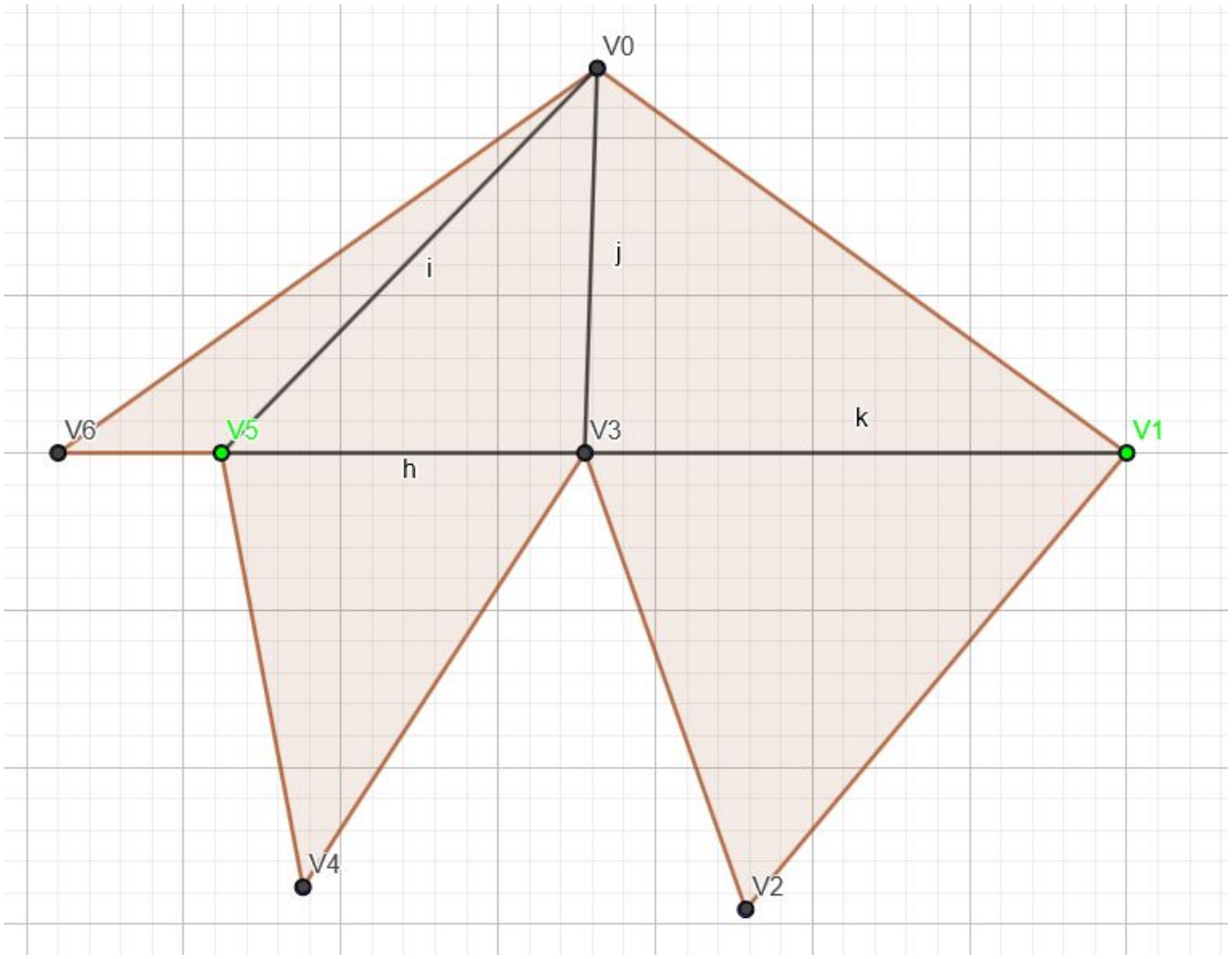



n = 6
Worst case solution = floor(n/3) = 2
Solution vertices = 1

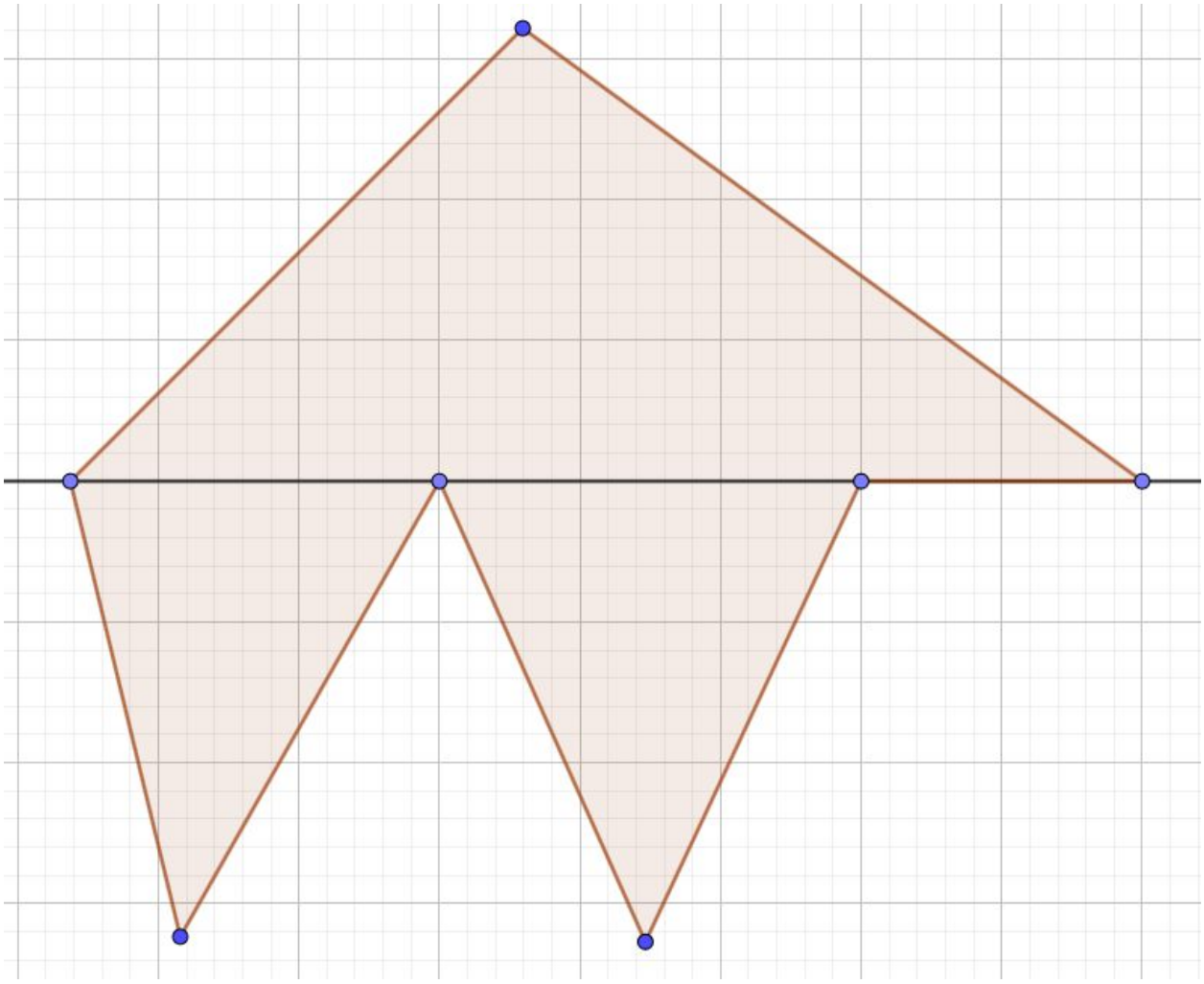


Coordinates list:
{(5.63, 6.45), (9, 4), (6.58, 1.1), (5.55, 4), (3.76, 1.24), (3.24, 4), (2.2, 4)}

Triangulation and solution vertices:

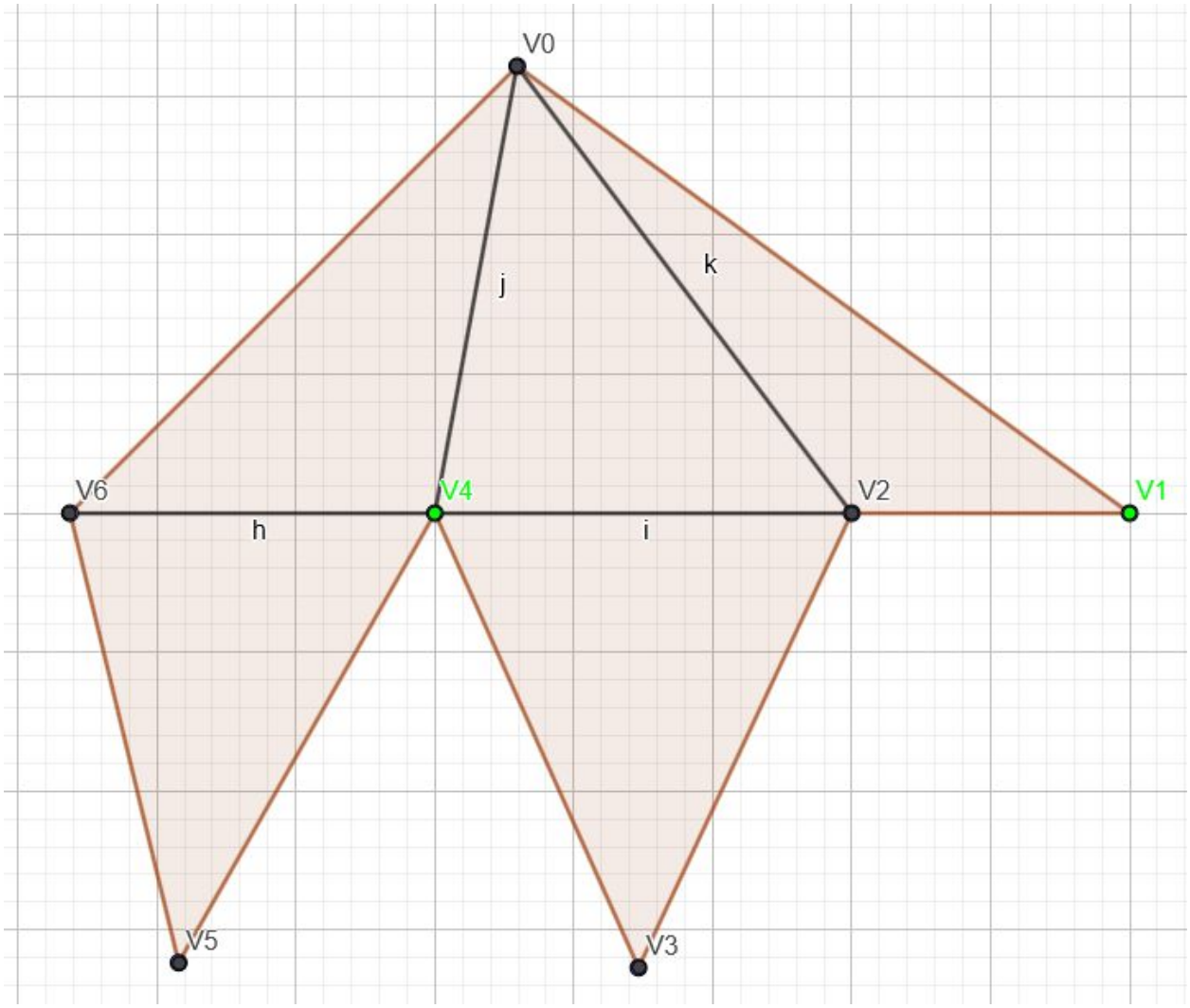



n = 7
Worst case solution = floor(n/3) = 2
Solution vertices = 2

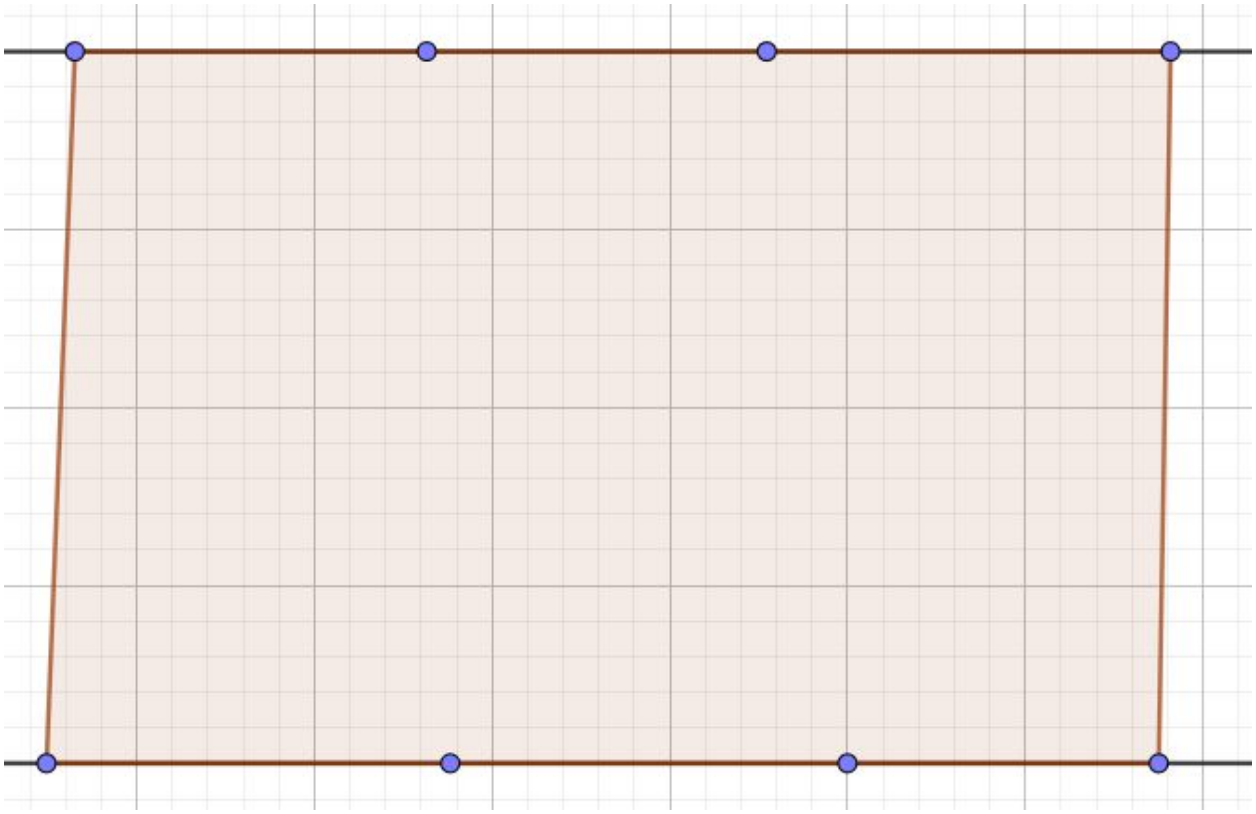


Coordinates list:
{(6.59, 7.22), (11, 4), (9, 4), (7.47, 0.73), (6, 4), (4.16, 0.76), (3.37, 4)}

Triangulation and solution vertices:

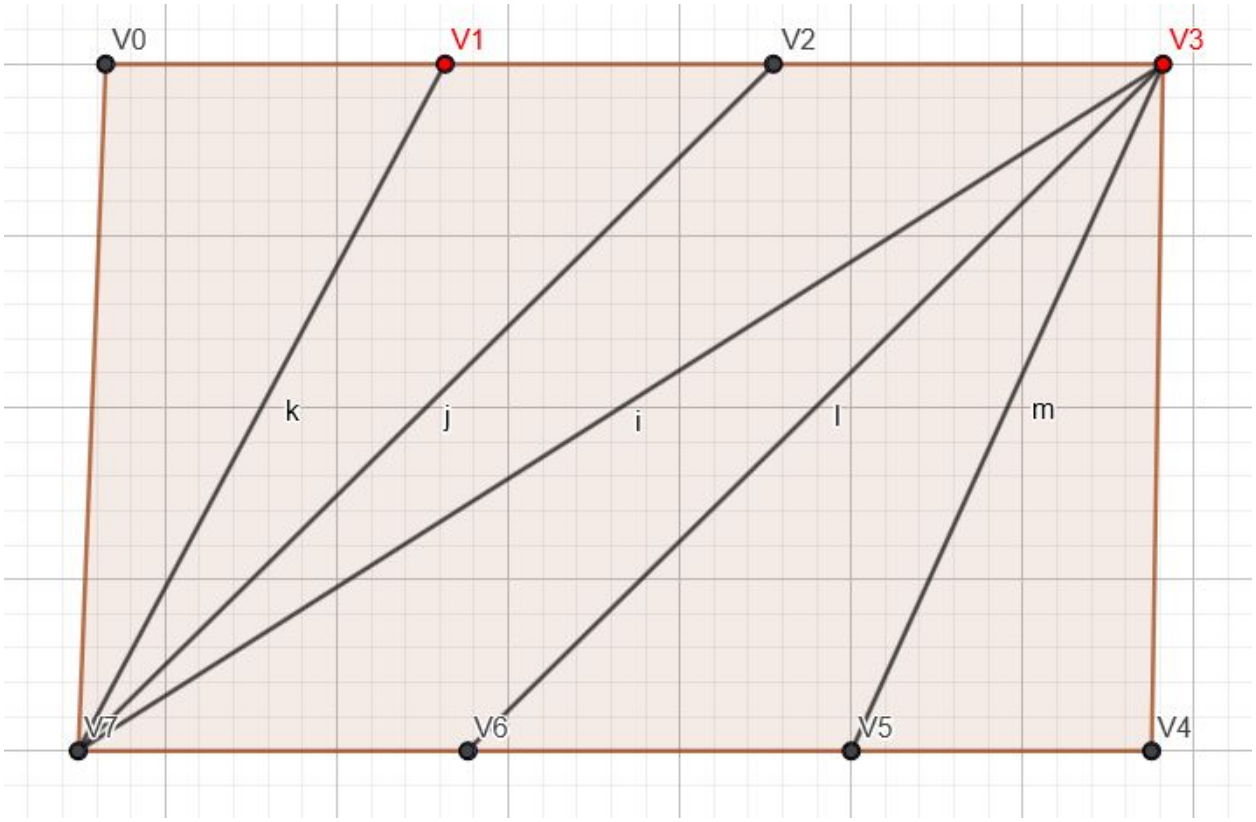



n = 7
Worst case solution = floor(n/3) = 2
Solution vertices = 2

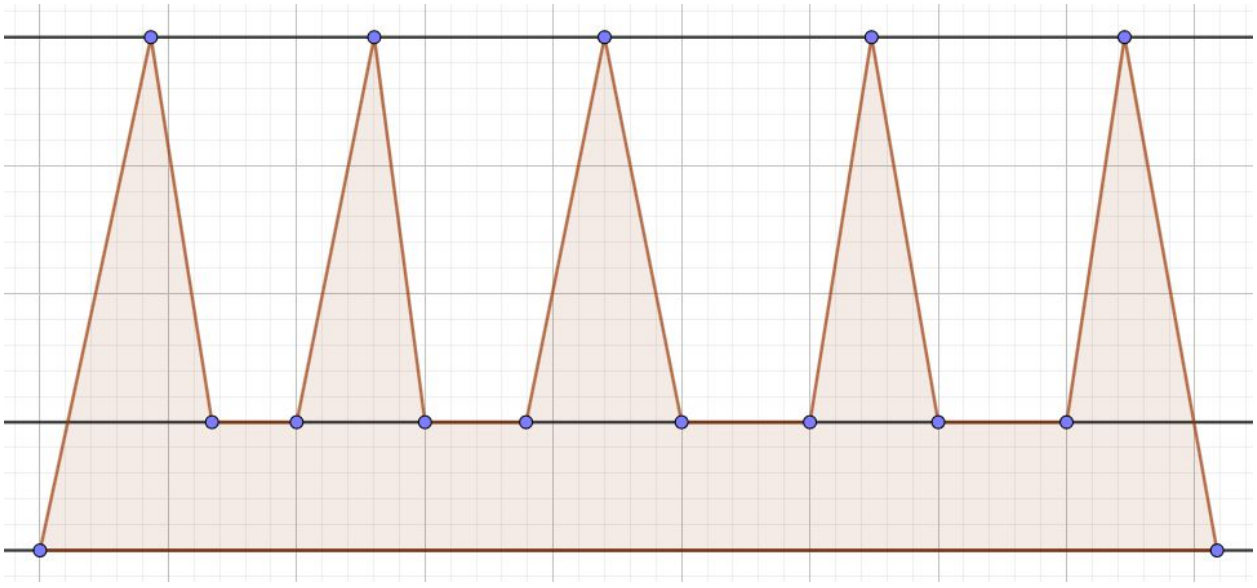


Coordinates list:
{(3.65, 6), (5.63, 6), (7.55, 6), (9.82, 6), (9.75, 2), (8, 2), (5.76, 2), (3.49, 2)}

Triangulation and solution vertices:

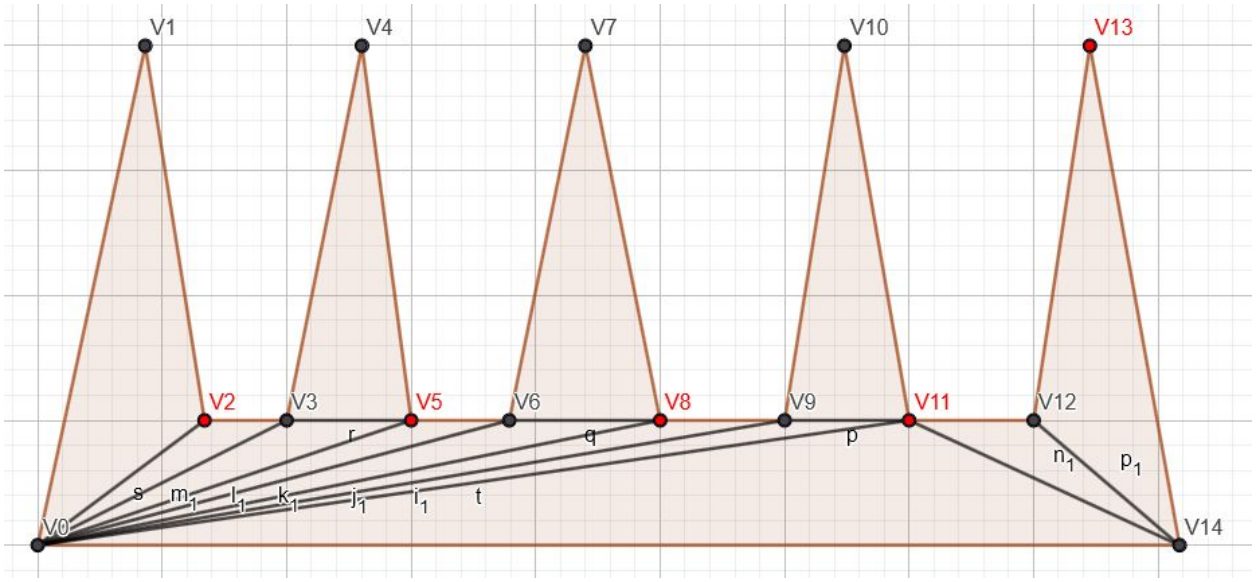



n = 8
Worst case solution = floor(n/3) = 2
Solution vertices = 2

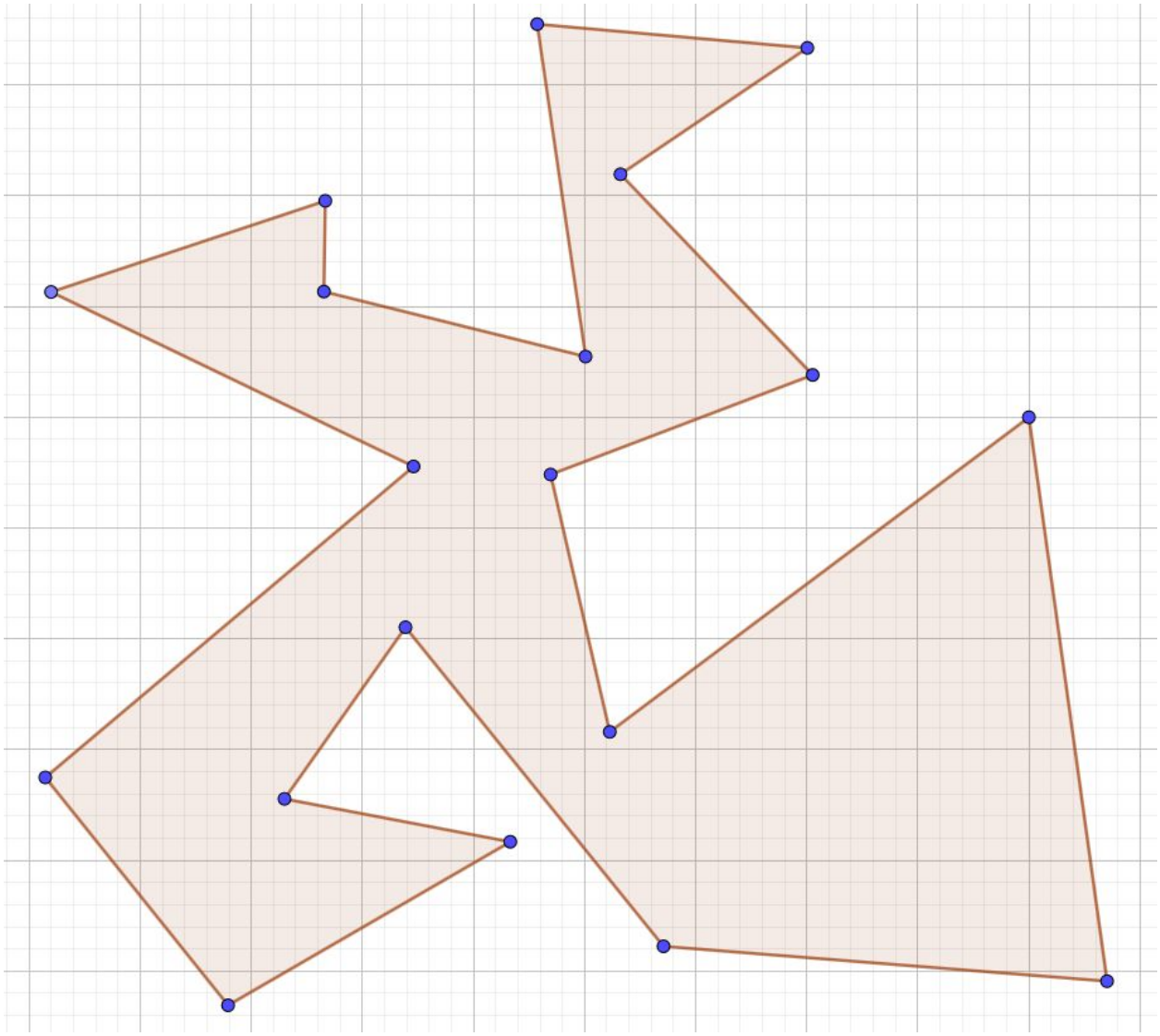


Coordinates list:
{(1, 2), (1.86, 6), (2.34, 3), (3, 3), (3.6, 6), (4, 3), (4.79, 3), (5.4, 6), (6, 3), (7, 3), (7.48, 6), (8, 3), (9, 3), (9.45, 6), (10.17, 2)}

Triangulation and solution vertices:

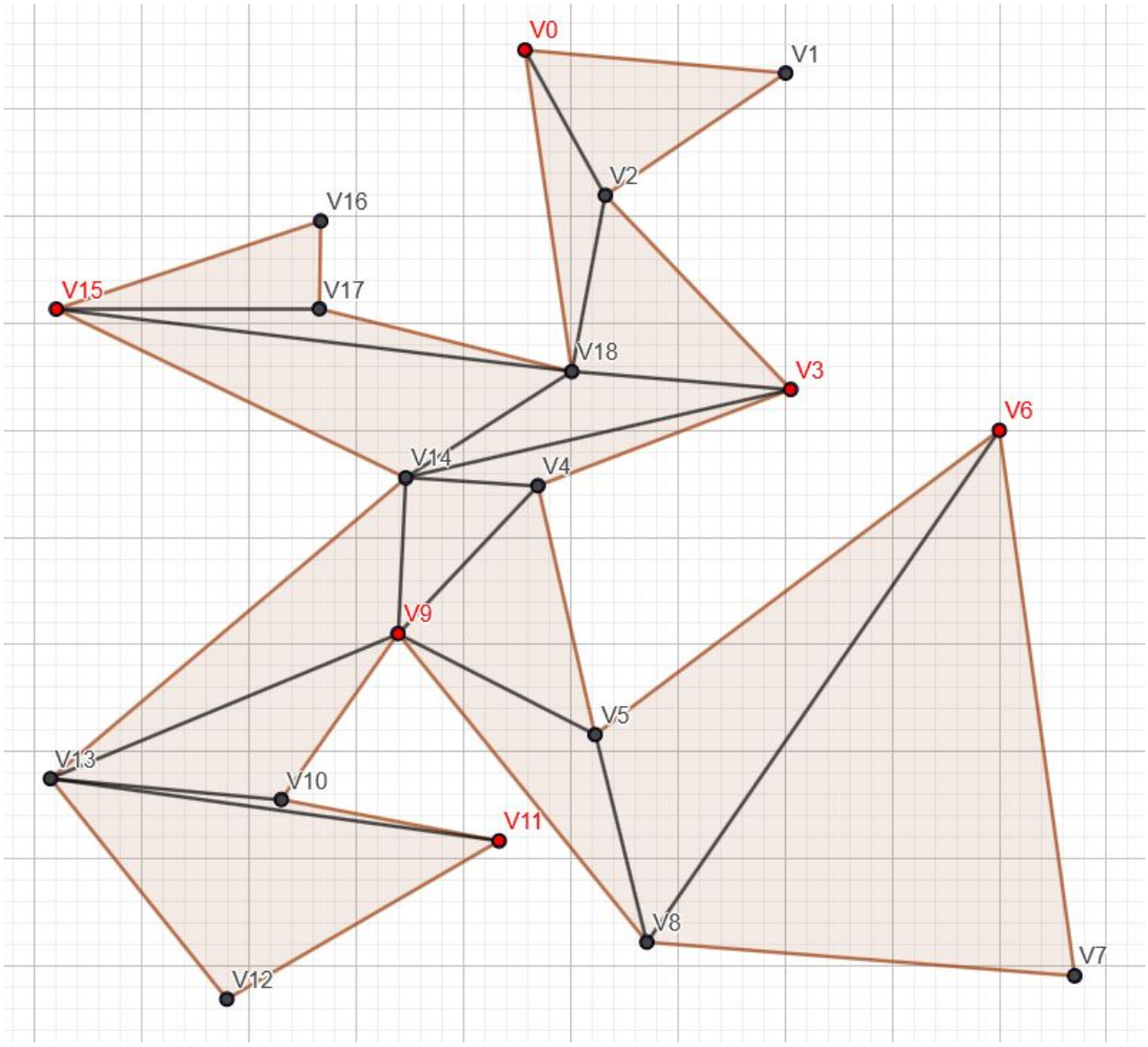



n = 15
Worst case solution = floor(n/3) = 5
Solution vertices = 5
***comb shape always requires worst-case solution

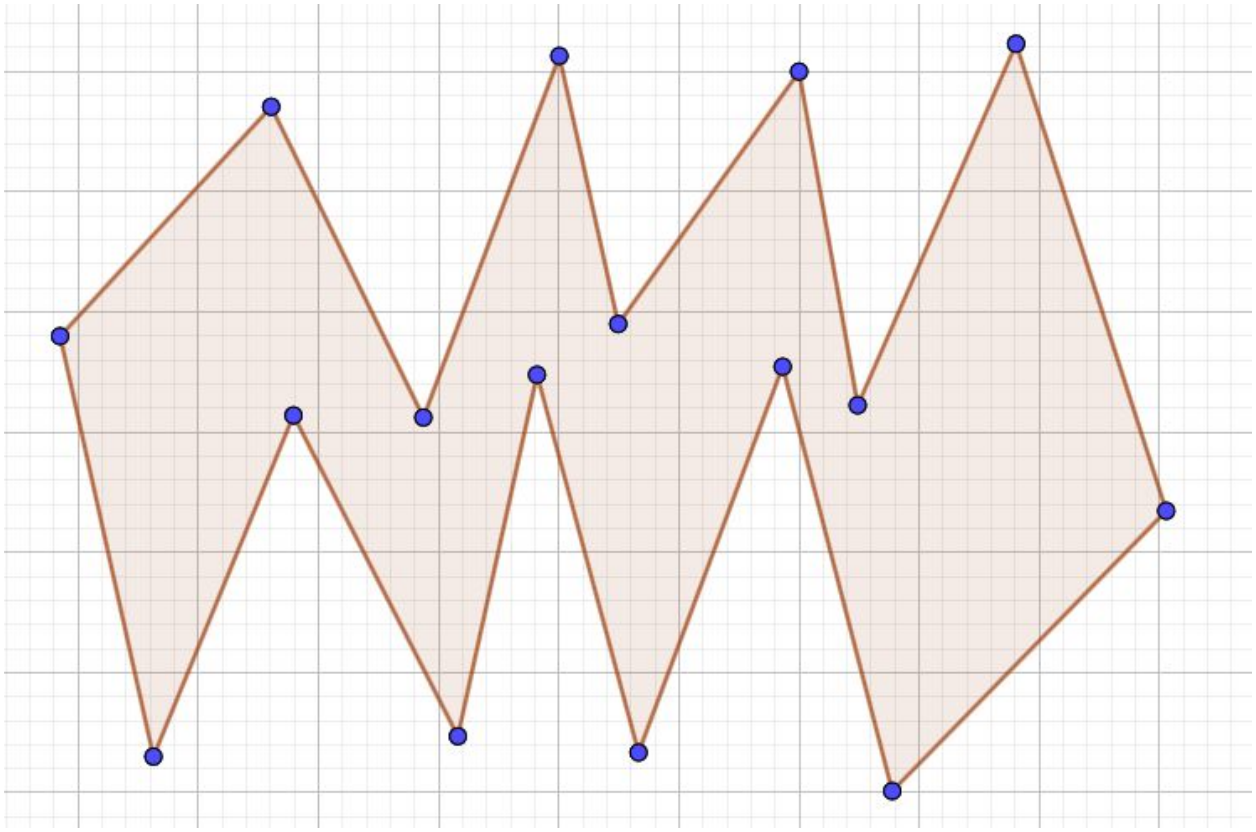


Coordinates list:
{(7.57, 13.55), (10.01, 13.33), (8.32, 12.19), (10.05, 10.38), (7.69, 9.48), (8.23, 7.16), (12, 10), (12.7, 4.91), (8.71, 5.22), (6.39, 8.1), (5.3, 6.55), (7.33, 6.17), (4.79, 4.69), (3.14, 6.75), (6.46, 9.56), (3.2, 11.13), (5.67, 11.95), (5.65, 11.13), (8.01, 10.55)}

Triangulation and solution vertices:

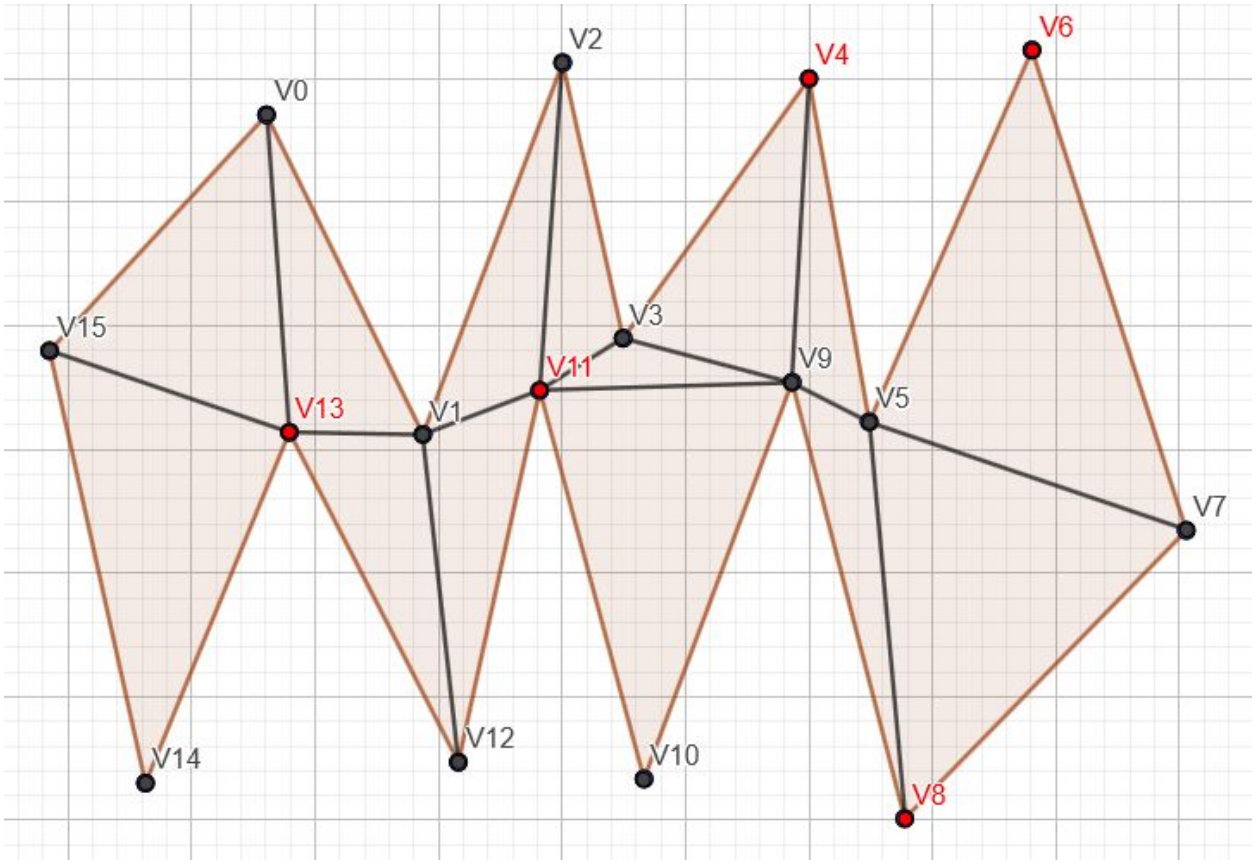



n = 19
Worst case solution = floor(n/3) = 6
Solution vertices = 6



Coordinates list:
{(2.61, 6.71), (3.87, 4.12), (5, 7.13), (5.49, 4.9), (7, 7), (7.49, 4.22), (8.81, 7.23), (10.06, 3.35), (7.78, 1.01), (6.86, 4.54), (5.66, 1.33), (4.82, 4.48), (4.16, 1.47), (2.79, 4.14), (1.63, 1.3), (0.85, 4.8)}

Triangulation and solution vertices:




n = 16
Worst case solution = floor(n/3) = 5
Solution vertices = 5

**References**

[1] - Philipp Kinderman's lecture series on the Art Gallery Problem
[2] - From Figures 2 and 3, accompanying slides to the above lecture
[3] - From Figure 1, overview of the DCEL structure and its components
[4] - Algorithm for correctly connecting all HalfEdges in a DCEL

Many of the helper functions in the program are based off of other sources not mentioned elsewhere in this document. To give proper credit, they will be listed here:

[x] - Detect if a new diagonal will lie inside or outside a polygon
[x] - Pretty much the same content from references [1] and [2], with a little more information added
[x] - Determine a left- or right-hand turn from 3 vectors using cross product
[x] - Determine if two line segments intersect