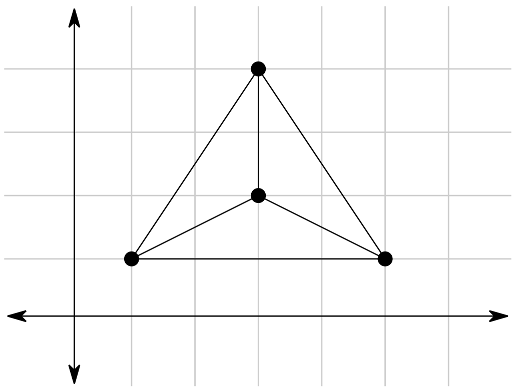
CS133 Lab 9–Voronoi Region

Objective

- In this lab, you will construct parts of the Voronoi diagram given an existing Delaunay triangulation. You will build on and refine the existing DCEL structure that you started in Lab 8. However, it is possible to finish this lab without completing Lab 8 in its entirety.

Detailed Requirements

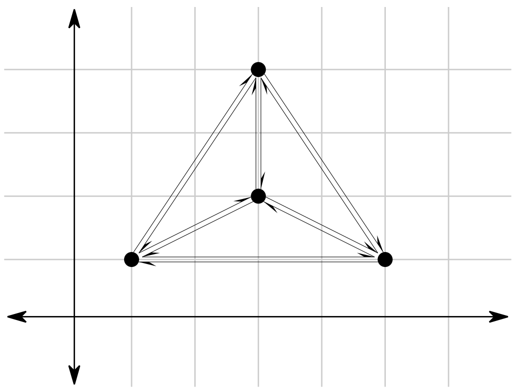
Let us start with the following simple Delaunay triangulation (DT) that contains four sites.



1. Can you identify the vertices, edges, and faces in the above planar graph?

Number of vertices:
Number of edges:
Number of faces:

To represent this DT as a DCEL structure, we need to convert each edge to two twin half edges as shown below.



2. How many vertices, half-edges, and faces does this DCEL structure have?
   Number of vertices:
   Number of half-edges:
   Number of faces:
3. How do these numbers compare to the corresponding planar graph?

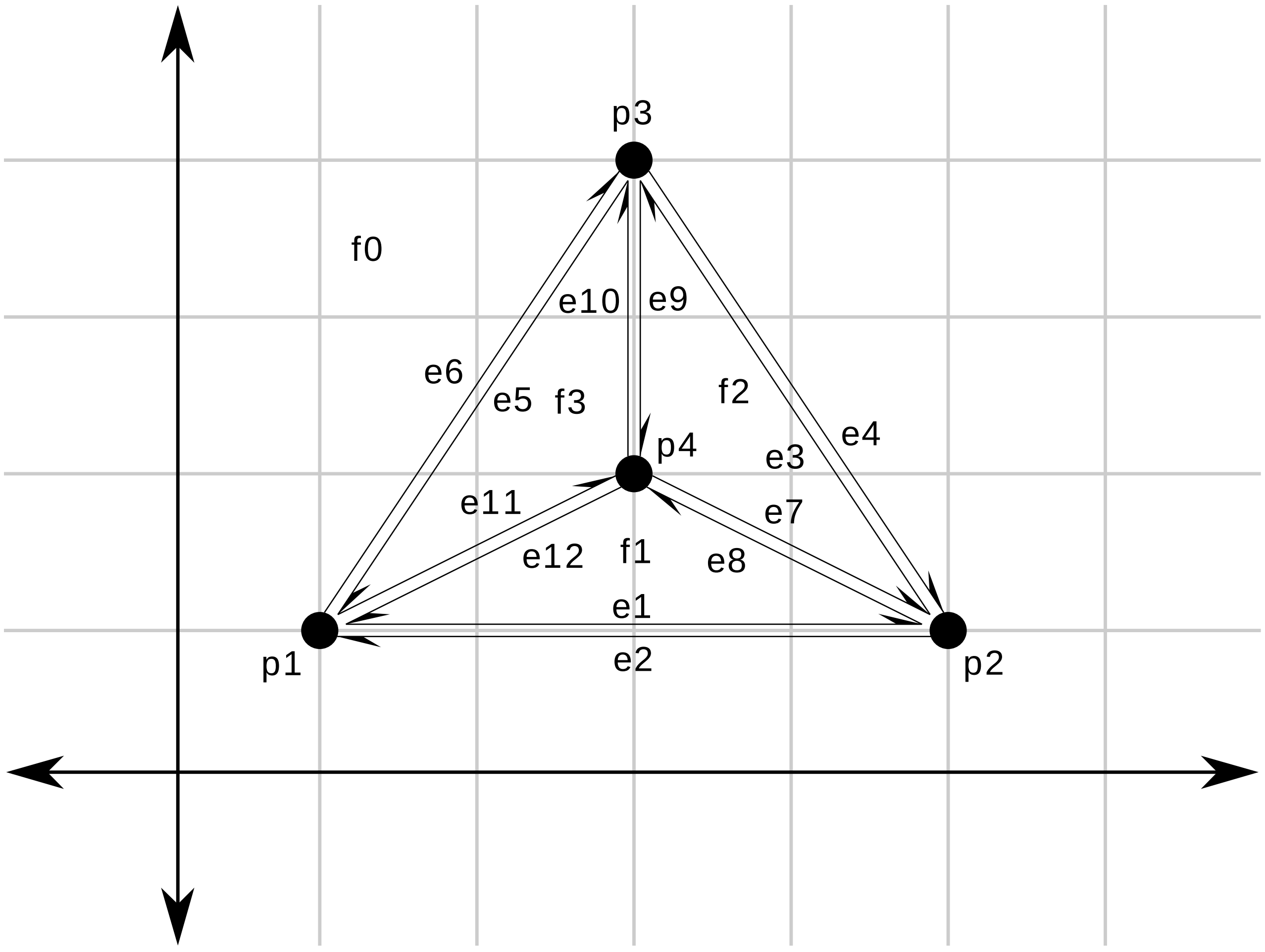
Now, recall the DCEL structure from class and Lab 8.

```
struct Vertex {
  double x, y;
  PEdge leaving;
};
```

```
struct HalfEdge {
  /// The vertex from which the HalfEdge starts
  PVertex origin;
  /// The face on the left side of the HalfEdge
  PFace face;
  /// The HalfEdge that starts from this->twin->origin and ends at the next
vertex in this->face
  PEdge twin;
  /// The next edge in the same face
  PEdge next;
};
```

```
struct Face {
  /// A single HalfEdge that has this Face object as its face
  PEdge edge;
};
```

4. Can you construct a correct DCEL structure for this DT?
   For consistency, use the labels from the figure below and fill in the given tables.



Vertices

Vertex	X	Y	leaving

HalfEdges

HalfEdge	Origin	Face	Twin	Next

Faces

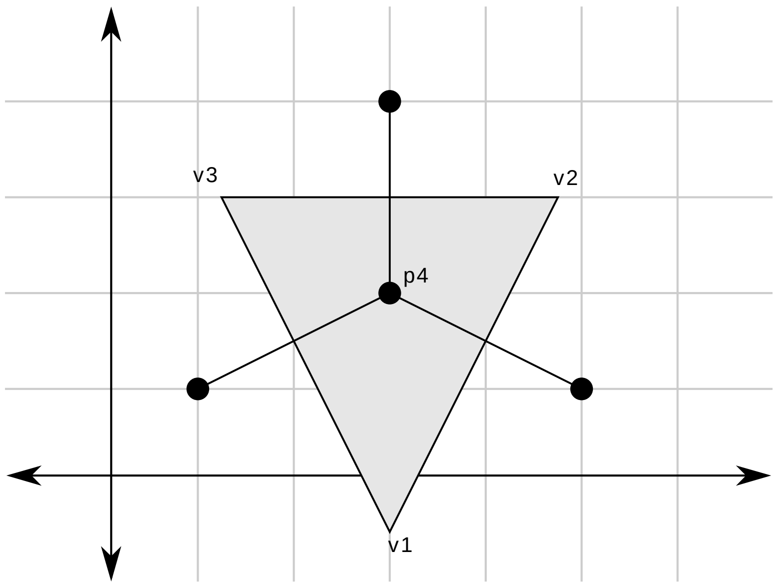
Face	Edge

5. Now, programmatically construct the above DCEL using your C++ code. At this step, you do not have to use the createEdge and splitFace functions if they do not work. Rather, you can manually create all the objects from the table above and fill in their attributes accordingly.
6. Now, try the following functions from your code in Lab 8. For each function, first fill in the expected result based on the DCEL structure. Then, run your function and verify that it gives the expected result. If it does not, this is a good time to debug and fix it.

Function	Expected result	Actual result
e1->destination();	p2	
e9->destination();		
e7->nextLeaving();		
e7->nextLeaving()->nextLeaving();		
e7->nextLeaving()->nextLeaving()->nextLeaving();		
findFaces(p4);		
findFaces(p1);		
isConnected(p1,p4);		
findIncidentEdge(p1,f1);		
findIncidentEdge(p1,f2);		
findIncidentEdge(p1,f0);		

✪ Can you also construct the same DCEL structure using the createVertex, createEdge, and splitFace functions from Lab8? Once you're done with this lab, you can go back and use the same example to debug these three functions from Lab 8.

Now, it is time to reconstruct the Voronoi diagram from the DCEL structure. We will start with the center vertex (p4) as it makes a closed Voronoi region.



7. Which half edges are needed to construct that Voronoi region?

8. Which half edges are need to find the coordinates of the vertex $v1$?

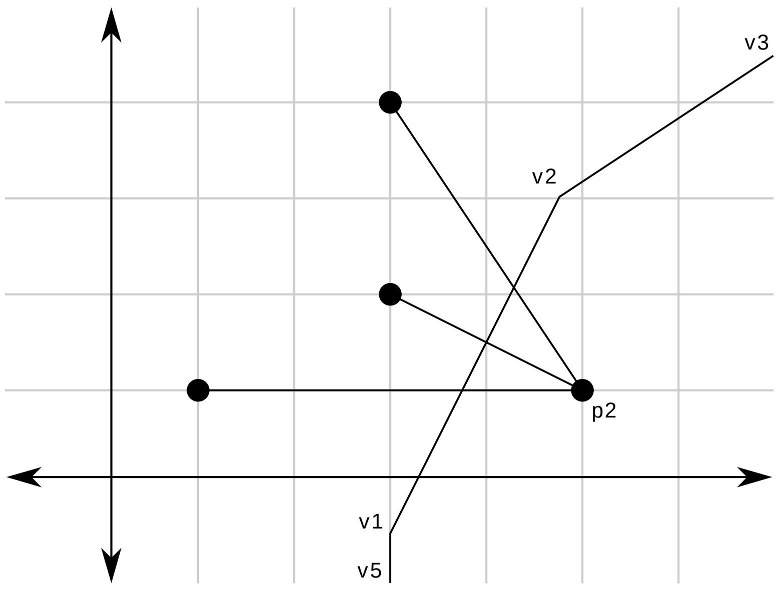
9. Write a function computeVoronoiVertex that computes the location of a vertex given two half edges.

```
Point computeVoronoiVertex(PHalfEdge, PHalfEdge);
```

10. Use the above function to compute the coordinates of the three vertices $v1$, $v2$, and $v3$.

Vertex	x	y
$v1$		
$v2$		
$v3$		

Now, let us compute the Voronoi region for the site $p2$.



Since $p2$ has an unbounded face, we clip it to some boundary rectangle which, in this case, is defined based on the border of the figure.

11. Programmatically compute the coordinates of $v3$ and $v5$. How many half edges are needed to compute each one?

Vertex	x	y
$v3$		
$v5$		

12. Repeat for the other two sites, $p1$ and $p3$. How many additional vertices do you need to compute?
13. Based on this work, write a function computeVoronoiRegion that takes a DT vertex and a clipping rectangle, and returns a list of points that make either a closed polygon for a closed Voronoi region or an open linestring for an unbounded Voronoi region. If the Voronoi region is closed, you do not have to clip it to the clipping rectangle.

```
std::vector<Point> computeVoronoiRegion(PVertex p, std::pair<Point> rectangle);
```