Objective

- In this lab, you will strengthen your DCEL implementation and extend it to prepare for the implementation of either Voronoi Diagram or Delaunay Triangulation.

Detailed Requirements

In Lab 8, you built an initial DCEL implementation with a few construction functions. Before using it to implement a Voronoi Diagram or a Delaunay Triangulation algorithm, you need to make sure it’s ready. Below, is a list of primitive operations in DCEL that you will implement in this lab. In these functions, PVertex, PEdge, and PFace, are pointers to a Vertex, a HalfEdge, or a Face, respectively.

1. PEdge Vertex::nextLeaving(PEdge)
   Given an edge leaving a vertex v, this function returns the next edge leaving the vertex v. Note that in the vertex, we only store one leaving edge. This function should be used to iterate over all leaving edges of a vertex.

2. PEdge::destination() : PVertex
   Returns the destination vertex of a HalfEdge. Note that we only store the source vertex of a HalfEdge. This function is used to return the destination vertex as well.

3. PVertex DCEL::createVertex(double x, double y)
   Creates and returns a new vertex at the given point location.

4. PFace DCEL::getUnboundedFace()
   Returns the single unbounded face in the DCEL.

5. PEdge DCEL::createEdge(PVertex, PVertex)
   Creates an edge, i.e., two half edges, between two existing vertices. One of the two half edges is returned. The other half edge can be found using the twin attribute. Keep in mind that inserting an edge could result in creating a new face.

6. std::vector<PFace> DCEL::findFaces(PVertex)
   Returns all faces that are adjacent to a vertex v.

7. PFace DCEL::findCommonFace(PVertex, PVertex)
   Finds a common face between two vertices. A common face has the two given vertices on its boundary. If the two vertices have more than one common face, any bounded face is returned. That is, if they have two common faces and one of them is the unbounded face, the other one is returned. If more than one bounded face exists, any of them is returned. If no common faces exist, a NULL pointer is returned.

8. bool DCEL::isConnected(PVertex, PVertex)
   Returns true if and only if the two given vertices have a common edge between them.
9. **PEdge DCEL::findIncidentEdge(PVertex v, PFace f)**
   If exists, returns the HalfEdge that has v as its source vertex and f as its face. If such an edge does not exist, a NULL pointer is returned.

10. **bool DCEL::deleteEdge(PVertex, PVertex)**
    Deletes an existing edge between two vertices. If an edge is successfully deleted, a true is returned. Otherwise, if the given two vertices are not connected, a false is returned. Keep in mind that deleting an edge might result in merging two faces together.

11. **DCEL mergeAndDestroy(DCEL& d1, DCEL& d2)**
    Given two DCEL structures, this function combines them together into a new DCEL. The returned DCEL combines all the vertices, edges, and faces in both DCELs. Except for the unbounded face, it is assumed that all vertices, edges, and faces in both DCELs are different. For efficiency purposes, the two input DCELs are destroyed as a result of calling this function.

Feel free to add more supporting functions as you need.

**Example**

Below is a simple example that you can use to test your implementation. It starts with a simple empty DCEL and manipulates it using the functions given above. You should add more sophisticated test cases to ensure the correctness of your implementation.

```cpp
def dcel:
    PVertex v1 = dcel.createVertex(1, 0);
    PVertex v2 = dcel.createVertex(2, 1);
    PVertex v3 = dcel.createVertex(0, 1);
    PVertex v4 = dcel.createVertex(0, 0);
    // v1 has one face (the unbounded face)
    assert(dcel.findFaces(v1).size() == 1);

    PEdge e1 = dcel.createEdge(v1, v2);
    // v1 still has one face which is not NULL
    assert(dcel.findFaces(v1).size() == 1);
    assert(dcel.findFaces(v1).front() != NULL);
    // v2 also has one unbounded face
    assert(dcel.findFaces(v2).size() == 1);
    // All vertices have one common face which is the unbounded face
    assert(dcel.findCommonFace(v1, v2) != NULL);
    assert(dcel.findCommonFace(v1, v3) != NULL);
    // Find the newly created edge using the findIncidentEdge function
    assert(dcel.findIncidentEdge(e1->origin, e1->face) == e1);
    dcel.createEdge(v2, v3);
    // v1 and v2 are still connected
    assert(dcel.isConnected(v1, v2));
    // v1 and v3 are not connected (i.e., not adjacent)
    assert(!dcel.isConnected(v1, v3));
    // Create two new edges to create the first face
```

dcel.createEdge(v3, v4);
dcel.createEdge(v4, v1);
// Now there are two faces, the newly created face and the unbounded face
assert(dcel.getFaceCount() == 2);
// All the four vertices are adjacent to the two faces
assert(dcel.findFaces(v1).size() == 2);
assert(dcel.findFaces(v2).size() == 2);
assert(dcel.findFaces(v3).size() == 2);
assert(dcel.findFaces(v4).size() == 2);
// v1 and v2 have two common faces, but the bounded face should be returned
assert(dcel.findCommonFace(v1, v2) != dcel.getUnboundedFace());
assert(dcel.findCommonFace(v1, v2)->getBoundary().size() == 4);
assert(dcel.getUnboundedFace()->getAdjacentFaces().size() == 1);
// Create a new edge that will result in a new face
dcel.createEdge(v4, v2);
assert(dcel.getFaceCount() == 3);
assert(dcel.findFaces(v4).size() == 3);
// Both bounded faces are adjacent to the unbounded face
assert(dcel.getUnboundedFace()->getAdjacentFaces().size() == 2);
deleteEdge(v2, v4);
assert(dcel.getFaceCount() == 2);