CSN-523

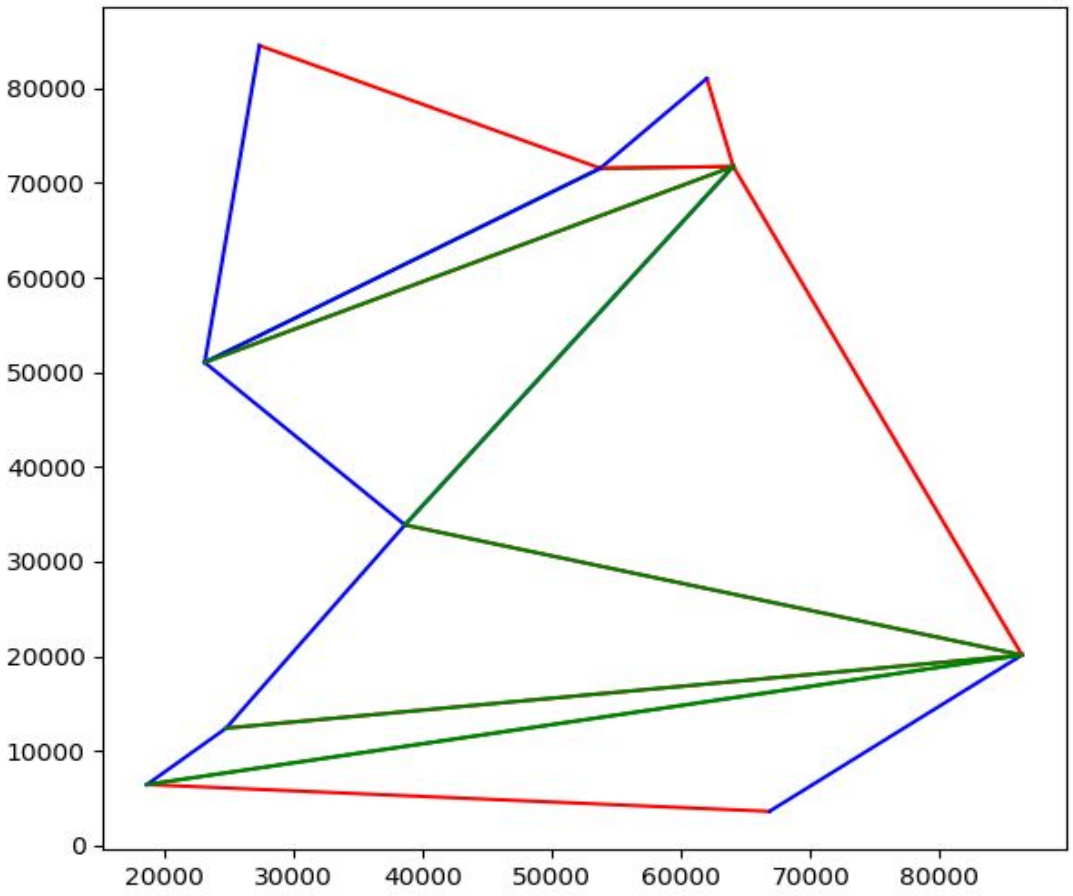
Computational Geometry

CODING PROJECT-2

GroupID: 26

Ambar Zaidi 14114009

Tarun Kumar 14114068



Objective

To know implementation of different Computational Geometry algorithms for solving the Art-Gallery Problem.

Problem Statement of CP-2:

- **Construct a simple polygon with n vertices. Your program should be able to take n as input (e.g., 20 or 100) from the user, then randomly generate n distinct points with x and y coordinates in 2D geometry.**

  Store the simple polygon using the required data structure DCEL.

Method

- Take the input value of ‘n’
- Generate ‘n’ random points in cartesian plane
- Generate a random simple polygon with ‘n’ vertices.
    - We sort the points from the bottom most point based on the angle they make.
- Store it as DCEL

Time Complexity = **O(n log n)** because of sorting the points.

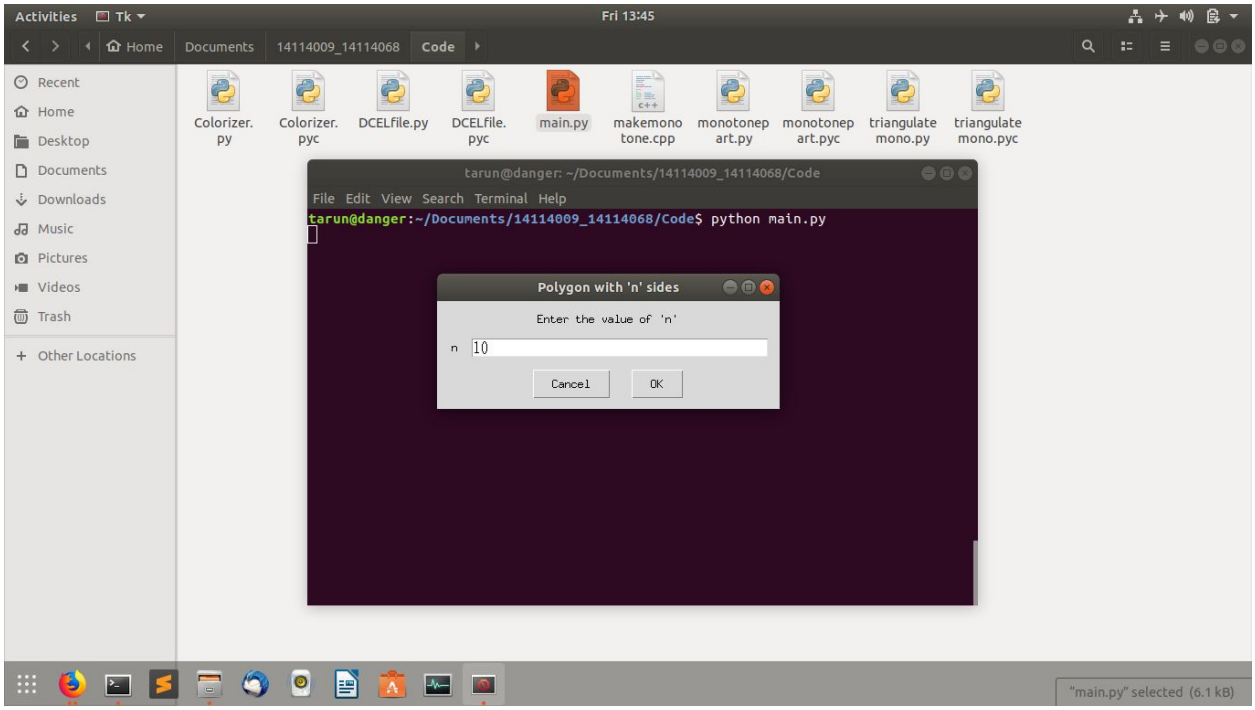


Fig: Execution of program

DCEL Data Structure implementation

Classes created:

- Point
- Edge
- Face
- DCEL

Classes also contain some helper functions.

buildSimplePolygon() can be used to create a polygon from a given polygon boundary,

Note: To start our coding, a stub from Saint Louis University has been adapted for DCEL.

```
class Point:
	def __init__(self, coordinates,auxData=None):
		self.data=auxData
		self.coords = coordinates
		self.edge = None
		self.ear = False
		self.next = None
		self.prev = None
		self.color= -1
	def ___str___(self):
		return str(self.ID)
	def __getitem__(self,key):
		return self.coords[key]
	def scale(self, k1, k2):
		self.coords = list(self.coords)
		self.coords[0] = int(self.coords[0] * k1)
		self.coords[1] = int(self.coords[1] * k2)
		self.coords = tuple(self.coords)
	def __hash__(self):
		return hash(id(self))
	def getData(self):
		return self.data
	def setData(self, auxData):
		self.data = auxData
	def getCoords(self):
		return Point(self.coords)
	def setCoords(self):
		self.coords = coordinates
	def getOutgoingEdges(self):
		visited = set()
		out = []
		here = self.edge
		while here and here not in visited:
			out.append(here)
			visited.add(here)
			temp = here.getTwin()
			if temp:
			here = temp.getNext()
			else:
			here = None
		return out
	def getIncidentEdge(self):
		return self.edge
	def setIncidentEdge(self, edge):
		self.edge = edge
	def __repr__(self):
		return 'DCEL.Point with coordnates (' +
	str(self.coords[0])+','+str(self.coords[1])+')'
class Edge:
	def __init__(self, auxData=None):
		self.data = auxData
		self.twin = None
		self.origin = None
```

```
                self.face = None
                self.next = None
                self.prev = None
        def __hash__(self):
                return hash(id(self))
        def getTwin(self):
                return self.twin
        def setTwin(self, twin):
                self.twin = twin
        def getData(self):
                return self.data
        def setData(self, auxData):
                self.data = auxData
        def getNext(self):
                return self.next
        def setNext(self, edge):
                self.next = edge
        def getOrigin(self):
                return self.origin
        def setOrigin(self, v):
                self.origin = v
        def getPrev(self):
                return self.prev
        def setPrev(self, edge):
                self.prev = edge
        def getDest(self):
                return self.twin.origin
        def getFace(self):
                return self.face
        def getFaceBoundary(self):
                visited = set()
                bound = []
                here = self
                while here and here not in visited:
                        bound.append(here)
                        visited.add(here)
                        here = here.getNext()
                return bound
        def setFace(self, face):
                self.face = face
        def clone(self):
                c = Edge()
                c.data,c.twin,c.origin,c.face,c.next,c.prev =
        self.data,self.twin,self.origin,self.face,self.next,self.prev
        def __repr__(self):
                return 'DCEL.Edge from Origin: DCEL.Point with coordinates (' +
        str(self.getOrigin().coords[0])+','+str(self.getOrigin().coords[1])+')' +
        '\nDestination: DCEL.Point with coordinates (' +
        str(self.getDest().coords[0])+','+str(self.getDest().coords[1])+')'
class Face:
        def __init__(self, auxData=None):
                self.data = auxData
                self.outer = None
```

```
                self.inner = set()
                self.isolated = set()
        def __hash__(self):
                return hash(id(self))
        def getOuterComponent(self):
                return self.outer
        def setOuterComponent(self, edge):
                self.outer = edge
        def getData(self):
                return self.data
        def setData(self, auxData):
                self.data = auxData
        def getOuterBoundary(self):
                if self.outer:
                        return self.outer.getFaceBoundary()
                else:
                        return []
        def getOuterBoundaryCoords(self):
                original_pts = self.getOuterBoundary()
                return [x.origin.coords for x in original_pts]
        def getInnerComponents(self):
                return list(self.inner)
        def addInnerComponent(self, edge):
                self.inner.add(edge)
        def removeInnerComponent(self, edge):
                self.inner.discard(edge)
        def removeIsolatedVertex(self,Point):
                self.isolated.discard(Point)
        def getIsolatedVertices(self):
                return list(self.isolated)
        def addIsolatedVertex(self,Point):
                self.isolated.add(Point)
class DCEL:
        def __init__(self):
                self.exterior = Face()
        def getExteriorFace(self):
                return self.exterior
        def getFaces(self):
                result = []
                known = set()
                temp = []
                temp.append(self.exterior)
                known.add(self.exterior)
                while temp:
                        f = temp.pop(0)
                        result.append(f)
                        for e in f.getOuterBoundary():
                        nb = e.getTwin().getFace()
                        if nb and nb not in known:
                                known.add(nb)
                                temp.append(nb)
                        for inner in f.getInnerComponents():
                        for e in inner.getFaceBoundary():
```

```
                        nb = e.getTwin().getFace()
                        if nb and nb not in known:
                                known.add(nb)
                                temp.append(nb)
                return result
        def getEdges(self):
                edges = set()
                for f in self.getFaces():
                        edges.update(f.getOuterBoundary())
                        for inner in f.getInnerComponents():
                        edges.update(inner.getFaceBoundary())
                return edges
        def getVertices(self):
                verts = set()
                for f in self.getFaces():
                        verts.update(f.getIsolatedVertices())
                        verts.update([e.getOrigin() for e in f.getOuterBoundary()])
                        for inner in f.getInnerComponents():
                        verts.update([e.getOrigin() for e in inner.getFaceBoundary()])
                return verts

def buildSimplePolygon(points):
        d = DCEL()
        if points:
                exterior = d.getExteriorFace()
                interior = Face()
                verts = []
                for p in points:
                        verts.append(Point(p))
                innerEdges = []
                outerEdges = []
                for i in range(len(verts)):
                        e = Edge()
                        e.setOrigin(verts[i])
                        verts[i].setIncidentEdge(e)
                        e.setFace(interior)
                        t = Edge()
                        t.setOrigin(verts[(i+1)%len(verts)])
                        t.setFace(exterior)
                        t.setTwin(e)
                        e.setTwin(t)
                        innerEdges.append(e)
                        outerEdges.append(t)

                for i in range(len(verts)):
                        innerEdges[i].setNext(innerEdges[(i+1)%len(verts)])
                        innerEdges[i].setPrev(innerEdges[i-1])
                        outerEdges[i].setNext(outerEdges[i-1])
                        outerEdges[i].setPrev(outerEdges[(i+1)%len(verts)])
                interior.setOuterComponent(innerEdges[0])
                exterior.addInnerComponent(outerEdges[0])
        return d
```

Screenshots:

I. N = 20

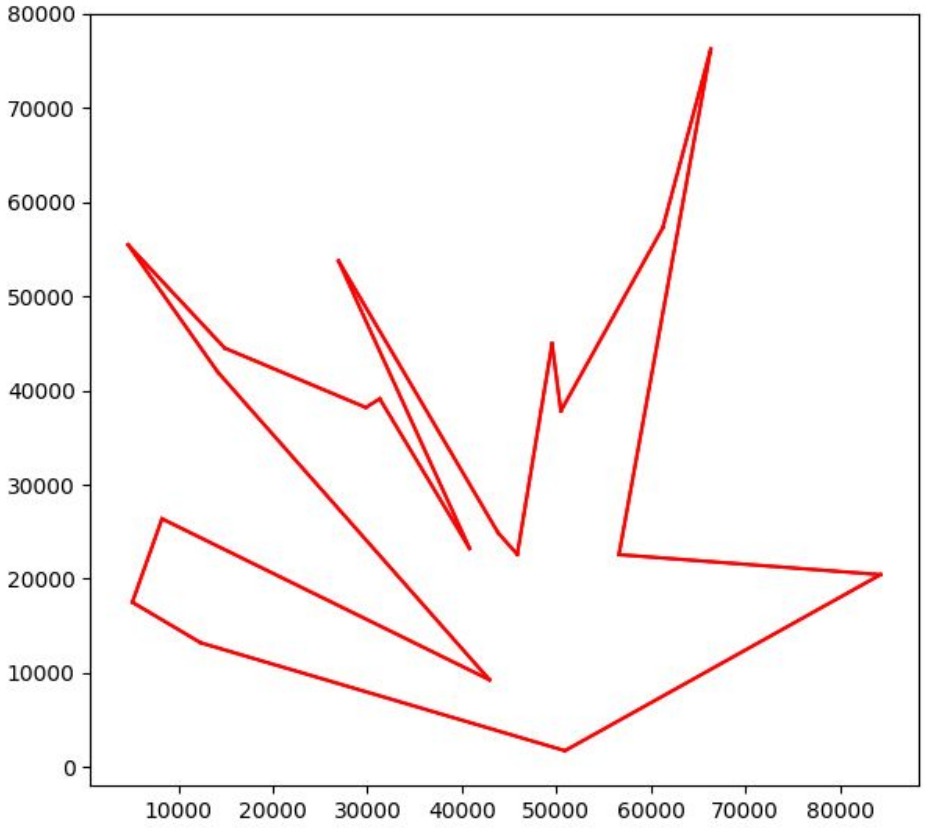


Fig: Polygon with N=20

N = 100

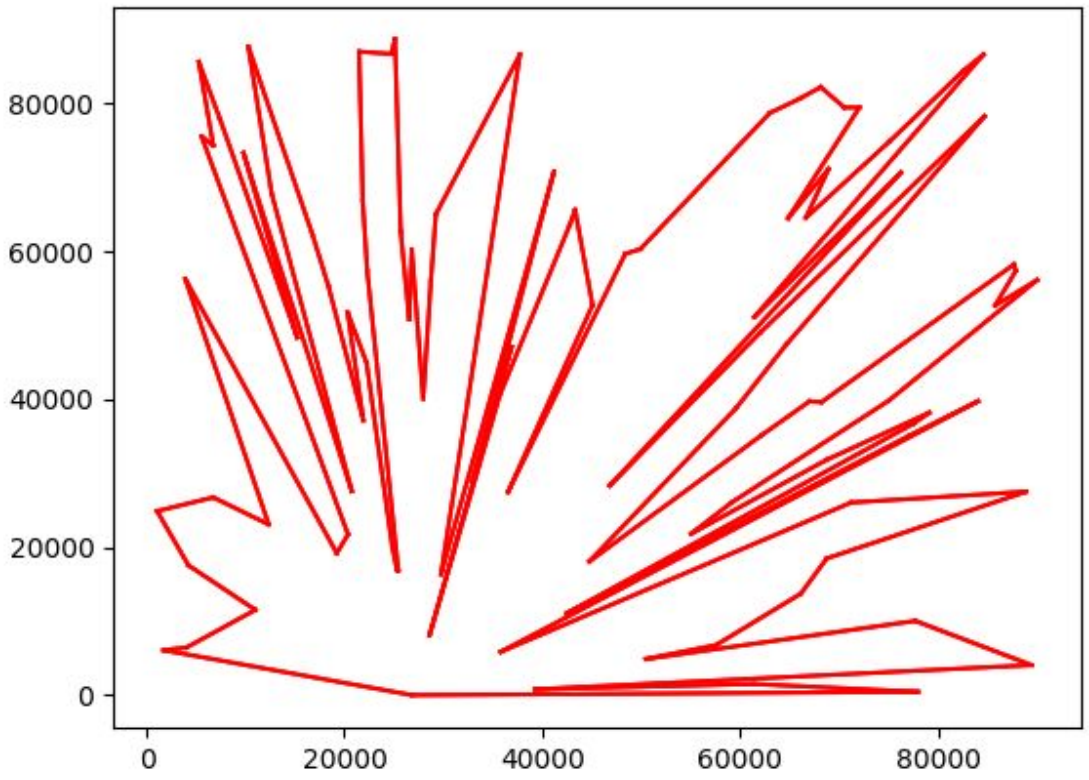


Fig: Polygon with N=100

- **Perform Trapezoidalization of the simple polygon you obtained. Store the information in the required data structure. Obtain monotone partitions from the trapezoidalization of the simple polygon. Implement the line-sweep (plane-sweep) algorithm to obtain monotone partitions of the n-gon.**

Method

- First we find intersection points of the lines parallel to x-axis passing through each vertex with the polygon edges. To do this, for each parallel line we first find the intersection on the left and on the right of the vertex.
    - If it is start or end vertex to nothing. If it is split vertex or merge vertex draw edge from immediate left to immediate right intersection.
    - If passing vertex draw edge from this vertex to immediate left or right intersection point (depending on the chain).
- For each of the vertex we stored the left and right edge (dictionary). And for each edge we stored the vertices corresponding to the intersections on that edge.
- Then for each vertex:
    - If it is on left chain we find the vertex with just smaller y coordinate and not on left chain and join these two vertex with diagonal. (similar to helper function).
    - Similarly for vertex on left chain.
    - If it is merge vertex, joining it with next split, pass,end or merge vertex (whichever has greatest y less than this vertex)
    - If it is split vertex, joining it with next split, merge or pass vertex.
- The figure we get is Monotone partitioned.

Complexity: O(n log n)

Note: *Our approach will give monotone mountains (as it is visible from Trapezoid diagonals added). Still our approach will be give correct results as monotone mountains are also monotone partitions. If we strictly want minimum number of monotone partitions, then we can modify monotonePartitioningDgnls() function.*

```
# divide polygon in two // insert diagonal
def insertDgnl(d, p1, p2):
      if DEBUG:
      print "Inserting diagonal: ",p1,p2
      pointlist1 = []
      pointlist2 = []
      original_pts = d.getFaces()[1].getOuterBoundaryCoords()
      if (p1 in original_pts and p2 in original_pts) and p1!=p2:
      tmp1 = min(original_pts.index(p1), original_pts.index(p2))
      tmp2 = max(original_pts.index(p1), original_pts.index(p2))
      pointlist1 = original_pts[tmp1:(tmp2+1)]
      pointlist2 = original_pts[tmp2:]+original_pts[:(tmp1+1)]
      d1 = buildSimplePolygon(pointlist1)
      d2 = buildSimplePolygon(pointlist2)
      return [d1,d2]
      return [d]

# divide polygon in many // insert list of diagonals
def insertDgnls(d, dgnls):
      ngons = [d]
      while dgnls != []:
      nxt = dgnls.pop(0)
      print "Current dgnl:",nxt
      ngons = [insertDgnl(x, nxt[0], nxt[1]) for x in ngons]
      ngons = [ngon for lngon in ngons for ngon in lngon]
      print len(ngons)
      return ngons

class trapEdge(object):
      def __init__(self,a,b,s,l,r):
      self.left = a
      self.right = b
      self.pivot = s
      self.le = l
      self.re = r

class point(object):
      def __init__(self, a, b):
      self.x = a
      self.y = b

def onSegment(p,q,r):
      if (q.x <= max(p.x, r.x) and q.x >= min(p.x, r.x) and q.y <= max(p.y,
r.y) and q.y >= min(p.y, r.y)):
      return True
      return False
def orientation(p,q,r):
      val = (q.y - p.y) * (r.x - q.x) -(q.x - p.x) * (r.y - q.y)
      if (val == 0):
      return 0
      if(val>0):
      return 1
      return 2
```

```
def doIntersect(p1,q1,p2,q2):
      o1 = orientation(p1, q1, p2);
      o2 = orientation(p1, q1, q2);
      o3 = orientation(p2, q2, p1);
      o4 = orientation(p2, q2, q1);
      A,B,C,D = p1,q1,p2,q2
      a1 = B.y - A.y
      b1 = A.x - B.x
      c1 = a1*(A.x) + b1*(A.y)
      a2 = D.y - C.y
      b2 = C.x - D.x
      c2 = a2*(C.x)+ b2*(C.y)
      determinant = a1*b2 - a2*b1
      if(determinant == 0):
      return False
      if (o1 != o2 and o3 != o4):
      return True
      if (o1 == 0 and onSegment(p1, p2, q1)):
      return True
      if (o2 == 0 and onSegment(p1, q2, q1)):
      return True
      if (o3 == 0 and onSegment(p2, p1, q2)):
      return True
      if (o4 == 0 and onSegment(p2, q1, q2)):
      return True
      return False

def findIt(A,B,C,D):
      a1 = B.y - A.y
      b1 = A.x - B.x
      c1 = a1*(A.x) + b1*(A.y)
      a2 = D.y - C.y
      b2 = C.x - D.x
      c2 = a2*(C.x)+ b2*(C.y)
      determinant = a1*b2 - a2*b1
      x = (b2*c1 - b1*c2)/determinant
      y = (a1*c2 - a2*c1)/determinant
      return (x, y)

def findIntersections(lines, hlines):
      res = {}
      for hline in hlines:
      p1 = point(hline[0],hline[1])
      q1 = point(hline[2],hline[3])
        for line in lines:
            p2 = point(line[0][0],line[0][1])
            q2 = point(line[0][2],line[0][3])
            if(doIntersect(p1,q1,p2,q2)):
            res[findIt(p1,q1,p2,q2)] = line[1]
      return res

def getTrapEdges(d):
      N = len(d.getVertices())
```

```
        verts = [ list(d.getVertices())[i].coords for i in range(N) ]
        verts = zip(verts, [i for i in list(d.getVertices())])
        edges = [(verts[i][1].next.coords,verts[i][1].coords) for i in range(N)
]
        edges = zip(edges, [v[1].getOutgoingEdges()[0] for v in verts])
        verts.sort(key=lambda x: -x[0][0])
        lines = []
        temp = []
        for e in edges:
        temp = [e[0][0][0],e[0][0][1],e[0][1][0],e[0][1][1]],e[1]
        lines.append(temp)
        lines2 = []
        temp = []
        for v in verts:
        temp = verts[0][0][0],v[0][1],verts[-1][0][0],v[0][1]
        lines2.append(temp)
        res = findIntersections(lines,lines2)

        res = [[x,y,res[(x,y)]] for (x,y) in res]
        res.sort(key = lambda x: -x[1])
        ret = []

        for v in verts:
        templ = [(x[0],x[1],x[2]) for x in res if (x[0]<v[0][0] and
x[1]==v[0][1])]
        tempr = [(x[0],x[1],x[2]) for x in res if (x[0]>v[0][0] and
x[1]==v[0][1])]
        templ.sort(key = lambda x: x[0])
        tempr.sort(key = lambda x: x[0])
        if( len(templ)%2==0 and len(tempr)%2==0 ):
              if v[1].getOutgoingEdges()[0].getTwin().origin.coords[1] <
v[1].coords[1] :
              tr =
trapEdge(v[0],v[0],v[1],v[1].getOutgoingEdges()[0],v[1].getOutgoingEdges()[1].
getTwin())
              else:
              tr =
trapEdge(v[0],v[0],v[1],v[1].getOutgoingEdges()[1],v[1].getOutgoingEdges()[0])
#                   tr = trapEdge(v[0],v[0],v[1],None,None)
              ret.append(tr)
        if( len(templ)%2==1 and len(tempr)%2==1 ):
              tr =
trapEdge(templ[-1][:2],tempr[0][:2],v[1],templ[-1][2],tempr[0][2])
              ret.append(tr)
        if( len(templ)%2==0 and len(tempr)%2==1 ):
              tr = trapEdge(v[0],tempr[0][:2],v[1],v[1].getOutgoingEdges()[0],
tempr[0][2])
              ret.append(tr)
        if( len(templ)%2==1 and len(tempr)%2==0 ):
              tr =
trapEdge(templ[-1][:2],v[0],v[1],templ[-1][2],v[1].getOutgoingEdges()[1].getTw
in())
              ret.append(tr)
```

```
      return ret
# returns list of diagonals for partioning
def monotonePartitioningDgnls(d):

      ret = getTrapEdges(d)
      ret = sorted(ret, key=lambda x:-x.pivot.coords[1])

      a = dict()
      b = dict()

      for x in ret:
      x.re = x.re.getTwin()
      if DEBUG:
            print "\n",x.left,x.right
            print "Pivot:",x.pivot.coords
            print
"Ledge:",x.le.origin.coords,"-->",x.le.getTwin().origin.coords
            print
"Redge:",x.re.origin.coords,"-->",x.re.getTwin().origin.coords


      if x.pivot.coords[1] > x.re.getTwin().origin.coords[1]:
            a[x.pivot] = (x.le,x.re)
            if x.le in b:
            b[x.le].append(x.pivot)
            else:
            b[x.le] = [x.pivot]

            if x.re in b:
            b[x.re].append(x.pivot)
            else:
            b[x.re] = [x.pivot]


      for e in b:
      b[e].append(e.getTwin().origin)

      if DEBUG:
      print "\n### a"
      for (i,x) in enumerate(a):
            print
            print i,x.coords
            for e in a[x]:
            print e.origin.coords, e.getTwin().origin.coords

      print "\n### b"
      for (i,x) in enumerate(b):
            print
            print i,x.origin.coords, x.getTwin().origin.coords
            print b[x]

      dgnls = []
```

```
      for pt in sorted(a, key=lambda x:-x.coords[1]):
      if DEBUG:
            print "\n]]]]",pt.coords
            print a[pt][0].origin.coords, a[pt][0].getTwin().origin.coords,
            print len(b[a[pt][0]]),[x.coords for x in b[a[pt][0]]
],b[a[pt][0]].index(pt)
            print a[pt][1].origin.coords, a[pt][1].getTwin().origin.coords,
            print len(b[a[pt][1]]),[x.coords for x in b[a[pt][1]]
],b[a[pt][1]].index(pt),"[[[[["

      if pt in ( a[pt][0].origin, a[pt][0].getTwin().origin ):
            dgnls.append((pt, b[a[pt][1]][b[a[pt][1]].index(pt)+1] ))
      elif pt in ( a[pt][1].origin, a[pt][1].getTwin().origin ):
            dgnls.append((pt, b[a[pt][0]][b[a[pt][0]].index(pt)+1] ))
      else:
            dgnls.append((pt,
                        min( b[a[pt][0]][b[a[pt][0]].index(pt)+1],
                              b[a[pt][1]][b[a[pt][1]].index(pt)+1],
                              key=lambda x:x.coords[1]
                              )
                        ))
      if DEBUG:
            print "Dgnls:",[(x[0].coords,x[1].coords) for x in dgnls]

      if DEBUG:
      print "ppp",[(x.origin,x.getTwin().origin) for x in d.getEdges()]
      for ww in dgnls:
            print ww in [(x.origin,x.getTwin().origin) for x in d.getEdges()]
      dgnls = list(set(dgnls)-set([(x.origin,x.getTwin().origin) for x in
d.getEdges()]))
      return dgnls
```

Screenshots:

Blue lines are the TrapEdges and Green lines are the diagonals added.

I. N = 20

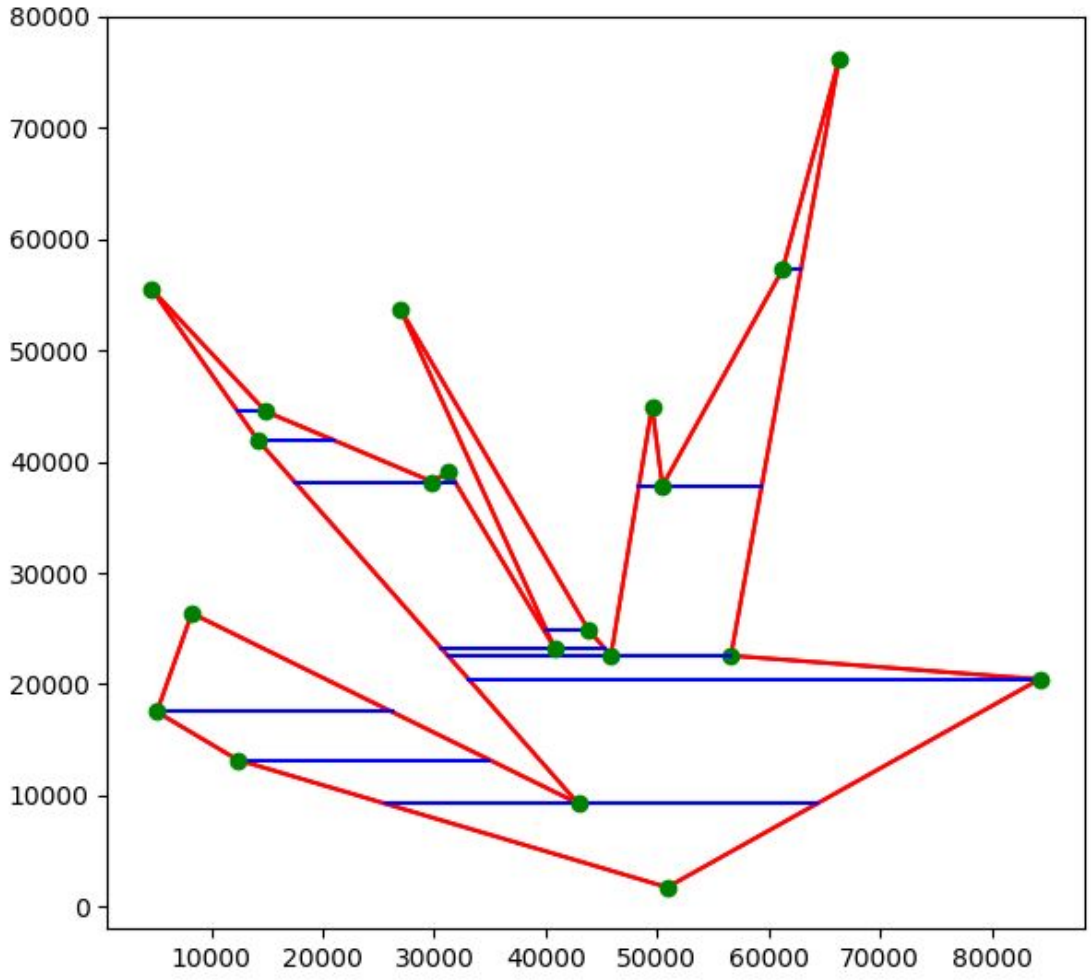


Fig: TrapEdges with N=20

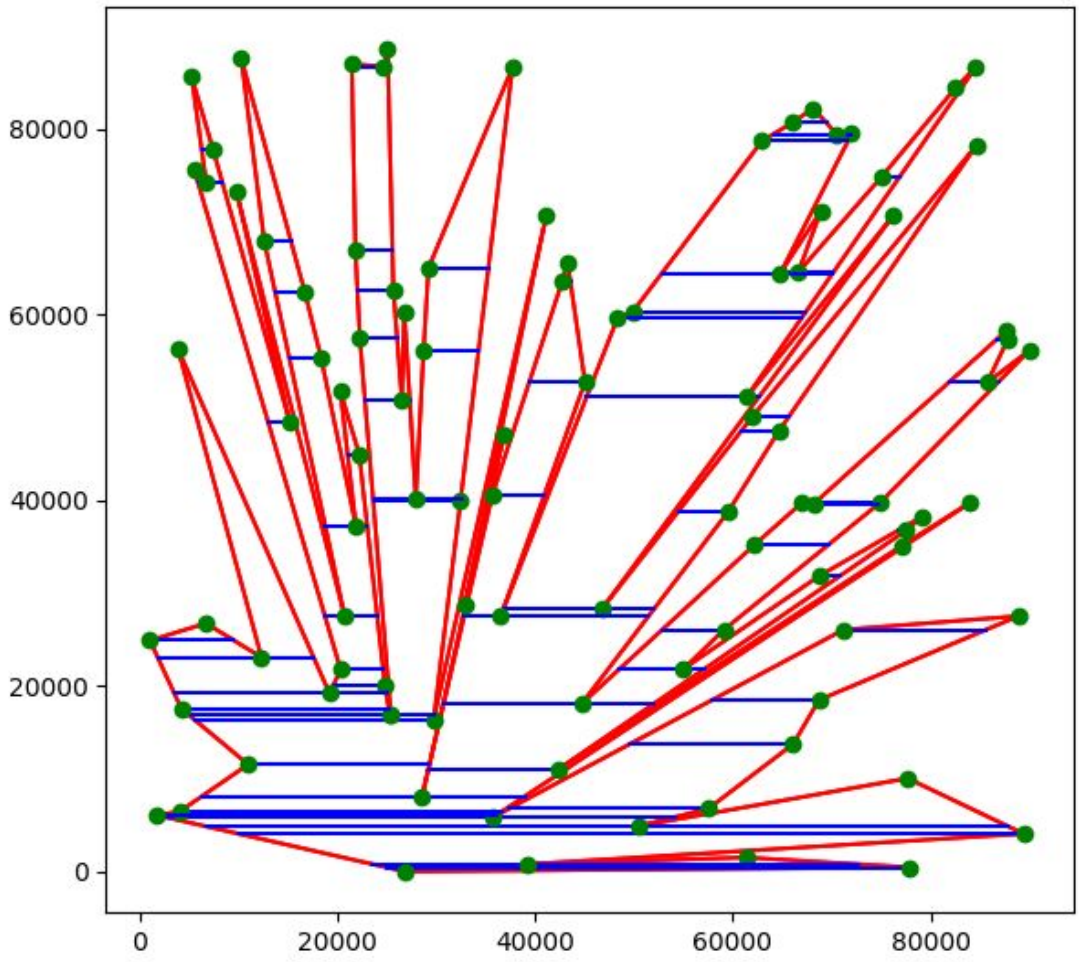


Fig: TrapEdges with N=100

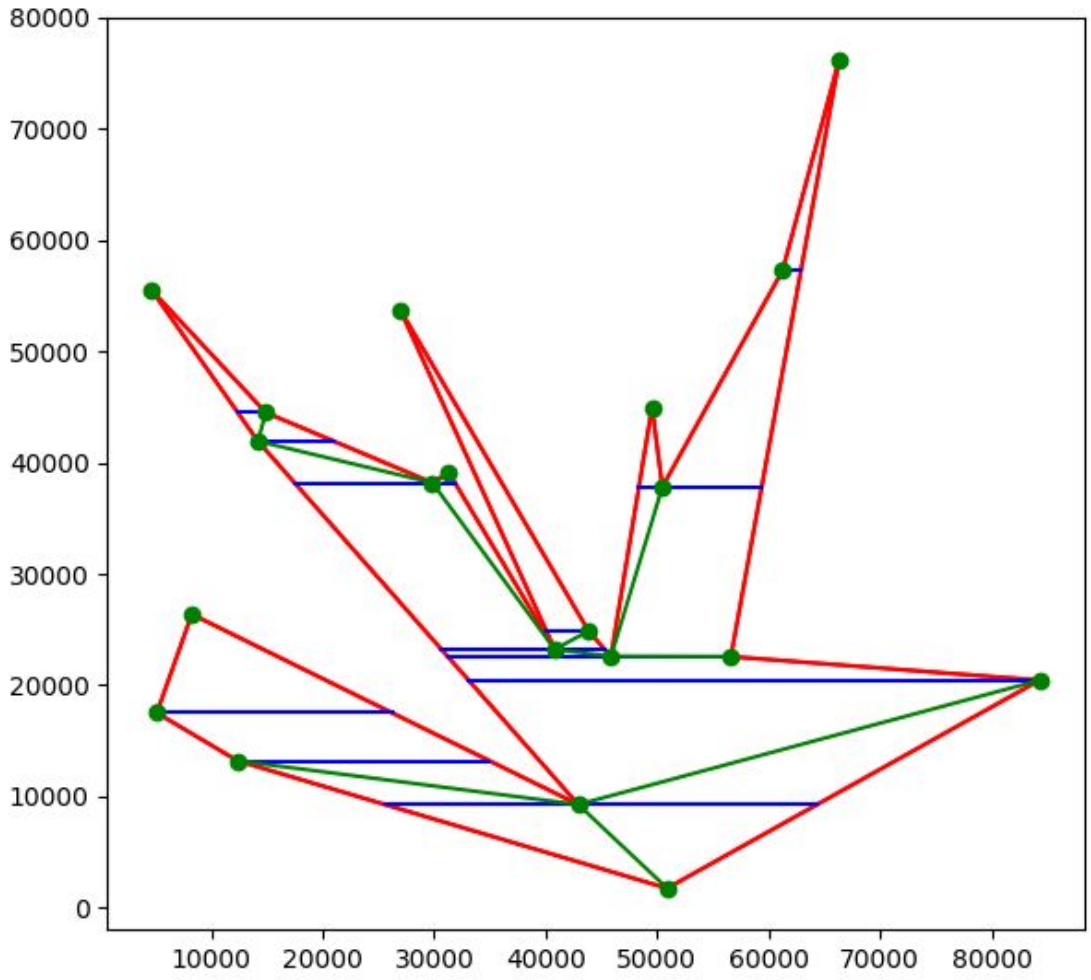


Fig: Trapezoid diagonals with N=20

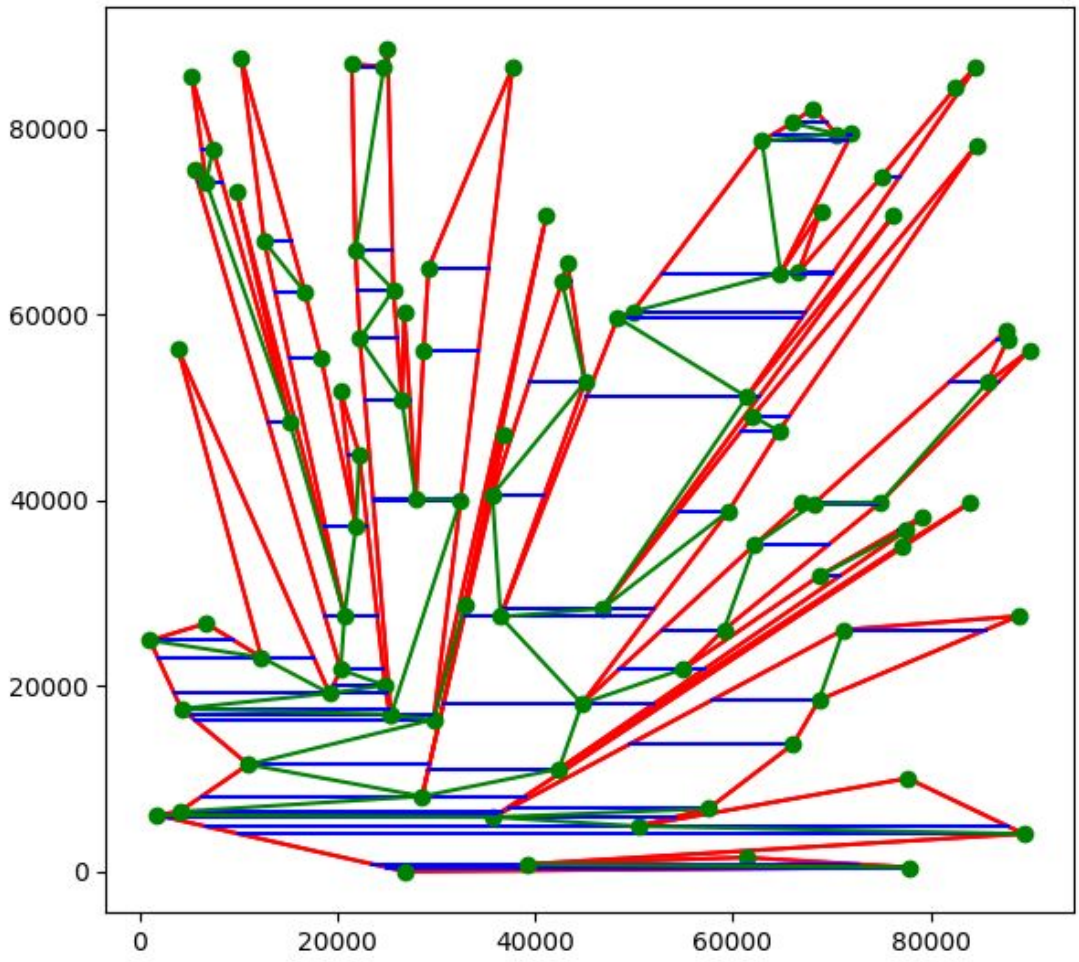


Fig: Trapezoid diagonals with N=100

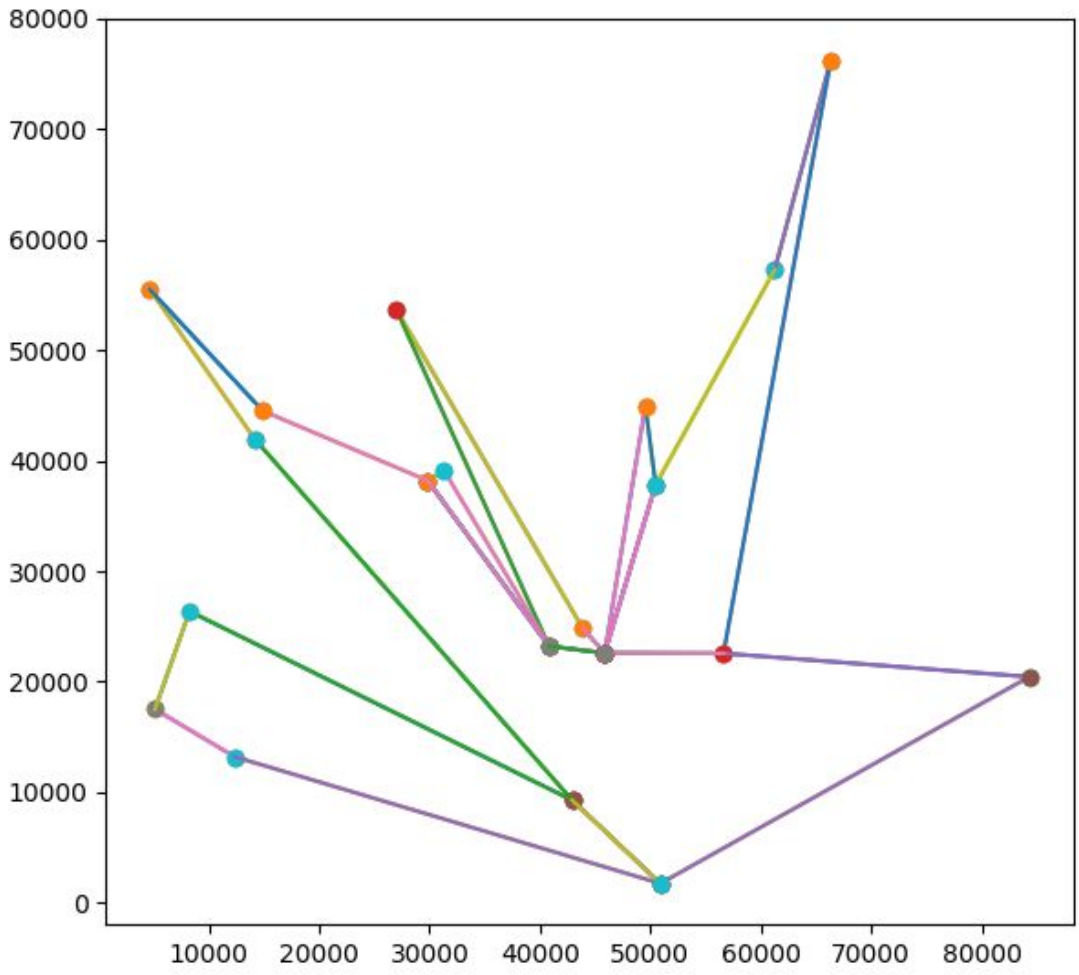


Fig: Monotone Partitions with N=20

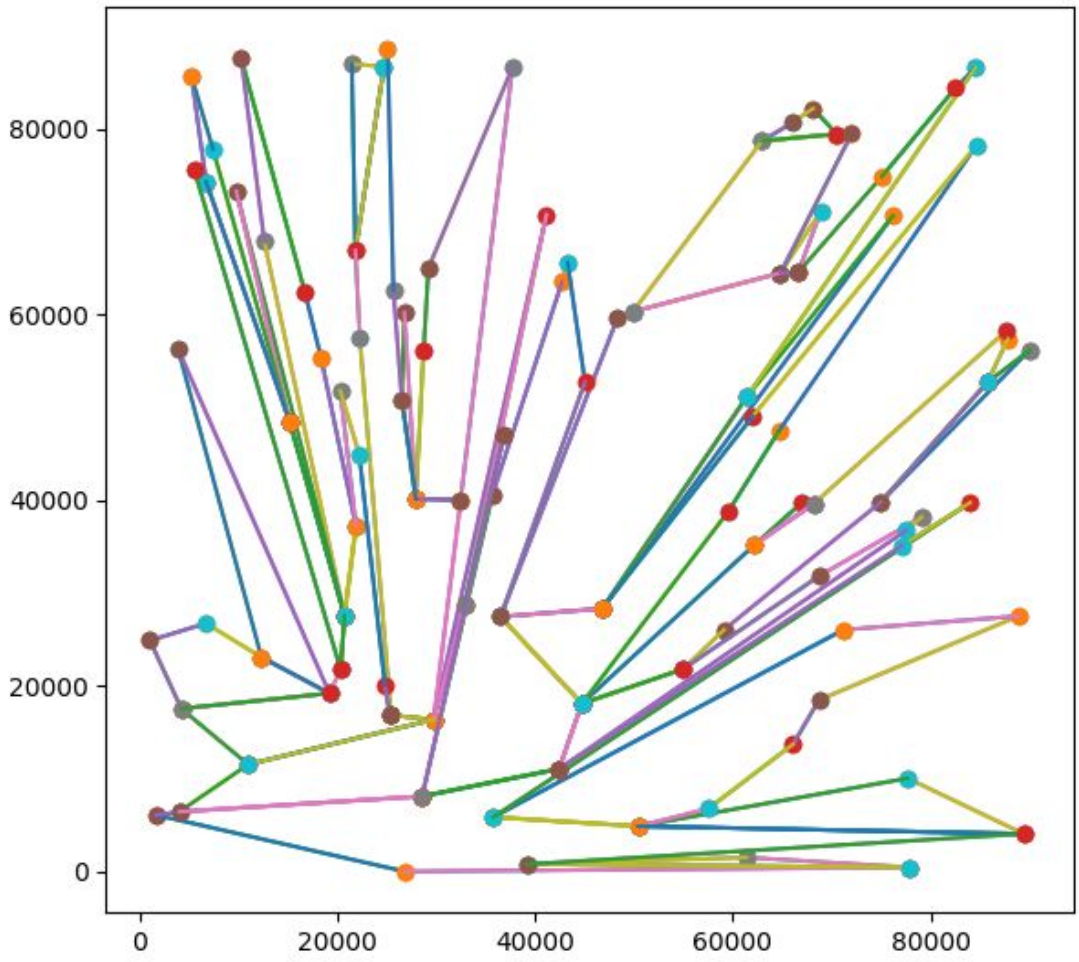


Fig: Monotone Partitions with N=100

- **For each monotone partition (polygon), perform triangulation using the line-sweep (plane-sweep) algorithm. Store the information in the required data structure (another DCEL)**

For each vertex accessed with decreasing y coordinate:

- If it belongs to opposite chain, add diagonal from it to every vertex from queue except last one. Add this vertex to queue.
- If it belongs to same chain and produces reflex turn, add it to queue
- If it belongs to same chain and produces convex turn, report the diagonal connecting it to penultimate element of queue, delete lat element from queue and process this vertex again.

Implementation

```
# triangulate Monotone Polygon // get list of diagonals

def Orientation(p,q,r):
	p=p.coords
	q=q.coords
	r=r.coords
	val = ( q[1] - p[1] )*( r[0] - q[0] ) - ( q[0] - p[0] )* ( r[1] - q[1] )
	return -val

def reflex(p,q,r,chain = 'l'):
	if(chain == 'r'):
	if Orientation(p,q,r)>=0:
		return True
	else:
		return False
	elif(chain == 'l'):
	if Orientation(p,q,r)>0:
		return False
	else:
		return True


def triangulateMonotonePolygon(d):
	pts = [x.origin for x in d.getFaces()[1].getOuterBoundary()]
	if DEBUG:
	print "Polygon Boundary:",[x.coords for x in pts]
	min_index = min(enumerate(pts), key=lambda x:x[1].coords[1])[0]
	max_index = max(enumerate(pts), key=lambda x:x[1].coords[1])[0]
```

```
        tmp1 = min(min_index,max_index)
        tmp2 = max(min_index,max_index)
        chain1 = pts[tmp1:(tmp2+1)]
        chain2 = pts[tmp2:]+pts[:(tmp1+1)]

        if(min(chain1, key=lambda x:x.coords[0]).coords[0] >min(chain2, key=lambda
x:x.coords[0]).coords[0]):   # ensuring chain1 is left chain
        if DEBUG:
                print "Monotone chains swapped"
        tmp = list(chain1)
        chain1 = chain2
        chain2 = tmp
        if DEBUG:
        print "Left Chain     : ",[x.coords for x in chain1]
        print "Right Chain    : ",[x.coords for x in chain2]
        pts = sorted(pts, key = lambda x:-x.coords[1])
        if DEBUG:
        print "\nSorted pts   : ",[x.coords for x in pts]
        print

        queue = []
        diagonals = []

        queue.append(pts[0])
        queue.append(pts[1])

        i = 2
        while i < (len(pts)-1):
        if DEBUG:
                print "\ni =",i,";",pts[i].coords
        #process(pts[i])
        tmp1 = queue[-1] in chain1
        tmp2 = pts[i] in chain1
        if (tmp1 and not tmp2) or (tmp2 and not tmp1):
                for qpt in queue[1:]:
                diagonals.append((pts[i], qpt))
                if DEBUG:
                        print "Case: a;  \nDiagonals:", [(x[0].coords,x[1].coords) for x in
diagonals]
                queue = [queue[-1],pts[i]]
                if DEBUG:
                print "Queue: ",[x.coords for x in queue]
        else:
                if DEBUG:
                print "||||||",queue[-2],queue[-1],pts[i],"chain =", ('l' if tmp1 else
'r'),"|||||"
                print reflex(queue[-2],queue[-1],pts[i],chain = ('l' if tmp1 else 'r') )
                print Orientation(queue[-2],queue[-1],pts[i] )
                if reflex(queue[-2],queue[-1],pts[i],chain = ('l' if tmp1 else 'r') ):
                queue.append(pts[i])
                if DEBUG:
                        print "Case: b;  \nDiagonals:",# reflex
                        print [(x[0].coords,x[1].coords) for x in diagonals]
                        print "Queue: ",[x.coords for x in queue]
                else:
                diagonals.append((pts[i], queue[-2]))
                if DEBUG:
                        print "Case: c;  \nDiagonals:", # convex
                        print [(x[0].coords,x[1].coords) for x in diagonals]
                        print "Queue: ",[x.coords for x in queue]
```

```
                queue.pop(-1)
                if len(queue) == 1:
                        queue.append(pts[i])
                else:
                        i-=1
        i+=1

        if len(queue)>2:
        for qpt in queue[1:-1]:
                diagonals.append((pts[i], qpt))
                if DEBUG:
                        print "Case: a;  \nDiagonals:", [(x[0].coords,x[1].coords) for x in
diagonals]

        if DEBUG:
        print "Queue: ",[x.coords for x in queue]
        return diagonals


triangulateMonotonePolygon(d)
listOfTriangles = insertDgnls(d,triangulateMonotonePolygon(d))
```

Time Complexity = O(n)

Screenshots

n=20

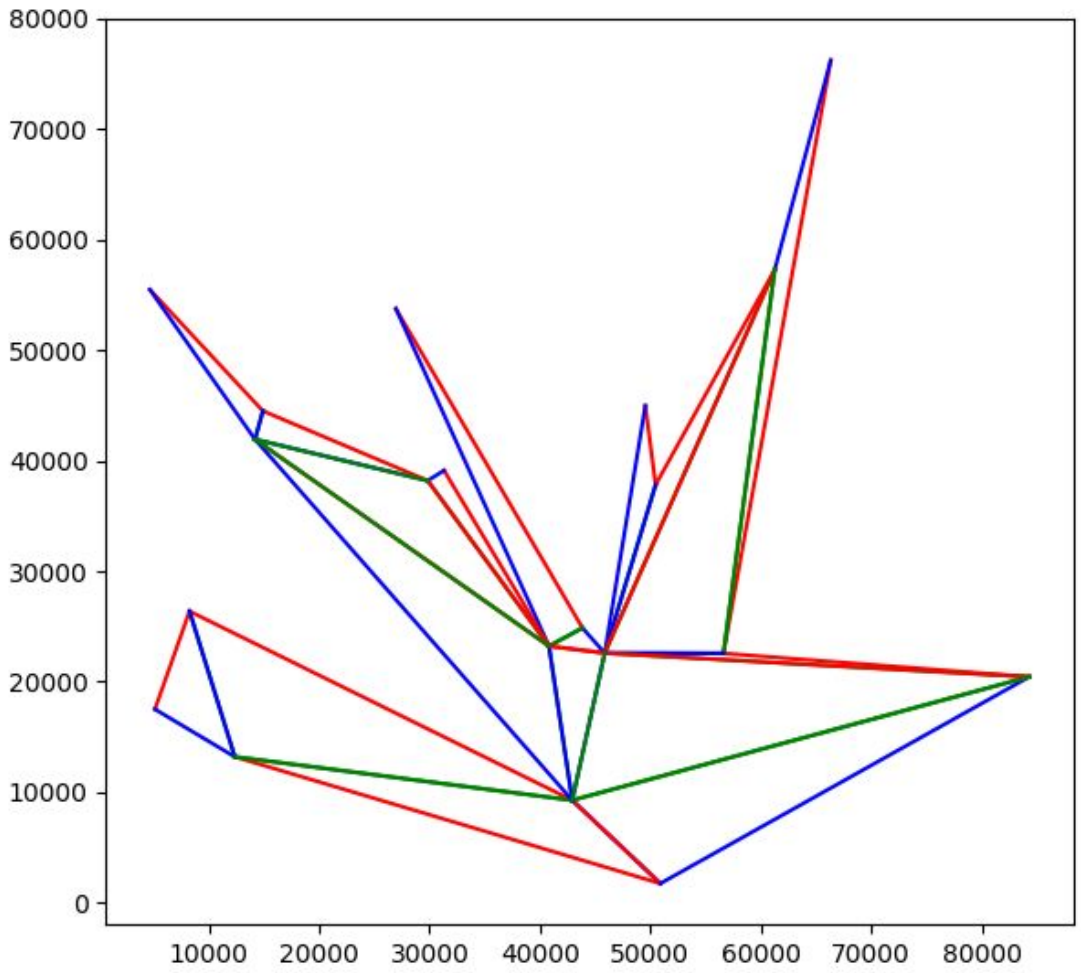


Fig: Triangulation with N=20

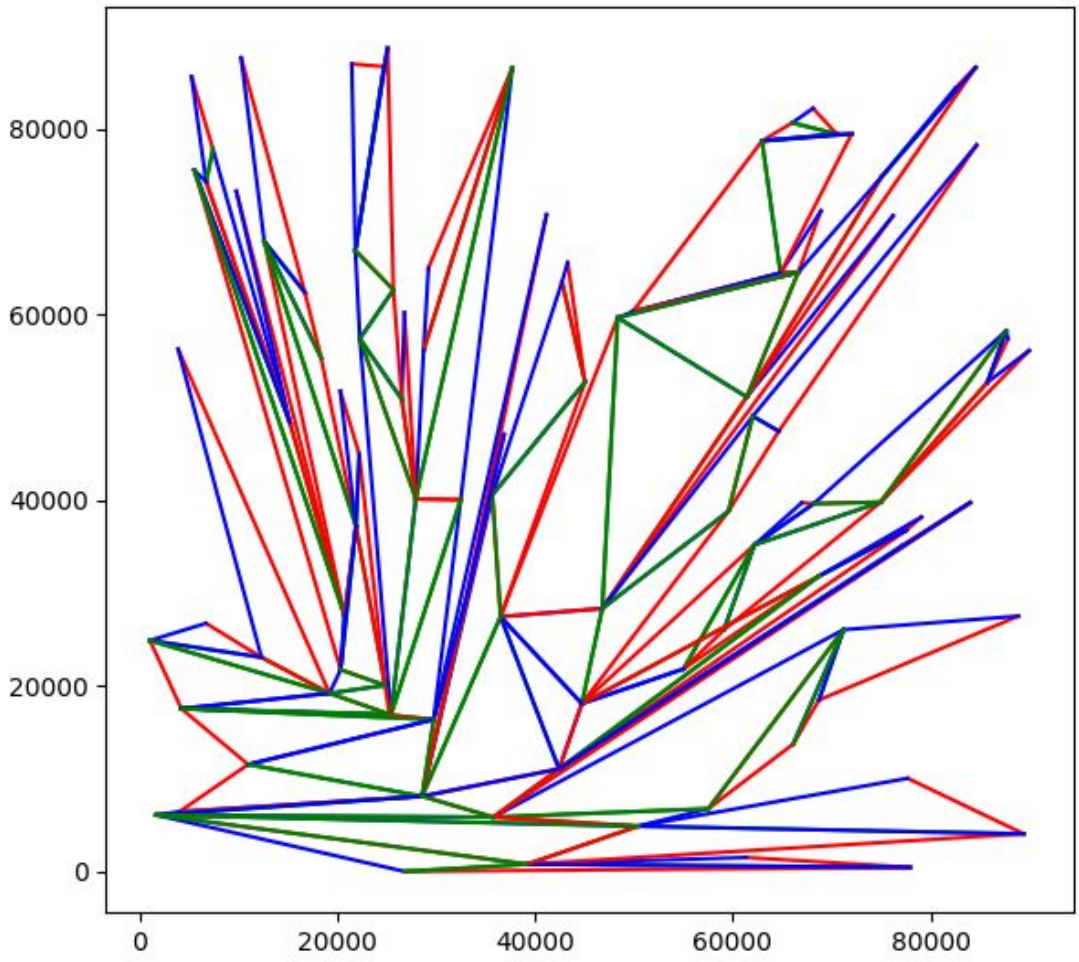


Fig: Triangulation with N=100

- **Now, assume that the simple polygon you constructed is the geometry of your Art-Gallery Problem. First, obtain the triangulated dual graph of that art-gallery. Then perform 3-coloring on the dual graph and determine the minimum number of vertex guards required to provide security of that Art-Gallery. Also, display the position of those vertex guards**

Implementation

- Colorizer class implements functions to obtain Dual Graph
    - For each face we have a new vertex and for each diagonal we have a new edge.
- To perform 3-coloring:
    - We perform DFS and stack stores the new vertices(old faces) and colorize three old vertices. Recursively go to adjacent old face(new vertex) perform coloring(if not visited and so on).
- To determine no. of vertex guards required as well as their positions. In function *findMinColor()*, all the color counts and their corresponding positions are there though it returns only for minimum one.

Time Complexity = **O(n)**

```
class Colorizer(object):
      def __init__(self,d,listTriangle):
      #Initialize color to -1
      self.colors = {v.coords:-1 for v in d.getVertices()}

      #Creating Dual Graph
      self.vdual={i:listTriangle[i] for i in range(0,len(listTriangle))}
      self.edual={}
      for i in range(0,len(listTriangle)):
            j=i+1
            for j in range(0,len(listTriangle)):
            triangle_i = [x.coords for x in listTriangle[i]]
```

```
            triangle_j = [x.coords for x in listTriangle[j]]
            if len(list(set(triangle_i)&set(triangle_j))) > 1:
                  if i in self.edual and j not in self.edual[i] and i is not
j:
                        self.edual[i].append(j)
                  elif i not in self.edual and i is not j:
                        self.edual[i]=[j]
                  if j in self.edual and i not in self.edual[j] and i is not
j:
                        self.edual[j].append(i)
                  elif j not in self.edual and i is not j:
                        self.edual[j]=[i]

      def DFS(self,s):
      visited, stack = set(), [s]
      while stack:
            vertex = stack.pop()
            if vertex not in visited:
            colorsum =
self.colors[self.vdual[vertex][0].coords]+self.colors[self.vdual[vertex][1].co
ords]+self.colors[self.vdual[vertex][2].coords]
            if DEBUG:
                  print "Changing Coloring of Triangle#:"+str(vertex)+"
from:
",self.colors[vdual[vertex][0].coords],self.colors[vdual[vertex][1].coords],se
lf.colors[vdual[vertex][2].coords]
            if colorsum<3:
                  if self.colors[self.vdual[vertex][0].coords] is -1:
                        self.colors[self.vdual[vertex][0].coords] =
3-self.colors[self.vdual[vertex][1].coords]-self.colors[self.vdual[vertex][2].
coords]
                  elif self.colors[self.vdual[vertex][1].coords] is -1:
                        self.colors[self.vdual[vertex][1].coords] =
3-self.colors[self.vdual[vertex][0].coords]-self.colors[self.vdual[vertex][2].
coords]
                  elif self.colors[self.vdual[vertex][2].coords] is -1:
                        self.colors[self.vdual[vertex][2].coords] =
3-self.colors[self.vdual[vertex][1].coords]-self.colors[self.vdual[vertex][0].
coords]
            if DEBUG:
                  print "to:
",self.colors[self.vdual[vertex][0].coords],self.colors[self.vdual[vertex][1].
coords],self.colors[self.vdual[vertex][2].coords]
            visited.add(vertex)
            stack.extend(set(self.edual[vertex]) - visited)

      def colorize(self):
      #key = first triangle to be 3-colored
      key = 0
            print ("Triangle #"+str(key)+" Vertex #0 colored to 0")
      self.colors[self.vdual[key][0].coords] = 0
      if DEBUG:
            print ("Triangle #"+str(key)+" Vertex #1 colored to 1")
```

```
        self.colors[self.vdual[key][1].coords] = 1
        if DEBUG:
                print ("Triangle #"+str(key)+" Vertex #2 colored to 2")
        self.colors[self.vdual[key][2].coords] = 2
        self.DFS(key)
        output,col = self.findMinColor()
        return output,col

        def findMinColor(self):
        rcount,gcount,bcount=0,0,0
        r,g,b=[],[],[]
        out= set()
        for t in self.vdual.values():
                for it in t:
                if it.coords not in out:
                        if self.colors[it.coords] is 0:
                                rcount+=1
                                r.append(it)
                        elif self.colors[it.coords] is 1:
                                gcount+=1
                                g.append(it)
                        elif self.colors[it.coords] is 2:
                                bcount+=1
                                b.append(it)
                        out.add(it.coords)
        if rcount is gcount and rcount is bcount:
                return r,rcount
        if rcount<=gcount and rcount<=bcount:
                return r,rcount
        if gcount<=rcount and gcount<=bcount:
                return g,gcount
        if bcount<=rcount and bcount<=gcount:
                return b,bcount

colorizer = Colorizer(d,listOfTriangles)
colorizer.colorize()
```

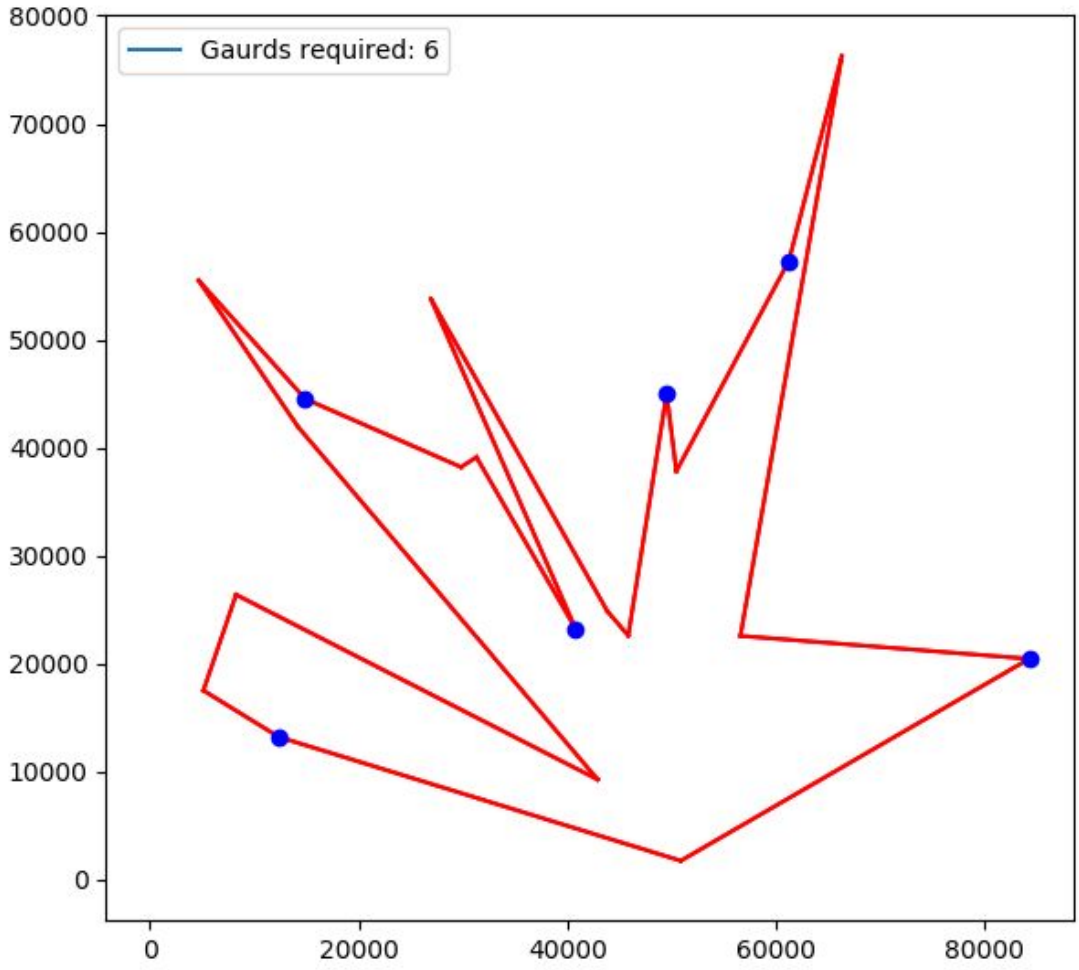


Fig: Guards for polygon with N=20

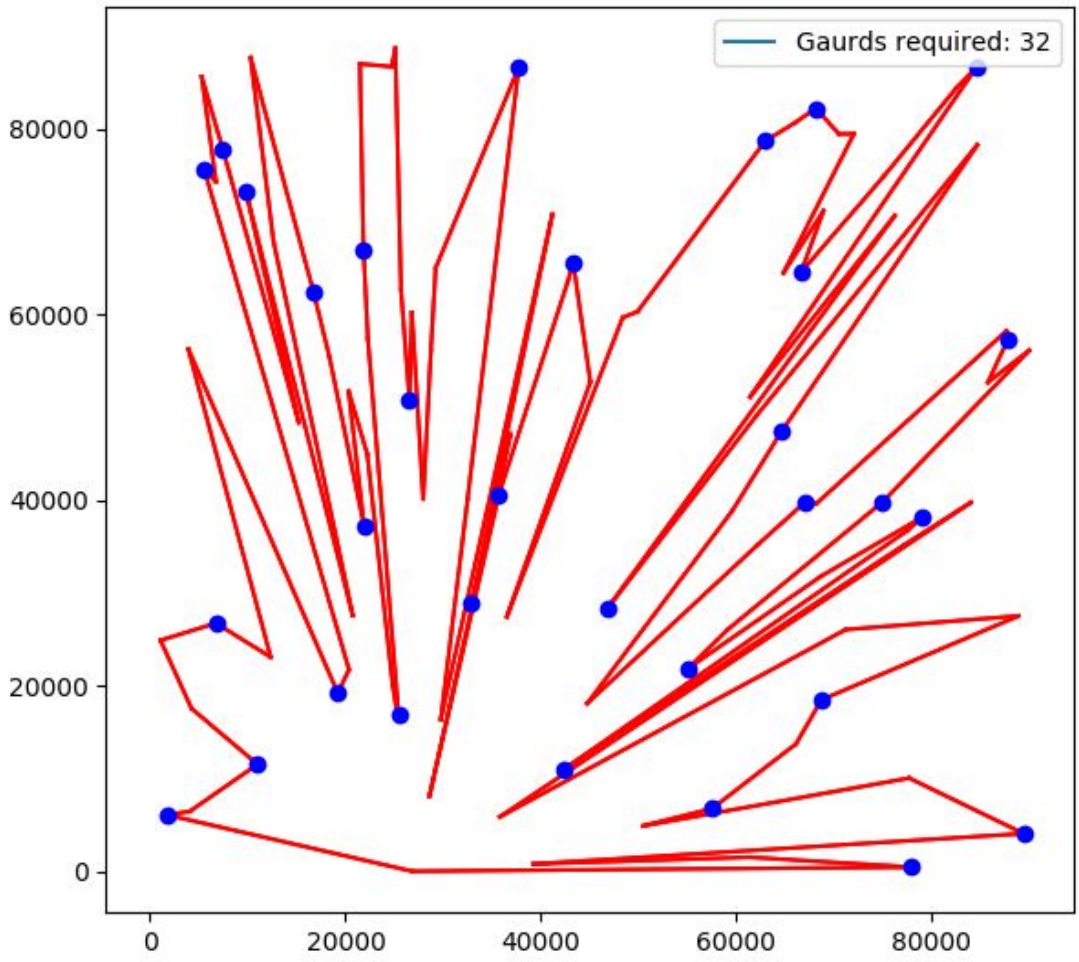
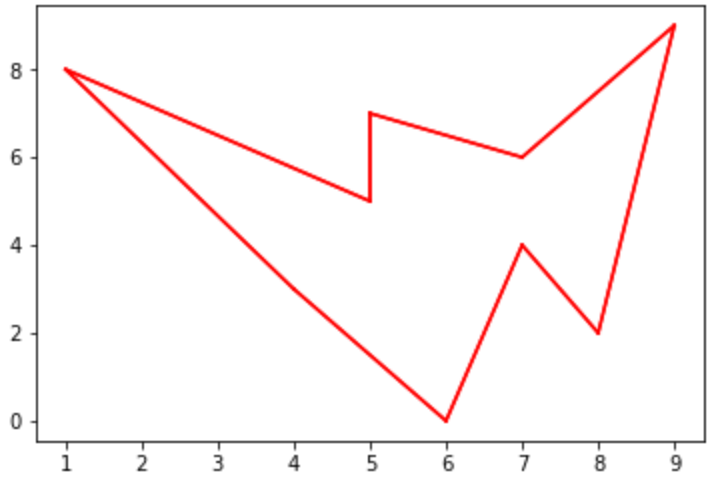


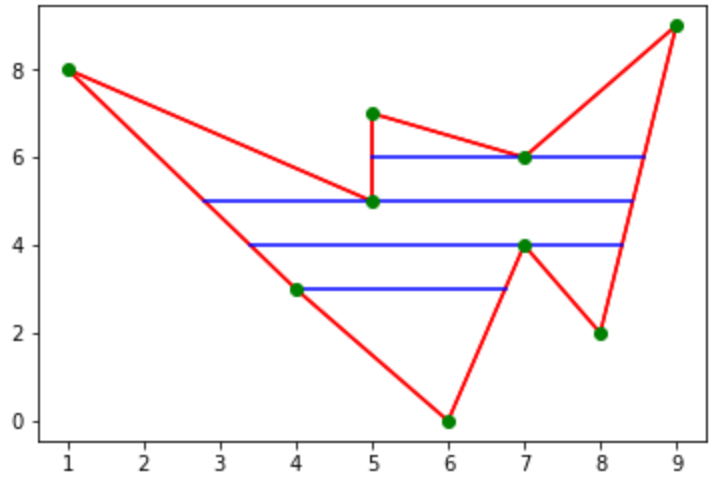
Fig: Guards for polygon with N=100

Screenshots for polygon with split vertices: (not random input):

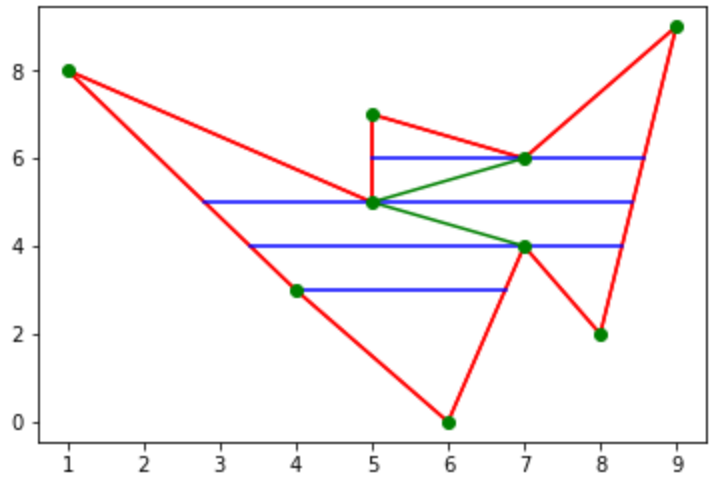
1. **Polygon**



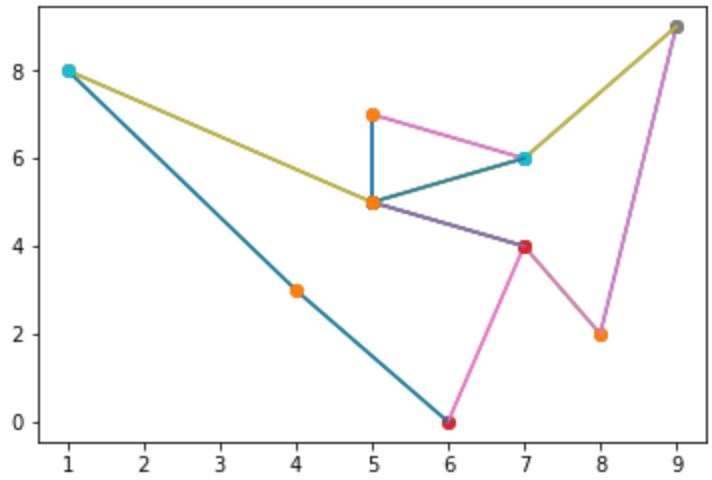
2. **TrapEdges**



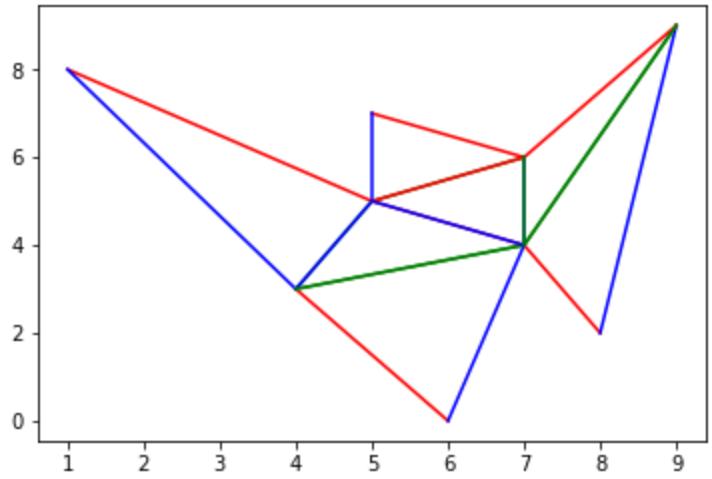
3. **TrapDiagonals**



4. Monotone Partitions



5. Triangulation



6. Vertex Guards