Stephanie Huang
May 8, 2023
Math 145A: Shapes of Surfaces
Creative Project
Prof. Bahar Acu

Using the Vietoris-Rips Complex to Test for Tornado Siren Coverage

My creative final project is a visual and interactive program written in Java-based Processing. This program allows users to place down tornado sirens and visualize the coverage of these sirens on a map. The program then translates this into a Vietoris-Rips complex and outputs corresponding boundary vectors and matrices that can be used to determine coverage and potentially minimize the number of sirens.

Background and Motivation

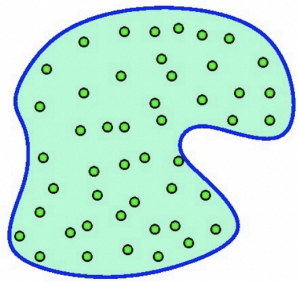
This project is motivated by an issue regarding the placement of tornado sirens. During a tornado warning, threatened communities may not be within audible range of a tornado siren. The range of a tornado siren is approximately 1-2 miles [1] and so this is especially an issue for rural communities in the Midwest that are not in range with more populated areas.

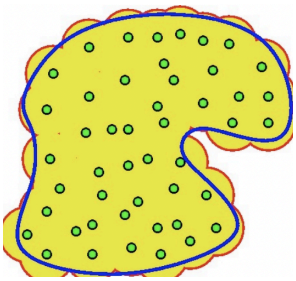
An important issue to note is that tornado sirens aren't very cost-effective; it costs around $20,000 - $30,000 to install one and this doesn't include yearly maintenance costs [2]. Full coverage can be quite expensive as multiple tornado sirens are needed. Because of this, some communities have abandoned the use of sirens entirely, opting for other, more effective methods [3].

However, many communities still rely on sirens for severe weather warnings and by doing so, need to balance coverage with cost. An ineffective plan that prioritizes coverage might be financially costly to the community while an ineffective plan that prioritizes budget might risk the community's safety. Communities want to strike a balance between the two: maximize coverage while minimizing the number of sirens. My program aims to give an approach to this problem.

Mathematical Theory

To determine if a set of sirens covers an area, we can reduce this to a classical problem called the Coverage Problem: We have a bounded 2-dimensional domain, and inside this domain are agents that have a radial range of coverage. Is the bounded domain covered by the agents?

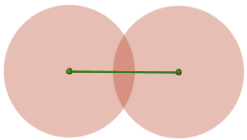




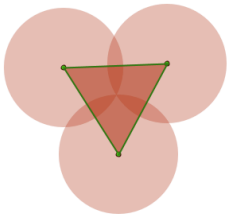
There is a method of testing for coverage using a Vietoris-Rips Complex. Below is the recipe for forming a Vietoris-Rips Complex:

We represent the agents (sirens in our case) with vertices.

If two agents have overlapping coverage areas, we form an edge (or 1-chain) between them:

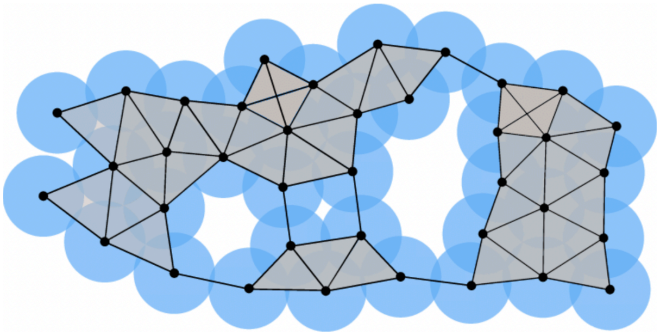


If three agents all have overlapping coverage areas with one another, a triangle will be formed by their edges. Fill in this triangle, thereby creating a face (or 2-chain):



To include the boundary, polygonize the boundary by making it into 1-cycle.

This turns out to model the coverage quite well. There seems to be a “hole” in the complex wherever there’s a “hole” in the actual coverage:



However, there can be some exceptions: 

A face can be formed whenever three vertices are pair-wise adjacent to each other. However, this is possible even if their coverages don't overlap with each other all at once, creating a "hole" (an alternative way of modeling this is the Čech complex, which avoids this problem but is more computationally expensive).

But under certain circumstances (which presumably avoid the above issue), this serves as a sufficient test for coverage; it has been proven that this test does not give false positives [4]. If the domain cannot be covered, the test will never say that it can be covered.

So to check for coverage, we just need to check for coverage in the Vietoris-Rips Complex. To do this, we'll apply some of the concepts we learned in Math 145A, specifically boundary maps and homology.

The technique used employs the same kind of technology we used for counting holes in simplices, which revolves around this equation:

$$H_k(M) = \frac{ker(\partial_k)}{im(\partial_{k+1})}$$

Recall that this counts the number of k-dimensional folds in a manifold M. $ker(\partial_k)$, which denotes all items $x \in M$ such that $\partial_k(x) = 0$, is the group of cycles. $im(\partial_{k+1})$, which denotes all items $y \in M$ such that $\exists x$ where $\partial_{k+1}(x) = y$, is the group cycles that bound something. Thus, by taking all cycles and quotienting (taking) out all those that bound something, we are left with cycles that don't bound anything, AKA holes.

Note we can do this because $im(\partial_{k+1}) \subseteq ker(\partial_k)$ as $\partial_{k+1}\partial_k = 0$. Thus, to check for full coverage, which is the absence of holes, we want to check that $H_k(M) = \frac{ker(\partial_k)}{im(\partial_{k+1})} = \emptyset$. In other words, $ker(\partial_k) = im(\partial_{k+1})$, all cycles bound something. Thus, we need to check $ker(\partial_k) \subseteq im(\partial_{k+1})$.

Because these are groups, we may think of $ker(\partial_k)$ and $im(\partial_{k+1})$ in terms of their generators and their sums. Assuming M is connected, all k-cycles generate the outer boundary, b. Thus, if $b \subseteq im(\partial_{k+1})$, then every k-cycle is also in $im(\partial_{k+1})$. So it is sufficient for us to check that $b \subseteq im(\partial_{k+1})$. In other words, there exists a $k + 1$-chain, x, such that $\partial_{k+1}(x) = b$.

In our case, our region is a 2-chain and our boundary is a 1-cycle, so we want to check if there exists a 2-chain, x, such that $\partial_2(x) = b$.

To do this, we may represent the boundary, b, as a vector, and the boundary operator, ∂_2, as a matrix. Recall that we may express a 1-cycle as the sum of 1-chains (its edges).

We'll give each of the vertices a unique ID and orient the 1-chains such that they are always going from the vertex with the smaller ID to the larger ID:

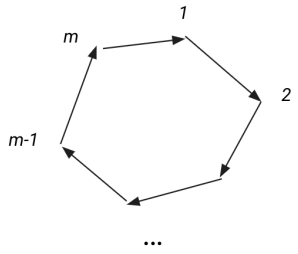
If $a < b$: $\quad a \rightarrow b$

Next, we'll create a vector whose rows represent all the 1-chains that are present in the Vietoris-Rips complex. The entry in each row denotes the contribution of that 1-chain to the boundary.

$$B = (12) + (23) + \ldots + (m\text{-}1\ m) - (1m)$$

$B =$

01	0
12	1
23	1
34	0
...	...
1m	-1
...	...



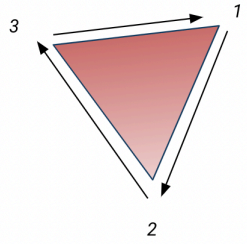
Next, we'll create the boundary matrix. Recall that the boundary operator on a 2-chain, c, is the alternating sum of all its edges. We express this in matrix form. So the matrix columns represent all the 2-chains and its rows represent all the 1-chains present in the Vietoris Rips complex.

To encode this, let the *ij*th entry denote the contribution of the 1-chain in the *i*th row to the 2-chain in the *j*th column.

For example, suppose a 2-chain is represented as *123*, in which the three letters denote the IDs of its vertices in numerical order. Thus, its boundary is formed by the 1-chains *12*, *23,* and *-13*. So the entries for *12*, *23*, and *13* under its column will be 1, 1, and -1 respectively.

$\partial =$

	… 012	123 …
01	1	0
12	1	1
23	0	1
…	…	…
02	-1	0
…	…	…
13	0	-1
…	…	…



By multiplying this boundary matrix with a vector representing a 2-chain, we get its boundary.

Thus $\partial_2(x) = \partial x$. So we want to check if there exists a 2-chain, x, such that $\partial x = b$.

If there is a solution for x, then there is (approximately) coverage. Otherwise, there is usually no coverage. Note that we may turn off all sirens that are not in x and the bounded region will still be covered.

Program

This program is written in Java-based Processing and aims to represent the placement of sirens as a Vietoris-Rips Complex. It then outputs the necessary contents to perform the aforementioned coverage test. This test allows users to see if their bounded region is covered and how they can minimize the number of sirens.

You'll first begin with a map of a target region; the default is a map of Stillwater, Oklahoma.

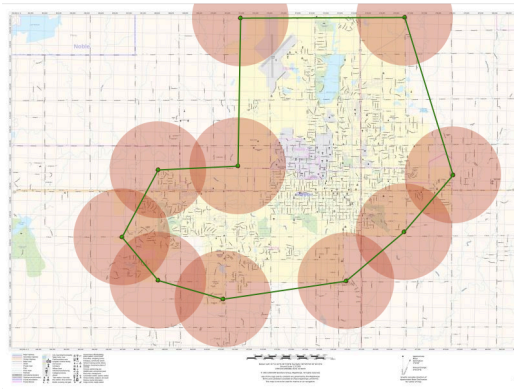
You can change the map by replacing the URL with the URL of your desired image on line 20. The URL of your image must end in ".png" or ".jpg". The line should be "`String url = "[URL]"`.

You also can adjust the size of the image on line 30 by adding two more parameters to `image(image, 0, 0);`. Inputting the desired width and height of your image in pixels this should be, `image(image, 0, 0, [desired width], [desired height]);`.

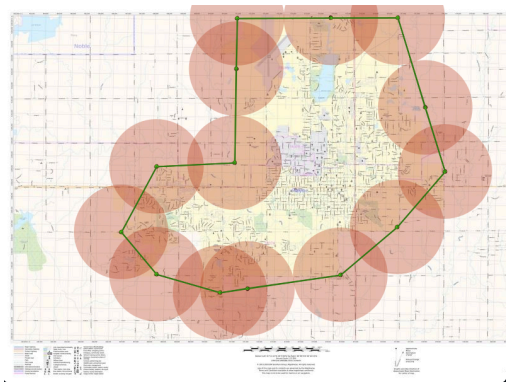
Note that the default metric is specifically tailored to the default image (75 pixels = 1 mile), so you'll have to translate what 1 mile into pixels and change line 8 to `float distance = [how many pixels is 1 mile];`.

If you make any changes to the program, you must click "run" (triangular button at the top left if you are using the Processing app) to see your changes.

You can then place sirens wherever you click on the map. You will first outline your boundary.

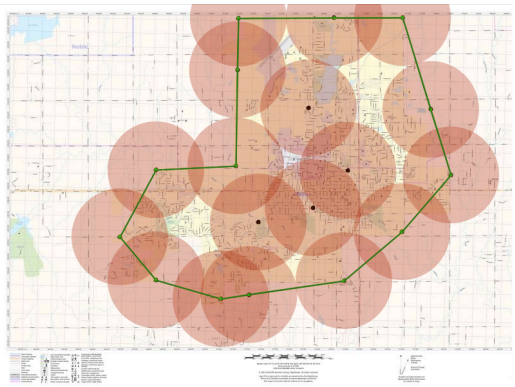


When you have finished outlining your boundary, press "b", and the program will automatically complete your boundary. It will insert additional sirens to fill in any gaps between successive sirens on the boundary. This ensures that the boundary is a 1-cycle.

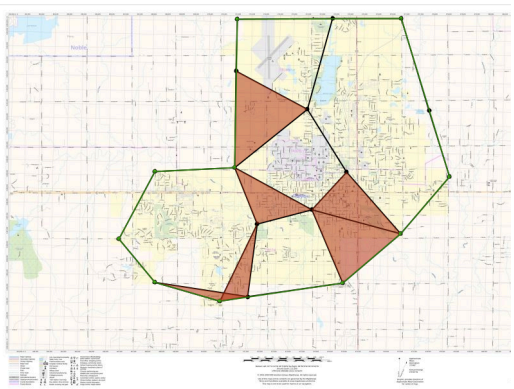


After forming the boundary, you may then place sirens inside the bounded region.

There are two modes in this program. In the default mode, you may see the actual radial coverage of the sirens. In the second mode, you may see the Vietoris-Rips Complex formed by the sirens. You can switch between these modes by pressing the spacebar.



Mode 1



Mode 2

When you are done placing your sirens, press enter. The program will then output the boundary vector, b, and the matrix for the boundary operator, ∂.

You may copy and paste these into a matrix equation solver. Because of their potentially large sizes, it's recommended that you do this in Python. Recall that we want to see if a solution for x exists in $\partial x = b$. If x has a solution, the bounded region is (approximately) covered. Otherwise, it is not covered. If the bounded region is covered, we may turn off the sirens that are not in x and the region will still be covered.

References

[1] Mele, C. (2016, May 23). *Tornado Sirens, an old technology, still play a vital role*. The New York Times. Retrieved May 7, 2023, from https://www.nytimes.com/2016/05/24/us/tornado-sirens-an-old-technology-still-play-a-vital-role.html

[2] Mersereau, D. (2021, May 28). *An Alabama Town Ditched Its Tornado Sirens. More Towns Should Follow Suit.* Forbes. Retrieved May 7, 2023, from https://www.forbes.com/sites/dennismersereau/2021/05/28/an-alabama-town-ditched-its-tornado-sirens-more-towns-should-follow-suit/?sh=591536a547d5

[3] Urbanowicz, A. (2018, May 30). *The truth behind Tornado Sirens*. WHVS. Retrieved May 7, 2023, from https://www.whsv.com/content/news/The-truth-behind-tornado-sirens-484002391.html

[4] Silva, V.D., Ghrist, R., & Muhammad, A. (2005). Blind Swarms for Coverage in 2-D. *Robotics: Science and Systems*.

Image Sources:

Coverage Problem: https://pages.pomona.edu/~vds04747/

Vietoris-Rips Complex:
https://www.researchgate.net/figure/A-Vietoris-Rips-complex-constructed-from-a-point-cloud-consisting-of-38-points_fig2_333330718

Stillwater, Oklahoma:
https://www.maptrove.com/media/catalog/product/900x600/s/t/stillwater-ok-map.jpg

Processing code:

```
import java.util.*;

ArrayList<Point> points = new ArrayList<>();
ArrayList<Edge> edges = new ArrayList<>();
ArrayList<Triangle> triangles = new ArrayList<>();
ArrayList<Edge> netEdges = new ArrayList<>();
int ps = 0;
float distance = 75;
ArrayList<Point> hullpts = new ArrayList<>();
float minx = 800;
Point hs = new Point(0, 0, -1);

ArrayList<Point> boundary = new ArrayList<>();
ArrayList<Point> eBoundary = new ArrayList<>();

boolean display = false;
boolean selectBoundary = true;
boolean done = false;

String url =
"https://www.maptrove.com/media/catalog/product/900x600/s/t/stil
lwater-ok-map.jpg";
PImage image = loadImage(url);

void setup() {
  size(800, 600);
}

void draw() {
  strokeWeight(2);
  background(200);
  image(image, 0, 0);

  for(int i = 0; i < points.size(); i++) {
    stroke(0);
    fill(0);
    ellipse(points.get(i).x, points.get(i).y, 5, 5);
    if(!display) {
      fill(180, 50, 20, 80);
      noStroke();
      ellipse(points.get(i).x, points.get(i).y, 2 * distance, 2
* distance);
    }
  }
  for(int i = 0; i < edges.size(); i++) {
```

```
    Point x = edges.get(i).x;
    Point y = edges.get(i).y;
    if(display) {
      line(x.x, x.y, y.x, y.y);
    }
  }
  strokeWeight(2);
  noStroke();
  fill(180, 50, 20, 80);

  if(display){
  for(int i = 0; i < triangles.size(); i++) {
    Triangle t = triangles.get(i);
    triangle(t.x.x, t.x.y, t.y.x, t.y.y, t.z.x, t.z.y);
  }
  }

  if((!display && selectBoundary) || (display &&
!selectBoundary)) {
  for(int i = 0; i < boundary.size(); i++) {
    noStroke();
    fill(70, 180, 20);
    ellipse(boundary.get(i).x, boundary.get(i).y, 5, 5);
    fill(180, 50, 20, 80);
    //ellipse(hullpts.get(i).x, hullpts.get(i).y, 75, 75);
    stroke(50, 120, 20);
    if(i < boundary.size() - 1) {
      line(boundary.get(i).x, boundary.get(i).y,
boundary.get(i+1).x, boundary.get(i+1).y);
    }
  }
  if(boundary.size() > 1) {
    line(boundary.get(boundary.size() - 1).x,
    boundary.get(boundary.size() - 1).y, boundary.get(0).x,
    boundary.get(0).y);
  }
  }
  else {
    for(int i = 0; i < eBoundary.size(); i++) {
      noStroke();
      fill(70, 180, 20);
      ellipse(eBoundary.get(i).x, eBoundary.get(i).y, 5, 5);
      fill(180, 50, 20, 80);
      //ellipse(hullpts.get(i).x, hullpts.get(i).y, 75, 75);
      stroke(50, 120, 20);
      if(i < eBoundary.size() - 1) {
        line(eBoundary.get(i).x, eBoundary.get(i).y,
```

```
      eBoundary.get(i+1).x, eBoundary.get(i+1).y);
    }
    else {
      line(eBoundary.get(eBoundary.size() - 1).x,
      eBoundary.get(eBoundary.size() - 1).y, eBoundary.get(0).x,
      eBoundary.get(0).y);
    }
  }
}
}

void keyPressed() {
  if(keyPressed == true) {
    if(key == ' ') {
   display = !display;
    }
    if(key == 'b' && selectBoundary) {
      println("Boundary Completed");
      selectBoundary = false;
      eBoundary = new ArrayList<>(boundary);
      for(int i = 0; i < boundary.size() - 1; i++) {
        Point a = boundary.get(i);
        Point b = boundary.get((i+1));
        float d = a.distTo(b);
        float drops = 0;
        if(d > 2 * distance) {
          drops = a.distTo(b) / (2 * distance);
        }
        drops += 1;
        eBoundary.add(a);
        for(int j = 1; j < (int)drops; j++) {
          float newX = (a.x * ((j * 2 * (distance-0.5))/d)) +
(b.x * (1-((j * 2 * (distance-0.5))/d)));
          float newY = (a.y * ((j * 2 * (distance-0.5))/d)) +
(b.y * (1-((j * 2 * (distance-0.5))/d)));
          Point nw = new Point(newX, newY, ps);
          nw.addThis();
          eBoundary.add(nw);
          ps++;
        }
        eBoundary.add(b);
      }
       Point a = boundary.get(boundary.size() - 1);
       Point b = boundary.get(0);
       float d = a.distTo(b);
        float drops = 0;
        if(d > 2 * distance) {
```

```
          drops = a.distTo(b) / (2 * distance);
        }
        drops += 1;
        eBoundary.add(a);
        for(int j = 1; j < (int)drops; j++) {
          float newX = (a.x * ((j * 2 * (distance-0.5))/d)) +
(b.x * (1-((j * 2 * (distance-0.5))/d)));
          float newY = (a.y * ((j * 2 * (distance-0.5))/d)) +
(b.y * (1-((j * 2 * (distance-0.5))/d)));
          Point nw = new Point(newX, newY, ps);
          nw.addThis();
          eBoundary.add(nw);
          ps++;
        }
        eBoundary.add(b);

       for(int i = 0; i < eBoundary.size(); i++) {
         Edge e = new Edge(eBoundary.get(i), eBoundary.get((i+1)
% (eBoundary.size())));
          for(int j = 0; j < edges.size(); j++) {
            if(e.equals(edges.get(j))) {
              edges.get(j).isBoundary = true;
            }
          }
      }

    }
    if(key == ENTER) {
      done = true;
      int[][] B = new int[edges.size()][triangles.size()];
    int[] g = new int[edges.size()];

    int ind = 0;
    for(int r = 0; r < g.length; r++) {
      if(edges.get(r).isBoundary) {
        ind++;
        g[r] = 1;
      }
    }
    println();
    println();

    g[ind - 1] = -1;
        for(int t = 0; t < triangles.size(); t++) {
          for(int e = 0; e < edges.size(); e++) {
            if(triangles.get(t).x.ID == edges.get(e).x.ID &&
triangles.get(t).y.ID == edges.get(e).y.ID) {
```

```
              B[e][t] = 1;
            }
            else if(triangles.get(t).y.ID == edges.get(e).x.ID
&& triangles.get(t).z.ID == edges.get(e).y.ID) {
              B[e][t] = 1;
            }

            else if(triangles.get(t).x.ID == edges.get(e).x.ID
&& triangles.get(t).z.ID == edges.get(e).y.ID) {
              B[e][t] = -1;
            }
          }
    }

    println();
    println("Boundary Vector: ");
    println("--------------------");
    println();
    for(int r = 0; r < g.length; r++) {
      println(g[r]);
    }


    println();
    println();
    println("Boundary Matrix: ");
    println("--------------------");
    println();
    for(int r = 0; r < B.length; r++) {
      for(int c = 0; c < B[0].length; c++) {
        print(B[r][c] + " ");
      }
      println();
    }

    }
  }
}

void mousePressed() {
  Point p = new Point(mouseX, mouseY, ps);
  p.addThis();
  ps++;
  done = false;

  if(p.x <= minx) {
    minx = p.x;
```

```
    hs = p;
  }

  //updateHull();


}

void updateHull() {
  hullpts.clear();
  for(int i = 0; i < edges.size(); i++) {
    //edges.get(i).isHull = false;
  }
  for(int i = 0; i < points.size(); i++) {
    points.get(i).intan(hs);
  }
  netEdges = new ArrayList<Edge>(edges);

  Collections.sort(points);

  hullpts.add(hs);
  if(ps >= 2) {
    hullpts.add(points.get(0));
    for(int i = 1; i < points.size(); i++) {
      float vec1_x = hullpts.get(hullpts.size() - 1).x -
hullpts.get(hullpts.size() - 2).x;
      float vec1_y = hullpts.get(hullpts.size() - 1).y -
hullpts.get(hullpts.size() - 2).y;

      float vec2_x = -(hullpts.get(hullpts.size() - 1).x -
points.get(i).x);
      float vec2_y = -(hullpts.get(hullpts.size() - 1).y -
points.get(i).y);

      float det = (vec1_x * vec2_y) - (vec1_y * vec2_x);

      while(det < 0 && hullpts.size() > 1) {
        hullpts.remove(hullpts.size() - 1);

        if (hullpts.size() >= 2) {
          vec1_x = hullpts.get(hullpts.size() - 2).x -
hullpts.get(hullpts.size() - 1).x;
          vec1_y = hullpts.get(hullpts.size() - 2).y -
hullpts.get(hullpts.size() - 1).y;
          vec2_x = hullpts.get(hullpts.size() - 1).x -
points.get(i).x;
```

```
          vec2_y = hullpts.get(hullpts.size() - 1).y -
points.get(i).y;
          det = (vec1_x * vec2_y) - (vec1_y * vec2_x);
        }
    }
    hullpts.add(points.get(i));
    }
  }
  for(int i = 0; i < hullpts.size(); i++) {
      if(hullpts.size() > 1) {
      Edge e = new Edge(hullpts.get(i), hullpts.get((i+1) %
(hullpts.size())));
      boolean found = false;
      for(int j = 0; j < edges.size(); j++) {
        if(e.equals(edges.get(j))) {
          //edges.get(j).isHull = true;
          found = true;
        }
      }
      if(!found) {
        //e.isHull = true;
        netEdges.add(e);
        //println(hullpts.get(i).ID + " " + hullpts.get((i+1)%
(hullpts.size())).ID);
      }
    }
  }

}


class Point implements Comparable<Point> {
  float x, y, angle;
  int ID;
  ArrayList<Point> connections = new ArrayList<>();

  Point(float x, float y, int i) {
    this.x = x;
    this.y = y;
    ID = i;
  }

  float distTo(Point p) {
    return (float)Math.sqrt(((x - p.x) * (x - p.x)) + ((y - p.y)
* (y - p.y)));
  }
```

```
  void intan(Point p2) {
        float xDiff = p2.x - this.x;
        if(xDiff == 0) angle = -1000;
        else angle = atan((p2.y - this.y)/(p2.x - this.x));
    }

    int compareTo(Point o) {
        if (o.angle > angle)  return -1;
        if (o.angle == angle) return 0;
        return 1;
    }

    void addThis() {
      for(int i = 0; i < points.size(); i++) {
    if(this.distTo(points.get(i)) <= 2 * distance) {
      this.connections.add(points.get(i));
      points.get(i).connections.add(this);
      edges.add(new Edge(this, points.get(i)));
    }
  }
  points.add(this);
  if(selectBoundary) {
    boundary.add(this);
  }

  for(Point p2 : this.connections) {
    for(Point p3: p2.connections) {
      if(p3.ID != this.ID) {
        if(p3.connections.contains(this)) {
          triangles.add(new Triangle(this, p2, p3));
        }
      }
    }
  }
    }

}

class Edge {
  Point x, y;
  boolean isBoundary;

  Edge(Point x, Point y) {
    if(x.ID < y.ID) {
      this.x = x;
      this.y = y;
    }
```

```
    else {
      this.x = y;
      this.y = x;
    }
    isBoundary = false;
  }

  boolean equals(Object o) {
   Edge other = (Edge)o;
   if(x.ID == other.x.ID && y.ID == other.y.ID) {
     return true;
   }
   return false;
  }
}

class Triangle {
  Point x, y, z;

  Triangle(Point x, Point y, Point z) {
    if(x.ID < y.ID && y.ID < z.ID) {
      this.x = x;
      this.y = y;
      this.z = z;
    }
    else if(x.ID < z.ID && z.ID < y.ID) {
      this.x = x;
      this.y = z;
      this.z = y;
    }
    else if(y.ID < x.ID && x.ID < z.ID) {
      this.x = y;
      this.y = x;
      this.z = z;
    }
    else if(y.ID < z.ID && z.ID < x.ID) {
      this.x = y;
      this.y = z;
      this.z = x;
    }
    else if(z.ID < x.ID && x.ID < y.ID) {
      this.x = z;
      this.y = x;
      this.z = y;
    }
    else {
      this.x = z;
```

```
      this.y = y;
      this.z = x;
    }
  }
}
```