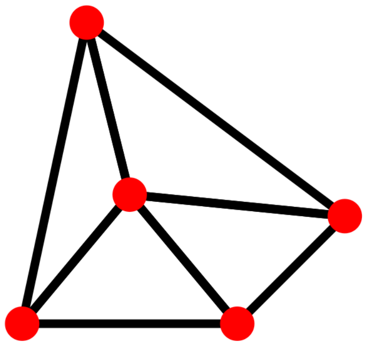
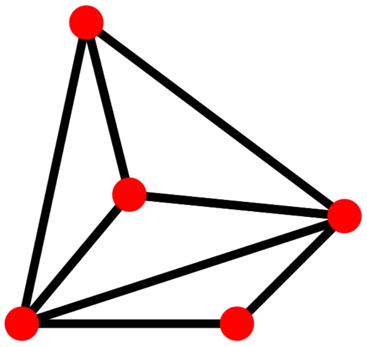
# Module 7 PA
# Delaunay Triangulations

CS 430 Programming Languages
Spring 2018

## 1. Introduction

Consider drawing a set of 2D points in the plane. A *triangulation* $T$ of the point set is a set of line segments formed by drawing a number of straight line segments between the points in the set in such a way that (1) no two line segments cross and (2) it is not possible to add a further line segment between to points $x$ and $y$ in the set without crossing one of the line segments already in $T$. Below we see two different triangulations of the same point set on five points (red).

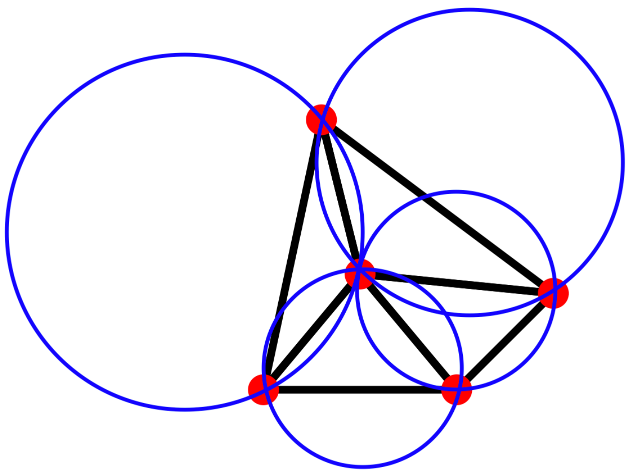
Triangulations are important for a variety of applications. For instance, finite element methods, which are used to simulate fluid dynamics (for instance, for simulating weather patterns) rely on triangulations to simplify and accelerate computation. In many uses of triangulations, it is particularly important to minimize the number of "skinny triangles," or triangles containing small angles. Notice that in the two triangulations above, the triangulation on the right contains more "skinny triangles" than the triangulation on the left.

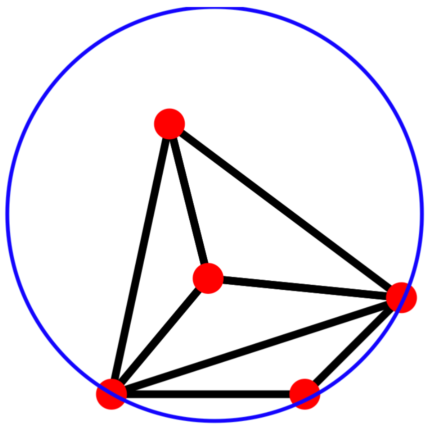
We can establish a natural comparison between two triangulations $T_1$ and $T_2$ of the same point set as follows. Compare the smallest angle in each triangulation made by a triangle that is not also a triangle of the other. The triangulation with the larger smallest angle is considered "better" than the triangle with the smaller smallest angle. In our illustration above, the two triangles incident to the highest vertex are common to both triangulations, while the remaining two triangles are not. Since the two lower triangles of the right triangulation have significantly smaller angles than the two lower triangles in the left triangulation, we say that the left triangulation is better than the right.

Given a point set, we would like to find the best possible triangulation. It turns out that such a triangulation exists and is unique for any given point set. The triangulation we want is called the *Delaunay Triangulation* (pronounced duh-LAHN-ee).

How do we compute this Delaunay triangulation? We could try enumerating all possible triangulations of a given point set, but this would lead to a very slow (exponential time) algorithm, since the number of triangulations of a point set with $n$ points is known to have $\Omega(2.33^n)$ triangulations!

It turns out that the Delaunay triangulation has a very simple and useful characterization that gives us an easy polynomial time algorithm for computing it. Recall the following fact from your high school geometry class—three non-collinear points uniquely determines a circle. The *Delaunay Triangulation* of a point set $S$ can be defined as follows. A triple of points $(p_1, p_2, p_3)$ from $S$ forms a triangle of the Delaunay triangulation if and only if for all other points $p$ in $S$, the circle through $p_1$, $p_2$, and $p_3$ does not contain $p$. The following two figures illustrate this property.





In the left, we have drawn the circles defined by each of the triangles. Notice that no point of the point set falls on the inside of any circle. This figure proves that the triangulation on the left is the Delaunay triangulation. For the right triangulation, we show the circle defined by the three corners of the bottom triangle. Notice that both of the other points of the point set fall on the inside of this circle. Since there exists a triangle whose circumcircle contains at least one point of the point set, the triangulation on the right is not a Delaunay triangulation.

## Computing the Delaunay Triangulation

Using this definition, we immediately have the following naive algorithm for computing a Delaunay triangulation of a set $S$. Consider all triples $(p_1, p_2, p_3)$ of points in our set. Test if all the other points $p$ in the set are outside the circle through $(p_1, p_2, p_3)$. If so, then $(p_1, p_2, p_3)$ is a triangle of the Delaunay triangulation, otherwise $(p_1, p_2, p_3)$ is not a triangle of the Delaunay triangulation. The running time of this algorithm is clearly $O(n^4)$, but that's significantly better than $\Omega(2.33^n)$!

## 2. Determining if a point is in a circle

We have provided as part of the starter code a function `in_circle?` that determines whether a point $p$ is in the circle defined by points $p_1, p_2, p_3$, with a catch—when you pass a triple of points $p_1, p_2, p_3$ to the `in_circle?` predicate you must specify them in counter-clockwise (CCW) order along the circle. Three points $p_1, p_2, p_3$ are in counter-clockwise order if and only if traveling from $p_1$ to $p_2$ to $p_3$ requires making a left-hand turn at $p_2$. We have supplied a second predicate function lefthand_turn? which returns true if and only if three points form a left-hand turn. Notice that if $p_1, p_2, p_3$ forms a right-hand turn instead of a left-hand turn then swapping the order of the last two points (i.e. $p_1, p_3, p_2$) forms a left-hand turn.

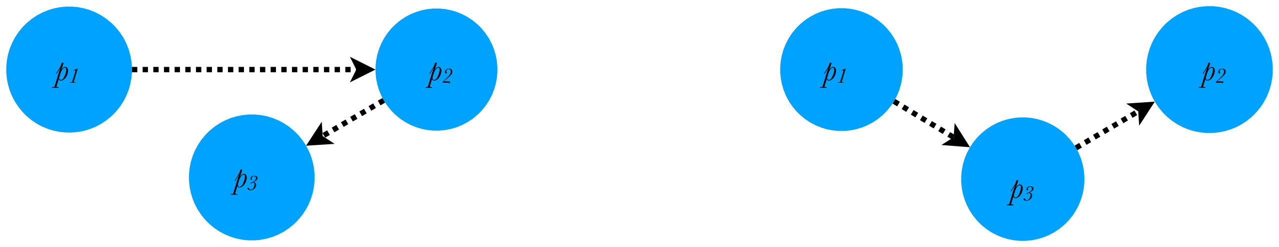


Fig. If $p_1, p_2, p_3$ forms a right-hand turn, then $p_1, p_3, p_2$ forms a left-hand turn (and vice versa).

This sensitivity to left-hand vs. right-hand turns often appears in geometric algorithms, and so it is useful to maintain one particular *orientation*. Because we need the points in counter-clockwise order for the `in_circle?` predicate, we will always store triples of points in counter-clockwise order.

## 3. Requirements

Your program must define and implement three classes: `Point`, `CCWTriple`, and `PointSet`.

The `Point` class must have two instance variables `x` and `y` (that is, `x` and `y` will be `attr_accessor` instance variables). `Point` must have the following methods:

> `initialize(x, y)`—must set `@x` and `@y`.
>
> `==(p)`—must redefine the equality operator so that it indicates whether the self and another Point instance have the same `@x` and `@y` values.
>
> `<=>(p)`—must redefine the comparison operator to enforce ordering based on x-coordinates (or y-coordinates if the x-coordinates are equal); e.g., (2,3) < (3,2) and (2,3) < (2,4), but (2,3) > (2,1).

The `CCWTriple` class must have three instance attributes `p1`, `p2`, and `p3`. These must be stored in counterclockwise order (i.e. so that `p1`, `p2`, `p3` forms a left-hand turn). `CCWTriple` must have the following methods:

> `initialize(p1, p2, p3)`—must set `@p1`, `@p2`, and `@p3`. Note that the input is not guaranteed to be in CCW order.

==(triple)—must redefine this equality operator so that it indicates whether the self and another `CCWTriple` contain the same three points. Note that the points are not guaranteed to be stored in the same order.

**Additionally, the `CCWTriple` class must have an alias `Triangle`**, which you can establish by simply defining the constant `Triangle` to be `CCWTriple`.

The `PointSet` class must have an instance attribute `@points` containing the list of points. It must also have the following methods:

`initialize(points=[])`—must initialize `@points` to `points`, or `[]` by default.

`size`—must return the size of the point set.

`add(p)`—must add a `Point p` to the set, but do nothing if it is already in the set.

`include?(p)`—must return `true` if and only if `Point p` is in the set.

`all_triples`—must generate a list of all the `CCWTriples` formed from the `Points` in the set, with no duplicates. If there are less than three `Points` in the set, then it must return `[]`.

`is_delaunay_triangle?(tri)`—must return true if and only if the `CCWTriple tri` is a Delaunay triangle of the set.

`delaunay_triangulation`—must return a list containing all of the `Triangles` in the Delaunay triangulation of the set; i.e., all of the Delaunay triangles.

`bounds`—must return a list `[minx, miny, width, height]` that constitute the bounds of the point set. `minx` is the smallest x-coordinate of any point in the set. `miny` is the smallest y-coordinate. `width` is the difference between the largest and smallest x-coordinates of any points in the set and `height` is the difference between the largest and smallest y-coordinates of any points in the set. This is used to generate graphical output in SVG format.

**Your submitted source file must be named `mod07pa.rb` and must not contain any Ruby testing framework test classes.** We will be using our own test classes to test your code.

## 4. Testing your code

You are advised to write test cases for your code as you develop it. Your test code can be in another file or in the file you turn in, but commented out before you submit it.

You might find it useful to write `to_s` methods for your classes (this is the Ruby equivalent of the `toString()` method in Java). This will allow you to print out your objects in a more readable format while debugging.

Additionally, we have included `delaunay.rb`, a simple application that will use your code to graphically depict the Delaunay triangulation of a point set. **Do not modify or submit the `delaunay.rb` file; your submission should work with the original version.** To use it, create a plaintext file of points with the following format—one point to each line, with x-coordinate and y-coordinate separated by a space.

For example, here is an example `points.txt` file:

```
50 50
100 50
65 120
75 80
125 75
```

Once you have finished your code, you can use `delaunay.rb` to compute the Delaunay triangulation of the point set and store a graphical depiction of it in the SVG format. As an example, if you have a `points.txt` file like the one above and want to generate an output named `out.svg`, you would run the following command.

```
ruby delaunay.rb points.txt out.svg
```

The output looks like the figure below. SVG files can be viewed in any modern web browser.