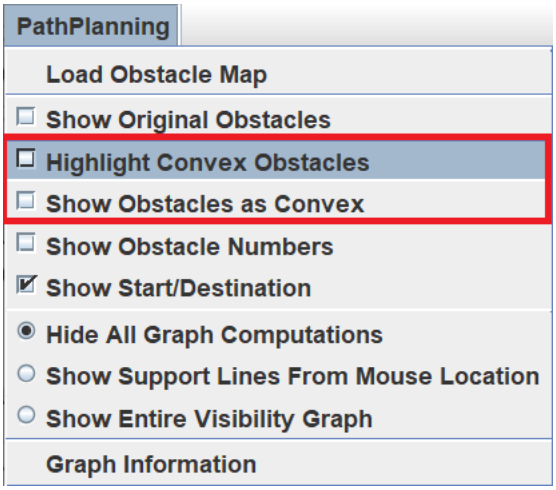
LAB 16 – Path Planning for Non-Convex Obstacles

(1) The goal of this lab is to adjust our code from the last lab so that we can compute a visibility graph for **non-convex** obstacles. Download the **Lab16_PathPlanningNonConvex.zip** file and unzip it. Again, in this lab, we will not use the Webots environment … you will compile the **MapperApp** program separately using a Java IDE. You will want to get all the source files into your IDE as well as the vector map files.

As with the previous lab, you will make use of the **Path Planning** menu. A new **Highlight Convex Obstacles** item has been added which will show the convex obstacles as yellow and non-convex as blue (once you get your code working). Also, a new **Show Obstacles as Convex** item has been added which will run the algorithm for splitting the non-convex obstacles into triangles.



You will ONLY be working within the **Obstacle** and **PathPlanner** classes, but you may want to open the **Graph**, **Edge** and **Node** classes as well, since you will make use of their various methods.

(2) There is a blank function in the **Obstacle** class called **isConvex()**. Complete this function so that it returns **true** if the obstacle is convex and **false** otherwise. Remember that all of our obstacles have their vertices stored in a counter-clockwise (i.e., CCW) order. You must make sure that all the points in sequence make left turns (see slide 8 in notes). When you get it completed, run the code, load up the map called **PathPlannerNonConvex.vmp** map and select the **Highlight Convex Obstcales** button. If all is working correctly, it will show the **5** convex obstacles as yellow (top, bottom & left) and the non-convex ones as light blue. Save a screen snapshot as **Snapshot1.png**. If you load up the **PathPlannerConvex.vmp** map, all should appear as yellow.

Debugging Tips:

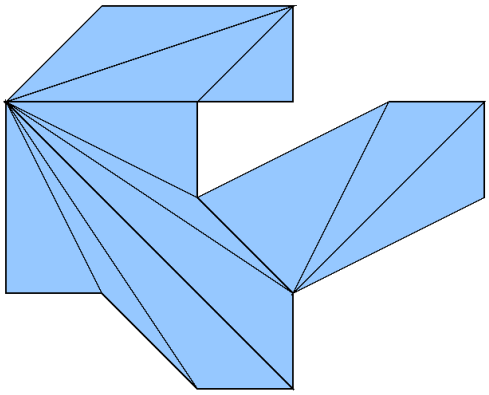
- If you **do not see any yellow obstacles**, then you did something wrong.
  - Make sure that you are checking for all **LEFT** turns … if you check for RIGHT turns, then it won't work.
  - It is also possible that you made a mistake in how you computed the **indices of vertices before and after** the current vertex. Double-check your code.
- If you see **6 convex obstacles**, then check your FOR loop boundary conditions. You are likely not handling the wrap-around case with respect to vertex indices.

(3) Run your code (if not already running) and load up the **PathPlannerNonConvex.vmp** map. Select the **Show Entire Visibilty Graph** radio button in the **Path Planning** menu. You will notice that the graph has many edges that go through the obstacles. The code only eliminates support lines that intersect *other* obstacles, not ones that intersect the same obstacle. This problem will go away once we split the polygons into non-convex pieces.

(4) Complete the **splitIntoTriangles()** function in the **Obstacle** class so that it does an ear-cutting triangulation of the obstacle. Follow the algorithm in slides 12 to 14 of the notes. The function MUST return an ArrayList of **Obstacle** objects … where each of the obstacles in that list is a triangle. Here are a few tips to keep in mind when implementing the algorithm:

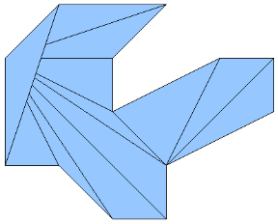
- When copying a point (x,y), make sure to make a real copy by calling the constructor like this: `copyOfPoint = new Point(x,y)`
- Remember … given an index `i` in a list of size `s` you can get the index before it by using `(i-1 + s)%s` and the index after it by using `(i+1)%s` so that you don't have to worry about any wrap-around cases 😊.
- There are two VERY useful **Obstacle** methods that you will want to use called `contains(Point p)` and `pointOnBoundary(Point p)`. 😉

Once your code compiles … try it out. Load up the **SingleNonConvex.vmp** map and select the **Show Obstcales as Convex** checkbox in the **PathPlanning** menu. If all worked out well, you should see a triangulation of the single obstacle that should look somewhat like this shown here (although your triangulation may be different, depending on the order that you processed the vertices). If you use **Show Obstacle Numbers** from the menu option, you should see the triangles numbered from **0** to **11**.

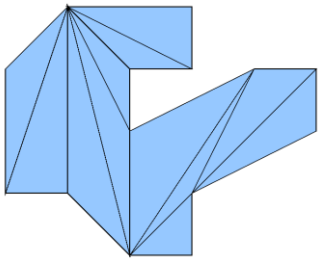


Debugging Tips:

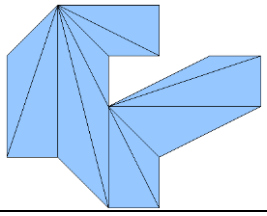
- If you see this triangulation, then you are adding the last ear wrong. Likely you are using the original obstacle vertices instead of the last three points in the list that you were looping through.



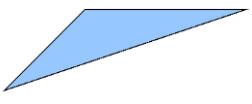
- If you see this triangulation, then it is likely that you forgot to check for wrap-around when comparing vertices to see if they are inside or on the boundary of the ear. It is still a valid triangulation but the algorithm is actually not implemented properly.



- If you see this triangulation, then you may have a misplaced the closing braces in your code … which changes the algorithm.



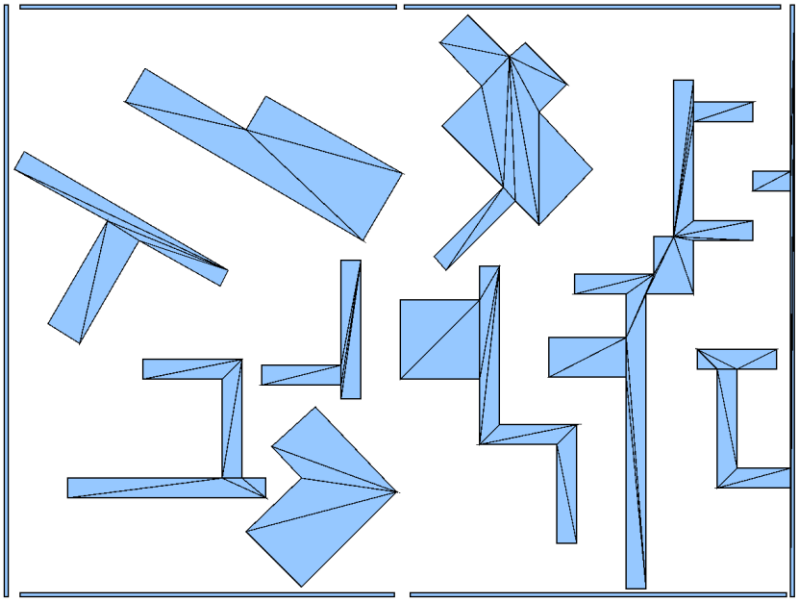
- If you are seeing **only one** triangle (**or just a few**), then you are likely returning from the method too early. This could be because you are using "**vertices**" instead of the variable that contains your list of copied points. Once you get the point copies, you should not need to access the obstacle **vertices** anymore for the rest of the code.



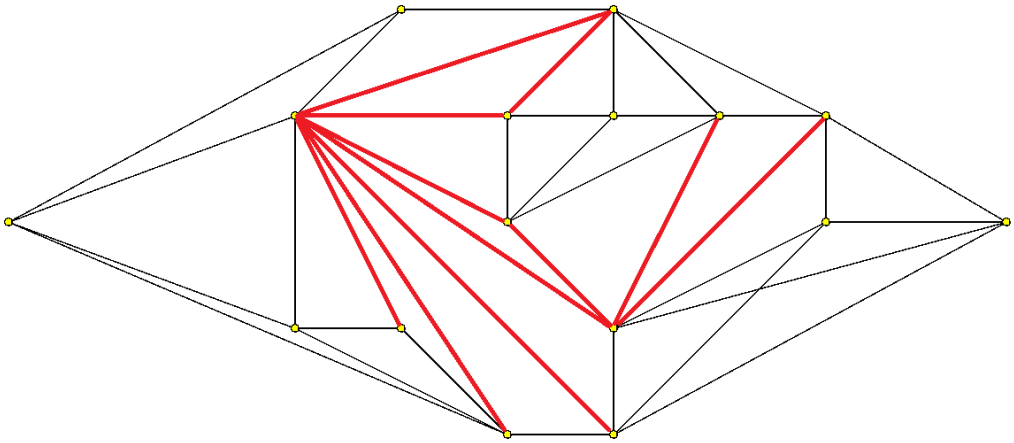
- If you only see obstacles numbered from **0 to 10**, then perhaps you are not adding the last ear outside of the **while** loop.

- If you get **no obstacles** appearing, then it could be one of many things wrong:
  - Perhaps you are only creating one obstacle outside the loop instead of creating a new obstacle for each ear.
  - Or maybe you are trying to do too much on the **IF** statement line that checks for the valid $\mathbf{p_k}$ index, contains and the boundary conditions. Break up the **IF** statement so that you are checking the valid $\mathbf{p_k}$ first and then another **IF** statement for checking if $\mathbf{p_k}$ is contained or on a boundary.
  - Or maybe you are returning from the method too early.
  - Or perhaps you are not adding the ears to the triangles list.

Once you are convinced that everything is working, load up the **PathPlannerNonConvex.vmp** map, select **Show Obstacles as Convex** and confirm that the triangulation shows **93** obstacles (use **Show Obstacle Numbers** from the menu option to verify … goes from **0** to **92**). Save a screen snapshot as **Snapshot2.png**. It is ok if your triangulations are a little different.



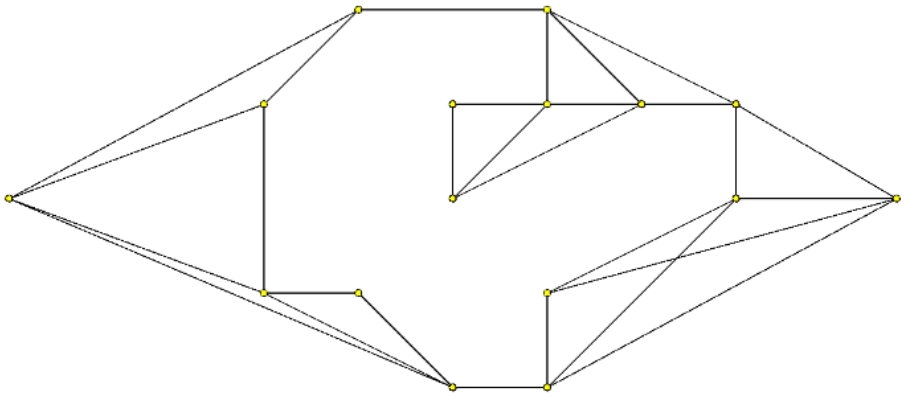
(5) Load up the **SingleNonConvex.vmp** map and select the **Show Entire Visibility Graph** radio button. You will see the computed visibility graph, which should still work. However, if you unselect the **Show Obstacle as Convex** checkbox while still displaying the visibility graph, you should see the graph as shown below (although I added the red coloring). The graph has **16** vertices and **39** edges, which you can verify from the **Graph Information** option in the **PathPlaning** menu.



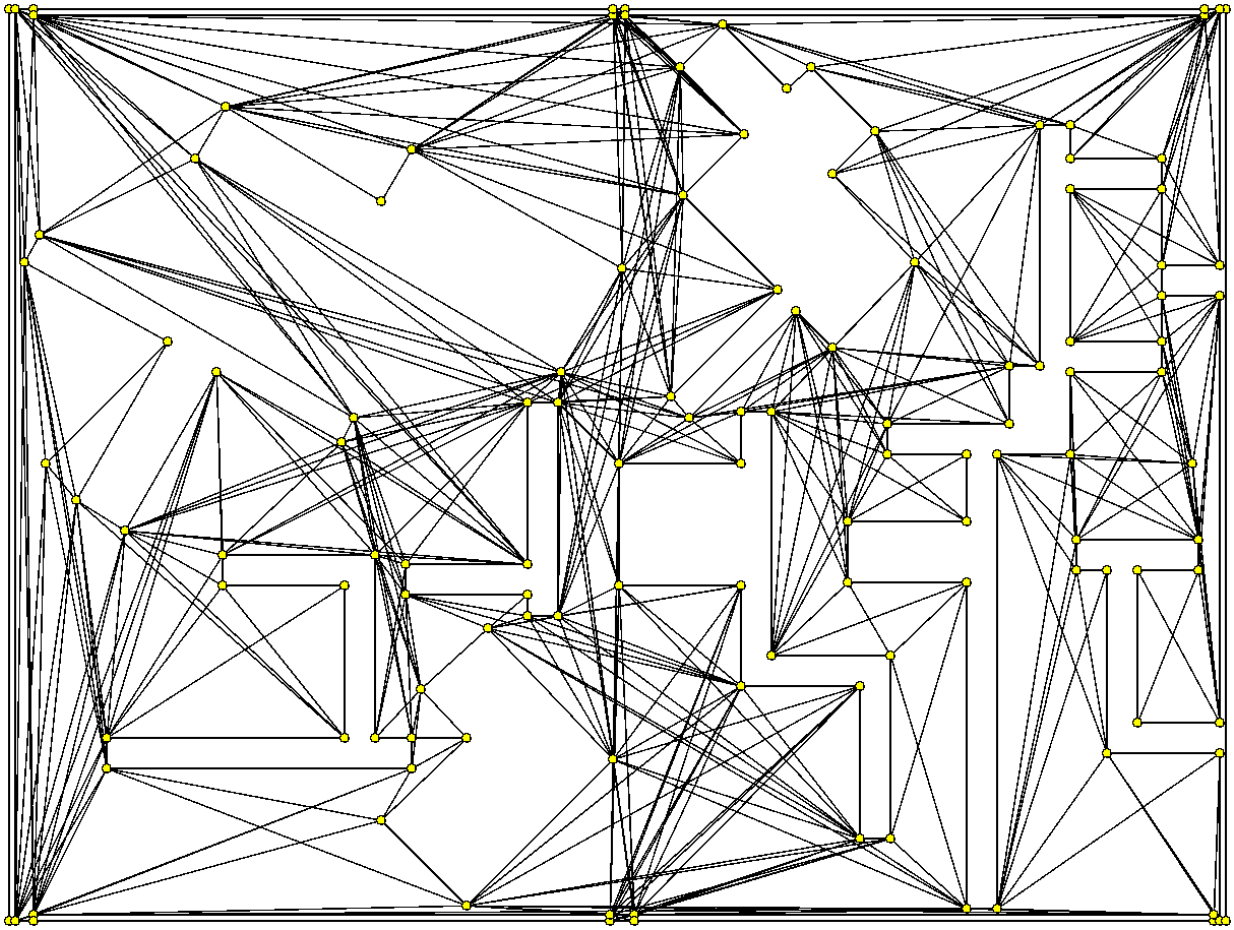
You will notice that all the graph edges that correspond to a boundary shared by two triangles of the same obstacle (i.e., the red ones) are showing up as edges in the graph. None of these edges should be in the graph because it represents travel within the original obstacle. These edges must NOT be added to the graph.

You MUST add code to an **IF** statement within the **supportLineIntersectsObstacle()** function in the **PathPlanner** class. This function takes a support line (defined by the (**x**, **y**) and (**supportX**, **supportY**) points passed in as parameters. It also has a **fromObst** parameter which is the obstacle that we are finding the support lines for. Finally, it takes the ArrayList of **allObstacles**.

You will need to handle the special case where the support line is actually an edge of the obstacle that you are checking against (see yellow shared edges in slide **9** in the notes). You need to check for these cases within the **IF** statement in order to eliminate them. You don't need to remove them from any list, just have your code return **true** … thereby indicating that the support line intersects that edge and the graph algorithm code will make sure not to add that support line to the graph. Once it works, you should see what is shown here (don't forget to uncheck the **Show Obstacles as Convex** and **Show Original Obstacles** buttons). The graph should now have **16** vertices and **28** edges. Save a snapshot as **Snapshot3.png**.



Once you believe that your code is fully working, you can test it by loading up the **PathPlannerNonConvex.vmp** map and then selecting **Show Entire Visibility Graph**. If all works well, you should not see any of those "shared" edges. You should see the graph as shown below. You should verify the number of edges now by examining the **Graph Information** from the **Path Planning** menu. It should have **128** vertices and **542** edges. Take a snapshot of the graph without the obstacles being shown and save it as **Snapshot4.png**.



Submit **ALL of your java code**, including the files that were given to you. It is important that there are no package statements at the top of your source files (some IDEs require you to put everything into a project or package). Just submit the source files without the package lines at the top (if any from a different IDE). If you are using IntelliJ and are unsure how to do this, please see step **(9)** of the "Using the IntelliJ IDE" file provided.

Submit the **4** snapshot **.png** files.

Make sure that your name and student number is in the first comment line of your code.