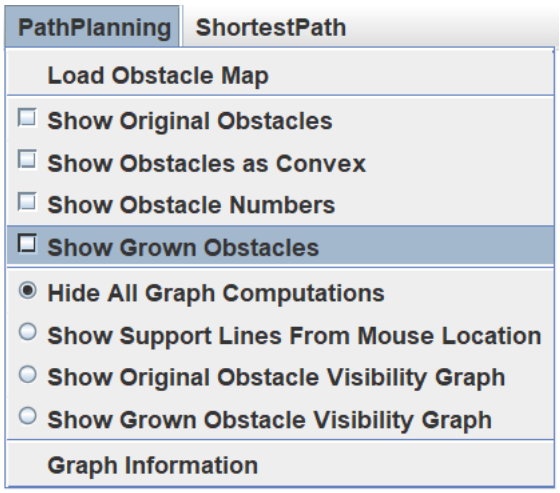
LAB 18 – Grown Obstacle Space

(1) The goal of this lab is to grow the convex and non-convex obstacles so that the shortest path computation allows collision-free travel for our robot. Download and unzip the file called **Lab18_GrownObstacles.zip**. The code will be completed in two parts. For parts **1** to **4**, you will work in your Java IDE as you have been doing in recent labs. Then for part **5**, you will copy over your code and run it in Webots to watch the robot travel along your computed path.

To begin, you will start by compiling and running the **MapperApp** class which now contains a modified and expanded **PathPlanning** menu →



The options allow you to show the Original Obstacles as well as the Grown Obstacles. In addition, you have the ability to compute the visibility graph on the original obstacles or on the grown obstacles and to show/hide that as well.

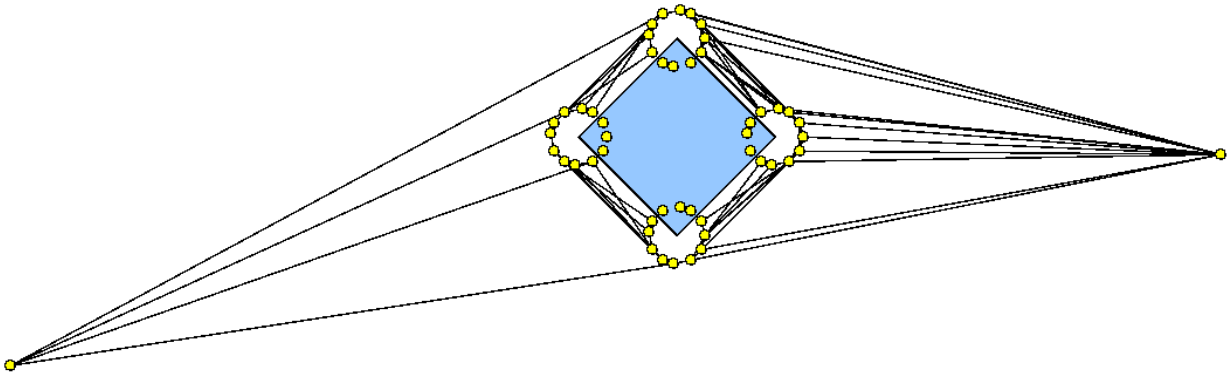
The same java classes (although some have been updated again) from the last lab have been provided. You will ONLY be working within the **Obstacle** and **PathPlanner** classes.

(2) **DO NOT implement the Convex Hull algorithm in this part**. You will do that in part 3. Follow the algorithm in the notes to expand an obstacle by placing a circle of points around each obstacle vertex (i.e., slide 9) and then computing the convex hull of the points (i.e., slide 8). In the **Obstacle** class, there is a function called **grow()** which takes a **radius** (which is the radius of the robot) and a **degreeRoundnessAngle** … which is the difference (in degrees) between two points being added in a circle around the obstacle vertex (e.g., **22.5°** in example on slide 9). There is also a function called **convexHull()** which takes an **ArrayList<Point>** of points and returns an **Obstacle** that represents the convex hull of this obstacle.

Complete the **grow()** function so that it creates an **ArrayList<Point>** of points representing all circles of points around each vertex of the obstacle. The points should be placed by using the **radius** and **degreeRoundnessAngle** parameters according to what was done in slide 9. Keep in mind that the code on slide 9 is for a single obstacle vertex and you will need to do this for each vertex. You will need to make use of Java's **Math.floor()** and **Math.ceil()** functions. The code will call the **convexHull()** method when completed.

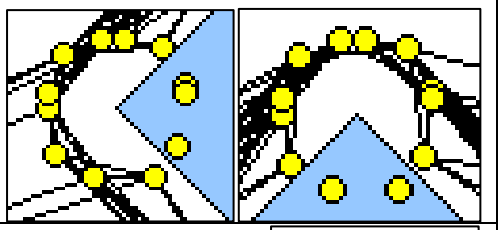
In the **convexHull()** method, you will notice that it creates a new obstacle called **hull** which is returned at the end of the method. Insert code that adds all the incoming points as vertices of the **hull** obstacle (do not worry about the fact that the vertices don't form a nice polygon. We are just trying to ensure that we placed the vertices properly around the original obstacle points). As a reminder … **do not compute the convex hull** yet.

Load up the **SingleBox.vmp** map, select the **Show Original Obstacles** checkbox and select **Show Grown Obstacle Visibility Graph** from the **PathPlanning** menu. You should see what is shown below … make sure to save it as **Snapshot1.png**. Do not worry about the graph edges … just ensure that you have the 12 nodes around each obstacle vertex.

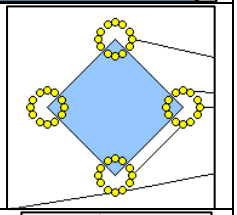


Debugging Tips:

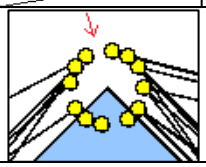
- If you do not see any vertices appear, then either your points list is empty (put in a print statement to check), or you did not add the points properly to the **hull** obstacle (read the instructions more carefully). There should be 12 nodes around each obstacle vertex.
- If your vertices do not form nice circles around the vertices as shown here, then you perhaps forgot to convert to radians when computing **sin()** and **cos()**. Note that even in the above "correct" image, there will still a couple of gaps due to round off errors.



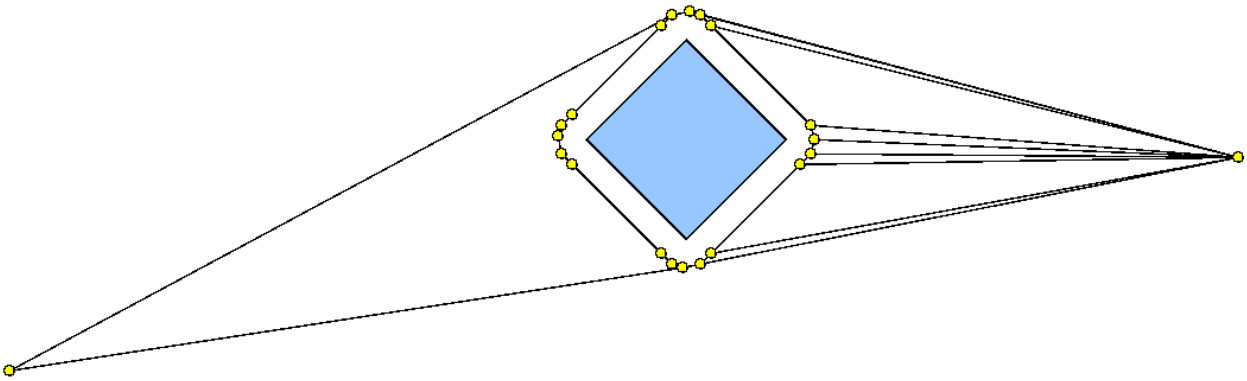
- If you are missing a lot of edges of the graph (as shown here), then you likely forgot to use the **ceil()** and **floor()** functions when adding the **x** and **y** offsets.



- If you are missing the edge joining the two top vertices, then you are probably using **radius** instead of the calculated **d** value or you are calculating **d** wrong.



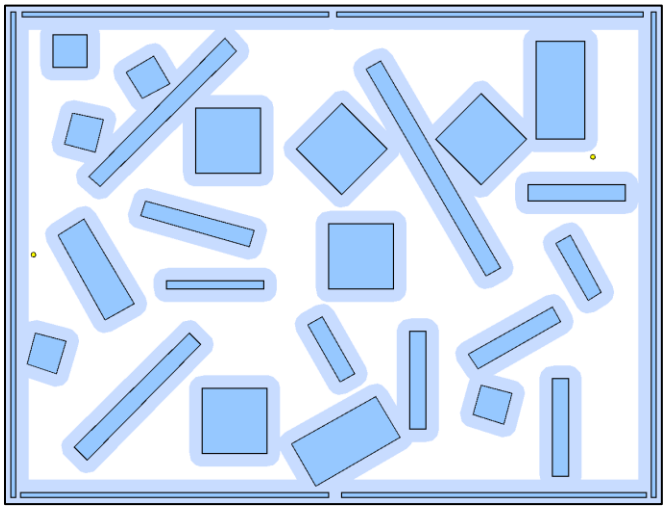
(3) Now it is time to get the convex hull code working "properly". In the **convexHull()** function, remove the few lines that you wrote which added ALL those circle points and then write code to apply the convex hull algorithm on those points (see slide 8). If you are having trouble figuring out how to determine the rightmost point … stop looking at your screen/code and simply think a little … what identifies the rightmost point in a set of points? Load up the **SingleBox.vmp** map again, select the **Show Original Obstacles** checkbox and then select **Show Grown Obstacle Visibility Graph** from the **PathPlanning** menu. The resulting graph should have **21** vertices and **41** edges (check by selecting **Graph Information** from the **PathPlanning** menu) as shown below … make sure to save it as **Snapshot2.png**.



Debugging Tips:

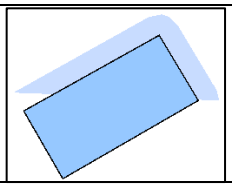
- If your code hangs, then you are in an infinite loop. Perhaps you did not select the rightmost point properly as your first hull point.

(4) Load up the **PathPlannerConvex.vmp** map and select the **Show Grown Obstacles** checkbox. You should see what is shown below (save it as **Snapshot3.png**):

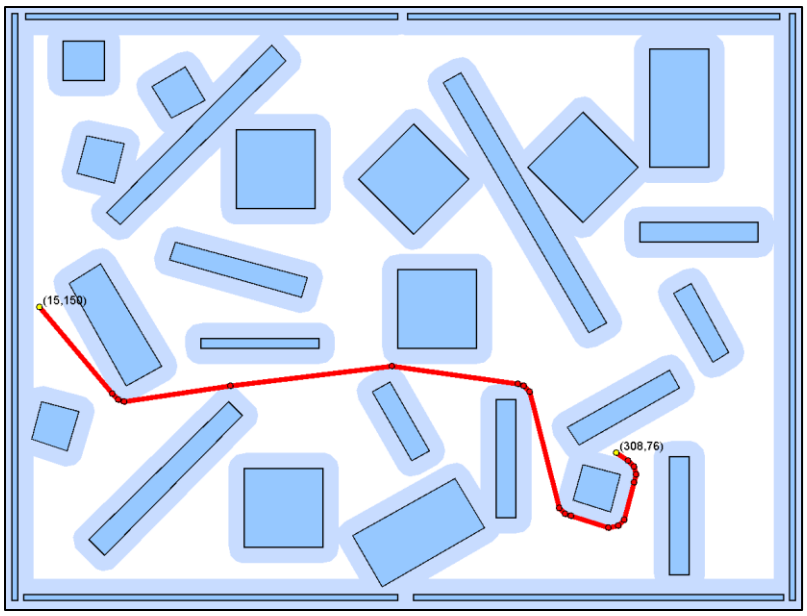


Debugging Tips:

- If some of the objects do not have grown borders or some are cut off as shown here on the right, then you are either stopping your **while** loop too early or not getting into the **while** loop at all. Check the logic very carefully, including your **while** loop condition.



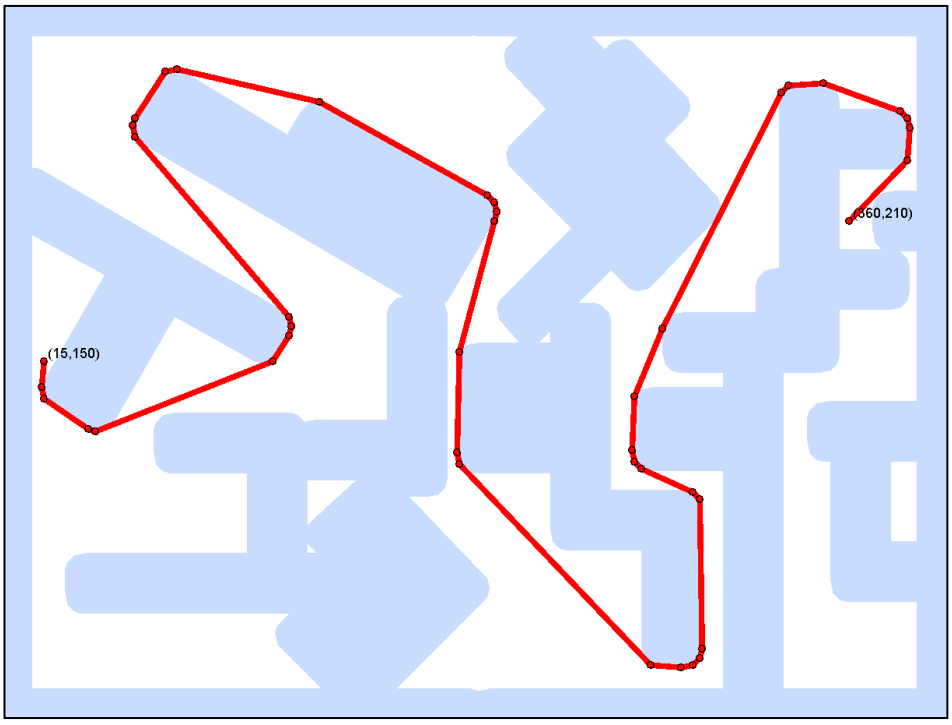
You can select the **Show Grown Obstacle Visibility Graph** radio button. You should not see any graph edges passing through the grown obstacles. You can then hide the graph (i.e., select **Hide All Graph Computations**) and then select **Show Shortest Path to Mouse Location** from the **ShortestPath** menu. You should see now that the shortest path remains outside the grown obstacle at all times. You need to do a mouse click to update the path to the latest mouse location. Click the mouse roughly at endpoint **(308, 76)** and save a snapshot of a path as **Snapshot4.png**:



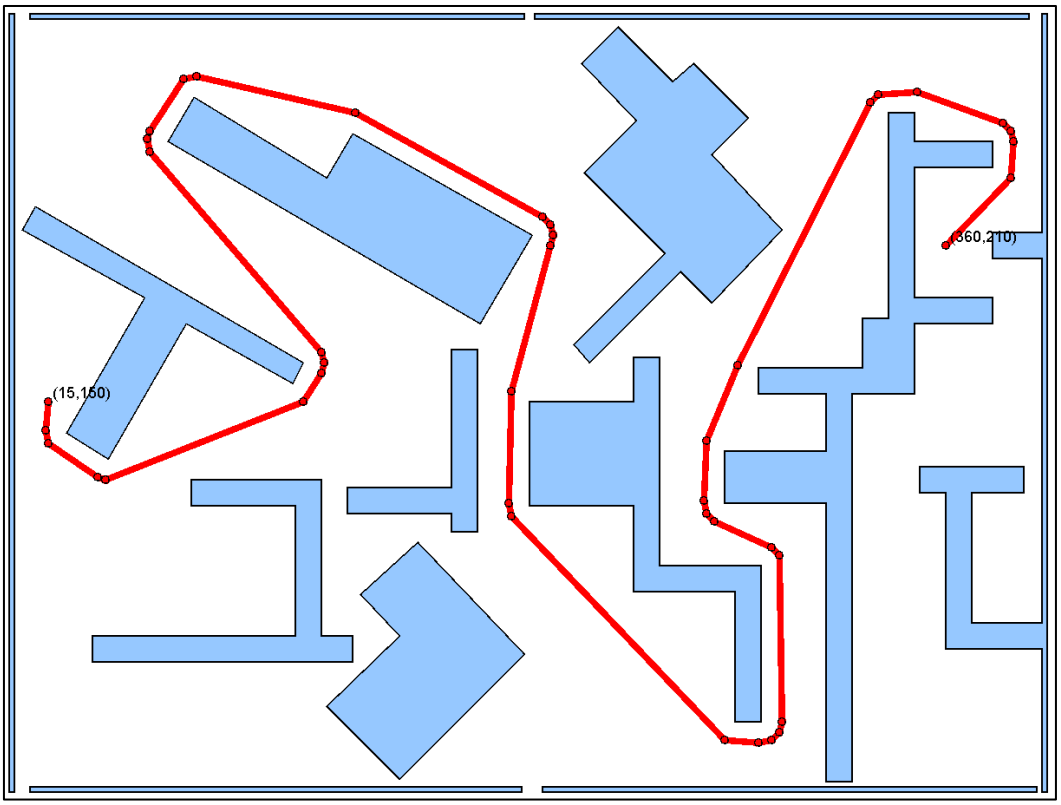
Debugging Tips:

- If you end up with a different path, for this end point, then you likely ended up growing the obstacles a little too big (or too small). Go back and check your **grow()** function to make sure that you used the proper circle radius when choosing your points.

Load up the **PathPlannerNonConvex.vmp** map and select the **Show Grown Obstacles** checkbox. You should see the grown obstacles again. This time … hide the original obstacles, select **Show Shortest Path** from the **ShortestPath** menu and save a snapshot as **Snapshot5.png**. It should look exactly as shown below.



From the **PathPlanning** menu, unselect **Show Grown Obstacles** and re-select **Show Original Obstacles**. Select the **Show Grown Obstacles Visibility Graph** radio button. Then select **Show Shortest Path** from the **ShortestPath** menu. Select **Hide All Graph Computations** from the **PathPlanning** menu. You should see what is shown below. This is the path that the robot can safely travel without hitting the obstacles. Save it as **Snapshot6.png**.



(5) Now we are ready to test the robot's ability to travel along this path without colliding. Copy your **Obstacle.java** source code into the **Part5/Lab18/controllers/Lab18Controller** folder. You may need to remove any package statements that your IDE inserted near the top of your file, if indeed it did so. Go to the **Part5/Lab18/worlds** folder and double-click on the **PathPlannerNonConvex** world top open up Webots. When opened, you will likely get the following error:

```
WARNING: Lab18Controller: Expected either: Lab18Controller.exe, Lab18Controller.jar, Lab18Controller.class, Lab18Controller.py, or Lab18Controller.m
INFO: Lab18Controller: Try to compile the Java source code, to get a new .class or .jar file.
WARNING: Lab18Controller: Starts the void controller instead.
```

This is because the code was not yet compiled. Compile it. If you get errors, it is likely because you did not put your **Obstacle.java** file in the proper controller folder … or you had compile errors leftover from parts 1 through 4. You may have to debug. Keep in mind that the other classes (e.g., Map, PathPlanner, Graph, Node, Edge, VectorMap) have also been changed from parts 1-4 so you SHOULD NOT have overwritten them.

The **Lab18Controller** code will be used to move the robot along the shortest path that you have computed in **Snapshot6.png**. The **VectorMap** class has been hard-coded to set up the **PathPlannerNonConvex** obstacles, so it does not load up any files. The code for moving the robot has been completed for you. You just need to supply the points along the shortest path that the robot needs to travel along. The code is similar to the inverse kinematics lab where the robot travelled from point to point. However, we will be using the exact robot location (and also using the compass to get the direction). The robot will NOT be using its collision sensors at all while it moves. It will blindly follow the path that you computed … so let's hope that you did everything correctly and accurately.

A **getShortestPath()** function has been started for you at the bottom of the code in the **PathPlanner** class. It will return an **ArrayList<Point>** of **Point** objects, called **sp**, that corresponds to the graph nodes along the shortest path from the graph's **start** node to its **end** node. You need to complete the code so that it returns the correct points along the shortest path. You will test everything with the shortest path shown in **Snapshot6.png**, which has **42** nodes along the path.

The **PathPlanner** has the shortest path's **start** and **end** point already defined as attributes. You just need to add points to the **sp** list, starting with the **end** point and working backwards towards the **start** point. This is described in slide 22 of the **Lab 17** notes (i.e., the shortest path lab that we did the last time). You will want to use a WHILE loop to keep adding points until either the **start** point is reached, or until a **null** node is found. Since the simulator needs the points to be in order from **start** to **end** … but you will be tracing the path backwards (i.e., from **end** to **start**) … you will need to always add points to the front of the array list.

One thing to keep in mind is that you need to trace back through the nodes of the visibility graph … which are **Node** objects, not **Point** objects. You should start at the node that corresponds to the **end** point. A function has been created in the **Graph** class called **node(x,y)** that allows you to find the **Node** object with the given **(x,y)** coordinate. Start by finding the end point's corresponding Node so that we can trace back in the graph from there. The **getPrevious()** method, in the **Node** class, returns the **Node** that comes before that one in the shortest path. As you trace back … you just need to add the node locations (as points) as you find them until you get back to the **start** point. Make sure that the **start** is also added as a point.

Before you return from the method, you can display (use **System.out.println**) the entire shortest path list. Make sure that it starts with point **(15, 150)** and ends with point **(360, 210)** … and it will have **42** points in total.

To test the code, you just need to run the **Lab18Controller**. Assuming that all went well … you should be able to see the robot moving along the path in the 3D world. It should move close to the object boundaries but it should not touch any obstacles. You can use the fast-forward button to make the robot go fast. You only get one run at it … then you need to reset the world and restart everything to try it another time. If the robot collides at any time … then you have a mistake in your computed path.

Debugging Tips:

- If your robot seems to bump quickly head on into a wall, then you likely didn't read the instructions above carefully. Check all the path points … they should start with **(15, 150)**.

Submit **ALL of your java code**, including the files that were given to you.

You will need to make two folders labelled **Parts1to4** and **Part5**. Please put all your .java files and snapshots into the appropriate folder. Then either zip both folders together or separately submit them.

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 you are using IntelliJ and are unsure how to do this, please see step **(10)** of the “Using the IntelliJ IDE” file provided.

Submit the **6** snapshot **.png** files.

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