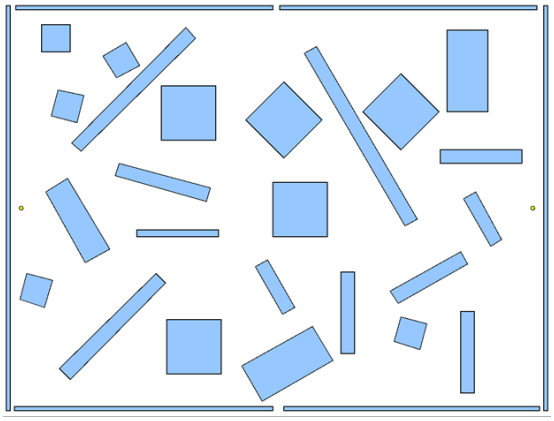
LAB 15 – Path Planning

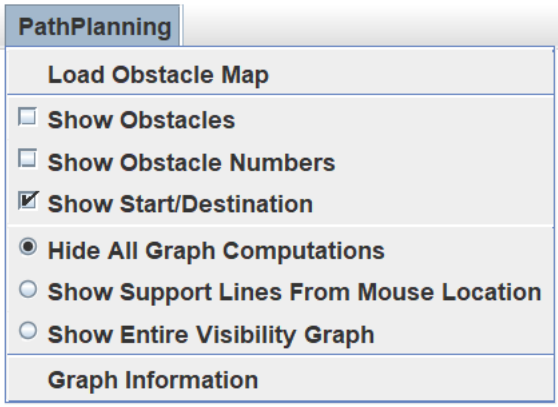
(1) The goal of this lab is not to move the robot around. Instead, we will be computing support lines and a visibility graph that can be used in a future lab to move the robot along a pre-defined path in the graph. Download the **Lab15_PathPlanning.zip** file and unzip it. Again, in this lab, we will not use the Webots environment. Instead, you will compile the **MapperApp** program separately, using any Java IDE that you like. We will be using a pre-computed vector map which was created from the Webots environment … both shown below. This vector map was manually built (painstakingly by me ☹) to match as closely as possible to a webots world of boxes.

PathPlannerConvex world



PathPlannerConvex.vmp vector map

All control of the program lies within the **MapperApp** user interface. The **MapperApp** now contains a single **PathPlanning** menu (shown here) →



The **Load Obstacle Map** button will open up a dialog box that will allow you to load up a pre-computed vector map called **PathPlannerConvex.vmp**. (There is one other test map called **SingleBox.vmap** that you may want to use when debugging. It has only a single box in it.).

There are many java classes provided:

- **MapperApp**, **MapperPanel** – the user interface files.
- **Map** – a grid map
- **VectorMap**, **Obstacle** – represents the obstacles.
- **PathPlanner** – you will write path-planning code here.
- **Graph**, **Node**, **Edge** – data structures to represent a visibility graph.

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

The map is represented using the **VectorMap** object from the previous lab, which is made up of a list of **Obstacle** objects that represent the wooden boxes from the environment.
The **Obstacle** class is the same as the last lab, except that it also has a VERY useful method called **contains(Point)** which lets you determine whether a given point is inside of the obstacle.

It makes use of a method from the **java.awt.Polygon** class on a pre-computed **Polygon**, but you don't need to know that to complete the lab.

The **Graph** object contains a list of **Node** objects which are the vertices of the graph. The edges of the graph will be stored within the nodes themselves, so the graph does not store them directly as a separate list. Glance over the available methods in the **Graph**, **Edge** and **Node** classes as you will make use of some of them, but you do not need to know the content of these methods … just what they can do for you.

The **PathPlanner** class is where you will work. Notice that a **PathPlanner** contains **start** & **end** **Point** objects. This is where the robot will start and end as it travels along a path. However, as mentioned, we will not be moving the robot in this lab session.

The **PathPlanner** also keeps a **VectorMap** called **vmap**, which will allow you to extract information from the obstacles. A **supportLines** attribute has been created. This will contain the **Points** that are visible from the mouse location, as you interact with the user interface. Finally, the **visibilityGraph** will contain the entire visibility graph that can be used for path planning in a future lab.

(2) To begin, you will use the algorithm in the notes (slides 5 and 6) to determine all support lines. A method called **computeSupportPointsFrom(ArrayList<obstacles>, x, y)** has been created in the **PathPlanner** class. You must write this method so that it determines the support lines for each obstacle (from the given **obstacles** parameter) with respect to the given (**x**,**y**) location … which is point **s = (x_s, y_s)** on slide 5. The method creates and returns a new **ArrrayList** of **Point** objects called **supports.** Once you write your code properly, **supports** should contain all support vertices for every obstacle. DO NOT worry about visibility at this time (i.e., DO NOT check for intersections) … just compute the left and right support points based on the algorithm in the notes.

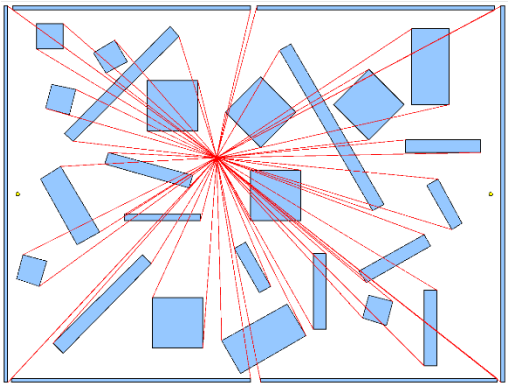
The (x,y) coordinate parameter will vary each time the method is called. As you go through an obstacle, **ob**, in the list, you can access each vertex by using `ob.getVertex(i)`, where `i` is the index of the vertex (from `0` to `ob.size()-1`). Given a vertex `i` in the list, you can access the vertex before it by using:

```
ob.getVertex((i-1+ob.numVertices())%ob.numVertices())
```

and the vertex after it by using:

```
ob.getVertex((i+1)%ob.numVertices())
```

Remember that each vertex is a **Point** object. You can access the **x** and **y** values of a **Point** object **p**, by using the dot operator (i.e., **p.x** and **p.y**). You will need to create a new **Point** object for each support vertex that you find. You can do this by calling the **Point** class constructor, passing in the **x** and **y** values as follows: `new Point(x,y)`. Points can be added to the **supports** **ArrayList** by just calling the **add()** method like this: `supports.add(p)`.



Once your feel that your code works, run the code, load up the vector map and then select the **Show Support Lines From Mouse Location** radio button in the **PathPlanning** menu. As you move the mouse around, you should see the support lines change. It should look something like what is shown here. Take a screen snapshot and save it to a file called **Snapshot1.png**.

Debugging Tips:

• If you don't see any support lines appearing, make sure that …
  ○ you are adding the support points that you found to the supports list.
  ○ you are adding the proper **IF** statement in the yellow box on slide 12.	

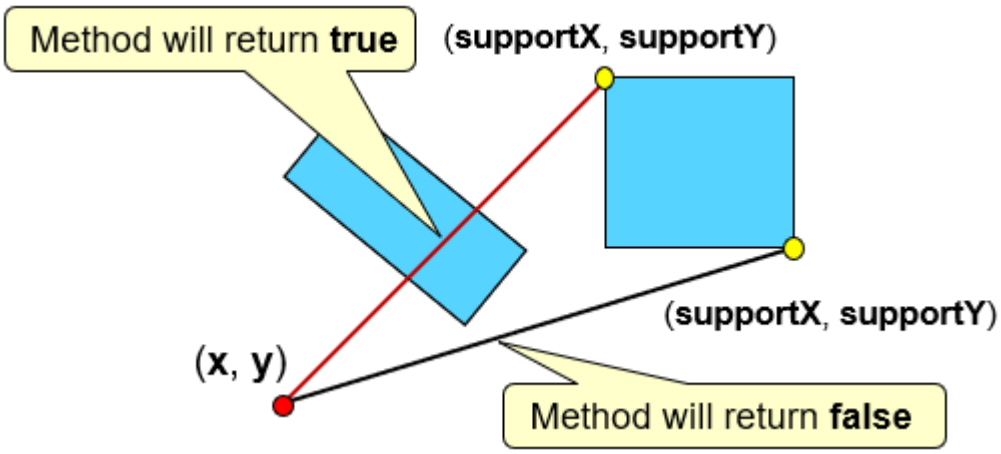
• If you find that the supports do not look correct … then you have something wrong in your formula or your IF statements. Try loading up the **SingleBox.vmp** map to debug. Note that you should have one support line also going to the end point, as shown here.	
• If you find that a support is sometimes missing from an obstacle … try loading up the **SingleBox.vmp** map to debug. If you see the situation shown here, then likely you made a mistake on your FOR loop boundaries. For an ArrayList of obstacles called **ob**, the indexing of **i** should be such that **0 ≤ i < ob.numVertices()** or **0 ≤ i ≤ ob.numVertices()-1**. You likely have something wrong here. Make sure to pay careful attention to ALL of your loop boundaries for the remainder of your coding.	

(3) Now you must modify the code so that it eliminates all the support points that are not visible from the (**x**,**y**) parameter's location based on the algorithm shown in slides 7 to 10 of the notes. Remember, a support line is to be discarded if it intersects any obstacle edge.

To make life easier, the following method should be used a helper function (a blank template has been given to you in the code):

```
supportLineIntersectsObstacle(int x, int y, int supportX, int supportY,
                              ArrayList<Obstacle> allObstacles)
```

Write this code so that it determines if the line segment (**x**, **y**)→(**supportX**, **supportY**) intersects one of the obstacles in the given **allObstacles** list. For example, consider the following two support lines. One of them will return **true** from the method (since there is an intersection with another obstacle), the other will return **false**:



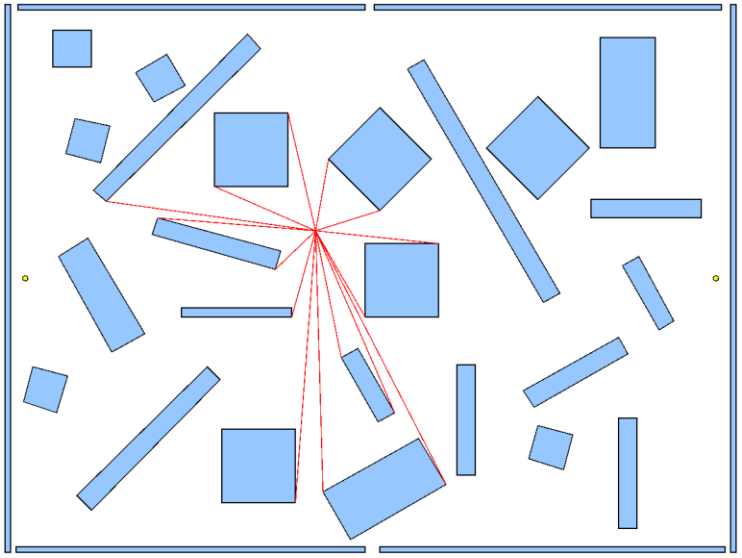
Rather than calculate intersections on your own, you will want to make use of the following "already existing" java library method which returns **true** if line segment (x1, y1)→(x2, y2) intersects line segment (x3, y3)→(x4, y4) and returns **false** otherwise:

```
java.awt.geom.Line2D.Double.linesIntersect(x1,y1, x2,y2, x3,y3, x4,y4)
```

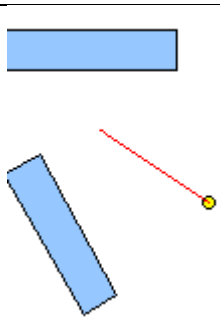
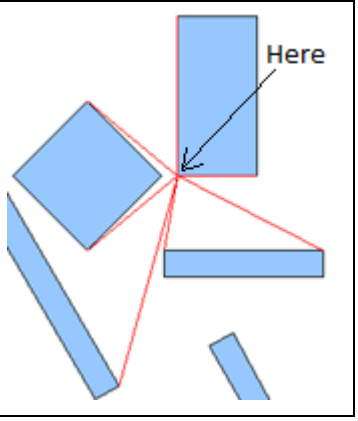
In your code, you will need to handle the special cases shown in slides 11 and 12 of the notes.

In your **computeSupportPointsFrom()** method, adjust your code so that it only adds the support points if the **supportLineIntersectsObstacle()** method returns **false**. Make sure to also **add** a support point to the path planner's **end** point, provided that the line from (**x**,**y**) to **end** is visible as well.

Once your feel that your code works, run the code, load up the vector map and then select the **Show Support Lines From Mouse Location** radio button in the **PathPlanning** menu. You should now see a reduced set of support lines and none of them should intersect any obstacles, except possibly along an obstacle edge. Make sure that the yellow end point on the right also has a support line if it is visible from the mouse location. Also, check the 2nd Debugging tip below to make sure that your code is working correctly. Take a screen snapshot and save it to a file called **Snapshot2.png**.



Debugging Tips:

• If you only see a single support line to the end location (or don't see any at all), then you likely make a mistake in your **IF** statement condition when checking if the support point was a vertex of an obstacle, as shown in the yellow box on slide 12.	
• Place the mouse cursor exactly at the bottom left corner of the box in the top right corner of the environment as shown here. You should see the given support lines. If none of the support lines show up, then you likely make a mistake in your **IF** statement condition when checking if the **(x, y)** point was a vertex of an obstacle, as shown in the yellow box on slide 12. However, if the mouse cursor is slightly inside the box, then you will not see the red lines (and this is ok) … so make sure that you are really checking the bottom left point exactly.	

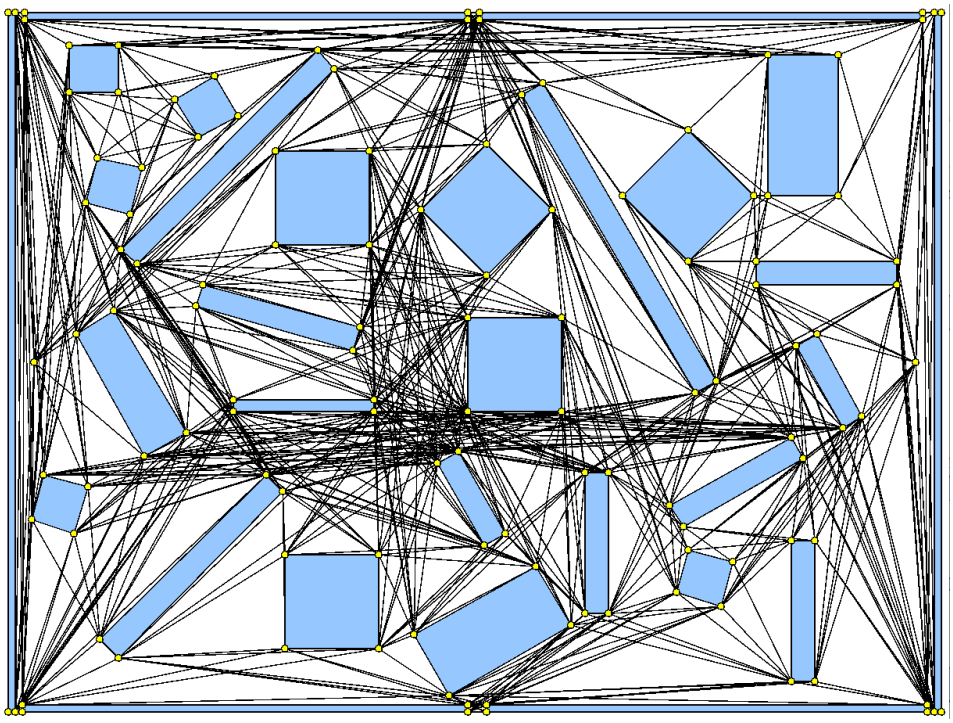
(4) Now you will compute the entire visibility graph based on slides 13 through 15 in the notes. A blank method called **computeVisibiltyGraph()** has been created. You must write this method so that it determines the visibility graph that includes the **start** and **end** locations.

The **vmap.getObstacles()** returns all the environment's obstacles. The **visibiltyGraph** attribute is set to a new **Graph** object each time that the method is called. This is the graph that you must build. You should make a graph node for the **start** and **end** locations as well as for each vertex of every obstacle.

Test your code at this point by loading up the **SingleBox.vmp** map and then selecting **Show Entire Visibility Graph** from the **PathPlanning** menu. You should see a small yellow circle for each obstacle vertex. Unselect the **Show Start/Destination** checkbox in the menu to ensure that your **start** and **end** are actually nodes in the graph. If they disappear when you unselect that option, then you forgot to add them to the graph. Select **Graph Information** from the menu. You should see that exactly **6** vertices have been created, with **0** edges. Load up the **PathPlannerConvex.vmp** map, show the visibility graph and then ensure that the **Graph Information** menu item indicates **122** vertices.

Now you will add the edges of the graph by making use of the **computeSupportPointsFrom()** method that you wrote. Compute the support lines only from the **start** node and add edges to the graph for each visible support line. You will need to use the **addEdge()** method in the **Graph** class. This method requires two **Node** objects to add the edge. So you will need to find the **Node** for each visible support point by using the **node()** method in the **Graph** class.

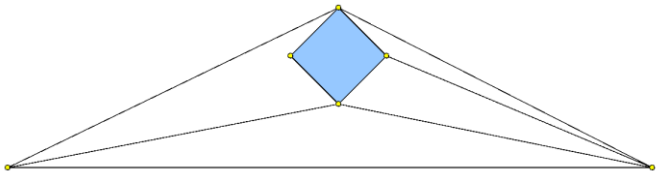
Make sure that your code works by loading up the **PathPlannerConvex.vmp** map and showing the visibility graph. Ensure that the **Graph Information** menu item indicates exactly **14** edges in the graph. Once you get this working, you should build up the graph by computing the support lines from every graph node (i.e., loop through them all). After you compute the graph, it should look like this:



Make sure to save a screen snapshot as **Snapshot3.png**.

Disable the **Show Obstacles** button in the menu to ensure that your obstacle edges are still showing as graph edges. Also, use the graph Information button to ensure that your graph has **122** vertices and **779** edges … otherwise … you have done something wrong.

You can use the **SingleBox** map to debug … which should have **6** vertices and **10** edges in the final map (4 edges are along the obstacle boundary):



Debugging Tips:

- If you are getting some edges that cross obstacles, then you likely made a mistake in your IF statement from slide 12. Some students get the logic wrong when checking if **s** has the same coordinates as **v** or **va**. Just check if the coordinates are equal … and put the NOT sign (i.e., !) in front of it … for each point comparison.
- If you find that you have **900** graph edges, then you are adding extra edges. It is likely that you are adding edges from a node to itself. Make sure that you only add a visibility graph edge between two nodes if the nodes are not the same.

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 **3** snapshot **.png** files.

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