

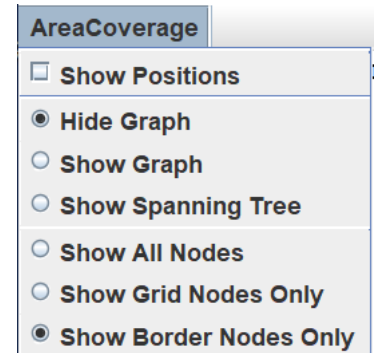
## LAB 21 – Improving Coverage

- (1) The goal of this lab is to improve upon the area coverage map from the last lab by adding additional branches to the spanning tree that will cover the areas along the borders of the obstacles.

Download the **Lab21\_ImprovingCoverage.zip** file and unzip it. The code will be completed in two parts. For parts 1 to 7, you will work in your Java IDE as you have been doing in recent labs. Then for part 8, you will copy over your code and run it in Webots to watch the robot travel along your computed path.

The **MapperApp** contains a modified **AreaCoverage** menu that now allows a spanning tree to be computed using **Grid Nodes Only**, **Border Nodes Only** or **All Nodes**.

You will want to open the **AreaCoverage**, **Node**, **Edge**, **Graph** and **Obstacle** classes for use in this lab. The **AreaCoverage** class has been modified to contain solution code from the last lab in that it creates a spanning tree with the grid nodes only. The functions have been adjusted. For example, the **computeGridGraph()** function first calls **addsGridNodesAndEdges()** (which creates the **gridGraph** that we used in the previous lab) and then calls **addBorderingNodesAndEdges()** which adds the extra nodes to the spanning tree to cover the missed areas.



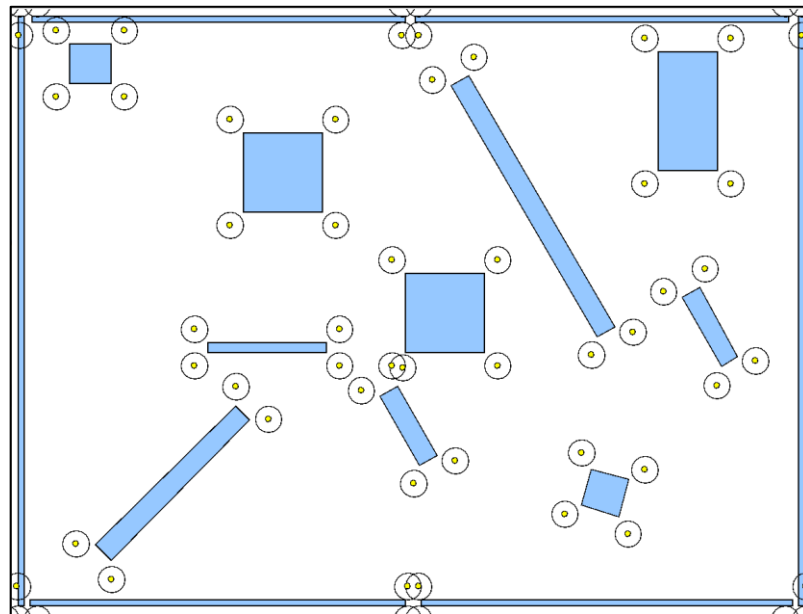
In the previous labs we used just the **Point** class for everything, which had integer precision, but for this lab we will need the double precision. Therefore, nearly all points have been changed to **java.awt.geom.Point2D.Double** throughout the code. These are point objects that have double precision in the coordinates.

- (2) Code has already been written in the **addBorderingNodesAndEdges()** method of the **AreaCoverage** class which has been broken down into three steps. First, it calls the **computeExtendedVertices()** helper method which creates the 4 extended bisector (i.e., “corner”) vertices for each obstacle based on the formulas that are in slide 4 of the notes, where **r** is **ROBOT\_RADIUS** and **ε** is **ROBOT\_BORDER\_TOLERANCE**. Each of these corner vertices are stored in the **extendedVertices** attribute of the **Obstacle** class as an object of type **java.awt.geom.Point2D.Double**. When you need to get this extended vertex point for vertex **i** of obstacle **ob**, you should call **ob.getExtendedVertex(i)**.

In the second step, it adds the nodes along the edges of the obstacle by calling the **addNodesAlongEdges()** method. In the last step it removes the invalid nodes that intersect either the boundary or other obstacles by calling the **findBadNodes()** method. You will be completing both of these methods.

Complete the code in the **addNodesAlongEdges()** method so that it loops through the obstacles, gets the extended vertices for that obstacle and then adds a new **Node** in the **gridGraph** for that extended vertex. The **gridGraph** is the **Graph** object to which we are adding nodes and you can use **addNode()**, provided that you create a **Node** object that corresponds to the extended vertex that was created.

Once you have this compiled properly, run the code, which will automatically create the hard-coded **AreaCoverageConvex** world. Select **Show Positions**, **Show Graph** and **Show Border Nodes Only**. If all is working well, you should see what is shown below. Look at the console output ... it should indicate that the graph has **64 nodes** and **0 edges**. Save a snapshot as **Snapshot1.png**.



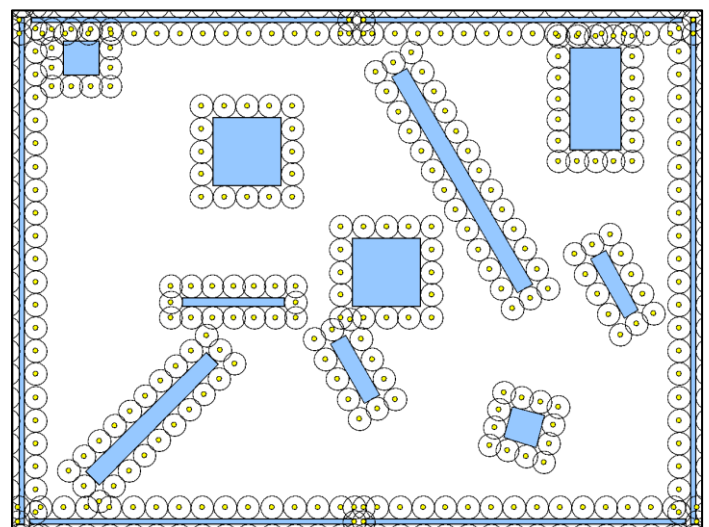
- (3) Now, you need to adjust the code so that it adds additional graph nodes along the edges of the obstacles (see pseudocode on slide 5). You do not need to go through the obstacle edges, you just need to consider the pairs of extended vertices  $\mathbf{v}_i$  and  $\mathbf{v}_{i+1}$  where  $0 \leq i < \text{obstacle size}$ . You can loop through all adjacent pairs of extended vertices and access the extended vertex points. You can access the extended vertices for vertices  $i$  and  $i+1$  of obstacle **ob** as follows:

```
java.awt.geom.Point2D.Double vi, vip1;

vi = ob.getExtendedVertex(i);
vip1 = ob.getExtendedVertex((i+1)%ob.numVertices());
```

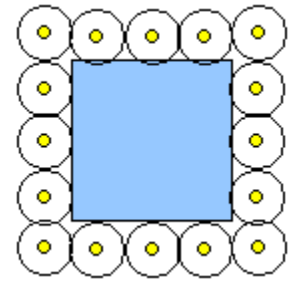
You just need to add the appropriate nodes with the correct gap between them. You can use the distance function in the **java.awt.geom.Point2D.Double** class to calculate the distance between two points like this: `vi.distance(vip1);`

When you believe your code is working, run it and select **Show Border Nodes Only**, **Show Positions** and then **Show Graph**. If all is working well, you should see what is shown here. You should see even spacing of the nodes along the edges. Look at the console output ... it should indicate that the graph has **402 nodes** and **0 edges**. Save a snapshot as **Snapshot2.png**.

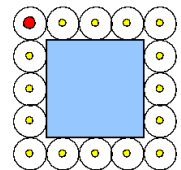


## Debugging Tips:

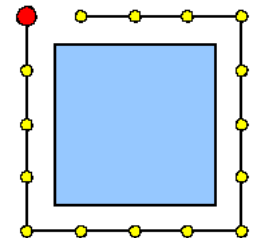
- If you notice that some of your circles are touching the obstacles as shown here, the problem is that you are using integer coordinates for your points instead of doubles. When computing your  $(X_k, Y_k)$  points from slide 5, make sure that you use variables of type **double** and that you are NOT typecasting to **int**. You will also need to make sure that you are using the class called **java.awt.geom.Point2D.Double** and NOT the class called **Point** when you are making new points. Fixing these will align your circles properly.



- (4) Now, we need to connect the nodes with edges by adding code to the same loop that you created in part 3 by connecting the edges as you add the nodes. For the first node of each obstacle (which is the first corner vertex node... we will call it the startNode) we would like to select that node for later. In your code, before you loop through an obstacle's vertices, create a lastNode variable that gets set to **null**. After you create your first node, call **setSelected(true)** to highlight that node, but then update lastNode to be that created node. Run your code. You should end up seeing the first extended vertex of each obstacle appearing larger and red like what is shown here on the right. You should have exactly one extended vertex highlighted for each obstacle, otherwise you did something wrong. For the top and right boundary obstacles, you won't see any though, since they are off the window.



Now, looking at your code that you already wrote for adding the nodes along the edges, as you add each node, you should update the lastNode variable to this new node so that you can keep track of this as a "previous" node so that we can connect it to the "next" node with an edge. Therefore, as you add the nodes, you can simply connect the new node to the lastNode by adding an edge to the graph that joins them (i.e., use the graph's **addEdge()** function). Once you get to the end of an obstacle edge you need to connect that lastNode to the next obstacle edge's startNode as well. Keep in mind that the first time through the loop, it will be a special case because your lastNode variable will be **null** when you start. As you go from one obstacle edge to the next ... make sure to reset the lastNode back to **null** so that you start fresh again. Run the code and select **Show Graph**. You should see a "ring" of nodes and edges around each obstacle as shown here (unselect **Show Positions**, select **Show Graph**, select **Show Border Nodes Only**). If you do not see this, then you did something wrong.

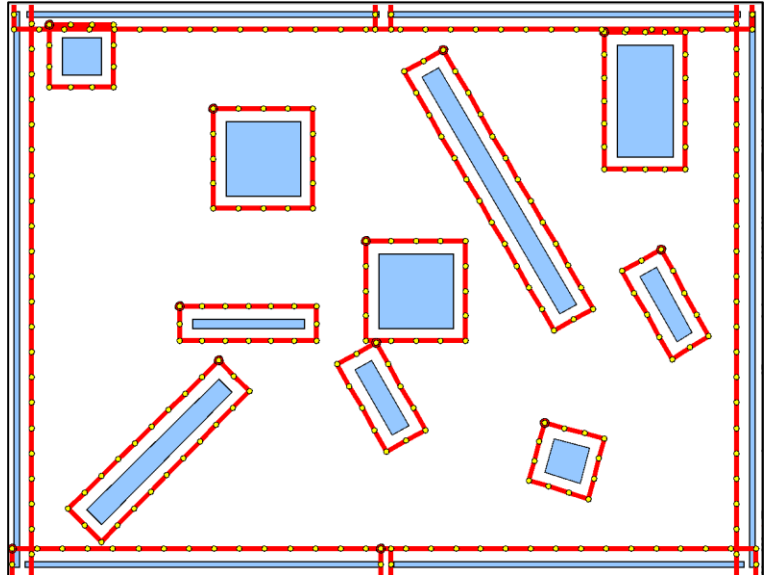


Finally, as you complete each obstacle, add one more **Node** to the graph for that obstacle which is a "copy" of the startNode, based on the first extended (i.e., corner) vertex 0. **Do NOT select it.** Add an edge to the graph from the "lastNode added from all the obstacle's edges" to this "copy node" (see slide 6 in the notes). Look back at your code and wherever you added an edge (should be in 3 spots), call **setSelected(true)** on the edge so that it is highlighted as thick red.

When you believe your code is working, run it and select **Show Graph**. If all is working well, you should see what is shown here.

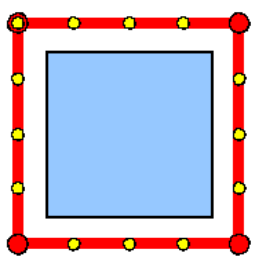
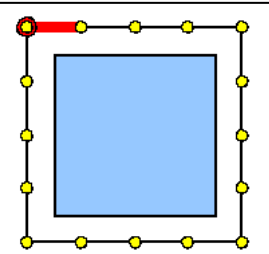
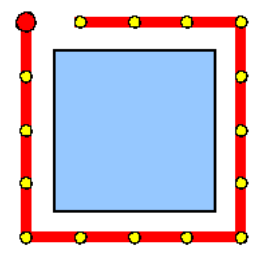
You should see exactly one large red node for each obstacle as the starting node and it should have an overlapping yellow node on it (which is the copy node). Some nodes will be outside the window boundaries and therefore not shown.

Looking at the console, you should see that the graph has **418** nodes and **402** edges. Save a snapshot as **Snapshot3.png**.



### Debugging Tips:

- |   |  |
|---|--|
| <ul style="list-style-type: none"> <li>If you have an infinite loop ... or your program does not seem to stop ... it is likely that you are calling the <b>equals()</b> method for the points or nodes. You should NOT be using that since it checks “identity” instead of “equality”. Instead, compare each of the <b>x</b> and <b>y</b> coordinates for equality.</li> </ul>  |  |
| <ul style="list-style-type: none"> <li>You should see a single large red circle at exactly one of the nodes of each obstacle. There should be a yellow dot in it. If you are missing the large red circle, then you forgot to select the <u>startNode</u>. If you are missing the yellow dot in the <u>startNode</u>, then you forgot (or did not add correctly) the <u>copy</u> of the <u>startNode</u> after you finished each obstacle.</li> </ul> |  |
| <ul style="list-style-type: none"> <li>If you see an obstacle disconnected like this, then you forgot (or did not do correctly) to add the edge from the <u>lastNode</u> to the starting corner node for the next obstacle edge. If, for example, you add three nodes along the edge between bisectors <b>vi</b> and <b>vip1</b>, then you need to add an edge from the last node (of the three you added) to <b>vip1</b>.</li> </ul>                 |  |
| <ul style="list-style-type: none"> <li>If you are getting this octagon shape, then you are connecting the first and last nodes that you added to the wrong start and end nodes. If, you add three nodes along the edge between bisectors <b>vi</b> and <b>vip1</b>, then you need to add an edge from the first node (of the three you added) to <b>vi</b> and add an edge from the last node (of the three you added) to <b>vip1</b>.</li> </ul>     |  |

<ul style="list-style-type: none"> <li>If you see more than one red node, then you are selecting too many nodes. You only want to select the first “corner” node of the <u>first</u> edge of each obstacle. Likely you are selecting the first “corner” node of <u>each</u> obstacle edge by mistake.</li> </ul>	
<ul style="list-style-type: none"> <li>If you do not see red edges, or just see some of them but not all of them, then you are forgetting to select the graph edges as you add them. There are likely three places that you are adding edges in your code. Make sure to select that edge each time.</li> </ul>	
<ul style="list-style-type: none"> <li>If everything looks right except that you are missing the last edge as shown here, then you forgot (or did not do correctly) to add the last edge connecting the <u>lastNode</u> to the <u>startNode</u>’s copy once the obstacle is completed. You likely forgot to add the copy vertex as well.</li> </ul>	
<ul style="list-style-type: none"> <li>If you get a <b>NullPointerException</b> when trying to select an edge, it is because the edge is <b>null</b>. That is likely because you are trying to add a duplicate edge to the graph but the <b>addEdge()</b> function checks for that situation and prevents it. So, you are likely connecting the wrong <u>lastNode</u> to the wrong <u>startNode</u>, or something like that</li> </ul>	

(5) Now it is time to eliminate all vertices that would cause an intersection between the robot and an obstacle. The following function has been written for you already:

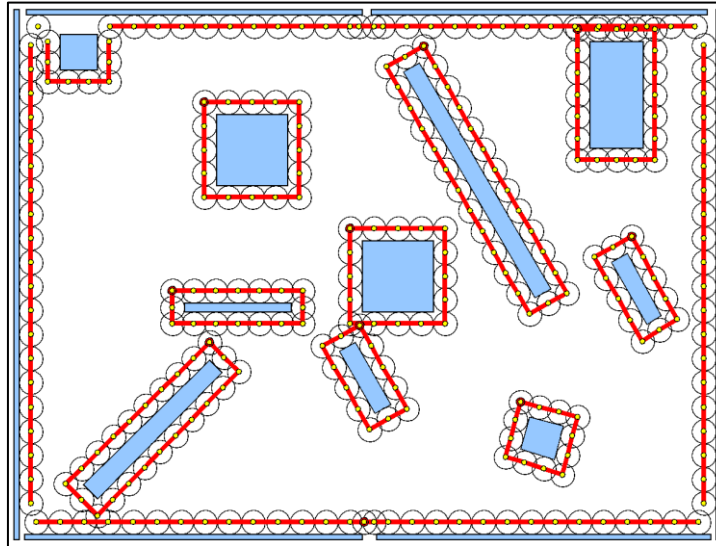
```
public boolean isInvalid(Node n, ArrayList<Obstacle> obstacles,
                        double width, double height)
```

The function takes a node **n** and determines whether it intersects an obstacle in the given **obstacles** list or if the node is outside of the boundaries defined by **width** and **height** (provided as parameters to the **addBorderNodesAndEdges()** function). It returns **true** if **n** intersects and **false** otherwise.

You need to complete the **findBadNodes()** method. It begins by creating an empty ArrayList of “bad” nodes and returns it at the end of the method. You need to add to this list all of your newly-created nodes that are considered to be invalid according to the **isInvalid()** function (see slide 7 in the notes). To begin, you should use `gridGraph.selectedNodes()` to get the **selectedNodes()** from the graph. There should be exactly one selected node for each obstacle in the graph. It represents the startNode of the “loop” of nodes around a single obstacle’s boundary. You should loop through these selected nodes one by one. This will allow you to handle each obstacle one by one. You need to trace the graph edges along this obstacle “loop” from the startNode and check which ones are bad, then add them to the **badNodes** list.



When you believe your code is working, run it. Select **Show Border Nodes Only**, **Show Positions** and then **Show Graph**. If all is working well, you should see what is shown on the next page. Look at the console output ... it should indicate that the graph has **270** nodes and **252** edges. You should notice that all the nodes outside of the border have been removed. Also, take note of the removed nodes in the left corner of the environment. You will also see that the “loops” along the environment’s border are now “strips” with no start node highlighted. Save a snapshot as **Snapshot4.png**.

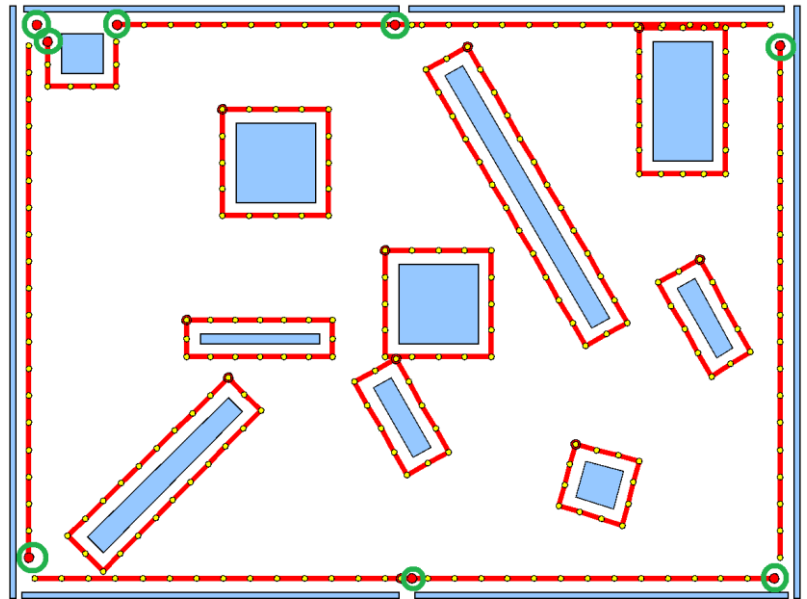


### Debugging Tips:

- If you see in the console that the graph has **276** nodes and **253** edges, then you likely forgot to handle the case after the while loop where the currentNode might be invalid. You will also notice that some nodes still exist outside the left and top borders.
- If you get an **IndexOutOfBoundsException**... could it be that you forgot to ensure that the currentNode has more than one edge? Maybe you wrote that case incorrectly.
- If your code seems to hang or go into an infinite loop, perhaps you have a loop that does not have a correct stopping condition. Did you make sure that you checked the case where the currentNode has the same coordinates as the start? Note that you must check the coordinates ... because the last currentNode will likely be the “copy” of the startNode ... so it will not be the identical **Node** object ... it will just have the same coordinates.

- (6) You will need to now adjust the code that you just wrote so that each time a “loop” is broken, you highlight (i.e., **select**) the next valid node in the loop. Just keep a **boolean** flag indicating whether you want to select a node as you are traversing through them. Start with a value of **true** (so that the first node is selected ... although it is already selected by default). If you find that the currentNode is valid and the flag is set to **true**, then select the currentNode and set the flag to **false**. If you find an invalid vertex, set the flag back to **true**. As a result, you should end up selecting the first node of each connected component (see green nodes of slide 8 in the notes).

When you believe your code is working, run it and select **Show Graph**. If all is working well, you should see what is shown below. Look specifically at the vertices that I have circled in green to make sure that they are now selected (i.e., red). Look at the console output ... it should indicate that the graph has **270** nodes and **252** edges. You should notice that all the unbroken loops still have the copy (i.e., yellow node over top the start (i.e., red selected) node). However, for each broken loop, each connected component will have a single selected node with no copy (i.e., yellow) on it.



Adjust your code so that you use another **boolean** which indicates whether the loop has been **broken**. Set it to **false** when you begin looking through the obstacles (i.e., same location that you set your other **boolean** to **true** when starting). It should get set to **true** if you find that the currentNode is invalid. After you check the whole obstacle (i.e., after your WHILE loop), if the loop has been **broken**, then you need to remove that start node's "copy" node and connect the previousNode of the last currentNode to the original startNode, so that there is no duplicated start node (see slide 9 in the notes).

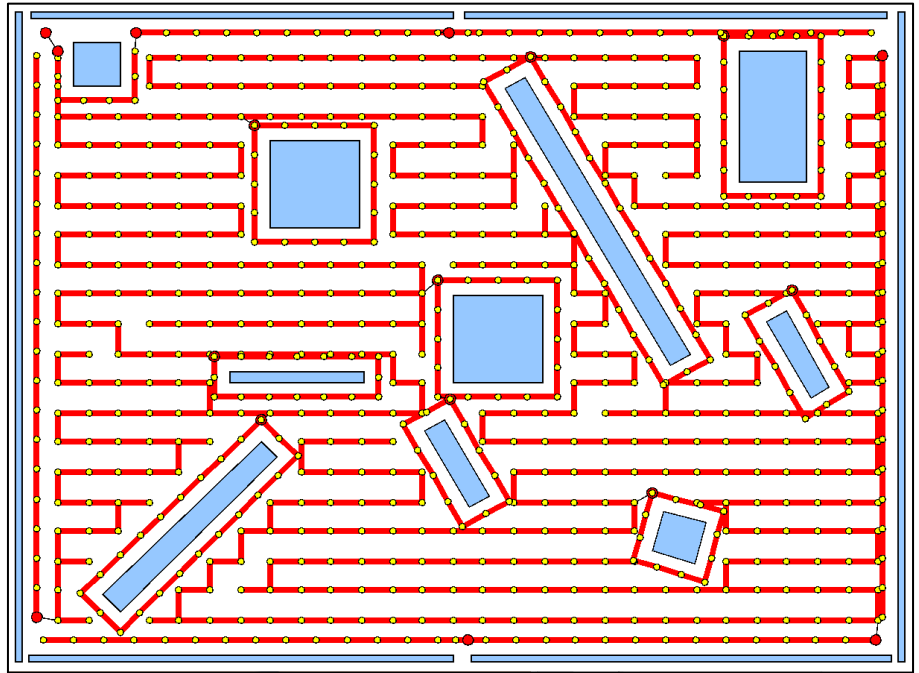
When you believe your code is working, run it and select **Show Graph**. If all is working well, you should see almost what is shown earlier except that there will just be one selected vertex at the bottom center, whereas previously there were two selected.



If you look at the console output ... it should now indicate that the graph has **269** nodes and **252** edges. Save a snapshot as **Snapshot5.png**.

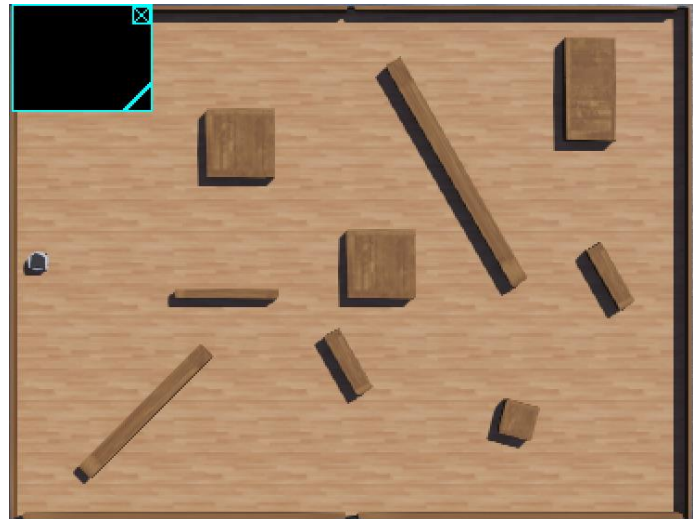
- (7) Now we just need to connect the selected startNode of each of these connected components to the grid graph's spanning tree. Since the only selected nodes in the graph are the start nodes for each connected component, then merging these to the spanning tree is easy. We start by computing the grid's spanning tree. Then for each selected node we check all nodes in the graph for the closest node to it and then connect that selected node to its closest node with a new spanning tree edge. We just need to be careful to NOT include those "copy" nodes for the full loops when we are looking for the closest node.

Since you have done enough work already, this task has been completed for you. Just run your code again, unselect **Show Positions**, select **Show All Nodes** and then **Show Spanning Tree**. If all is working well, you should see what is shown here. Save a snapshot as **Snapshot6.png**. You should notice that each start node (i.e., selected red node) is connected to the spanning tree by an unselected edge. Some connections will be impossible to see though, as they are behind other selected edges or nodes. The console should indicate **712** nodes and **1014** edges.



- (8) Now we want to get the robot to travel along the spanning tree and see how well it covers the environment. You will now need to work within the **webots** environment instead of your Java IDE. You will need to load up the code for this portion of the lab by double-clicking the **AreaCoverageConvex** world in the **Lab21** subfolder of the **Part8** portion of the given code. You should see the world appear like this.

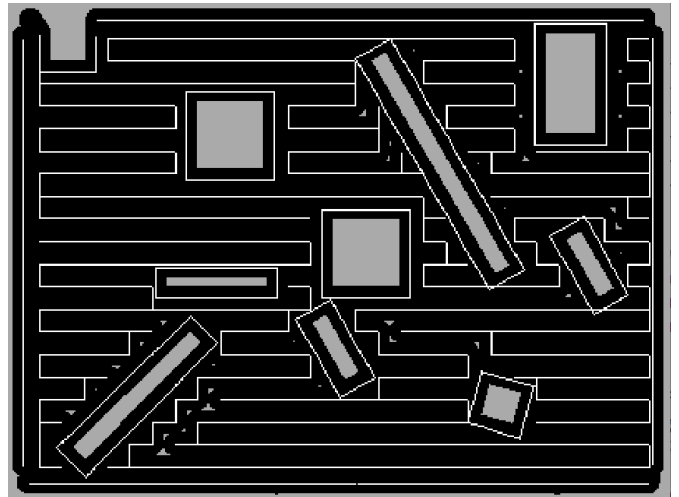
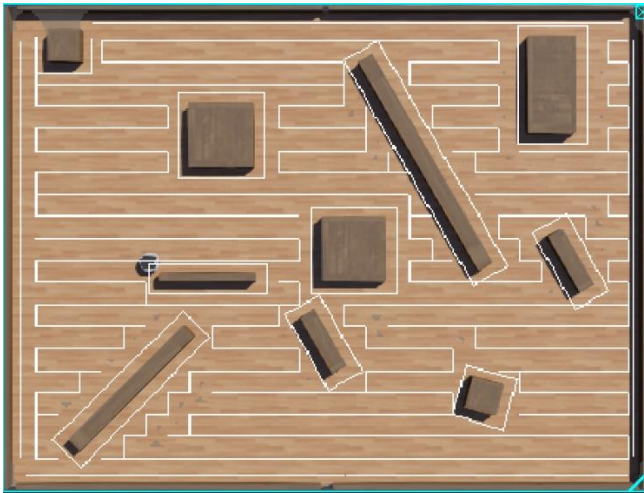
You will get a runtime error upon loading as you do not have a compiled **AreaCoverage** class in the **Lab21Controller** folder. Copy your **AreaCoverage.java** file into the **Lab21Controller** folder and then open that file in the webots editor and compile it.



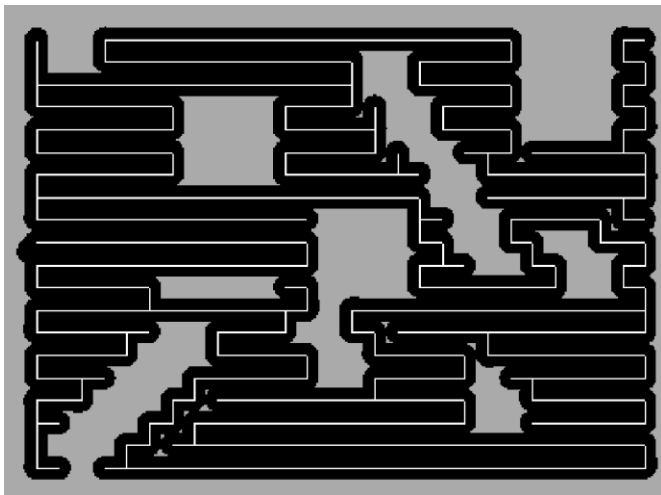
Run the code on fast forward. Stretch the display window to cover the environment “perfectly”. You should see the robot start moving in the Webots environment. As in the last lab, you should see the robot traversing along the spanning tree and now also traveling along the borders of the objects as it gets near them. Let the algorithm run to completion. If you computed all the points correctly, you won’t see the robot bump into any obstacles. The result should look something like what is shown below. Double-click the display window so that it appears in a separate window (shown below on the right). Save a snapshot of this black and gray window as **Snapshot7.png**.

It is possible that webots may crash. I found that setting the **MAX\_SPEED** to **3** was necessary because other speeds tended to cause the robot to bump into walls.

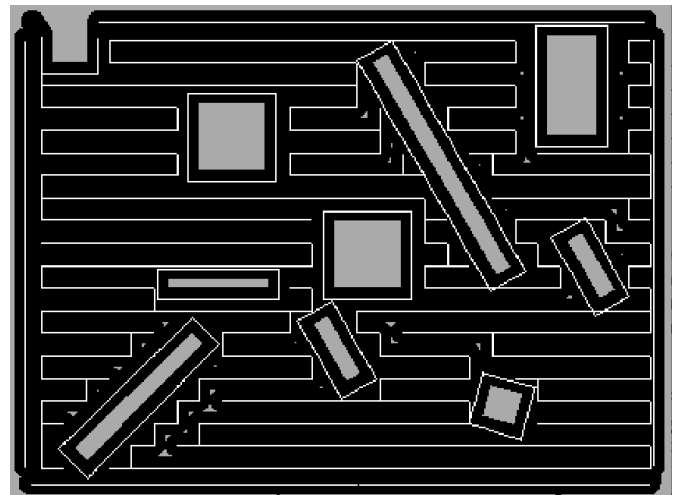




Compare side-by-side with the coverage from the last time ... was it worth all this effort to cover along the obstacle borders?



With Only Grid Coverage



With Border Coverage

Do you think that there is a way to ensure that there are no gaps in between adjacent back & forth “strips” of coverage? What about the spots that are missing due to the robot turning around ... can those be fixed? There is no need to submit those answers ... just think about it.

Submit your **AreaCoverage.java** file ... as well as the **7** snapshot **.png** files. Make sure that your name and student number is in the first comment line of your code.