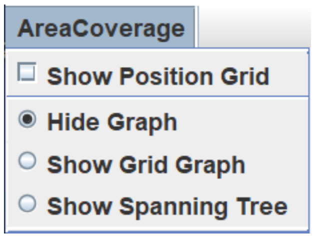
LAB 20 – Area Coverage

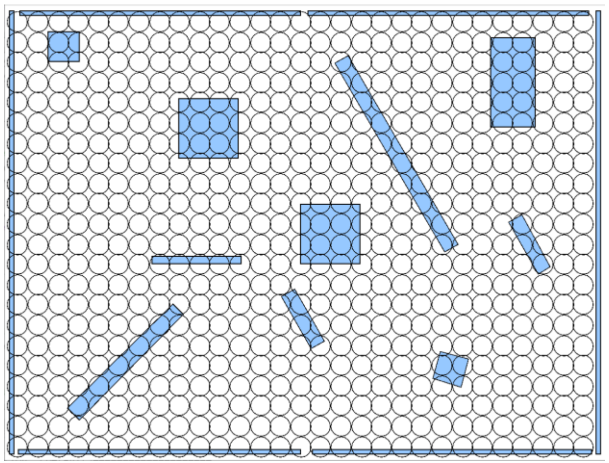
(1) The goal of this lab is to produce a path that will allow the robot to cover the floor area as much as possible by following nodes in a spanning tree based on a grid approximation of the environment. Download the **Lab20_AreaCoverage.zip** file and unzip it. The code will be completed in two parts. For parts 1 to 5, you will work in your Java IDE as you have been doing in recent labs. Then for parts 6 to 8, you will copy over your code and run it in Webots to watch the robot travel along your computed path.

The **MapperApp** now contains only an **AreaCoverage** menu. The **Show Position Grid** option allows the circular robot positions to be computed and shown. The **Show Grid Graph** creates and displays a grid graph in the environment and the **Show Spanning Tree** option creates and displays the reduced graph that represents the spanning tree. You will not need to load up any maps for this lab. A hard-coded vector map will automatically be generated and displayed when you run the code.



You will want to open the **AreaCoverage**, **Node**, **Edge**, **Graph** and **Obstacle** java classes for use in this lab. The **AreaCoverage** class has been created that contains the base code that will be used to run the algorithm. A **gridGraph** attribute has been added to the class. This will represent the graph to be used by the coverage algorithm. After computing the spanning tree, this original **gridGraph** will have been reduced in size in that the non-spanning-tree edges will have been removed.

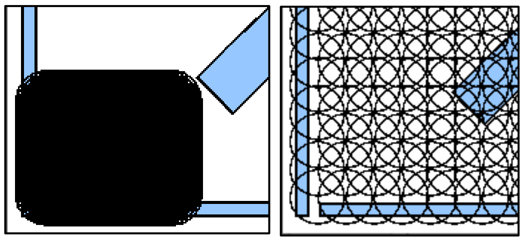
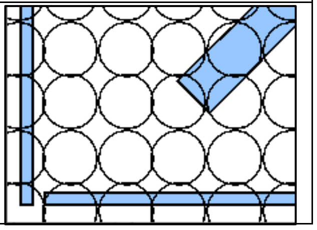
(2) In the **AreaCoverage** class, the **computeGridGraph()** method has been created, but is incomplete. The method first gets the **obstacles** and the environment's **width** & **height** from the map. You will first add code that just produces the nodes of a basic 2D grid graph with as many nodes as possible that fit side-by-side based on the ROBOT_DIAMETER constant (see slide 4 and first part of slide 5 of the notes). Make use of the **Graph** class and **Node** class constructors as needed … making sure to store your graph in the **gridGraph** attribute of the **AreaCoverage** class! DO NOT add any edges yet. The bottom-left node of the graph must be placed at position (x, y) = (ROBOT_RADIUS, ROBOT_RADIUS). For now, do not remove any nodes that intersect obstacles.



Run your code. Select the **Show Position Grid** option in the menu. You should see what is shown here. Take a snapshot and save it as **Snapshot1.png**. Note that there will be a little bit of a margin around the outside of the environment … so your bottom left circle will not touch the left and bottom borders.

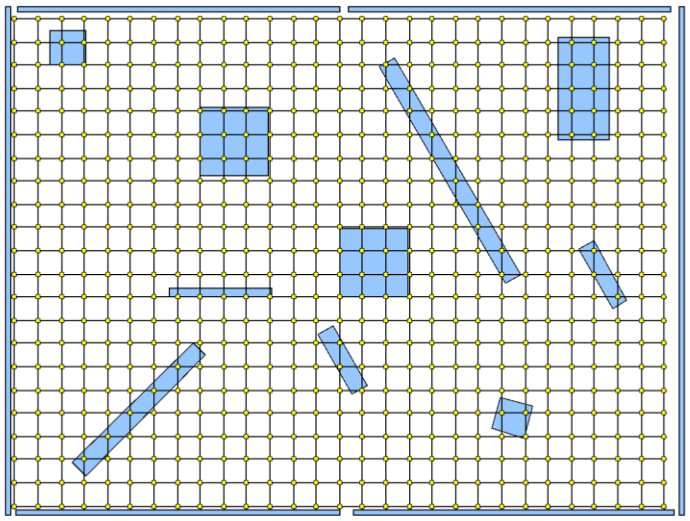
Debugging Tips:

- If you don't see any circles … then you probably didn't add the nodes to the **gridGraph** properly.

• If you end up with either of these situations in the bottom left corner, then you did not separate the X and Y coordinates from each other by the robot's diameter.	
• If you end up with this in the bottom left corner, then you didn't offset all your vertices by the robot's radius.	

(3) Now you need to adjust your code so that the nodes that you created are joined with edges for all nodes adjacent to each other horizontally or vertically (see slide 4 and second part of slide 5 of the notes). Make use of **Graph** methods when doing this.

Run your code. Select the **Show Grid Graph** option in the menu. You should see what is shown here. Take a snapshot and save it as **Snapshot2.png**.



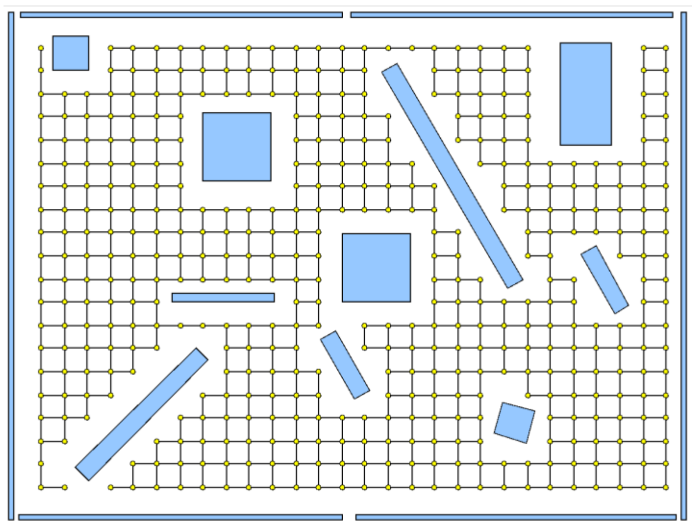
Debugging Tips:

• If you end up with an **ArrayIndexOutOfBounds** exception, then you are likely going 1 too far in your rows or columns.

(4) Now you need to remove all nodes that represent locations that the robot would intersect an obstacle if placed there (see slide 6). Go through all **nodes** and compare them with all the obstacles in the **obstacles** ArrayList. Follow the pseudocode of slide 7. Make use of the **contains()** method in the **Obstacle** class to determine if a node lies within an obstacle. You must also check if the robot would intersect any obstacle edge if the robot was placed at that node's location. You will need to check if the node is within a certain distance from each obstacle edge. Instead of writing all that fun math from slide 7, there is a nice function already created for us that computes and returns the distance from a line segment `(x1,y1)`→`(x2,y2)` to a point `(x,y)` as follows:

```
java.awt.geom.Line2D.ptSegDist(x1, y1, x2, y2, x, y)
```

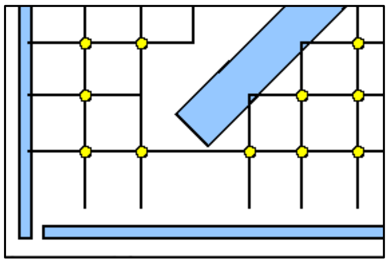
You cannot remove nodes from the graph while you are looping through them because you will get a **Concurrent Modification Exception** in java. Instead, you will need to make a new ArrayList of all nodes that are "bad". Once you have the list of "bad" nodes … you can loop through that list and delete the nodes from the graph one by one by calling an appropriate method in the **Graph** class.



Run your code. Select the **Show Grid Graph** option in the menu. You should see what is shown here. Take a snapshot and save it as **Snapshot3.png**.

Debugging Tips:

- If you end up with something like this → at the bottom left corner of the map, then you are not removing the edges properly. Likely, you removed the nodes from the graph's list of nodes by using the **remove()** method. However, an edge that connects two nodes in the graph is actually stored twice … once in the edge's start node's list of incident edges and another time in the edge's end node's list of edges. By removing nodes only from the graph's **nodes** list, this does not go to each of that node's neighbours to remove the edge. So, that is why you see some "dangling edges" here that have no node at the other end. The **Graph** class has a **deleteNode()** method that does this deletion for you … so use that method on the graph instead of using **remove()** on the graph's nodes.



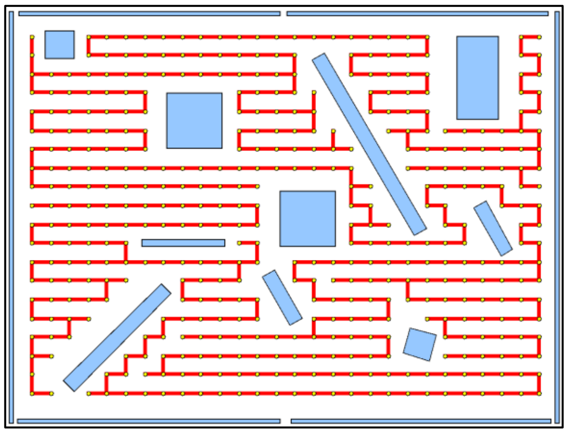
- If you get a **ConcurrentModificationException**, then you have likely not followed the instructions and are trying to remove from the list of nodes that you are looping through.

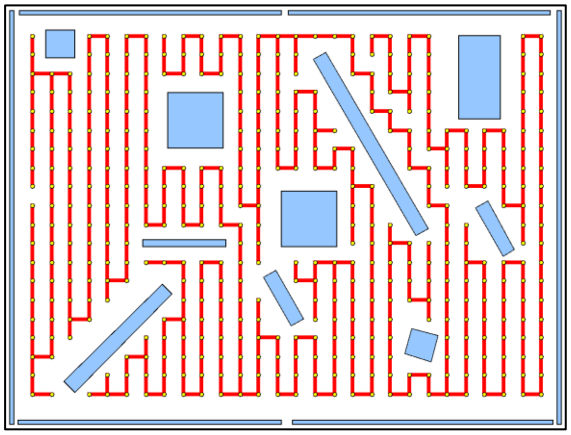
(5) Now you are ready to create the spanning tree. A **computeSpanningTree()** method has already been written for you that does the "housekeeping" work for the creation of the spanning tree. The method first ensures that a graph has been created and has some nodes in it. You will notice in the **Node** class, that each node has a **previous** and a **distance** attribute. The **computeSpanningTree()** method initializes these values for all nodes to be **null** and **0**. It then looks at the starting location in the robot's environment and determines the closest node in the graph. This will be the `startNode` (or root node) of the spanning tree. The method calls a recursive helper function `computeSpanningTreeFrom(startNode, null)`.

This is the function that you must complete. It takes two parameters. The first is the node, **n**, that we are currently at in our graph traversal. It will start as the root node of the spanning tree. The second parameter is the edge, **e**, of the graph that we came in along to get to node **n**. Therefore, **n** is always one of the nodes of edge **e**. However, the very first time the function is called, **e** is set to **null**, since it is the first node in the graph traversal and we did not come in on any edge to get there.

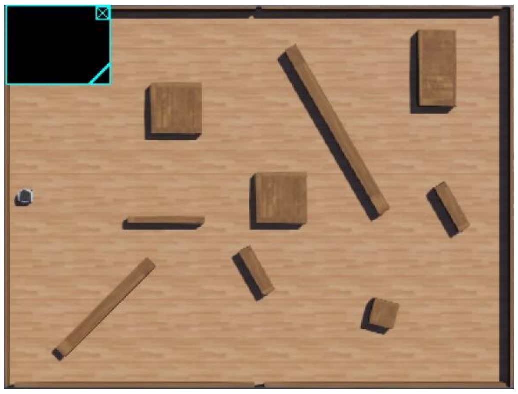
Complete this recursive function by following slide 9 in the notes. Keep in mind that we already have a graph full of edges. To get the spanning tree, we will be removing edges. However, you should NOT remove any edges in your code. Instead, you just need to "mark" the edges as being part of the spanning tree. You "mark" an edge **e** by calling **e.setSelected(true)**. The code has already been set up so that all edges have a selection value of **false** before your method is called. Once your code marks all the edges, all the non-selected edges will be removed from the **gridGraph** (by the **computeSpanningTree()** method)) … thereby leaving the spanning tree as the **gridGraph**.

Run your code. Select the **Show Spanning Tree** option in the menu. You should see something similar to one of the images below, depending on the order that you created your graph edges. Although, you may end up with a different spanning tree altogether. Take a snapshot and save it as **Snapshot4.png**.





(6) 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 **Lab20** subfolder of the **Parts6-8** portion of the given code. You should see the world appear like this.



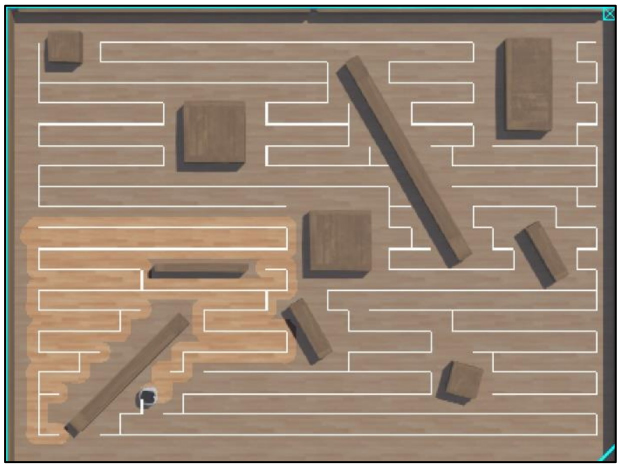
You will get a runtime error upon loading as you do not have a compiled **AreaCoverage** class in the **Lab20Controller** folder. Copy your **AreaCoverage.java** file into the **Lab20Controller** subfolder under the **Lab20/controllers** folders and then open that file in the webots editor and compile it.

You may have noticed that the **AreaCoverage** class has an attribute of type **ArrayList<Point>** called **path**. This will be the path that the robot will travel along. You will notice the following at the end of the **computeSpanningTree()** method:

```
path.clear();  // Clear the previously-computed path, if there was one
edgeCount = gridGraph.getEdges().size();
System.out.println("Spanning Tree has " + edgeCount + " edges");
for (Node n: gridGraph.getNodes())  // Reset all the previous pointers
  n.setPrevious(null);
// We pass in a "dummy edge" that is not part of the tree
// just so that we don't have to check for null
computeCoveragePathFrom(startNode, dummyEdge);
```

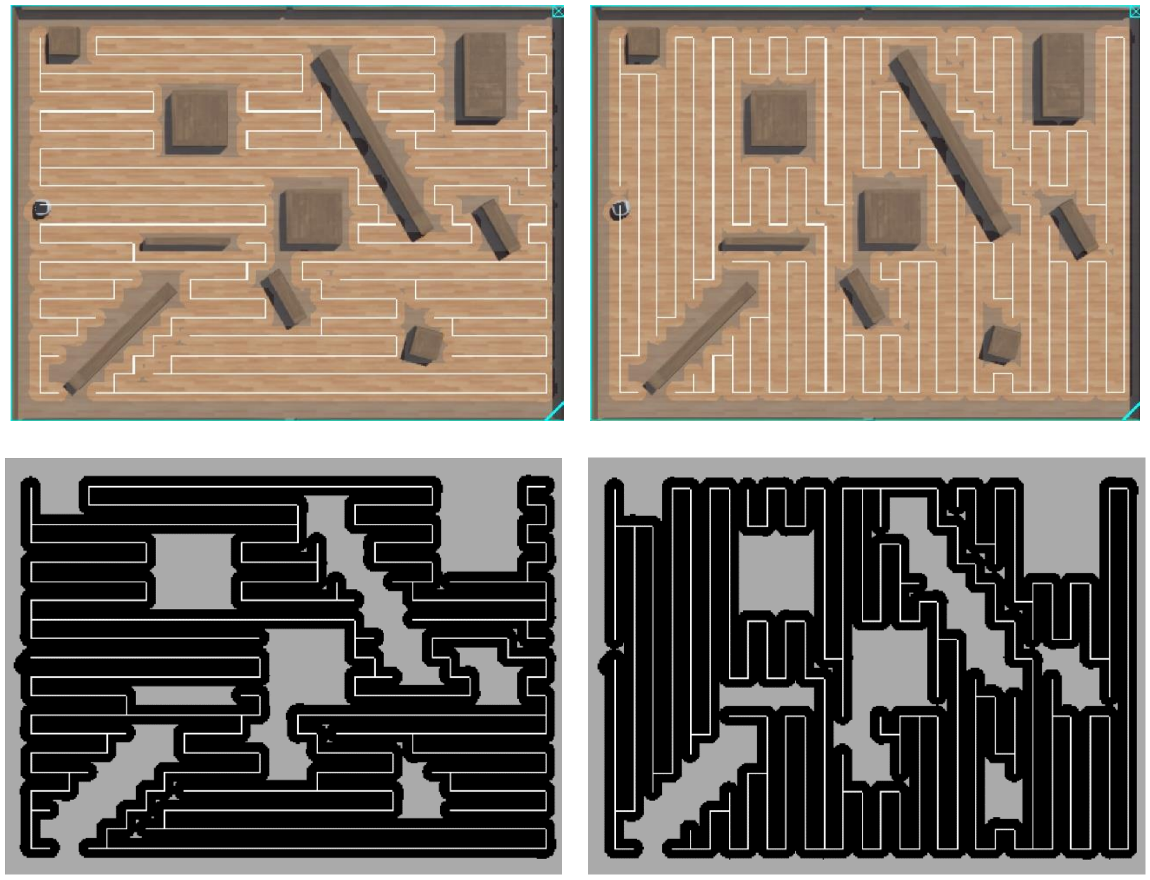
This code will prepare for the computing of the path. You will need to insert code into the `computeCoveragePathFrom()` helper function that will do the recursive traversal of the tree to build up the path starting at the **startNode**. The coverage path will be computed as a set of vertices that the robot will visit in sequence. So, all you need to do is visit all the nodes of the tree, adding their locations to the **path** when you encounter them the first time in the tree traversal, and then adding that node's **previous** node's location on the way back from the recursion. The **previous** attribute for each node has been set to **null** before the method is called. In your helper function, for the current node, **n**, you will need to set the **previous** value to point to the node that was before **n** in the path (see slide 12 for pseudocode).

Once you believe that your code is correct, run it. Stretch the display window to cover the environment "perfectly". You should see the robot start moving in the Webots environment. You will also begin to see a cleaned area where the robot has travelled and it should follow along the spanning tree as shown here (keep in mind though, that your spanning tree may be different):



Unfortunately … it is painfully slow to watch … so make sure that you use the fast-forward button in the simulator. Let the algorithm run to completion. If you computed all the points correctly, you won't see the robot bump into any obstacles. However, in some cases, the simulator has problems with high robot speeds. You may need to reduce the robot's speed to **5** instead of **10** to prevent collisions. The robot will stop when it gets back to the starting location. You will notice that the robot traces back along the path that it had already traversed.

The result should look something like what is shown below. The top two images show the cleaned environment for both horizontally-oriented and vertically-oriented spanning trees … although it may differ if your spanning tree was different. Double-click the display window so that it appears in a separate window. You will get something like one of the two black windows below. Save a snapshot of your black and gray window as **Snapshot5.png**.



Don’t worry too much about the uncovered areas along the obstacle boundaries … we will cover this in the next lab 😉.

(7) You may have noticed that the robot travelled all the way through the graph but then had to travel almost all the way back just to cover a few nodes that it missed. We will now adjust the code so that the robot stops once it has visited all the nodes (see slide 14 for explanation).

An attribute called **edgeCount** has been added to the **AreaCoverage** class which is set it to **gridGraph.getEdges().size()** right after the **path.clear()** in the **computeSpanningTree()** method. Adjust the `computeCoveragePathFrom()` method so that it decreases the **edgeCount** by 1 when it travels on a spanning tree edge for the first time and so that it only adds points to the **path** (i.e., the last line of the method) if **edgeCount** > 0. This will allow the robot to stop as soon as it has reached all the nodes in the spanning tree. Therefore, it will not return to its original location afterwards. You will need to be careful not to count an edge traversal the first time through the method since we came in on a “dummy edge”.

Once you believe that your code is correct, run it. If all is working correctly, you should now notice that the robot stops once it reaches all the nodes without having to go back to the start again, even though it does a little bit of trace-back to get to the last branch that it missed.

Debugging Tips:

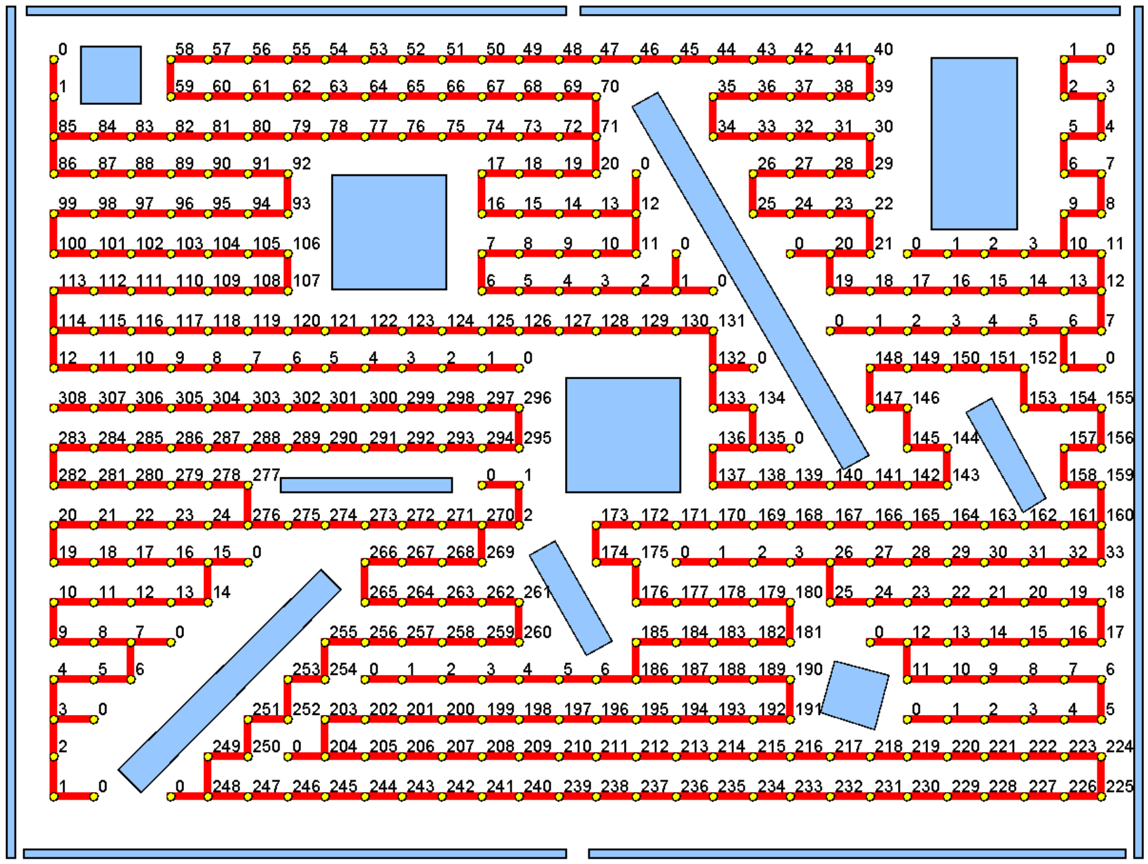
- If your robot seems to head into some weird direction after you think it went to the last node, then visually examine your spanning tree coverage carefully. It is likely that one of your nodes was not visited, and that is why the robot is trying to go directly there when you think it is done. You likely counted the first time through the method as an edge traversal by mistake.

(8) Finally, you will now adjust the computing of the path points so that the robot goes down the smaller branches first and the largest ones last (see slides 15 to 18).

Notice the following in the **computeSpanningTree()** method just <u>before</u> the code for clearing and computing the path:

```
for (Node n: gridGraph.getNodes()) {
  n.setVisited(false);
  n.setDistance(0);
}
computeNodeHeightsFrom(startNode);
```

The **computeNodeHeightsFrom()** method has already been completed for you so that it computes the height of each vertex accordingly. The heights are stored in the **distance** attribute for each node. Nodes which are leaves of the tree will have height **0**, whereas other nodes will have the height of the maximum of their children. The following image gives the heights of all the nodes in the spanning tree that I had computed (your tree may be different):



Modify the `computeCoveragePathFrom()` method so that it traverses the smaller branches first (see slide 18 … although we will still keep our code in there which stops after a certain edge count). There is a static method that you can use for sorting called `Collections.sort(aList)` where **aList** is the list of things that you want to sort. The **Node** class has already been set up with a **compareTo()** method that allows sorting by increasing order of the **distance** attribute.

Once you believe that your code is correct, run it. You should now notice that the robot visits all the smaller branches from a node before visiting the larger branches.

Submit your **AreaCoverage.java** file … as well as the **5** snapshot files. Make sure that your name and student number is in the first comment line of your code.