COMP 3203 Assignment 2

Project Report

Anthony D'Angelo # 100773125, Nicolas Porter # 100760059

11/30/2011

This document provides instructions on how to use the project application followed by class diagrams, implementation details, rudimentary test cases and, lastly, example log file output.

Contents

Table of Figures

Table of Tables

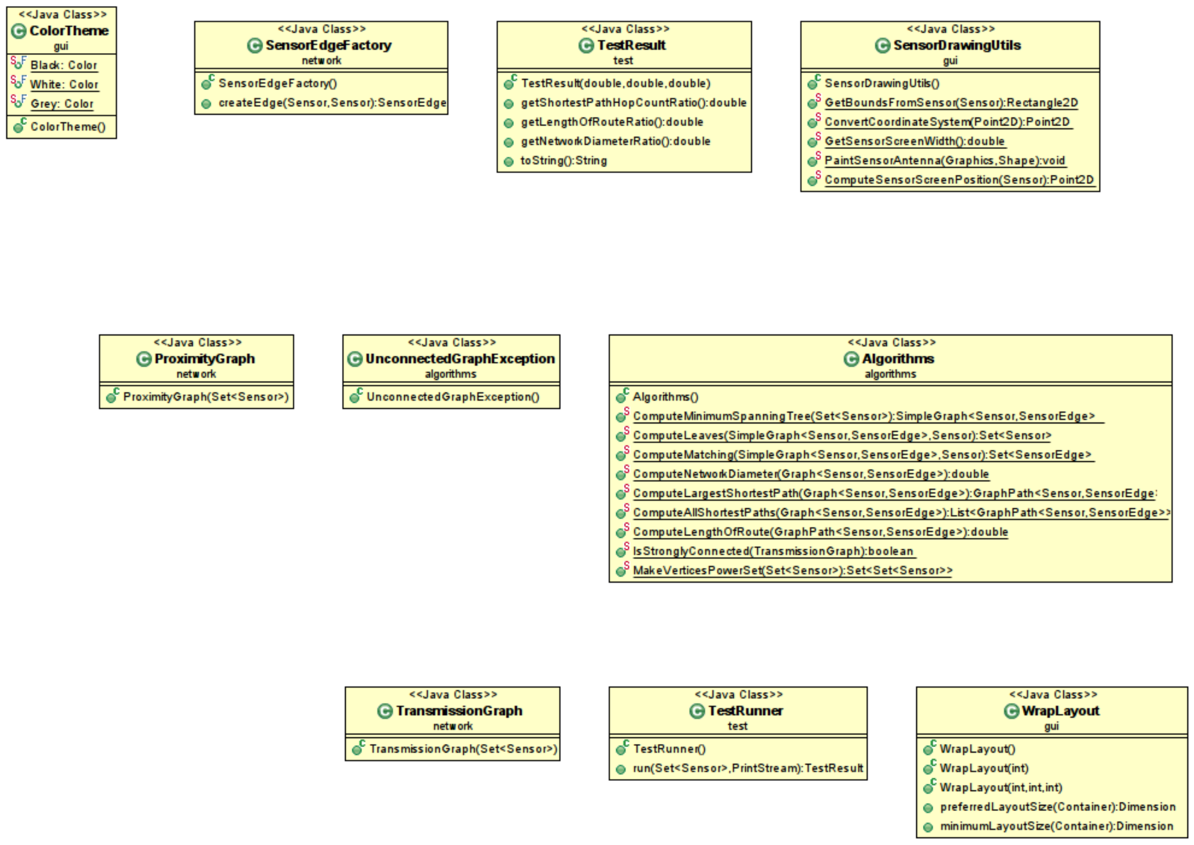
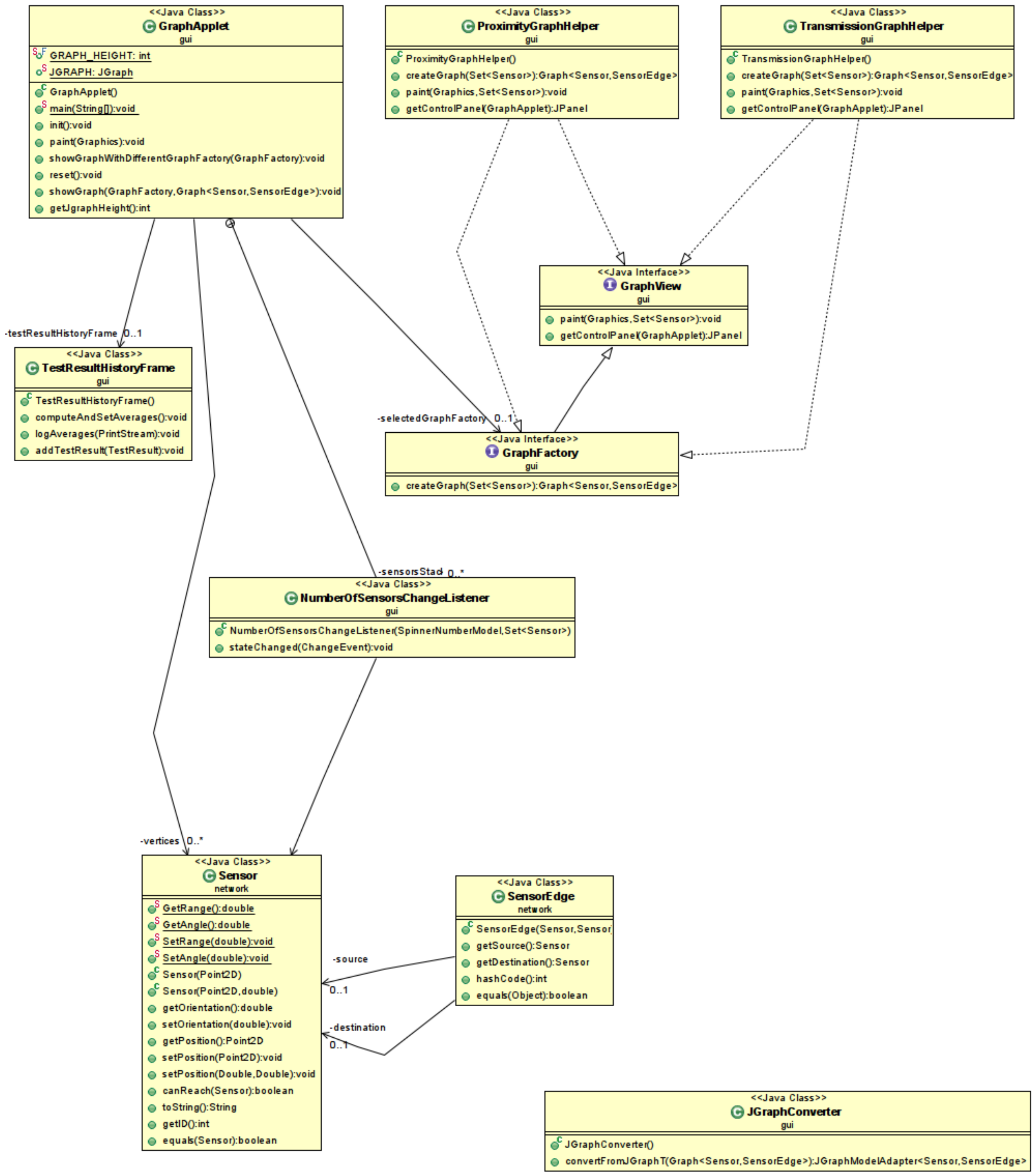

<<Java Class>>
ColorTheme
gui
Black: Color
White: Color
Grey: Color
ColorTheme()
<<Java Class>>
SensorEdgeFactory
network
SensorEdgeFactory()
createEdge(Sensor,Sensor):SensorEdge
<<Java Class>>
TestResult
test
TestResult(double,double,double)
getShortestPathHopCountRatio():double
getLengthOfRouteRatio():double
getNetworkDiameterRatio():double
toString():String
<<Java Class>>
SensorDrawingUtils
gui
SensorDrawingUtils()
GetBoundsFromSensor(Sensor):Rectangle2D
ConvertCoordinateSystem(Point2D):Point2D
GetSensorScreenWidth():double
PaintSensorAntenna(Graphics,Shape):void
ComputeSensorScreenPosition(Sensor):Point2D
<<Java Class>>
ProximityGraph
network
ProximityGraph(Set<Sensor>)
<<Java Class>>
UnconnectedGraphException
algorithms
UnconnectedGraphException()
<<Java Class>>
Algorithms
algorithms
Algorithms()
ComputeMinimumSpanningTree(Set<Sensor>):SimpleGraph<Sensor,SensorEdge>
ComputeLeaves(SimpleGraph<Sensor,SensorEdge>,Sensor):Set<Sensor>
ComputeMatching(SimpleGraph<Sensor,SensorEdge>,Sensor):Set<SensorEdge>
ComputeNetworkDiameter(Graph<Sensor,SensorEdge>):double
ComputeLargestShortestPath(Graph<Sensor,SensorEdge>):GraphPath<Sensor,SensorEdge
ComputeAllShortestPaths(Graph<Sensor,SensorEdge>):List<GraphPath<Sensor,SensorEdge>>
ComputeLengthOfRoute(GraphPath<Sensor,SensorEdge>):double
IsStronglyConnected(TransmissionGraph):boolean
MakeVerticesPowerSet(Set<Sensor>):Set<Set<Sensor>>
<<Java Class>>
TransmissionGraph
network
TransmissionGraph(Set<Sensor>)
<<Java Class>>
TestRunner
test
TestRunner()
run(Set<Sensor>,PrintStream):TestResult
<<Java Class>>
WrapLayout
gui
WrapLayout()
WrapLayout(int)
WrapLayout(int,int,int)
preferredLayoutSize(Container):Dimension
minimumLayoutSize(Container):Dimension

Figure 1: Class diagrams



<<Java Class>>
GraphApplet
gui
GRAPH_HEIGHT: int
JGRAPH: JGraph
GraphApplet()
main(String[]):void
init():void
paint(Graphics):void
showGraphWithDifferentGraphFactory(GraphFactory):void
reset():void
showGraph(GraphFactory,Graph<Sensor,SensorEdge>):void
getJgraphHeight():int
<<Java Class>>
ProximityGraphHelper
gui
ProximityGraphHelper()
createGraph(Set<Sensor>):Graph<Sensor,SensorEdge>
paint(Graphics,Set<Sensor>):void
getControlPanel(GraphApplet):JPanel
<<Java Class>>
TransmissionGraphHelper
gui
TransmissionGraphHelper()
createGraph(Set<Sensor>):Graph<Sensor,SensorEdge>
paint(Graphics,Set<Sensor>):void
getControlPanel(GraphApplet):JPanel
<<Java Interface>>
GraphView
gui
paint(Graphics,Set<Sensor>):void
getControlPanel(GraphApplet):JPanel
-testResultHistoryFrame 0..1
<<Java Class>>
TestResultHistoryFrame
gui
TestResultHistoryFrame()
computeAndSetAverages():void
logAverages(PrintStream):void
addTestResult(TestResult):void
-selectedGraphFactory 0..1
<<Java Interface>>
GraphFactory
gui
createGraph(Set<Sensor>):Graph<Sensor,SensorEdge>
-sensorsStad 0..*
<<Java Class>>
NumberOfSensorsChangeListener
gui
NumberOfSensorsChangeListener(SpinnerNumberModel,Set<Sensor>)
stateChanged(ChangeEvent):void
-vertices 0..*
<<Java Class>>
Sensor
network
GetRange():double
GetAngle():double
SetRange(double):void
SetAngle(double):void
Sensor(Point2D)
Sensor(Point2D,double)
getOrientation():double
setOrientation(double):void
getPosition():Point2D
setPosition(Point2D):void
setPosition(Double,Double):void
canReach(Sensor):boolean
toString():String
getID():int
equals(Sensor):boolean
-source 0..1
-destination 0..1
<<Java Class>>
SensorEdge
network
SensorEdge(Sensor,Sensor)
getSource():Sensor
getDestination():Sensor
hashCode():int
equals(Object):boolean
<<Java Class>>
JGraphConverter
gui
JGraphConverter()
convertFromJGraphT(Graph<Sensor,SensorEdge>):JGraphModelAdapter<Sensor,SensorEdge>

Figure 2: Class diagrams continued

Use Instructions

Once running, resize the windows to a comfortable size to work with. Initially there are no sensors, the range is 100 and we are working with omnidirectional graphs. The labels are mostly self-explanatory. Selecting one of the 'show' radio buttons changes the kind of graph being displayed. The 'Sensor Range' field shows the value of the range of transmission for the sensors and can be changed by writing new numeric values or using the spin-box (the minimum range is 10,000 and the maximum range is 0). The 'Sensor #' field shows the number of sensors that exist in the graph and can be changed in the same way the 'Sensor Range' values can be(the minimum is 0, the maximum is 1,000 and please note that when new sensors are added they are randomly placed on the screen). The 'Draw sensor antenna' checkbox indicates whether or not the outline of the transmission area of the sensors should be displayed (this displaying has a bug we could not track down in which sometimes this outline will not be drawn for some of the sensors). The "Test Result History Frame" window is discussed more in the "Performance Comparisons" section, but worth noting here is that there is a field in which you can enter the path and filename of the log file to be logged to (the default writes to the file bin/comp3203_as02_<current Timestamp>.txt in the current directory). The logging is done in append mode which means old data is not deleted when new tests are run, but when every time the program starts it will write to a new file by default. The logging is only done when you run tests (which can only be done on strongly connected graphs and if it is not strongly connected nothing is logged and no tests are run) and it outputs the sensor data, the *intermediate* results of the various performance comparison calculations and the final results of the calculations of this test instance and then the newly computed statistical/compounded averages. If we are dealing with a directed antennae graph, we are presented with the 'Sensor Angle' spin-box field (which represents the value in degrees of the sector/transmission angle used by the sensors with a minimum of 0 and a maximum of 360 and can be changed in the same way as the other spin-boxes) and we are also presented with a label telling us whether or not our directed graph is strongly connected.

To use the program should now be straightforward but one thing to note is that the user can and will probably need to drag and relocate the sensors in the graph.

Implementation Details

JGraphT

Our program implements graphs using graph data structures and graph algorithms from the JGraphT library. The *ProximityGraph* class extends from *SimpleGraph*, a JGraphT class which models an undirected graph with no self loops or multiple edges between a pair of vertices. We use this *ProximityGraph* class to represent our omnidirectional graphs. Instances of the *Sensor* class have properties like orientation, position and unique numeric identifiers. This *ProximityGraph's* constructor requires a set of sensors which is then used as its vertex set which is itself then used to compute the graph's edges. We store the vertices and edges of the graphs in instances of the standard Java *Set* template classes because they are easy to work with and they do not allow duplicate items. The *ProximityGraph* computes its edge set by creating/adding an edge between every node pair whose Euclidian distance is smaller than the range of the sensors. Note that the range and sector angle of sensors is a class static variable in the *Sensor* class which means that all sensors will have the same range and sector angle (although only directed graphs care about the sector angle). The term "ProximityGraph" was used because the paper describing the orientation algorithm for directed graphs (mentioned below in the "Directed Sensor Antennae Orientation Algorithm" section) used this term which, to us, seemed to describe an omnidirectional graph because the edges are between a given vertex and the vertices in the proximity of the given vertex.

The *TransmissionGraph* class extends the *SimpleDirectedGraph* class (another JGraphT class) which is the same as a simple graph but its edges are directed. Like the *ProximityGraph*, this class' constructor takes in a set of sensors to use as its vertex set which is then used to compute the graph's edges. This constructor differs from *ProximityGraph*'s though because to compute the edge set it uses the sensor antennae orientations which it also computes (by using the orient antenna algorithm which is described in the "Directed Sensor Antennae Orientation Algorithm" section). The term "TransmissionGraph" was used because the paper describing the orientation algorithm for directed graphs (mentioned below in the "Directed Sensor Antennae Orientation Algorithm" section) used this term which, to us, seemed to describe a directional graph because the creation of edges must now restrict and define to whom a given vertex can transmit.

Implementing our graph data structures by extending JGraphT classes allows us to make use of the JGraphT's of the JGraphT's algorithm classes in

Table 1.

Table 1: Useful JGraphT algorithms used on the graph classes in our program

Class/Algorithm	**Used to:**
StrongConnectivityInspector	test a graph for strong connectivity
FloydWarshallShortestPaths	compute all of the shortest paths from some node to every other node in a graph
BreadthFirstIterator	graph traversal
KruskalMinimumSpanningTree	compute the minimum spanning tree of a graph

JGraph

To display our graphs we use the JGraph library. This library is used to draw nodes as well as undirected and directed edges on a Swing JComponent. It also supports the dragging of nodes (user relocation of nodes). JGraphT provides a *JGraphAdapter* class, which allows you to use a JGraphT graph (data structure) in JGraph. Since our graph classes inherit from JGraphT classes, we can wrap them in the adapter and display them with minimal effort.

At the Graphical User Interface (GUI) level, our graphs are built using the *GraphFactory* interface whose purpose is to give the GUI a way to build any graph from a set of sensors. We have two implementations of this interface, the *ProximityGraphFactory* and *TransmissionGraphFactory* classes.

To draw the antenna broadcast area of a sensor, we hook into JGraph's *paint*() method and draw the area (a disk or a disk sector) in the method's off-screen buffer.

We create a new graph when the set of sensors is modified through the GUI. This ensures that all edges are recomputed accurately.

Directed Sensor Antennae Orientation Algorithm

The algorithm used for orienting the directed sensors was extracted from section 4 (*Approximating the minimum radius*) from the paper *Communication in Wireless Networks with Directional Antennae* by Ioannis Caragiannis, Christos Kaklamanis, Evangelos Kranakis, Danny Krizanc and Andreas Wiese and then converted into code that simply orients the sensors. The coded algorithm uses the same predefined radius and sector angle for all nodes and does not try to determine an optimum radius or an optimum angle.

The algorithm takes in a set of vertices V and computes the minimum spanning forest on it, but it only uses one tree of the forest to make its computations. This would mean that only one connected subgraph is computed, but by iterating through the connected sets of V we are able to effectively orient all subgraphs regardless of whether or not they are in the first tree we saw. At a high level, the operations we perform are as follows: we compute the minimum spanning tree (MST); we use the MST to determine a matching; any vertex not in the matching is oriented towards its parent; every vertex in every couple in the matching orients itself with its coupled neighbor to cover the maximum area they can and then they make connections to any vertices they can reach (in a strongly connected graph, this last step is when the communication to the leaves is enabled). Below we talk a little more in depth about how these are accomplished.

Computing the MST was quite straightforward thanks to the JGraphT library. There is a class *KruskalMinimumSpanningTree* (KMST) to whose constructor you pass the graph on which to make the MST. We pass in a new *ProximityGraph* created using V and are returned an object of the type KMST. Then we populate a new *SimpleGraph* (which we treat as our MST) with all of the vertices in V and all of the edges in the edge set returned by the KMST object.

The pain of creating a matching was also eased, once again, by the JGraphT library. There is a class *BreadthFirstSearchIterator* (BFSI) to whose constructor is passed the graph on which we want to traverse and the start vertex which is treated as the root of the tree. To this class we pass our MST as the graph and as the start vertex, we get a pseudo-random one by calling *next()* on our V iterator. As per the algorithm, the next step we take is adding a random edge connected to the root and we simulate that by calling *next()* on the BFSI and then asking the MST for the edge connecting them. That edge is the first one added to our matching. We keep track of the vertices we've used to compute the matching so we add the first edge's vertices to this *verticesAdded* set. Why do we keep track of the ones we've used? That ties in with the next step. The next step is looping through all of the vertices to be visited by the BFSI where, for each iteration, we check if the vertex being examined is a leaf (by examining the number of edges connected to it) or not and act accordingly. If it is not a leaf, if none of its edges (and consequently it) have been added to the matching, we grab the set of edges connected to this vertex and iterate through it until we find an edge where the neighbor vertex has not been added to the matching where we then proceed to add the edge to the matching and record these vertices as having been used. Note that this allows a leaf to be part of the matching if the vertex used an edge connected to a child who is a leaf. If the vertex being examined is a leaf and it has not been used in the matching, we add it to our set of leaves.

One of the products of creating the matching and running the BFS is that we're left with a set of leaves that have not been used in the matching. The next step in the algorithm is to orient these leaves towards their parent in the MST. This is where we actually start adding edges to our *TransmissionGraph*. This is pretty simple to do, although the math for figuring out the orientation angle can get a bit messy.

In any case, if this leaf has an edge (it wouldn't if the tree isn't connected), we grab the vertex associated with the other edge, create a directed edge *from* the leaf *to* the parent and add it to the *TransmissionGraph* edge set. Then we have to orient the leaves so that the middle of their transmitted sector is aimed at the parent vertex.

The first step to finding the orientation angle is to find the height and stride (difference between the x-coordinates) difference *from* the vertex being oriented *to* the vertex being 'pointed to'. The arctangent of this ratio will only give you a value between -90 and 90 degrees but depending on which quadrant we found the angle for (quadrant 1 has both delta height and delta stride positive, quadrant 2 has delta height positive and delta stride negative, quadrant 3 has both delta values negative and quadrant 4 has negative delta height and positive delta stride), the final orientation angle this represents is calculated differently. If we're in quadrant 1, we can just use the calculated angle as the orientation angle; if we're in quadrant 2, the orientation angle will be the calculated angle plus 180 degrees (since the angle is negative this gives us a result between 90 and 180 degrees); if we're in quadrant 3, the orientation angle is the calculated angle plus 180 degrees; if we're in quadrant 4, the orientation angle will be the calculated angle plus 360 degrees (since the angle is negative this gives us a result between 270 and 360 degrees).

The final step in the algorithm is to finish strongly connecting the MST. This is accomplished by: looping through every edge in the matching, orienting the two vertices of that edge so that they include their coupled neighbor at the extremity of their sector (done by setting the orientation to be the computed angle between the vertices as described above and then adding half of our sector angle to this result) and adding the two edges connecting this coupling to the TransmissionGraph edge set; then, looping through all of our vertices, if the vertex being examined is not one of the ones for the matching edge we're looking at, if the non-coupled vertex lies in the sector area of either of the coupled vertex, add an edge to our TransmissionGraph going from the respective coupled vertex to the non-coupled vertex (this is where a leaf that wasn't in our matching would get an edge connected to it). By the end of the iteration of the matching edges, if the graph was able to be strongly connected, it is oriented such that it is.

Performance Comparisons

The shortest path comparison value represents an averaged ratio over the two graphs for the given trial. The way it's computed is as follows: compute the set of all of the pairs in the power set of the set of vertices and call it P; for all sets S in P (they all have two sensors) get the hop count of the shortest path between the sensors in S for the Transmission Graph and the Proximity Graph; compute the ratio "Transmission Graph hop count" divided by "Proximity Graph hop count"; add this ratio to the result (which started at zero); once we've finished iterating through P, divide the result by the size of P.

The length of routes comparison value also represents an averaged ratio over the two graphs for the given trial. Its result is also initialized to zero and ends up being computed in parallel to the shortest path comparison value (i.e. during the same looping) as follows: for all sets S in P, compute the sum of the Euclidian distance of the shortest path between the sensors in S for the Transmission Graph and the Proximity Graph; compute the ratio "Transmission Graph route length" divided by "Proximity Graph route length"; add this ratio to the result; once we've finished iterating through P, divide the result by the size of P.

Computing the network diameter ratio is quite simple thanks to the JGraphT library. To compute the network diameter ratio we create two instances of the class *FloydWarshallShortestPath* (FWSP), one with the Transmission Graph passed into the constructor and one with the Proximity Graph passed into the constructor. We then iterate over the set of vertices and for each vertex, we call *getShortestPaths()* on both graphs' instances of FWSP(which returns a list of all of the shortest paths from that vertex so we need to go through all of the vertices and combine the results). We then iterate over our lists of shortest paths and keep track of the sizes of the largest edge sets of each type of graph (the number of edges in the edge sets in this case represent the number of hops to travel between the two vertices in question and so the largest edge set count is the diameter of the network). Finally to get our ratio we divide the Transmission Graph diameter by the Proximity Graph diameter.

It should also be stated that computing the hop count and the shortest paths in these graphs were also done for us by the FWSP class.

The "Test Result History Frame" window displays (in a chronologically ordered list) all of the previously calculated ratio averages for all of the tests the user ran. Also in the window is presented an average over all of those previous averages. This average of previous averages is recalculated when a new test is run. It is calculated by taking its previous value, multiplying it by the number of items that were in the list prior to this new test, adding the resulting average ratio from the new test, and then dividing by the new number of tests performed. We thought this method for calculating the new average of averages was worth mentioning because it is potentially orders of magnitude faster than the naïve way of doing (looping through all of our records summing up their results and dividing by the number we added).

Testing

We didn't do full-scale, professional testing on our program with formal test cases. Instead, as we implemented features we tested to make sure the new features worked and the old features were not broken. In that regard we did some kind of black box testing. The main testing to do was making sure the implemented orientation algorithm worked since that is what this assignment was really about. That being the case, we've come up with some basic cases to test against. Note that, thanks to JGraphT's *StrongConnectivityInspector*, we are informed in the Graphical User Interface (GUI) whether or not our directed graph is strongly connected.

Table 2: Test 1

Objectives	Determine whether or not our implementation's results matched the behavior described in the class notes for directed antennae with a sector angle of 180 degrees and compare this to the omnidirectional version
Expected Results	We expected it to match and that is indeed what we saw
Conditions	The formed line has to be very close to being straight, sector angle 180 degrees
Assumptions	Same range and sector angle for all sensors

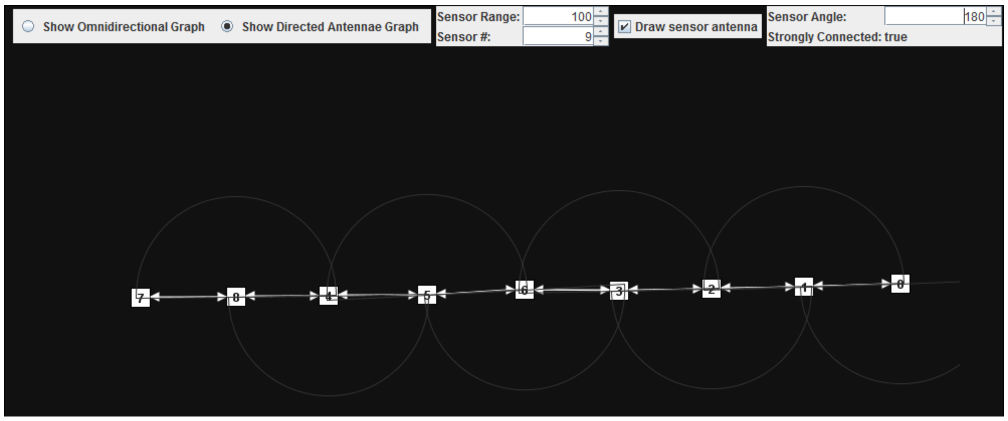


Figure 3: As we can see, when the sector angle is 180 degrees and they are aligned near perfectly straight, the algorithm orients the antennae such that they form a strongly connected graph

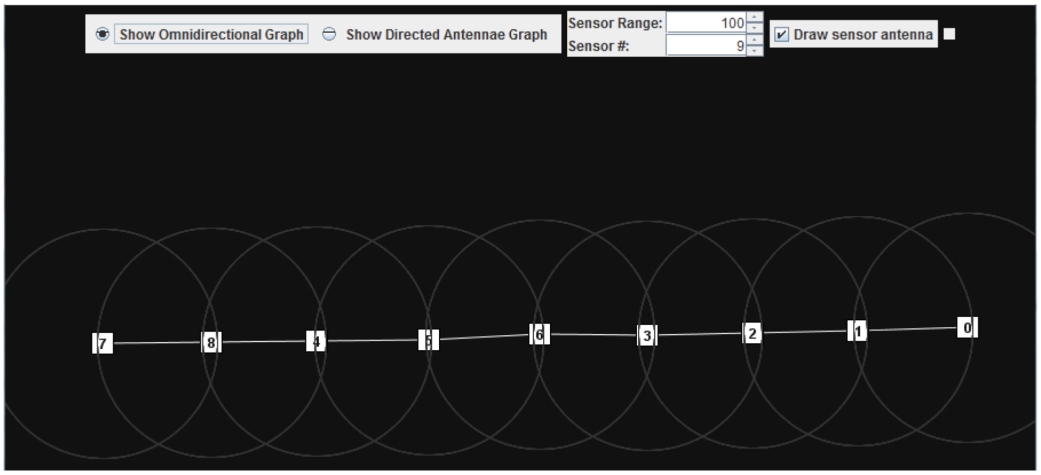


Figure 4: The omnidirectional version of the graph shown in Figure 3

Table 3: Test 2

Objectives	Testing the algorithm on a more complex graph shape, ensuring proper antennae orientation without needing to change the sector angle
Expected Results	If there exists a placing of sensors into a complex shape such that the oriented graph is strongly connected, the algorithm will be able to properly orient the antennae and. When the sensors are spread out, the directed graph looks like the omnidirectional graph. Note that there are many placings that would result in a graph that is not strongly connected.
Conditions	The vertices have to be placed such that they should be strongly connected so the user needs to be able to figure that out and verify the result of the algorithm
Assumptions	Same range and sector angle for all sensors

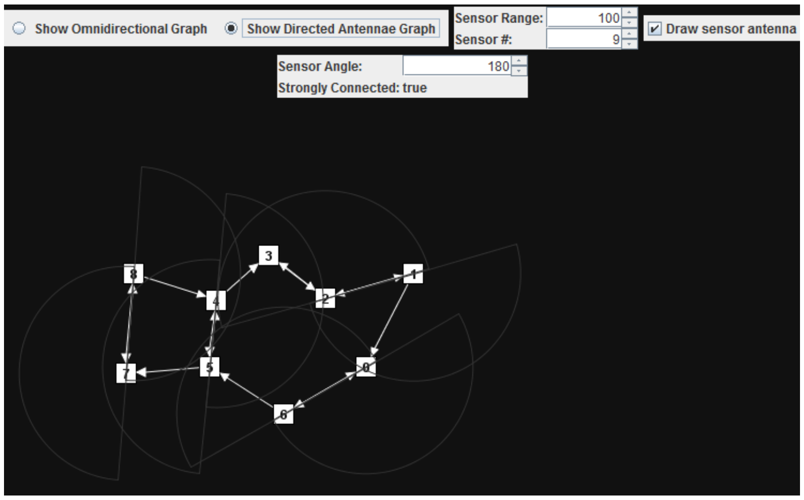


Figure 5: A manually, randomly shaped network using a 180 degree sector angle. Note that it is strongly connected. If we refer to Figure 6 we'll see that this is in fact a strongly connected graph of the omnidirectional counterpart.

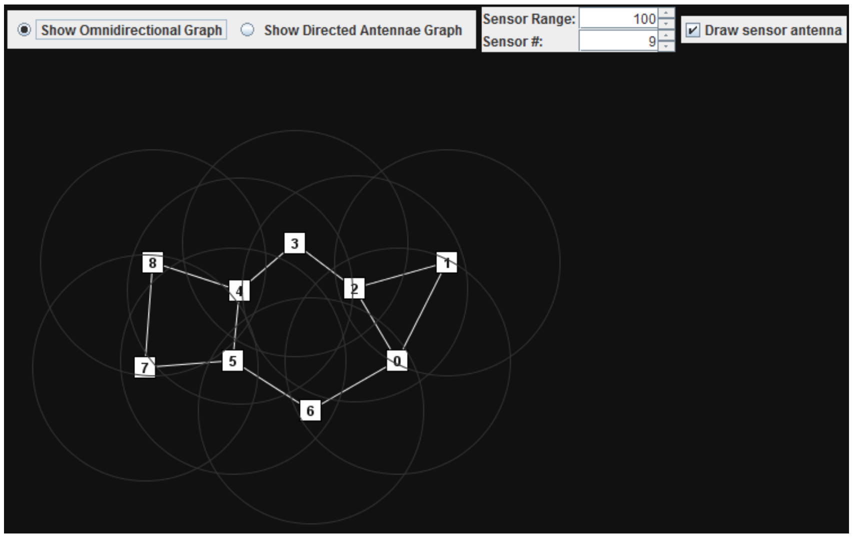
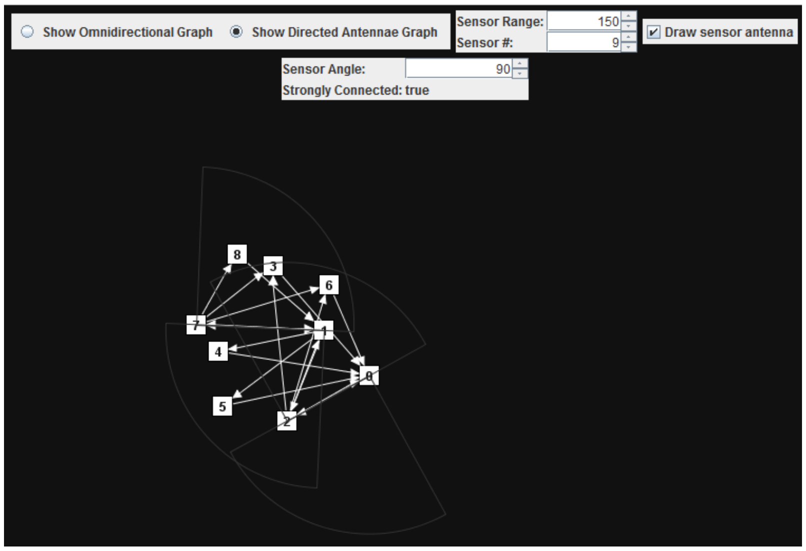


Figure 6: The omnidirectional graph of the network in Figure 5

Table 4: Test 3

Objectives	Test the effects of altering the sector angle and range
Expected Results	Changing just one of these can connect or disconnect your directed graph and changing both may have the effect of preserving the connectivity status. We expect the algorithm to properly orient the sensors.
Conditions	The user needs to be able to figure out the connectivity of the graph and verify the result of the algorithm
Assumptions	Same range and sector angle for all sensors in a trial



Show Omnidirectional Graph [✓ Show Directed Antennae Graph]

Sensor Range: 150

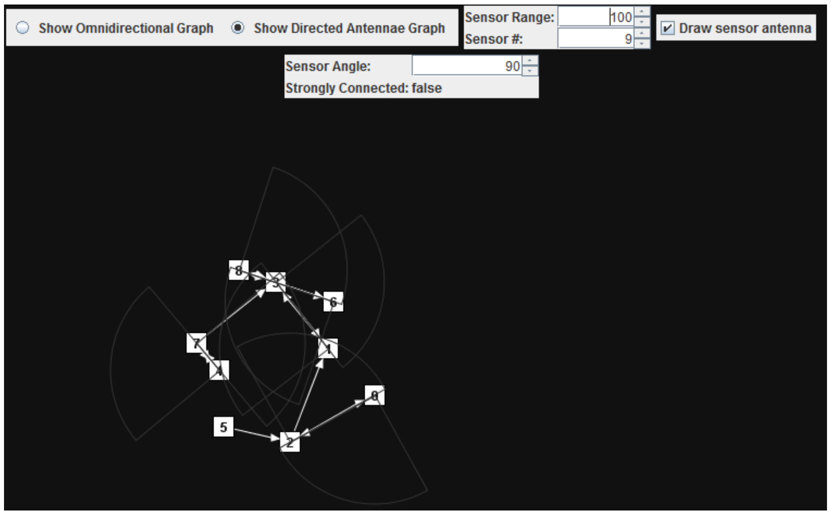
Sensor #: 9

[✓] Draw sensor antenna

Sensor Angle: 90

Strongly Connected: true

Figure 7: When contrasted to Figure 8 shows how a range change for a given sector angle can affect your connectivity. With a difference of 50 units (randomly chosen difference) these two graphs differ in connectivity.



Show Omnidirectional Graph [✓ Show Directed Antennae Graph]

Sensor Range: 100

Sensor #: 9

[✓] Draw sensor antenna

Sensor Angle: 90

Strongly Connected: false

Figure 8: Refer to Figure 7

Table 5: Test 4

Objectives	Test the algorithm output when the sector angle is 360 degrees
Expected Results	The algorithm should produce a graph that very closely resembles the omnidirectional counterpart when the sector angle is 360 degrees
Conditions	The sector angle is 360 degrees
Assumptions	Same range and sector angle for all sensors

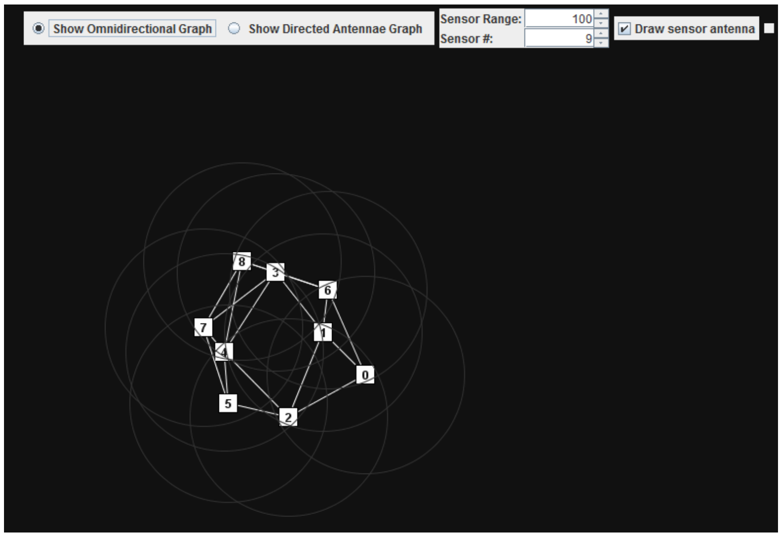


Figure 9: Another manually, randomly shaped omnidirectional graph. In some cases (this being one of them) the graph created by the orientation algorithm when its sector angle is 360 degrees (Figure 10) will have edges in the same spots as the omnidirection version which helps us validate the result of the algorithm.

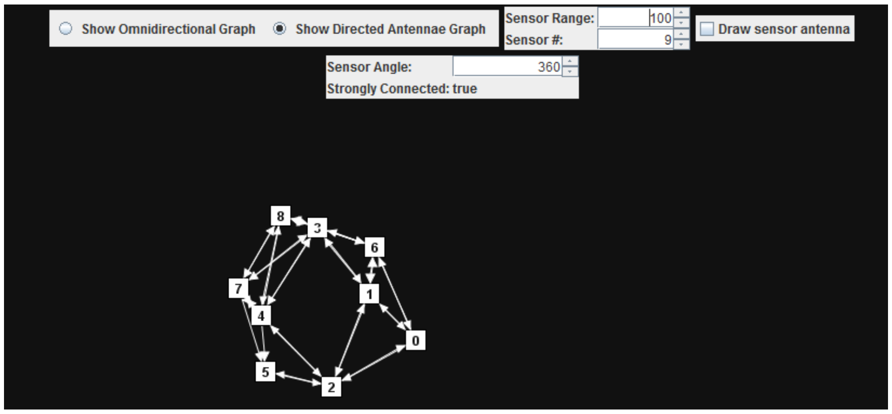
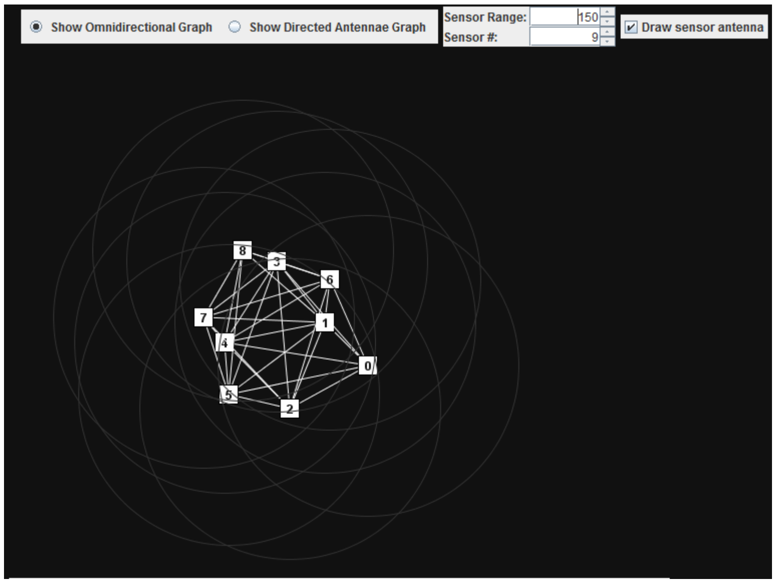


Figure 10: Refer to Figure 9

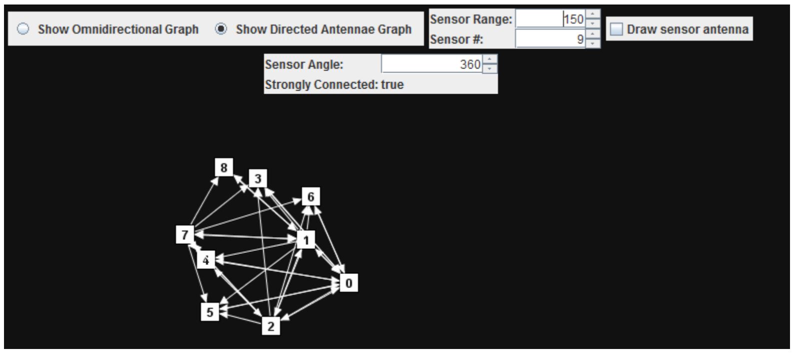
Table 6: Test 5

Objectives	Explore a case where the sector angle is 360 degrees but the algorithm output is noticeably different than the omnidirectional counterpart
Expected Results	This does not mean the algorithm is broken. In fact, it is an indication of it working. See Figure 12's caption for an explanation.
Conditions	Sector angle of 360 degrees
Assumptions	Same range and sector angle for all sensors



Sensor Range: 150
Sensor #: 9

Figure 11: When viewed with Figure 12 goes to show that the edges in the omnidirectional graph don't always get mapped to the directed version as was sort of the case in Figure 9 and Figure 10.



Sensor Range: 150
Sensor #: 9
Sensor Angle: 360
Strongly Connected: true

Figure 12: When we compare this graph to the one in Figure 11, although it is still strongly connected, it doesn't have all of the same edges. This is a direct consequence of the algorithm and happens because the only vertices for which we make a connection to everything within reach are the vertices in the matching whereas the omnidirectional graph connects each vertex to everything it can reach.

Logging

For each test we run, we log the data of the sensors in our vertex set, the individual/intermediate results of the shortest path and route length comparisons and finally the average shortest path ratio, average route length ratio and the average network diameter ratio and then the compounded average determined using all of the previous results. Below is some sample log file output.

```
==== Running tests on sensors [Wed Nov 30 15:31:08 EST 2011]
====================
Sensor Range: 100.0
Sector Angle: 270.0

Sensor 2 [orientation: 102.14427804956699 , position: (326.0, 328.0)]
Sensor 4 [orientation: 157.30620505490765 , position: (231.0, 375.0)]
Sensor 1 [orientation: 270.0 , position: (309.0, 472.0)]
Sensor 3 [orientation: NaN , position: (205.0, 436.0)]
Sensor 5 [orientation: 337.30620505490765 , position: (309.0, 407.0)]
Sensor 0 [orientation: NaN , position: (294.0, 315.0)]
Sensors: Source = 4 Destination = 2

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 1 Destination = 3

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 1

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 2 Destination = 3

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 3

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 3 Destination = 0

Shortest Path Ratio: 1.5
Route Length Ratio: 1.5904626769702264
Sensors: Source = 2 Destination = 1

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 5 Destination = 0

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 3 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 5
```

```
Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 0

Shortest Path Ratio: 2.0
Route Length Ratio: 2.040503003728432
Sensors: Source = 1 Destination = 0

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 2 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 2 Destination = 0

Shortest Path Ratio: 2.0
Route Length Ratio: 5.038334312094532
Sensors: Source = 1 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0

Average Shortest Path Ratio: 1.1666666666666667
Average Route Length Ratio: 1.3779533328528795
Network Diameter Ratio: 1.0

--------------------------------------------
So far, the average of all test results are:
 Shortest Path Ratio: 1.1666666666666667
  Route Length Ratio: 1.3779533328528795
      Diameter Ratio: 1.0
--------------------------------------------

==== Running tests on sensors [Wed Nov 30 15:31:28 EST 2011]
====================
Sensor Range: 100.0
Sector Angle: 200.0

Sensor 2 [orientation: 102.14427804956699 , position: (326.0, 328.0)]
Sensor 4 [orientation: 122.30620505490765 , position: (231.0, 375.0)]
Sensor 1 [orientation: 190.8230112262071 , position: (377.0, 420.0)]
Sensor 3 [orientation: NaN , position: (166.0, 381.0)]
Sensor 5 [orientation: 302.30620505490765 , position: (309.0, 407.0)]
Sensor 0 [orientation: NaN , position: (216.0, 458.0)]
Sensors: Source = 4 Destination = 2

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
```

```
Sensors: Source = 1 Destination = 3

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 1

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 2 Destination = 3

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 3

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 3 Destination = 0

Shortest Path Ratio: 2.0
Route Length Ratio: 1.6296867730192601
Sensors: Source = 2 Destination = 1

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 5 Destination = 0

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 3 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 0

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 1 Destination = 0

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 2 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 2 Destination = 0
```

```
Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 1 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0

Average Shortest Path Ratio: 1.0666666666666667
Average Route Length Ratio: 1.041979118201284
Network Diameter Ratio: 1.0

--------------------------------------------
So far, the average of all test results are:
 Shortest Path Ratio: 1.1166666666666667
  Route Length Ratio: 1.2099662255270818
      Diameter Ratio: 1.0
--------------------------------------------

==== Running tests on sensors [Wed Nov 30 15:31:50 EST 2011]
====================
Sensor Range: 100.0
Sector Angle: 200.0

Sensor 4 [orientation: 418.4336303624505 , position: (231.0, 375.0)]
Sensor 2 [orientation: 238.4336303624505 , position: (284.0, 328.0)]
Sensor 1 [orientation: 290.8230112262071 , position: (377.0, 420.0)]
Sensor 3 [orientation: 135.15417763214742 , position: (199.0, 438.0)]
Sensor 6 [orientation: 315.1541776321475 , position: (270.0, 488.0)]
Sensor 5 [orientation: 110.8230112262071 , position: (309.0, 407.0)]
Sensor 0 [orientation: NaN , position: (246.0, 444.0)]
Sensors: Source = 3 Destination = 6

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 2 Destination = 4

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 2 Destination = 6

Shortest Path Ratio: 1.5
Route Length Ratio: 1.3216963034891414
Sensors: Source = 1 Destination = 3

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 1
```

```
Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 2 Destination = 3

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 6 Destination = 0

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 1 Destination = 6

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 3

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 3 Destination = 0

Shortest Path Ratio: 2.0
Route Length Ratio: 2.890558282548967
Sensors: Source = 2 Destination = 1

Shortest Path Ratio: 1.5
Route Length Ratio: 1.475272047626588
Sensors: Source = 5 Destination = 0

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 3 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.1404706723571967
Sensors: Source = 4 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 0

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 1 Destination = 0

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 6

Shortest Path Ratio: 1.0
```

```
Route Length Ratio: 1.0
Sensors: Source = 6 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 2 Destination = 5

Shortest Path Ratio: 2.0
Route Length Ratio: 1.8723667449337704
Sensors: Source = 2 Destination = 0

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 1 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0

Average Shortest Path Ratio: 1.1428571428571428
Average Route Length Ratio: 1.1762078119502697
Network Diameter Ratio: 1.3333333333333333

--------------------------------------------
So far, the average of all test results are:
 Shortest Path Ratio: 1.1253968253968254
  Route Length Ratio: 1.1987134210014778
      Diameter Ratio: 1.111111111111111
--------------------------------------------

==== Running tests on sensors [Wed Nov 30 15:32:09 EST 2011]
====================
Sensor Range: 100.0
Sector Angle: 360.0

Sensor 4 [orientation: 495.9391909457357 , position: (168.0, 365.0)]
Sensor 2 [orientation: 315.93919094573556 , position: (230.0, 305.0)]
Sensor 1 [orientation: 382.10944834375164 , position: (377.0, 420.0)]
Sensor 3 [orientation: 192.1836565859874 , position: (199.0, 438.0)]
Sensor 6 [orientation: 372.18365658598736 , position: (287.0, 457.0)]
Sensor 5 [orientation: NaN , position: (353.0, 367.0)]
Sensor 0 [orientation: 202.1094483437517 , position: (313.0, 394.0)]
Sensors: Source = 3 Destination = 6

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 2 Destination = 4

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
```

```
Sensors: Source = 2 Destination = 6

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 1 Destination = 3

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 1

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 2 Destination = 3

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 6 Destination = 0

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 1 Destination = 6

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 3

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 3 Destination = 0

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 2 Destination = 1

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 5 Destination = 0

Shortest Path Ratio: 2.0
Route Length Ratio: 2.6369906970670822
Sensors: Source = 3 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 0
```

```
Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 1 Destination = 0

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 4 Destination = 6

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 6 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 2 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 2 Destination = 0

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0
Sensors: Source = 1 Destination = 5

Shortest Path Ratio: 1.0
Route Length Ratio: 1.0

Average Shortest Path Ratio: 1.0476190476190477
Average Route Length Ratio: 1.0779519379555755
Network Diameter Ratio: 1.0

--------------------------------------------
So far, the average of all test results are:
 Shortest Path Ratio: 1.105952380952381
  Route Length Ratio: 1.1685230502400024
      Diameter Ratio: 1.0833333333333333
--------------------------------------------
```