Wireless Sensor Network – Final

Rishab Vaishya

47505527

1.1. Introduction to Summary :

Wireless sensor network is a network to transfer data from a sensor to a server via a highway of sensors (also called backbone) which lead to the server. This highway sensors is a bipartite graph within the sensor network. Every sensor can connect to at least one of this highway sensors. [5]
The goal of this project is to randomly distribute sensors around a unit area (Square or disk) and identify the biggest bipartite graph of sensors (highway sensors) which will be used to transfer data around. This distributed system is used everywhere so it becomes important to implement it efficiently.

So, starting off, we randomly generated x & y points which are within the boundaries to the unit size. we add those points in cell & then generate its neighbors for our project using cell method explained below as it greatly reduces the time needed to generate it.
Now, we calculate degrees of each point & apply 'smallest last ordering' method to sort points into order for coloring, This doesn't require any additional sorting or non-linear way of it, So it further optimizes the project time.
Later, we take the 4 largest color sets & create a combination of 6 bipartite graphs by joining 2 color-set points at a time. Deleting minor components & keep the largest component of the bipartite graphs using 'Breath first search' algorithm. Then we delete the tails of the largest component of each bipartite graph and calculate the total number of edges. Based on the number of edges we select the 2 largest bipartite graphs and plot them. we calculate the domination (coverage) of the graphs by dividing 'total degree of bipartite graph' by 'total degree of whole graphs'.

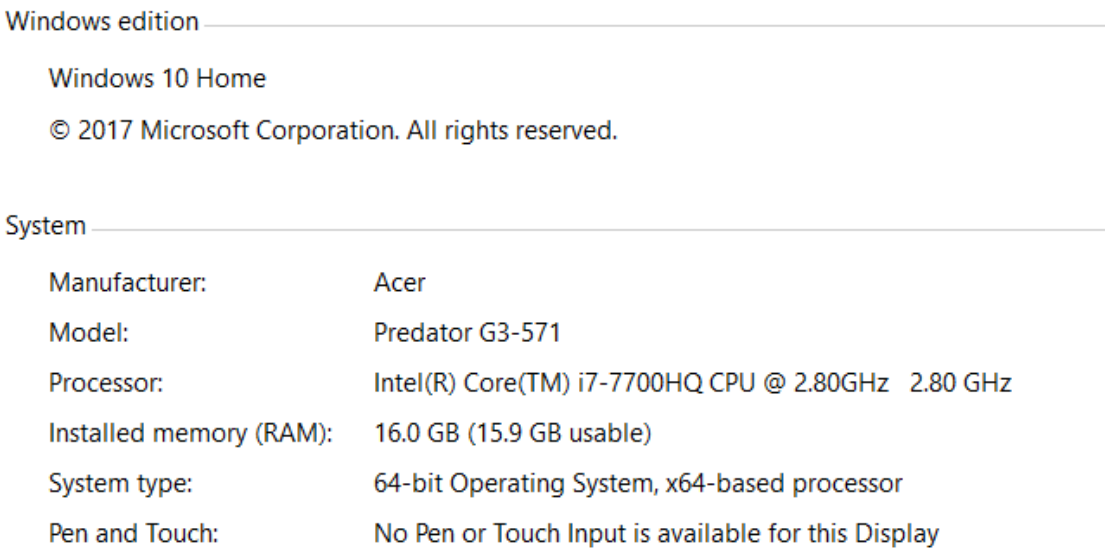
Strong feature : Visualizations is added in project to better understand the flow. The entire project can be completed within 8 seconds for sensor# of more than 100,000 with all data being displayed both graphically using graphs & in writing to quickly and clearly understand the project. All this programmatically calculated with just a single click.

Weak feature : edges are not shown for points more than as that can cause slowdown, space complexity might not be the most optimized as an extra layers of cells are added to reduce time complexity which is explained below. Due to Visualization accurate run-time is not calculated.

1.2. Execution Summary : Summary of the results after the project was executed. (Visualization for plotting & erasing points affects Run-time, other visualization time are not considered in order to accurately measure time-performance of each benchmark)

	Run-Time in sec	0.063	0.233	0.504	2.2700002	3.2959998	5.308	6.785	0.256	2.2729998	3.069
Phase 3	**Order for BG2**	117	567	1806	4556	2567	9072	5137	569	4475	2534
	Order for BG1	121	587	1946	4603	2570	9118	5182	592	4520	2586
	Edges for BG2	146	771	2243	6042	3570	12003	7171	752	5908	3505
	Edges for BG1	150	777	2459	6166	3570	12059	7262	789	5986	3618
	Domination of BG2 (%)	91.7	96.9375	93.668755	98.15313	99.328125	98.24375	99.25938	97.399994	97.84844	99.370316
	Domination of BG1 (%)	95.200005	99.0625	95.49375	98.425	99.56094	98.29765	99.56484	98.7375	97.984375	99.654686
Phase 2	**Terminal Clique**	16	37	22	40	56	38	53	35	33	53
	Max Degree when deleted	20	40	25	43	74	44	74	40	42	75
	color size	18	37	22	40	64	40	67	36	39	69
Phase 1	**Realized average degree**	29	62	32	63	126	63	126	62	64	127
	Max Degree	47	89	52	98	171	103	183	91	101	173
	Minimum Degree	11	17	7	13	36	15	34	22	22	46
	Number of distinct Edges	14844	248769	259060	2045371	4037268	4094143	8109456	249650	2071577	4076418
	Radius (R)	0.100925300	0.050462650	0.025231325	0.017841241	0.025231325	0.012615662	0.017841241	0.089442719	0.031622776	0.044721359
	Topology	Square	Square	Square	Square	Square	Square	Square	Disk	Disk	Disk
	Estimated Average Degrees	32	64	32	64	128	64	128	64	64	128
	Number of sensors	1000	8000	16000	64000	64000	128000	128000	8000	64000	64000
	Benchmark #	1	2	3	4	5	6	7	8	9	10

2.1 Programing Environment :



More information - https://www.acer.com/ac/en/US/content/predator-helios300-series

2.2 Programming Language & Version : Java build : 1.8.0_161-b12

IDE : Eclipse IDE for Java Developers

- Version: Oxygen.2 Release (4.7.2)
- Build id: 20171218-0600
- Reason for choosing java is explained later.

2.3 Classes : Following are the list of Imports I've done for my project

- **import** java.awt.BasicStroke; - for drawing strings in canvas.
- **import** java.awt.Canvas; - Used for rendering points, edges, Strings, etc.
- **import** java.awt.Color; - to color the edges of min & max degree sensor.
- **import** java.awt.Font; - to customize the font.
- **import** java.awt.Graphics; - the draw graphical content in frame using canvas.
- **import** java.awt.Graphics2D; - to draw line (edges) on the frame. A library provided by java
- **import** java.awt.geom.Line2D; - to customize line (edges) on the frame. A library provided by java.
- **import** java.awt.image.AreaAveragingScaleFilter; - to rescale the content drawn on the frame.
- **import** java.util.ArrayList; - to store array of undefined size dynamically.
- **import** java.util.Collections; - to sort the array list.
- **import** java.util.Iterator; - to iterate through a collection.
- **import** java.util.List; - to store data (cell) used in the project.
- **import** java.util.Random; - to generate random numbers.
- **import** java.util.Scanner; - to import values from the user.
- **import** javax.swing.JFrame; - to hold the canvas and display graphical content in a frame.

2.4 Why I chose Java ? :

- **Platform independent –** we can run this code in any programming environment once built.
- **Built-in data structures –** can make use of ready-made data structures to use. These are optimized & can fall under eligibility of garbage collection in java if not referred to release memory.
- **Many built-in & 3rd party libraries –** Java has a lot of variety of libraries which a programmer can make use of in their project. Including the graphic rendering libraries like Java.AWT.
- The graphic class in java provides simple ways to plot points, text, lines & other shapes like oval, circle, rectangles ETC in Java frames (JFrames) which enough for this project.
- The 'draw' method in Graphic class is overridden & does all the plotting visuals on the JFrames.
- **Compilers –** java uses just in time compilers which makes execution of code faster and more memory efficient.
- **Support –** Since java is used in so many applications it has a lot of other supporting programming languages which we can use if needed.
- The reason for using eclipse was easy implement java in eclipse, it provides a lot of easy shortcuts and auto-completion, building & cleaning project is easy and I was most comfortable with it.

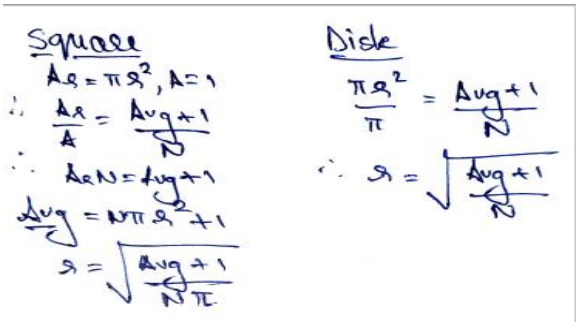
3.0. Reduction To Practice : In this section I'll describe the flow of the program, description of the algorithm used & how I've implemented it. The time complexity is also mentioned in section wherever needed. I've taken 1000 pixel as 1 unit for square & 500 pixel as 1 unit for Disk.

3.1. Data structures : The following are the built-in and custom data structures I used for my projects[4]

- **Array List :** stores array of similar datatype with undefined size. I've used this for stores Sensor data like X, Y Co-ordinates & neighboring sensors of that sensor
- Created array-List of Points, Degrees, Colored Points Bipartite Pairs etc. throughout the project.
- **2D array :** stores 2d array of similar datatype with fixed size. I used it to store data of cells where the sensors are divided into.
- **Cell :** block of the plane in which sensors are distributed. Contains an array list of Point which will be present in that cell.
- **Degree :** contains the degree of a sensor and number of occurrence of that degree
- **Point :** represents sensor, has X,Y & array list of neighboring Points in it.

- **Input Values from User :**
  - This is a fairly simple process.
  - I'm inputting values of Expected average density, no of sensors and topology from the user and storing it in a variable.

- **Calculate Radius depending on the topology :**
  - Now according to the topology, we calculate the radius by using the following formulas
  - If Topology = unit square
    - A = N * π * R * R
  - If Topology = unit disk
    - A = N * R * R



- After calculation of radius it is stored and used to calculate the number of cells which is
  - 1-unit size/radius

- Time complexity for this would be a constant. I.E- **Θ(C)**

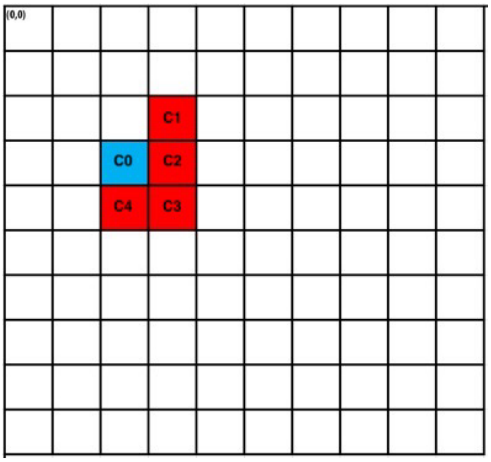
- **Generate Random Points depending on the topology :**
  - Since we have to calculate x & y co-ordinates for all sensors, the time complexity here would be **O(N)**
  - from the unit size we generate random values between 0 and unit size.
  - These numbers are used to get X & Y co-ordinates of sensors on the graph.
  - We generate for all the sensors.
  - For disk topology, if distance between center of the disk and generate co-ordinate is greater than unit length then it is ignored and another value is generated in place of it.
  - Distance formula: `Math.sqrt(((X – X1) * (X – X1)) + (Y-Y1) * (Y-Y1)) <= distance)`

- **Assign cells to points :**
  - After generating x & y co-ordinates for all sensors, I add them to correct 2d array of cell.
  - This assignment is done as the points are being generated.
  - The X & Y index of Cell is calculated based upon the X & Y co-ordinate of Point and the sensor radius.
  - It is the sensor radius which determines the size of cell by using the formula
    `noOfCellRowsCoulmns = (int) (unitSizeForSquare / sensorRadius);`
  - Code :

```
private static void SortIntoCell(Point p)
{
	int X = (int) (p.getX() / (radius));
	int Y = (int) (p.getY() / (radius));
	cell[X][Y].getPoints().add(p);
}
```

- **Generate potential neighbors[5] (Cell Method) :**

  Algorithm Desciption :
  - Potential neighbor are those sensors are in same and neighboring Cells
  - If the distance between the node and potential neighbor is less than the sensor radius then it is their actual neighbor.
  - From the figure below, if current sensor lie in cell c0, then all sensors in cell c1,c2,c3,c4 are its potential neighbor.



Src : Project Part I ppt by Zizhen Chen.

- Here we prevent the program to check distance of a point with every other point on the graph I.E by using brute force method.
- So here the time complexity drops from **Θ(n^2)** to **Θ(N*APC*5)** where APC is average points in cell which is linear.

Algorithm Engineering :

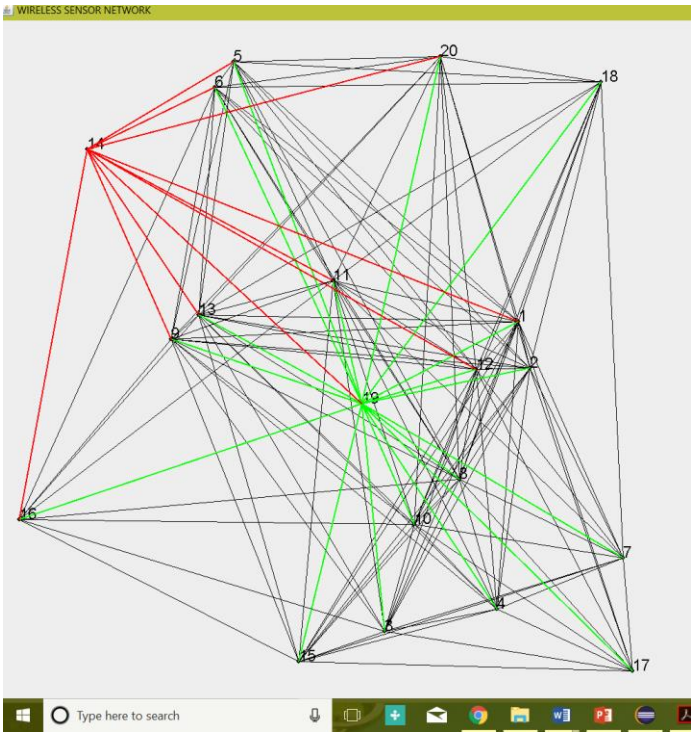
- Now if the distance between all the shortlist points (potential neighbors) from the graph are less than the sensor radius they are added in the neighbor's array List.
- This has been my most improved feature as I managed to reduce compute time for 128K sensors of 128 average degree from **1.36 minutes to 1.9 seconds**.
- This was a 5000% increase in speed performance.
- The X,Y index of cell[X][Y] are calculated from X & Y co-ordinates of a point by using same formula as used for adding points into cell mentioned above.
- At first, I used to put lot of conditional checks on calculated X & Y indexes of Cell array to prevent the cell to get an out of bound exception
  Eg : if C0 cell is in the right most column then C1.C2,C3 will go out of bound.
- This conditional check was taking lot of time. So, to prevent that, I added an extra row of Cell on top, bottom and an extra column on the right & removed all the conditions. These new cells had no points in it. This reduced the time from 1.36 minutes to around 18 seconds.
- Now that we have potential neighbor for all points it's time to filter out actual neighbor from the potential, we do by calculating distance between those neighbors. Here the distance is the sensor radius (R) calculate earlier.
- Time complexity her would be to **Θ(N*APC*5)** where APC is average points in cell which is linear.
- Now, after many trial and error I found another bottle neck.
- I was using the distance formula: `Math.hypot(X1 - x2, Y - y2) < sensorRadius) which I then replaced with Math.sqrt(((X - X1) * (X - X1) + (Y - Y1) * (Y - Y1))) <= sensorRadius`
- This further reduced by time from 18 seconds to 1.9 seconds.

- **Store values in array :**
  - The neighboring sensors for a sensor is stored in an array list in the class called points.
  - This array points along with its neighbor and other data is passed to canvas class which is then used to generate graph.

- **Ordering Points & Generate Color**[1][5] **:**

  Algorithm Description :

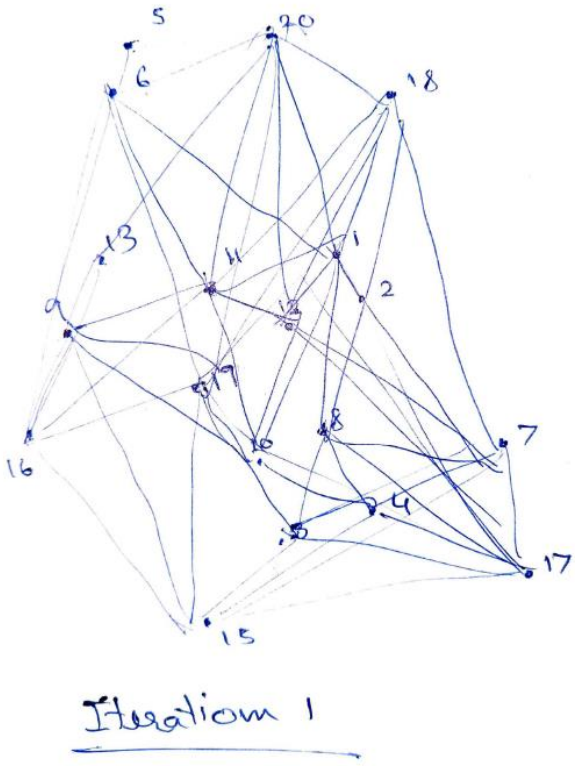
  - Now, since we have degree of each points, I can use smallest – Last ordering method to calculate the coloring order of the points.
  - As the name suggests, the points with the smallest degree is colored last.
  - The smallest degree point is taken and pushed into a stack (say colorOrderList)
  - The point is removed from the graph affecting the degrees of its neighbors, therefore the remaining points are re-arranged according to the new degrees.
  - These steps are repeated till there are no points left on the graph.
  - After this, the coloring is done in 'Last in first out' order same property as of the stack.
  - The following is a small example with 20 points, where point **No.14** has the smallest degree with size 10 & point **No.19** is the largest degree of size 18.
  - This shows the results after each iteration, as which points is removed & which points were its neighbors while deleting and how the degree table is updated after the deletion of point.



Iteration 1

Removed Point :14 : {5 6 9 11 12 13 16 1 19 20 }

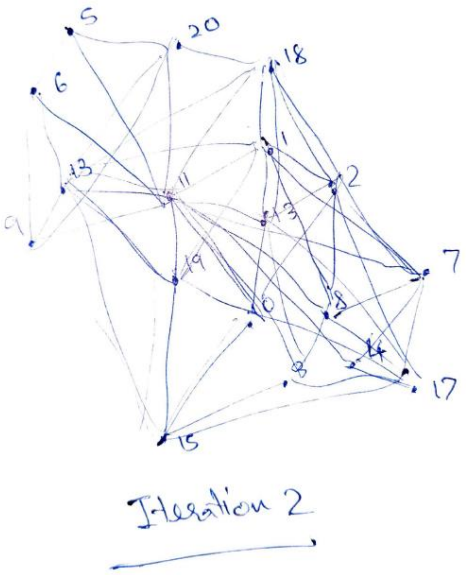
Degree Size	Points
10	16 17
11	5 7 18
12	4 6 20
14	3 15
15	9 10
16	2 13
17	1 8 11 12
18	19



Iteration 2

Removed Point :16 : {3 6 8 9 10 11 13 15 19 }

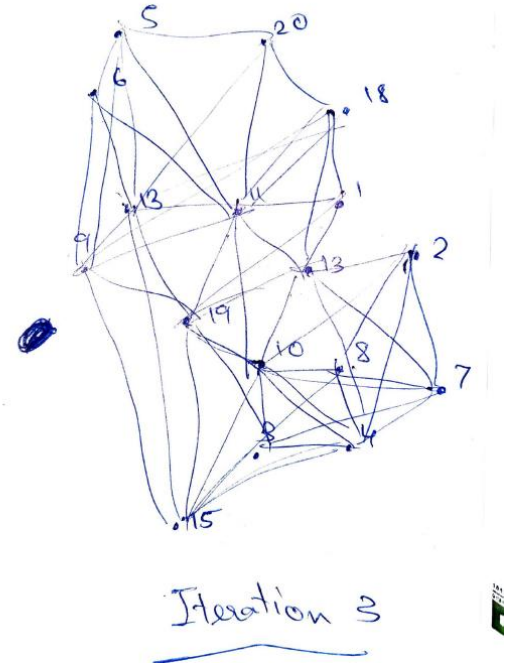
Degree Size	Points
9	17
10	5
11	7 18 6 20
12	4
13	3 15
14	9
15	10 13
16	2 11 12 1
17	8 19



Iteration 3

Removed Point :17 : {1 2 3 4 7 8 10 12 15 19 }

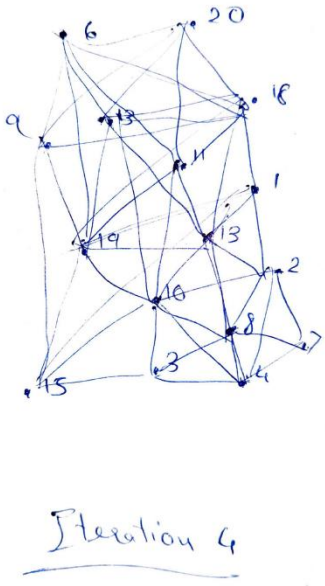
Degree Size	Points
10	5 6
11	7 18 20
12	4 3 15
13	9
14	10 13
15	11
16	2 12 1 8 19



Iteration 4

Removed Point :5 : {1 2 18 6 8 9 11 12 13 19 20 }

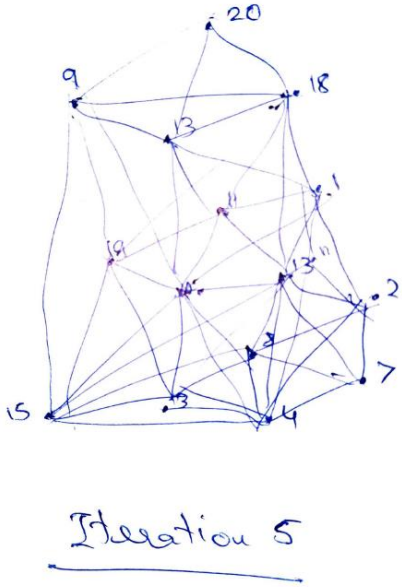
Degree Size	Points
10	6 7
11	18 20 3 4 15
13	9 10
14	13
15	11 1 2 8 12 19



Iteration 5

Removed Point :6 : {1 2 18 8 9 11 12 13 19 20 }

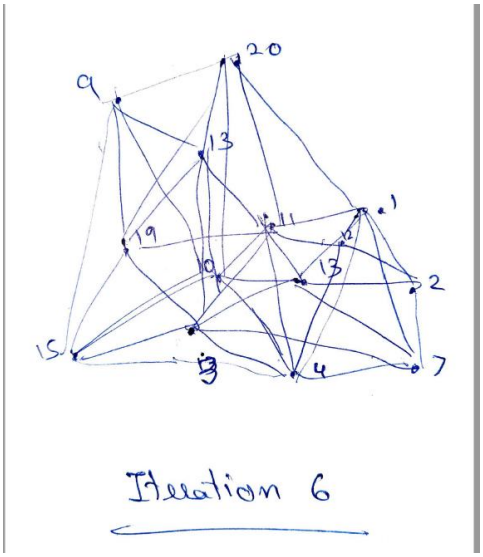
Degree Size	Points
10	7 18 20
11	3 4 15
12	9
13	10 13
14	1 2 8 11 12 19



Iteration 6

Removed Point :18 : {1 2 8 10 11 12 13 7 19 20 }

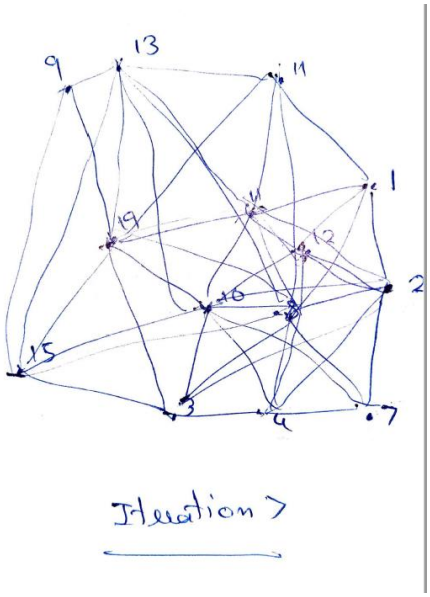
Degree Size	Points
9	20
10	7
11	3 4 15 9
12	13
13	10 1 2 8 11 12 19



Iteration 7

Removed Point :20 : {8 9 11 12 13 19 10 1 2 }

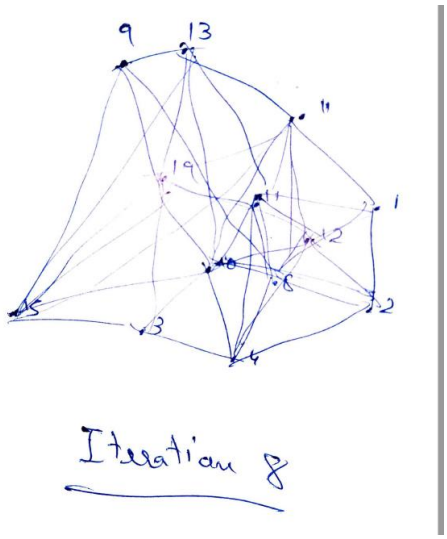
Degree Size	Points
10	7
11	3 4 15 9 13
12	1 2 8 10 11 12 19



Iteration 8

Removed Point :7 : {1 2 3 4 8 10 11 12 15 19 }

Degree Size	Points
10	9 13
11	3 4 15 8 11 12 19 10 1 2

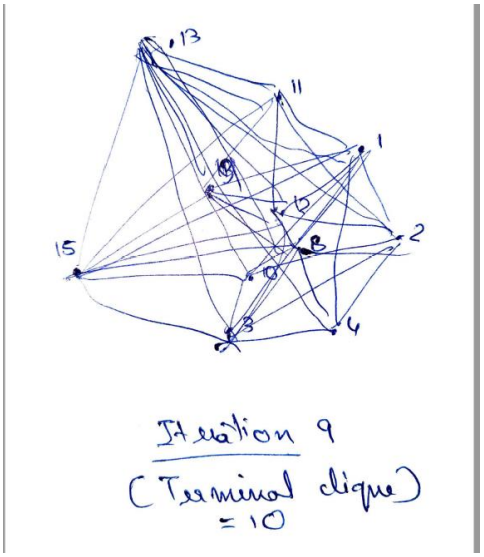


Iteration 9

Removed Point :9 : {8 3 10 15 1 2 4 11 12 13 19 }

After removing 9 we get Terminal Clique of **10** or complete graph, removing points further will keep giving us a complete graph from here on.

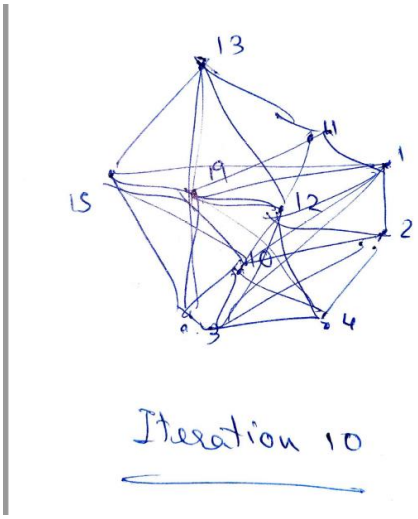
Degree Size	Points
10	13 1 2 3 4 8 10 11 12 15 19



Iteration 10

Removed Point :8 : {3 10 15 1 2 4 11 12 13 19 }

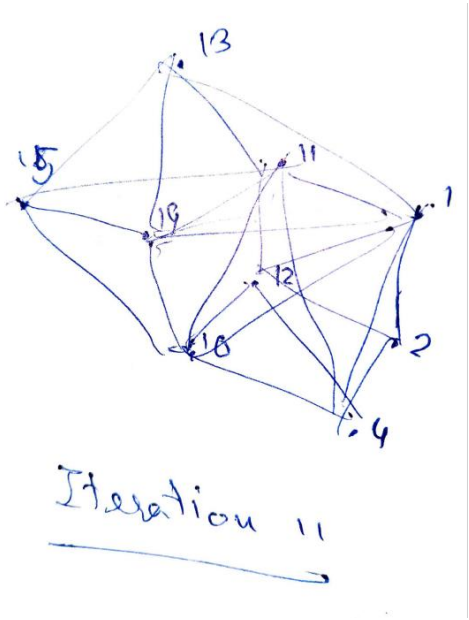
Degree Size	Points
9	3 10 15 1 2 4 11 12 13 19



Iteration 11

Removed Point :3 : {1 2 4 10 11 12 13 15 19 }

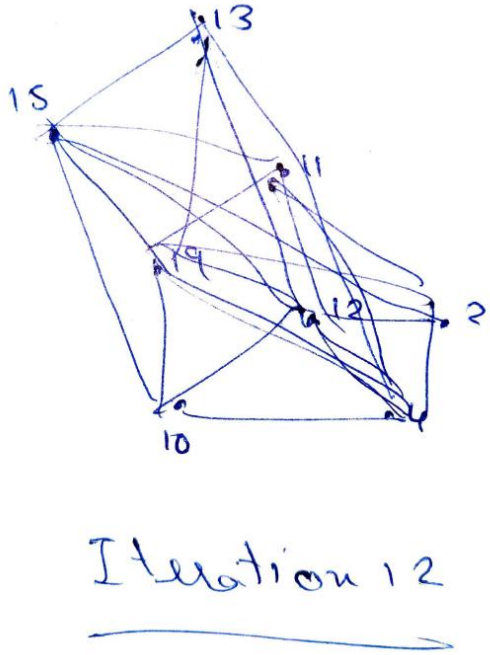
Degree Size	Points
8	10 15 1 2 4 11 12 13 19



Iteration 12

Removed Point :1 : {4 2 10 11 12 13 15 19 }

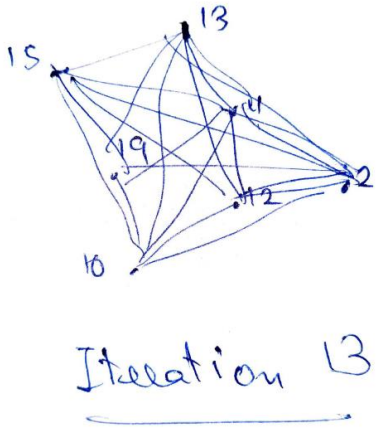
Degree Size	Points
7	2 4 10 11 12 13 15 19



Iteration 13

Removed Point :4 : {2 10 11 12 13 15 19 }

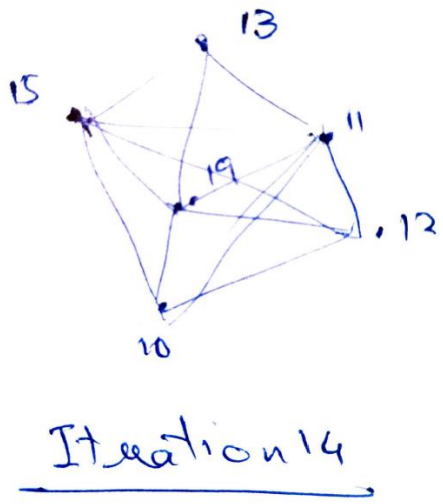
Dgree Size	Points
6	2 10 11 12 13 15 19



Iteration 14

Removed Point :2 : {10 11 12 13 15 19 }

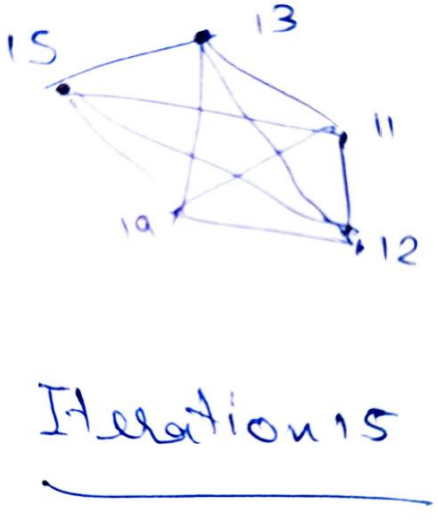
Degree Size	Points
5	10 11 12 13 15 19



Iteration 15

Removed Point :10 : {11 12 13 15 19 }

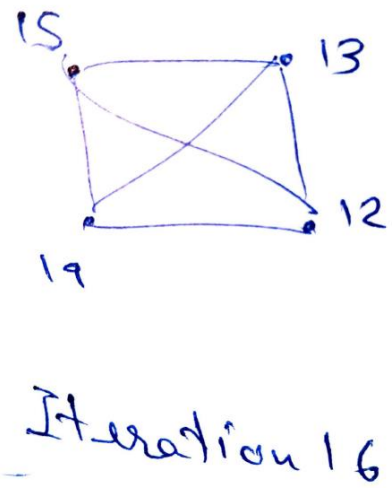
Degree Size	Points
4	11 12 13 15 19



Iteration 16

Removed Point :11 : {15 12 13 19 }

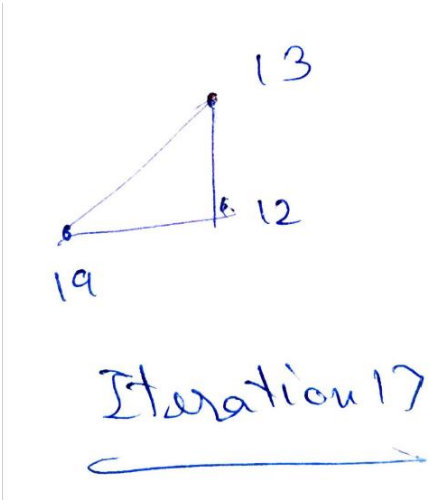
Degree Size	Points
3	12 13 15 19



Iteration 17

Removed Point :15 : {12 13 19 }

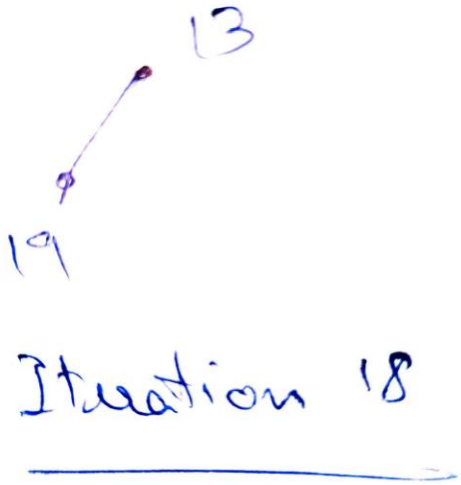
Degree Size	Points
2	12 13 19



Iteration 18

Removed Point :12 : {13 19 }

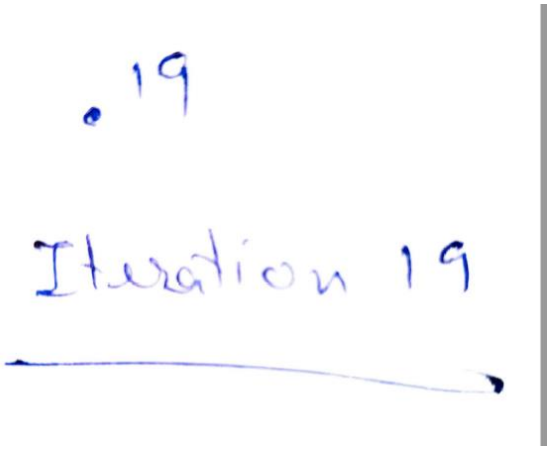
Degree Size	Points
1	13 19



Iteration 19

Removed Point :13 : {19 }

Degree Size	Points
0	19



Iteration 20

Removed Point :19 : {}

Degree Size	Points

- We just saw how step by step how smallest -last coloring work.

Algorithm Engineering :

- For this, we are assigning points into degree List, where index of degree list is also the degree value of that degree. Eg : degree[5] will contain all points with degree 5.
- By doing this, we can quickly add & remove points from 1 degree to lower, this is useful when a point is removed from graph and pushed into the stack, the degree of its neighboring points is lowered by 1.
- Code :

```
for (Point neibghor : currentPoint.getNeighbourPoints())
{
	degrees[ neibghor.getNeighbourPoints().size() ].removePoint( neibghor );
	neibghor.getNeighbourPoints().remove( currentPoint );
	degrees[ neibghor.getNeighbourPoints().size() ].addPoint (neibghor );
}
```

- Here we do not need looping for sorting the graph all over again according to the new updated degree values. So, this becomes linear time where time complexity becomes **θ(N*AN)**
  Where N is number of points & AN is average degree of points.
- The point is then added to the coloring queue and is all necessary data like, terminal clique, max degree when deleted, etc. are collected.
- Coloring is done in the reverse order of the queue as it is a smallest-last coloring algorithm.
- I get the colors already acquired by its neighbors and then I add assign a new color to the point which is not assigned to any of its neighbors. For acquiring colors, we need to iterate through all its neighbors, so time complexity here would be **θ(N*NE)** where NE – number of neighbors
- After coloring all points, I plot the points on the graph.
- During this process, we also collect the data relevant for part 2 & 3 of the project.

- **Pairing Top 4 Color Points :** from the color-List we select the 4 largest color sets, make all possible pairs of them & get total of 6 pairs.
  - These pairs will be used to plot bipartite graphs.
  - But before generating the bipartite graphs, we need to clean the pairs before using it for optimal backbone for the graphs.

- **Deleting Minor components :** we need the backbone to be connected to each other, then only the data would be able to flow between them. So, we eliminate all the smaller disconnected graph formed by pairing points of any 2 colors at a time.
  - This is done using the 'Breath First Search' algorithm.
  - Where we visited every sibling points before doing into the next dept.
  - As we visit the points I assign them with visited flag as true.
  - After visiting all the points when the execution returns to the starting Point, we get 1 complete component.
  - We then check we there are any points of the pair whose visited flag is not set to true. If there exist such points, it means that there are more than 1 component in the backbone graphs.
  - We keep search for components until all points are not visited.
  - Once done, we take the component with the largest order (number of Points)
  - The time complexity would be **Θ(V + E)**, where V is the number of vertices, and E is the number of Edges , this is same as the time complexity of 'Breath first search' algorithm.

- **Removing tails :** Tail is points with degree 1.
  - We remove all points with degree 1 & update its neighboring point.
  - It is possible that after updating the degree of the neighbors also become 1.
  - We delete them also & keep doing till there are no points with degree 0 or 1.
  - **θ(N*NE)** where N is number of points with degree 1 & NE are its neighbor with degree 2

- **Calculating size & selecting Top 2 :** This part is simple, we calculate the number of edges in the 6 pairs of bipartite & select the top 2 from them.
    - These 2 pairs are the final backbone set for our graphs, these are plotted on a separate frame
    - Time complexity here would be **θ(N*AN)**
      Where N is number of points in the Bipartite graph & AN is average degree of points.

- **Calculate Domination :** This is the coverage of our backbone graph. The number of points it covers throughout our topology**.**
    - We calculate this by summing up the original edges or degrees of points in the backbone bipartite graph (call it SUM1). Here the original degrees mean edges between all its neighbors and not just the backbone points.
    - We also Sum the edges or degrees of points in the original non-colored graph (call it SUM2).
    - By calculating what percent of SUM2 is SUM2 will give us the coverage.
    - Here I was previously using array-List and contains() method to calculate unique edges in the bipartite graphs which was taking me around 6,7 seconds to calculate for 128k by 128 benchmark
    - I replaced that with HashSet & removed the conditional check, this reduced performance time to 0.8 seconds.
    - This was more than 5000% increase in speed.
    - Time complexity here would be **θ(N*AN)**
      Where N is number of points in the Bipartite graph & AN is average degree of points.

- **Generate Graph**[5] **:** the data stored or calculated above is passed over to canvas object which generates the graph.
    - **Plot Points :**
        - runs a loop for each point to draw point on graph.
        - This is done using graphic's drawRectOval() function
        - setColor() function sets color for the points.
        - I'm coloring only the top 4 colors to avoid cluttering on graph.
        - Time complexity o(n). n = number of points.
    - **Plot Edges :**
        - runs a loop for each points to draw edges with it neighboring points on graph.
        - This is done using graphic's drawLine() function
        - Time complexity o (n * avgDegree). n = number of points, average degree is a constant value

    - **Plot Min Max Degree sensors :**
        - As we plot edges , I also store values for all degree values and store them in an array of degree class
        - Lowest degree is colored in RED while highest is colored in GREEN on the graph.

Screenshots : The screenshots of the outputs, degree & color plot are following.

Generate Histogram[4] **:**

- From the degree data collected, histogram is generated on a different frame.
- The X axis the degree values in ascending order.
- Y axis are the number of occurrences of that degree.
- From the histogram we see that degree occurrences are standardly distributed.
- This shows that the graph generated is correct.

Generate Degree Line plot :

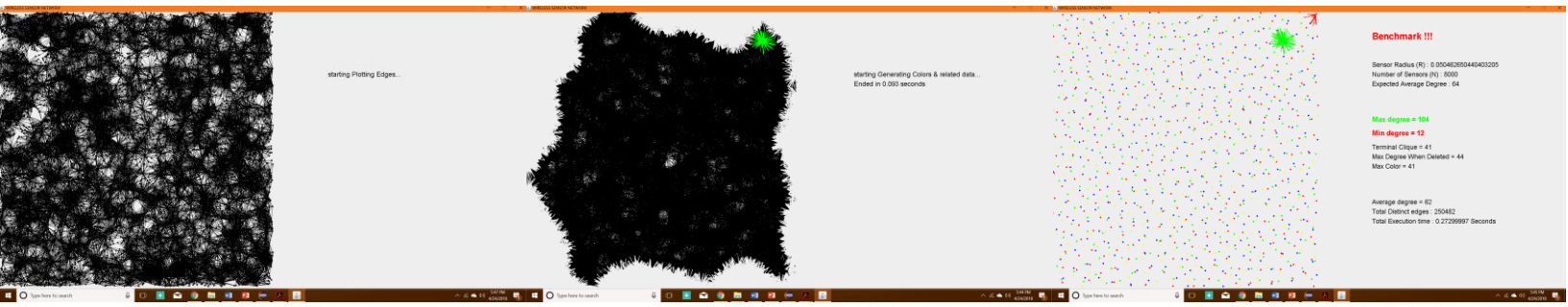
- From the original degree & degree when deleted data collected, Line can be plotted on a different frame.
- The X axis the number of points.
- Y axis are the value of the 2 degrees of that particular point.
- From the line plot we see that 2 degrees line has a horizontal average values.
- This shows that the graph generated is correct.

Generate Color Line plot :

- After coloring is done on every point, Line can be plotted on a different frame.
- There is no need to sort it as color 1 will always be greater than color 2.
- The X axis the number of color sets.
- Y axis is number of points in that color set.
- Looking at the screen shots of the graph below shows the line goes down as color sets increases.
- This shows that the graph generated is correct as color 1 will always be greater than color 2.

Visualizations : Visualizations are done in the project to better understand the flow.

- Since I didn't wanted the animation to be too slow or too fast, I'm operating the wait time in nano-seconds.
- I did it by creating a custom class instead of using Thread.sleep() method.



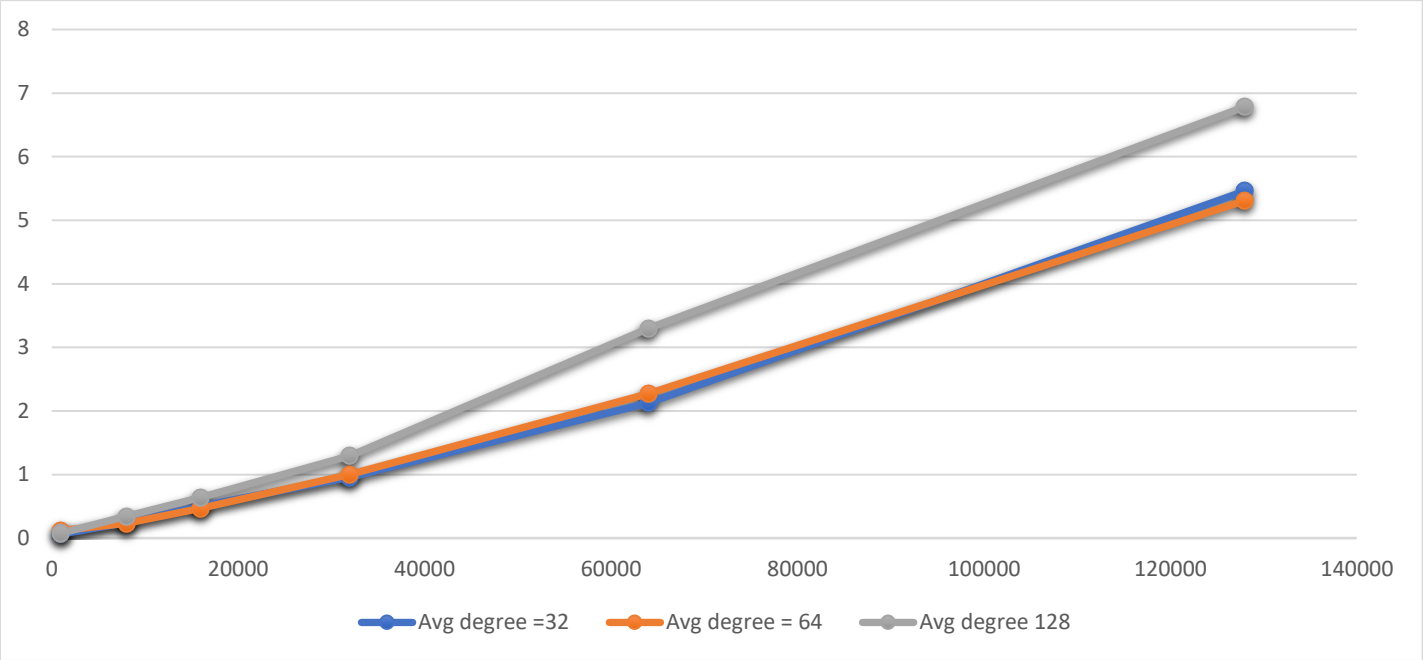
Running the project : To run the project you need, you need to have java environment set up on your computer.

- Copy paste the 'algoFinal.jar' to your desired location.
- Open terminal and change the directory to the location where the file is stored.
- Run the command '**java -jar algoFinal.jar**'

Runtime Plots : Shows the time performance in milliseconds of the application. The line is almost a linear graph, Indicating us that the project runs in linear time.

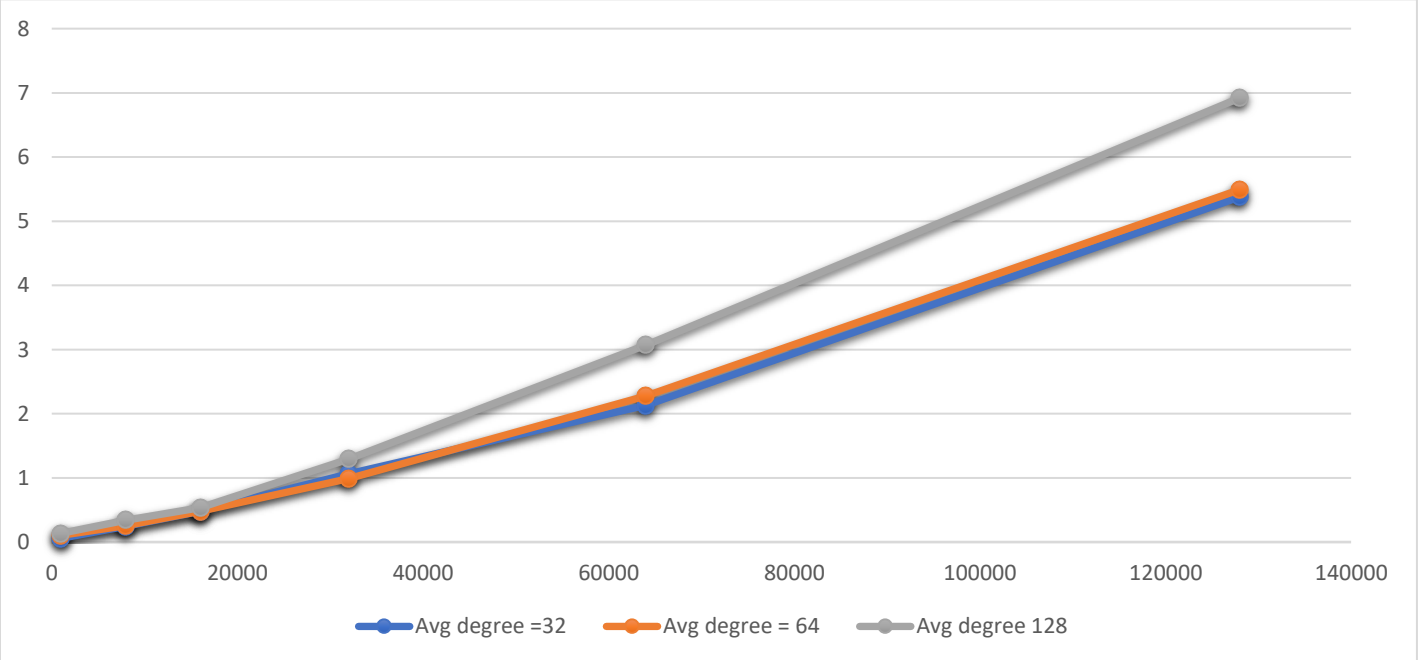
Topology : Square

X axis = time in Seconds , Y axis = Number of sensors



Topology : Disk

X axis = time in Seconds , Y axis = Number of sensors

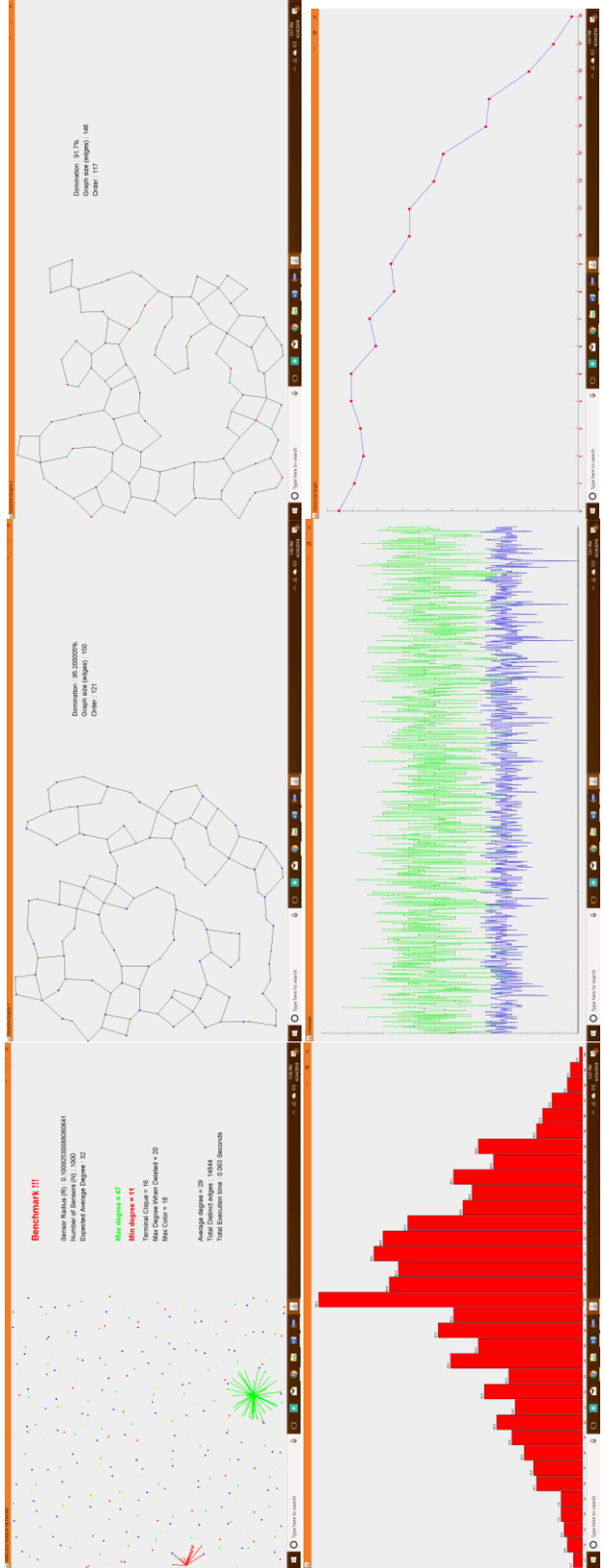


Output Screenshots for each benchmark : each page contains details of 2 benchmark followed by screen-shots of those 2 benchmarks respectively.

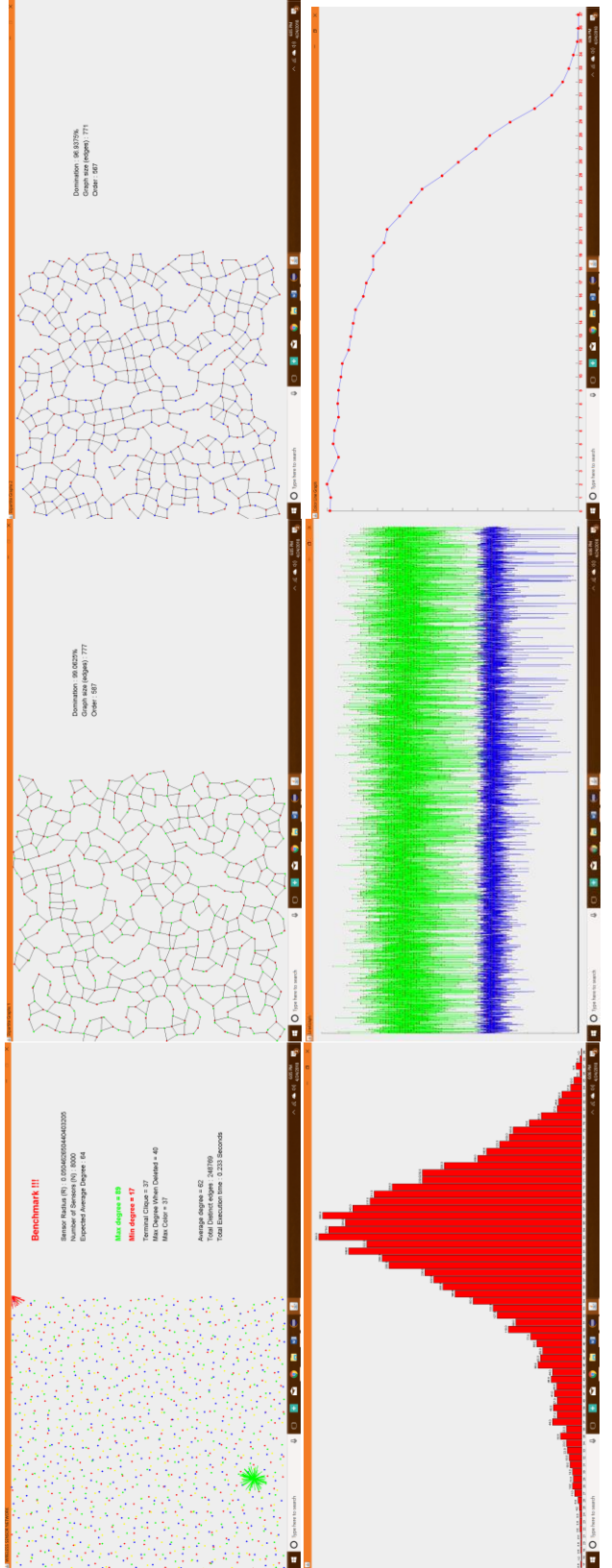
- The summary table repeated again for quicker access during observation.
- There are 6 Screen-Shots for each benchmark, which are in order from left to right, top to bottom:
    1. Top 4 colored points shown in graph.
    2. Bipartite graph 1 (largest)
    3. Bipartite graph 2 (2^{nd} largest)
    4. Histogram of distribution of points according to their degrees
    5. Line graph of original degree value & degree when deleted value for each point.
    6. Line graph of distribution of colors. Number of points which are assigned a color.

	Run-time in seconds	0.0 63	0.2 33
Phase 3	Order for BG2	117	567
	Order for BG1	121	587
	Edges for BG2	146	771
	Edges for BG1	150	777
	Domination of BG2 (%)	91. 7	96. 937
	Domination of BG1 (%)	95. 200	99. 062
Phase 2	Max Terminal Clique	16	37
	Max Degree when deleted	20	40
	color size	18	37
Phase 1	Realized average degree	29	62
	Max Degree	47	89
	Minimum Degree	11	17
	Number of distinct Edges	148 44	248 769
	Radius (R)	0.1 009	0.0 504
	Topology	Squ are	Squ are
	Estimated Average Degrees	32	64
	Number of sensors	100 0	800 0
	Benchmark #	1	2

BenchMark 1 :

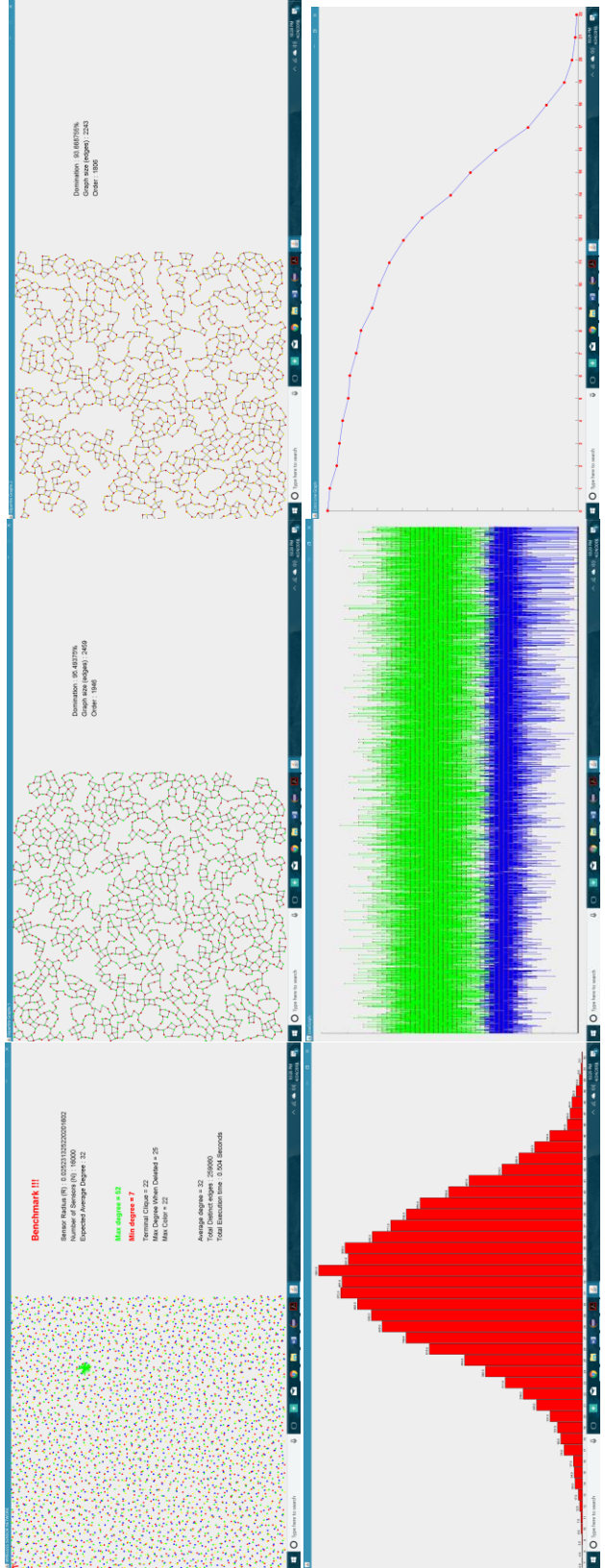


BenchMark 2 :

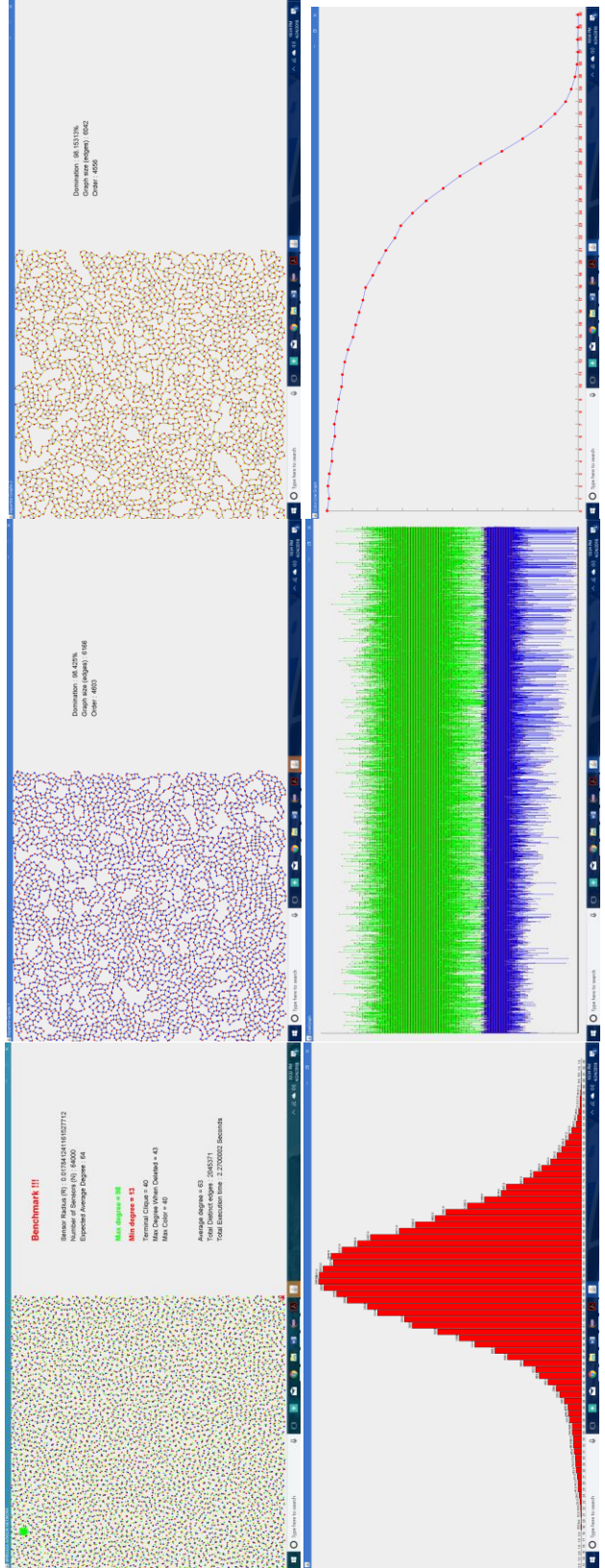


	Run-time in seconds	0.504	2.2700002
Phase 3	Order for BG2	1806	4556
	Order for BG1	1946	4603
	Edges for BG2	2243	6042
	Edges for BG1	2459	6166
	Domination of BG2 (%)	93.668	98.153
	Domination of BG1 (%)	95.493	98.425
Phase 2	Max Terminal Clique	22	40
	Max Degree when deleted	25	43
	color size	22	40
Phase 1	Realized average degree	32	63
	Max Degree	52	98
	Minimum Degree	7	13
	Number of distinct Edges	259060	2045371
	Radius (R)	0.0252	0.0178
	Topology	Square	Square
	Estimated Average Degrees	32	64
	Number of sensors	16000	64000
	Benchmark #	3	4

BenchMark 3 :

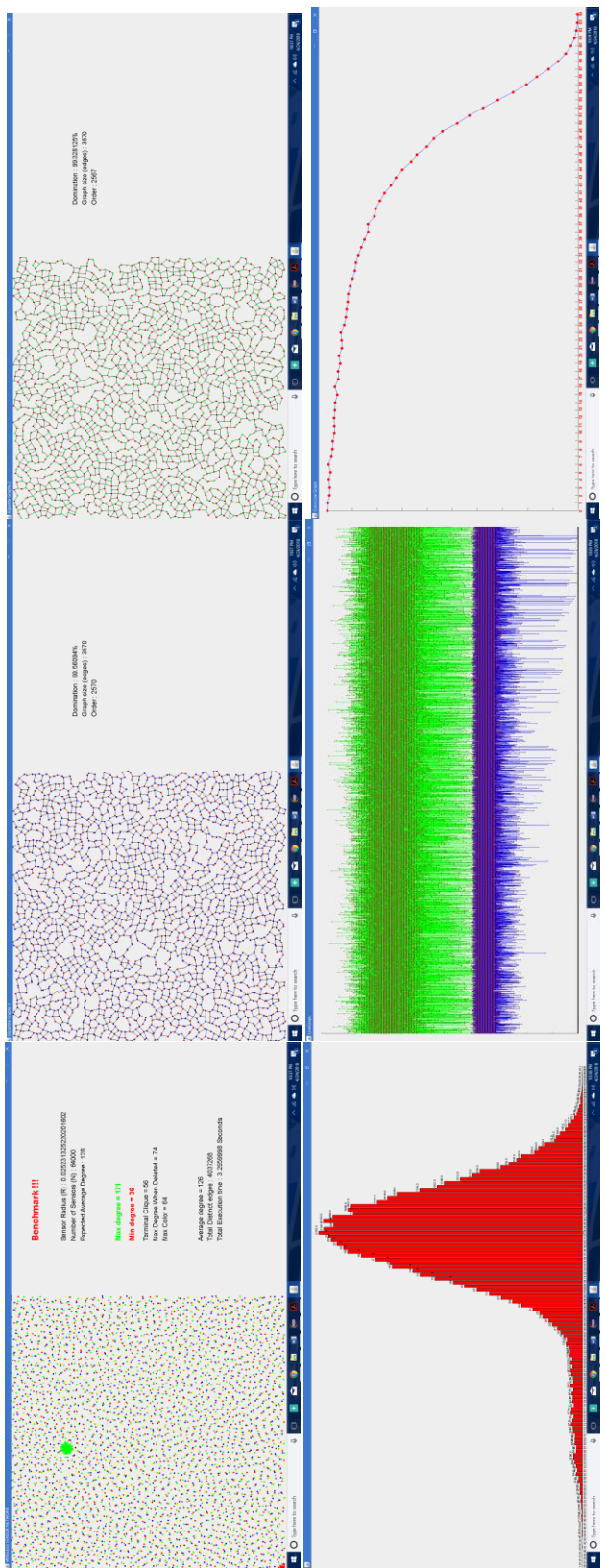


BenchMark 4 :

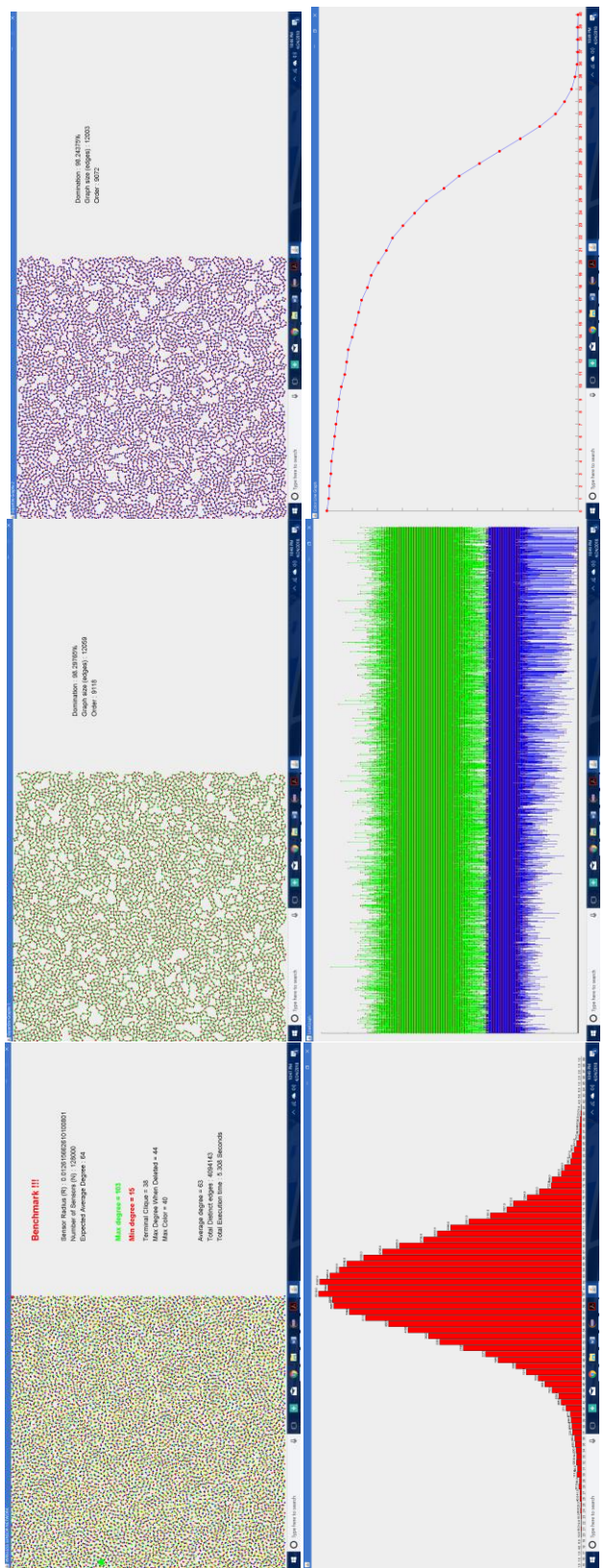


	Run-time in seconds	3.2959998	5.30 8
Phase 3	Order for BG2	2567	9072
	Order for BG1	2570	9118
	Edges for BG2	3570	12003
	Edges for BG1	3570	12059
	Domination of BG2 (%)	99.328	98.243
	Domination of BG1 (%)	99.560	98.297
Phase 2	Max Terminal Clique	56	38
	Max Degree when deleted	74	44
	color size	64	40
Phase 1	Realized average degree	126	63
	Max Degree	171	103
	Minimum Degree	36	15
	Number of distinct Edges	4037268	4094143
	Radius (R)	0.0252	0.0126
	Topology	Square	Square
	Estimated Average Degrees	128	64
	Number of sensors	64000	128000
	Benchmark #	5	6

BenchMark 5 :

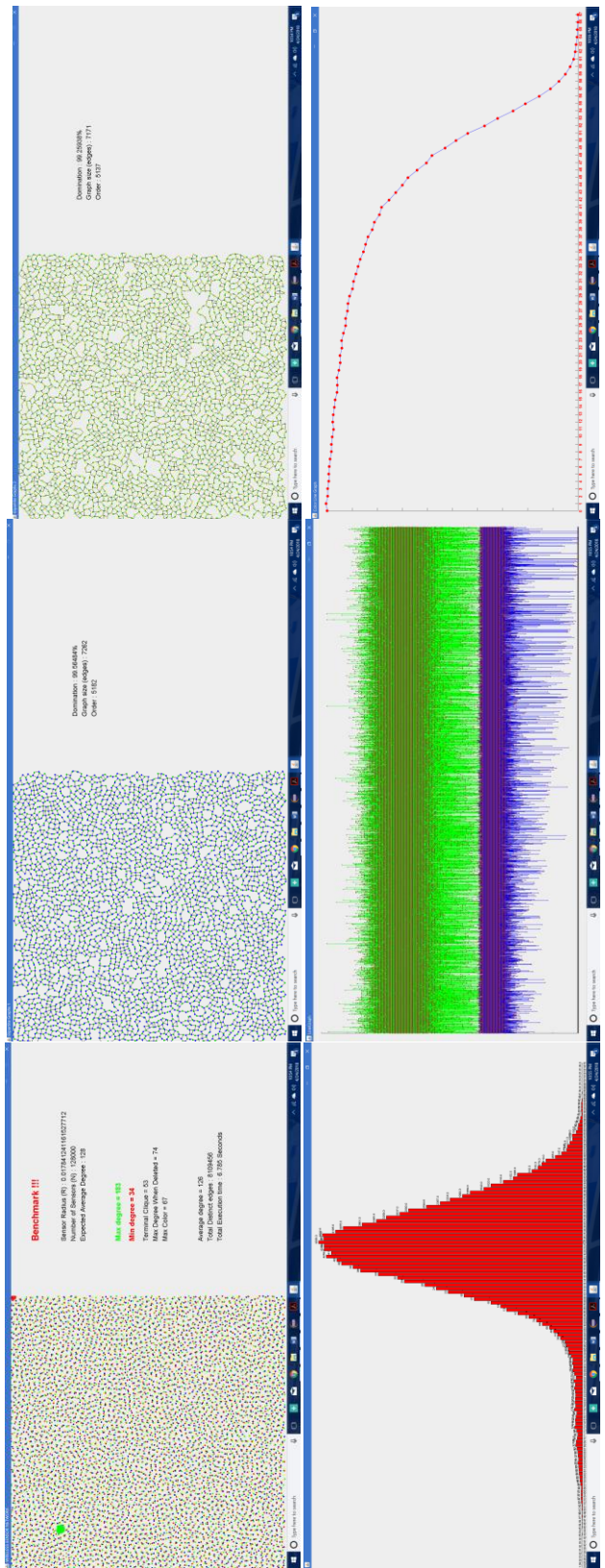


BenchMark 6 :

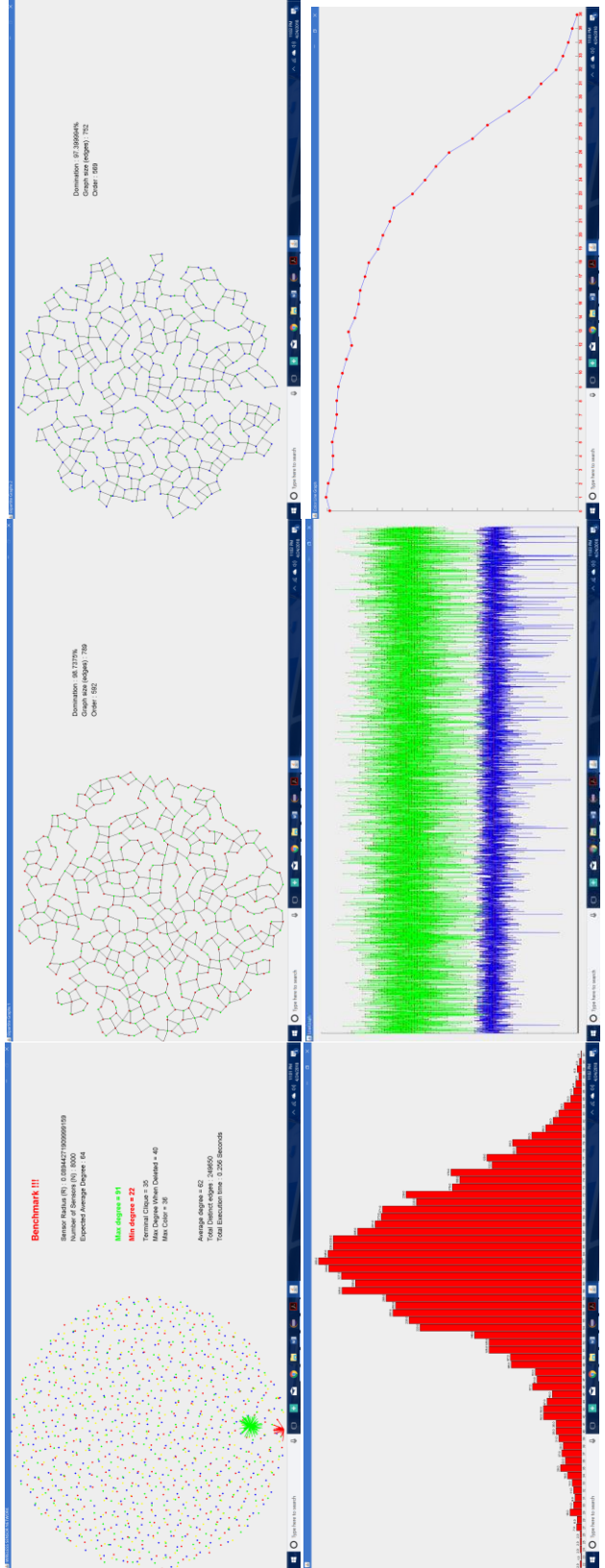


	Run-time in seconds	6.785	0.256
Phase 3	Order for BG2	5137	569
	Order for BG1	5182	592
	Edges for BG2	7171	752
	Edges for BG1	7262	789
	Domination of BG2 (%)	99.259	97.399
	Domination of BG1 (%)	99.564	98.737
Phase 2	Max Terminal Clique	53	35
	Max Degree when deleted	74	40
	color size	67	36
Phase 1	Realized average degree	126	62
	Max Degree	183	91
	Minimum Degree	34	22
	Number of distinct Edges	8109456	2496506
	Radius (R)	0.0178	0.0894
	Topology	Square	Disk
	Estimated Average Degrees	128	64
	Number of sensors	128000	80000
	Benchmark #	7	8

BenchMark 7 :

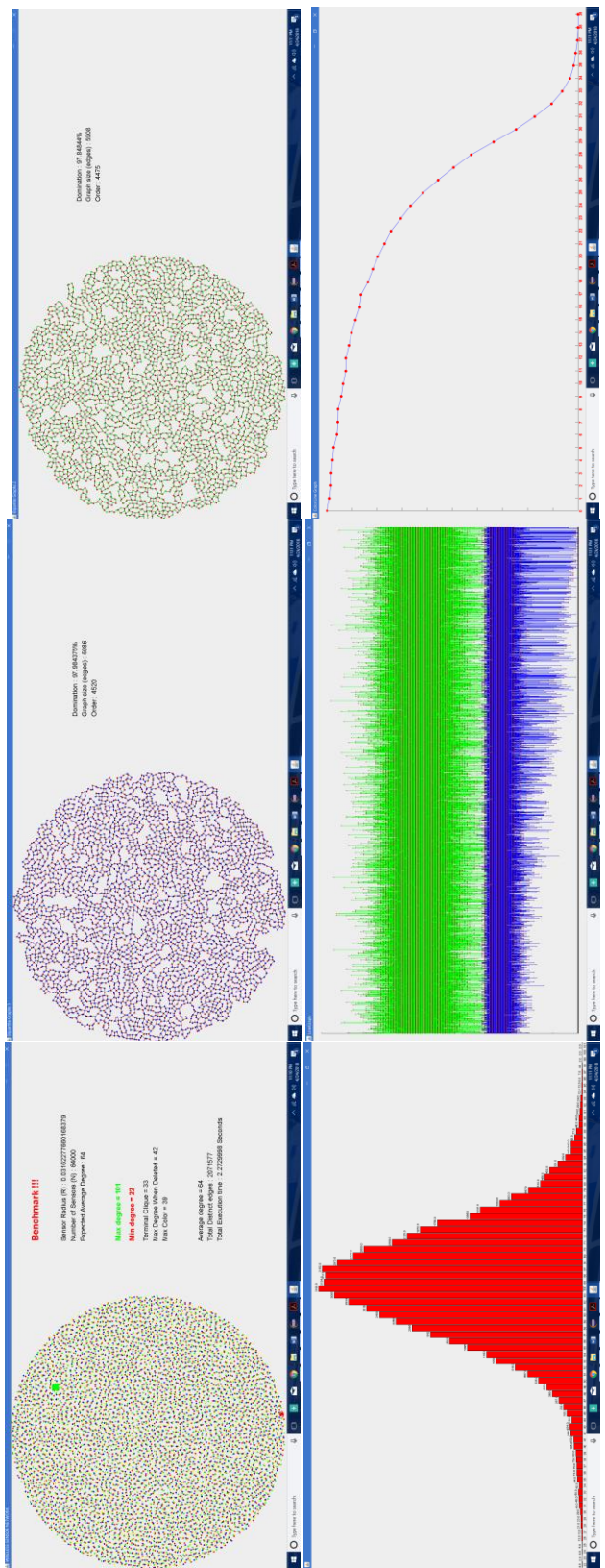


BenchMark 8 :

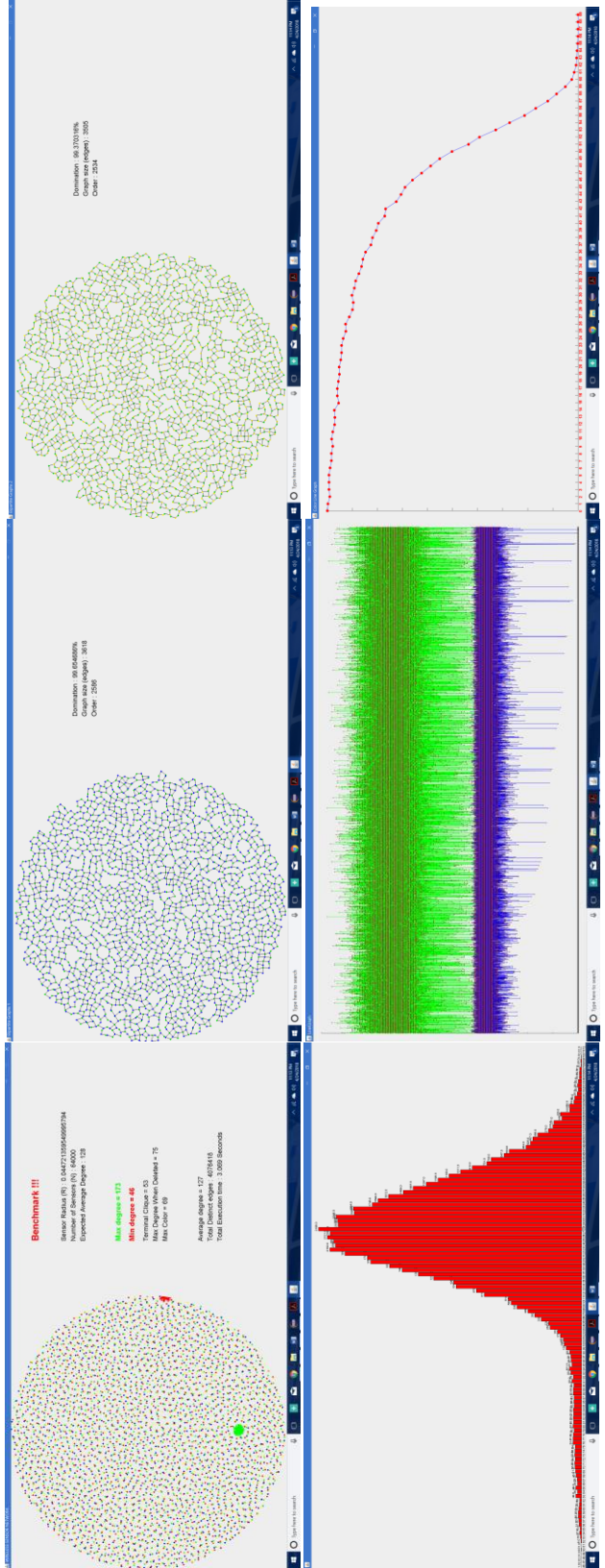


	Run-time in seconds	2.2729998	3.069
Phase 3	Order for BG2	4475	2534
	Order for BG1	4520	2586
	Edges for BG2	5908	3505
	Edges for BG1	5986	3618
	Domination of BG2 (%)	97.848	99.370
	Domination of BG1 (%)	97.984	99.654
Phase 2	Max Terminal Clique	33	53
	Max Degree when deleted	42	75
	color size	39	69
Phase 1	Realized average degree	64	127
	Max Degree	101	173
	Minimum Degree	22	46
	Number of distinct Edges	2071577	4076418
	Radius (R)	0.0316	0.0447
	Topology	Disk	Disk
	Estimated Average Degrees	64	128
	Number of sensors	64000	64000
	Benchmark #	9	10

BenchMark 9 :



BenchMark 10 :



References :

- [1] DAVID W MATULA AND LELAND L. BECK
  Smallest-Last Ordering and Clustering and Graph Coloring Algorithms,
  https://smu.instructure.com/courses/36633/files/folder/Term%20Project?preview=938531, Sept 1982

- [2] ZIZHEN CHEN AND DAVID MATULA
  Bipartite Grid Partitioning of a Random Geometric Graph,
  2017 13th International Conference on Distributed Computing in Sensor Systems

- [3] **Piyush Metkar's report (previous student) :** 4750918 (only report , no codes
-)
- [4] **Histogram :** https://stackoverflow.com/questions/6849151/bar-chart-in-java
- [5] **Zizhen PPT & slides**