Wireless Sensor Network Project

Advisor: Zizhen Chen

Jiaqing Ji

SMUID:47508881

1. Executive Summary

I. Introduction and Summary

Wireless sensor network(WSN) is physical network which is built to connect the whole area using wireless sensors, to optimize this network, we try to find the most scientific layout of the wireless sensors, that is, providing a communication backbone for every network which is randomized generated, to build a backbone, we have to first do coloring to all vertices in RGG, choose 4 biggest color class and combine every distinct two of them and get a network.

In this project, I implement an algorithm for determining a coloring, terminal clique, and a selection of bipartite subgraphs that are produced by an algorithm for graph coloring in a random geometric Graph(RGG). In Part I, I implement several RGG's on the unit square and several RGG's on a disk which model WSN's geographic regions, both are based on benchmark set and are randomized generated. For each RGG, I maintained an adjacency list and a degree list as output to be input for Part II.

In Part II, I implement the smallest last coloring algorithm for vertex coloring and terminal clique identification in the RGG using adjacency list and degree list as input in Part I. I also give a walkthrough for an RGG in the unit square with a small amount of vertex number to show the whole smallest last ordering procedure and coloring procedure step by step.

In Part III, I choose four largest color classes of each of the benchmark graphs, and build 6 pairs of bipartite subgraph, for these pairs, I determined which top two possible bipartite subgraphs with most edges in the maximum connected subgraph, and determined that two as backbone of the RGG. The domination percentage of the backbone shows the performance of my algorithm and model the Coverage of the WSN.

In my implementation, due to the data structure and algorithm I used, I achieve linear time (Θ (|V|+|E|)) for Part I and Part II, and the running time of the whole procedure is considerably short especially for large number of vertices in the RGG. Other than the walkthrough of a small graph mentioned before, I also provided graphs like degree distribution, color class distribution, original and deleted degree information for vertices for RGG in each benchmark, the graphs will be showed in ***Result Summary part***. Running time plot for each Part of the project to test the time complexity is given in algorithm engineering part in **Reduction to Practice part**.

II. Programming Environment Description

I. Hardware Environment:

Brand: MacBook Pro (13-inch, 2017, Four Thunderbolt 3 Ports)
Processor: 3.5GHz Intel Core i7
Memory: 16GB 2133MHz LPDDR3
OS: mac OS Sierra version 10.12.6

II. Software Environment:

Language: Java 1.8.0_144, Processing 3
PDE: Processing 3.3.6
Library: Java.util.*, Java.math.*, Java.io.*

III. Description:

Java is the language that is most familiar to me, so it's my first choice of programming language. Java is an object-oriented language, it's simplicity lies great on its consistency. The principle of Java is encapsulation, inheritance and polymorphism. It is an interpreted and portable language which means application written in Java can write once and run everywhere. It has a great memory mechanism and allocates and
deallocates memory for you. It needs you to design everything as an object and it also have

the best library. In this project, we need to use collection frameworks in java.util.* and math functions in java.math.*.

I used Processing to create a graphic drawing and display. Processing is a programming language based on java, so it's very friendly, powerful and also very flexible. Processing is designed by and for artists, so it can create a wonderful graphic interface, it can draw basic flat graphics, it supports dynamic drawings and it's easy to build interaction with users. Also, I can use processing to implement different kinds of data visualization like dot plots, scatter plots, line plots and bar charts. These can cover most of the needs in graphic user interface in this project, and the project can be saved as a desktop application or a JavaScript or an Android application so it supports cross-platform utilization, so the combination of Java + Processing is the best choice for my project design.

During the runtime of 128000 vertices with 64 estimate degrees on a square area, , my program uses 6.4% of CPU, 21% of memory, and create 44 threads for data structure creating, memory allocation and computing. I think the hardware environment and performance will greatly affect the running time of a program, even with the same algorithm.

III. Summary Table:

	#	Number of Vertices	Est.Avg Degree	R	Number of Edges	Min. Degree	Max. Degree	Avg. Degree	Part I Running Time	Max Degree when	Number of Colors	Max Color Class Size	Terminal Clique Size	Part II Running Time	Number of Edges in Largest bipartite	Domination Percentage	Part III Running Time
Square	1	4000	32	0.05125	63375	6	56	31.69	0.686s	23	21	288	20	0.055s	568	92.73%	0.034s
	2	8000	64	0.05086	256402	15	90	62.101	2.590s	38	37	329	28	0.116s	716	95.36%	0.082s
	3	16000	64	0.03596	505706	16	94	63.21	4.074s	40	38	634	32	0.173s	1511	98.58%	0.14s
	4	32000	64	0.02543	1018361	16	98	63.65	7.097s	42	40	1259	37	0.272s	3036	98.64%	0.202s
	5	64000	32	0.01265	1043155	5	58	32.602	5.969s	25	25	4443	24	0.289s	9013	93.59%	0.515s
	6	64000	64	0.01798	2049124	17	101	64.04	12.642s	42	41	2483	36	0.435s	6040	98.81%	0.411s
	7	64000	128	0.02533	4036083	27	175	126.13	37.505s	74	66	1358	61	0.674s	3502	99.28%	0.363s
	8	128000	64	0.01271	4114168	33	98	64.28	22.766s	40	39	5085	36	0.59s	13945	98.09%	0.588s
Disk	9	8000	64	0.09013	250066	17	96	62.52	2.281s	39	38	321	35	0.135s	774	98.48%	0.066s
	10	8000	128	0.12697	488375	34	166	122.09	5.356s	72	65	177	58	0.176s	444	98.76%	0.072s

IV. Reference:

- [1] Introduce and Summary: reference to

Introduce and Summary q7350-f17 PROJECT WSN V1_0.docx

- [2] Reduction to Practice Part II: reference to

Zizhen Chen and David W Matula, Bipartite Grid Partitioning of a Random Geometric Graph, 2017

2. Reduction to Practice

I. Part I: Random Geometric Graph Generation and Display

1. *Demand Analyze:*

The main purpose of Part I is to random generate thousands of vertices in a given topology (Unit Square, Unit Disk, Unit Sphere), then use given total vertices number **Num** and Expected Average Degree **avgD** to calculate the radius of area that each vertex searches adjacent vertices in:

Num / AreaTotal = (avgD + 1) / AreaSearch

AreaSearch = PI × R^2

So:

R^2 = ((avgD+1) × AreaTotal × PI) ÷ Num

If you random generate Num vertices in an area AreaTotal, then since Num is big enough to make all the vertices randomly distributed, circle an area with (avgD+1) numbers of vertices, the area will be (AreaTotal × (avgD+1)) ÷ Num. *I use (avgD + 1) because that will assure the degree of center of the circle is avgD, if we use avgD for calculate, the degree of center of the circle will be avgD-1.*

With the radius R, we have to go through every vertex to line every vertex which distance to it is less than R, that is, use every vertex as center and R as radius and line every vertex in the circle to the center. We also need to build an adjacent list for each vertex. And then get some statistic information about degrees and edges.

2. *Algorithm Description:*

The basic way to random generate vertices is to use random number.

1) In a Square, we can create a Cartesian Coordinate System, use a random number X between minimum abscissa and maximum abscissa, use random number Y between minimum ordinate and maximum ordinate, then use Point (X, Y) to generate and draw a point.

2) In a Disk, we can create a **Polar Coordinate System,** since $x^2 + y^2 = R^2$, let x = RcosΘ and y = RsinΘ, use random number r between 0 and sqrt(R), use random number Θ between 0 and

2×PI, in this way we can mapping polar coordinate to Cartesian coordinate.

I use ***Cell Method*** to implement Unit Square and Unit Disk, the basic idea is:

1) Split the whole area into several same small squares with side length = R, even in the Disk, we'll also find its external rectangle and split it. We call the small squares ***cells.***

Horizontally we will have 1/R (take the ceiling) cells, so will vertically, so the whole area is divided into (1/R × 1/R) cells, and I give each of them a ID cellID.

2) When I randomly generate vertices in the area, I'll calculate which cell does each vertex belong to base on its coordinate.

3) After all vertices are draw, traverse all cells in order of cellID. For each cell C, traverse all vertices inside the cell, and instead of calculate distance between a vertex to every vertex in the whole area, we just have to calculate all vertices within cell C, in the cell below cell C, in the next cell, the cell on top of next cell and the cell below next cell.

4) The reason is because the length of side is equal to R, the only chance that a vertex is in the range of R to a vertex in cell C is it's in the cells around cell C or it's in cell C. That's *9 cells totally*. However, there's 4 cells that already calculated distance to cell C. ***So instead of calculate distance between a vertex to every vertex in the whole area, we just have to calculate all vertices in 5 cells.***

5) Draw a line between every pair of vertices whose distance shorter than R, and add 1 degree to the both vertices (add 0.5 if they are in the same cell), add each of these two vertices to another's adjacent list for input of Part II.

6) Find the vertex with the minimum degree and the one with the maximum degree, and shading the disk with radius R around that two vertices with different colors.

```
//Cell Structure
Map<Integer, ArrayList<Point>> Cells =
  new HashMap<Integer, ArrayList<Point>>();
```

3. *Data Structure Design in Part I:*

1) *Point Class:*

Define a Point Class which has Class member variable *x* implies X coordinate, *y* implies Y coordinate, *degree* implies number of degree this vertex has, and a Point type ArrayList *AdjPoints* to store adjacent list of this vertex.

```
class Point{
   private float x;
   private float y;
   private float degree;
   private ArrayList<Point> AdjPoints = new ArrayList<Point>();

   Point(float X, float Y, float d) {
     x = X;
     y = Y;
     degree = d;
   }
}
```

The reason why I set degree as float type is, if two vertices A and B are in the same cell and the distance between them is less than R, first A find B and both plus 1 to their degrees, then B find A and both plus 1 to their degrees, then that's a duplicate. My way is when a vertex finding points in its own cell and get one, it only plus 0.5 to their degrees.

2) Points:

Define a Point Type Array Points to store all these vertices' information.

```
int Num = 4000;
Point [] Points = new Point[Num];
int avgD = 32;
```

3) Cells:

Define a Map implies Cells structure, use cellID as key and a Point type ArrayList as value.

Each Point type ArrayList implies a single cell, it stores all the points which locate in the cell in the Coordinate System based on cellID after being random generalized.

4) DegreeDistribute:

Define a int type Array DegreeDistribute to count how many vertices with each degree from 0 to Num there is.

```
//Degree Distribute
int [] DegreeDistribute = new int[avgD * 3];
```

It's not likely for a vertex to have a degree larger than 3 times avgD because it's random generalize. Otherwise there'll be an intensive area. So, I set the length of the array to 3 * avgD.

4. *Algorithm Engineering:*

For the scale of N points, if we use Brute-force method to draw lines, then we have to compare every vertex to other N-1 vertices, it will get a $O(N^2)$ time complexity, however, because this method doesn't need other data structure other than an Array, it's space complexity is only O(N).

If we use Cell Method to draw lines, we need a space complexity of $O(2N + 1/R^2) =$ $O(N)$. According to time complexity, we only have to find 5 adjacent cells, if N is big enough, we can assume every cell is full of vertices, then for each vertex, search in one cell takes NR^2 time, search in 5 cells takes $5NR^2$ time. Hence, this algorithm will estimate have a $O(N^2R^2)$ time complexity.

- **In a *square***, $R^2 = (\text{AreaTotal} \times \text{avgD}) / N \times \text{PI})$, so this algorithm is also:

$O(\text{AreaTotal} \times \text{avgD} \times N / \text{PI}) =$ **$O(\text{avgD} \times N)$**

That is, it's ***linear*** for both argument avgD and N.

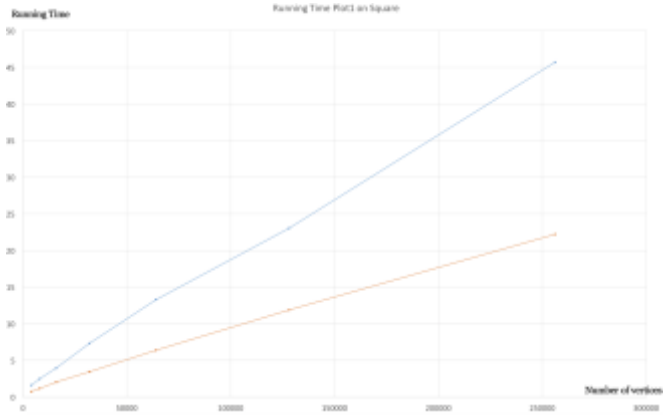
Assume we have an input of avgD and N, it takes time T, then with an input of 2avgD and N, it supposes to take 2T, with an input of avgD and 2N, it supposed to take 2T,with an input of 2avgD and 2N, it supposed to take 4T.

Here's the runtime of this program (all based on first time test):

N	32000	64000	64000	64000	128000
EstimateDegree	64	32	64	128	64
Running Time	6.572s	6.459s	13.237s	25.204s	23.619

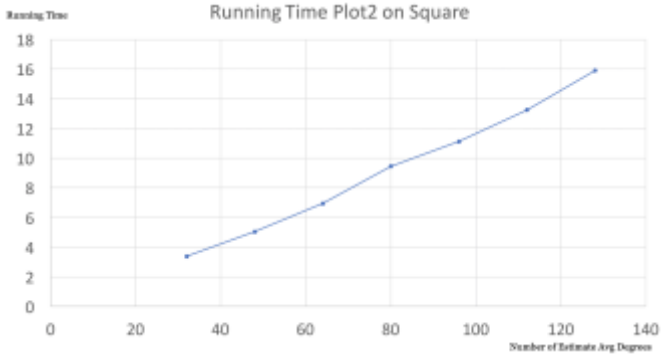
Running Time Graph Plot1:

Blue Line (64 average degrees), Red Line (32 average degrees)



Running Time Graph Plot2:

Number of Vertices (32000)



We can see it's basically linear.

- **In a disk,** $R^2 = (\text{AreaTotal} \times \text{avgD}) / N$, so this algorithm is also:

$O(N \times \text{AreaTotal} \times \text{avgD}) = O(N \times \text{avgD})$

That is also ***linear*** for both argument avgD and N.

Here's the runtime of this program (all based on first time test):

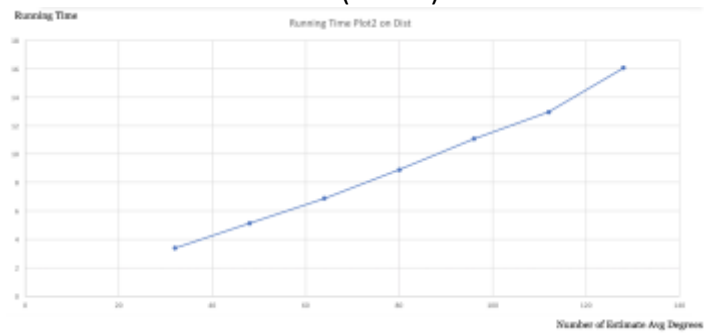
N	8000	8000	16000	32000	64000	64000
EstimateDegree	64	128	64	64	64	128
Running Time	2.224s	5.756	3.876	6.579s	12.75	26.138s

Running Time Graph Plot1:

Blue Line (64 average degrees), Red Line (32 average degrees)

Running Time Graph Plot2:

Number of Vertices (32000)



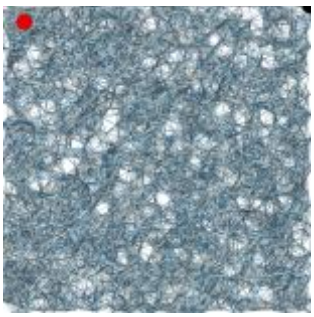
We can see it's also basically linear. Actually, if we double N or double avgD, the runtime T_{new} should be less than $2 \times T_{old}$, because there's four corner area that we don't have vertices, so there's fewer cells to access to when we go through all cells and Cells.contains(cellID) returns a false.

5. *Verification:*

1) Benchmark#1 (N:4000, Estimate Average Degree:32)

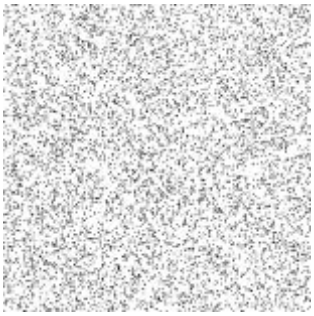
RGG without edges:

RGG with edges:

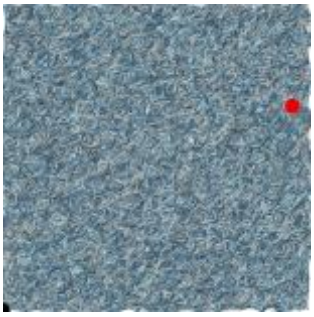


2) Benchmark#2 (N:8000, Estimate Average Degree:64)

RGG without edges:

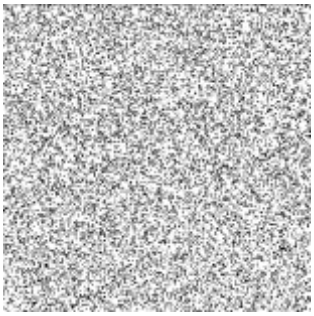


RGG with edges:

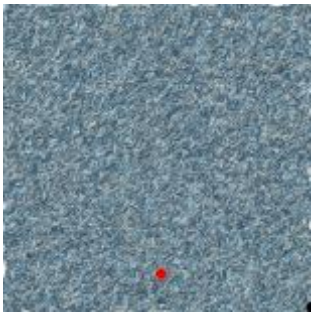


3) Benchmark#3 (N:16000, Estimate Average Degree:64)

RGG without edges:

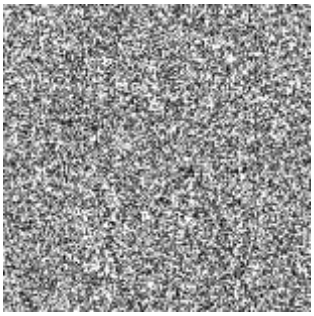


RGG with edges:

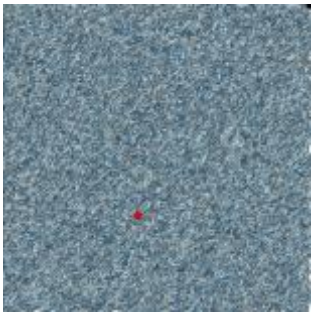


4) Benchmark#4 (N:32000, Estimate Average Degree:64)

RGG without edges:

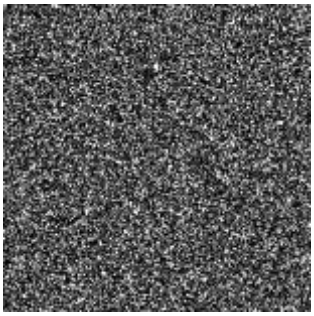


RGG with edges:



5) Benchmark#5 (N:64000, Estimate Average Degree:32)

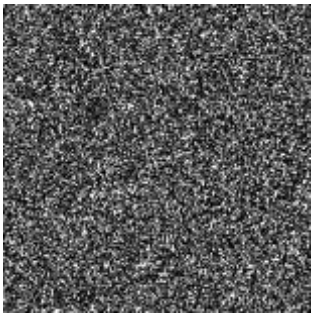
RGG without edges:



RGG with edges

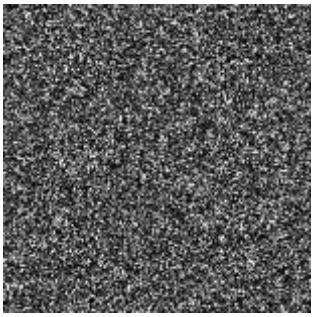
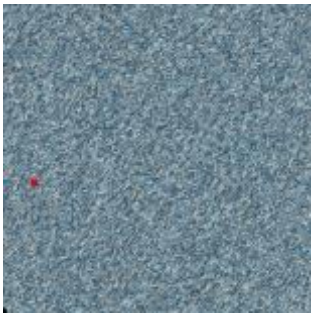
6) Benchmark#6 (N:64000, Estimate Average Degree:64)

RGG without edges: RGG with edges:




7) Benchmark#7 (N:64000, Estimate Average Degree:128)

RGG without edges: RGG with edges:

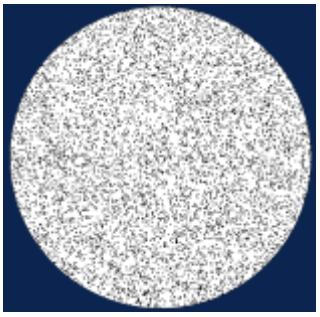
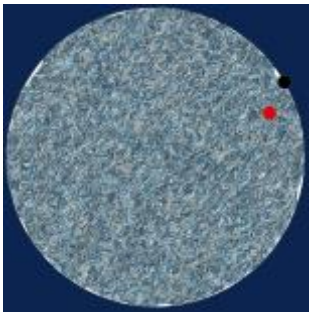



8) Benchmark#8 (N:128000, Estimate Average Degree:64)

RGG without edges: RGG with edges:

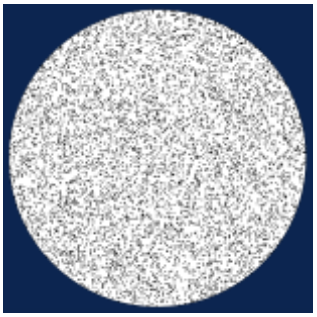
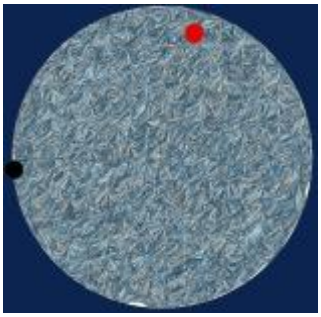

9) Benchmark#9 (N:8000, Estimate Average Degree:64)

RGG without edges: RGG with edges:

10) Benchmark#10 (N:8000, Estimate Average Degree:128)

RGG without edges: RGG with edges:

The red disk's center is the Maximum degree vertex, the Black disk's center is the Minimum degree vertex.

It makes sense that the black disk is at the edge of the square.

II. Part II: Graph Coloring

1. Demand Analyze:

The main purpose of Part II is to separate the whole graph into several independent sets, which is several vertices sets V' who is a subset of V, and each edge in E is incident on at most one vertex in V'.

The way to get independent sets is to Color the graph using Graph-coloring Algorithm,
which is a bijection from Color to V such that for every u, v belongs to V, (u, v) belongs to E,
Color(u) not equals to Color(v). That means we will use Adjacent List of the graph and make sure every pairs of adjacent vertices are colored by different color.

2. Algorithm Description:

First step is to generate a Smallest Last Vertex Ordering. We use SL because vertices with few neighbors ought to be colored as late as possible. The whole procedure is, build a Degree List which can show all vertices in a specific degree, and go through the Degree List from smallest degree vertices to largest degree vertices, and delete vertices from Degree List, after doing this, all vertices adjacent to the deleted one should be moved in Degree List since their degree has to minus one, that means we have to go through the to be

deleted vertex's Adjacent List and move it in correspond Degree List.

We do this delete procedure recursively until one time we find that all remain vertices has the same degree, that is a ***Clique,*** we record the information of the Clique and start the coloring procedure. In coloring procedure, we go through the Smallest Last Vertex Ordering from back to beginning and give each vertex color number, I set an array called CidSet which is a tag array, its size is the color needed when coloring and it's dynamically increased, for each vertex we go through, I set part of values in CidSet to 1 based on index if the vertex's neighbor has this color number, and pick the smallest index in CidSet that value equals to 0 and set the vertex's color number as this index number. In this way, we can guarantee we use the minimum numbers of colors needed to color the graph.

3. Data Structure Design in Part II:

1) Point Class:

Updated the Point Class, record each vertex's index in Array Points as its ***id***, and add a Point type ArrayList ***AdjPoints*** which can contain adjacent points, add an attribute ***c*** means color id of the Point, and add an attribute ***degreeInList*** to record a point's degree in the Degree List (different from its original degree), in this way, a Point's Original degree stays the same, and the final degreeInList value indicates the degree it has when being deleted from the Degree List.

I also add two Point type references ***Next*** and ***Prev*** in Point class, and the ***Degree List*** is built as a ***doubly linked list*** using these references.

```
class Point{
   private int   id;
   private float x;
   private float y;
   private float degree;
   private ArrayList<Point> AdjPoints = new ArrayList<Point>();
   public  int c;
   public  int degreeInList;
   // Reference from former and latter Point in Degree List
   public Point Next;
   public Point Prev;
```

2) Degree List ---- DL:

Define a HashMap implies Degree List, use Integer type degree as key and a Point type as value, for each degree there will be a doubly linked list, and the head of each doubly linked list is stored as Point type value in the HashMap ***DL***, the head of each doubly linked list points to the first element in this degree when there's a point add or delete in DL, it'll add or delete from the head of the doubly linked list.

```
Map<Integer, Point> DL =
  new HashMap<Integer, Point>();
```

```
for(int i = 0; i < Num; i++) {
    int d = (int)Points[i].degree;
    if(DL.containsKey(d)) {
        DL.get(d).Next.Prev = Points[i];
        Points[i].Next = DL.get(d).Next;
        Points[i].Prev = DL.get(d);
        DL.get(d).Next = Points[i];
    } else {
        Point headPoint = new Point(0, 0, 0, -1);
        headPoint.Next = Points[i];
        Points[i].Prev = headPoint;
        Points[i].Next = null;
        DL.put(d, headPoint);
    }
```

3) Smallest Last Ordering ----SLOrder:

Define a Point Type List to store points in the order of what we deleted from the degree list, and the ***Reverse*** go-***through*** of this List is the Smallest Last Order.

```
List<Point> SLOrder = new ArrayList<Point>();
```

4) Average degree after deleted:

Define a LinkedList implies the average degree information after each point is deleted.

```
Deque<Float> DegAfterDel = new LinkedList<Float>();
```

5) A Color ID tag set:

Define a ArrayList implies the colors which a certain point's adjacent points have used.

It's a Tag array.

```
ArrayList<Integer> CidSet = new ArrayList<Integer>();
```

4. *Algorithm Engineering:*

In the first part, we generated the smallest last point order. Since when we go through the whole Degree List to delete each vertex, we have to go through every vertex's Adjacent List to move all the adjacent points in Degree List, so, there's 2×E delete operations needed.
Then, whole time complexity depends on time complexity on delete and add operation from the degree list.

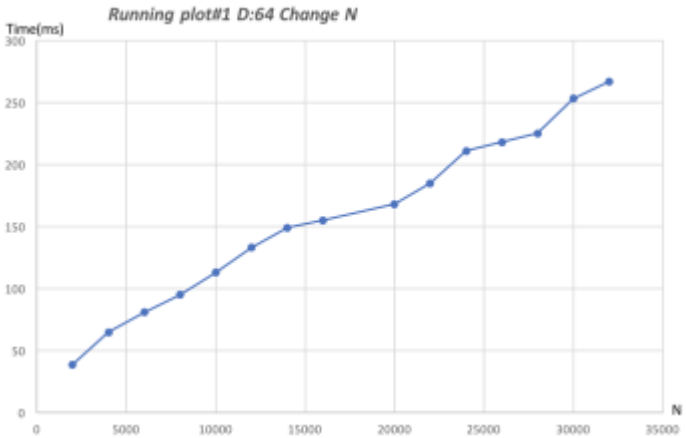
Time complexity of delete operation different depends on the data structure of the Degree List, since Degree List is a HashMap, the way I implement it is to use Doubly Linked List for each degree, since each point has reference point to its former and latter points in Degree List, the operation we need to delete a point is to link its former and latter points, so the time complexity of delete a point in a Doubly Linked List is Θ(1), also, if we want to add a point in Degree List, just add it at the beginning of its degree's doubly linked list. The time complexity is also Θ(1), in this way the time complexity is ***Θ(E), and it's linear***.

In the second part, we have to color the points in the order of Smallest Last Order, the size of the SLOrder array is the number of points, and for each vertex, we have to go through its adjacent list again to check used colors. So, the time complexity of this procedure is also Θ(E).

The result of my design should be ***in linear time***.

Running Time Graph Plot1:

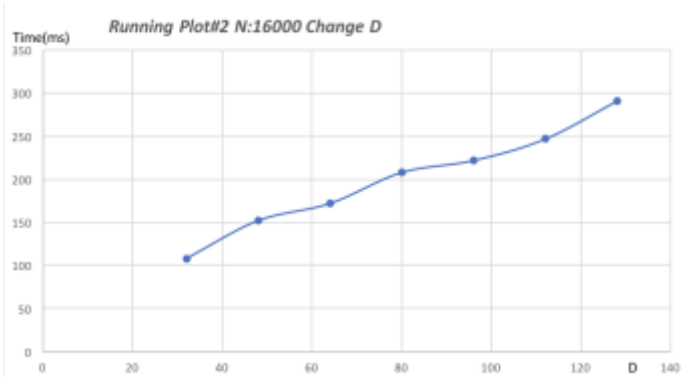
Estimate average Degree = 64, change number of Vertices.



We can see that when number of vertices = 16000, when we change estimate average degree, since Total degree is depended on estimate average degree ***D*** times number of Vertices ***N***. And number of edges ***E*** equals to half of total degree, so when ***D*** stays the same, if running time is linear to ***N***, it is also linear to number of edges E. That is, Θ(E).

Running Time Graph Plot2:

Number of Vertices = 16000, change Estimate Average Degree.



We can see that when number of vertices = 16000, when we change estimate average degree, since Total degree is depended on estimate average degree ***D*** times number of Vertices ***N***. And number of edges ***E*** equals to half of total degree, so when ***N*** stays the same, if running time is linear to D, it is also linear to number of edges ***E***. That is, Θ(E).

5. *Verification Walkthrough:*

I will use a small RGG graph of 20 vertices R = 0.6 in a unit Square, and whenever a vertex is deleted from the Degree List, I will print the information in the Degree list to see which vertex is moved.

At the beginning:

col: 2, R: 0.6050259049522304

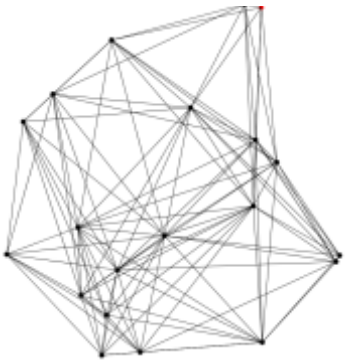
Time used in DrawLine step is: 19ms.
Total degree is: 222, avgDegree is: 11.1
The difference with estimate is: 10.9
The point with biggest degree is: Point#17, it's degree is: 17
The point with smallest degree is: Point#12, it's degree is: 6

Time Info:
Within Coloring step, time used in SLO generating step is: 13ms.
Within Coloring step, time used in coloring step is: 0ms.
Time used in whole Coloring step is: 13ms.
Time used in Both steps is: 76ms.

```
void DrawRemains() {
    for(int i = 0; i <= MaxDegree; i ++){
        if(DL.containsKey(i)) {
            System.out.println("\nDegree is : " + i + ", points in this degree are: ");
            for(Point toPrint = DL.get(i).Next; toPrint != null; toPrint = toPrint.Next) {
                System.out.print("#" + toPrint.id);
            }
        }
    }
}
```

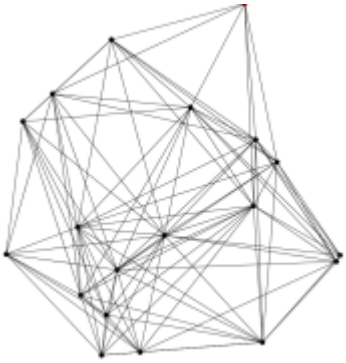
SLO Generating:

Degree is : 6, points in this degree are:
#12
Degree is : 7, points in this degree are:
#10#8
Degree is : 9, points in this degree are:
#18#13#7#3
Degree is : 10, points in this degree are:
#15#4#2#1
Degree is : 12, points in this degree are:
#19#16#6
Degree is : 13, points in this degree are:
#14
Degree is : 14, points in this degree are:
#0
Degree is : 15, points in this degree are:
#11#5
Degree is : 16, points in this degree are:
#9
Degree is : 17, points in this degree are:
#17



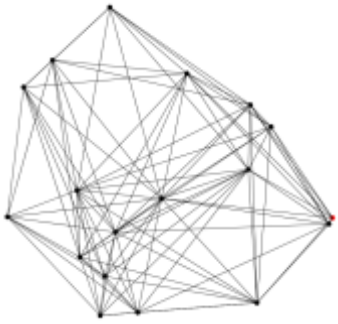
After delete point #12

Degree is : 6, points in this degree are:
#12
Degree is : 7, points in this degree are:
#8
Degree is : 9, points in this degree are:
#4#18#13#7#3
Degree is : 10, points in this degree are:
#15#2#1
Degree is : 11, points in this degree are:
#16
Degree is : 12, points in this degree are:
#14#19#6
Degree is : 13, points in this degree are:
#0
Degree is : 15, points in this degree are:
#9#11#5
Degree is : 17, points in this degree are:
#17



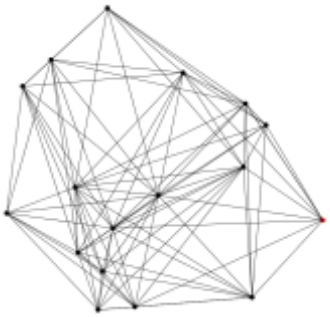
After delete point #10

Degree is : 7, points in this degree are:
#8
Degree is : 8, points in this degree are:
#4
Degree is : 9, points in this degree are:
#1#18#13#7#3
Degree is : 10, points in this degree are:
#16#15#2
Degree is : 11, points in this degree are:
#14
Degree is : 12, points in this degree are:
#0#19#6
Degree is : 14, points in this degree are:
#9
Degree is : 15, points in this degree are:
#11#5
Degree is : 17, points in this degree are:
#17



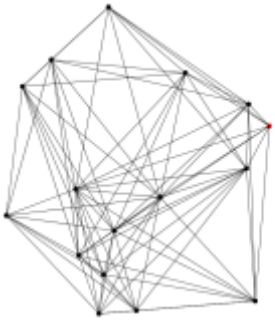
After delete point #8

Degree is : 8, points in this degree are:
#7#4
Degree is : 9, points in this degree are:
#16#1#18#13#3
Degree is : 10, points in this degree are:
#14#15#2
Degree is : 11, points in this degree are:
#19#0
Degree is : 12, points in this degree are:
#6
Degree is : 13, points in this degree are:
#9
Degree is : 15, points in this degree are:
#11#5
Degree is : 16, points in this degree are:
#17



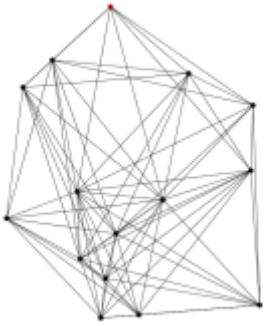
After delete point #7

Degree is : 8, points in this degree are:
#16#4
Degree is : 9, points in this degree are:
#2#14#1#18#13#3
Degree is : 10, points in this degree are:
#19#0#15
Degree is : 12, points in this degree are:
#9#6
Degree is : 14, points in this degree are:
#5
Degree is : 15, points in this degree are:
#17#11



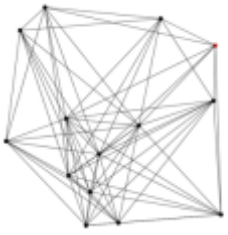
After delete point #16

Degree is : 7, points in this degree are:
#4
Degree is : 8, points in this degree are:
#14
Degree is : 9, points in this degree are:
#19#0#2#1#18#13#3
Degree is : 10, points in this degree are:
#15
Degree is : 11, points in this degree are:
#9
Degree is : 12, points in this degree are:
#6
Degree is : 13, points in this degree are:
#5
Degree is : 14, points in this degree are:
#17#11



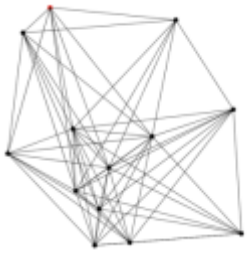
After delete point #4

Degree is : 7, points in this degree are:
#14
Degree is : 8, points in this degree are:
#3#1#0
Degree is : 9, points in this degree are:
#19#2#18#13
Degree is : 10, points in this degree are:
#9#15
Degree is : 12, points in this degree are:
#6
Degree is : 13, points in this degree are:
#17#11#5



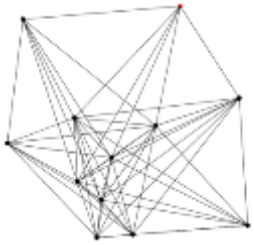
After delete point #14

Degree is : 7, points in this degree are:
#1#0
Degree is : 8, points in this degree are:
#19#3
Degree is : 9, points in this degree are:
#9#2#18#13
Degree is : 10, points in this degree are:
#15
Degree is : 12, points in this degree are:
#17#11#5#6



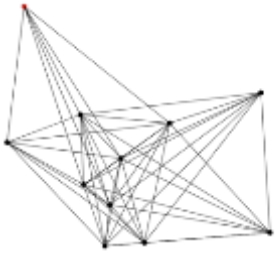
After delete point #1

Degree is : 6, points in this degree are:
#0
Degree is : 7, points in this degree are:
#3
Degree is : 8, points in this degree are:
#13#19
Degree is : 9, points in this degree are:
#9#2#18
Degree is : 10, points in this degree are:
#15
Degree is : 11, points in this degree are:
#17#11#6#5



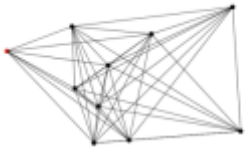
After delete point #0

Degree is : 6, points in this degree are:
#3
Degree is : 8, points in this degree are:
#9#13#19
Degree is : 9, points in this degree are:
#2#18
Degree is : 10, points in this degree are:
#17#11#6#5#15



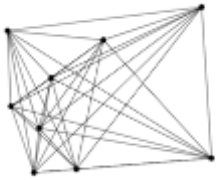
After delete point #3

Degree is : 7, points in this degree are:
#13
Degree is : 8, points in this degree are:
#9#19
Degree is : 9, points in this degree are:
#17#15#11#6#5#2#18



After delete point #13

Degree is : 8, points in this degree are:
#18#17#15#11#6#5#2#9#19



CliqueDegree is :8
CliqueSize is :9
The size of the SLOrder is: 20
SLOrder is: #12 #10 #8 #7 #16 #4 #14 #1 #0 #3 #13 #18 #17 #15 #11 #6 #5 #2 #9 #19
The Max Degree when delete is: 8
Number of colors used is: 9

Coloring:

The point coloring now is:#19, its adjacent points' color are:

Paint #19 color 1

The point coloring now is:#9, its adjacent points' color are:
#19 color is 1
Paint #9 color 2

The point coloring now is:#2, its adjacent points' color are:
#19 color is 1#9 color is 2
Paint #2 color 3

The point coloring now is:#5, its adjacent points' color are:
#2 color is 3#19 color is 1#9 color is 2
Paint #5 color 4

The point coloring now is:#6, its adjacent points' color are:
#2 color is 3#5 color is 4#19 color is 1#9 color is 2
Paint #6 color 5

The point coloring now is:#11, its adjacent points' color are:
#2 color is 3#5 color is 4#6 color is 5#19 color is 1#9 color is 2
Paint #11 color 6

The point coloring now is:#15, its adjacent points' color are:
#2 color is 3#5 color is 4#6 color is 5#11 color is 6#19 color is 1#9 color is 2
Paint #15 color 7

The point coloring now is:#17, its adjacent points' color are:
#2 color is 3#5 color is 4#6 color is 5#11 color is 6#15 color is 7#19 color is 1#9 color is 2
Paint #17 color 8

The point coloring now is:#18, its adjacent points' color are:
#2 color is 3#5 color is 4#6 color is 5#11 color is 6#15 color is 7#17 color is 8#19 color is 1#9 color is 2
Paint #18 color 9

The point coloring now is:#13, its adjacent points' color are:
#2 color is 3#5 color is 4#6 color is 5#11 color is 6#15 color is 7#17 color is 8
Paint #13 color 1

The point coloring now is:#3, its adjacent points' color are:
#5 color is 4#6 color is 5#11 color is 6#13 color is 1#15 color is 7#17 color is 8
Paint #3 color 2

The point coloring now is:#0, its adjacent points' color are:
#3 color is 2#5 color is 4#6 color is 5#11 color is 6#17 color is 8#9 color is 2
Paint #0 color 1

The point coloring now is:#1, its adjacent points' color are:
#0 color is 1#3 color is 2#5 color is 4#6 color is 5#11 color is 6#13 color is 1#17 color is 8
Paint #1 color 3

The point coloring now is:#14, its adjacent points' color are:
#0 color is 1#1 color is 3#9 color is 2#19 color is 1#5 color is 4#11 color is 6#17 color is 8
Paint #14 color 5

The point coloring now is:#4, its adjacent points' color are:
#0 color is 1#1 color is 3#3 color is 2#11 color is 6#17 color is 8#9 color is 2#14 color is 5
Paint #4 color 4

The point coloring now is:#16, its adjacent points' color are:
#0 color is 1#4 color is 4#9 color is 2#14 color is 5#19 color is 1#5 color is 4#11 color is 6#17 color is 8
Paint #16 color 3

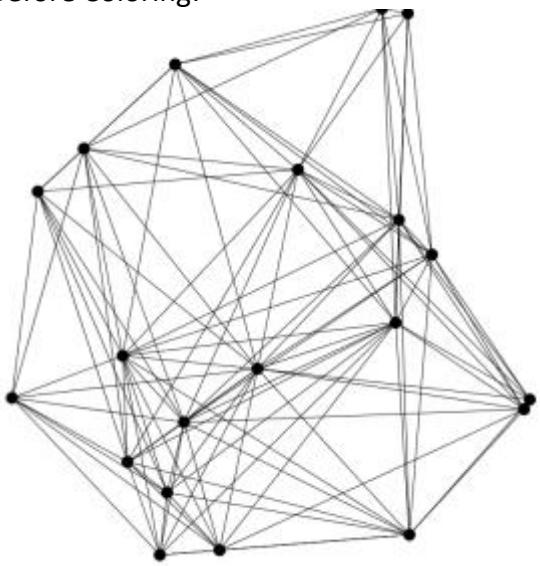
The point coloring now is:#7, its adjacent points' color are:
#0 color is 1#1 color is 3#4 color is 4#9 color is 2#14 color is 5#16 color is 3
Paint #7 color 6

The point coloring now is:#8, its adjacent points' color are:
#0 color is 1#9 color is 2#14 color is 5#16 color is 3#17 color is 8#7 color is 6#9 color is 2
Paint #8 color 4

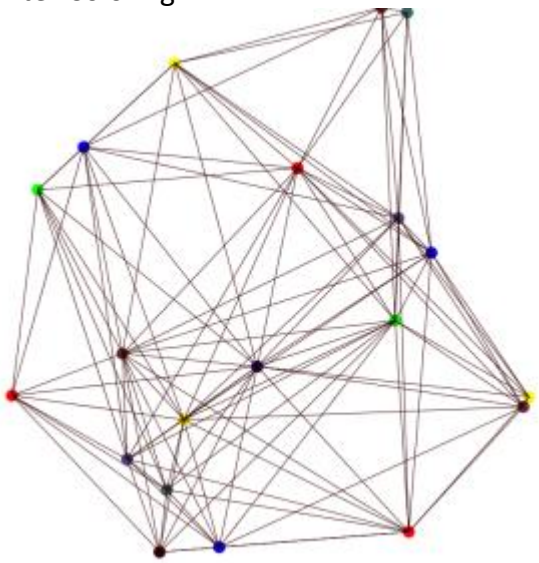
The point coloring now is:#10, its adjacent points' color are:
#0 color is 1#1 color is 3#4 color is 4#9 color is 2#14 color is 5#16 color is 3
Paint #10 color 6

The point coloring now is:#12, its adjacent points' color are:
#0 color is 1#4 color is 4#9 color is 2#10 color is 6#14 color is 5#16 color is 3
Paint #12 color 7

Before Coloring:



After Coloring:



III. Part III: Bipartite Backbone Selection

1. *Demand Analyze:*

The main purpose of this part is to find the backbone of the RGG based on the result of Coloring procedure. It needs us to get the color of 4 biggest color class, and combine each distinct two of them, that gives us 6 pairs of colors and that's 6 networks. We have to modify it, first step is to delete all tails of the network, that is, the vertex who only connect to one vertex in the network, then we have to delete all the minor components. After that, we get a largest connected component. Compare the 6 components and choose 2 with maximum number of edges, and determine that 2 as backbone of the RGG.

2. *Data Structure Design:*

1) Color Distribution:

Integer Type Array ---- Color Distribution, it's defined in Part II, and in Part III we need it to get the top 4 largest color class (Color is represented by Integer).

```
int[] ColorDistribute = new int[Num];
```

2) Backbone Color:

Define an Integer Type Array of size 4, it will store the top 4 largest color's corresponded integer.

```
int[] BackboneColor = new int[4];
```

3) Network ---- BMap:

Let's temporally call all points and edges in a combination of two colors in BackboneColor a *Network*. A *Network* is a candidate Backbone.

I defined a List which each element is a <Point, Integer> Type Map, there's 6 elements in the List and each of them contains all Points in this *Network*. The Integer Type is used later.

```
List<Map<Point, Integer>> BMap
    = new ArrayList<Map<Point, Integer>>();
```

Initiate:

```
// Init all the Networks.
for(int i = 0; i < 6; i ++)
    BMap.add(new HashMap<Point, Integer>());
```

4) Number of Edges:

For each Network we defined, we should store the number of edges in it. So, I defined a Integer Type Array of size 6 called BENum, later we have to choose two networks of top two numbers of edges using this array.

```
int[] BENum = new int[6];
```

5) Point Class:

Update the Point Class, add an Integer Type Array attribute called degreeInBB of size 6 for each Point, if a Point is in a Network, we have to record

its degree in this Network so that later we can delete tails based on this information.

```
class Point{
   private int   id;
   private float x;
   private float y;
   private float degree;
   private ArrayList<Point> AdjPoints = new ArrayList<Point>();
   public  int c;
   public  int degreeInList;
   public  int degreeInBB[] = new int[6];    // Degree in back bone;
   // Reference from former and latter Point in Degree List
   public Point Next;
   public Point Prev;
```

6) Result Information:

Defined two arrays of size 2 to record result information. After modified all 6 networks and choose 2 of them as Backbone, record its ID in Tag[], and record its Size of Points in Size[].

```
int[] Size = new int[2];
int[] Tag = new int[2];
```

3. *Algorithm Description:*

1) Choose 4 largest color class:

We can choose 4 largest color class from ColorDistribution array I defined, and record the ID of these 4 colors in BackboneColor array.

2) Build BMap:

We can add 6 Maps to List BMap, each Map will represent a Network build based on 2 distinct colors from BackboneColor array. For each point, if it's colored by that two color, add it to correspond Network(Map), after all this procedure, size of a Map should equals to sum of number of points of its 2 colors.

3) Delete tails:

For each Map(Network), we need to delete all tail Points ---- all points which has a degree of 1 in its degreeInBB array in this particular map, and after we delete a tail Point, there's a chance that it's adjacent point become a tail Point.

I use a do-while loop to delete tail Points until all the points in this Map has a degree larger than one.

4) Delete minor components:

For each Map, it may consist of more than 1 component, that is, it has minor components, to delete that, I used Breath-First-Search, it starts from a random point in the graph, and use BFS to go through all neighbors it can reach to, after one BFS, I marked all the points it goes through a component. Next time, I used BFS start from a point that haven't go through before, and so on.

This can give us several components, and I choose the one with the largest size of Point, and it should be our largest connected component, and delete all other components.

5) Get Result:

I record the number of edges of the 6 Maps after modified them, and choose 2 of the largest one as the ***Backbones***. Record their information and calculate how many points the points in a Backbone are adjacent with (no repeat), the amount divided by the Total amount of points gives us the Domination Percentage of the RGG, that is, *how many percentage of the whole area a Backbone can dominate. In reality, it means how many percentage of the whole area can all the sensors we put covers.*

4. *Algorithm Engineering:*

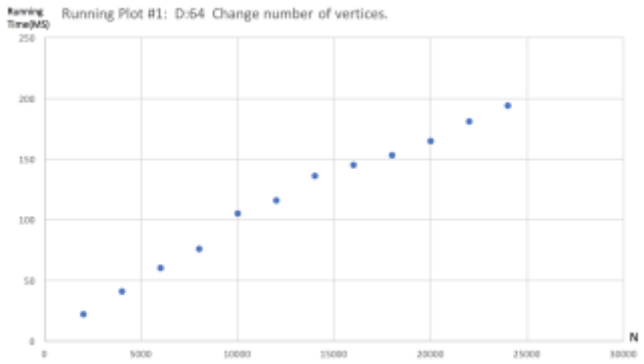
In the first step that I build the 6 Maps (Networks), I go through all points in Points[] one time, so the time complexity is $\Theta(N)$.

Since I used HashMap structure for each Map, all time complexity related to check if a Point is in a Map is $\Theta(1)$. In the delete tails step, for six times I go through all the points in a Map several times until there's no point has a degree of 1, that is $O(k \times n)$, which n represent number of points in a Map, that is the Sum of two number of points whose color is in BackboneColor[].

In the delete minor components step, also, for six times I go through all the points in a Map several times until all points has been visited. That is also $O(i \times n)$. Since all the steps are in order, the time complexity of this Part should be $\Theta(n)$. It should be linear.

Running Time Graph Plot #1:

Set Estimate Average Degree = 64, change number of Vertices.



We can see when number of vertices grow up. The running time grows up in **linear**.

We don't have to set the number of vertices to a constant number and change the estimate average degree, because from the analyze, number of vertices is the only parameter that influence running time.

5. Results:

- Benchmark#1 (N:4000, Estimate Average Degree:32)

The #1 Backbone is build based on points of color #1 and color #3
There's 571 edges in this backbone.
The backbone has a size of 453

The #2 Backbone is build based on points of color #3 and color #2
There's 569 edges in this backbone.
The backbone has a size of 453

The backbone of #1 schema covers 3721 points.
Domination percentage is: 93.025%

The backbone of #2 schema covers 3759 points.
Domination percentage is: 93.975%

- Benchmark#2 (N:8000, Estimate Average Degree:64)

The #1 Backbone is build based on points of color #6 and color #1
There's 761 edges in this backbone.
The backbone has a size of 569

The #2 Backbone is build based on points of color #3 and color #2
There's 760 edges in this backbone.
The backbone has a size of 574

The backbone of #1 schema covers 7780 points.
Domination percentage is: 97.25%

The backbone of #2 schema covers 7818 points.
Domination percentage is: 97.725%

- Benchmark#3 (N:16000, Estimate Average Degree:64)

The #1 Backbone is build based on points of color #2 and color #1
There's 1538 edges in this backbone.
The backbone has a size of 1132

The #2 Backbone is build based on points of color #2 and color #3
There's 1533 edges in this backbone.
The backbone has a size of 1134

The backbone of #1 schema covers 15644 points.
Domination percentage is: 97.775%

The backbone of #2 schema covers 15475 points.
Domination percentage is: 96.71875%

- Benchmark#4 (N:32000, Estimate Average Degree:64)

The #1 Backbone is build based on points of color #1 and color #4
There's 2976 edges in this backbone.
The backbone has a size of 2239

The #2 Backbone is build based on points of color #2 and color #4
There's 2948 edges in this backbone.

The backbone has a size of 2222
The backbone of #1 schema covers 31415 points.
Domination percentage is: 98.171875%

The backbone of #2 schema covers 31318 points.
Domination percentage is: 97.86875%

- Benchmark#5 (N:64000, Estimate Average Degree:32)

The #1 Backbone is build based on points of color #1 and color #3
There's 8955 edges in this backbone.
The backbone has a size of 7127

The #2 Backbone is build based on points of color #1 and color #2
There's 8904 edges in this backbone.
The backbone has a size of 7109

The backbone of #1 schema covers 60005 points.
Domination percentage is: 93.75781%

The backbone of #2 schema covers 59596 points.
Domination percentage is: 93.11875%

- Benchmark#6 (N:64000, Estimate Average Degree:64)

The #1 Backbone is build based on points of color #2 and color #4
There's 5855 edges in this backbone.
The backbone has a size of 4447

The #2 Backbone is build based on points of color #1 and color #4
There's 5849 edges in this backbone.
The backbone has a size of 4422

The backbone of #1 schema covers 62965 points.
Domination percentage is: 98.38281%

The backbone of #2 schema covers 62731 points.
Domination percentage is: 98.01719%

- Benchmark#7 (N:64000, Estimate Average Degree:128)

The #1 Backbone is build based on points of color #1 and color #2
There's 3630 edges in this backbone.
The backbone has a size of 2573

The #2 Backbone is build based on points of color #1 and color #6
There's 3560 edges in this backbone.
The backbone has a size of 2531

The backbone of #1 schema covers 63641 points.
Domination percentage is: 99.439064%

The backbone of #2 schema covers 63380 points.
Domination percentage is: 99.03125%

- Benchmark#8 (N:128000, Estimate Average Degree:64)

The #1 Backbone is build based on points of color #1 and color #3
There's 12105 edges in this backbone.
The backbone has a size of 9039

The #2 Backbone is build based on points of color #1 and color #2
There's 12082 edges in this backbone.
The backbone has a size of 9067

The backbone of #1 schema covers 125541 points.
Domination percentage is: 98.0789%

The backbone of #2 schema covers 126073 points.
Domination percentage is: 98.49453%

- Benchmark#9 (N:8000, Estimate Average Degree:64)

The #1 Backbone is build based on points of color #1 and color #4
There's 740 edges in this backbone.
The backbone has a size of 566

The #2 Backbone is build based on points of color #2 and color #1
There's 738 edges in this backbone.
The backbone has a size of 559

The backbone of #1 schema covers 7887 points.
Domination percentage is: 98.5875%

The backbone of #2 schema covers 7804 points.
Domination percentage is: 97.549995%

- Benchmark#10 (N:8000, Estimate Average Degree:128)

The #1 Backbone is build based on points of color #1 and color #5
There's 452 edges in this backbone.
The backbone has a size of 329

The #2 Backbone is build based on points of color #4 and color #1
There's 450 edges in this backbone.
The backbone has a size of 322

The backbone of #1 schema covers 7963 points.
Domination percentage is: 99.5375%

The backbone of #2 schema covers 7853 points.
Domination percentage is: 98.1625%

From the results, we can see that the backbone can cover over 90% of the area, it should be correct.

IV. Summary Table

- Part I

	#	Number of Vertices	EstAvg Degree	R	Number of Edges	Min. Degree	Max. Degree	Avg. Degree	Running Time
Square	1	4000	32	0.051245	63079	8	53	31.54	0.698s
	2	8000	64	0.050855	248237	14	89	62.06	2.427s
	3	16000	64	0.03596	504894	17	96	63.12	4.249s
	4	32000	64	0.025428	1020334	19	101	63.77	7.105s
	5	64000	32	0.012811	1043897	5	60	32.62	6.325s
	6	64000	64	0.01798	2048227	9	98	64.007	12.990s
	7	64000	128	0.025329	4039461	32	181	126.24	26.133s
	8	128000	64	0.012714	4115999	13	100	64.31	23.580s
Disk	9	8000	64	0.090125	250212	18	93	62.67	2.327s
	10	8000	128	0.126974	488517	48	175	122.14	5.501s

The average degree and the estimated degree is almost equal, there are two situation that difference between average and estimated degree is bigger than 1.2 in square, that's 8000 vertices with 64 average degree, and 640000 vertices with 128 average degree. My guess is 8000 vertices is not enough to maintain an average of 64. That is, in the real world, we need more than 8000 wireless sensors to connected to 64 sensors each.

- Part II

	#	N	EstAvg Degree	Number of Edges	Max Degree when Deleted	Number of colors	Max color class size	Terminal clique size	Running time
Square	1	4000	32	63142	23	24	288	22	57ms
	2	8000	64	248379	38	36	321	30	106ms
	3	16000	64	502696	39	39	635	38	182ms
	4	32000	64	1015851	41	39	1264	33	264ms
	5	64000	32	1046087	27	28	4458	27	311ms
	6	64000	64	2048622	43	43	2505	42	394ms
	7	64000	128	4039916	73	66	1363	59	803ms
	8	128000	64	4116709	43	40	4972	38	687ms
Disk	9	8000	64	250123	39	36	320	28	108ms
	10	8000	128	490419	70	62	172	52	176ms

Number of colors used shouldn't be greater than Max Degree when Deleted plus 1, and should be less than terminal clique size, we can see from the table the result seems to be right.

Max color class size is proportional to (N ÷ EstAvg Degree), because it reflects distribution of each color. The running time is proportional to number of edges.

- Part III

	#	Number of Vertices	EstAvg Degree	E# in Backbone#1	N# in Backbone #1	Domination Percentage #1	E# in Backbone#1	N# in Backbone #1	Domination Percentage #1	Running Time
Square	1	4000	32	571	455	99.03%	569	455	99.98%	0.044s
	2	8000	64	781	569	97.25%	768	574	97.73%	0.076s
	3	16000	64	1538	1132	97.78%	1533	1134	98.72%	0.180s
	4	32000	64	2976	2239	98.17%	2948	2222	97.87%	0.232s
	5	64000	32	8955	7127	99.76%	8904	7108	99.12%	0.370s
	6	64000	64	5855	4447	98.28%	5849	4422	98.02%	0.455s
	7	64000	128	3630	2573	99.44%	3560	2531	99.04%	0.575s
	8	128000	64	12105	9039	98.08%	12082	9067	98.49%	0.776s
Disk	9	8000	64	740	566	98.59%	738	559	97.55%	0.076s
	10	8000	128	452	329	99.54%	450	322	98.16%	0.073s

When Number of Vertices = 8000, no matter what the Estimate Average Degree is, the running time don't change much, so running time is proportional to N, when EstAvg degree grows, domination percentage grows.

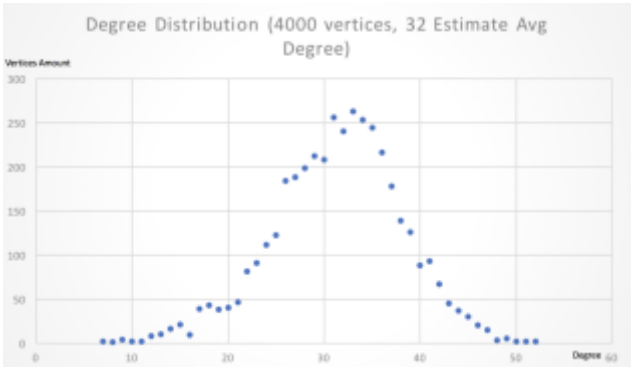
3. Result Summary

I. Benchmark Result Summary and Display

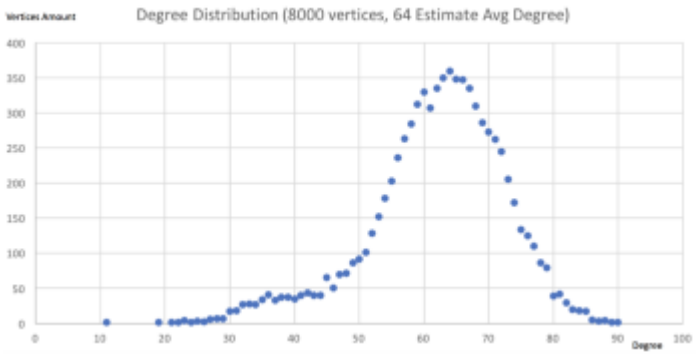
1. Part I

1) Degree Distribution

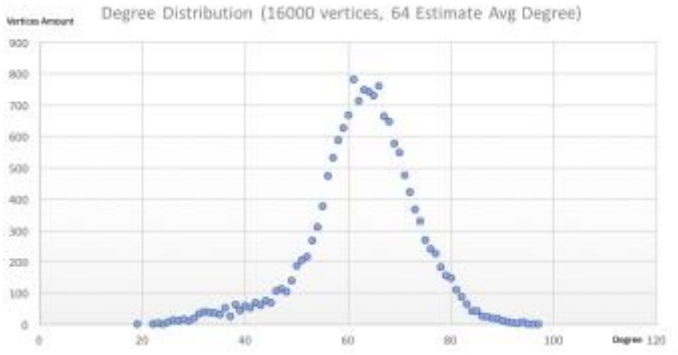
- Benchmark#1 (N:4000, Estimate Average Degree:32)



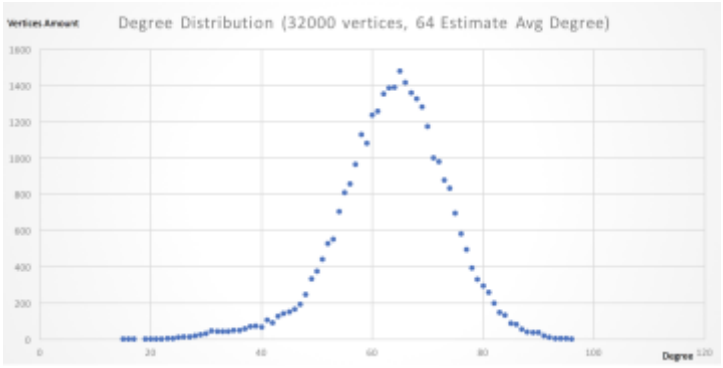
- Benchmark#2 (N:8000, Estimate Average Degree:64)



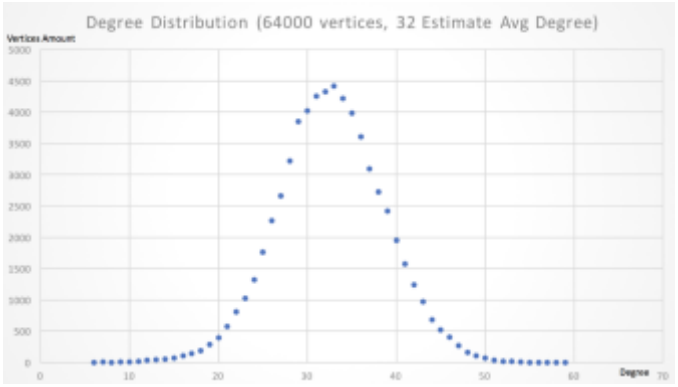
- Benchmark#3 (N:16000, Estimate Average Degree:64)



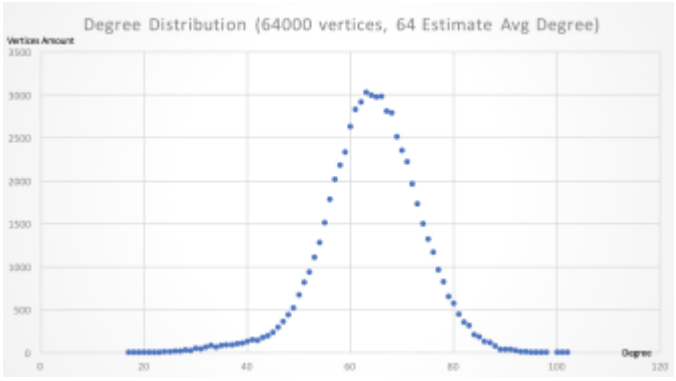
- Benchmark#4 (N:32000, Estimate Average Degree:64)



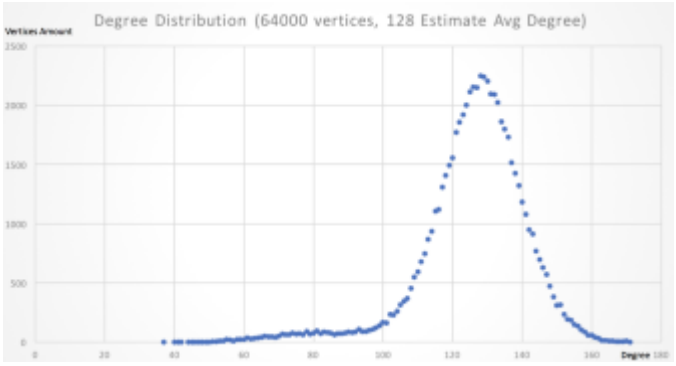
- Benchmark#5 (N:64000, Estimate Average Degree:32)



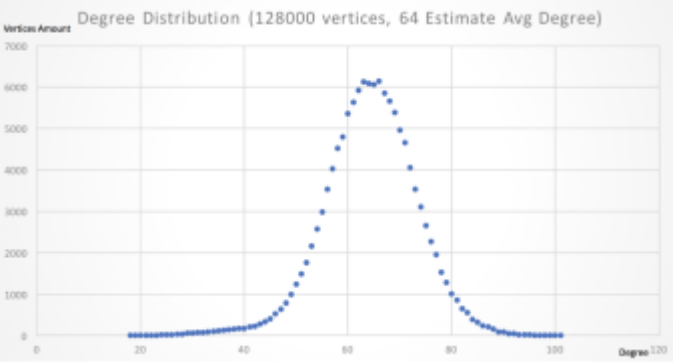
- Benchmark#6 (N:64000, Estimate Average Degree:64)



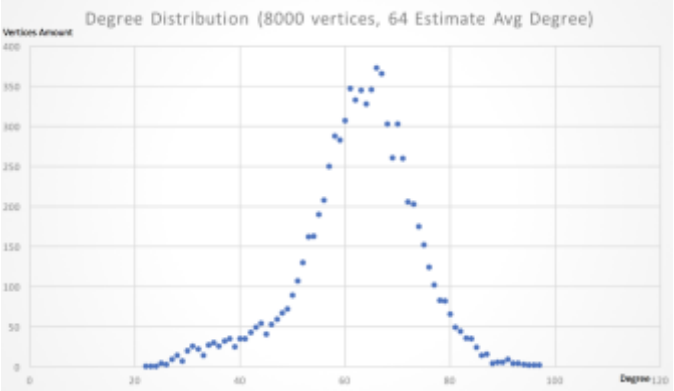
- Benchmark#7 (N:64000, Estimate Average Degree:128)



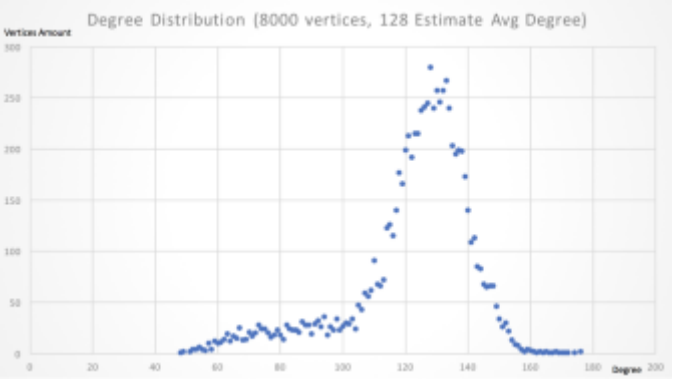
- Benchmark#8 (N:128000, Estimate Average Degree:64)



- Benchmark#9 (N:8000, Estimate Average Degree:64)



- Benchmark#10 (N:8000, Estimate Average Degree:128)

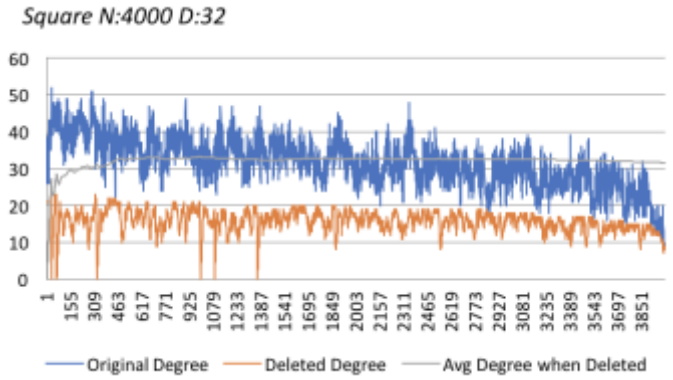


It looks like a normal distribution. The irregular is because there's too many vertices on the edge of the figure, which have small degree.

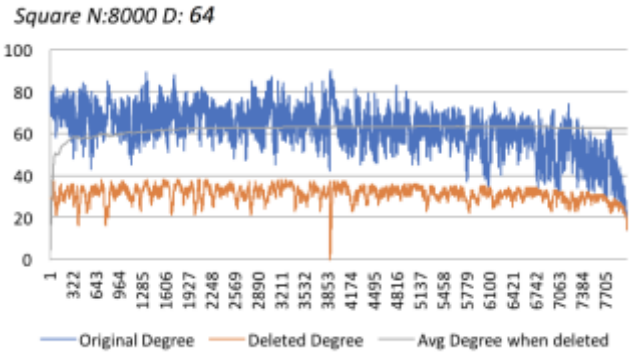
2. *Part II*

2) Sequential Order Plot

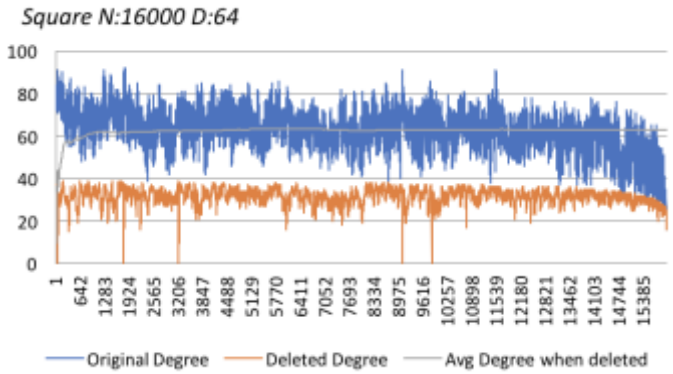
- Benchmark#1 (N:4000, Estimate Average Degree:32)



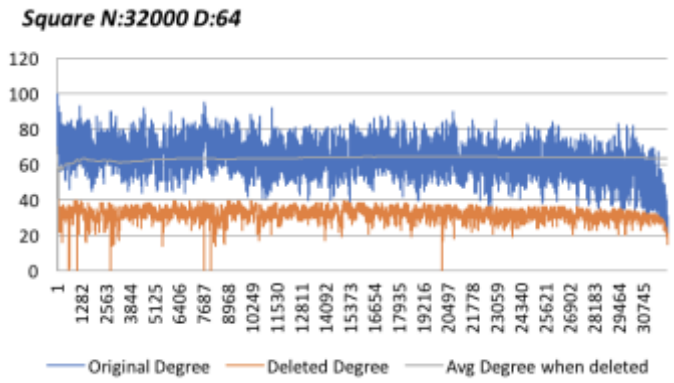
- Benchmark#2 (N:8000, Estimate Average Degree:64)



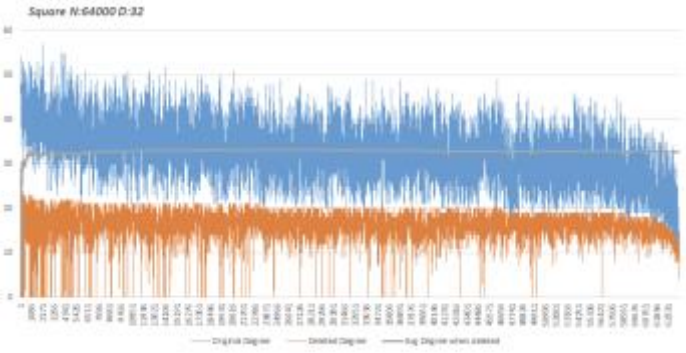
- Benchmark#3 (N:16000, Estimate Average Degree:64)



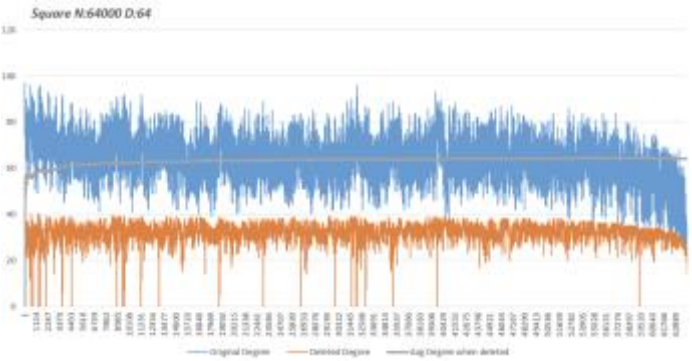
- Benchmark#4 (N:32000, Estimate Average Degree:64)



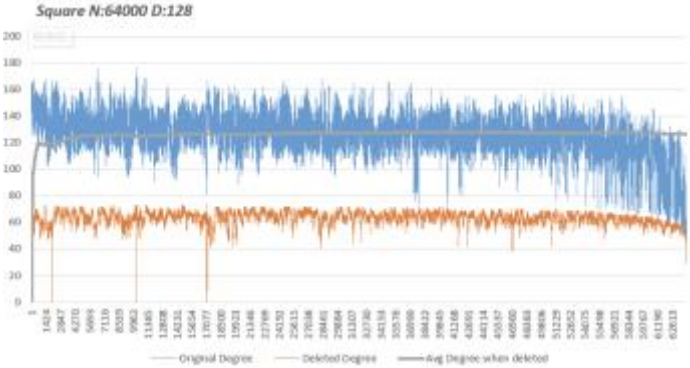
- Benchmark#5 (N:64000, Estimate Average Degree:32)



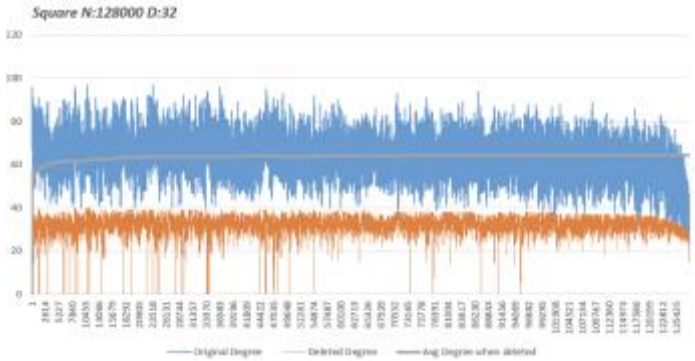
- Benchmark#6 (N:64000, Estimate Average Degree:64)



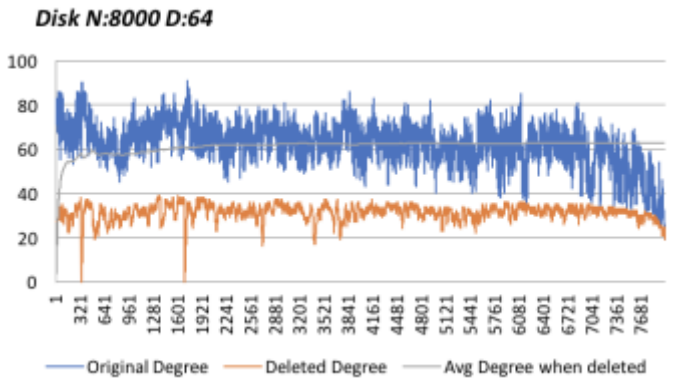
- Benchmark#7 (N:64000, Estimate Average Degree:128)



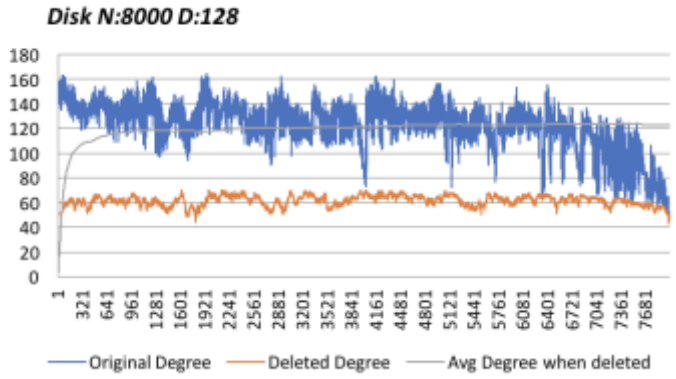
- Benchmark#8 (N:128000, Estimate Average Degree:64)



- Benchmark#9 (N:8000, Estimate Average Degree:64)

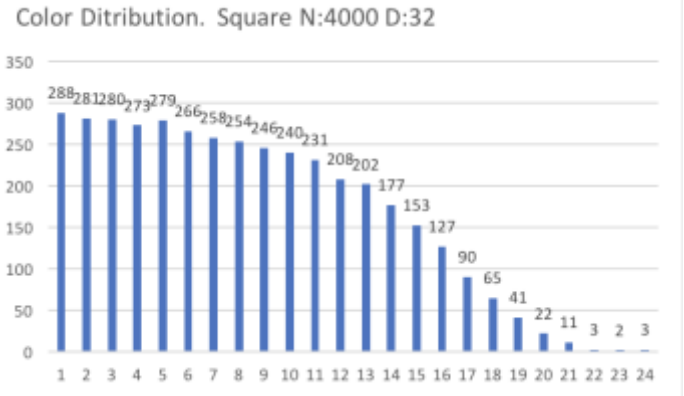


- Benchmark#10 (N:8000, Estimate Average Degree:128)

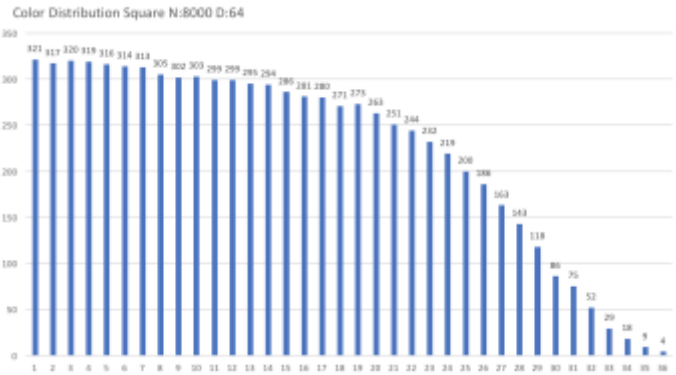


3) Color Class Size Distribution

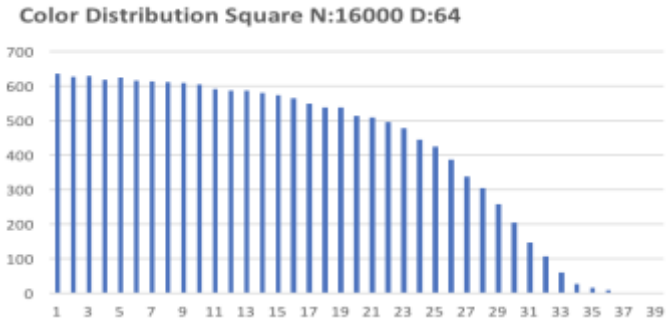
- Benchmark#1 (N:4000, Estimate Average Degree:32)



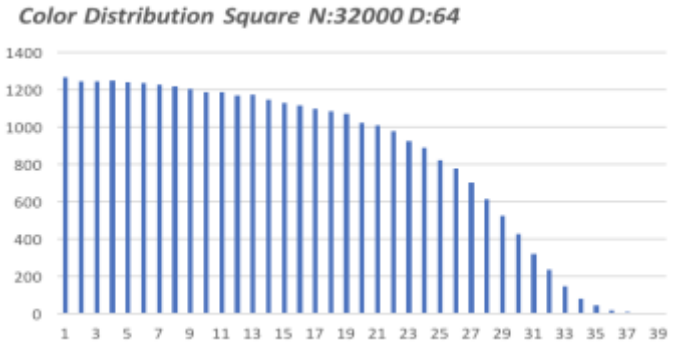
- Benchmark#2 (N:8000, Estimate Average Degree:64)



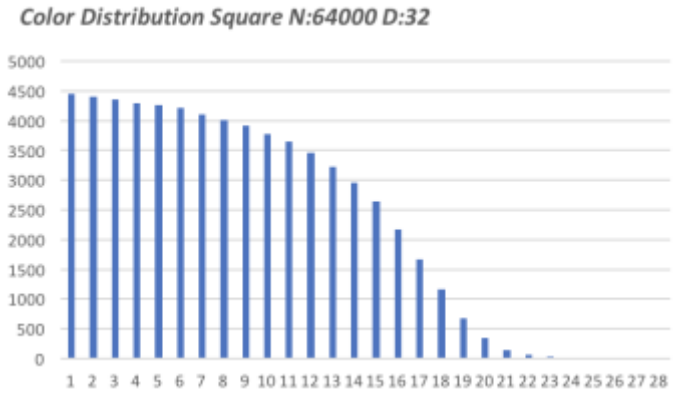
- Benchmark#3 (N:16000, Estimate Average Degree:64)



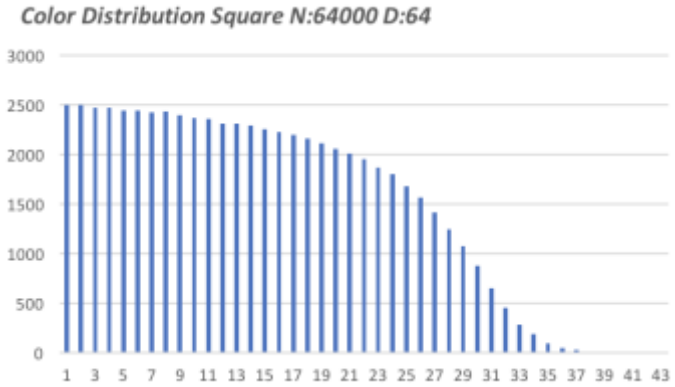
- Benchmark#4 (N:32000, Estimate Average Degree:64)



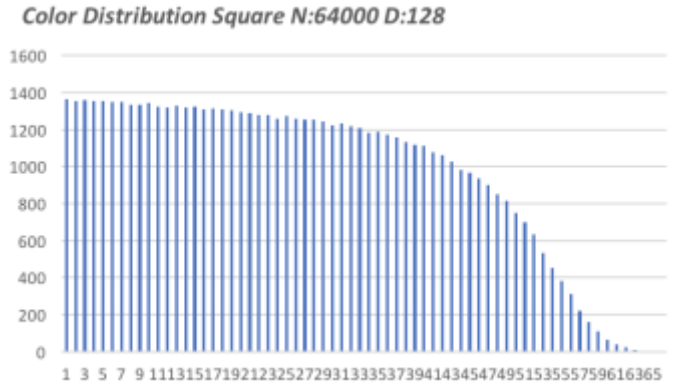
- Benchmark#5 (N:64000, Estimate Average Degree:32)



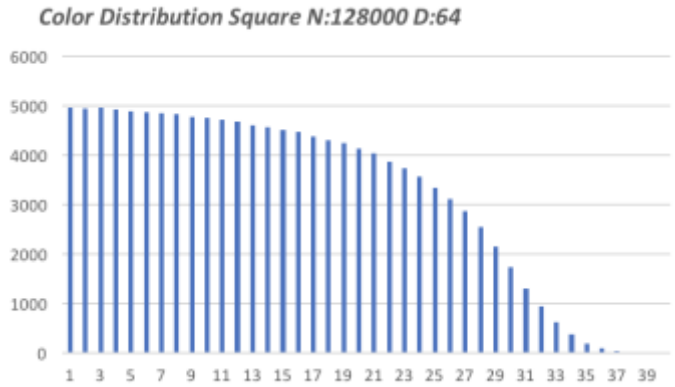
- Benchmark#6 (N:64000, Estimate Average Degree:64)



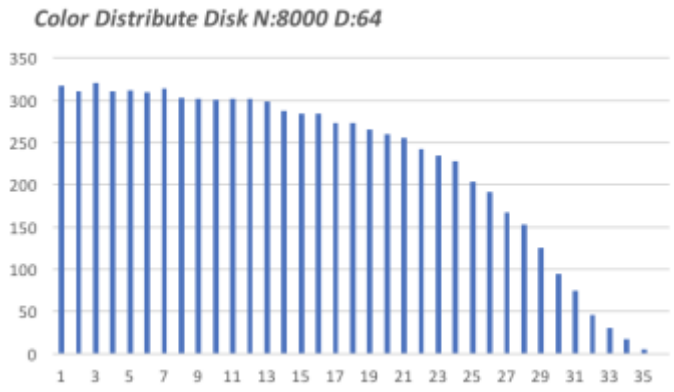
- Benchmark#7 (N:64000, Estimate Average Degree:128)



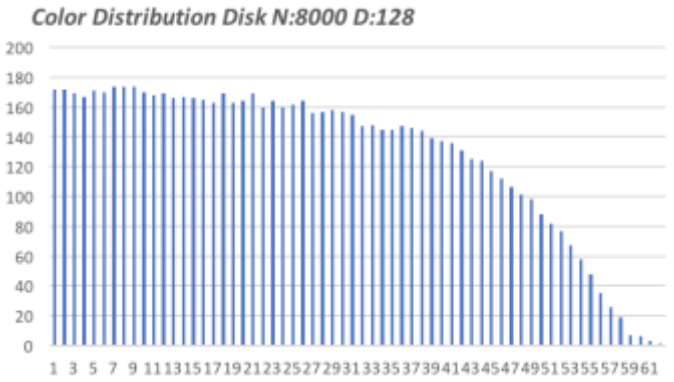
- Benchmark#8 (N:128000, Estimate Average Degree:64)



- Benchmark#9 (N:8000, Estimate Average Degree:64)



- Benchmark#10 (N:8000, Estimate Average Degree:128)



We can see the size of first several colors is larger than latter ones, this make scene. When N = 64000, larger the degree, faster the slope drops, this also make sense because larger the degree, the distribution of points is more centralized, the difference of number of first several colors is smaller, all the points with large degree connect to too many points.

3. *Part III*

- Benchmark#1 (N:4000, Estimate Average Degree:32)

In #1 bipartite partition:
Original size is: 567
After delete tails, modified size is: 453
After delete small components, modified size is: 453

In #2 bipartite partition:
Original size is: 564
After delete tails, modified size is: 426
After delete small components, modified size is: 426

In #3 bipartite partition:
Original size is: 556
After delete tails, modified size is: 381
After delete small components, modified size is: 375

In #4 bipartite partition:
Original size is: 561
After delete tails, modified size is: 453
After delete small components, modified size is: 453

In #5 bipartite partition:
Original size is: 553
After delete tails, modified size is: 364
After delete small components, modified size is: 320

In #6 bipartite partition:
Original size is: 550
After delete tails, modified size is: 426
After delete small components, modified size is: 426

Time used in whole Backbone Creating step is: 44ms.

- Benchmark#2 (N:8000, Estimate Average Degree:64)

In #1 bipartite partition:
Original size is: 640
After delete tails, modified size is: 551
After delete small components, modified size is: 551

In #2 bipartite partition:
Original size is: 639
After delete tails, modified size is: 569
After delete small components, modified size is: 569

In #3 bipartite partition:
Original size is: 639
After delete tails, modified size is: 561
After delete small components, modified size is: 561

In #4 bipartite partition:
Original size is: 637
After delete tails, modified size is: 561
After delete small components, modified size is: 561

In #5 bipartite partition:
Original size is: 637
After delete tails, modified size is: 574
After delete small components, modified size is: 574

In #6 bipartite partition:
Original size is: 636
After delete tails, modified size is: 540
After delete small components, modified size is: 540

- Benchmark#3 (N:16000, Estimate Average Degree:64)

```
In #1 bipartite partition:
Original size is: 1271
After delete tails, modified size is: 1132
After delete small components, modified size is: 1132

In #2 bipartite partition:
Original size is: 1269
After delete tails, modified size is: 1134
After delete small components, modified size is: 1134

In #3 bipartite partition:
Original size is: 1267
After delete tails, modified size is: 1101
After delete small components, modified size is: 1101

In #4 bipartite partition:
Original size is: 1262
After delete tails, modified size is: 1098
After delete small components, modified size is: 1098

In #5 bipartite partition:
Original size is: 1260
After delete tails, modified size is: 1114
After delete small components, modified size is: 1114

In #6 bipartite partition:
Original size is: 1258
After delete tails, modified size is: 1121
After delete small components, modified size is: 1121

Time used in whole Backbone Creating step is: 160ms.
```

- Benchmark#4 (N:32000, Estimate Average Degree:64)

```
In #1 bipartite partition:
Original size is: 2511
After delete tails, modified size is: 2216
After delete small components, modified size is: 2216

In #2 bipartite partition:
Original size is: 2508
After delete tails, modified size is: 2217
After delete small components, modified size is: 2217

In #3 bipartite partition:
Original size is: 2504
After delete tails, modified size is: 2239
After delete small components, modified size is: 2239

In #4 bipartite partition:
Original size is: 2499
After delete tails, modified size is: 2215
After delete small components, modified size is: 2215

In #5 bipartite partition:
Original size is: 2495
After delete tails, modified size is: 2199
After delete small components, modified size is: 2199

In #6 bipartite partition:
Original size is: 2492
After delete tails, modified size is: 2222
After delete small components, modified size is: 2222

Time used in whole Backbone Creating step is: 232ms.
```

- Benchmark#5 (N:64000, Estimate Average Degree:32)

```
In #1 bipartite partition:
Original size is: 8853
After delete tails, modified size is: 7115
After delete small components, modified size is: 7109

In #2 bipartite partition:
Original size is: 8797
After delete tails, modified size is: 7127
After delete small components, modified size is: 7127

In #3 bipartite partition:
Original size is: 8741
After delete tails, modified size is: 7047
After delete small components, modified size is: 7047

In #4 bipartite partition:
Original size is: 8780
After delete tails, modified size is: 6906
After delete small components, modified size is: 6888

In #5 bipartite partition:
Original size is: 8724
After delete tails, modified size is: 6855
After delete small components, modified size is: 6839

In #6 bipartite partition:
Original size is: 8668
After delete tails, modified size is: 6819
After delete small components, modified size is: 6799

Time used in whole Backbone Creating step is: 370ms.
```

- Benchmark#6 (N:64000, Estimate Average Degree:64)

```
In #1 bipartite partition:
Original size is: 4988
After delete tails, modified size is: 4371
After delete small components, modified size is: 4371

In #2 bipartite partition:
Original size is: 4988
After delete tails, modified size is: 4385
After delete small components, modified size is: 4385

In #3 bipartite partition:
Original size is: 4965
After delete tails, modified size is: 4422
After delete small components, modified size is: 4422

In #4 bipartite partition:
Original size is: 4970
After delete tails, modified size is: 4397
After delete small components, modified size is: 4397

In #5 bipartite partition:
Original size is: 4947
After delete tails, modified size is: 4447
After delete small components, modified size is: 4447

In #6 bipartite partition:
Original size is: 4947
After delete tails, modified size is: 4415
After delete small components, modified size is: 4415

Time used in whole Backbone Creating step is: 455ms.
```

- Benchmark#7 (N:64000, Estimate Average Degree:128)

```
In #1 bipartite partition:
Original size is: 2719
After delete tails, modified size is: 2531
After delete small components, modified size is: 2531

In #2 bipartite partition:
Original size is: 2718
After delete tails, modified size is: 2573
After delete small components, modified size is: 2573

In #3 bipartite partition:
Original size is: 2717
After delete tails, modified size is: 2536
After delete small components, modified size is: 2536

In #4 bipartite partition:
Original size is: 2709
After delete tails, modified size is: 2540
After delete small components, modified size is: 2540

In #5 bipartite partition:
Original size is: 2708
After delete tails, modified size is: 2496
After delete small components, modified size is: 2496

In #6 bipartite partition:
Original size is: 2707
After delete tails, modified size is: 2551
After delete small components, modified size is: 2551

Time used in whole Backbone Creating step is: 575ms.
```

- Benchmark#8 (N:128000, Estimate Average Degree:64)

```
In #1 bipartite partition:
Original size is: 9981
After delete tails, modified size is: 9067
After delete small components, modified size is: 9067

In #2 bipartite partition:
Original size is: 9954
After delete tails, modified size is: 9039
After delete small components, modified size is: 9039

In #3 bipartite partition:
Original size is: 9945
After delete tails, modified size is: 8990
After delete small components, modified size is: 8990

In #4 bipartite partition:
Original size is: 9925
After delete tails, modified size is: 8872
After delete small components, modified size is: 8872

In #5 bipartite partition:
Original size is: 9916
After delete tails, modified size is: 8903
After delete small components, modified size is: 8899

In #6 bipartite partition:
Original size is: 9889
After delete tails, modified size is: 8876
After delete small components, modified size is: 8876

Time used in whole Backbone Creating step is: 774ms.
```

- Benchmark#9 (N:8000, Estimate Average Degree:64)

```
In #1 bipartite partition:
Original size is: 634
After delete tails, modified size is: 541
After delete small components, modified size is: 541

In #2 bipartite partition:
Original size is: 633
After delete tails, modified size is: 559
After delete small components, modified size is: 559

In #3 bipartite partition:
Original size is: 633
After delete tails, modified size is: 552
After delete small components, modified size is: 552

In #4 bipartite partition:
Original size is: 631
After delete tails, modified size is: 551
After delete small components, modified size is: 551

In #5 bipartite partition:
Original size is: 631
After delete tails, modified size is: 539
After delete small components, modified size is: 539

In #6 bipartite partition:
Original size is: 630
After delete tails, modified size is: 566
After delete small components, modified size is: 566

Time used in whole Backbone Creating step is: 76ms.
```

- Benchmark#10 (N:8000, Estimate Average Degree:128)

```
In #1 bipartite partition:
Original size is: 351
After delete tails, modified size is: 322
After delete small components, modified size is: 322

In #2 bipartite partition:
Original size is: 350
After delete tails, modified size is: 321
After delete small components, modified size is: 321

In #3 bipartite partition:
Original size is: 350
After delete tails, modified size is: 300
After delete small components, modified size is: 300

In #4 bipartite partition:
Original size is: 347
After delete tails, modified size is: 312
After delete small components, modified size is: 312

In #5 bipartite partition:
Original size is: 347
After delete tails, modified size is: 329
After delete small components, modified size is: 329

In #6 bipartite partition:
Original size is: 346
After delete tails, modified size is: 316
After delete small components, modified size is: 316

Time used in whole Backbone Creating step is: 73ms.
```

4. Performance Summary

Assume Benchmark#11 has 256,000 nodes with an average degree of 64, the number of vertices is double of Benchmark#8, this is the statistic of Benchmark #8.

#	Number of Vertices	EstAvg Degree	R	Number of Edges	Min. Degree	Max. Degree	Avg. Degree	Part I Running Time	Max Degree when Deleted	Number of Colors	Max Color Class Size	Terminal Clique Size	Part II Running Time	Number of Edges in Largest bipartite	Domination Percentage	Part III Running Time
8	128000	64	0.0127	4114168	20	98	64.28	22.766s	40	39	5005	36	0.59s	11943	98.09%	0.588s

Since running time in Part 1 is Θ(avgD × N), running time in Part 1 of Benchmark#11 should double. (45.5s)

Since running time in Part2 is Θ(E), number of edges in Benchmark#11 should be double, running time in Part 2 should also be double. (1.18s)

Since running time in Part3 is Θ(N), running time in Part 3 should be double. (1.166s)

#	Number of Vertices	EstAvg Degree	R	Number of Edges	Min. Degree	Max. Degree	Avg. Degree	Part I Running Time	Max Degree when Deleted	Number of Colors	Max Color Class Size	Terminal Clique Size	Part II Running Time	Number of Edges in Largest bipartite	Domination Percentage	Part III Running Time
8	128000	64	0.0127	4114168	20	98	64.28	22.766s	40	39	5005	36	0.59s	11943	98.09%	0.588s
11	256000	64	0.009	8261279	12	101	64.54	45.437s	43	40	9931	36	0.995s	11943	98.09%	1.209s

We can compare the previous table to see the assumptions are basically right.

Assume Benchmark#12 has 1,000,000,000 nodes with an average degree of 64. Same as Benchmark#11:

1,000,000,000 ÷ 128,000 = 7812.5

Running time in Part I should about equals to (22.766 × 7812.5) = 177859s

Running time in Part II should about equals to (0.59 × 7812.5) = 4609s

Running time in Part III should about equals to (0.588 × 7812.5) = 4594s

	#	Number of Vertices	EstAvg Degree	R	Number of Edges	Min. Degree	Max. Degree	Avg. Degree	Part I Running Time	Max Degree when Deleted	Number of Colors	Max Color Class Size	Terminal Clique Size	Part II Running Time	Number of Edges in Largest bipartite subgraph	Domination Percentage	Part III Running Time
Square	1	4000	32	0.051245	63375	6	56	31.69	0.686s	23	23	288	20	0.055s	568	92.73%	0.034s
	2	8000	64	0.050855	298402	15	90	62.101	2.590s	38	37	319	28	0.136s	716	95.36%	0.082s
	3	16000	64	0.03596	505706	16	94	63.21	4.074s	40	38	634	32	0.173s	1511	98.58%	0.14s
	4	32000	64	0.025428	1018361	16	98	63.65	7.097s	42	40	1259	37	0.272s	3036	98.64%	0.202s
	5	64000	32	0.012811	1043253	5	58	32.602	5.969s	25	25	4443	24	0.289s	9013	93.59%	0.515s
	6	64000	64	0.01798	2049524	17	101	64.04	12.642s	42	41	2493	36	0.435s	6048	98.81%	0.411s
	7	64000	128	0.025329	4036003	27	175	126.13	27.505s	74	66	1358	61	0.674s	3502	99.28%	0.363s
	8	128000	64	0.012714	4114168	20	98	64.28	22.766s	40	39	5005	36	0.59s	11943	98.09%	0.588s
	11	256000	64	0.00899	8261279	12	101	64.54	45.437s	43	40	9931	36	0.995s	11943	98.09%	1.209s
	12	1,000,000,000	64						177859s					4609s			4594s
Disk	9	8000	64	0.090125	250066	17	96	62.52	2.291s	39	38	321	35	0.133s	774	98.46%	0.066s
	10	8000	128	0.126974	488375	34	166	122.09	5.356s	72	63	177	56	0.176s	444	98.76%	0.072s

In this project, I managed to keep the whole procedure executed in linear time, it should give attribute to the data structure I was suggested to use. In Java, HashMap and Doubly Linked List can effectively reduce time complexity for search operation or delete operation. Hence, when it comes to a huge amounts of nodes, my implement can still execute pretty well.

5. Appendices

- ## Source Code

```
import java.util.*;
import java.math.*;
import java.io.*;

color[] styleyou = {#D9C6B0, #314650, #2D4761, #45718C, #B6E1F2};
color[] lineColor = styleyou;

int Num = 256000;
Point [] Points = new Point[Num];
int avgD = 64;
int SqWidth = 700;
int SqHeight = 700;
float PI = 3.14159265;
double R = Math.sqrt(SqWidth*SqHeight*(avgD+1) / (Num*PI));
int cNum = (int)(SqWidth/R) + 1;
//Degree Info
int totalDegree = 0;
float avgDegree = 0;
int MaxDuringDel = 0;
//Cell structure
Map<Integer, ArrayList<Point>> Cells = new HashMap<Integer, ArrayList<Point>>();
//Degree Distribute
int MinDegree = Num;
int MaxDegree = 0;
List<Point> SLOrder = new ArrayList<Point>();
Map<Integer, Point> DL = new HashMap<Integer, Point>();

int CS = 0;
Deque<Float> DegAfterDel = new LinkedList<Float>();

//Color ID set
ArrayList<Integer> CidSet = new ArrayList<Integer>();
int[] ColorDistribute = new int[Num];

//Backbone
int[] BackboneColor = new int[4];
int[] BENum = new int[6];
List<Map<Point, Integer>> BMap = new ArrayList<Map<Point, Integer>>();

int[] Size = new int[2];
int[] Tag = new int[2];

void setup() {
  size(950, 770);
  smooth();
  background(#0C2550);
  //DrawButton();
  DrawSquare();
  noLoop();
}

void draw() {
  System.out.println("col: " + cNum + ", R: " + R/SqWidth + "\n");
  noLoop();
  stroke(0);
  strokeWeight(3);
  long t1 = System.currentTimeMillis();
  DrawDots();
  SquareLine();
  DegreeInfo();
    //DrawDisk(1);
  // Initialize the num of colors as 0.
  CidSet.add(0);  // Initialize the num of colors as 0.
  Coloring();
  CreateBackbone();
```

```
  long t2 = System.currentTimeMillis();
  System.out.println("Time used in the whole procedure is: " + (t2 - t1) + "ms.");
}

void DrawSquare() {
  //bottom,top&left margin of 35px.
  fill(255);
  stroke(0);
  strokeWeight(1);
  rectMode(CENTER);
  int xMin = width - SqWidth - (height-SqHeight)/2;
  int xMax = width - (height-SqHeight)/2;
  int yMin = (height-SqHeight)/2, yMax = (height+SqHeight)/2;

  rect(width - SqWidth/2 - (height-SqHeight)/2 , height/2, SqWidth, SqHeight);


  for(int i = 0; i < Num; i ++){
    //Create a point.
    Points[i] = new Point(random(xMin, xMax), random(yMin, yMax), 0, i);

    //Calculate CellId of a Point
    int x =(int)((Points[i].x - xMin) / R), y = (int)((Points[i].y - yMin) / R);
    int cellId = y * cNum + x;
    //System.out.println(Points[i].x + ", " + Points[i].y + ", " + x + ", " + y + ", CellId: " + cellId);

    //Add point to Cells
    if(Cells.containsKey(cellId)){
      Cells.get(cellId).add(Points[i]);
    } else {
      ArrayList<Point> temp = new ArrayList<Point>();
      temp.add(Points[i]);
      Cells.put(cellId, temp);
    }
  }
}

void DrawDots() {
  for(int i = 0; i < Num; i ++){
    point(Points[i].x, Points[i].y);
    //System.out.println("x: " + Points[i].x + ", y: " + Points[i].y);
  }
}

void SquareLine() {
  long t1 = System.currentTimeMillis();
  strokeWeight(1);

  for(int i = 0; i < (cNum * cNum); i++) {
     if(Cells.containsKey(i)) {
        for(int tag1 = 0; tag1 < Cells.get(i).size(); tag1++) {
           //find in the same cell
           for(int tag2 = 0; tag2 < Cells.get(i).size(); tag2++) {
             if(tag1 != tag2)
               LineUp(Cells.get(i).get(tag1), Cells.get(i).get(tag2), 0.5);
           }
           //find in the below cell
           if(Cells.containsKey(i + cNum)){
             for(int tag2 = 0; tag2 < Cells.get(i + cNum).size(); tag2++) {
                LineUp(Cells.get(i).get(tag1), Cells.get(i + cNum).get(tag2), 1);
             }
           }
           if((i+1)%cNum != 0) {
             //find in the next cell
             if(Cells.containsKey(i + 1)){
               for(int tag2 = 0; tag2 < Cells.get(i + 1).size(); tag2++) {
                  LineUp(Cells.get(i).get(tag1), Cells.get(i + 1).get(tag2), 1);
               }
             }
```

```
            //find in the top of the next cell
            if(Cells.containsKey(i + 1 - cNum)){
              for(int tag2 = 0; tag2 < Cells.get(i + 1 - cNum).size(); tag2++) {
                 LineUp(Cells.get(i).get(tag1), Cells.get(i + 1 - cNum).get(tag2), 1);
              }
            }
            //find in the bottom of the next cell
            if(Cells.containsKey(i + 1 + cNum)){
              for(int tag2 = 0; tag2 < Cells.get(i + 1 + cNum).size(); tag2++) {
                 LineUp(Cells.get(i).get(tag1), Cells.get(i + 1 + cNum).get(tag2), 1);
              }
            }
          }
        }
      }
  }
  long t2 = System.currentTimeMillis();
  System.out.println("Time used in DrawLine step is: " + (t2 - t1) + "ms.");
}

void LineUp(Point p1, Point p2, float flag) {
    //flag and 'if' to avoid duplicate add in the same cell
    if(dist(p1.x, p1.y, p2.x, p2.y) <= R){
      stroke(lineColor[int(random(0,5))]);
      line(p1.x, p1.y, p2.x, p2.y);
      p1.degree = p1.degree + flag;
      p2.degree = p2.degree + flag;
      if( !p1.AdjPoints.contains(p2) ) { p1.AdjPoints.add(p2); }
      if( !p2.AdjPoints.contains(p1) ) { p2.AdjPoints.add(p1); }
    }
}

void DegreeInfo() {
  int MaxTag = 0;
  int MinTag = 0;

  for(int i = 0; i < Num; i++) {
      int d = (int)Points[i].degree;
      /*
        Update Degree List.
        Insert from Beginning.
      */
      if(DL.containsKey(d)) {
          DL.get(d).Next.Prev = Points[i];
          Points[i].Next = DL.get(d).Next;
          Points[i].Prev = DL.get(d);
          DL.get(d).Next = Points[i];
      } else {
          Point headPoint = new Point(0, 0, 0, -1);
          headPoint.Next = Points[i];
          Points[i].Prev = headPoint;
          Points[i].Next = null;
          DL.put(d, headPoint);
      }
      // Set Point's degree in Degree List
      Points[i].setDIL(d);
      // Calculate Degree Info of the whole graph
      totalDegree += d;
      if(d > MaxDegree){
        MaxDegree = d;
        MaxTag = i;
      }
      if(d < MinDegree) {
        MinDegree = d;
        MinTag = i;
      }
  }
  avgDegree = (float)totalDegree / Num;
  float dif = (float)(avgD - avgDegree);
```

```
 System.out.println("Total degree is: " + totalDegree + ", avgDegree is: " + avgDegree +
                    "\nThe difference with estimate is: " + dif);
 System.out.println("The point with biggest degree is: Point#" + MaxTag + ", it's degree is: "+ MaxDegree);
 System.out.println("The point with smallest degree is: Point#" + MinTag + ", it's degree is: "+ MinDegree);

/*
 ellipseMode(CENTER);
 fill(#EA0000, 500);
 ellipse(Points[MaxTag].x, Points[MaxTag].y, (float)R, (float)R);
 fill(#000000, 500);
 ellipse(Points[MinTag].x, Points[MinTag].y, (float)R, (float)R);
 */
}

void CreateSLOList(){
      int CliqueDegree = 0;
      int min = MinDegree;
      int tag = 1;
      int TDegree = totalDegree;
      int n = Num;
      //DrawRemains();
      System.out.println("\n\nStarts here:");

lable:for(int i = MinDegree; i <= MaxDegree && DL.size()>1; i++) {
        //System.out.println("\n\n\nSize of the degree list is: " + DL.size());
        //System.out.println("Now in degree : " + i);
           /*Determin the Max degree when delete.*/
           if(i > MaxDuringDel) {MaxDuringDel = i;}
           tag = 1;
           min = i;
           /*Get every point of this degree*/
           while(DL.containsKey(i) && DL.get(i).Next != null) {
              Point toBeDeleted = DL.get(i).Next;
              toBeDeleted.Prev.Next = toBeDeleted.Next;
              if(toBeDeleted.Next != null)
                 toBeDeleted.Next.Prev = toBeDeleted.Prev;
              if(DL.get(i).Next == null)  DL.remove(i);
              //System.out.println("\nPoint to be deleted is: " + toBeDeleted.id);
              SLOrder.add(toBeDeleted);  // Add a point to SLO list.
              //System.out.println(toBeDeleted.degree);
              for(int k = 0; k < toBeDeleted.AdjPoints.size(); k++) {
                  /*remove every adjpoints from DegreeList and add it to degree-1*/
                 //System.out.println("Point to be move is :" + toBeDeleted.AdjPoints.get(k).id);
                 Point toBeMoved = toBeDeleted.AdjPoints.get(k);

                 // A point that has already been deleted.
                 if(toBeMoved.Next == null && toBeMoved.Prev == null){
                    continue;
                 }
                 int degree = toBeMoved.getDIL();
                 if(DL.containsKey(degree)) {
                   // O(1)
                   toBeMoved.Prev.Next = toBeMoved.Next;
                   if(toBeMoved.Next != null)
                      toBeMoved.Next.Prev = toBeMoved.Prev;
                    else toBeMoved.Prev.Next = null;
                   toBeMoved.setDIL(toBeMoved.getDIL()-1);
                   //System.out.println(degree + " to " + (degree-1));
                   if(DL.containsKey(degree - 1)){
                      toBeMoved.Next = DL.get(degree - 1).Next;
                      toBeMoved.Prev = DL.get(degree - 1);
                      if(DL.get(degree - 1) == null) System.out.println("!!!!!!!!!!!!!!!!!!!!");
                      if(DL.get(degree - 1).Next == null) System.out.println("???????");
                      if(DL.get(degree - 1).Next.Prev == null) System.out.println("++++++++++++");

                      DL.get(degree - 1).Next.Prev = toBeMoved;
                      DL.get(degree - 1).Next = toBeMoved;
                   }
```

```
                else {
                  Point headPoint = new Point(0,0,0,-1);
                  headPoint.Next = toBeMoved;
                  toBeMoved.Next = null;
                  toBeMoved.Prev = headPoint;
                  DL.put(degree-1, headPoint);
                  //System.out.println("New I is : " + (degree - 1));
                }
                if(degree - 1 < min) {
                  //System.out.println("Minimun degree is : " + (degree - 1));
                  min = degree - 1;  // min = Smallest degree in list
                  tag = 0;
                }
              }
              // Remove a degree if when go through a point's adj list a degree becomes empty.
              if(DL.get(degree).Next == null){
                DL.remove(degree);
              }
            }
            n--;
            TDegree -= (toBeDeleted.getDIL() * 2);
            float avg = (float) TDegree / n;
            DegAfterDel.addFirst(avg);

            /*Delete this Point from DL*/
            toBeDeleted.Next = null;
            toBeDeleted.Prev = null;

            //DrawRemains();

            //System.out.println("\nAfter delete point #" + toBeDeleted.id);
            if(tag ==0){
               i = min - 1;
               continue lable;
            }
          }
        // Delete a degree when go over all points in a degree.
        DL.remove(i);
      }
      for(int i = MinDegree; i <= MaxDegree; i++) {
        if(DL.containsKey(i)) {
          CliqueDegree = i;
          break;
        }
      }
      //Remain a clique, degree is CliqueDegree
      System.out.println("\nColoring Info:" + "\nCliqueDegree is :" + CliqueDegree);
      int CliqueSize = CliqueDegree + 1;
      CS = CliqueSize;
      System.out.println("CliqueSize is :" + CliqueSize);
      Point temp = DL.get(CliqueDegree).Next;
      while(temp != null){
        SLOrder.add(temp);
        temp = temp.Next;
      }
      System.out.println("The size of the SLOrder is: " + SLOrder.size());  //Should be Num.
      System.out.println("The Max Degree when delete is: " + MaxDuringDel);
}

void Coloring(){
    long t1 = System.currentTimeMillis();
    CreateSLOList();
    long t2 = System.currentTimeMillis();
    int size = SLOrder.size();
    int tag = 0;

    for(int i = size - 1; i >= 0; i--) {
      tag = 0;
      //System.out.println("\nThe Point coloring now is: #" + SLOrder.get(i).id + ", it's adjacent points' colors are: ");
```

```
        for(int j = 0; j < SLOrder.get(i).AdjPoints.size(); j++) {
            if(SLOrder.get(i).AdjPoints.get(j).getColor() != 0) {
                //System.out.print("#" + SLOrder.get(i).AdjPoints.get(j).id + " color is " + SLOrder.get(i).AdjPoints.get(j).getColor() + ".  ");
                CidSet.set(SLOrder.get(i).AdjPoints.get(j).getColor(), 1);
            }
        }
        // need to give this point a color.
        for(int k = 1; k < CidSet.size(); k++) {
            if(CidSet.get(k) == 0 && tag == 0) {    //find the first position that equals 0.
              SLOrder.get(i).setColor(k);
              //System.out.println("\nChoose color: " + k);
              tag = 1;
            }
            CidSet.set(k, 0);
        }
        if(tag == 0) {
            CidSet.add(0);
            SLOrder.get(i).setColor(CidSet.size() - 1);
            CidSet.set(0, CidSet.size() - 1);
        }
    }

    System.out.println("Number of colors used is: " + (CidSet.size()-1));
    System.out.println("\nTime Info:" + "\nWithin Coloring step, time used in SLO generating step is: " + (t2 - t1) + "ms.");
    long t3 = System.currentTimeMillis();
    System.out.println("Within Coloring step, time used in coloring step is: " + (t3 - t2) + "ms.");
    System.out.println("Time used in whole Coloring step is: " + (t3 - t1) + "ms.");

    for(int i = size - 1; i >= 0; i--) {
        ColorDistribute[SLOrder.get(i).getColor()] ++;
    }

    try{
        String pathName = "/Users/lionel/Desktop/Color.txt";
        File Writename = new File(pathName);
        Writename.createNewFile();
        BufferedWriter out  = new BufferedWriter(new FileWriter(Writename));
        for(int i = 1; i <= CidSet.get(0);i ++){
            String write = Integer.toString(ColorDistribute[i]);
            out.write(write + '\n');
            out.flush();
            //System.out.println("The number of color " + i + " has is: " + ColorDistribute[i]);
        }
        out.close();
    } catch(Exception e) {
            e.printStackTrace();
    }
    //WriteToFile();
}

void WriteToFile(){
    int size = SLOrder.size();
    //Write Original degree to file.
    try {
        String pathName = "/Users/lionel/Desktop/Original.txt";
        File Writename = new File(pathName);
        Writename.createNewFile();
        BufferedWriter out = new BufferedWriter(new FileWriter(Writename));
        for(int i = size - 1; i >= 0; i--) {
            String write = Float.toString(SLOrder.get(i).degree);
            out.write(write + '\n');
            out.flush();
        }
        out.close();
    } catch(Exception e) {
            e.printStackTrace();
    }
    //Write deleted degree to file.
    try {
```

```
        String pathName = "/Users/lionel/Desktop/Deleted.txt";
        File Writename = new File(pathName);
        Writename.createNewFile();
        BufferedWriter out = new BufferedWriter(new FileWriter(Writename));
        for(int i = size - 1; i >= 0; i--) {
            String write = Integer.toString(SLOrder.get(i).getDIL());
            out.write(write + '\n');
            out.flush();
        }
        out.close();
    } catch(Exception e) {
            e.printStackTrace();
    }
    //Write deleted degree to file.
    try {
        String pathName = "/Users/lionel/Desktop/Average.txt";
        File Writename = new File(pathName);
        Writename.createNewFile();
        BufferedWriter out = new BufferedWriter(new FileWriter(Writename));
        out.write("0\n");
        out.flush();
        for(int i = 0; i < CS -1; i++) {
            String write = Integer.toString(i);
            out.write(write + '\n');
            out.flush();
        }
        while(DegAfterDel.size()>0){
            float avg = DegAfterDel.pop();
            String write = Float.toString(avg);
            out.write(write + '\n');
            out.flush();
        }
        out.close();
    } catch(Exception e) {
            e.printStackTrace();
    }
}

void DrawRemains() {
    for(int i = 0; i <= MaxDegree; i ++){
        if(DL.containsKey(i)) {
            System.out.println("\nDegree is : " + i + ", points in this degree are: ");
            for(Point toPrint = DL.get(i).Next; toPrint != null; toPrint = toPrint.Next) {
                System.out.print("#" + toPrint.id);
            }
        }
    }
}

void CreateBackbone() {
    long t1 = System.currentTimeMillis();
    // maxTag: color# with max distribution.
    // Initiate BackboneColor with 4 biggest distribution.
    for(int k = 1, maxSize = 0, maxTag = 0; k <= 4; k++, maxSize = 0, maxTag = 0){
        for(int i = 1; i <= CidSet.get(0); i++) {
            if(ColorDistribute[i] > maxSize) {
                maxSize = ColorDistribute[i];
                maxTag = i;
            }
        }
        ColorDistribute[maxTag] = 0;
        BackboneColor[k-1] = maxTag;
    }

    // Init all the Networks.
    for(int i = 0; i < 6; i ++)
        BMap.add(new HashMap<Point, Integer>());

    // Go through all the point(edges) and find 6 biggest amount of edges.  O(E)
```

```
for(int i = 0; i < Num; i ++) {
   if(Points[i].c == BackboneColor[0]) {
      BMap.get(0).put(Points[i], -1);
      BMap.get(1).put(Points[i], -1);
      BMap.get(2).put(Points[i], -1);
      for(int j = 0; j < Points[i].AdjPoints.size(); j++) {
         if(Points[i].AdjPoints.get(j).c == BackboneColor[0]) {
            Points[i].degreeInBB[0] += 0.5;   Points[i].AdjPoints.get(j).degreeInBB[0] += 0.5;
            Points[i].degreeInBB[1] += 0.5;   Points[i].AdjPoints.get(j).degreeInBB[1] += 0.5;
            Points[i].degreeInBB[2] += 0.5;   Points[i].AdjPoints.get(j).degreeInBB[2] += 0.5;
          }
         if(Points[i].AdjPoints.get(j).c == BackboneColor[1])  {
            Points[i].degreeInBB[0]++;
            Points[i].AdjPoints.get(j).degreeInBB[0] ++;
         }
         if(Points[i].AdjPoints.get(j).c == BackboneColor[2])  {
            Points[i].degreeInBB[1]++;
            Points[i].AdjPoints.get(j).degreeInBB[1]++;
         }
         if(Points[i].AdjPoints.get(j).c == BackboneColor[3])  {
            Points[i].degreeInBB[2]++;
            Points[i].AdjPoints.get(j).degreeInBB[2]++;
         }
      }
   } else if(Points[i].c == BackboneColor[1]) {
      BMap.get(0).put(Points[i], -1);
      BMap.get(3).put(Points[i], -1);
      BMap.get(4).put(Points[i], -1);
      for(int j = 0; j < Points[i].AdjPoints.size(); j++) {
         if(Points[i].AdjPoints.get(j).c == BackboneColor[1]) {
            Points[i].degreeInBB[0] += 0.5;   Points[i].AdjPoints.get(j).degreeInBB[0] += 0.5;
            Points[i].degreeInBB[3] += 0.5;   Points[i].AdjPoints.get(j).degreeInBB[3] += 0.5;
            Points[i].degreeInBB[4] += 0.5;   Points[i].AdjPoints.get(j).degreeInBB[4] += 0.5;
         }
         if(Points[i].AdjPoints.get(j).c == BackboneColor[2]) {
            Points[i].degreeInBB[3]++;
            Points[i].AdjPoints.get(j).degreeInBB[3]++;
         }
         if(Points[i].AdjPoints.get(j).c == BackboneColor[3]) {
            Points[i].degreeInBB[4]++;
            Points[i].AdjPoints.get(j).degreeInBB[4]++;
         }
      }
   } else if(Points[i].c == BackboneColor[2]) {
      BMap.get(1).put(Points[i], -1);
      BMap.get(3).put(Points[i], -1);
      BMap.get(5).put(Points[i], -1);
      for(int j = 0; j < Points[i].AdjPoints.size(); j++) {
         if(Points[i].AdjPoints.get(j).c == BackboneColor[2]) {
            Points[i].degreeInBB[1] += 0.5;   Points[i].AdjPoints.get(j).degreeInBB[1] += 0.5;
            Points[i].degreeInBB[3] += 0.5;   Points[i].AdjPoints.get(j).degreeInBB[3] += 0.5;
            Points[i].degreeInBB[5] += 0.5;   Points[i].AdjPoints.get(j).degreeInBB[5] += 0.5;
         }
         if(Points[i].AdjPoints.get(j).c == BackboneColor[3]) {
            Points[i].degreeInBB[5]++;
            Points[i].AdjPoints.get(j).degreeInBB[5]++;
         }
      }
   } else if(Points[i].c == BackboneColor[3]) {
      BMap.get(2).put(Points[i], -1);
      BMap.get(4).put(Points[i], -1);
      BMap.get(5).put(Points[i], -1);
      for(int j = 0; j < Points[i].AdjPoints.size(); j++) {
         if(Points[i].AdjPoints.get(j).c == BackboneColor[3]) {
            Points[i].degreeInBB[2] += 0.5;   Points[i].AdjPoints.get(j).degreeInBB[2] += 0.5;
            Points[i].degreeInBB[4] += 0.5;   Points[i].AdjPoints.get(j).degreeInBB[4] += 0.5;
            Points[i].degreeInBB[5] += 0.5;   Points[i].AdjPoints.get(j).degreeInBB[5] += 0.5;
         }
      }
```

```
        }
    }
    // Delete tails and not-major components.
    Modify();
    // Determine the two backbones.
    BackboneJudge();
    // Calculate the domination percentage of the two backbone.
    Domination();
    long t2 = System.currentTimeMillis();
    System.out.println("Time used in whole Backbone Creating step is: " + (t2 - t1) + "ms.");
}

void Modify(){
  for(int i = 0; i <= 5; i ++) {
      int flag = 0;
      int count = 0;
      // Delete tails.
      //System.out.println("In #" + (i+1) + " bipartite partition:");
      //System.out.println("Original size is: " + BMap.get(i).size());
      do {
          flag = 0;
          Set<Point> Set1 = new HashSet<Point>(BMap.get(i).keySet());
          for(Point P : Set1) {
              if(P.degreeInBB[i] <= 1){
                  // The one it adjacent with degree - 1.
                  for(Point Q : P.AdjPoints) {
                      if(BMap.get(i).containsKey(Q)) {
                          Q.degreeInBB[i]--;
                      }
                  }
                  // Remove this point.
                  BMap.get(i).remove(P);
                  flag = 1;
              }
          }
      } while(flag == 1);
      //System.out.println("After delete tails, modified size is: " + BMap.get(i).size());

      // Delete other components.  BFS.   <Point, Integer> Integer: -1: white  0: grey  tag: black
      int tag = 0;
      Point start = new Point(0,0,0,0);
      List<Integer> componentSize = new ArrayList<Integer>();
      componentSize.add(tag, 0);
      Deque<Point> BFSQueue = new LinkedList<Point>();
      do {
          flag = 0;
          tag++;
          // Size of component #'tag'.
          componentSize.add(tag, 0);
          componentSize.set(0, componentSize.get(0) + 1);  // How many components are there.
          Set<Point> Set2 = new HashSet<Point>(BMap.get(i).keySet());
          for(Point P : Set2) {
              if(BMap.get(i).get(P) == -1) {
                  flag = 1;
                  start = P;
                  break;
              }
          }
          if (flag == 1) {
              BMap.get(i).put(start, 0);
              BFSQueue.push(start);
              while(BFSQueue.size() > 0) {
                  Point temp = BFSQueue.pop();
                  for(Point Neighbor : temp.AdjPoints) {
                      if(BMap.get(i).containsKey(Neighbor)) {
                          if(BMap.get(i).get(Neighbor) == -1) {
                              BMap.get(i).put(Neighbor, 0);
                              BFSQueue.push(Neighbor);
                          }
```

```
                }
            }
            BMap.get(i).put(temp, tag);
            componentSize.set(tag, componentSize.get(tag) + 1);
          }
        }
      } while(flag == 1);

      int maxSize = 0, maxTag = 0;
      for(int h = 1; h < componentSize.get(0); h ++) {
        if(componentSize.get(h) > maxSize) {
          maxSize = componentSize.get(h);
          maxTag = h;
        }
        //System.out.println("Component #" + h + " have a size of " + componentSize.get(h));
      }
      count = 0;
      Set<Point> Set3 = new HashSet<Point>(BMap.get(i).keySet());
      for(Point P : Set3) {
        if(BMap.get(i).get(P) != maxTag)
          BMap.get(i).remove(P);
        else
          count += P.degreeInBB[i];
      }
      //System.out.println("After delete small components, modified size is: " + BMap.get(i).size() + "\n");
      BENum[i] = count/2;
    }
}

void BackboneJudge() {
    // Get top 2 as backbones.
    for(int h = 0, max = 0; h <= 1; h++, max = 0){
      for(int i = 0; i < 6; i ++) {
        //System.out.println("Number of Edges in box# " + i + " is: " + BENum[i]);
        if(BENum[i] > max) {
          max = BENum[i];
          Size[h] = BENum[i];
          Tag[h] = i;
        }
      }
      BENum[Tag[h]] = 0;
    }
    BENum[Tag[0]] = Size[0];
    BENum[Tag[1]] = Size[1];

    for(int h = 0; h <= 1; h++){
      System.out.print("The #" + (h+1) + " Backbone is build based on points of color  #");
      switch(Tag[h]) {
        case 0:
          System.out.println(BackboneColor[0] + " and color #" + BackboneColor[1]);
          System.out.println("There's " + BENum[0] + " edges in this backbone.");
          break;
        case 1:
          System.out.println(BackboneColor[0] + " and color #" + BackboneColor[2]);
          System.out.println("There's " + BENum[1] + " edges in this backbone.");
          break;
        case 2:
          System.out.println(BackboneColor[0] + " and color #" + BackboneColor[3]);
          System.out.println("There's " + BENum[2] + " edges in this backbone.");
          break;
        case 3:
          System.out.println(BackboneColor[1] + " and color #" + BackboneColor[2]);
          System.out.println("There's " + BENum[3] + " edges in this backbone.");
          break;
        case 4:
          System.out.println(BackboneColor[1] + " and color #" + BackboneColor[3]);
          System.out.println("There's " + BENum[4] + " edges in this backbone.");
          break;
        case 5:
```

```
            System.out.println(BackboneColor[2] + " and color #" + BackboneColor[3]);
            System.out.println("There's " + BENum[5] + " edges in this backbone.");
            break;
        }
        System.out.println("The backbone has a size of " + BMap.get(Tag[h]).size());
    }
}

void Domination() {
    float percent;
    for(int h = 0; h <=1; h++) {
        System.out.print("The backbone of #" + (h+1) + " schema ");
        Set<Point> set0 = new HashSet<Point>(BMap.get(Tag[h]).keySet());
        for(Point P : set0)
            for(Point Q : P.AdjPoints)
                if (!BMap.get(Tag[h]).containsKey(Q))
                    BMap.get(Tag[h]).put(Q, -2);
        System.out.println(" covers " + BMap.get(Tag[h]).size() + " points.");
        percent = (float)BMap.get(Tag[h]).size()/ (float)Num;
        System.out.println("Domination percentage is: " + percent*100 + "%");
    }
}

class Point{
   private int   id;
   private float x;
   private float y;
   private float degree;
   private ArrayList<Point> AdjPoints = new ArrayList<Point>();
   public  int c;
   public  int degreeInList;
   public  int degreeInBB[] = new int[6];    // Degree in back bone;
   // Reference from former and latter Point in Degree List
   public Point Next;
   public Point Prev;

   Point(float X, float Y, float d, int ID) {
     this.id = ID;
     x = X;
     y = Y;
     degree = d;
     c = 0;  //  0 represent have no color
     degreeInList = 0;
     Prev = null;
     Next = null;
   }

   public int getColor() {
      return c;
   }

   public void setColor(int C) {
      this.c = C;
   }

   public int getDIL() {
      return degreeInList;
   }

   public void setDIL(int degree) {
      this.degreeInList = degree;
   }

}
```