CS7350 Graph Coloring Analysis Project
Final Report

Name: Bingying Liang
ID: 48999397

May 3 2023

This project looks at implementing an algorithm in multiple ways to solve a problem, analyzing the algorithms' implementations for running time, and testing and your implementations to show they match your analysis. The project also looks at testing to determine how well the multiple implementations solve the problem.

The particular problem for the project is scheduling via graph coloring. For the first part of the project, different conflict graphs which can represent real world problems will be created and saved in files. For the second part of the problem, carious coloring orders will be implemented and analyzed for runtime efficiency and coloring efficiency. These efficiencies are intimately related to the data structures used for both the input graph data and the intermediate data necessary for the ordering and coloring algorithms

1 Computing Environment

OS: macOS 13.3.1 22E261 arm64
IntelliJ IDEA 2022.2.3 (Ultimate Edition)
Language: Java

2 Part One

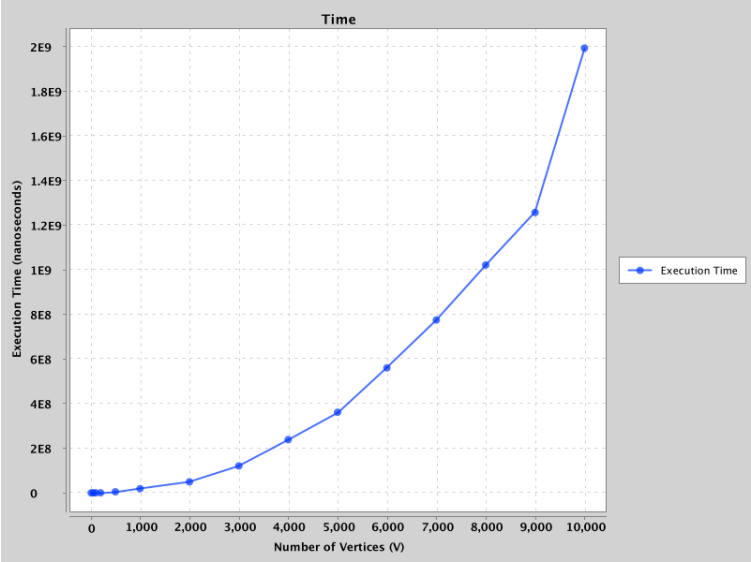
2.1 A description and analysis of the algorithms for generating the conflict graphs

2.1.1 Asymptoric

1. COMPLETE

   The complete graph has an edge from every vertex to other vertex. This result in every vertex has exactly $n-1$ neighbors.[1]

```java
static class Complete{
    static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
        am.get(s).add(d);
        am.get(d).add(s);
    }

    static void allEdge(ArrayList<ArrayList<Integer>> am, int V) {
        for (int i = 0; i < V; i++) {
            for (int j = i + 1; j < V; j++) {
                addEdge(am, i, j);
            }
        }
    }
}
```



Create a chart showing the running times for various values of "v"

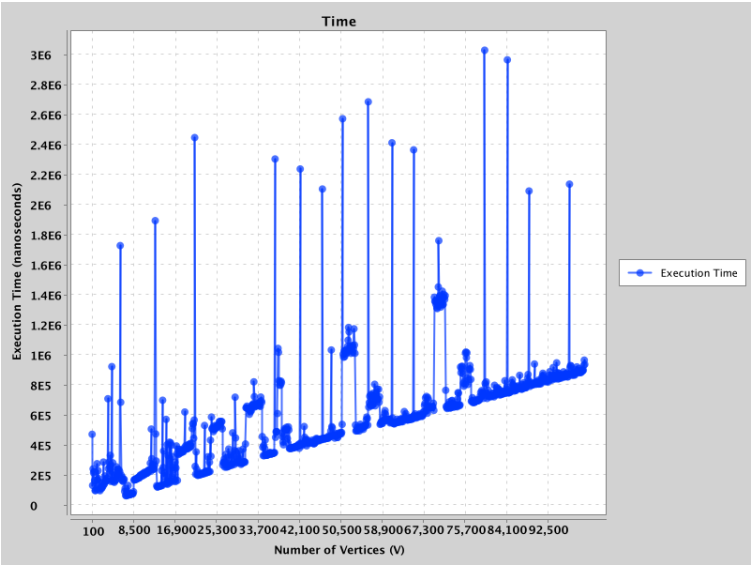
v	times(ns)
10	294583
50	434166
100	608666
200	836625
500	5381125
1000	19269750
2000	49767459
3000	121378208
4000	239193292
5000	361908791
6000	560953084
7000	775675041
8000	1022001459
9000	1257583417
10000	1994619792

2. CYCLE The cycle graph is formed of a single cycle that contains every vertex in the graph. This is formed by iterating over every vertex and connecting it to the next one. Finally, the las vertex is connected back to the start.

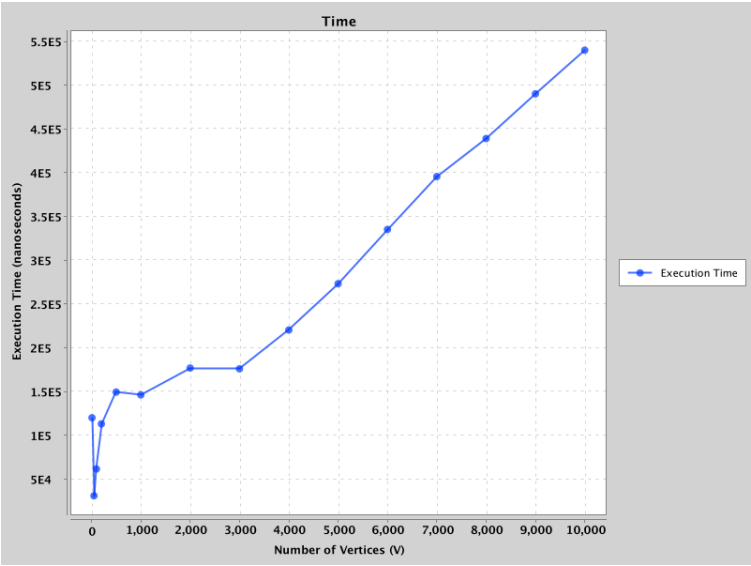
```
static class Cycle {
    static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
        am.get(s).add(d);
        am.get(d).add(s);
    }
    static void allEdgeCycle(ArrayList<ArrayList<Integer>> am, int V) {
        //long startTime = System.nanoTime();
        for (int i = 0; i < V; i++) {
            if (i == V - 1) {
                addEdge(am, i, 0);    // head -- tail
            } else {
                addEdge(am, i, i + 1);
            }
```

```
14                  }
15              }
16      }
17
```

The running time: $\Theta(V)$.
Use the data from $V = (100, 200, 300, ..., 100000)$



Use the data from [10, 50, 100, 200, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]



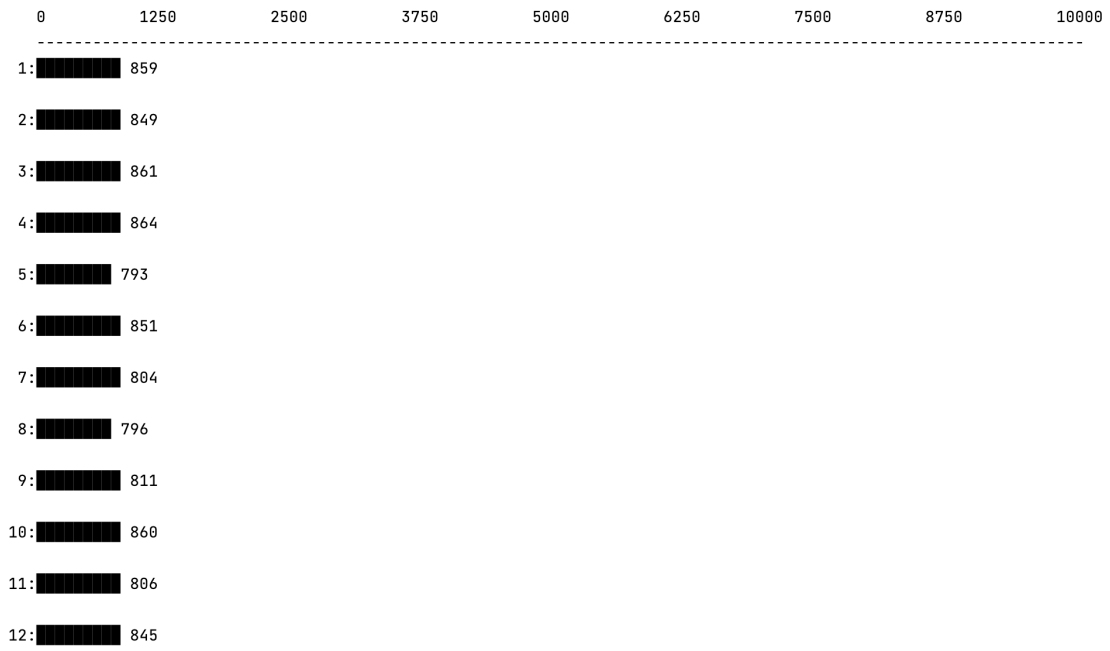
Create a chart showing the running times for various values of "v"

v	times(ns)
50	31083
100	61709
200	113250
500	149541
1000	146334
2000	177000
3000	176125
4000	220459
5000	273375
6000	335333
7000	395708
8000	439167
9000	490250
10000	540084

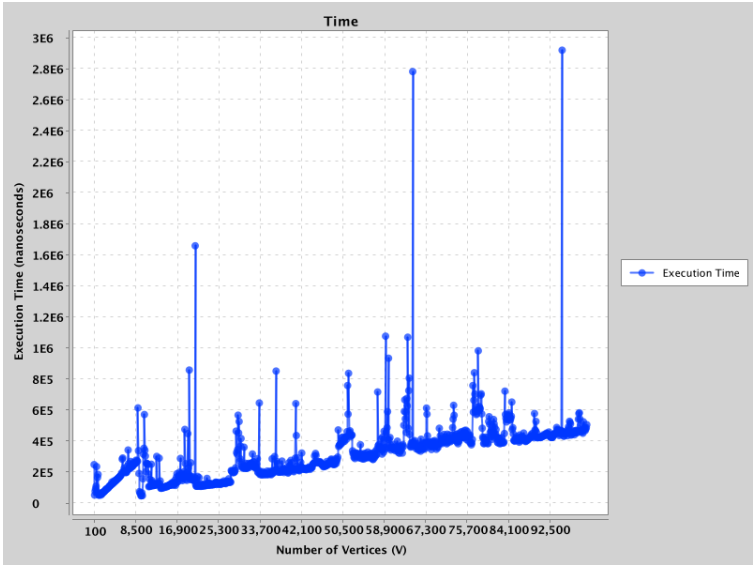
3. UNIFORM

```
static class Uniform {

    static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
        am.get(s).add(d);
        am.get(d).add(s);
    }

    // If you provide an integer parameter to "nextInt",
    // it will return an integer from a uniform distribution
    // between 0 and one less than the parameter.
    static void uniformRandom(ArrayList<ArrayList<Integer>> am, int v, int e) {
        //Random rand = new Random();
        while (e > 0) {
            int source = (int) (Math.random() * v); // Printing the random number between [0,v-1]
            int dest = (int) (Math.random() * v);
            // edge exit?
            if (source == dest || am.get(source).contains(dest)) {
                continue;
            } else {
                addEdge(am, source, dest);
                e--;
            }
        }
    }
}
```

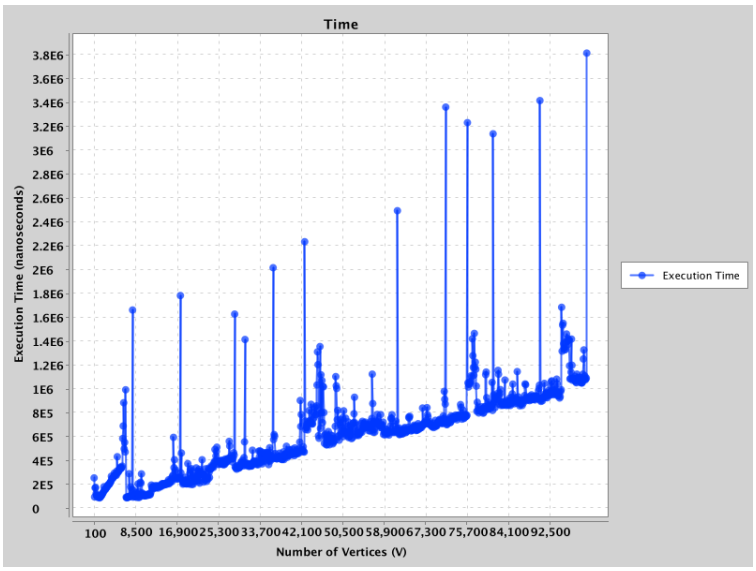
The uniform distribution pick up the vertices looks like in the following, the x axis means the vertex total number, pick up 10000 times, and divide 10000 vertices into 12 groups:



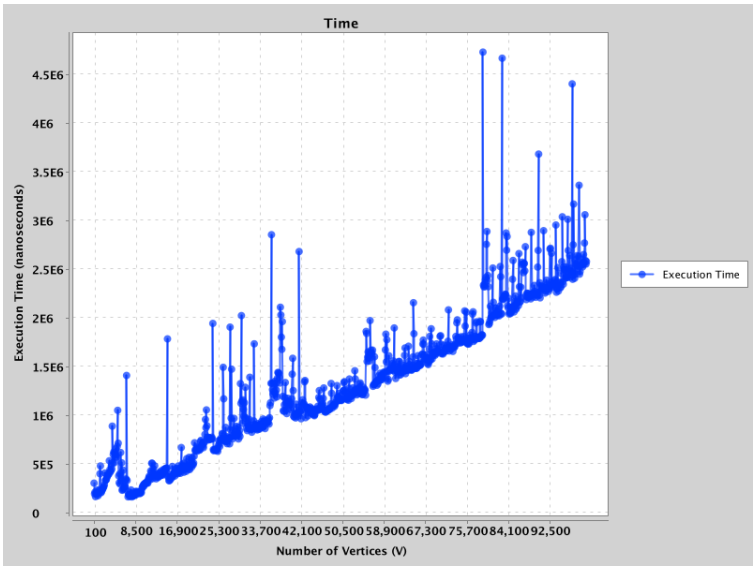
The running time : $\Theta(V \times E)$
Use the data from $V = (100, 200, 300, ..., 100000)$, and different edges as test to run:
$E = \frac{V}{8}$:



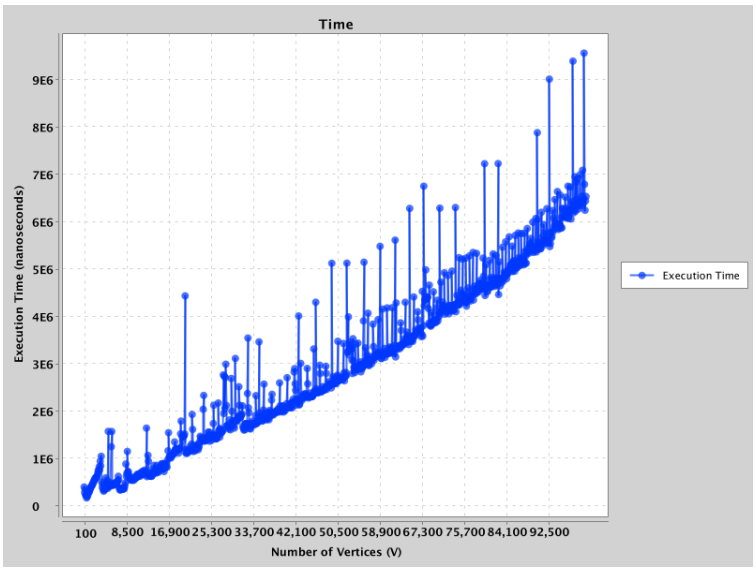
$E = \frac{V}{4}$:


Time
Execution Time (nanoseconds)
3.8E6
3.6E6
3.4E6
3.2E6
3E6
2.8E6
2.6E6
2.4E6
2.2E6
2E6
1.8E6
1.6E6
1.4E6
1.2E6
1E6
8E5
6E5
4E5
2E5
0
100
8,500
16,900
25,300
33,700
42,100
50,500
58,900
67,300
75,700
84,100
92,500
Number of Vertices (V)
Execution Time

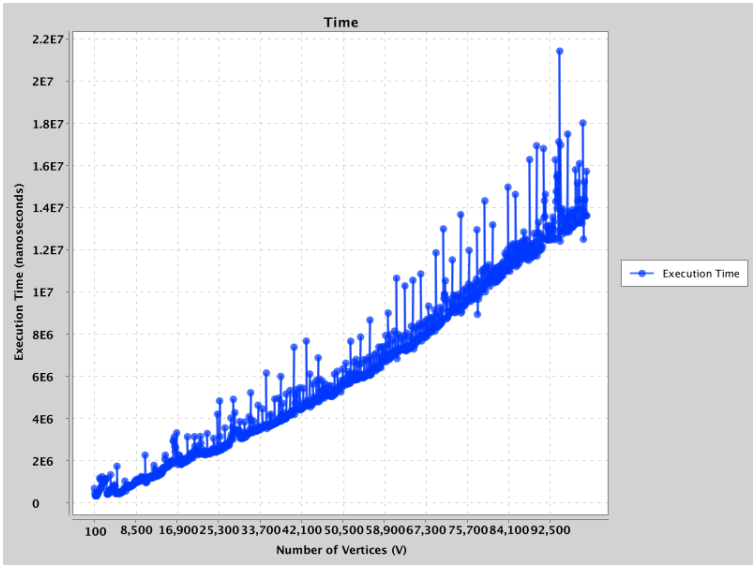
$E = \frac{V}{2}$


Time
Execution Time (nanoseconds)
4.5E6
4E6
3.5E6
3E6
2.5E6
2E6
1.5E6
1E6
5E5
0
100
8,500
16,900
25,300
33,700
42,100
50,500
58,900
67,300
75,700
84,100
92,500
Number of Vertices (V)
Execution Time

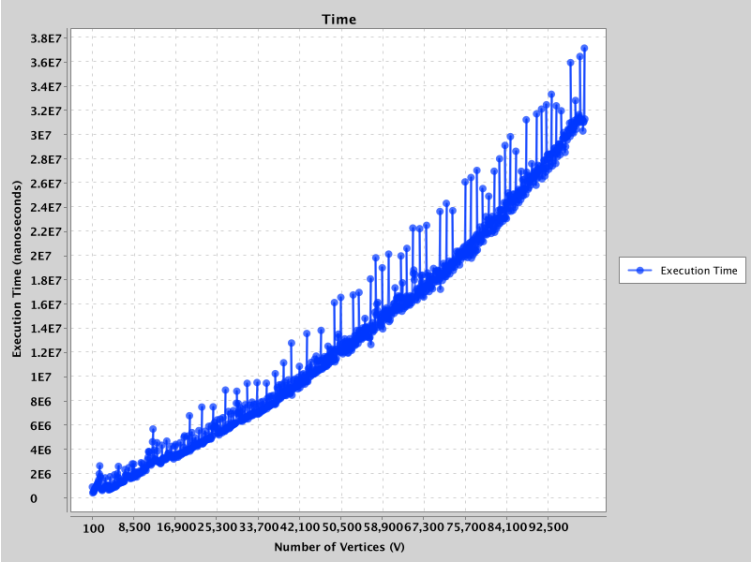
$E = V$:


Time
Execution Time (nanoseconds)
9E6
8E6
7E6
6E6
5E6
4E6
3E6
2E6
1E6
0
100
8,500
16,900
25,300
33,700
42,100
50,500
58,900
67,300
75,700
84,100
92,500
Number of Vertices (V)
Execution Time

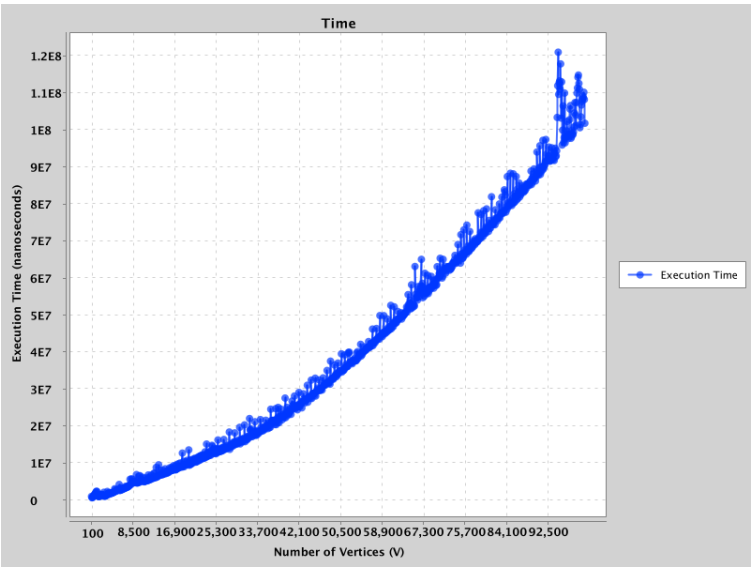
$E = 2V$:


Time
Execution Time (nanoseconds)
2.2E7
2E7
1.8E7
1.6E7
1.4E7
1.2E7
1E7
8E6
6E6
4E6
2E6
0
100
8,500
16,900
25,300
33,700
42,100
50,500
58,900
67,300
75,700
84,100
92,500
Number of Vertices (V)
Execution Time

$E = 4V$:



$E = 8V$

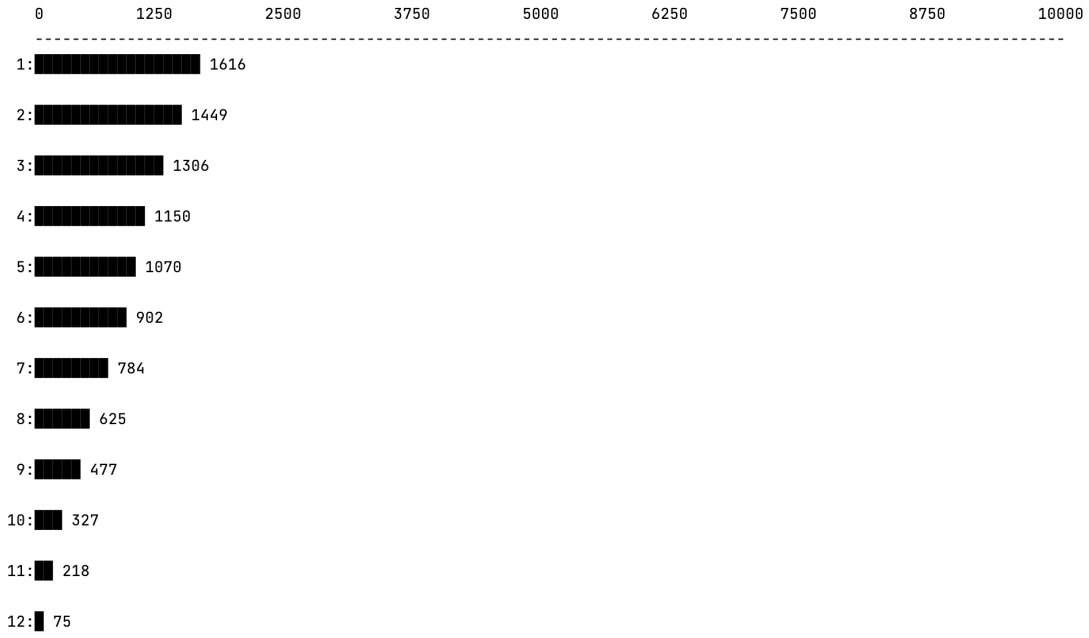


4. SKEWED

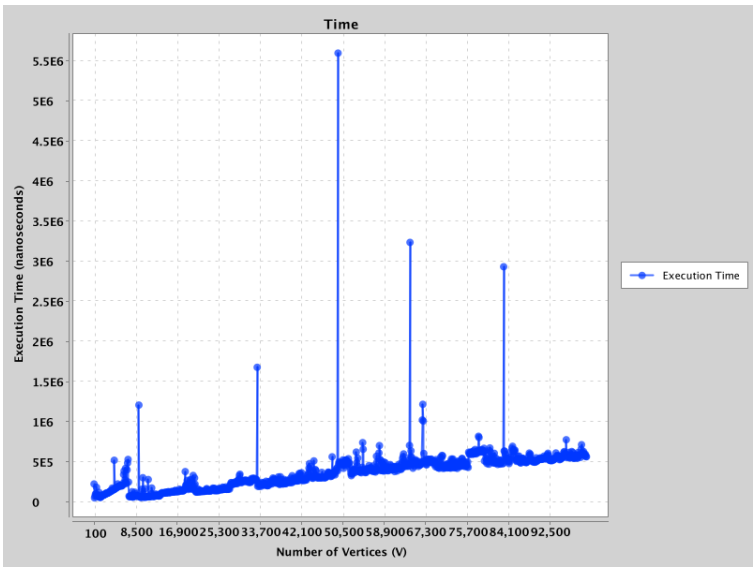
```
    static class Skewed {
        static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
            am.get(s).add(d);
            am.get(d).add(s);
        }

        static void skewedRandom(ArrayList<ArrayList<Integer>> am, int v, int e) {
            int source = -1;
            int dest = -1;
            int a = 0;
            int b = v;
            int c = 0;
            double F = (c - a) / (b - a);
```

```
            while (e > 0) {
                double rand = Math.random();
                if (rand < F) {
                    source = (int) (a + Math.sqrt(rand * (b - a) * (c - a)));
                } else {
                    source = (int) (b - Math.sqrt((1 - rand) * (b - a) * (b - c)));
                }

                double rand2 = Math.random();
                if (rand2 < F) {
                    dest = (int) (a + Math.sqrt(rand2 * (b - a) * (c - a)));
                } else {
                    dest = (int) (b - Math.sqrt((1 - rand2) * (b - a) * (b - c)));
                }

                if (source == dest || am.get(source).contains(dest)) {
                    continue;
                } else {
                    addEdge(am, source, dest);
                    e--;
                }
            }

        }
    }
```

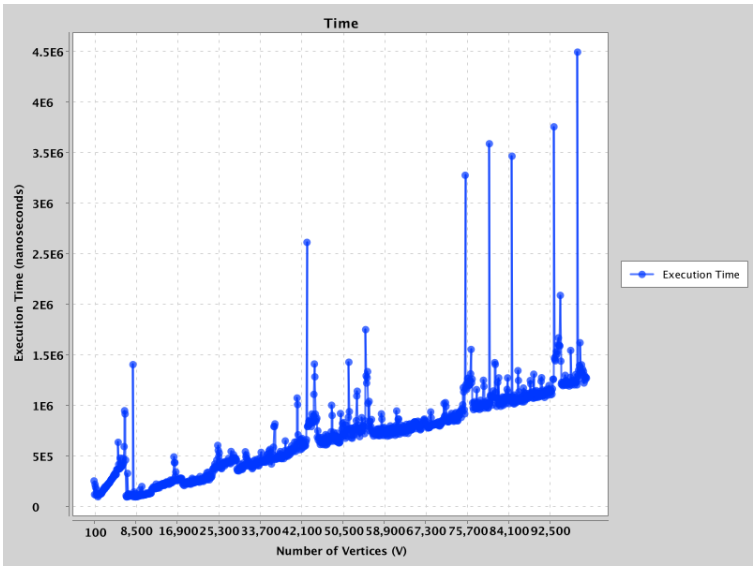
The skewed distribution pick up the vertices looks like in the following, the x axis means the vertex total number, pick up 10000 times, and divide 10000 vertices into 12 groups:



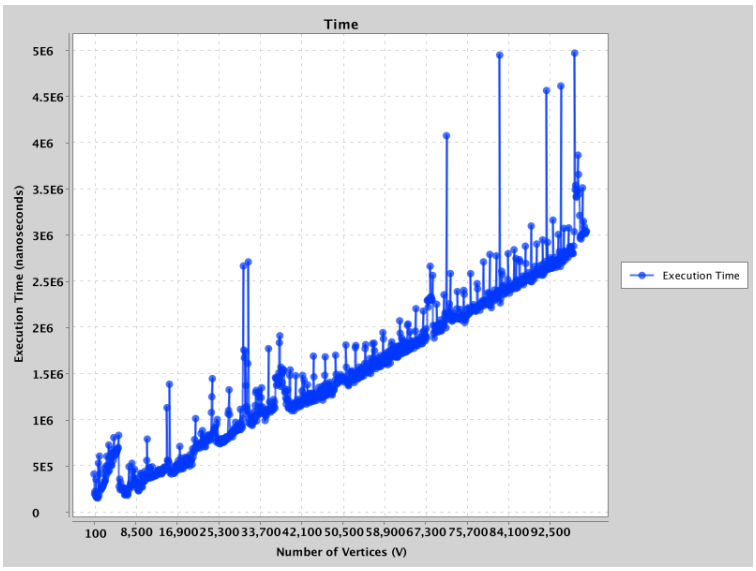
The running time : $\Theta(V \times E)$
Use the data from $V = (100, 200, 300, ..., 100000)$, and different edges as test to run:
$E = \frac{V}{8}$:



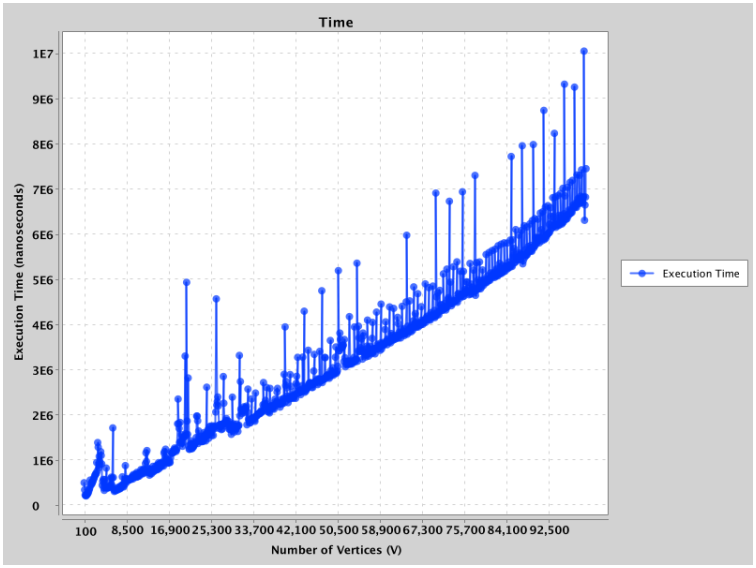
$E = \frac{V}{4}$:



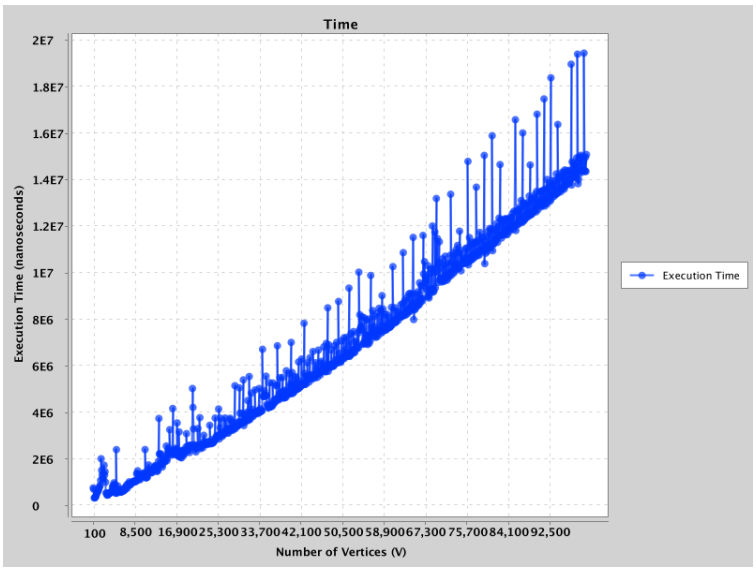
$E = \frac{V}{2}$


Time
Execution Time (nanoseconds)
5E6
4.5E6
4E6
3.5E6
3E6
2.5E6
2E6
1.5E6
1E6
5E5
0
100
8,500
16,900
25,300
33,700
42,100
50,500
58,900
67,300
75,700
84,100
92,500
Number of Vertices (V)
Execution Time

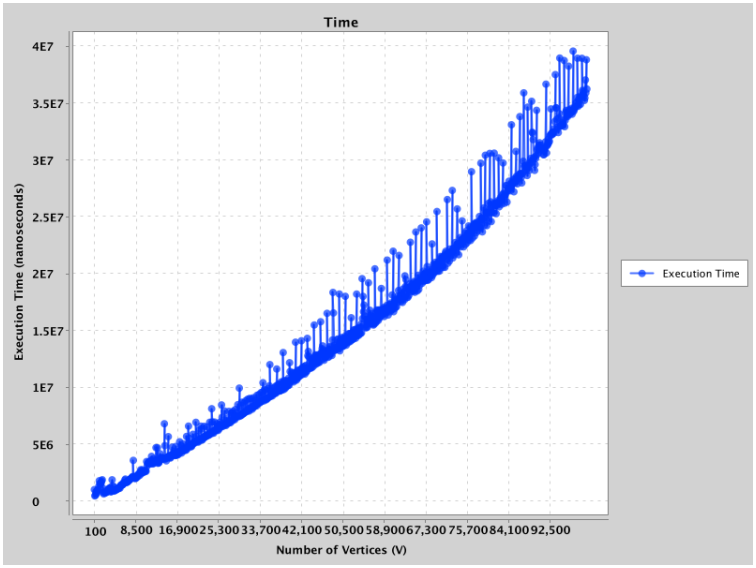
$E = V$:


Time
Execution Time (nanoseconds)
1E7
9E6
8E6
7E6
6E6
5E6
4E6
3E6
2E6
1E6
0
100
8,500
16,900
25,300
33,700
42,100
50,500
58,900
67,300
75,700
84,100
92,500
Number of Vertices (V)
Execution Time

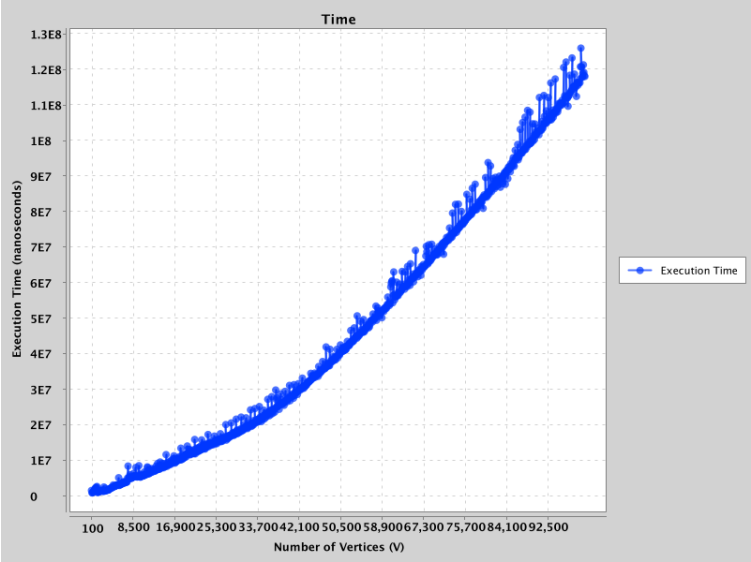
$E = 2V$:


Time
Execution Time (nanoseconds)
2E7
1.8E7
1.6E7
1.4E7
1.2E7
1E7
8E6
6E6
4E6
2E6
0
100
8,500
16,900
25,300
33,700
42,100
50,500
58,900
67,300
75,700
84,100
92,500
Number of Vertices (V)
Execution Time

$E = 4V$:


Time
Execution Time (nanoseconds)
4E7
3.5E7
3E7
2.5E7
2E7
1.5E7
1E7
5E6
0
100
8,500
16,900
25,300
33,700
42,100
50,500
58,900
67,300
75,700
84,100
92,500
Number of Vertices (V)
Execution Time

$E = 8V$

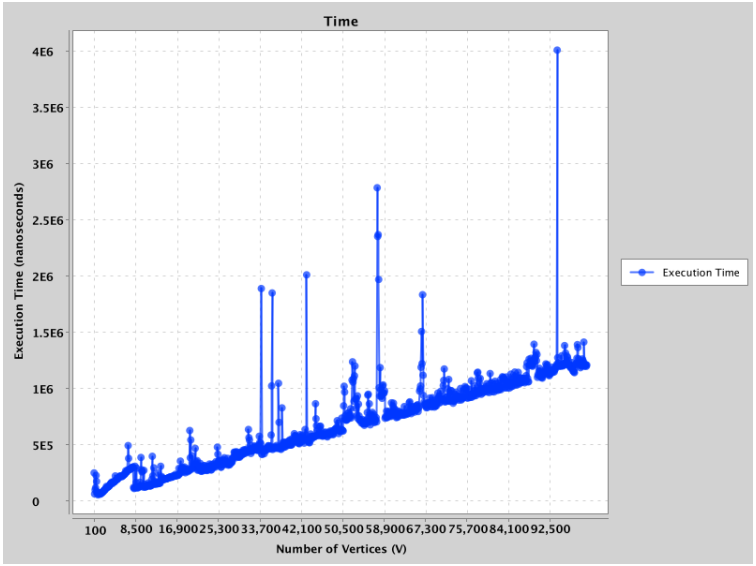


5. GAUSS

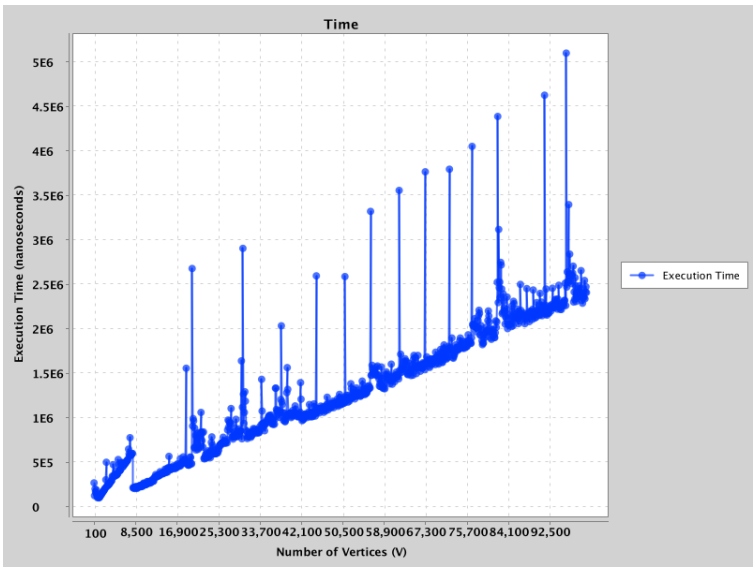
```
    static class Gauss {

        static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
            am.get(s).add(d);
            am.get(d).add(s);
        }

        static void gaussRandom(ArrayList<ArrayList<Integer>> am, int v, int e) {
            Random rand = new Random();
            while (e > 0) {

                int source = (int) (v / 10 * rand.nextGaussian() + v / 2);
                int dest = (int) (v / 10 * rand.nextGaussian() + v / 2);

                if (source == dest || am.get(source).contains(dest)) {
                    continue;
                } else {
                    addEdge(am, source, dest);
                    e--;
                }
            }
        }
```

The guass distribution pick up the vertices looks like in the following, the x axis means the vertex total number, pick up 10000 times, and divide 10000 vertices into 12 groups:

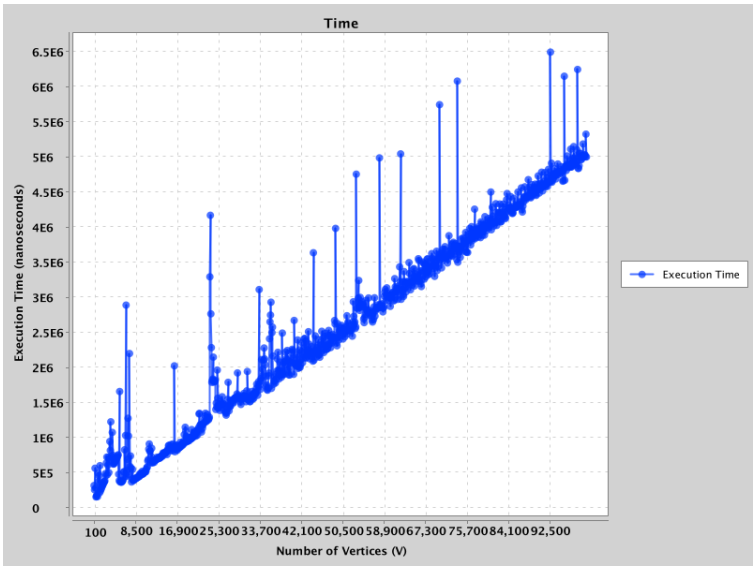
The running time : $\Theta(V \times E)$
Use the data from $V = (100, 200, 300, ..., 100000)$, and different edges as test to run:
$E = \frac{V}{8}$:



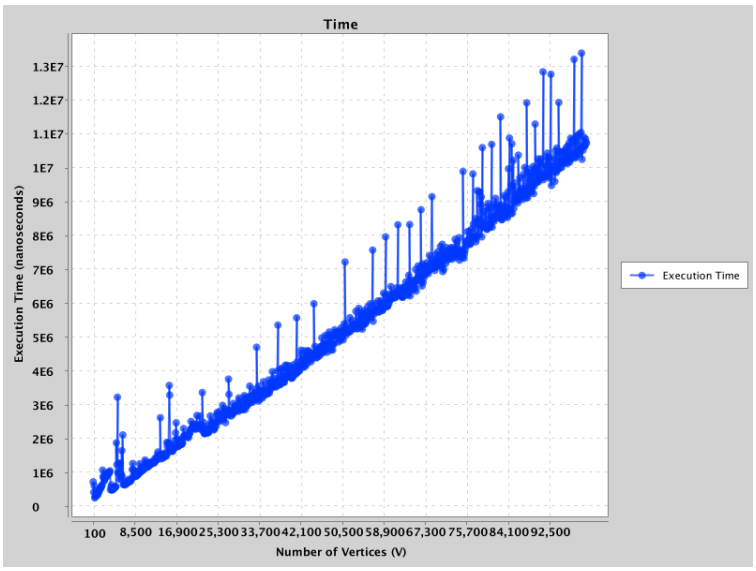
$E = \frac{V}{4}$:


Time
Execution Time (nanoseconds)
5E6
4.5E6
4E6
3.5E6
3E6
2.5E6
2E6
1.5E6
1E6
5E5
0
100
8,500
16,900
25,300
33,700
42,100
50,500
58,900
67,300
75,700
84,100
92,500
Number of Vertices (V)
Execution Time

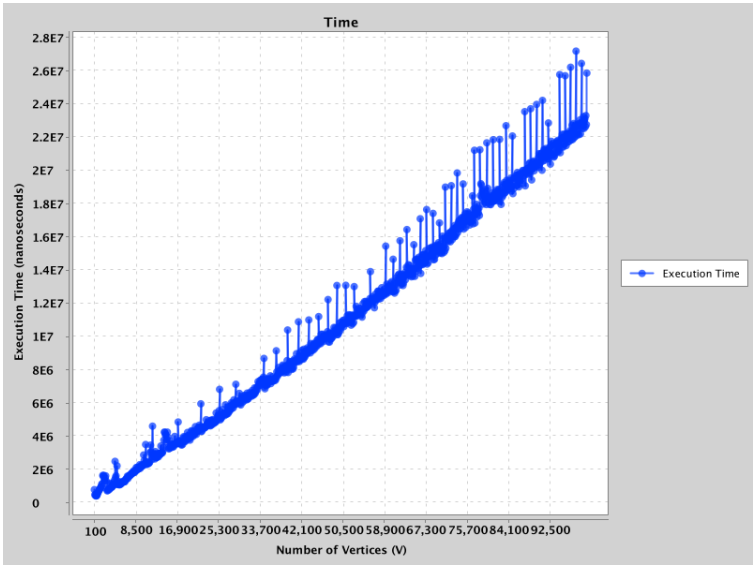
$E = \frac{V}{2}$


Time
Execution Time (nanoseconds)
6.5E6
6E6
5.5E6
5E6
4.5E6
4E6
3.5E6
3E6
2.5E6
2E6
1.5E6
1E6
5E5
0
100
8,500
16,900
25,300
33,700
42,100
50,500
58,900
67,300
75,700
84,100
92,500
Number of Vertices (V)
Execution Time

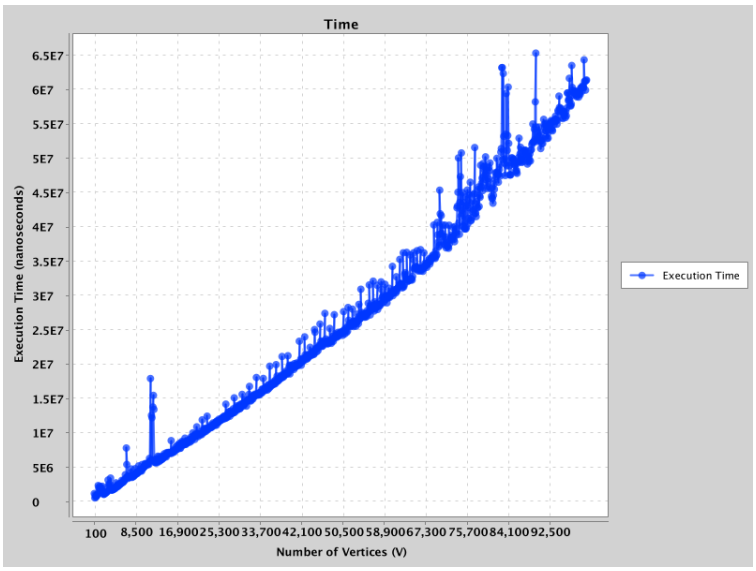
$E = V$:


Time
Execution Time (nanoseconds)
1.3E7
1.2E7
1.1E7
1E7
9E6
8E6
7E6
6E6
5E6
4E6
3E6
2E6
1E6
0
100
8,500
16,900
25,300
33,700
42,100
50,500
58,900
67,300
75,700
84,100
92,500
Number of Vertices (V)
Execution Time

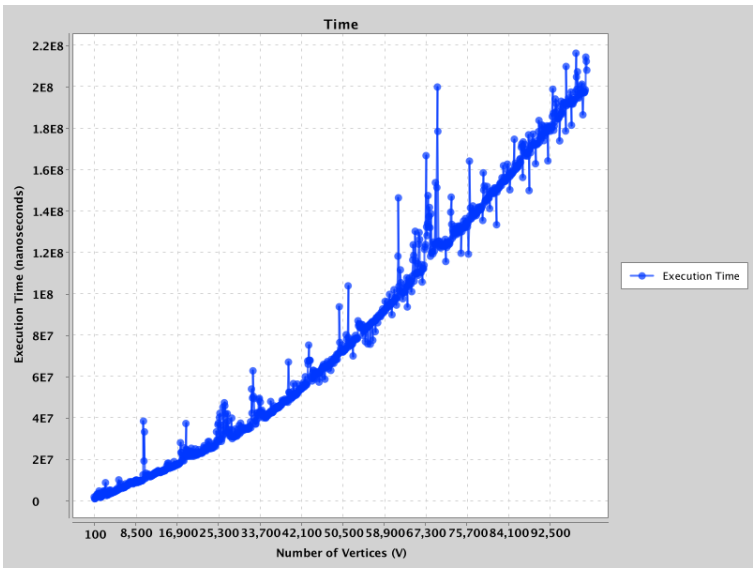
$E = 2V$:


Time
Execution Time (nanoseconds)
2.8E7
2.6E7
2.4E7
2.2E7
2E7
1.8E7
1.6E7
1.4E7
1.2E7
1E7
8E6
6E6
4E6
2E6
0
100
8,500
16,900
25,300
33,700
42,100
50,500
58,900
67,300
75,700
84,100
92,500
Number of Vertices (V)
Execution Time

$E = 4V$:

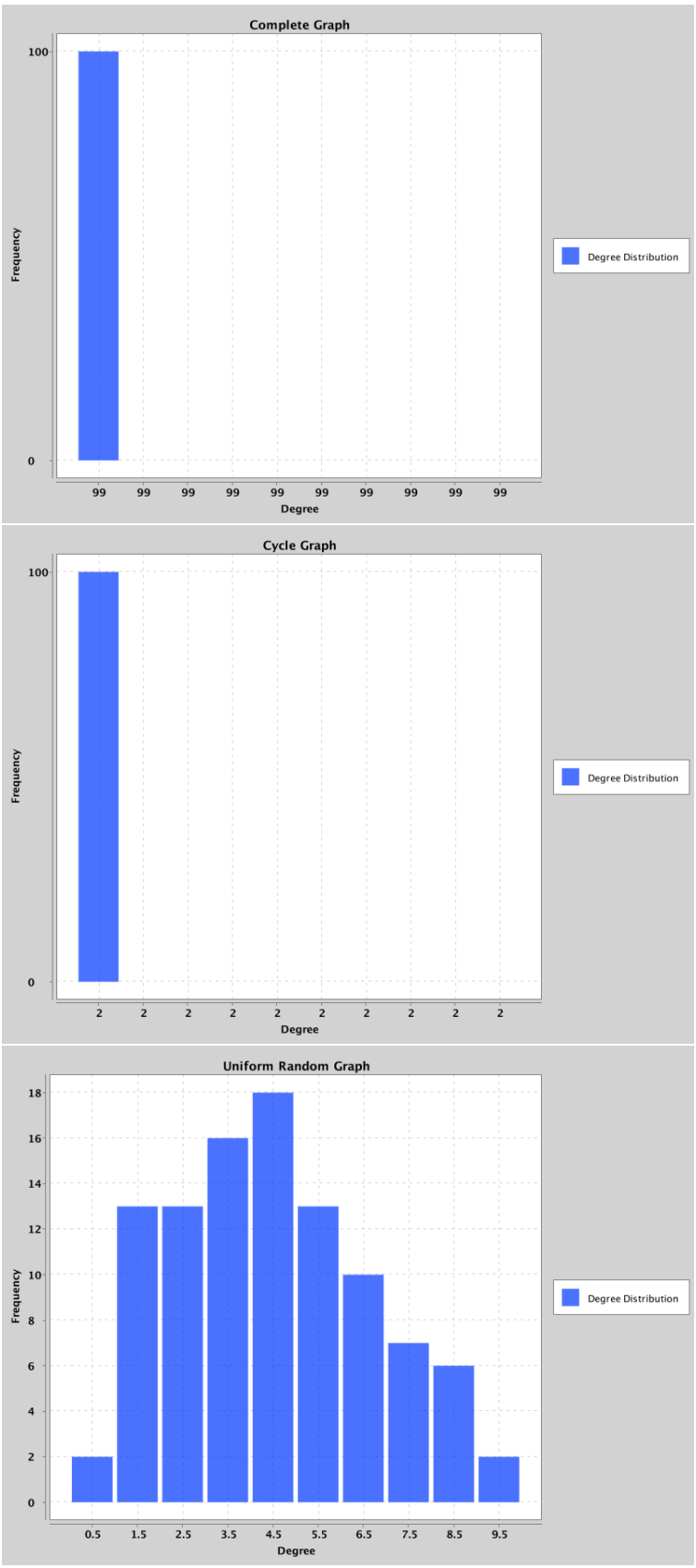


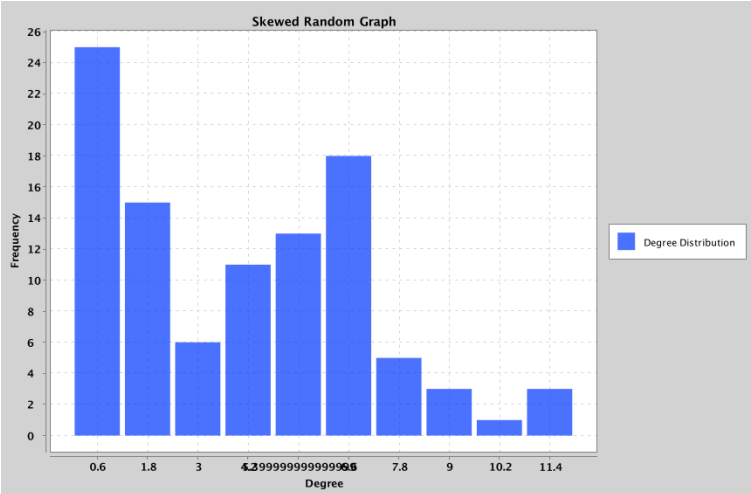
$E = 8V$

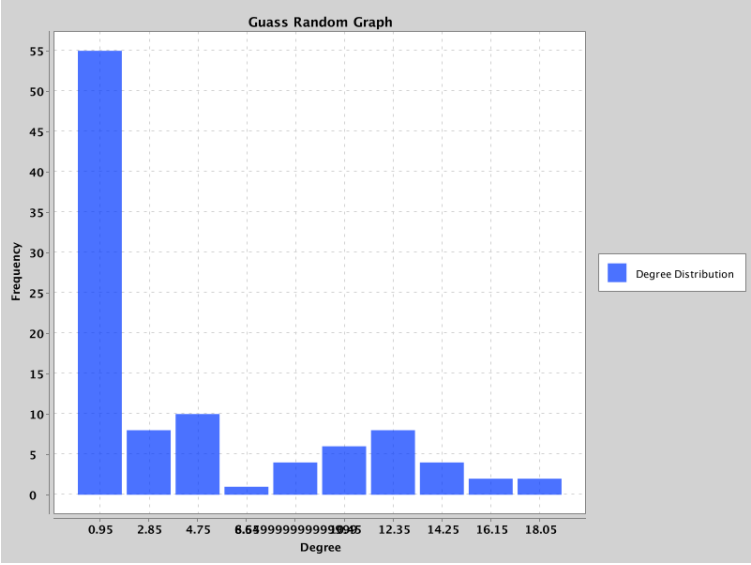


2.1.2 Histograms showing how many conflicts each vertex has for each method

To create histograms showing the number of conflicts (degree) each vertex has for each method, I use the XChart library to create the histograms. I use $v = 100, E = 200$ as example, the result in the following:


Complete Graph
100
0
Frequency
99 99 99 99 99 99 99 99 99 99
Degree
Degree Distribution
Cycle Graph
100
0
Frequency
2 2 2 2 2 2 2 2 2 2
Degree
Degree Distribution
Uniform Random Graph
18
16
14
12
10
8
6
4
2
0
Frequency
0.5 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5
Degree
Degree Distribution

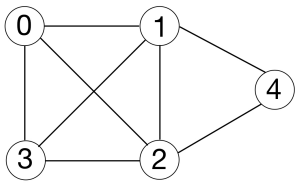




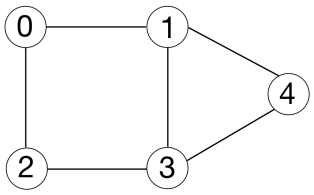
3 Part Two

3.1 Vertex Ordering

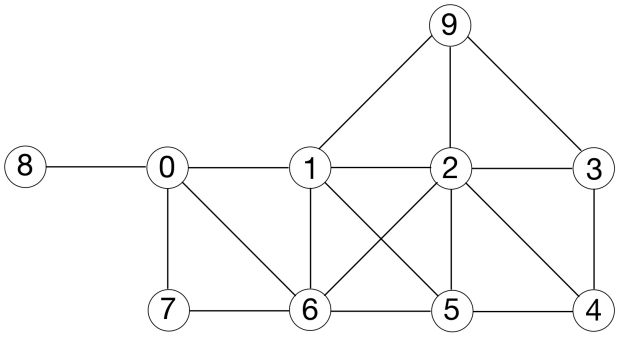
In these part, I use three graph as example:
Example 1:



Example 2:



Example 3:



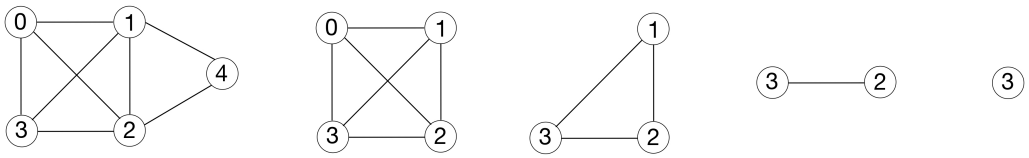
1. Smallest Last Vertex (SLV) Ordering: This algorithm aims to order the vertices in a way that the vertex with the smallest degree comes last. The process is repeated for the remaining vertices until all vertices are ordered. SLV ordering is mainly used for graph coloring and reducing the fill-in of sparse matrices.

```
    // Method 1: Smallest Last Vertex (SLV) Ordering:
    public static int terminalclique;
    public static int maxdegree;
    public static void removeVertex(int vertex, ArrayList<ArrayList<Integer>> am) {
        for (int i = 0; i < am.size(); i++) {
            am.get(i).remove((Integer) vertex);
        }
        am.set(vertex, new ArrayList<>());
    }

    public static int findSmallestDegreeVertex(ArrayList<ArrayList<Integer>> am) {
        int minDegree = Integer.MAX_VALUE;
        int minDegreeVertex = -1;

        for (int i = 0; i < am.size(); i++) {
            int degree = am.get(i).size();

            System.out.println("The degree of Vertex " + i + ": " + degree);
            if (degree < minDegree && degree > 0) {
                minDegree = degree;
                minDegreeVertex = i;

            }
        }

```

```
26          return minDegreeVertex;
27      }
28
29      public static ArrayList<ArrayList<Integer>> copy(ArrayList<ArrayList<Integer>> am){
30          ArrayList<ArrayList<Integer>> am2 = new ArrayList<ArrayList<Integer>>();
31          for (int i = 0; i < am.size(); i++){
32              am2.add(am.get(i));
33          }
34
35          return am2;
36      }
37
38      public static ArrayList<Integer> smallestLastVertexOrdering(ArrayList<ArrayList<Integer>> originalam) {
39          ArrayList<ArrayList<Integer>> am = copy(originalam);
40          ArrayList<Integer> slvOrder = new ArrayList<>();
41          int res = -1;
42          maxdegree = -1;
43          terminalclique = -1;
44
45          while (slvOrder.size() < am.size()) {
46              printAdjacencyList(am);
47              int minDegreeVertex = findSmallestDegreeVertex(am);
48
49              if (minDegreeVertex == -1) {
50                  slvOrder.add(0, res);
51                  break;
52              }
53
54              System.out.println("The min degree Vertex is (will be deleted): " + minDegreeVertex);
55              System.out.println();
56
57              if (am.get(minDegreeVertex).size() > maxdegree){
58                  maxdegree = am.get(minDegreeVertex).size();
59              }
60
61              if (terminalclique < am.get(minDegreeVertex).size() &&
62                      ((am.size()-slvOrder.size()-1) == am.get(minDegreeVertex).size())){
63                  terminalclique = am.get(minDegreeVertex).size();
64              }
65
66
67              slvOrder.add(0, minDegreeVertex);
68              System.out.println(slvOrder);
69              if (am.get(minDegreeVertex).size() == 1){
70                  res = am.get(minDegreeVertex).get(0);
71              }
72              removeVertex(minDegreeVertex,am);
73
74          }
75          System.out.println("The order to delete (from right to left): " + slvOrder);
76          return slvOrder;
77      }
78
```

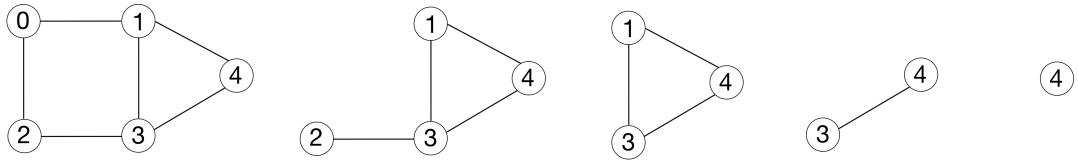
The running time: $\Theta(v)$

Example 1:



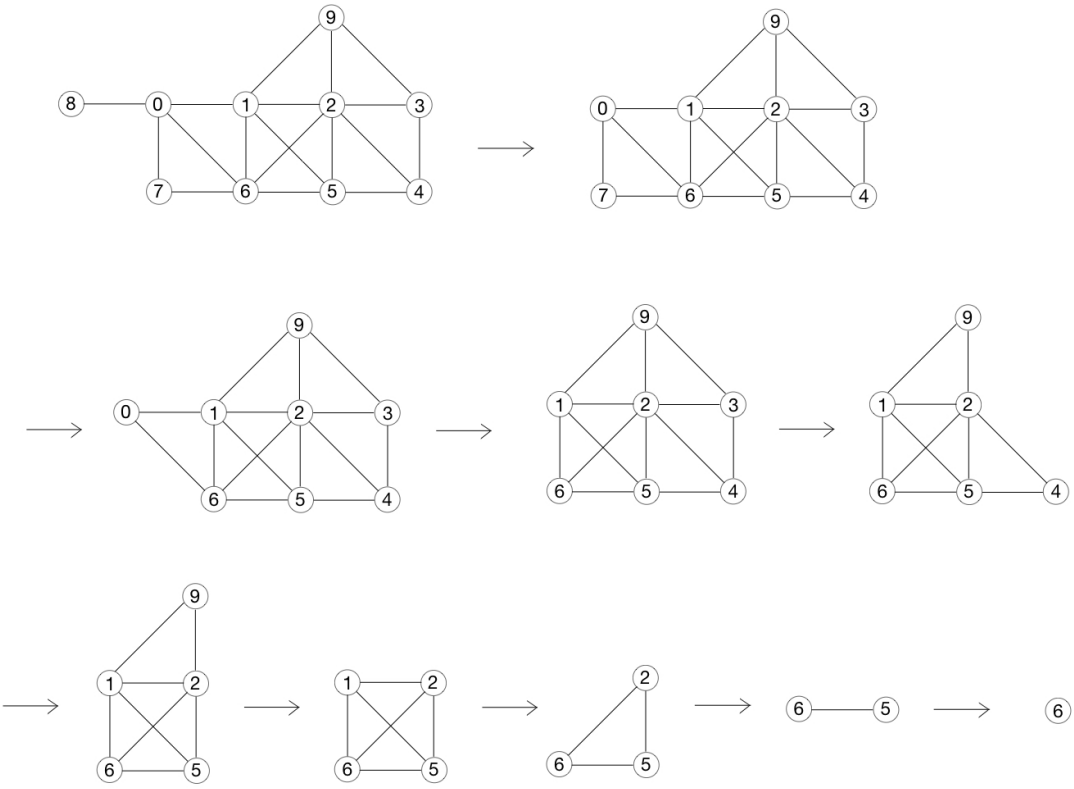
Degrees	0	1	2	3	2
Vertex	3	2	1	0	4

Example 2:



Degrees	0	1	2	1	2
Vertex	4	3	1	2	0

Example 3:



Degrees	0	1	2	3	2	2	3	2	2	1
Vertex	6	5	2	1	9	4	3	0	7	8

The example test result:

```
--------------------Example 1--------------------------------
The original adjacencyList:
Vertex 0 is connected to: 1 2 3
Vertex 1 is connected to: 0 4 3 2
Vertex 2 is connected to: 0 1 3 4
Vertex 3 is connected to: 0 1 2
Vertex 4 is connected to: 1 2

Vertex 0 is connected to: 1 2 3
Vertex 1 is connected to: 0 4 3 2
Vertex 2 is connected to: 0 1 3 4
Vertex 3 is connected to: 0 1 2
Vertex 4 is connected to: 1 2
The degree of Vertex 0: 3
The degree of Vertex 1: 4
The degree of Vertex 2: 4
The degree of Vertex 3: 3
The degree of Vertex 4: 2
The min degree Vertex is (will be deleted): 4

[4]
Vertex 0 is connected to: 1 2 3
Vertex 1 is connected to: 0 3 2
Vertex 2 is connected to: 0 1 3
Vertex 3 is connected to: 0 1 2
Vertex 4 is connected to:
The degree of Vertex 0: 3
The degree of Vertex 1: 3
The degree of Vertex 2: 3
The degree of Vertex 3: 3
The degree of Vertex 4: 0
The min degree Vertex is (will be deleted): 0

[0, 4]
Vertex 0 is connected to:
Vertex 1 is connected to: 3 2
Vertex 2 is connected to: 1 3
Vertex 3 is connected to: 1 2
Vertex 4 is connected to:
The degree of Vertex 0: 0
The degree of Vertex 1: 2
The degree of Vertex 2: 2
The degree of Vertex 3: 2
The degree of Vertex 4: 0
The min degree Vertex is (will be deleted): 1
```

```
[1, 0, 4]
Vertex 0 is connected to:
Vertex 1 is connected to:
Vertex 2 is connected to: 3
Vertex 3 is connected to: 2
Vertex 4 is connected to:
The degree of Vertex 0: 0
The degree of Vertex 1: 0
The degree of Vertex 2: 1
The degree of Vertex 3: 1
The degree of Vertex 4: 0
The min degree Vertex is (will be deleted): 2

[2, 1, 0, 4]
Vertex 0 is connected to:
Vertex 1 is connected to:
Vertex 2 is connected to:
Vertex 3 is connected to:
Vertex 4 is connected to:
The degree of Vertex 0: 0
The degree of Vertex 1: 0
The degree of Vertex 2: 0
The degree of Vertex 3: 0
The degree of Vertex 4: 0
The order to delete (from right to left): [3, 2, 1, 0, 4]
The color result:
Vertex 3 --->  Color 0
Vertex 2 --->  Color 1
Vertex 1 --->  Color 2
Vertex 0 --->  Color 3
Vertex 4 --->  Color 0
The size of terminal clique degree: 3
The max degree during removing order: 3
```

```
--------------------Example 2--------------------------------
The original adjacencyList:
Vertex 0 is connected to: 1 2
Vertex 1 is connected to: 0 4 3
Vertex 2 is connected to: 0 3
Vertex 3 is connected to: 1 2 4
Vertex 4 is connected to: 1 3

Vertex 0 is connected to: 1 2
Vertex 1 is connected to: 0 4 3
Vertex 2 is connected to: 0 3
Vertex 3 is connected to: 1 2 4
Vertex 4 is connected to: 1 3
The degree of Vertex 0: 2
The degree of Vertex 1: 3
The degree of Vertex 2: 2
The degree of Vertex 3: 3
The degree of Vertex 4: 2
The min degree Vertex is (will be deleted): 0

[0]
Vertex 0 is connected to:
Vertex 1 is connected to: 4 3
Vertex 2 is connected to: 3
Vertex 3 is connected to: 1 2 4
Vertex 4 is connected to: 1 3
The degree of Vertex 0: 0
The degree of Vertex 1: 2
The degree of Vertex 2: 1
The degree of Vertex 3: 3
The degree of Vertex 4: 2
The min degree Vertex is (will be deleted): 2

[2, 0]
Vertex 0 is connected to:
Vertex 1 is connected to: 4 3
Vertex 2 is connected to:
Vertex 3 is connected to: 1 4
Vertex 4 is connected to: 1 3
The degree of Vertex 0: 0
The degree of Vertex 1: 2
The degree of Vertex 2: 0
The degree of Vertex 3: 2
The degree of Vertex 4: 2
The min degree Vertex is (will be deleted): 1
```

```
[1, 2, 0]
Vertex 0 is connected to:
Vertex 1 is connected to:
Vertex 2 is connected to:
Vertex 3 is connected to: 4
Vertex 4 is connected to: 3
The degree of Vertex 0: 0
The degree of Vertex 1: 0
The degree of Vertex 2: 0
The degree of Vertex 3: 1
The degree of Vertex 4: 1
The min degree Vertex is (will be deleted): 3

[3, 1, 2, 0]
Vertex 0 is connected to:
Vertex 1 is connected to:
Vertex 2 is connected to:
Vertex 3 is connected to:
Vertex 4 is connected to:
The degree of Vertex 0: 0
The degree of Vertex 1: 0
The degree of Vertex 2: 0
The degree of Vertex 3: 0
The degree of Vertex 4: 0
The order to delete (from right to left): [4, 3, 1, 2, 0]
The color result:
Vertex 4 --->  Color 0
Vertex 3 --->  Color 1
Vertex 1 --->  Color 2
Vertex 2 --->  Color 0
Vertex 0 --->  Color 1
The size of terminal clique degree: 2
The max degree during removing order: 2
```

```
----------------------Example 3---------------------------------
The original adjacencyList:
Vertex 0 is connected to: 1 6 7 8
Vertex 1 is connected to: 0 6 2 9 5
Vertex 2 is connected to: 1 9 5 3 4 6
Vertex 3 is connected to: 2 4 9
Vertex 4 is connected to: 2 3 5
Vertex 5 is connected to: 1 2 4 6
Vertex 6 is connected to: 0 1 2 5 7
Vertex 7 is connected to: 0 6
Vertex 8 is connected to: 0
Vertex 9 is connected to: 1 2 3

Vertex 0 is connected to: 1 6 7 8
Vertex 1 is connected to: 0 6 2 9 5
Vertex 2 is connected to: 1 9 5 3 4 6
Vertex 3 is connected to: 2 4 9
Vertex 4 is connected to: 2 3 5
Vertex 5 is connected to: 1 2 4 6
Vertex 6 is connected to: 0 1 2 5 7
Vertex 7 is connected to: 0 6
Vertex 8 is connected to: 0
Vertex 9 is connected to: 1 2 3
The degree of Vertex 0: 4
The degree of Vertex 1: 5
The degree of Vertex 2: 6
The degree of Vertex 3: 3
The degree of Vertex 4: 3
The degree of Vertex 5: 4
The degree of Vertex 6: 5
The degree of Vertex 7: 2
The degree of Vertex 8: 1
The degree of Vertex 9: 3
The min degree Vertex is (will be deleted): 8
```

```
[8]
Vertex 0 is connected to: 1 6 7
Vertex 1 is connected to: 0 6 2 9 5
Vertex 2 is connected to: 1 9 5 3 4 6
Vertex 3 is connected to: 2 4 9
Vertex 4 is connected to: 2 3 5
Vertex 5 is connected to: 1 2 4 6
Vertex 6 is connected to: 0 1 2 5 7
Vertex 7 is connected to: 0 6
Vertex 8 is connected to:
Vertex 9 is connected to: 1 2 3
The degree of Vertex 0: 3
The degree of Vertex 1: 5
The degree of Vertex 2: 6
The degree of Vertex 3: 3
The degree of Vertex 4: 3
The degree of Vertex 5: 4
The degree of Vertex 6: 5
The degree of Vertex 7: 2
The degree of Vertex 8: 0
The degree of Vertex 9: 3
The min degree Vertex is (will be deleted): 7

[7, 8]
Vertex 0 is connected to: 1 6
Vertex 1 is connected to: 0 6 2 9 5
Vertex 2 is connected to: 1 9 5 3 4 6
Vertex 3 is connected to: 2 4 9
Vertex 4 is connected to: 2 3 5
Vertex 5 is connected to: 1 2 4 6
Vertex 6 is connected to: 0 1 2 5
Vertex 7 is connected to:
Vertex 8 is connected to:
Vertex 9 is connected to: 1 2 3
The degree of Vertex 0: 2
The degree of Vertex 1: 5
The degree of Vertex 2: 6
The degree of Vertex 3: 3
The degree of Vertex 4: 3
The degree of Vertex 5: 4
The degree of Vertex 6: 4
The degree of Vertex 7: 0
The degree of Vertex 8: 0
The degree of Vertex 9: 3
The min degree Vertex is (will be deleted): 0
```

```
[0, 7, 8]
Vertex 0 is connected to:
Vertex 1 is connected to: 6 2 9 5
Vertex 2 is connected to: 1 9 5 3 4 6
Vertex 3 is connected to: 2 4 9
Vertex 4 is connected to: 2 3 5
Vertex 5 is connected to: 1 2 4 6
Vertex 6 is connected to: 1 2 5
Vertex 7 is connected to:
Vertex 8 is connected to:
Vertex 9 is connected to: 1 2 3
The degree of Vertex 0: 0
The degree of Vertex 1: 4
The degree of Vertex 2: 6
The degree of Vertex 3: 3
The degree of Vertex 4: 3
The degree of Vertex 5: 4
The degree of Vertex 6: 3
The degree of Vertex 7: 0
The degree of Vertex 8: 0
The degree of Vertex 9: 3
The min degree Vertex is (will be deleted): 3

[3, 0, 7, 8]
Vertex 0 is connected to:
Vertex 1 is connected to: 6 2 9 5
Vertex 2 is connected to: 1 9 5 4 6
Vertex 3 is connected to:
Vertex 4 is connected to: 2 5
Vertex 5 is connected to: 1 2 4 6
Vertex 6 is connected to: 1 2 5
Vertex 7 is connected to:
Vertex 8 is connected to:
Vertex 9 is connected to: 1 2
The degree of Vertex 0: 0
The degree of Vertex 1: 4
The degree of Vertex 2: 5
The degree of Vertex 3: 0
The degree of Vertex 4: 2
The degree of Vertex 5: 4
The degree of Vertex 6: 3
The degree of Vertex 7: 0
The degree of Vertex 8: 0
The degree of Vertex 9: 2
The min degree Vertex is (will be deleted): 4
```

```
[4, 3, 0, 7, 8]
Vertex 0 is connected to:
Vertex 1 is connected to: 6 2 9 5
Vertex 2 is connected to: 1 9 5 6
Vertex 3 is connected to:
Vertex 4 is connected to:
Vertex 5 is connected to: 1 2 6
Vertex 6 is connected to: 1 2 5
Vertex 7 is connected to:
Vertex 8 is connected to:
Vertex 9 is connected to: 1 2
The degree of Vertex 0: 0
The degree of Vertex 1: 4
The degree of Vertex 2: 4
The degree of Vertex 3: 0
The degree of Vertex 4: 0
The degree of Vertex 5: 3
The degree of Vertex 6: 3
The degree of Vertex 7: 0
The degree of Vertex 8: 0
The degree of Vertex 9: 2
The min degree Vertex is (will be deleted): 9

[9, 4, 3, 0, 7, 8]
Vertex 0 is connected to:
Vertex 1 is connected to: 6 2 5
Vertex 2 is connected to: 1 5 6
Vertex 3 is connected to:
Vertex 4 is connected to:
Vertex 5 is connected to: 1 2 6
Vertex 6 is connected to: 1 2 5
Vertex 7 is connected to:
Vertex 8 is connected to:
Vertex 9 is connected to:
The degree of Vertex 0: 0
The degree of Vertex 1: 3
The degree of Vertex 2: 3
The degree of Vertex 3: 0
The degree of Vertex 4: 0
The degree of Vertex 5: 3
The degree of Vertex 6: 3
The degree of Vertex 7: 0
The degree of Vertex 8: 0
The degree of Vertex 9: 0
The min degree Vertex is (will be deleted): 1
```

```
[1, 9, 4, 3, 0, 7, 8]
Vertex 0 is connected to:
Vertex 1 is connected to:
Vertex 2 is connected to: 5 6
Vertex 3 is connected to:
Vertex 4 is connected to:
Vertex 5 is connected to: 2 6
Vertex 6 is connected to: 2 5
Vertex 7 is connected to:
Vertex 8 is connected to:
Vertex 9 is connected to:
The degree of Vertex 0: 0
The degree of Vertex 1: 0
The degree of Vertex 2: 2
The degree of Vertex 3: 0
The degree of Vertex 4: 0
The degree of Vertex 5: 2
The degree of Vertex 6: 2
The degree of Vertex 7: 0
The degree of Vertex 8: 0
The degree of Vertex 9: 0
The min degree Vertex is (will be deleted): 2

[2, 1, 9, 4, 3, 0, 7, 8]
Vertex 0 is connected to:
Vertex 1 is connected to:
Vertex 2 is connected to:
Vertex 3 is connected to:
Vertex 4 is connected to:
Vertex 5 is connected to: 6
Vertex 6 is connected to: 5
Vertex 7 is connected to:
Vertex 8 is connected to:
Vertex 9 is connected to:
The degree of Vertex 0: 0
The degree of Vertex 1: 0
The degree of Vertex 2: 0
The degree of Vertex 3: 0
The degree of Vertex 4: 0
The degree of Vertex 5: 1
The degree of Vertex 6: 1
The degree of Vertex 7: 0
The degree of Vertex 8: 0
The degree of Vertex 9: 0
The min degree Vertex is (will be deleted): 5
```

```
[5, 2, 1, 9, 4, 3, 0, 7, 8]
Vertex 0 is connected to:
Vertex 1 is connected to:
Vertex 2 is connected to:
Vertex 3 is connected to:
Vertex 4 is connected to:
Vertex 5 is connected to:
Vertex 6 is connected to:
Vertex 7 is connected to:
Vertex 8 is connected to:
Vertex 9 is connected to:
The degree of Vertex 0: 0
The degree of Vertex 1: 0
The degree of Vertex 2: 0
The degree of Vertex 3: 0
The degree of Vertex 4: 0
The degree of Vertex 5: 0
The degree of Vertex 6: 0
The degree of Vertex 7: 0
The degree of Vertex 8: 0
The degree of Vertex 9: 0
The order to delete (from right to left): [6, 5, 2, 1, 9, 4, 3, 0, 7, 8]
The color result:
Vertex 6 --->  Color 0
Vertex 5 --->  Color 1
Vertex 2 --->  Color 2
Vertex 1 --->  Color 3
Vertex 9 --->  Color 0
Vertex 4 --->  Color 0
Vertex 3 --->  Color 1
Vertex 0 --->  Color 1
Vertex 7 --->  Color 2
Vertex 8 --->  Color 0
The size of terminal clique degree: 3
The max degree during removing order: 3

Process finished with exit code 0
```

2. Smallest Original Degree Last (SODL) Ordering: This algorithm is similar to the Smallest Last Vertex (SLV) ordering, but instead of using the current degree during the algorithm's execution, it considers the original degree of each vertex. The vertex with the smallest original degree is placed last, and the process is repeated for the remaining vertices. The Smallest Original Degree Last method is a subset of the smallest last ordering. Should determine the vertices to color based on their original degree, but not remove them from graph. This should run in $\Theta(V + E)$.

```java
// Method 2: Smallest Original Degree Last (SODL) Ordering:
    public static int findMaxDegreeVertex(ArrayList<ArrayList<Integer>> am, ArrayList<Integer> exit){
        int maxDegree = Integer.MIN_VALUE;
        int maxDegreeVertex = -1;

        for (int i = 0; i < am.size(); i++){
            int degree = am.get(i).size();

            System.out.println("The degree of Vertex " + i + ": " + degree);
```

```
            if (degree > maxDegree && degree > 0 && !exit.contains(i)){
                maxDegree = degree;
                maxDegreeVertex = i;
            }
        }

        return maxDegreeVertex;
    }


    private static ArrayList<Integer> smallestOriginalDegreeLast(ArrayList<ArrayList<Integer>> am){
        ArrayList<Integer> solOrder = new ArrayList<>();
        int V = am.size();
        System.out.println(V);


        while(solOrder.size() < am.size()){
            int minDegreeVertex = findMaxDegreeVertex(am, solOrder);

            System.out.println("The vertex add to solOrder "+ minDegreeVertex);

            solOrder.add(0, minDegreeVertex);

        }

        System.out.println("smallest orginal degree Last(from right to left) " + solOrder);

        return solOrder;
    }

```

The example test result:

```
---------------------Example 1--------------------------------
The original adjacencyList:
Vertex 0 is connected to: 1 2 3
Vertex 1 is connected to: 0 4 3 2
Vertex 2 is connected to: 0 1 3 4
Vertex 3 is connected to: 0 1 2
Vertex 4 is connected to: 1 2

The Method for graph order result(from right to left):
The degree of Vertex 0: 3
The degree of Vertex 1: 4
The degree of Vertex 2: 4
The degree of Vertex 3: 3
The degree of Vertex 4: 2
The vertex add to solOrder 1

The degree of Vertex 0: 3
The degree of Vertex 1: 4
The degree of Vertex 2: 4
The degree of Vertex 3: 3
The degree of Vertex 4: 2
The vertex add to solOrder 2

The degree of Vertex 0: 3
The degree of Vertex 1: 4
The degree of Vertex 2: 4
The degree of Vertex 3: 3
The degree of Vertex 4: 2
The vertex add to solOrder 0

The degree of Vertex 0: 3
The degree of Vertex 1: 4
The degree of Vertex 2: 4
The degree of Vertex 3: 3
The degree of Vertex 4: 2
The vertex add to solOrder 3

The degree of Vertex 0: 3
The degree of Vertex 1: 4
The degree of Vertex 2: 4
The degree of Vertex 3: 3
The degree of Vertex 4: 2
The vertex add to solOrder 4

Smallest orginal degree Last ORder (from right to left):
[4, 3, 0, 2, 1]
The color result:
Vertex 4 --->  Color 0
Vertex 3 --->  Color 0
Vertex 0 --->  Color 1
Vertex 2 --->  Color 2
Vertex 1 --->  Color 3

The running time: 11023583
```

```
----------------------Example 2--------------------------------
The orignial graph:
Vertex 0 is connected to: 1 2
Vertex 1 is connected to: 0 4 3
Vertex 2 is connected to: 0 3
Vertex 3 is connected to: 1 2 4
Vertex 4 is connected to: 1 3

The Method for graph order result(from right to left):
The degree of Vertex 0: 2
The degree of Vertex 1: 3
The degree of Vertex 2: 2
The degree of Vertex 3: 3
The degree of Vertex 4: 2
The vertex add to solOrder 1

The degree of Vertex 0: 2
The degree of Vertex 1: 3
The degree of Vertex 2: 2
The degree of Vertex 3: 3
The degree of Vertex 4: 2
The vertex add to solOrder 3

The degree of Vertex 0: 2
The degree of Vertex 1: 3
The degree of Vertex 2: 2
The degree of Vertex 3: 3
The degree of Vertex 4: 2
The vertex add to solOrder 0

The degree of Vertex 0: 2
The degree of Vertex 1: 3
The degree of Vertex 2: 2
The degree of Vertex 3: 3
The degree of Vertex 4: 2
The vertex add to solOrder 2

The degree of Vertex 0: 2
The degree of Vertex 1: 3
The degree of Vertex 2: 2
The degree of Vertex 3: 3
The degree of Vertex 4: 2
The vertex add to solOrder 4

Smallest orginal degree Last ORder (from right to left):
[4, 2, 0, 3, 1]
The color result:
Vertex 4 --->  Color 0
Vertex 2 --->  Color 0
Vertex 0 --->  Color 1
Vertex 3 --->  Color 1
Vertex 1 --->  Color 2

The running time: 911125
```

```
----------------------Example 3--------------------------------
The original adjacencyList:
Vertex 0 is connected to: 1 6 7 8
Vertex 1 is connected to: 0 6 2 9 5
Vertex 2 is connected to: 1 9 5 3 4 6
Vertex 3 is connected to: 2 4 9
Vertex 4 is connected to: 2 3 5
Vertex 5 is connected to: 1 2 4 6
Vertex 6 is connected to: 0 1 2 5 7
Vertex 7 is connected to: 0 6
Vertex 8 is connected to: 0
Vertex 9 is connected to: 1 2 3

The Method for graph order result(from right to left):
The degree of Vertex 0: 4
The degree of Vertex 1: 5
The degree of Vertex 2: 6
The degree of Vertex 3: 3
The degree of Vertex 4: 3
The degree of Vertex 5: 4
The degree of Vertex 6: 5
The degree of Vertex 7: 2
The degree of Vertex 8: 1
The degree of Vertex 9: 3
The vertex add to solOrder 2

The degree of Vertex 0: 4
The degree of Vertex 1: 5
The degree of Vertex 2: 6
The degree of Vertex 3: 3
The degree of Vertex 4: 3
The degree of Vertex 5: 4
The degree of Vertex 6: 5
The degree of Vertex 7: 2
The degree of Vertex 8: 1
The degree of Vertex 9: 3
The vertex add to solOrder 1

The degree of Vertex 0: 4
The degree of Vertex 1: 5
The degree of Vertex 2: 6
The degree of Vertex 3: 3
The degree of Vertex 4: 3
The degree of Vertex 5: 4
The degree of Vertex 6: 5
The degree of Vertex 7: 2
The degree of Vertex 8: 1
The degree of Vertex 9: 3
The vertex add to solOrder 6
```

```
The degree of Vertex 0: 4
The degree of Vertex 1: 5
The degree of Vertex 2: 6
The degree of Vertex 3: 3
The degree of Vertex 4: 3
The degree of Vertex 5: 4
The degree of Vertex 6: 5
The degree of Vertex 7: 2
The degree of Vertex 8: 1
The degree of Vertex 9: 3
The vertex add to solOrder 0

The degree of Vertex 0: 4
The degree of Vertex 1: 5
The degree of Vertex 2: 6
The degree of Vertex 3: 3
The degree of Vertex 4: 3
The degree of Vertex 5: 4
The degree of Vertex 6: 5
The degree of Vertex 7: 2
The degree of Vertex 8: 1
The degree of Vertex 9: 3
The vertex add to solOrder 5

The degree of Vertex 0: 4
The degree of Vertex 1: 5
The degree of Vertex 2: 6
The degree of Vertex 3: 3
The degree of Vertex 4: 3
The degree of Vertex 5: 4
The degree of Vertex 6: 5
The degree of Vertex 7: 2
The degree of Vertex 8: 1
The degree of Vertex 9: 3
The vertex add to solOrder 3

The degree of Vertex 0: 4
The degree of Vertex 1: 5
The degree of Vertex 2: 6
The degree of Vertex 3: 3
The degree of Vertex 4: 3
The degree of Vertex 5: 4
The degree of Vertex 6: 5
The degree of Vertex 7: 2
The degree of Vertex 8: 1
The degree of Vertex 9: 3
The vertex add to solOrder 4
```

```
The degree of Vertex 0: 4
The degree of Vertex 1: 5
The degree of Vertex 2: 6
The degree of Vertex 3: 3
The degree of Vertex 4: 3
The degree of Vertex 5: 4
The degree of Vertex 6: 5
The degree of Vertex 7: 2
The degree of Vertex 8: 1
The degree of Vertex 9: 3
The vertex add to solOrder 9

The degree of Vertex 0: 4
The degree of Vertex 1: 5
The degree of Vertex 2: 6
The degree of Vertex 3: 3
The degree of Vertex 4: 3
The degree of Vertex 5: 4
The degree of Vertex 6: 5
The degree of Vertex 7: 2
The degree of Vertex 8: 1
The degree of Vertex 9: 3
The vertex add to solOrder 7

The degree of Vertex 0: 4
The degree of Vertex 1: 5
The degree of Vertex 2: 6
The degree of Vertex 3: 3
The degree of Vertex 4: 3
The degree of Vertex 5: 4
The degree of Vertex 6: 5
The degree of Vertex 7: 2
The degree of Vertex 8: 1
The degree of Vertex 9: 3
The vertex add to solOrder 8

Smallest orginal degree Last ORder (from right to left):
[8, 7, 9, 4, 3, 5, 0, 6, 1, 2]
The color result:
Vertex 8 --->  Color 0
Vertex 7 --->  Color 0
Vertex 9 --->  Color 0
Vertex 4 --->  Color 0
Vertex 3 --->  Color 1
Vertex 5 --->  Color 1
Vertex 0 --->  Color 1
Vertex 6 --->  Color 2
Vertex 1 --->  Color 3
Vertex 2 --->  Color 4
The running time: 1090792
```

3. Uniform Random Ordering: As the name suggests, this algorithm generates a random ordering of the vertices in the graph. This can be useful in certain heuristics or sampling techniques where multiple random orderings are tested to find a suitable solution. It can be implemented simply by shuffling the array of vertex indices.

```java
    // Method 3: Uniform Random Ordering:
    public static ArrayList<Integer> uniformrandom(ArrayList<ArrayList<Integer>> am){
        ArrayList<Integer> unOrder = new ArrayList<Integer>();
        for (int i = 0; i < am.size(); i++){
            unOrder.add(i);
        }

        Collections.shuffle(unOrder,new Random());
        // the list to be shuffled
        // the source of randomness to use to shuffle the list.


        System.out.println(unOrder);


        return unOrder;
    }
```

The example test result:

```
----------------------Example 1--------------------------------
The original adjacencyList:
Vertex 0 is connected to: 1 2 3
Vertex 1 is connected to: 0 4 3 2
Vertex 2 is connected to: 0 1 3 4
Vertex 3 is connected to: 0 1 2
Vertex 4 is connected to: 1 2

The Method for graph order result(from right to left):
Uniform random Order is : [2, 4, 1, 3, 0]

The color result:
Vertex 2 --->  Color 0
Vertex 4 --->  Color 1
Vertex 1 --->  Color 2
Vertex 3 --->  Color 1
Vertex 0 --->  Color 3

The running time: 8730083
```

```
----------------------Example 2-------------------------------
The orignial graph:
Vertex 0 is connected to: 1 2
Vertex 1 is connected to: 0 4 3
Vertex 2 is connected to: 0 3
Vertex 3 is connected to: 1 2 4
Vertex 4 is connected to: 1 3

The Method for graph order result(from right to left):
Uniform random Order is : [4, 1, 3, 0, 2]

The color result:
Vertex 4 --->  Color 0
Vertex 1 --->  Color 1
Vertex 3 --->  Color 2
Vertex 0 --->  Color 0
Vertex 2 --->  Color 1

The running time: 149500

----------------------Example 3-------------------------------
The original adjacencyList:
Vertex 0 is connected to: 1 6 7 8
Vertex 1 is connected to: 0 6 2 9 5
Vertex 2 is connected to: 1 9 5 3 4 6
Vertex 3 is connected to: 2 4 9
Vertex 4 is connected to: 2 3 5
Vertex 5 is connected to: 1 2 4 6
Vertex 6 is connected to: 0 1 2 5 7
Vertex 7 is connected to: 0 6
Vertex 8 is connected to: 0
Vertex 9 is connected to: 1 2 3

The Method for graph order result(from right to left):
Uniform random Order is : [3, 1, 0, 7, 9, 2, 5, 6, 8, 4]

The color result:
Vertex 3 --->  Color 0
Vertex 1 --->  Color 0
Vertex 0 --->  Color 1
Vertex 7 --->  Color 0
Vertex 9 --->  Color 1
Vertex 2 --->  Color 2
Vertex 5 --->  Color 1
Vertex 6 --->  Color 3
Vertex 8 --->  Color 0
Vertex 4 --->  Color 3
The running time: 414250
```

4. Breadth-First Search (BFS) Ordering: This algorithm traverses the graph in a breadth-first manner, exploring all the neighbors of a vertex before moving on to their neighbors. The order in which the vertices are visited during the traversal is the BFS ordering. BFS is used in various graph search algorithms, pathfinding, and connectivity testing.[2]

   (a) Let G = (V, E) be a graph with vertex set V and edge set E.

(b) Choose a starting vertex s in V.

(c) Initialize a queue Q and a list P (to store the BFS ordering).

(d) Mark vertex s as visited and enqueue it into Q.

(e) While Q is not empty:

  i. Dequeue a vertex v from Q and add it to the BFS ordering list P.

  ii. For each unvisited neighbor w of vertex v: Mark w as visited and enqueue it into Q.

```java
    // Method 4: Breadth-First Search (BFS) Ordering:
    public static ArrayList<Integer> BFSOrder(ArrayList<ArrayList<Integer>> am, int startNode){
        // Create a queue for bfs;
        ArrayList<Integer> bfsOrder = new ArrayList<>();

        // Make all the vertices as not visited(By default set as false)
        boolean visited[] = new boolean[am.size()];
        Queue<Integer> queue = new LinkedList<>();

        // Mark the current node as visited and enqueue it
        // start from the first one.
        visited[startNode] = true;
        queue.offer(startNode);

        while(!queue.isEmpty()){
            int node = queue.poll();
            bfsOrder.add(node);
            System.out.println(node + " ");

            for (int neighbor : am.get(node)){
                if (!visited[neighbor]){
                    visited[neighbor] = true;
                    queue.offer(neighbor);
                }
            }

        }

        return bfsOrder;
    }
```

The example test result:

```
--------------------Example 1--------------------------------
The original adjacencyList:
Vertex 0 is connected to: 1 2 3
Vertex 1 is connected to: 0 4 3 2
Vertex 2 is connected to: 0 1 3 4
Vertex 3 is connected to: 0 1 2
Vertex 4 is connected to: 1 2

The Method for graph order result(from right to left):
BFSOrder is : [2, 0, 1, 3, 4]

The color result:
Vertex 2 --->  Color 0
Vertex 0 --->  Color 1
Vertex 1 --->  Color 2
Vertex 3 --->  Color 3
Vertex 4 --->  Color 1

The running time: 9816709
----------------------Example 2--------------------------------
The orignial graph:
Vertex 0 is connected to: 1 2
Vertex 1 is connected to: 0 4 3
Vertex 2 is connected to: 0 3
Vertex 3 is connected to: 1 2 4
Vertex 4 is connected to: 1 3

The Method for graph order result(from right to left):
BFSOrder is : [2, 0, 3, 1, 4]

The color result:
Vertex 2 --->  Color 0
Vertex 0 --->  Color 1
Vertex 3 --->  Color 1
Vertex 1 --->  Color 0
Vertex 4 --->  Color 2

The running time: 327042
```

```
----------------------Example 3--------------------------------
The original adjacencyList:
Vertex 0 is connected to: 1 6 7 8
Vertex 1 is connected to: 0 6 2 9 5
Vertex 2 is connected to: 1 9 5 3 4 6
Vertex 3 is connected to: 2 4 9
Vertex 4 is connected to: 2 3 5
Vertex 5 is connected to: 1 2 4 6
Vertex 6 is connected to: 0 1 2 5 7
Vertex 7 is connected to: 0 6
Vertex 8 is connected to: 0
Vertex 9 is connected to: 1 2 3

The Method for graph order result(from right to left):
BFSOrder is : [2, 1, 9, 5, 3, 4, 6, 0, 7, 8]

The color result:
Vertex 2 --->  Color 0
Vertex 1 --->  Color 1
Vertex 9 --->  Color 2
Vertex 5 --->  Color 2
Vertex 3 --->  Color 1
Vertex 4 --->  Color 3
Vertex 6 --->  Color 3
Vertex 0 --->  Color 0
Vertex 7 --->  Color 1
Vertex 8 --->  Color 1
The running time: 193875
```

5. Depth-First Search (DFS) Ordering: This algorithm traverses the graph in a depth-first manner, visiting a vertex and recursively exploring its neighbors as deeply as possible before backtracking. The order in which the vertices are visited during the traversal is the DFS ordering. DFS is used in various graph search algorithms, topological sorting, and cycle detection.[3]

   (a) Let $G = (V, E)$ be a graph with vertex set V and edge set E.

   (b) Let visited(v) be a boolean function that returns whether vertex v in V has been visited or not. Initially, visited(v) = False for all v in V.

   (c) Let P be an empty list that will store the DFS ordering of the vertices.

   (d) For each vertex u in V: If visited(u) = False, call DFS-Visit(G, u, visited, P) The DFS-Visit function is a recursive function that performs the depth-first traversal, so DFS-Visit(G, v, visited, P):

      i. Mark vertex v as visited: visited(v) = True

      ii. For each neighbor w of vertex v: If visited(w) = False, call DFS-Visit(G, w, visited, P)

      iii. Add vertex v to the end of the list P (or to the beginning for topological sorting)

```java
// Method 5: Depth-First Search (DFS) Ordering:
    public static ArrayList<Integer> DFSOrder(ArrayList<ArrayList<Integer>> am, int startNode){
        ArrayList<Integer> dfsOrder = new ArrayList<>();
        boolean[] visited = new boolean[am.size()];

        dfsRecursive(am, startNode, visited, dfsOrder);
        return dfsOrder;
```

```
    }

    private static void dfsRecursive(ArrayList<ArrayList<Integer>> am, int startNode,
                                     boolean[] visited, ArrayList<Integer> dfsOrder){
        visited[startNode] = true;
        dfsOrder.add(startNode);

        for (int neighbor : am.get(startNode)){
            if(!visited[neighbor]){
                dfsRecursive(am,neighbor,visited,dfsOrder);
            }
        }
    }

```

The example test result:

```
----------------------Example 1--------------------------------
The original adjacencyList:
Vertex 0 is connected to: 1 2 3
Vertex 1 is connected to: 0 4 3 2
Vertex 2 is connected to: 0 1 3 4
Vertex 3 is connected to: 0 1 2
Vertex 4 is connected to: 1 2

The Method for graph order result(from right to left):
DFSOrder is : [2, 0, 1, 4, 3]

The color result:
Vertex 2 --->  Color 0
Vertex 0 --->  Color 1
Vertex 1 --->  Color 2
Vertex 4 --->  Color 1
Vertex 3 --->  Color 3

The running time: 7764667
----------------------Example 2--------------------------------
The orignial graph:
Vertex 0 is connected to: 1 2
Vertex 1 is connected to: 0 4 3
Vertex 2 is connected to: 0 3
Vertex 3 is connected to: 1 2 4
Vertex 4 is connected to: 1 3

The Method for graph order result(from right to left):
DFSOrder is : [2, 0, 1, 4, 3]

The color result:
Vertex 2 --->  Color 0
Vertex 0 --->  Color 1
Vertex 1 --->  Color 0
Vertex 4 --->  Color 1
Vertex 3 --->  Color 2

The running time: 154042
```

```
----------------------Example 3--------------------------------
The original adjacencyList:
Vertex 0 is connected to: 1 6 7 8
Vertex 1 is connected to: 0 6 2 9 5
Vertex 2 is connected to: 1 9 5 3 4 6
Vertex 3 is connected to: 2 4 9
Vertex 4 is connected to: 2 3 5
Vertex 5 is connected to: 1 2 4 6
Vertex 6 is connected to: 0 1 2 5 7
Vertex 7 is connected to: 0 6
Vertex 8 is connected to: 0
Vertex 9 is connected to: 1 2 3

The Method for graph order result(from right to left):
DFSOrder is : [2, 1, 0, 6, 5, 4, 3, 9, 7, 8]

The color result:
Vertex 2 --->  Color 0
Vertex 1 --->  Color 1
Vertex 0 --->  Color 0
Vertex 6 --->  Color 2
Vertex 5 --->  Color 3
Vertex 4 --->  Color 1
Vertex 3 --->  Color 2
Vertex 9 --->  Color 3
Vertex 7 --->  Color 1
Vertex 8 --->  Color 1
The running time: 185458
```

6. Lexicographic Breadth-First Search (LexBFS) Ordering: This algorithm is an extension of BFS that maintains a lexicographically ordered list of unvisited vertices. The LexBFS ordering has applications in graph isomorphism testing and chordal graph recognition.[6]

   (a) Let $G = (V, E)$ be a graph with vertex set V and edge set E.

   (b) Let $L(v)$ be a label assigned to vertex v in V. Initially, $L(v) = [\,]$ (an empty list) for all v in V.

   (c) Let Q be a priority queue that stores vertices based on their labels. Vertices with lexicographically larger labels have higher priority.

   (d) While there are unvisited vertices in V:

      i. Initialize an empty set S.

      ii. Find the vertex u with the highest priority in Q and add it to S. Remove u from Q.

      iii. While S is not empty:

         A. Select a vertex v from S and remove it from S.

         B. Visit vertex v, and add it to the ordered list P.

         C. For each unvisited neighbor w of vertex v:

      iv. Append v to $L(w)$ (i.e., $L(w) = L(w) + [v]$).

      v. If w is not in S, add it to S.

   (e) The ordered list P represents the LexBFS ordering of the vertices in the graph G.

```
1  // Method 6: Lexicographic Breadth-First Search (LexBFS) Ordering
2      public static ArrayList<Integer> lexBFS(ArrayList<ArrayList<Integer>> am, int startnode){
```

```
        int n = am.size();
        ArrayList<Integer> lexOrder = new ArrayList<>();


        // Create a mapping from node index to its neighbor set
        Map<Integer, Set<Integer>> nodeToNeighbors = new HashMap<>();
        for (int i = 0; i < n; i++){
            Set<Integer> neighbors = new HashSet<>();
            for (int neighbor:am.get(i)){
                neighbors.add(neighbor);
            }
            nodeToNeighbors.put(i, neighbors);
        }

        // Perform lexicographic BFS starting from node 0
        Queue<Integer> bfsQueue = new LinkedList<>();
        Set<Integer> visited = new HashSet<>();
        bfsQueue.add(startnode);
        visited.add(startnode);

        while(!bfsQueue.isEmpty()){
            int node = bfsQueue.poll();
            lexOrder.add(node);

            ArrayList<Integer> unvisitedNeighbors = new ArrayList<>();
            for (int neihbor: nodeToNeighbors.get(node)){
                if (!visited.contains(neihbor)){
                    unvisitedNeighbors.add(neihbor);
                }
            }
            Collections.sort(unvisitedNeighbors);

            for (int neighbor: unvisitedNeighbors){
                bfsQueue.add(neighbor);
                visited.add(neighbor);
            }
        }

        System.out.println("LexOrder is : " + lexOrder +"\n");


        return lexOrder;
    }

```

The example test result:

```
--------------------Example 1--------------------------------
The original adjacencyList:
Vertex 0 is connected to: 1 2 3
Vertex 1 is connected to: 0 4 3 2
Vertex 2 is connected to: 0 1 3 4
Vertex 3 is connected to: 0 1 2
Vertex 4 is connected to: 1 2

The Method for graph order result(from right to left):
LexOrder is : [2, 0, 1, 3, 4]

----------------------Example 2--------------------------------
The orignial graph:
Vertex 0 is connected to: 1 2
Vertex 1 is connected to: 0 4 3
Vertex 2 is connected to: 0 3
Vertex 3 is connected to: 1 2 4
Vertex 4 is connected to: 1 3

The Method for graph order result(from right to left):
LexOrder is : [2, 0, 3, 1, 4]

The color result:
Vertex 2 --->  Color 0
Vertex 0 --->  Color 1
Vertex 3 --->  Color 1
Vertex 1 --->  Color 0
Vertex 4 --->  Color 2

The running time: 361542
```

```
----------------------Example 3--------------------------------
The original adjacencyList:
Vertex 0 is connected to: 1 6 7 8
Vertex 1 is connected to: 0 6 2 9 5
Vertex 2 is connected to: 1 9 5 3 4 6
Vertex 3 is connected to: 2 4 9
Vertex 4 is connected to: 2 3 5
Vertex 5 is connected to: 1 2 4 6
Vertex 6 is connected to: 0 1 2 5 7
Vertex 7 is connected to: 0 6
Vertex 8 is connected to: 0
Vertex 9 is connected to: 1 2 3

The Method for graph order result(from right to left):
LexOrder is : [2, 1, 3, 4, 5, 6, 9, 0, 7, 8]

The color result:
Vertex 2 --->  Color 0
Vertex 1 --->  Color 1
Vertex 3 --->  Color 1
Vertex 4 --->  Color 2
Vertex 5 --->  Color 3
Vertex 6 --->  Color 2
Vertex 9 --->  Color 2
Vertex 0 --->  Color 0
Vertex 7 --->  Color 1
Vertex 8 --->  Color 1
The running time: 210208
```

3.2 Coloring Algorithm

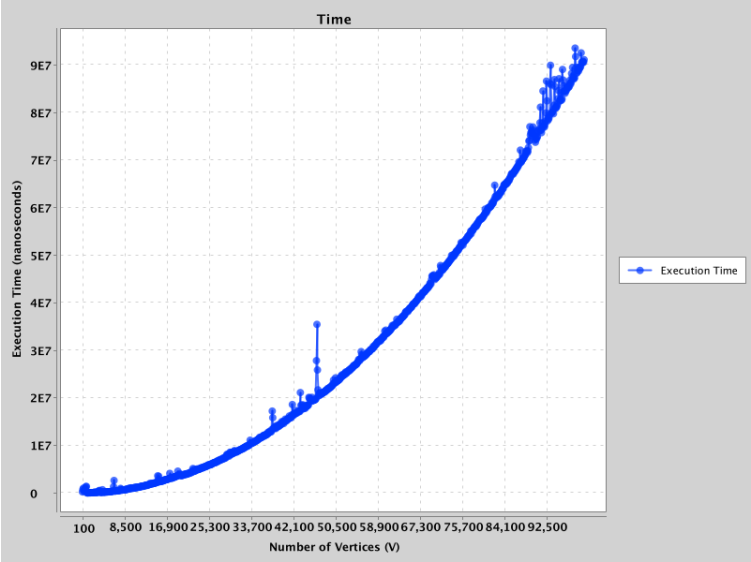
A coloring algorithm on a graph is a method for assigning colors to the vertices of a graph in such a way that no adjacent vertices have the same color. This is known as a vertex coloring or graph coloring.

The code in the following provided appears to be an implementation of a graph coloring algorithm in Java. The algorithm is used to assign colors to vertices in a graph represented as an adjacency matrix (am) based on a specified order (Order) of vertices. The algorithm follows a greedy approach where it iterates over each vertex in the given order and assigns the lowest available color that is not used by its adjacent vertices.[4]

```
1     public static int[] color(ArrayList<ArrayList<Integer>> am, ArrayList<Integer> Order){
2         int[] colorassign = new int[am.size()];
3         for(int i = 0; i < am.size(); i++){
4             colorassign[i] = -1;
5         }
6
7         // The first
8         colorassign[0] = 0;
9
10        // A temporary array to store the available colors. False
11        // value of available[cr] would mean that the color cr is
12        // assigned to one of its adjacent vertices
13        boolean available[] = new boolean[am.size()];
14
15        // Initially, all colors are available
16        for (int i = 0; i < am.size(); i++){
```

```
            available[i] = true;
        }

        // Assign colors to remaining V-1 vertices
        for (int i = 1; i < am.size(); i++){
            // Process all adjacent vertices and flag their colors
            // as unavailable
            Iterator<Integer> it = am.get(Order.get(i)).iterator();
            while (it.hasNext()){
                int k = it.next();
                if (colorassign[Order.indexOf(k)] != -1){
                    available[colorassign[Order.indexOf(k)]] = false;

                }
            }

            // Find the first available color
            int cr;
            for (cr = 0; cr < am.size(); cr++) {
                if (available[cr]) {
                    break;
                }
            }

            // Assign the found color;
            colorassign[i] = cr;

            // Reset the values back to true for the next iteration
            for (int j = 0; j < am.size(); j++) {
                available[j] = true;
            }

        }

        // print the result
        System.out.println("The color result:");
        for (int i = 0; i < am.size(); i++)
            System.out.println("Vertex " + Order.get(i) + " --->  Color "
                    + colorassign[i]);


        return colorassign;
    }
```

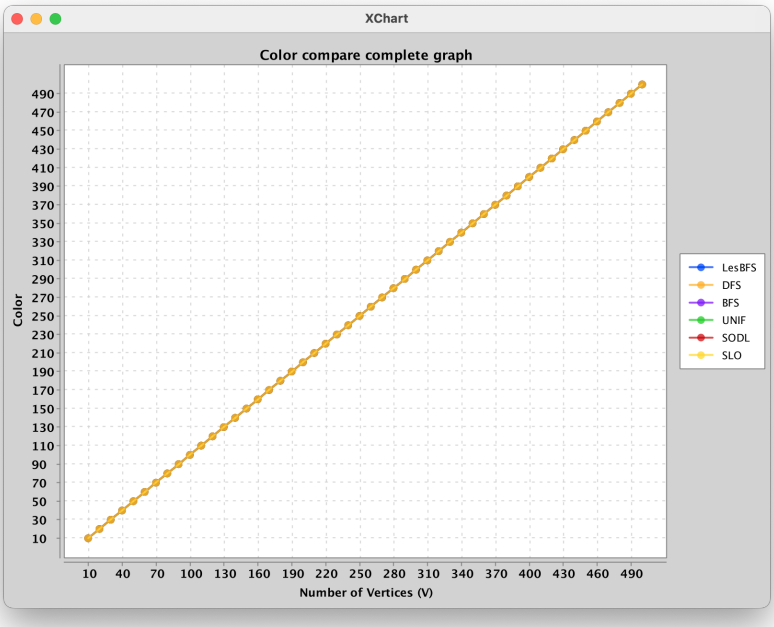
The time complexity of the code is $O(v^2)$.
Use the data from $V = (100, 200, 300, ..., 100000)$:



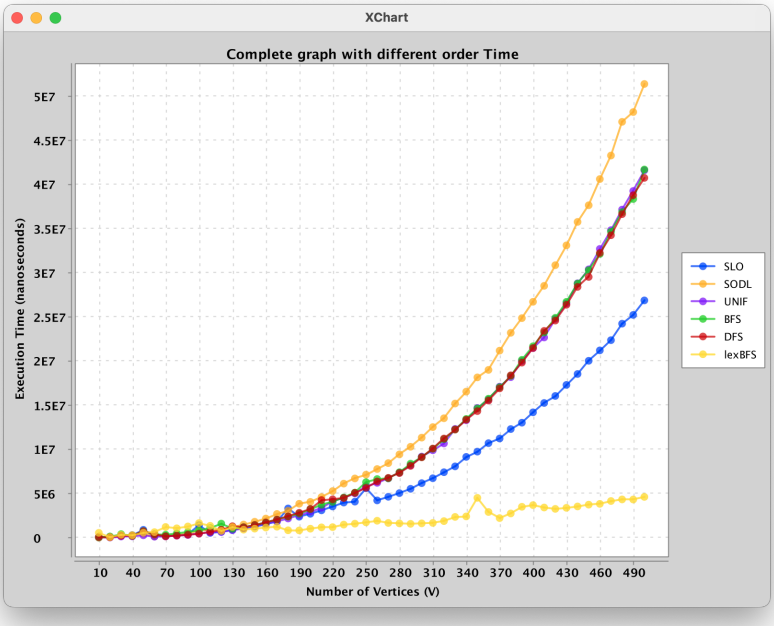
4 Vertex Ordering Capabilities

In this part, I depend on the part one Five different Graph in Adjacency List to compare the different orders effect on coloring. The test vertex is $V = (10, 20, 30, 40, ..., 500)$

4.1 Complete Graph with different order



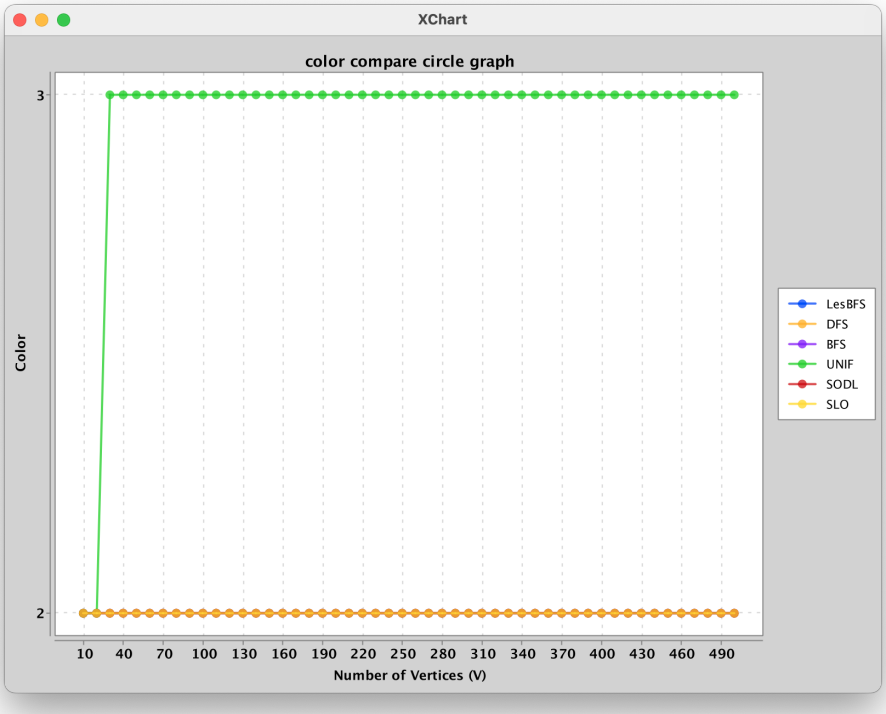

```
/Users/eve/Library/Java/JavaVirtualMachines/openjdk-19.0.1/Contents/Home/bin/java ...
colorsSLOmax: 500
colorsSODLmax: 500
colorsUNIFmax: 500
colorsBFSmax: 500
colorsDFSmax: 500
colorslexBFSmax: 500
```



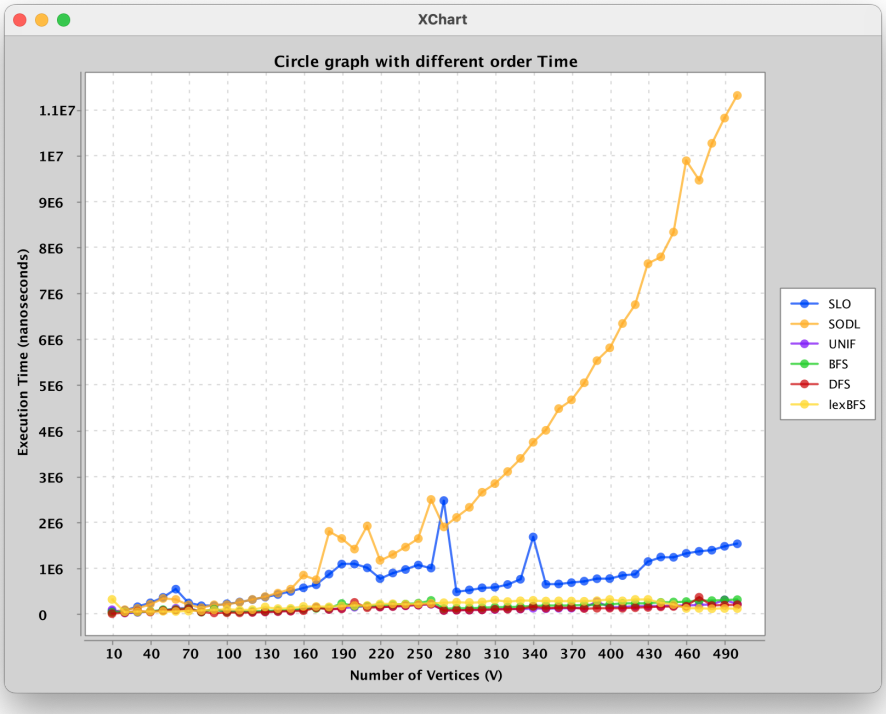
For the complete graph, six different graph ordering Algorithms on coloring performance same. But the running time,

SODL a little longer than other. The BFS, DFS, UNIF almost same. The SLO running time is lower than them. The lexBFS has the shortest running time. In this situation.

4.2 Circle Graph with different order

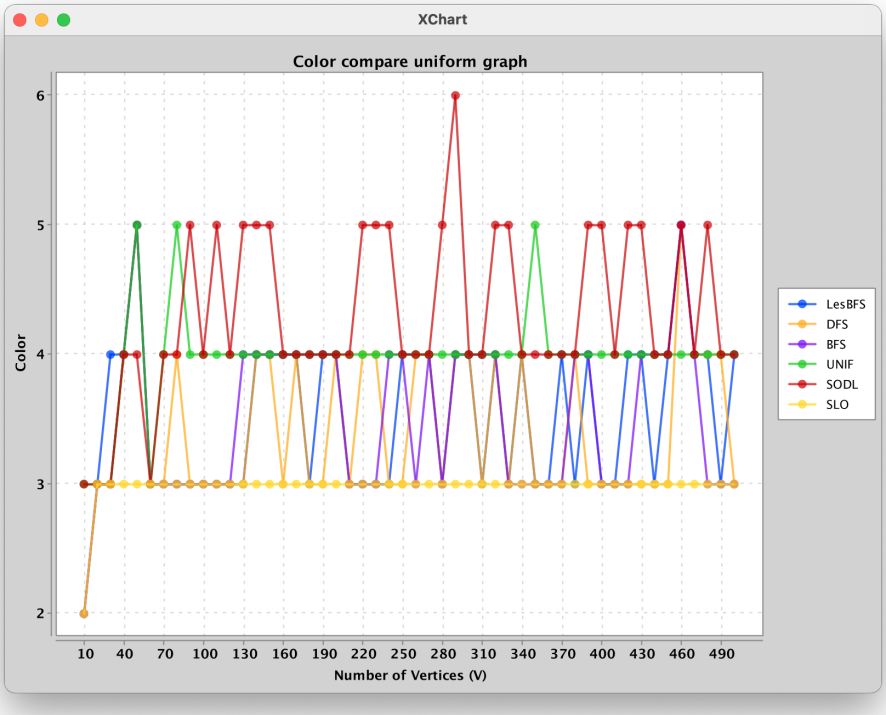



```
/Users/eve/Library/Java/JavaVirtualMachines/openjdk-19.0.1/Contents/Home/bin/java ...
colorsSLOmax: 2
colorsSODLmax: 2
colorsUNIFmax: 3
colorsBFSmax: 2
colorsDFSmax: 2
colorslexBFSmax: 2
```

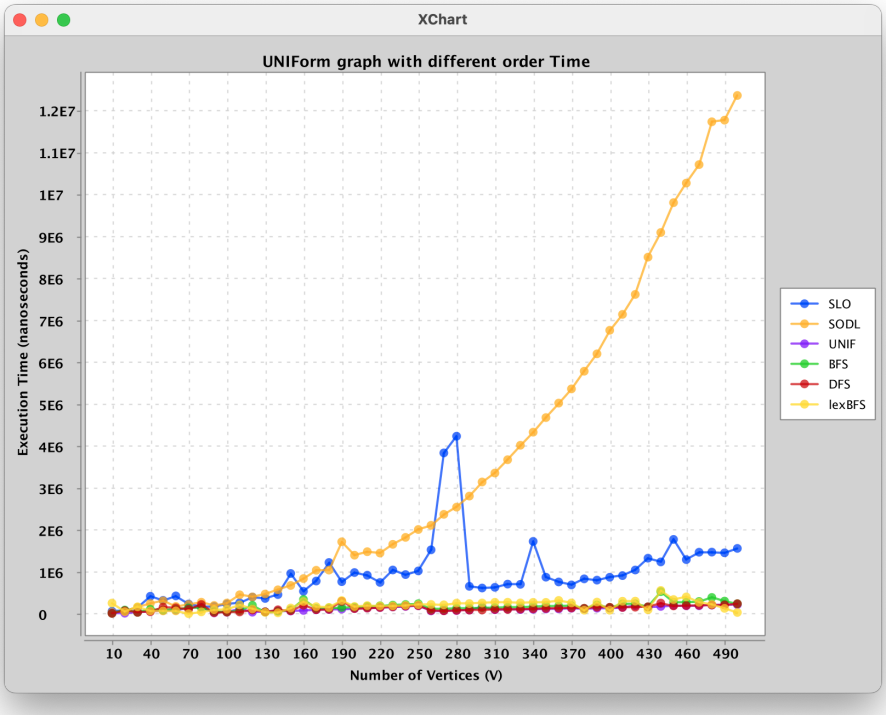


For the Circle Graph, the UNIF order on coloring performance instability. It might use more colors than others. And the running time, the SODL still is the worst, and then SLO. The UNIF, BFS, DFS, lexBFS performance almost same in this situation. They performance well.

4.3 Uniform Graph with different order

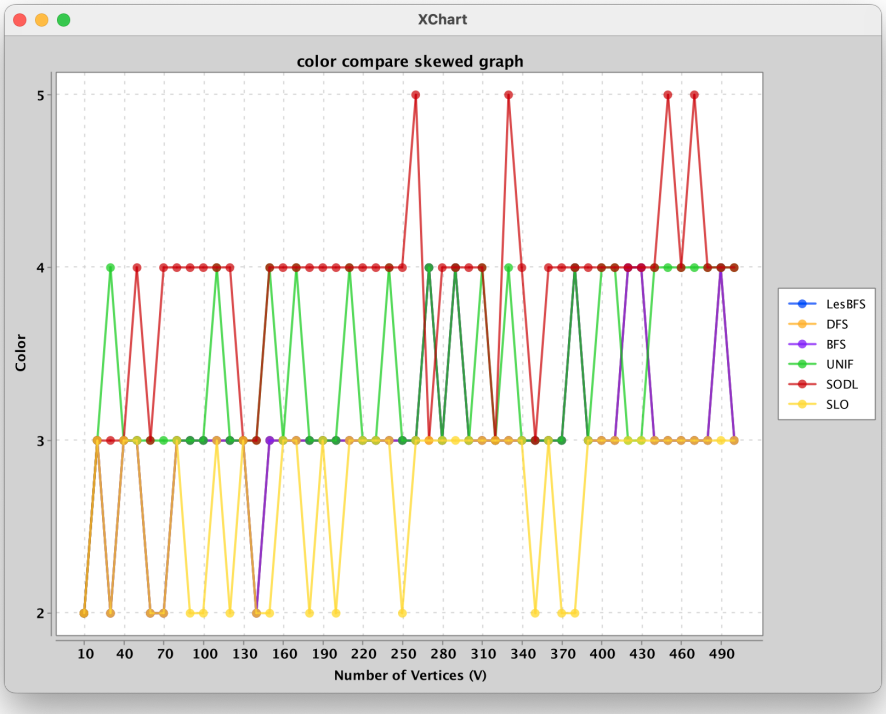



```
/Users/eve/Library/Java/JavaVirtualMachines/openjdk-19.0.1/Contents/Home/bin/java ...
colorsSLOmax: 3
colorsSODLmax: 6
colorsUNIFmax: 5
colorsBFSmax: 5
colorsDFSmax: 5
colorslexBFSmax: 5
```

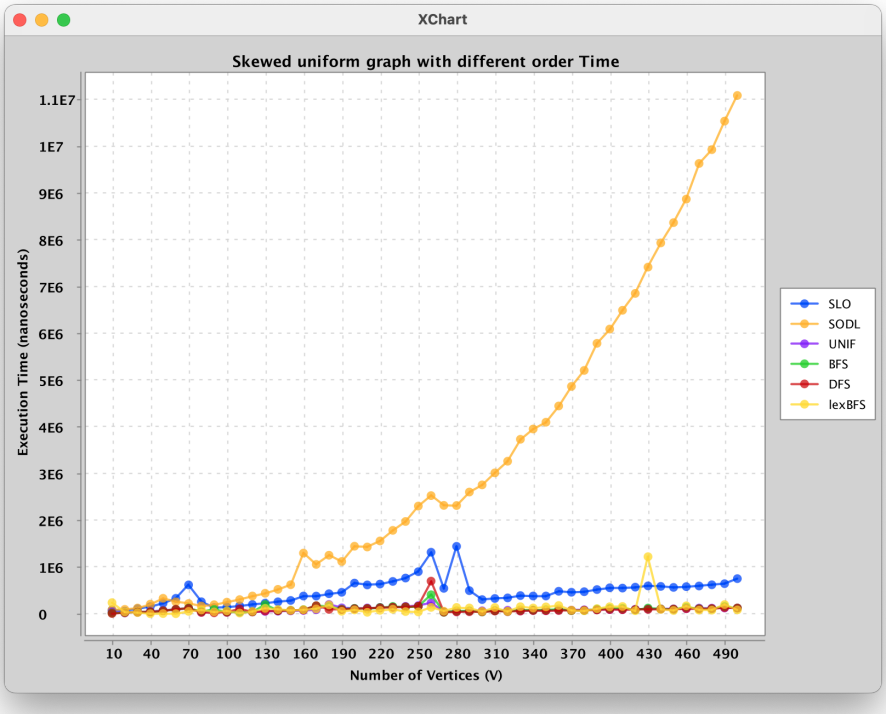


For the Uniform Graph, the SLO will use the less color 3. The SODL will use the most colors 6. Others will use 5 colors. And the running time the SODL still performance the worst. And then SLO. UNIF, BFS, DFS, lexBFS running time performance well.

4.4 Skewed Graph with different order

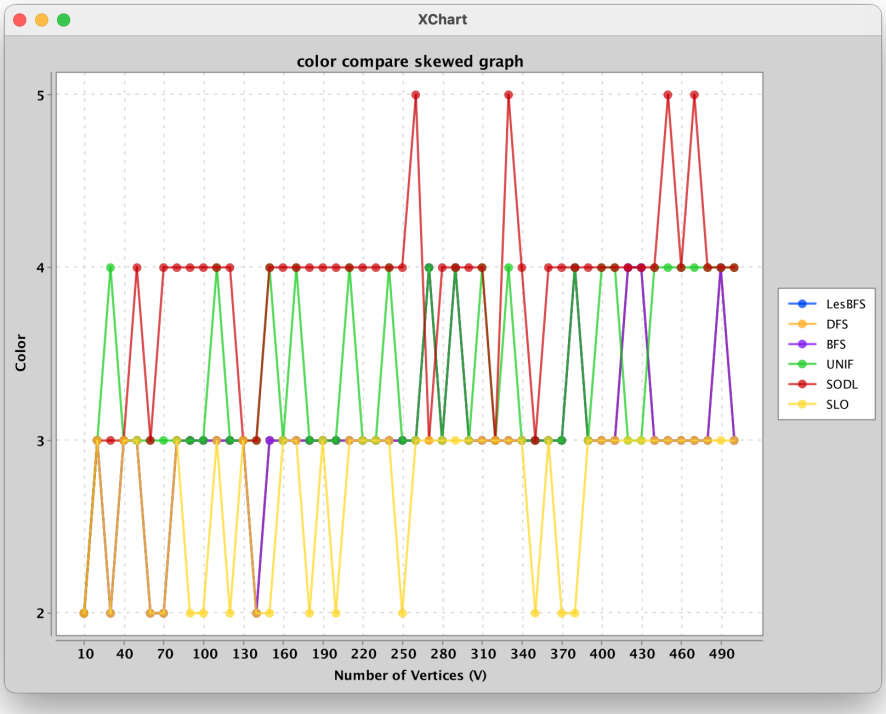



```
/Users/eve/Library/Java/JavaVirtualMachines/openjdk-19.0.1/Contents/Home/bin/java ...
colorsSLOmax: 3
colorsSODLmax: 5
colorsUNIFmax: 4
colorsBFSmax: 4
colorsDFSmax: 4
colorslexBFSmax: 4
```

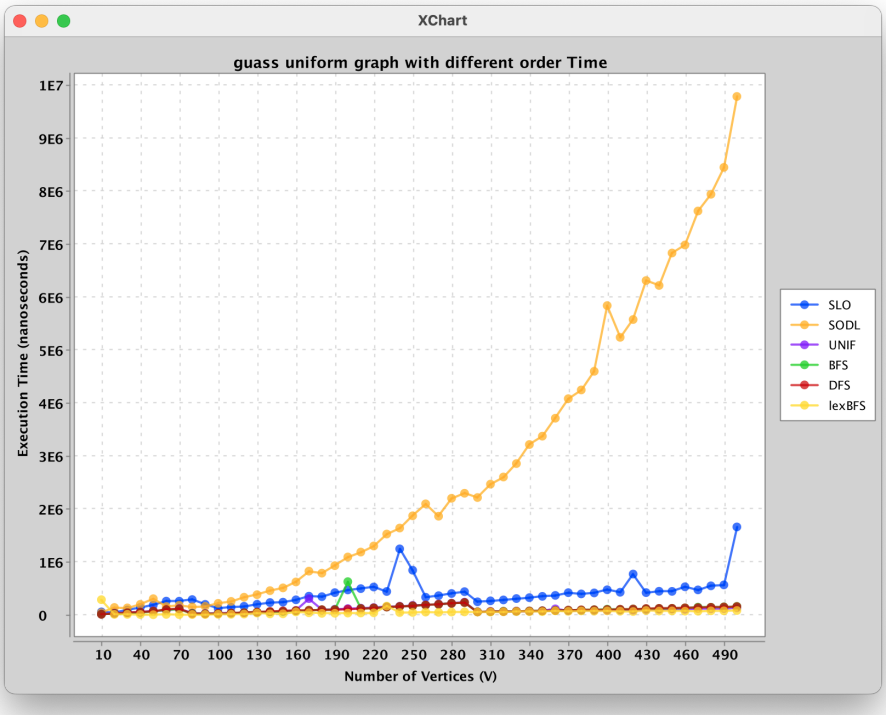


For the Skewed Graph, the SLO will use the less color 3. The SODL will use the most colors 5. Others will use 4 colors. And the running time the SODL still performance the worst. And then SLO. UNIF, BFS, DFS, lexBFS running time performance well.

4.5 Gauss Graph with different order




```
/Users/eve/Library/Java/JavaVirtualMachines/openjdk-19.0.1/Contents/Home/bin/java ...
colorsSLOmax: 3
colorsSODLmax: 5
colorsUNIFmax: 4
colorsBFSmax: 4
colorsDFSmax: 4
colorslexBFSmax: 4
```



For the Gauss Graph, the SLO will use the less color 3. The SODL will use the most colors 5. Others will use 4 colors. And the running time the SODL still performance the worst. And then SLO. UNIF, BFS, DFS, lexBFS running time performance well.

4.6 Conclusion

In different Graphs to color: The Complete colors needs the most colors to color. And then Circle Graph use the less colors to color. The uniform, skewed, Gauss need same colors.
In different Order to color: The SLO can use the less colors to order, but the performance is normal compare others. The UNIF use more colors in Circle Graph but sometimes same as others. The SODL might use more colors than others and the running time performance is worst. The DFS, BFS, LexBFS performance almost same.

5 Code

PartOne.java: This file include the different distribute graph.[5]

```
1  import org.jfree.chart.ChartFactory;
2  import org.jfree.chart.ChartFrame;
3  import org.jfree.chart.JFreeChart;
4  import org.jfree.chart.plot.PlotOrientation;
5  import org.jfree.data.category.DefaultCategoryDataset;
6
7  import java.util.*;
8  import java.util.function.Supplier;
9  import java.util.stream.Stream;
10
11 public class PartOne {
```

```
12      public static class Graph {
13
14          // Add edge
15          static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
16              am.get(s).add(d);  // for a directed graph with an edge pointing from s d
17              am.get(d).add(s);
18          }
19      }
20
21      public static class Complete {
22          public static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
23              am.get(s).add(d);
24              am.get(d).add(s);
25          }
26
27          public static void allEdge(ArrayList<ArrayList<Integer>> am, int V) {
28              for (int i = 0; i < V; i++) {
29                  for (int j = i + 1; j < V; j++) {
30                      addEdge(am, i, j);
31                  }
32              }
33          }
34      }
35
36      static class Cycle {
37          //private static long time;
38          static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
39              am.get(s).add(d);
40              am.get(d).add(s);
41          }
42          static void allEdgeCycle(ArrayList<ArrayList<Integer>> am, int V) {
43              //long startTime = System.nanoTime();
44              for (int i = 0; i < V; i++) {
45                  if (i == V - 1) {
46                      addEdge(am, i, 0);   // head -- tail
47                  } else {
48                      addEdge(am, i, i + 1);
49                  }
50              }
51              //long endTime = System.nanoTime();
52              //time = endTime - startTime;
53          }
54      }
55
56      //    static class Random{
57  //
58  //    }
59      // DIST
60      static class Uniform {
61
62          static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
63              am.get(s).add(d);
64              am.get(d).add(s);
65          }
66
67          // If you provide an integer parameter to "nextInt",
68          // it will return an integer from a uniform distribution between 0 and one less than the parameter.
```

```
69          static void uniformRandom(ArrayList<ArrayList<Integer>> am, int v, int e) {
70              //Random rand = new Random();
71              while (e > 0) {
72                  int source = (int) (Math.random() * v); // Printing the random number between [0,v-1]
73                  int dest = (int) (Math.random() * v);
74                  // edge exit?
75                  if (source == dest || am.get(source).contains(dest)) {
76                      continue;
77                  } else {
78                      addEdge(am, source, dest);
79                      e--;
80                  }
81              }
82          }
83
84          // ------------------------ graph----------------------------
85          // if want to see the graph please remove the annotation
86
87          private int ySize;
88          private int dataNumber;
89          private int xSize;
90          ArrayList<Double> list = new ArrayList<>();
91          Map<Integer, Integer> map = new HashMap<>();
92
93
94          public void uniformDistribution(int xSize, int ySize, int dataNumber) {
95              //int ySize = 8;
96              //int xSize = 12;
97              //int dataNumber = v; //
98              this.ySize = ySize > 3 ? ySize : 3;
99              this.xSize = xSize > 3 ? xSize : 3;
100             this.dataNumber = dataNumber > 1000 ? dataNumber : 1000;
101             init();
102         }
103
104         private void init() {
105             for (int i = 0; i < dataNumber; i++) {
106                 list.add( Math.random() * dataNumber);
107             }
108
109             for (int i = 1; i <= ySize; i++) {
110                 map.put(i, 0);
111             }
112         }
113
114         public void analysis() {
115
116             Supplier<Stream<Double>> supp = () -> list.stream();
117
118             Comparator<Double> comp = (e1, e2) -> e1 > e2 ? 1 : -1;
119
120             double max = supp.get().max(comp).get();
121             double min = supp.get().min(comp).get();
122             double range = (max - min) / this.ySize;
123
124             for (int i = 1; i <= this.ySize; i++) {
125                 double start = min + (i - 1) * range;
```

```
126                 double end = min + i * range;
127                 Stream<Double> stream = supp.get()
128                         .filter((e) -> e >= start).filter((e) -> e < end);
129                 map.put(i, (int) stream.count());
130             }
131         }
132 
133         public void grawValue() {
134             int ScaleSize = 14;
135             int avgScale = this.dataNumber / xSize;
136             int printSize = ScaleSize - String.valueOf(avgScale).length();
137 
138             for (int i = 0; i <= xSize; i++) {
139                 printChar(' ', printSize);
140                 System.out.print(i * avgScale);
141             }
142             System.out.println("");
143             for (int i = 0; i <= xSize; i++) {
144                 if (i == 0) {
145                     printChar(' ', printSize);
146                 } else {
147                     printChar('-', ScaleSize);
148                 }
149             }
150             System.out.println();
151 
152             for (int i = 1; i <= ySize; i++) {
153                 printChar(' ', printSize - 1 - String.valueOf(i).length());
154                 System.out.print(i + ":");
155                 int scaleValue = map.get(i);
156                 double grawSize = scaleValue / (avgScale * 1.0 / ScaleSize);
157                 grawSize = (grawSize > 0 && grawSize < 1) ? 1 : grawSize;
158                 printChar('', (int) grawSize);
159                 System.out.println(" " + scaleValue + "\n");
160             }
161 
162 
163         }
164         // ----------------------------------------------------------------------
165     }
166 
167     static class Skewed {
168         static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
169             am.get(s).add(d);
170             am.get(d).add(s);
171         }
172 
173         static void skewedRandom(ArrayList<ArrayList<Integer>> am, int v, int e) {
174 //            Random rand = new Random();
175             int source = -1;
176             int dest = -1;
177             int a = 0;
178             int b = v;
179             int c = 0;
180             double F = (c - a) / (b - a);//
181             while (e > 0) {
182                 double rand = Math.random();
```

```
183                 if (rand < F) {
184                     source = (int) (a + Math.sqrt(rand * (b - a) * (c - a)));
185                 } else {
186                     source = (int) (b - Math.sqrt((1 - rand) * (b - a) * (b - c)));
187                 }
188 
189                 double rand2 = Math.random();
190                 if (rand2 < F) {
191                     dest = (int) (a + Math.sqrt(rand2 * (b - a) * (c - a)));
192                 } else {
193                     dest = (int) (b - Math.sqrt((1 - rand2) * (b - a) * (b - c)));
194                 }
195 
196                 if (source == dest || am.get(source).contains(dest)) {
197                     continue;
198                 } else {
199                     addEdge(am, source, dest);
200                     e--;
201                 }
202             }
203 
204         }
205 
206         // ------------------------ graph-----------------------------
207         // if want to see the graph please remove the annotation
208 
209         // draw
210         private int ySize;
211         private int dataNumber;
212         private int xSize;
213         ArrayList<Double> list = new ArrayList<>(); //
214         Map<Integer, Integer> map = new HashMap<>(); //
215 
216 
217         public void skewedDistribution(int xSize, int ySize, int dataNumber) {
218             //int ySize = 8;
219             //int xSize = 12;
220             //int dataNumber = v; //
221             this.ySize = ySize > 3 ? ySize : 3;
222             this.xSize = xSize > 3 ? xSize : 3;
223             this.dataNumber = dataNumber > 1000 ? dataNumber : 1000;
224             init();
225         }
226 
227         private void init() {
228             double source;
229             int a = 0;
230             int b = dataNumber;
231             int c = 0;
232             double F = (c - a) / (b - a);
233 
234             for (int i = 0; i < dataNumber; i++) {
235                 double rand = Math.random();
236                 if (rand < F) {
237                     source = (a + Math.sqrt(rand * (b - a) * (c - a)));
238                 } else {
239                     source = (b - Math.sqrt((1 - rand) * (b - a) * (b - c)));
```

```
240                 }
241                 list.add(source);
242             }
243 
244             for (int i = 1; i <= ySize; i++) {
245                 map.put(i, 0);
246             }
247         }
248 
249         public void analysis() {
250 
251             Supplier<Stream<Double>> supp = () -> list.stream();
252 
253             Comparator<Double> comp = (e1, e2) -> e1 > e2 ? 1 : -1;
254 
255             double max = supp.get().max(comp).get();
256             double min = supp.get().min(comp).get();
257             double range = (max - min) / this.ySize;
258 
259             for (int i = 1; i <= this.ySize; i++) {
260                 double start = min + (i - 1) * range;
261                 double end = min + i * range;
262                 Stream<Double> stream = supp.get()
263                         .filter((e) -> e >= start).filter((e) -> e < end);
264                 map.put(i, (int) stream.count());
265             }
266         }
267 
268         public void grawValue() {
269             int ScaleSize = 14;
270             int avgScale = this.dataNumber / xSize;
271             int printSize = ScaleSize - String.valueOf(avgScale).length();
272 
273             for (int i = 0; i <= xSize; i++) {
274                 printChar(' ', printSize);
275                 System.out.print(i * avgScale);
276             }
277             System.out.println("");
278             for (int i = 0; i <= xSize; i++) {
279                 if (i == 0) {
280                     printChar(' ', printSize);
281                 } else {
282                     printChar('-', ScaleSize);
283                 }
284             }
285             System.out.println();
286 
287             for (int i = 1; i <= ySize; i++) {
288                 printChar(' ', printSize - 1 - String.valueOf(i).length());
289                 System.out.print(i + ":");
290                 int scaleValue = map.get(i);
291                 double grawSize = scaleValue / (avgScale * 1.0 / ScaleSize);
292                 grawSize = (grawSize > 0 && grawSize < 1) ? 1 : grawSize;
293                 printChar('', (int) grawSize);
294                 System.out.println(" " + scaleValue + "\n");
295             }
296 
```

```
297         }
298         //-----------------------------------------------------------------------------------
299     }
300 
301 
302     // YOURS
303     static class Gauss {
304 
305         static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
306             am.get(s).add(d);
307             am.get(d).add(s);
308         }
309 
310         //Gauss
311         static void gaussRandom(ArrayList<ArrayList<Integer>> am, int v, int e) {
312             Random rand = new Random();
313 
314             while (e > 0) {
315 
316                 int source = (int) (v / 10 * rand.nextGaussian() + v / 2);
317                 int dest = (int) (v / 10 * rand.nextGaussian() + v / 2);
318 
319                 if (source == dest || am.get(source).contains(dest)) {
320                     continue;
321                 } else {
322                     addEdge(am, source, dest);
323                     e--;
324                 }
325             }
326         }
327 
328 
329         // ------------------------ graph----------------------------
330         // if want to see the graph please remove the annotation
331 
332         private int ySize;
333         private int dataNumber;
334         private int xSize;
335         ArrayList<Double> list = new ArrayList<>();
336         Map<Integer, Integer> map = new HashMap<>();
337 
338         public void gaussDistribution(int xSize, int ySize, int dataNumber) {
339             //int ySize = 8;
340             //int xSize = 12;
341             //int dataNumber = v; //
342             this.ySize = ySize > 3 ? ySize : 3;
343             this.xSize = xSize > 3 ? xSize : 3;
344             this.dataNumber = dataNumber > 1000 ? dataNumber : 1000;
345             init();
346         }
347 
348         private void init() {
349             Random rand = new Random();
350             for (int i = 0; i < dataNumber; i++) {
351                 list.add(dataNumber/10 * rand.nextGaussian() +dataNumber/2);
352             }
353 
```

```
354             for (int i = 1; i <= ySize; i++) {
355                 map.put(i, 0);
356             }
357         }
358 
359         public void analysis() {
360 
361             Supplier<Stream<Double>> supp = () -> list.stream();
362             Comparator<Double> comp = (e1, e2) -> e1 > e2 ? 1 : -1;
363             double max = supp.get().max(comp).get();
364             double min = supp.get().min(comp).get();
365             double range = (max - min) / this.ySize;
366             for (int i = 1; i <= this.ySize; i++) {
367                 double start = min + (i - 1) * range;
368                 double end = min + i * range;
369                 Stream<Double> stream = supp.get()
370                         .filter((e) -> e >= start).filter((e) -> e < end);
371                 map.put(i, (int) stream.count());
372             }
373         }
374 
375         public void grawValue() {
376             int ScaleSize = 14;
377             int avgScale = this.dataNumber / xSize;
378             int printSize = ScaleSize - String.valueOf(avgScale).length();
379             for (int i = 0; i <= xSize; i++) {
380                 printChar(' ', printSize);
381                 System.out.print(i * avgScale);
382             }
383             System.out.println("");
384             for (int i = 0; i <= xSize; i++) {
385                 if (i == 0) {
386                     printChar(' ', printSize);
387                 } else {
388                     printChar('-', ScaleSize);
389                 }
390             }
391             System.out.println();
392             for (int i = 1; i <= ySize; i++) {
393                 printChar(' ', printSize - 1 - String.valueOf(i).length());
394                 System.out.print(i + ":");
395                 int scaleValue = map.get(i);
396                 double grawSize = scaleValue / (avgScale * 1.0 / ScaleSize);
397                 grawSize = (grawSize > 0 && grawSize < 1) ? 1 : grawSize;
398                 printChar('', (int) grawSize);
399                 System.out.println(" " + scaleValue + "\n");
400             }
401 
402 
403         }
404         // ------------------------------------------------------------------------
405 
406     }
407 
408     public static void printChar(char c, int number) {
409         for (int i = 0; i < number; i++) {
410             System.out.print(c);
```

```
411             }
412         }
413 
414 
415 
416 
417 
418         public static void printGraph(ArrayList<ArrayList<Integer>> am) {
419             for (int i = 0; i < am.size(); i++) {
420                 System.out.print("\nVertex " + i + ":");
421                 for (int j = 0; j < am.get(i).size(); j++) {
422                     System.out.print(" -> " + am.get(i).get(j));
423                 }
424                 System.out.println();
425             }
426         }
427 
428         public static void input(int V, int E, int G) {
429             // V: Number of vertices. (Max 10,000)
430             // E: Number of conflicts between pairs of vertices for random graphs. (MAX - 2,000,000)
431             // G: COMPTLETE| CYCLE | RANDOM (with DIST below)
432             // DIST = UNIFORM | SKEWED |YOURS
433             // 1: COMPTLETE
434             // 2: CYCLE
435             // 3: UNIFORM
436             // 4: SKEWED
437             // 5: YOURS
438 
439             if (G == 1) {
440                 comptele(V);
441             } else if (G == 2) {
442                 cycle(V);
443             } else if (G == 3) {
444                 uniform(V, E);
445             } else if (G == 4) {
446                 skewed(V, E);
447             } else if (G == 5) {
448                 gauss(V, E);
449             }
450         }
451 
452         public static void comptele(int v) {
453             //System.out.println("Comptele graph:");
454             ArrayList<ArrayList<Integer>> completegraph = new ArrayList<ArrayList<Integer>>(v);
455             for (int i = 0; i < v; i++) {
456                 completegraph.add((new ArrayList<Integer>()));
457             }
458 
459             Complete.allEdge(completegraph, v);
460             //printGraph(completegraph);
461         }
462 
463         public static void cycle(int v) {
464             System.out.println("Cycle graph:");
465             ArrayList<ArrayList<Integer>> cyclegraph = new ArrayList<ArrayList<Integer>>(v);
466             for (int i = 0; i < v; i++) {
467                 cyclegraph.add(new ArrayList<Integer>());
```

```
468             }
469             Cycle.allEdgeCycle(cyclegraph, v);
470 
471             printGraph(cyclegraph);
472 
473 
474         }
475 
476         public static void uniform(int v, int e) {
477             System.out.println("Uniform graph:");
478             ArrayList<ArrayList<Integer>> uniformgraph = new ArrayList<ArrayList<Integer>>(v);
479             for (int i = 0; i < v; i++) {
480                 uniformgraph.add(new ArrayList<>());
481             }
482             Uniform.uniformRandom(uniformgraph, v, e);
483             printGraph(uniformgraph);
484 
485             //Draw picture
486             Uniform u = new Uniform();
487             u.uniformDistribution(8,12,10000);
488             u.analysis();
489             u.grawValue();
490 
491         }
492 
493         public static void skewed(int v, int e) {
494             System.out.println("Skewed graph:");
495             ArrayList<ArrayList<Integer>> skewedgraph = new ArrayList<ArrayList<Integer>>(v);
496             for (int i = 0; i < v; i++) {
497                 skewedgraph.add(new ArrayList<>());
498             }
499             Skewed.skewedRandom(skewedgraph, v, e);
500             printGraph(skewedgraph);
501 
502 //          Draw picture
503             Skewed s = new Skewed();
504             s.skewedDistribution(8, 12, 10000);
505             s.analysis();
506             s.grawValue();
507         }
508 
509         public static void gauss(int v, int e) {
510             System.out.println("Gauss graph:");
511             ArrayList<ArrayList<Integer>> gaussgraph = new ArrayList<ArrayList<Integer>>(v);
512             for (int i = 0; i < v; i++) {
513                 gaussgraph.add(new ArrayList<>());
514             }
515 
516             Gauss.gaussRandom(gaussgraph, v, e);
517             printGraph(gaussgraph);
518 
519 
520             // draw picture
521             Gauss g = new Gauss();
522             g.gaussDistribution(8, 12, 10000);
523             g.analysis();
524             g.grawValue();
```

```
525
526         }
527
528
529         public static long time_calculate(int V, int G, int E) {
530             long time = 0;
531             if (G == 1) {
532                 long startTime = System.nanoTime();
533                 comptele(V);
534                 long endTime = System.nanoTime();
535                 time = endTime - startTime;
536
537             } else if (G == 2) {
538                 long startTime = System.nanoTime();
539                 cycle(V);
540                 long endTime = System.nanoTime();
541                 time = endTime - startTime;
542
543             } else if (G == 3) {
544                 long startTime = System.nanoTime();
545                 uniform(V, E);
546                 long endTime = System.nanoTime();
547                 time = endTime - startTime;
548
549             } else if (G == 4) {
550                 long startTime = System.nanoTime();
551                 skewed(V, E);
552                 long endTime = System.nanoTime();
553                 time = endTime - startTime;
554
555             } else if (G == 5) {
556                 long startTime = System.nanoTime();
557                 gauss(V, E);
558                 long endTime = System.nanoTime();
559                 time = endTime - startTime;
560             }
561             return time;
562
563         }
564
565         public static void time(int G){
566             int n0 = 1000;
567             int n1 = 2000;
568             int n2 = 3000;
569             int n3 = 4000;
570             int n4 = 5000;
571             int n5 = 6000;
572             int n6 = 7000;
573             int n7 = 8000;
574             int n8 = 9000;
575             int n9 = 10000;
576
577             long[] result = new long[10];
578             result[0] = time_calculate(n0,G,n0-1);
579             result[1] = time_calculate(n1, G, n1-1);
580             result[2] = time_calculate(n2,G,n2-1);
581             result[3] = time_calculate(n3, G, n3-1);
```

```
582             result[4] = time_calculate(n4, G, n4-1);
583             result[5] = time_calculate(n5,G,n5-1);
584             result[6] = time_calculate(n6,G,n6-1);
585             result[7] = time_calculate(n7,G,n7-1);
586             result[8] = time_calculate(n8,G,n8-1);
587             result[9] = time_calculate(n9,G,n9-1);
588 
589             System.out.println(Arrays.toString(result));
590 
591             DefaultCategoryDataset dataset = new DefaultCategoryDataset();
592             dataset.addValue(result[0],"time", "1000");
593             dataset.addValue(result[1],"time", "2000");
594             dataset.addValue(result[2],"time", "3000");
595             dataset.addValue(result[3],"time", "4000");
596             dataset.addValue(result[4],"time", "5000");
597             dataset.addValue(result[5],"time", "6000");
598             dataset.addValue(result[6],"time", "7000");
599             dataset.addValue(result[7],"time", "8000");
600             dataset.addValue(result[8],"time", "9000");
601             dataset.addValue(result[9],"time", "10000");
602 
603             JFreeChart chart = ChartFactory.createLineChart(
604                     "Result",
605                     "n",
606                     "time(ns)",
607                     dataset,
608                     PlotOrientation.VERTICAL,
609                     false,true, false
610             );
611 
612             ChartFrame chartFrame = new ChartFrame("Test", chart);
613             chartFrame.pack();;
614             chartFrame.setVisible(true);
615 
616         }
617 
618         public static void main(String[] args) {
619             // V: Number of vertices. (Max 10,000)
620             // E: Number of conflicts between pairs of vertices for random graphs. (MAX - 2,000,000)
621             // G: COMPTLETE| CYCLE | RANDOM (with DIST below)
622             // DIST = UNIFORM | SKEWED |YOURS
623             // 1: COMPTLETE
624             // 2: CYCLE
625             // 3: UNIFORM
626             // 4: SKEWED
627             // 5: YOURS
628     //
629             int v = 10;
630             int e = 20;
631 
632             input(v, e, 1);
633     //
634             input(v, e, 2);
635     //
636             input(v, e, 3);
637             input(v, e, 4);
638             input(v, e, 5);
```

```
639
640
641 //         //Draw picture
642         Uniform u = new Uniform();
643         u.uniformDistribution(8,12,10000);
644         u.analysis();
645         u.grawValue();
646
647         Skewed s = new Skewed();
648         s.skewedDistribution(8, 12, 10000);
649         s.analysis();
650         s.grawValue();
651
652         Gauss g = new Gauss();
653         g.gaussDistribution(8, 12, 10000);
654         g.analysis();
655         g.grawValue();
656
657     }
658 }
```

time1.java:

```
1  import org.knowm.xchart.*;
2  import org.knowm.xchart.style.markers.SeriesMarkers;
3
4  import java.util.ArrayList;
5  import java.util.Arrays;
6  import java.util.List;
7  import java.util.Random;
8
9  public class Time1 {
10     public static class Complete {
11         public static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
12             am.get(s).add(d);
13             am.get(d).add(s);
14         }
15
16         public static void allEdge(ArrayList<ArrayList<Integer>> am, int V) {
17             for (int i = 0; i < V; i++) {
18                 for (int j = i + 1; j < V; j++) {
19                     addEdge(am, i, j);
20                 }
21             }
22         }
23     }
24
25     static class Cycle {
26         //private static long time;
27         static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
28             am.get(s).add(d);
29             am.get(d).add(s);
30         }
31         static void allEdgeCycle(ArrayList<ArrayList<Integer>> am, int V) {
32
33             for (int i = 0; i < V; i++) {
```

```
                if (i == V - 1) {
                    addEdge(am, i, 0);   // head -- tail
                } else {
                    addEdge(am, i, i + 1);
                }
            }

        }
    }

    static class Uniform {
        static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
            am.get(s).add(d);
            am.get(d).add(s);
        }

        // If you provide an integer parameter to "nextInt",
        // it will return an integer from a uniform distribution between 0 and one less than the parameter.
        static void uniformRandom(ArrayList<ArrayList<Integer>> am, int v, int e) {
            //Random rand = new Random();
            while (e > 0) {
                int source = (int) (Math.random() * v); // Printing the random number between [0,v-1]
                int dest = (int) (Math.random() * v);
                // edge exit?
                if (source == dest || am.get(source).contains(dest)) {
                    continue;
                } else {
                    addEdge(am, source, dest);
                    e--;
                }
            }
        }
    }

    static class Skewed {
        static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
            am.get(s).add(d);
            am.get(d).add(s);
        }

        static void skewedRandom(ArrayList<ArrayList<Integer>> am, int v, int e) {
//            Random rand = new Random();
            int source = -1;
            int dest = -1;
            int a = 0;
            int b = v;
            int c = 0;
            double F = (c - a) / (b - a);//
            while (e > 0) {
                double rand = Math.random();
                if (rand < F) {
                    source = (int) (a + Math.sqrt(rand * (b - a) * (c - a)));
                } else {
                    source = (int) (b - Math.sqrt((1 - rand) * (b - a) * (b - c)));
                }

                double rand2 = Math.random();
```

```
                if (rand2 < F) {
                    dest = (int) (a + Math.sqrt(rand2 * (b - a) * (c - a)));
                } else {
                    dest = (int) (b - Math.sqrt((1 - rand2) * (b - a) * (b - c)));
                }

                if (source == dest || am.get(source).contains(dest)) {
                    continue;
                } else {
                    addEdge(am, source, dest);
                    e--;
                }
            }

        }
    }

    static class Gauss {
        static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
            am.get(s).add(d);
            am.get(d).add(s);
        }

        //Gauss
        static void gaussRandom(ArrayList<ArrayList<Integer>> am, int v, int e) {
            Random rand = new Random();
            while (e > 0) {

                int source = (int) (v / 10 * rand.nextGaussian() + v / 2);
                int dest = (int) (v / 10 * rand.nextGaussian() + v / 2);
//                source = Math.abs(source);
//                dest = Math.abs(dest);
                try{
                    if (source == dest || am.get(source).contains(dest)) {
                        continue;
                    } else {
                        addEdge(am, source, dest);
                        e--;
                    }
                }catch (Exception ignored){}

            }
        }
    }

    private static void plotResults(int[] vertexCounts, long[] executionTimes) {
        XYChart chart = new XYChartBuilder()
                .width(800)
                .height(600)
                .title("Time")
                .xAxisTitle("Number of Vertices (V)")
                .yAxisTitle("Execution Time (nanoseconds)")
                .build();

        int[] executionTimecopy = new int[executionTimes.length];
        for (int i = 0; i < executionTimes.length; i++){
            executionTimecopy[i] = (int) executionTimes[i];
```

```
148             }
149             XYSeries series = chart.addSeries("Execution Time", vertexCounts, executionTimecopy);
150             series.setMarker(SeriesMarkers.CIRCLE);
151             new SwingWrapper<>(chart).displayChart();
152         }
153 
154 
155         public static int[] countConflicts(ArrayList<ArrayList<Integer>> am){
156             int[] conflicts = new int[am.size()];
157             for (int i = 0; i < am.size(); i++){
158                 conflicts[i] = am.get(i).size();
159             }
160             return conflicts;
161         }
162 
163             public static ArrayList<ArrayList<Integer>> initGraph(int v) {
164             ArrayList<ArrayList<Integer>> am = new ArrayList<>(v);
165             for (int i = 0; i < v; i++) {
166                 am.add(new ArrayList<>());
167             }
168             return am;
169         }
170 
171         public static int[] getDegrees(ArrayList<ArrayList<Integer>> am) {
172             int[] degrees = new int[am.size()];
173             for (int i = 0; i < am.size(); i++) {
174                 degrees[i] = am.get(i).size();
175             }
176             return degrees;
177         }
178 
179         public static void plotHistogram(String title, int[] degrees) {
180             List<Integer> data = new ArrayList<Integer>();
181             for (int i = 0; i < degrees.length; i++){
182                 data.add(degrees[i]);
183             }
184 
185             Histogram histogram = new Histogram(data, 10);
186             CategoryChart chart = new CategoryChartBuilder()
187                     .width(800)
188                     .height(600)
189                     .title(title)
190                     .xAxisTitle("Degree")
191                     .yAxisTitle("Frequency")
192                     .build();
193 
194             chart.addSeries("Degree Distribution", histogram.getxAxisData(), histogram.getyAxisData());
195             new SwingWrapper<>(chart).displayChart();
196         }
197 
198 
199 
200         public static void main(String[] args) {
201             // -------------------Part one different generate graph running time-------------------------------
202 //            int[] vertexCounts = new int[1000];
203 //            vertexCounts[0] = 100;
204 //            for (int i = 1; i < vertexCounts.length; i++){
```

```
205 //             vertexCounts[i] = vertexCounts[i-1]+vertexCounts[0];
206 //         }
207         int[] vertexCounts = {10, 50, 100, 200, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000};
208 
209         long[] executionTimes = new long[vertexCounts.length];
210 
211         for (int i = 0; i < vertexCounts.length; i++) {
212             int V = vertexCounts[i];
213             ArrayList<ArrayList<Integer>> am = new ArrayList<>(V);
214             for (int j = 0; j < V; j++) {
215                 am.add(new ArrayList<>());
216             }
217 
218             long startTime = System.nanoTime();
219             Complete.allEdge(am, V);
220             //Cycle.allEdgeCycle(am,V);
221             // Uniform.uniformRandom(am, V, V/2);
222             // Uniform.uniformRandom(am, V, V/4);
223             //Uniform.uniformRandom(am, V, V/8);
224             // Uniform.uniformRandom(am, V, 2*V);
225             //Uniform.uniformRandom(am, V, 4*V);
226             //Uniform.uniformRandom(am, V, 8*V);
227 
228 //             Skewed.skewedRandom(am, V, V/8);
229 //             Skewed.skewedRandom(am,V,V/4);
230 //             Skewed.skewedRandom(am, V, V/2);
231 //             Skewed.skewedRandom(am, V, V);
232 //             Skewed.skewedRandom(am, V, 2*V);
233 //             Skewed.skewedRandom(am, V, 4* V);
234 //             Skewed.skewedRandom(am, V, 8* V);
235 //             Gauss.gaussRandom(am,V, V/8);
236 //             Gauss.gaussRandom(am,V, V/4);
237 //             Gauss.gaussRandom(am, V, V/2);
238 //             Gauss.gaussRandom(am, V, V);
239 //             Gauss.gaussRandom(am,V, 2*V);
240 //             Gauss.gaussRandom(am,V, 4*V);
241 //             Gauss.gaussRandom(am, V, 8*V);
242 //
243 //
244             long endTime = System.nanoTime();
245             executionTimes[i] = endTime - startTime;
246         }
247         System.out.println(Arrays.toString(executionTimes));
248         plotResults(vertexCounts, executionTimes);
249 
250         // ---------------------- conflict -------------------------------
251             int v = 100;
252             int Edges = 200;
253 
254             ArrayList<ArrayList<Integer>> amCycle = initGraph(v);
255             ArrayList<ArrayList<Integer>> amComplete = initGraph(v);
256             ArrayList<ArrayList<Integer>> amUniform = initGraph(v);
257             ArrayList<ArrayList<Integer>> amSkewed = initGraph(v);
258             ArrayList<ArrayList<Integer>> amGuass = initGraph(v);
259 
260             Cycle.allEdgeCycle(amCycle, v);
261             Complete.allEdge(amComplete, v);
```

```
            Uniform.uniformRandom(amUniform, v,  Edges);
            Skewed.skewedRandom(amSkewed,v, Edges);
            Gauss.gaussRandom(amGuass,v, Edges);

            int[] cycleDegrees = getDegrees(amCycle);
            int[] completeDegrees = getDegrees(amComplete);
            int[] uniformDegrees = getDegrees(amUniform);
            int[] skewedDegrees = getDegrees(amSkewed);
            int[] guassDegrees = getDegrees(amGuass);

            plotHistogram("Cycle Graph", cycleDegrees);
            plotHistogram("Complete Graph", completeDegrees);
            plotHistogram("Uniform Random Graph", uniformDegrees);
            plotHistogram("Skewed Random Graph", skewedDegrees);
            plotHistogram("Guass Random Graph", guassDegrees);
        //
    }
}

```

PartTwo.java: This file include the different orders and some graphs.

```
package PartTwo;

import java.util.*;

public class PartTwo {
    public static class Graph{
        static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d){
            am.get(s).add(d);
            am.get(d).add(s);
        }
    }

    public static void printAdjacencyList(ArrayList<ArrayList<Integer>> am){
        for (int i = 0; i < am.size(); i++){
            System.out.print("Vertex " + i + " is connected to: ");
            for (int j = 0; j < am.get(i).size(); j++){
                System.out.print(am.get(i).get(j) + " ");
            }
            System.out.println();
        }
    }

    public static int[] color(ArrayList<ArrayList<Integer>> am, ArrayList<Integer> Order){
        int[] colorassign = new int[am.size()];
        for(int i = 0; i < am.size(); i++){
            colorassign[i] = -1;
        }

        // The first
        colorassign[0] = 0;

        // A temporary array to store the available colors. False
        // value of available[cr] would mean that the color cr is
        // assigned to one of its adjacent vertices
```

```
35          boolean available[] = new boolean[am.size()];
36
37          // Initially, all colors are available
38          for (int i = 0; i < am.size(); i++){
39              available[i] = true;
40          }
41
42          // Assign colors to remaining V-1 vertices
43          for (int i = 1; i < am.size(); i++){
44              // Process all adjacent vertices and flag their colors
45              // as unavailable
46              Iterator<Integer> it = am.get(Order.get(i)).iterator();
47              while (it.hasNext()){
48                  int k = it.next();
49                  if (colorassign[Order.indexOf(k)] != -1){
50                      available[colorassign[Order.indexOf(k)]] = false;
51
52                  }
53              }
54
55              // Find the first available color
56              int cr;
57              for (cr = 0; cr < am.size(); cr++) {
58                  if (available[cr]) {
59                      break;
60                  }
61              }
62
63              // Assign the found color;
64              colorassign[i] = cr;
65
66              // Reset the values back to true for the next iteration
67              for (int j = 0; j < am.size(); j++) {
68                  available[j] = true;
69              }
70
71          }
72
73          // print the result
74          System.out.println("The color result:");
75          for (int i = 0; i < am.size(); i++)
76              System.out.println("Vertex " + Order.get(i) + " --->  Color "
77                      + colorassign[i]);
78
79
80          return colorassign;
81      }
82      // -------------------------------------------------------------------------------------------
83      // Method 1: Smallest Last Vertex (SLV) Ordering:
84      public static int terminalclique;
85      public static int maxdegree;
86      public static void removeVertex(int vertex, ArrayList<ArrayList<Integer>> am) {
87          for (int i = 0; i < am.size(); i++) {
88              am.get(i).remove((Integer) vertex);
89          }
90          am.set(vertex, new ArrayList<>());
91      }
```

```
92
93      public static int findSmallestDegreeVertex(ArrayList<ArrayList<Integer>> am) {
94          int minDegree = Integer.MAX_VALUE;
95          int minDegreeVertex = -1;
96
97          for (int i = 0; i < am.size(); i++) {
98              int degree = am.get(i).size();
99
100             System.out.println("The degree of Vertex " + i + ": " + degree);
101             if (degree < minDegree && degree > 0) {
102                 minDegree = degree;
103                 minDegreeVertex = i;
104
105             }
106         }
107
108         return minDegreeVertex;
109     }
110
111     public static ArrayList<ArrayList<Integer>> copy(ArrayList<ArrayList<Integer>> am){
112         ArrayList<ArrayList<Integer>> am2 = new ArrayList<ArrayList<Integer>>();
113         for (int i = 0; i < am.size(); i++){
114             am2.add(am.get(i));
115         }
116
117         return am2;
118     }
119
120     public static ArrayList<Integer> smallestLastVertexOrdering(ArrayList<ArrayList<Integer>> originalam) {
121         ArrayList<ArrayList<Integer>> am = copy(originalam);
122         ArrayList<Integer> slvOrder = new ArrayList<>();
123         int res = -1;
124         maxdegree = -1;
125         terminalclique = -1;
126
127         while (slvOrder.size() < am.size()) {
128             printAdjacencyList(am);
129             int minDegreeVertex = findSmallestDegreeVertex(am);
130
131             if (minDegreeVertex == -1) {
132                 slvOrder.add(0, res);
133                 break;
134             }
135
136             System.out.println("The min degree Vertex is (will be deleted): " + minDegreeVertex);
137             System.out.println();
138
139             if (am.get(minDegreeVertex).size() > maxdegree){
140                 maxdegree = am.get(minDegreeVertex).size();
141             }
142
143             if (terminalclique < am.get(minDegreeVertex).size() &&
144                     ((am.size()-slvOrder.size()-1) == am.get(minDegreeVertex).size())){
145                 terminalclique = am.get(minDegreeVertex).size();
146             }
147
148
```

```
            slvOrder.add(0, minDegreeVertex);
            System.out.println(slvOrder);
            if (am.get(minDegreeVertex).size() == 1){
                res = am.get(minDegreeVertex).get(0);
            }
            removeVertex(minDegreeVertex,am);

        }
        System.out.println("The order to delete (from right to left): " + slvOrder);
        return slvOrder;
    }

    // ------------------------------------------------------------------------------------------
    // Method 2: Smallest Original Degree Last (SODL) Ordering:
    public static int findMaxDegreeVertex(ArrayList<ArrayList<Integer>> am, ArrayList<Integer> exit){
        int maxDegree = Integer.MIN_VALUE;
        int maxDegreeVertex = -1;

        for (int i = 0; i < am.size(); i++){
            int degree = am.get(i).size();

            System.out.println("The degree of Vertex " + i + ": " + degree);
            if (degree > maxDegree && degree > 0 && !exit.contains(i)){
                maxDegree = degree;
                maxDegreeVertex = i;
            }
        }

        return maxDegreeVertex;
    }


    private static ArrayList<Integer> smallestOriginalDegreeLast(ArrayList<ArrayList<Integer>> am){
        ArrayList<Integer> solOrder = new ArrayList<>();
        int V = am.size();
        //System.out.println(V);


        while(solOrder.size() < am.size()){
            int minDegreeVertex = findMaxDegreeVertex(am, solOrder);

            System.out.println("The vertex add to solOrder "+ minDegreeVertex + "\n");

            solOrder.add(0, minDegreeVertex);

        }

        System.out.println("Smallest orginal degree Last ORder (from right to left): " + "\n" + solOrder);
        return solOrder;
    }

    // ------------------------------------------------------------------------------------------
    // Method 3: Uniform Random Ordering:
    public static ArrayList<Integer> uniformrandom(ArrayList<ArrayList<Integer>> am){
        ArrayList<Integer> unOrder = new ArrayList<Integer>();
        for (int i = 0; i < am.size(); i++){
            unOrder.add(i);
```

```
206             }
207
208             Collections.shuffle(unOrder,new Random());
209             // the list to be shuffled
210             // the source of randomness to use to shuffle the list.
211
212
213             System.out.println("Uniform random Order is : " + unOrder +"\n");
214
215
216             return unOrder;
217         }
218
219         // ------------------------------------------------------------------------------------------------
220         // Method 4: Breadth-First Search (BFS) Ordering:
221         public static ArrayList<Integer> BFSOrder(ArrayList<ArrayList<Integer>> am, int startNode){
222             // Create a queue for bfs;
223             ArrayList<Integer> bfsOrder = new ArrayList<>();
224
225             // Make all the vertices as not visited(By default set as false)
226             boolean visited[] = new boolean[am.size()];
227             Queue<Integer> queue = new LinkedList<>();
228
229             // Mark the current node as visited and enqueue it
230             // start from the first one.
231             visited[startNode] = true;
232             queue.offer(startNode);
233
234             while(!queue.isEmpty()){
235                 int node = queue.poll();
236                 bfsOrder.add(node);
237                 //System.out.print(node + " ");
238
239                 for (int neighbor : am.get(node)){
240                     if (!visited[neighbor]){
241                         visited[neighbor] = true;
242                         queue.offer(neighbor);
243                     }
244                 }
245
246             }
247             System.out.println("BFSOrder is : " + bfsOrder +"\n");
248
249             return bfsOrder;
250         }
251
252         // ------------------------------------------------------------------------------------------------
253         // Method 5: Depth-First Search (DFS) Ordering:
254         public static ArrayList<Integer> DFSOrder(ArrayList<ArrayList<Integer>> am, int startNode){
255             ArrayList<Integer> dfsOrder = new ArrayList<>();
256             boolean[] visited = new boolean[am.size()];
257
258             dfsRecursive(am, startNode, visited, dfsOrder);
259             System.out.println("DFSOrder is : " + dfsOrder +"\n");
260             return dfsOrder;
261         }
262
```

```java
    private static void dfsRecursive(ArrayList<ArrayList<Integer>> am, int startNode,
                                     boolean[] visited, ArrayList<Integer> dfsOrder){
        visited[startNode] = true;
        dfsOrder.add(startNode);

        for (int neighbor : am.get(startNode)){
            if(!visited[neighbor]){
                dfsRecursive(am,neighbor,visited,dfsOrder);
            }
        }
    }

    // ------------------------------------------------------------------------------------------
    // Method 6: Lexicographic Breadth-First Search (LexBFS) Ordering
    public static ArrayList<Integer> lexBFS(ArrayList<ArrayList<Integer>> am, int startnode){
        int n = am.size();
        ArrayList<Integer> lexOrder = new ArrayList<>();


        // Create a mapping from node index to its neighbor set
        Map<Integer, Set<Integer>> nodeToNeighbors = new HashMap<>();
        for (int i = 0; i < n; i++){
            Set<Integer> neighbors = new HashSet<>();
            for (int neighbor:am.get(i)){
                neighbors.add(neighbor);
            }
            nodeToNeighbors.put(i, neighbors);
        }

        // Perform lexicographic BFS starting from node 0
        Queue<Integer> bfsQueue = new LinkedList<>();
        Set<Integer> visited = new HashSet<>();
        bfsQueue.add(startnode);
        visited.add(startnode);

        while(!bfsQueue.isEmpty()){
            int node = bfsQueue.poll();
            lexOrder.add(node);

            ArrayList<Integer> unvisitedNeighbors = new ArrayList<>();
            for (int neihbor: nodeToNeighbors.get(node)){
                if (!visited.contains(neihbor)){
                    unvisitedNeighbors.add(neihbor);
                }
            }
            Collections.sort(unvisitedNeighbors);

            for (int neighbor: unvisitedNeighbors){
                bfsQueue.add(neighbor);
                visited.add(neighbor);
            }
        }

        System.out.println("LexOrder is : " + lexOrder +"\n");


        return lexOrder;
```

```
320         }
321
322         public static long time_calculate(){
323             long startTime = System.nanoTime();
324
325
326             long endTime = System.nanoTime();
327             long Time = endTime - startTime;
328             return Time;
329         }
330
331
332
333         public static void main(String[] args){
334             //---------------------- Example 1 init-----------------------------------
335             int v = 5;
336             ArrayList<ArrayList<Integer>> adjacencyList1 = new ArrayList<ArrayList<Integer>>(v);
337             for (int i = 0; i < v; i++) {
338                 adjacencyList1.add((new ArrayList<Integer>()));
339             }
340
341             // Complete.allEdge(adjacencyList, v);
342             PartTwo.Graph.addEdge(adjacencyList1,0,1);
343             PartTwo.Graph.addEdge(adjacencyList1, 0,2);
344             PartTwo.Graph.addEdge(adjacencyList1,0,3);
345             PartTwo.Graph.addEdge(adjacencyList1,1,4);
346             PartTwo.Graph.addEdge(adjacencyList1,1,3);
347             PartTwo.Graph.addEdge(adjacencyList1,1,2);
348             PartTwo.Graph.addEdge(adjacencyList1,2,3);
349             PartTwo.Graph.addEdge(adjacencyList1,2,4);
350
351             //---------------------- Example 2 init -----------------------------------
352             int v2 = 5;
353             ArrayList<ArrayList<Integer>> adjacencyList2 = new ArrayList<ArrayList<Integer>>(v2);
354             for (int i = 0; i < v2; i++) {
355                 adjacencyList2.add((new ArrayList<Integer>()));
356             }
357
358             PartTwo.Graph.addEdge(adjacencyList2,0,1);
359             PartTwo.Graph.addEdge(adjacencyList2, 0,2);
360             PartTwo.Graph.addEdge(adjacencyList2,1,4);
361             PartTwo.Graph.addEdge(adjacencyList2,1,3);
362             PartTwo.Graph.addEdge(adjacencyList2,2,3);
363             PartTwo.Graph.addEdge(adjacencyList2,3,4);
364
365
366             //------------------- Example 3 init -----------------------------------
367             int v3 = 10;
368             ArrayList<ArrayList<Integer>> adjacencyList3 = new ArrayList<ArrayList<Integer>>(v3);
369             for (int i = 0; i < v3; i++) {
370                 adjacencyList3.add((new ArrayList<Integer>()));
371             }
372
373             PartTwo.Graph.addEdge(adjacencyList3,0,1);
374             PartTwo.Graph.addEdge(adjacencyList3, 0,6);
375             PartTwo.Graph.addEdge(adjacencyList3,0,7);
376             PartTwo.Graph.addEdge(adjacencyList3, 0, 8);
```

```
377         PartTwo.Graph.addEdge(adjacencyList3,1,6);
378         PartTwo.Graph.addEdge(adjacencyList3,1,2);
379         PartTwo.Graph.addEdge(adjacencyList3,1,9);
380         PartTwo.Graph.addEdge(adjacencyList3,1,5);
381         PartTwo.Graph.addEdge(adjacencyList3,2,9);
382         PartTwo.Graph.addEdge(adjacencyList3,2,5);
383         PartTwo.Graph.addEdge(adjacencyList3,2,3);
384         PartTwo.Graph.addEdge(adjacencyList3, 2, 4);
385         PartTwo.Graph.addEdge(adjacencyList3,2,6);
386         PartTwo.Graph.addEdge(adjacencyList3, 3,4);
387         PartTwo.Graph.addEdge(adjacencyList3,3,9);
388         PartTwo.Graph.addEdge(adjacencyList3,4,5);
389         PartTwo.Graph.addEdge(adjacencyList3,5,6);
390         PartTwo.Graph.addEdge(adjacencyList3, 6, 7);
391
392         long startTime, endTime, Time;
393
394         // Method 1: (SLV)---------------------- test -------------------
395         System.out.println("---------------------Example 1---------------------------------");
396
397         ArrayList<ArrayList<Integer>> adjacencyList1copy = copy(adjacencyList1);
398
399         System.out.println("The original adjacencyList:");
400         printAdjacencyList(adjacencyList1copy);
401         System.out.println();
402         color(adjacencyList1,smallestLastVertexOrdering(adjacencyList1copy));
403         System.out.println("The size of terminal clique degree: " + terminalclique);
404         System.out.println("The max degree during removing order: " + maxdegree);
405         System.out.println();
406
407
408         System.out.println("---------------------Example 2---------------------------------");
409         ArrayList<ArrayList<Integer>> adjacencyList2copy = copy(adjacencyList2);
410
411         System.out.println("The original adjacencyList:");
412         printAdjacencyList(adjacencyList2copy);
413         System.out.println();
414         color(adjacencyList2,smallestLastVertexOrdering(adjacencyList2copy));
415         System.out.println("The size of terminal clique degree: " + terminalclique);
416         System.out.println("The max degree during removing order: " + maxdegree);
417         System.out.println();
418
419         System.out.println("---------------------Example 3---------------------------------");
420
421         ArrayList<ArrayList<Integer>> adjacencyList3copy = copy(adjacencyList3);
422
423         System.out.println("The original adjacencyList:");
424         printAdjacencyList(adjacencyList3copy);
425         System.out.println();
426         color(adjacencyList3,smallestLastVertexOrdering(adjacencyList3copy));
427         System.out.println("The size of terminal clique degree: " + terminalclique);
428         System.out.println("The max degree during removing order: " + maxdegree);
429
430
431 //        // Method 2: (SODL) ------------------ test -------------------
432 //        System.out.println("---------------------Example 1---------------------------------");
433 //        System.out.println("The original adjacencyList:");
```

```
434 //         printAdjacencyList(adjacencyList1);
435 //         System.out.println();
436 //
437 //         System.out.println("The Method for graph order result(from right to left):");
438 //         startTime = System.nanoTime();
439 //         color(adjacencyList1,smallestOriginalDegreeLast(adjacencyList1));
440 //         endTime = System.nanoTime();
441 //         System.out.println();
442 //
443 //         Time = endTime -startTime;
444 //         System.out.println("The running time: " + Time);
445 //         System.out.println();
446 //
447 //         System.out.println("----------------------Example 2--------------------------------");
448 //         System.out.println("The orignial graph:");
449 //
450 //         printAdjacencyList(adjacencyList2);
451 //         System.out.println();
452 //         System.out.println("The Method for graph order result(from right to left):");
453 //         startTime = System.nanoTime();
454 //         color(adjacencyList2,smallestOriginalDegreeLast(adjacencyList2));
455 //         endTime = System.nanoTime();
456 //         System.out.println();
457 //
458 //         Time = endTime -startTime;
459 //         System.out.println("The running time: " + Time);
460 //         System.out.println();
461 //
462 //
463 //         System.out.println("----------------------Example 3--------------------------------");
464 //         System.out.println("The original adjacencyList:");
465 //         printAdjacencyList(adjacencyList3);
466 //         System.out.println();
467 //         System.out.println("The Method for graph order result(from right to left):");
468 //         startTime = System.nanoTime();
469 //         color(adjacencyList3,smallestOriginalDegreeLast(adjacencyList3));
470 //         endTime = System.nanoTime();
471 //         Time = endTime -startTime;
472 //         System.out.println("The running time: " + Time);
473 //         System.out.println();
474
475
476         // Method 3: (URO) ------------------ test -------------------
477 //         System.out.println("----------------------Example 1--------------------------------");
478 //         System.out.println("The original adjacencyList:");
479 //         printAdjacencyList(adjacencyList1);
480 //         System.out.println();
481 //
482 //         System.out.println("The Method for graph order result(from right to left):");
483 //         long startTime1 = System.nanoTime();
484 //         color(adjacencyList1,uniformrandom(adjacencyList1));
485 //         long endTime1 = System.nanoTime();
486 //         System.out.println();
487 //
488 //         long Time1 = endTime1 -startTime1;
489 //         System.out.println("The running time: " + Time1);
490 //         System.out.println();
```

```
491 //
492 //        System.out.println("----------------------Example 2--------------------------------");
493 //        System.out.println("The orignial graph:");
494 //
495 //        printAdjacencyList(adjacencyList2);
496 //        System.out.println();
497 //        System.out.println("The Method for graph order result(from right to left):");
498 //        long startTime2 = System.nanoTime();
499 //        color(adjacencyList2,uniformrandom(adjacencyList2));
500 //        long endTime2 = System.nanoTime();
501 //        System.out.println();
502 //
503 //        long Time2 = endTime2 -startTime2;
504 //        System.out.println("The running time: " + Time2);
505 //        System.out.println();
506 //
507 //
508 //        System.out.println("----------------------Example 3--------------------------------");
509 //        System.out.println("The original adjacencyList:");
510 //        printAdjacencyList(adjacencyList3);
511 //        System.out.println();
512 //        System.out.println("The Method for graph order result(from right to left):");
513 //        long startTime3 = System.nanoTime();
514 //        color(adjacencyList3,uniformrandom(adjacencyList3));
515 //        long endTime3 = System.nanoTime();
516 //        long Time3 = endTime3 -startTime3;
517 //        System.out.println("The running time: " + Time3);
518 //        System.out.println();
519
520         // Method 4: (BFS) ------------------ test --------------------
521 //        System.out.println("----------------------Example 1--------------------------------");
522 //        System.out.println("The original adjacencyList:");
523 //        printAdjacencyList(adjacencyList1);
524 //        System.out.println();
525 //
526 //        System.out.println("The Method for graph order result(from right to left):");
527 //        long startTime1 = System.nanoTime();
528 //        color(adjacencyList1,BFSOrder(adjacencyList1,2));
529 //        long endTime1 = System.nanoTime();
530 //        System.out.println();
531 //
532 //        long Time1 = endTime1 -startTime1;
533 //        System.out.println("The running time: " + Time1);
534 //        System.out.println();
535 //
536 //        System.out.println("----------------------Example 2--------------------------------");
537 //        System.out.println("The orignial graph:");
538 //
539 //        printAdjacencyList(adjacencyList2);
540 //        System.out.println();
541 //        System.out.println("The Method for graph order result(from right to left):");
542 //        long startTime2 = System.nanoTime();
543 //        color(adjacencyList2,BFSOrder(adjacencyList2,2));
544 //        long endTime2 = System.nanoTime();
545 //        System.out.println();
546 //
547 //        long Time2 = endTime2 -startTime2;
```

```
548 //         System.out.println("The running time: " + Time2);
549 //         System.out.println();
550 //
551 //
552 //         System.out.println("----------------------Example 3---------------------------------");
553 //         System.out.println("The original adjacencyList:");
554 //         printAdjacencyList(adjacencyList3);
555 //         System.out.println();
556 //         System.out.println("The Method for graph order result(from right to left):");
557 //         long startTime3 = System.nanoTime();
558 //         color(adjacencyList3,BFSOrder(adjacencyList3,2));
559 //         long endTime3 = System.nanoTime();
560 //         long Time3 = endTime3 -startTime3;
561 //         System.out.println("The running time: " + Time3);
562 //         System.out.println();
563
564
565         // Method 5: (DFS) ------------------ test --------------------
566 //         System.out.println("----------------------Example 1---------------------------------");
567 //         System.out.println("The original adjacencyList:");
568 //         printAdjacencyList(adjacencyList1);
569 //         System.out.println();
570 //
571 //         System.out.println("The Method for graph order result(from right to left):");
572 //         long startTime1 = System.nanoTime();
573 //         color(adjacencyList1,DFSOrder(adjacencyList1,2));
574 //         long endTime1 = System.nanoTime();
575 //         System.out.println();
576 //
577 //         long Time1 = endTime1 -startTime1;
578 //         System.out.println("The running time: " + Time1);
579 //         System.out.println();
580 //
581 //         System.out.println("----------------------Example 2---------------------------------");
582 //         System.out.println("The orignial graph:");
583 //
584 //         printAdjacencyList(adjacencyList2);
585 //         System.out.println();
586 //         System.out.println("The Method for graph order result(from right to left):");
587 //         long startTime2 = System.nanoTime();
588 //         color(adjacencyList2,DFSOrder(adjacencyList2,2));
589 //         long endTime2 = System.nanoTime();
590 //         System.out.println();
591 //
592 //         long Time2 = endTime2 -startTime2;
593 //         System.out.println("The running time: " + Time2);
594 //         System.out.println();
595 //
596 //
597 //         System.out.println("----------------------Example 3---------------------------------");
598 //         System.out.println("The original adjacencyList:");
599 //         printAdjacencyList(adjacencyList3);
600 //         System.out.println();
601 //         System.out.println("The Method for graph order result(from right to left):");
602 //         long startTime3 = System.nanoTime();
603 //         color(adjacencyList3,DFSOrder(adjacencyList3,2));
604 //         long endTime3 = System.nanoTime();
```

```
605 //         long Time3 = endTime3 -startTime3;
606 //         System.out.println("The running time: " + Time3);
607 //         System.out.println();
608
609
610         // Method 6: lexBFS ----------------- test -------------------
611 //         System.out.println("---------------------Example 1---------------------------------");
612 //         System.out.println("The original adjacencyList:");
613 //         printAdjacencyList(adjacencyList1);
614 //         System.out.println();
615 //
616 //         System.out.println("The Method for graph order result(from right to left):");
617 //         long startTime1 = System.nanoTime();
618 //         color(adjacencyList1,lexBFS(adjacencyList1,2));
619 //         long endTime1 = System.nanoTime();
620 //         System.out.println();
621 //
622 //         long Time1 = endTime1 -startTime1;
623 //         System.out.println("The running time: " + Time1);
624 //
625 //         System.out.println("----------------------Example 2---------------------------------");
626 //         System.out.println("The orignial graph:");
627 //
628 //         printAdjacencyList(adjacencyList2);
629 //         System.out.println();
630 //         System.out.println("The Method for graph order result(from right to left):");
631 //         long startTime2 = System.nanoTime();
632 //         color(adjacencyList2,lexBFS(adjacencyList2,2));
633 //         long endTime2 = System.nanoTime();
634 //         System.out.println();
635 //
636 //         long Time2 = endTime2 -startTime2;
637 //         System.out.println("The running time: " + Time2);
638 //
639 //
640 //         System.out.println("----------------------Example 3---------------------------------");
641 //         System.out.println("The original adjacencyList:");
642 //         printAdjacencyList(adjacencyList3);
643 //         System.out.println();
644 //         System.out.println("The Method for graph order result(from right to left):");
645 //         long startTime3 = System.nanoTime();
646 //         color(adjacencyList3,lexBFS(adjacencyList3,2));
647 //         long endTime3 = System.nanoTime();
648 //         long Time3 = endTime3 -startTime3;
649 //         System.out.println("The running time: " + Time3);
650
651     }
652 }
653
654
655
```

Time2.java: This file include the different orders and some graphs.

```
1 import org.knowm.xchart.*;
2 import org.knowm.xchart.style.markers.SeriesMarkers;
```

```
3
4   import java.util.*;
5
6   public class Time2 {
7       // Part One
8
9       public static class Complete {
10          public static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
11              am.get(s).add(d);
12              am.get(d).add(s);
13          }
14
15          public static void allEdge(ArrayList<ArrayList<Integer>> am, int V) {
16              for (int i = 0; i < V; i++) {
17                  for (int j = i + 1; j < V; j++) {
18                      addEdge(am, i, j);
19                  }
20              }
21          }
22      }
23
24      static class Cycle {
25          //private static long time;
26          static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
27              am.get(s).add(d);
28              am.get(d).add(s);
29          }
30          static void allEdgeCycle(ArrayList<ArrayList<Integer>> am, int V) {
31
32              for (int i = 0; i < V; i++) {
33                  if (i == V - 1) {
34                      addEdge(am, i, 0);    // head -- tail
35                  } else {
36                      addEdge(am, i, i + 1);
37                  }
38              }
39
40          }
41      }
42
43      static class Skewed {
44          static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
45              am.get(s).add(d);
46              am.get(d).add(s);
47          }
48
49          static void skewedRandom(ArrayList<ArrayList<Integer>> am, int v, int e) {
50  //              Random rand = new Random();
51              int source = -1;
52              int dest = -1;
53              int a = 0;
54              int b = v;
55              int c = 0;
56              double F = (c - a) / (b - a);//
57              while (e > 0) {
58                  double rand = Math.random();
59                  if (rand < F) {
```

```
60                      source = (int) (a + Math.sqrt(rand * (b - a) * (c - a)));
61                  } else {
62                      source = (int) (b - Math.sqrt((1 - rand) * (b - a) * (b - c)));
63                  }
64
65                  double rand2 = Math.random();
66                  if (rand2 < F) {
67                      dest = (int) (a + Math.sqrt(rand2 * (b - a) * (c - a)));
68                  } else {
69                      dest = (int) (b - Math.sqrt((1 - rand2) * (b - a) * (b - c)));
70                  }
71
72                  if (source == dest || am.get(source).contains(dest)) {
73                      continue;
74                  } else {
75                      addEdge(am, source, dest);
76                      e--;
77                  }
78              }
79
80          }
81      }
82
83      static class Gauss {
84          static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
85              am.get(s).add(d);
86              am.get(d).add(s);
87          }
88
89          //Gauss
90          static void gaussRandom(ArrayList<ArrayList<Integer>> am, int v, int e) {
91              Random rand = new Random();
92              while (e > 0) {
93
94                  int source = (int) (v / 10 * rand.nextGaussian() + v / 2);
95                  int dest = (int) (v / 10 * rand.nextGaussian() + v / 2);
96  //                source = Math.abs(source);
97  //                dest = Math.abs(dest);
98                  try{
99                      if (source == dest || am.get(source).contains(dest)) {
100                         continue;
101                     } else {
102                         addEdge(am, source, dest);
103                         e--;
104                     }
105                 }catch (Exception ignored){}
106
107             }
108         }
109     }
110     static class Uniform {
111         static void addEdge(ArrayList<ArrayList<Integer>> am, int s, int d) {
112             am.get(s).add(d);
113             am.get(d).add(s);
114         }
115
116         // If you provide an integer parameter to "nextInt",
```

```
117         // it will return an integer from a uniform distribution between 0 and one less than the parameter.
118         static void uniformRandom(ArrayList<ArrayList<Integer>> am, int v, int e) {
119             //Random rand = new Random();
120             while (e > 0) {
121                 int source = (int) (Math.random() * v); // Printing the random number between [0,v-1]
122                 int dest = (int) (Math.random() * v);
123                 // edge exit?
124                 if (source == dest || am.get(source).contains(dest)) {
125                     continue;
126                 } else {
127                     addEdge(am, source, dest);
128                     e--;
129                 }
130             }
131         }
132     }
133 
134     public static int color(ArrayList<ArrayList<Integer>> am, ArrayList<Integer> Order){
135         int[] colorassign = new int[am.size()];
136         for(int i = 0; i < am.size(); i++){
137             colorassign[i] = -1;
138         }
139 
140         int max = 0;
141         int count = 0;
142 
143         // The first
144         colorassign[0] = 0;
145 
146         // A temporary array to store the available colors. False
147         // value of available[cr] would mean that the color cr is
148         // assigned to one of its adjacent vertices
149         boolean available[] = new boolean[am.size()];
150 
151         // Initially, all colors are available
152         for (int i = 0; i < am.size(); i++){
153             available[i] = true;
154         }
155 
156         // Assign colors to remaining V-1 vertices
157         for (int i = 1; i < am.size(); i++){
158             // Process all adjacent vertices and flag their colors
159             // as unavailable
160             //System.out.println("Accessing Order.get(" + i + ")");
161 //            System.out.println("Size of am: " + am.size());
162 //            System.out.println("Size of Order: " + Order.size());
163             Iterator<Integer> it = am.get(Order.get(i)).iterator();
164             while (it.hasNext()){
165                 int k = it.next();
166                 if (colorassign[Order.indexOf(k)] != -1){
167                     available[colorassign[Order.indexOf(k)]] = false;
168 
169                 }
170             }
171 
172             // Find the first available color
173             int cr;
```

```
174                 for (cr = 0; cr < am.size(); cr++) {
175                     if (available[cr]) {
176                         break;
177                     }
178                 }
179 
180                 // Assign the found color;
181                 colorassign[i] = cr;
182 
183                 if (cr > max){
184                     max = cr;
185                 }
186 
187                 // Reset the values back to true for the next iteration
188                 for (int j = 0; j < am.size(); j++) {
189                     available[j] = true;
190                 }
191 
192             }
193             max = max+1;
194 
195 //          // print the result
196 //          System.out.println("The color result:");
197 //          for (int i = 0; i < am.size(); i++)
198 //              System.out.println("Vertex " + Order.get(i) + " --->  Color "
199 //                      + colorassign[i]);
200 //
201 //
202 //          //return colorassign;
203 //          System.out.println("Need " + max + " colors.");
204             return max;
205         }
206 
207         private static void plotResults(String title, int[] vertexCounts, long[] executionTimes) {
208             XYChart chart = new XYChartBuilder()
209                     .width(800)
210                     .height(600)
211                     .title(title)
212                     .xAxisTitle("Number of Vertices (V)")
213                     .yAxisTitle("Execution Time (nanoseconds)")
214                     .build();
215 
216             int[] executionTimecopy = new int[executionTimes.length];
217             for (int i = 0; i < executionTimes.length; i++){
218                 executionTimecopy[i] = (int) executionTimes[i];
219             }
220             XYSeries series = chart.addSeries("Execution Time", vertexCounts, executionTimecopy);
221             series.setMarker(SeriesMarkers.CIRCLE);
222             new SwingWrapper<>(chart).displayChart();
223         }
224 
225         // Color
226         public static void plotResults2(String title, int[] vertexCounts, List<int[]> colorsList, List<String> labels
227             if (colorsList.size() != labels.size()) {
228                 throw new IllegalArgumentException("Colors and labels must have the same size.");
229             }
230 
```

```
231             XYChart chart = new XYChartBuilder()
232                     .width(800)
233                     .height(600)
234                     .title(title)
235                     .xAxisTitle("Number of Vertices (V)")
236                     .yAxisTitle("Color")
237                     .build();
238 
239             for (int i = 0; i < colorsList.size(); i++) {
240                 int[] colors = colorsList.get(i);
241                 XYSeries series = chart.addSeries(labels.get(i), vertexCounts, colors);
242                 series.setMarker(SeriesMarkers.CIRCLE);
243             }
244 
245             new SwingWrapper<>(chart).displayChart();
246         }
247 
248         // Time
249         public static void plotResults3(String title, int[] vertexCounts, List<long[]> executionTimesList, List<Strin
250             if (executionTimesList.size() != labels.size()) {
251                 throw new IllegalArgumentException("Execution times list and labels must have the same size.");
252             }
253 
254             XYChart chart = new XYChartBuilder()
255                     .width(800)
256                     .height(600)
257                     .title(title)
258                     .xAxisTitle("Number of Vertices (V)")
259                     .yAxisTitle("Execution Time (nanoseconds)")
260                     .build();
261 
262             for (int i = 0; i < executionTimesList.size(); i++) {
263                 long[] executionTimes = executionTimesList.get(i);
264 
265                 int[] executionTimeCopy = new int[executionTimes.length];
266                 for (int j = 0; j < executionTimes.length; j++) {
267                     executionTimeCopy[j] = (int) executionTimes[j];
268                 }
269 
270                 XYSeries series = chart.addSeries(labels.get(i), vertexCounts, executionTimeCopy);
271                 series.setMarker(SeriesMarkers.CIRCLE);
272             }
273 
274             new SwingWrapper<>(chart).displayChart();
275         }
276 
277 
278         public static int[] getDegrees(ArrayList<ArrayList<Integer>> am) {
279             int[] degrees = new int[am.size()];
280             for (int i = 0; i < am.size(); i++) {
281                 degrees[i] = am.get(i).size();
282             }
283             return degrees;
284         }
285 
286         public static void plotHistogram(String title, List<int[]> datasets, List<String> labels) {
287             if (datasets.size() != labels.size()) {
```

```
            throw new IllegalArgumentException("Datasets and labels must have the same size.");
        }

        CategoryChart chart = new CategoryChartBuilder()
                .width(800)
                .height(600)
                .title(title)
                .xAxisTitle("Degree")
                .yAxisTitle("Frequency")
                .build();

        for (int i = 0; i < datasets.size(); i++) {
            List<Integer> data = new ArrayList<>();
            for (int degree : datasets.get(i)) {
                data.add(degree);
            }
            Histogram histogram = new Histogram(data, 10);
            chart.addSeries(labels.get(i), histogram.getxAxisData(), histogram.getyAxisData());
        }

        new SwingWrapper<>(chart).displayChart();
    }




    // ------------------------------------------------------------------------------------------
    // Method 1: Smallest Last Vertex (SLV) Ordering:
    public static int terminalclique;
    public static int maxdegree;
    public static void removeVertex(int vertex, ArrayList<ArrayList<Integer>> am) {
        for (int i = 0; i < am.size(); i++) {
            am.get(i).remove((Integer) vertex);
        }
        am.set(vertex, new ArrayList<>());
    }

    public static int findSmallestDegreeVertex(ArrayList<ArrayList<Integer>> am) {
        int minDegree = Integer.MAX_VALUE;
        int minDegreeVertex = -1;

        for (int i = 0; i < am.size(); i++) {
            int degree = am.get(i).size();

            //System.out.println("The degree of Vertex " + i + ": " + degree);
            if (degree < minDegree && degree > 0) {
                minDegree = degree;
                minDegreeVertex = i;

            }
        }

        return minDegreeVertex;
    }

    public static ArrayList<ArrayList<Integer>> copy(ArrayList<ArrayList<Integer>> am){
```

```
345         ArrayList<ArrayList<Integer>> am2 = new ArrayList<ArrayList<Integer>>();
346         for (int i = 0; i < am.size(); i++){
347             am2.add(am.get(i));
348         }
349 
350         return am2;
351     }
352 
353     public static ArrayList<Integer> smallestLastVertexOrdering(ArrayList<ArrayList<Integer>> originalam) {
354         ArrayList<ArrayList<Integer>> am = copy(originalam);
355         ArrayList<Integer> slvOrder = new ArrayList<>();
356         int res = -1;
357         maxdegree = -1;
358         terminalclique = -1;
359 
360         while (slvOrder.size() < am.size()) {
361             //printAdjacencyList(am);
362             int minDegreeVertex = findSmallestDegreeVertex(am);
363 
364             if (minDegreeVertex == -1) {
365                 slvOrder.add(0, res);
366                 break;
367             }
368 
369             //System.out.println("The min degree Vertex is (will be deleted): " + minDegreeVertex);
370             //System.out.println();
371 
372             if (am.get(minDegreeVertex).size() > maxdegree){
373                 maxdegree = am.get(minDegreeVertex).size();
374             }
375 
376             if (terminalclique < am.get(minDegreeVertex).size() &&
377                     ((am.size()-slvOrder.size()-1) == am.get(minDegreeVertex).size())){
378                 terminalclique = am.get(minDegreeVertex).size();
379             }
380 
381 
382             slvOrder.add(0, minDegreeVertex);
383             //System.out.println(slvOrder);
384             if (am.get(minDegreeVertex).size() == 1){
385                 res = am.get(minDegreeVertex).get(0);
386             }
387             removeVertex(minDegreeVertex,am);
388 
389         }
390         //System.out.println("The order to delete (from right to left): " + slvOrder);
391         return slvOrder;
392     }
393 
394 
395     // ------------------------------------------------------------------------------------------------
396     // Method 2: Smallest Original Degree Last (SODL) Ordering:
397     public static int findMaxDegreeVertex(ArrayList<ArrayList<Integer>> am, ArrayList<Integer> exit){
398         int maxDegree = Integer.MIN_VALUE;
399         int maxDegreeVertex = 0;
400 
401         for (int i = 0; i < am.size(); i++){
```

```
            int degree = am.get(i).size();

            //System.out.println("The degree of Vertex " + i + ": " + degree);
            if (degree > maxDegree && degree > 0 && !exit.contains(i)){
                maxDegree = degree;
                maxDegreeVertex = i;
            }
        }

        return maxDegreeVertex;
    }


    private static ArrayList<Integer> smallestOriginalDegreeLast(ArrayList<ArrayList<Integer>> am){
        ArrayList<Integer> solOrder = new ArrayList<>();
        int V = am.size();
        //System.out.println(V);


        while(solOrder.size() < am.size()){
            int minDegreeVertex = findMaxDegreeVertex(am, solOrder);

            //System.out.println("The vertex add to solOrder "+ minDegreeVertex + "\n");

            solOrder.add(0, minDegreeVertex);

        }

        //System.out.println("Smallest orginal degree Last ORder (from right to left): " + "\n" + solOrder);
        return solOrder;
    }

    // ------------------------------------------------------------------------------------------
    // Method 3: Uniform Random Ordering:
    public static ArrayList<Integer> uniformrandom(ArrayList<ArrayList<Integer>> am){
        ArrayList<Integer> unOrder = new ArrayList<Integer>();
        for (int i = 0; i < am.size(); i++){
            unOrder.add(i);
        }

        Collections.shuffle(unOrder,new Random());
        // the list to be shuffled
        // the source of randomness to use to shuffle the list.


        //System.out.println("Uniform random Order is : " + unOrder +"\n");


        return unOrder;
    }

    // ------------------------------------------------------------------------------------------
    // Method 4: Breadth-First Search (BFS) Ordering:
    public static ArrayList<Integer> BFSOrder(ArrayList<ArrayList<Integer>> am, int startNode){
        // Create a queue for bfs;
        ArrayList<Integer> bfsOrder = new ArrayList<>();

```

```
459         // Make all the vertices as not visited(By default set as false)
460         boolean visited[] = new boolean[am.size()];
461         Queue<Integer> queue = new LinkedList<>();
462 
463         // Mark the current node as visited and enqueue it
464         // start from the first one.
465         visited[startNode] = true;
466         queue.offer(startNode);
467 
468         while(!queue.isEmpty()){
469             int node = queue.poll();
470             bfsOrder.add(node);
471             //System.out.print(node + " ");
472 
473             for (int neighbor : am.get(node)){
474                 if (!visited[neighbor]){
475                     visited[neighbor] = true;
476                     queue.offer(neighbor);
477                 }
478             }
479 
480         }
481         //System.out.println("BFSOrder is : " + bfsOrder +"\n");
482 
483         return bfsOrder;
484     }
485 
486     // ------------------------------------------------------------------------------------------------
487     // Method 5: Depth-First Search (DFS) Ordering:
488     public static ArrayList<Integer> DFSOrder(ArrayList<ArrayList<Integer>> am, int startNode){
489         ArrayList<Integer> dfsOrder = new ArrayList<>();
490         boolean[] visited = new boolean[am.size()];
491 
492         dfsRecursive(am, startNode, visited, dfsOrder);
493         //System.out.println("DFSOrder is : " + dfsOrder +"\n");
494         return dfsOrder;
495     }
496 
497     private static void dfsRecursive(ArrayList<ArrayList<Integer>> am, int startNode,
498                                      boolean[] visited, ArrayList<Integer> dfsOrder){
499         visited[startNode] = true;
500         dfsOrder.add(startNode);
501 
502         for (int neighbor : am.get(startNode)){
503             if(!visited[neighbor]){
504                 dfsRecursive(am,neighbor,visited,dfsOrder);
505             }
506         }
507     }
508 
509     // ------------------------------------------------------------------------------------------------
510     // Method 6: Lexicographic Breadth-First Search (LexBFS) Ordering
511     public static ArrayList<Integer> lexBFS(ArrayList<ArrayList<Integer>> am, int startnode){
512         int n = am.size();
513         ArrayList<Integer> lexOrder = new ArrayList<>();
514 
515         // Create a mapping from node index to its neighbor set
```

```
        Map<Integer, Set<Integer>> nodeToNeighbors = new HashMap<>();
        for (int i = 0; i < n; i++) {
            Set<Integer> neighbors = new HashSet<>();
            for (int neighbor : am.get(i)) {
                neighbors.add(neighbor);
            }
            nodeToNeighbors.put(i, neighbors);
        }

        // Perform lexicographic BFS starting from the startnode
        Queue<Integer> bfsQueue = new LinkedList<>();
        Set<Integer> visited = new HashSet<>();

        bfsQueue.add(startnode);
        visited.add(startnode);

        while (!bfsQueue.isEmpty()) {
            int currentNode = bfsQueue.poll();
            lexOrder.add(currentNode);

            // Get unvisited neighbors sorted lexicographically
            ArrayList<Integer> unvisitedNeighbors = new ArrayList<>();
            for (int neighbor : nodeToNeighbors.get(currentNode)) {
                if (!visited.contains(neighbor)) {
                    unvisitedNeighbors.add(neighbor);
                }
            }
            Collections.sort(unvisitedNeighbors);

            // Add unvisited neighbors to the bfsQueue and mark them as visited
            for (int neighbor : unvisitedNeighbors) {
                bfsQueue.add(neighbor);
                visited.add(neighbor);
            }
        }

        //  If the graph is not connected, it's possible that not all nodes will be visited and added to lexOrder
        //  which would result in lexOrder.size() being less than am.size().

        return lexOrder;
    }

    public static ArrayList<Integer> check(ArrayList<ArrayList<Integer>> am, ArrayList<Integer> order){
        if (am.size() != order.size()){
            for (int i = 0; i < am.size(); i++){
                if (!order.contains(i)){
                    order.add(i);
                }
            }
        }
        return order;
    }


    public static ArrayList<ArrayList<Integer>> initGraph(int v) {
        ArrayList<ArrayList<Integer>> am = new ArrayList<>(v);
        for (int i = 0; i < v; i++) {
```

```
573             am.add(new ArrayList<>());
574         }
575         return am;
576     }
577 
578 
579 
580     public static void main(String[] args){
581         int[] vertexCounts = new int[50];
582         vertexCounts[0] = 10;
583         for (int i = 1; i < vertexCounts.length; i++){
584             vertexCounts[i] = vertexCounts[i-1]+vertexCounts[0];
585         }
586 //        int[] vertexCounts = {10, 50, 100, 200, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 1000
587 
588         long[] executionTimesSLO = new long[vertexCounts.length];
589         long[] executionTimesSODL = new long[vertexCounts.length];
590         long[] executionTimeUNIF = new long[vertexCounts.length];
591         long[] executionTimeBFS = new long[vertexCounts.length];
592         long[] executionTimeDFS = new long[vertexCounts.length];
593         long[] executionTimelexBFS = new long[vertexCounts.length];
594         int[] colorsSLO = new int[vertexCounts.length];
595         int[] colorsSODL = new int[vertexCounts.length];
596         int[] colorsUNIF = new int[vertexCounts.length];
597         int[] colorsBFS = new int[vertexCounts.length];
598         int[] colorsDFS = new int[vertexCounts.length];
599         int[] colorslexBFS = new int[vertexCounts.length];
600         int colorsSLOmax = -1;
601         int colorsSODLmax = -1;
602         int colorsUNIFmax = -1;
603         int colorsBFSmax = -1;
604         int colorsDFSmax = -1;
605         int colorslexBFSmax = -1;
606 
607 
608         for (int i = 0; i < vertexCounts.length; i++) {
609             int V = vertexCounts[i];
610             ArrayList<ArrayList<Integer>> am = initGraph(V);
611 //            ArrayList<ArrayList<Integer>> am2 = initGraph(V);
612 //            ArrayList<ArrayList<Integer>> am3 = initGraph(V);
613 //            ArrayList<ArrayList<Integer>> am4 = initGraph(V);
614 //            ArrayList<ArrayList<Integer>> am5 = initGraph(V);
615 //            ArrayList<ArrayList<Integer>> am6 = initGraph(V);
616 
617 //            ArrayList<Integer> order = new ArrayList<>();
618 //            for (int j = 0; j < V; j++) {
619 //                am.add(new ArrayList<>());
620 //                order.add(j);
621 //            }
622 
623             //Uniform.uniformRandom(am, V, V/8);
624             //Uniform.uniformRandom(am, V, 2*V);
625 
626             //Complete.allEdge(am, V);
627             //Cycle.allEdgeCycle(am, V);
628             //Uniform.uniformRandom(am, V, V);
629             //Skewed.skewedRandom(am, V, V/2);
```

```
630             Gauss.gaussRandom(am, V, V/2);
631
632
633             long startTime, endTime;
634
635             startTime = System.nanoTime();
636             ArrayList<Integer> order6 = lexBFS(am, 0);
637             endTime = System.nanoTime();
638             colorslexBFS[i] = color(am, check(am,order6));
639             if (colorslexBFS[i] > colorslexBFSmax){
640                 colorslexBFSmax = colorslexBFS[i];
641             }
642             executionTimelexBFS[i] = endTime - startTime;
643
644
645             startTime = System.nanoTime();
646             //color(am, order);
647             ArrayList<Integer> order5 = DFSOrder(am, 0);
648             colorsDFS[i] = color(am, check(am,order5));
649             if (colorsDFS[i] > colorsDFSmax){
650                 colorsDFSmax = colorsDFS[i];
651             }
652             endTime = System.nanoTime();
653             executionTimeDFS[i] = endTime - startTime;
654
655             startTime = System.nanoTime();
656             //color(am, order);
657             ArrayList<Integer> order4 = BFSOrder(am, 0);
658             colorsBFS[i] = color(am, check(am,order4));
659             if (colorsBFS[i] > colorsBFSmax){
660                 colorsBFSmax = colorsBFS[i];
661             }
662             endTime = System.nanoTime();
663             executionTimeBFS[i] = endTime - startTime;
664
665             startTime = System.nanoTime();
666             //color(am, order);
667             ArrayList<Integer> order3 = uniformrandom(am);
668             colorsUNIF[i] = color(am, check(am,order3));
669             if (colorsUNIF[i] > colorsUNIFmax){
670                 colorsUNIFmax = colorsUNIF[i];
671             }
672             endTime = System.nanoTime();
673             executionTimeUNIF[i] = endTime - startTime;
674 //
675             startTime = System.nanoTime();
676             //color(am, order);
677             ArrayList<Integer> order2 = smallestOriginalDegreeLast(am);
678             colorsSODL[i] = color(am, check(am,order2));
679             if (colorsSODL[i] > colorsSODLmax){
680                 colorsSODLmax = colorsSODL[i];
681             }
682             endTime = System.nanoTime();
683             executionTimesSODL[i] = endTime - startTime;
684 //
685 //
686             startTime = System.nanoTime();
```

```
687             //color(am, order);
688             ArrayList<Integer> order1 = smallestLastVertexOrdering(am);
689             colorsSLO[i] = color(am, check(am,order1));
690             if (colorsSLO[i] > colorsSLOmax){
691                 colorsSLOmax = colorsSLO[i];
692             }
693             endTime = System.nanoTime();
694             executionTimesSLO[i] = endTime - startTime;
695
696
697
698         }
699         List<int[]> datastes = new ArrayList<>();
700         datastes.add(colorslexBFS);
701         datastes.add(colorsDFS);
702         datastes.add(colorsBFS);
703         datastes.add(colorsUNIF);
704         datastes.add(colorsSODL);
705         datastes.add(colorsSLO);
706         List<String> labels = new ArrayList<>();
707         labels.add("LesBFS");
708         labels.add("DFS");
709         labels.add("BFS");
710         labels.add("UNIF");
711         labels.add("SODL");
712         labels.add("SLO");
713 //         System.out.println(Arrays.toString(executionTimes));
714 //         plotResults(vertexCounts, executionTimes);
715
716
717
718         // Compare color
719
720         System.out.println("colorsSLOmax: " + colorsSLOmax);
721         System.out.println("colorsSODLmax: " + colorsSODLmax);
722         System.out.println("colorsUNIFmax: " + colorsUNIFmax);
723         System.out.println("colorsBFSmax: " + colorsBFSmax);
724         System.out.println("colorsDFSmax: " + colorsDFSmax);
725         System.out.println("colorslexBFSmax: " + colorslexBFSmax);
726
727         //plotResults2("Color compare complete graph", vertexCounts, datastes, labels);
728         //plotResults2("color compare circle graph", vertexCounts, datastes, labels);
729         //plotResults2("Color compare uniform graph", vertexCounts, datastes,labels);
730         //plotResults2("color compare skewed graph ", vertexCounts, datastes, labels);
731         //plotResults2("color compare guass graph", vertexCounts, datastes, labels);
732
733
734
735         // Time
736         List<long[]> executionTimesList = new ArrayList<>();
737         executionTimesList.add(executionTimesSLO);
738         executionTimesList.add(executionTimesSODL);
739         executionTimesList.add(executionTimeUNIF);
740         executionTimesList.add(executionTimeBFS);
741         executionTimesList.add(executionTimeDFS);
742         executionTimesList.add(executionTimelexBFS);
743         List<String> labels2 = new ArrayList<>();
```

```
744             labels2.add("SLO");
745             labels2.add("SODL");
746             labels2.add("UNIF");
747             labels2.add("BFS");
748             labels2.add("DFS");
749             labels2.add("lexBFS");
750 
751             //plotResults3("Complete graph with different order Time", vertexCounts,executionTimesList,labels2);
752             //plotResults3("Circle graph with different order Time", vertexCounts, executionTimesList,labels2);
753             //plotResults3("UNIForm graph with different order Time", vertexCounts, executionTimesList,labels2);
754             //plotResults3("Skewed uniform graph with different order Time", vertexCounts, executionTimesList,labels2
755             plotResults3("guass uniform graph with different order Time", vertexCounts, executionTimesList,labels2);
756 
757         }
758     }
759 
760 
```

References

[1] URL: https://www.programiz.com/dsa/graph-adjacency-list.

[2] URL: https://www.programiz.com/dsa/graph-bfs.

[3] URL: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/.

[4] URL: https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/.

[5] URL: https://zetcode.com/java/jfreechart/.

[6] F.F. Dragan A. Brandstadt and F. Nicolai. “LexBFS-orderings and powers of chordal graphs, Disc”. In: *Math* 171 (1997).