CS7350 Project--Graph Coloring Analysis Project

StudentName:Liangchao Cai
StudentId:48389579

1.Extra Credit

Base=11

Modulus=109

I send to you the value 48

You are to pick a value>50

Q1:Tell me what value you send to me

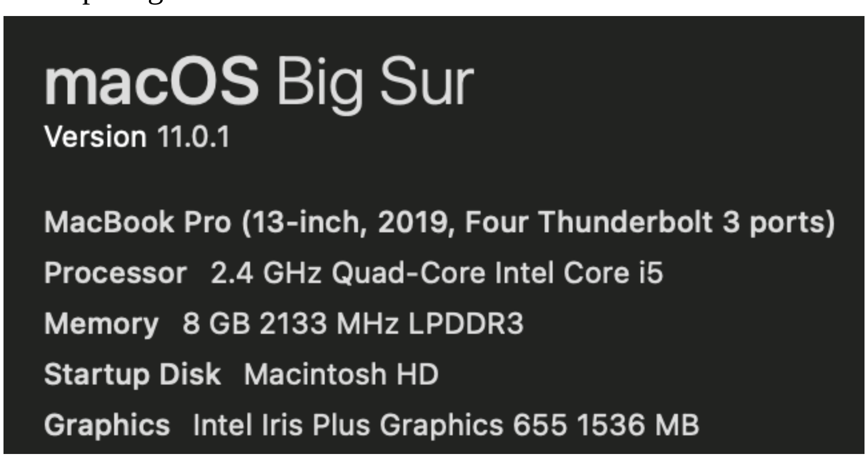
I pick a value of 64, then

11^64%109=35^16%109=26^8%109=22^4%109=15, thus I'll send a value of 15 to you.

Q2:Tell me our secret key

The secret key is 48^64%109=15^32%109=49^8%109=3^4%109=81

2.computing environment



Programming Language: go version go1.15 darwin/amd64
It's my working laptop. Because it's not a pure computational environment, there will be some unintended spikes in the graph, however, they have minor impact to the calculation.

Project Part1:

1.Random graph of different distribution
Before starting to generate the random graph, I write the skewed distribution pseudo-random number generator and another pseudo-random number generator which I call it "Amor" based on the built-in uniform random number generator.
Skewed distribution random generator: for input value x, the generator will output a integer between 1 and x. For input x, I assume a set there are x copies of 1, x-1 copies of 2, x-2 copies of 3, x-4 copies of 4, ... 2 copies of x-1,1 copies of x, thus
the probability of 1 = 2*x/(x*(x+1))
the probability of 2 = 2*(x-1)/(x*(x+1))
the probability of 3 = 2*(x-2)/(x*(x+1))
the probability of x = 2/(x*(x+1))
Then I use uniform random generator to produce a value "k", by locating the position of k in this set I can get the result number. This is a O(n) function since I trade time for space. However, I could get O(1) in this function if I priorly set a big array which has the same set, and use uniform random generator to produce a location number to get it. **For the convenience of analysis and comparing, I see it as O(1) in this report.**

```
/**

generate skewed random number between 1 and max

deprecated because of its low efficient

*/

func skewedRandomNumber(max int) int {

	v := max * (max + 1) / 2

	k := uniformRandomNumber(v)

	m := 0

	for i := 1; i <= max; i++ {
```

```
            m += max - i + 1
            if k <= m {
                return i
            }
        }
        return max
}
```

My own distribution of random number generator, I call it "Amor random number generator", though the idea was copied from professor you talked in the class. For input number x, it has a probability of 0.8 to produce a number between 1 and 0.2*x, and probability of 0.2 to produce a number between 0.2*x and x. It's a O(1) function.

```
//create my own random number between 1 and max
func amorRandomNumber(max int) int {
        if max == 1 {
            return 1
        }
        a := uniformRandomNumber(10)
        if a > 2 {
            if max < 5 {
                return 1
            } else {
                return uniformRandomNumber(max / 5)
```

```
        }
    } else {
        return max/5 + uniformRandomNumber(max*4/5)
    }
}
```

Now I have three different distribution "dice" here, so I just throw the dice and get the number as an index of the set, which have all ascending order pairs of vertices(or edges/conflicts) in a graph. I set a threshold, 1/3 density, if the density is greater than the threshold, it'll use the uniform distribution random number generator no matter what distribution it originally is, for purpose of efficiency and functionality.
I use an auxiliary array which I call it adjacent matrix to avoid the duplicate edges and to determine if two vertices are adjacent to each in O(1).

```
func createRandomGraphWithSkewedDistribution(v int, e int) ([]*Vertex, [][]int, []*Vertex) {
    if v < 2 {
        panic("v should greater than 1")
    }
    graph, adjacentMatrix := initGraph(v)
    maxLength := v * (v - 1) / 2
    points := make([]*Point, maxLength)
    count := 0
    for i := 0; i < v; i++ {
        for j := i + 1; j < v; j++ {
            points[count] = &Point{x: i, y: j}
            count++
```

```
        }
    }
    if e*3 > maxLength {
        sequence := randomNumberSequenceGenerator(e, maxLength, Skewed)
        for i := 0; i < len(sequence); i++ {
            graph[points[sequence[i]-1].x].addAdjacentVertex(points[sequence[i]-1].y + 1)
            graph[points[sequence[i]-1].y].addAdjacentVertex(points[sequence[i]-1].x + 1)
            adjacentMatrix[points[sequence[i]-1].x][points[sequence[i]-1].y] = 1
            adjacentMatrix[points[sequence[i]-1].y][points[sequence[i]-1].x] = 1
        }
    } else {
        icount := 0
        for i := 0; i < e; i++ {
            index := skewedRandomNumber(maxLength) - 1
            if adjacentMatrix[points[index].x][points[index].y] != 1 {
                graph[points[index].x].addAdjacentVertex(points[index].y + 1)
                graph[points[index].y].addAdjacentVertex(points[index].x + 1)
                adjacentMatrix[points[index].x][points[index].y] = 1
                adjacentMatrix[points[index].y][points[index].x] = 1
            } else {
```

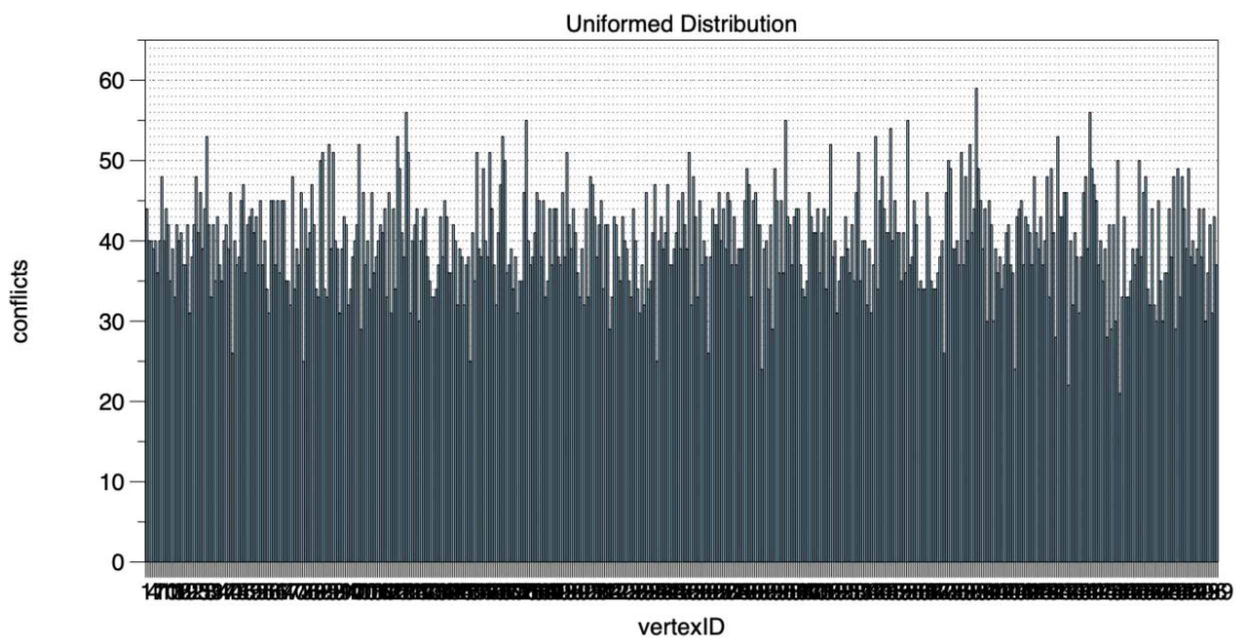
```go
				i--
			}
			icount++
		}
	}
	sameCurrentDegreeList := initSDgreeList(graph)
	return graph, adjacentMatrix, sameCurrentDegreeList
}
func createRandomGraphWithAmorDistribution(v int, e int) ([]*Vertex, [][]int, []*Vertex) {
	if v < 2 {
		panic("v should greater than 1")
	}
	graph, adjacentMatrix := initGraph(v)
	maxLength := v * (v - 1) / 2
	points := make([]*Point, maxLength)
	count := 0
	for i := 0; i < v; i++ {
		for j := i + 1; j < v; j++ {
			points[count] = &Point{x: i, y: j}
			count++
```

```
        }
    }
    if e*3 > maxLength {
        sequence := randomNumberSequenceGenerator(e, maxLength, Amor)
        for i := 0; i < len(sequence); i++ {
            graph[points[sequence[i]-1].x].addAdjacentVertex(points[sequence[i]-1].y + 1)
            graph[points[sequence[i]-1].y].addAdjacentVertex(points[sequence[i]-1].x + 1)
            adjacentMatrix[points[sequence[i]-1].x][points[sequence[i]-1].y] = 1
            adjacentMatrix[points[sequence[i]-1].y][points[sequence[i]-1].x] = 1
        }
    } else {
        icount := 0
        for i := 0; i < e; i++ {
            index := amorRandomNumber(maxLength) - 1
            if adjacentMatrix[points[index].x][points[index].y] != 1 {
                graph[points[index].x].addAdjacentVertex(points[index].y + 1)
                graph[points[index].y].addAdjacentVertex(points[index].x + 1)
                adjacentMatrix[points[index].x][points[index].y] = 1
                adjacentMatrix[points[index].y][points[index].x] = 1
```

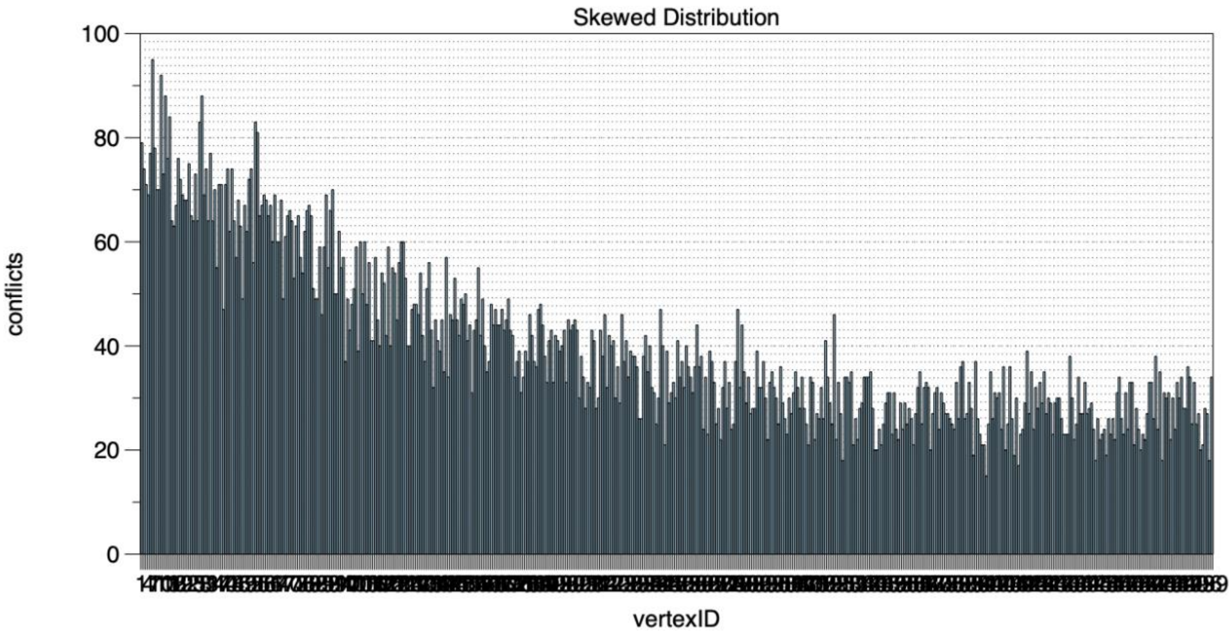
```
            } else {
                i--
            }
            icount++
        }
        // fmt.Printf("\nin fact, it runs %d times\n", icount)
    }
    sameCurrentDegreeList := initSDgreeList(graph)
    return graph, adjacentMatrix, sameCurrentDegreeList
}
```

To make sure the functionality of three different distribution of random graph, here is the histogram showing how many conflicts each vertex has for each method:

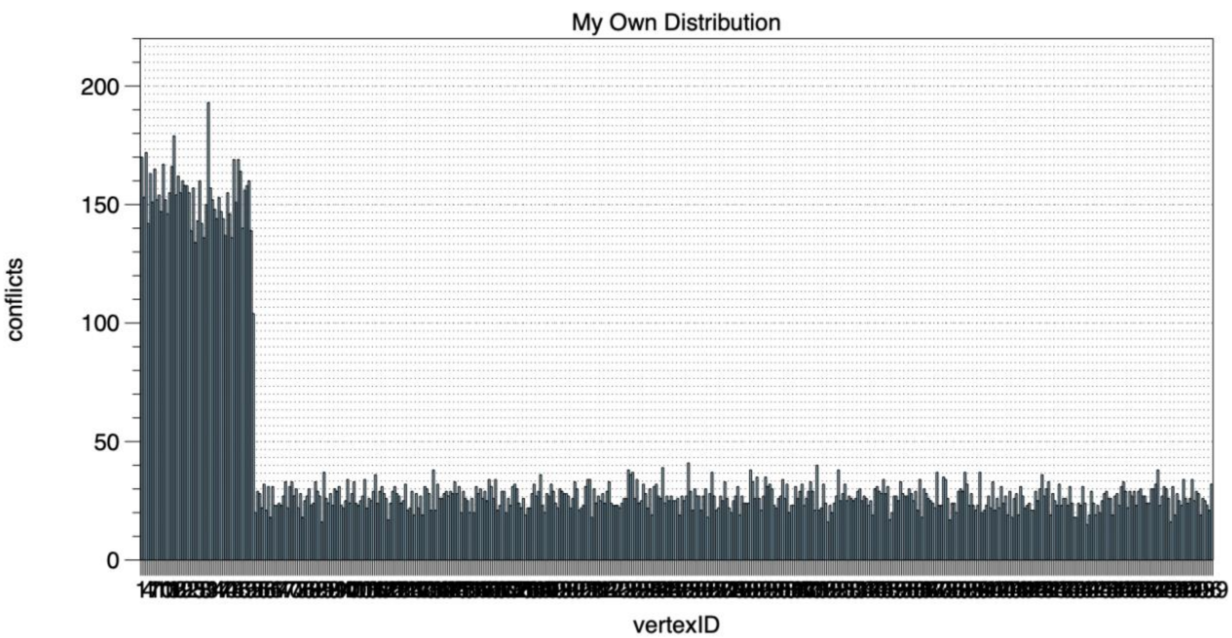
1.Uniform Distribution(v=500,e=10000,density=8%)



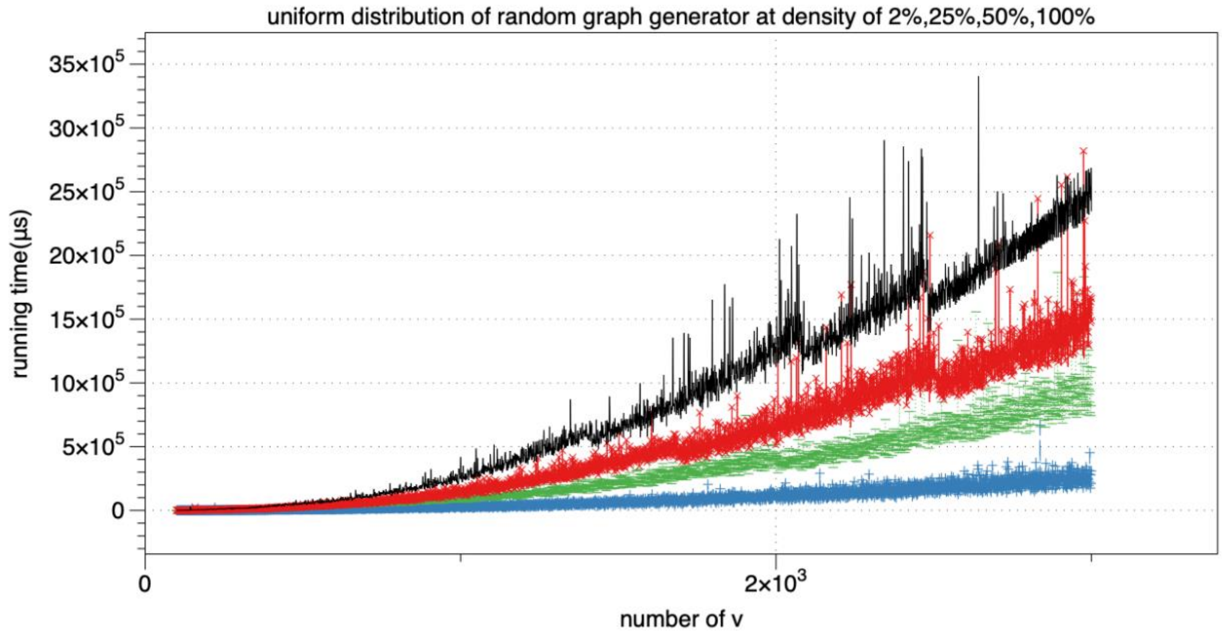
2.Skewed Distribution(v=500,e=10000,density=8%)



3.Own Distribution(v=500,e=10000,density=8%)



Here is a comparison of different density of uniform distribution random graph



density of 2% blue x density of 25% green - density of 50% red x density of 100% black line

uVertex(2%)	uTime(2%)	uVertex(25%)	uTime(25%)	uVertex(50%)	uTime(50%)	uVertex(100%)	uTime(100%)
100	336	100	688	100	707	100	1161
200	680	200	2024	200	4891	200	8584
300	2337	300	5003	300	13828	300	19727
400	6802	400	11024	400	20490	400	33366
500	7488	500	19644	500	33478	500	57453
600	8899	600	25557	600	52489	600	71014
700	8016	700	36513	700	63963	700	124136
800	20790	800	50540	800	80607	800	147573
900	23879	900	77247	900	113319	900	247763
1000	34827	1000	72926	1000	**159811**	1000	416638
1100	34158	1100	99525	1100	171650	1100	329458
1200	50577	1200	104140	1200	218460	1200	485331
1300	46220	1300	157877	1300	300993	1300	571652
1400	54356	1400	196414	1400	342791	1400	598402
1500	57075	1500	202156	1500	374368	1500	622262
1600	65479	1600	221575	1600	430057	1600	817392
1700	68457	1700	342487	1700	422681	1700	908969
1800	128775	1800	392359	1800	502629	1800	927514
1900	104159	1900	318964	1900	622810	1900	1196306
2000	105661	2000	419851	2000	**705368**	2000	1128540

2100	158944	2100	349580	2100	627259	2100	1336405
2200	134358	2200	426519	2200	974565	2200	1354053
2300	135237	2300	481067	2300	971090	2300	1565328
2400	133083	2400	602169	2400	994133	2400	1705733
2500	213420	2500	589894	2500	1392904	2500	1707753
2600	207080	2600	781338	2600	1056627	2600	1869239
2700	167618	2700	895667	2700	1075627	2700	2041237
2800	206947	2800	745080	2800	1413429	2800	2210581
2900	272460	2900	1076354	2900	1281454	2900	2181082
3000	309751	3000	1119859	3000	1662023	3000	2346301

it's asymptotic time is O(v^2)

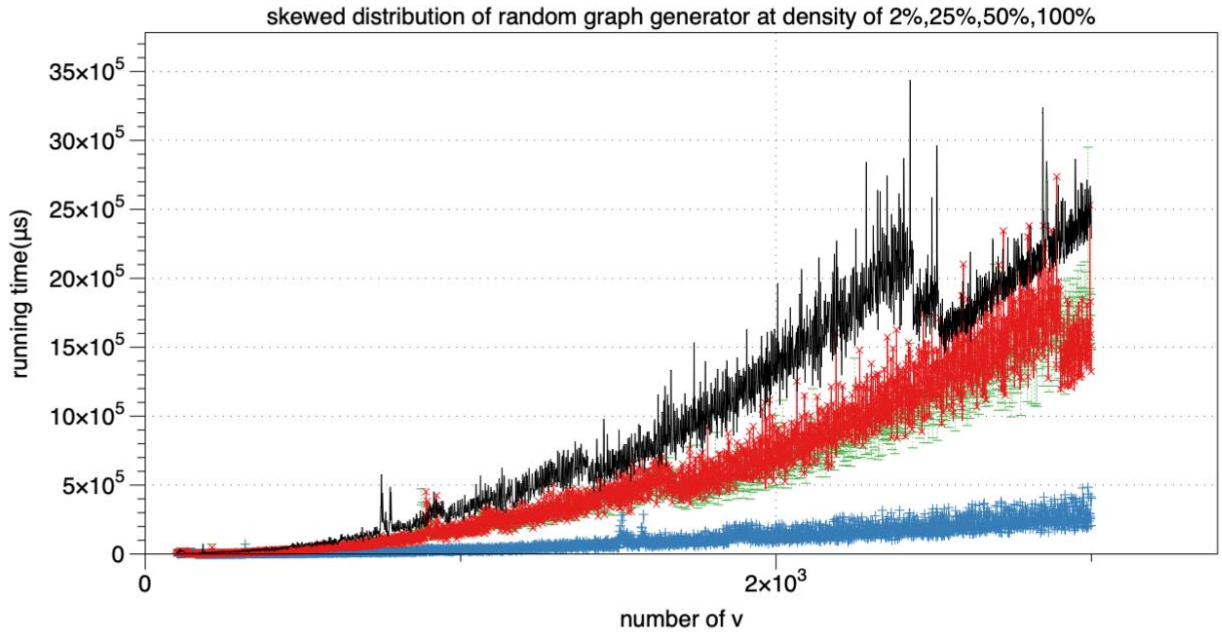
When v=1000 and density=50%, the running time is 159811

when v=2000 and density=50%, the running time is 705368, is roughly 4 times 159811

```
/**

create random graph with uniform distribution

*/

func createRandomGraphWithUniformDistribution(v int, e int) ([]*Vertex, [][]int, []*Vertex) {

	if v < 2 {

		panic("v should greater than 1")

	}

	graph, adjacentMatrix := initGraph(v)

	maxLength := v * (v - 1) / 2

	points := make([]*Point, maxLength)

	count := 0

	for i := 0; i < v; i++ {

		for j := i + 1; j < v; j++ {

			// fmt.Printf("count is %d \n", count)
```

```
                points[count] = &Point{x: i, y: j}

                count++

            }

        }

        sequence := randomNumberSequenceGenerator(e, maxLength, Uniform)

        for i := 0; i < len(sequence); i++ {

            graph[points[sequence[i]-1].x].addAdjacentVertex(points[sequence[i]-1].y + 1)

            graph[points[sequence[i]-1].y].addAdjacentVertex(points[sequence[i]-1].x + 1)

            adjacentMatrix[points[sequence[i]-1].x][points[sequence[i]-1].y] = 1

            adjacentMatrix[points[sequence[i]-1].y][points[sequence[i]-1].x] = 1

        }

        sameCurrentDegreeList := initSDgreeList(graph)

        return graph, adjacentMatrix, sameCurrentDegreeList

}
```

Here is a comparison of different density of skewed distribution random graph

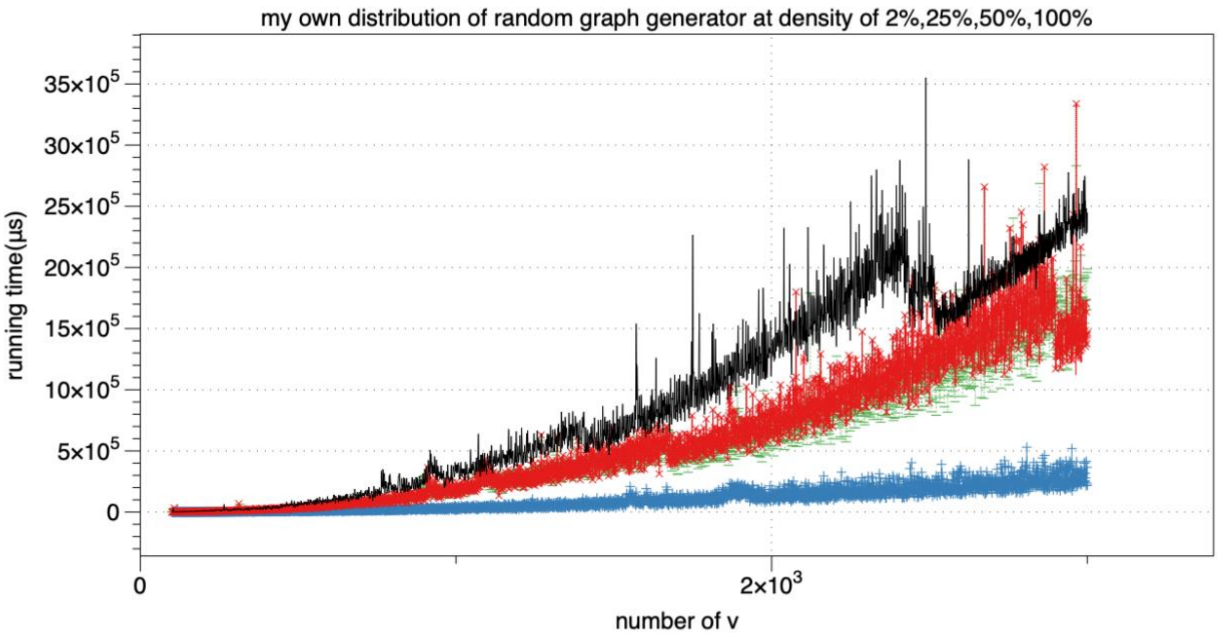


density of 2% blue x density of 25% green - density of 50% red x density of 100% black line

sVertex(2%)	sTime(2%)	sVertex(25%)	sTime(25%)	sVertex(50%)	sTime(50%)	sVertex(100%)	sTime(100%)
100	3778	100	6549	100	8852	100	10437
200	786	200	2909	200	3162	200	7089
300	1758	300	14104	300	14647	300	18844
400	7622	400	29420	400	15285	400	44119
500	7584	500	40806	500	33176	500	52221
600	9473	600	61070	600	49975	600	113036
700	10890	700	54223	700	48761	700	144182
800	19532	800	124636	800	76378	800	174393
900	33322	900	138903	900	199733	900	255006
1000	17729	1000	211874	1000	177246	1000	275437
1100	22536	1100	264756	1100	316484	1100	440128
1200	39481	1200	270470	1200	276369	1200	419527
1300	41476	1300	242974	1300	296000	1300	522595
1400	37819	1400	471024	1400	343167	1400	688078
1500	42713	1500	515968	1500	486120	1500	621744
1600	81133	1600	617303	1600	412494	1600	757536
1700	146321	1700	495967	1700	393874	1700	874619
1800	120721	1800	551772	1800	531738	1800	1008266
1900	110299	1900	453387	1900	750893	1900	1284672
2000	95700	2000	647454	2000	800692	2000	1419670
2100	112076	2100	647594	2100	883500	2100	1505154

2200	181993	2200	800099	2200	847073	2200	1766792
2300	238876	2300	972789	2300	1128009	2300	1729933
2400	174799	2400	1338487	2400	1283410	2400	2058438
2500	175632	2500	975532	2500	1349917	2500	1925486
2600	252757	2600	1294989	2600	1434298	2600	1717738
2700	210605	2700	1197524	2700	1804537	2700	1819265
2800	226133	2800	1447304	2800	1412386	2800	2125004
2900	232461	2900	1731148	2900	1694000	2900	2344325
3000	403372	3000	1481337	3000	1509328	3000	2285661

Here is a comparison of different density of my own distribution random graph



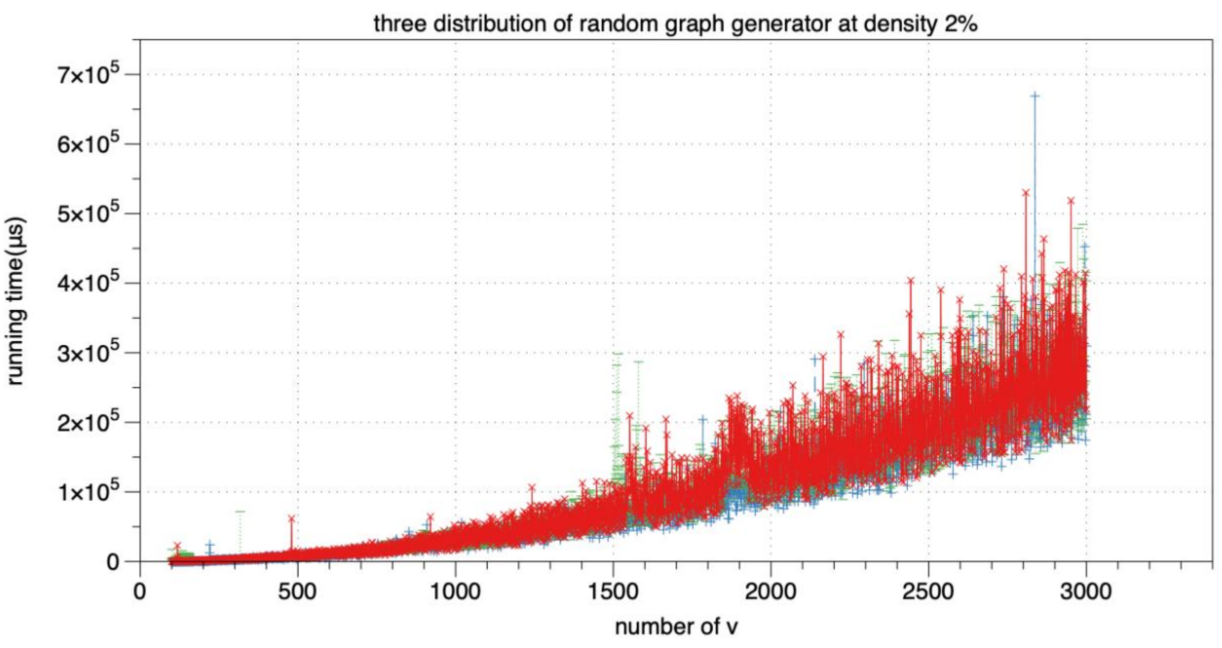
density of 2% blue x density of 25% green - density of 50% red x density of 100% black line

At low density, for example, 2%, it's asymptotic time is O(v^2)

aVertex(2%)	aTime(2%)	aVertex(25%)	aTime(25%)	aVertex(50%)	aTime(50%)	aVertex(100%)	aTime(100%)
100	**347**	100	1347	100	1074	100	1452
200	505	200	4841	200	4094	200	11025
300	2905	300	9034	300	7609	300	16750
400	**5741**	400	26264	400	33775	400	34821
500	5982	500	38682	500	31353	500	54654
600	8523	600	60032	600	54894	600	93348
700	10478	700	97679	700	71159	700	155813
800	21882	800	104433	800	145965	800	189113
900	22407	900	298828	900	122910	900	233708

1000	43129	1000	187492	1000	167887	1000	341918
1100	50218	1100	224040	1100	272544	1100	477602
1200	22974	1200	256051	1200	288831	1200	448449
1300	53717	1300	372962	1300	258169	1300	710205
1400	87031	1400	437467	1400	314573	1400	653842
1500	52469	1500	474067	1500	311053	1500	605752
1600	95312	1600	475434	1600	617011	1600	742010
1700	142835	1700	494132	1700	513410	1700	873969
1800	121383	1800	552538	1800	555647	1800	1162968
1900	199303	1900	475703	1900	614967	1900	1138211
2000	132895	2000	636580	2000	807267	2000	1262288
2100	117662	2100	741925	2100	941755	2100	1369889
2200	119016	2200	831041	2200	889160	2200	1735763
2300	148839	2300	785172	2300	1106039	2300	2139307
2400	176856	2400	1040410	2400	1213910	2400	2447125
2500	242936	2500	1206335	2500	891530	2500	2357849
2600	349126	2600	1080249	2600	1404788	2600	1744402
2700	249220	2700	1491830	2700	1223006	2700	2089446
2800	218708	2800	1392212	2800	1532998	2800	2184750
2900	203173	2900	1613430	2900	1170909	2900	2341074
3000	219090	3000	1987454	3000	1465015	3000	2447354

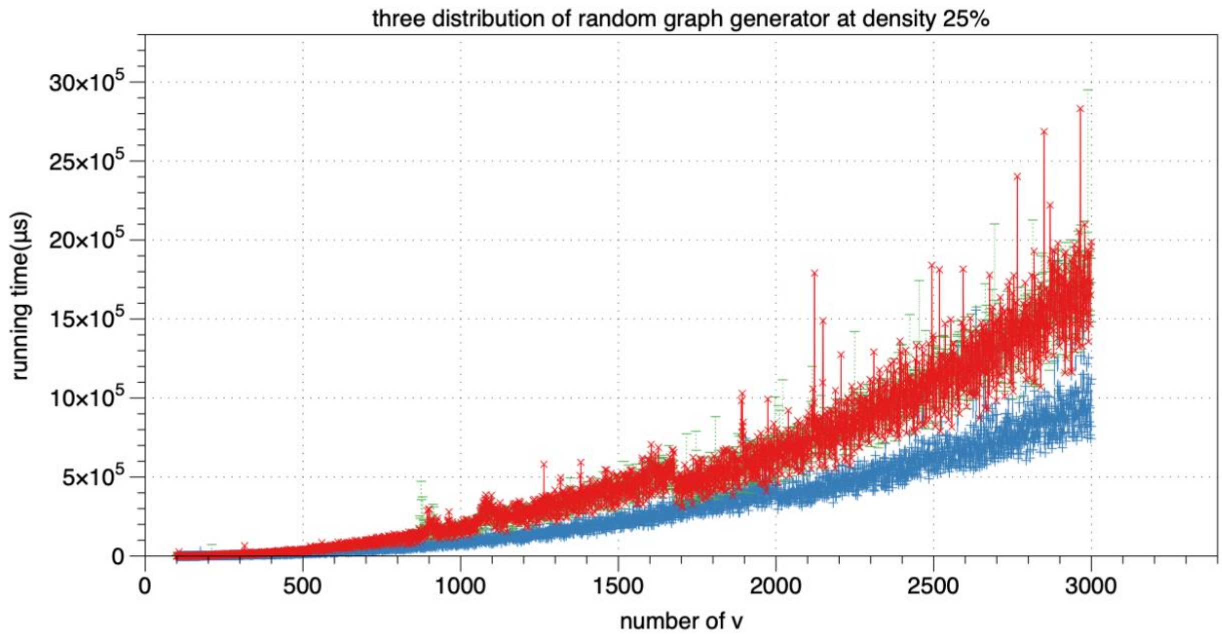
comparison of three distribution random graph generator at density of 2%



uniform blue +   skewed green -   my own red x

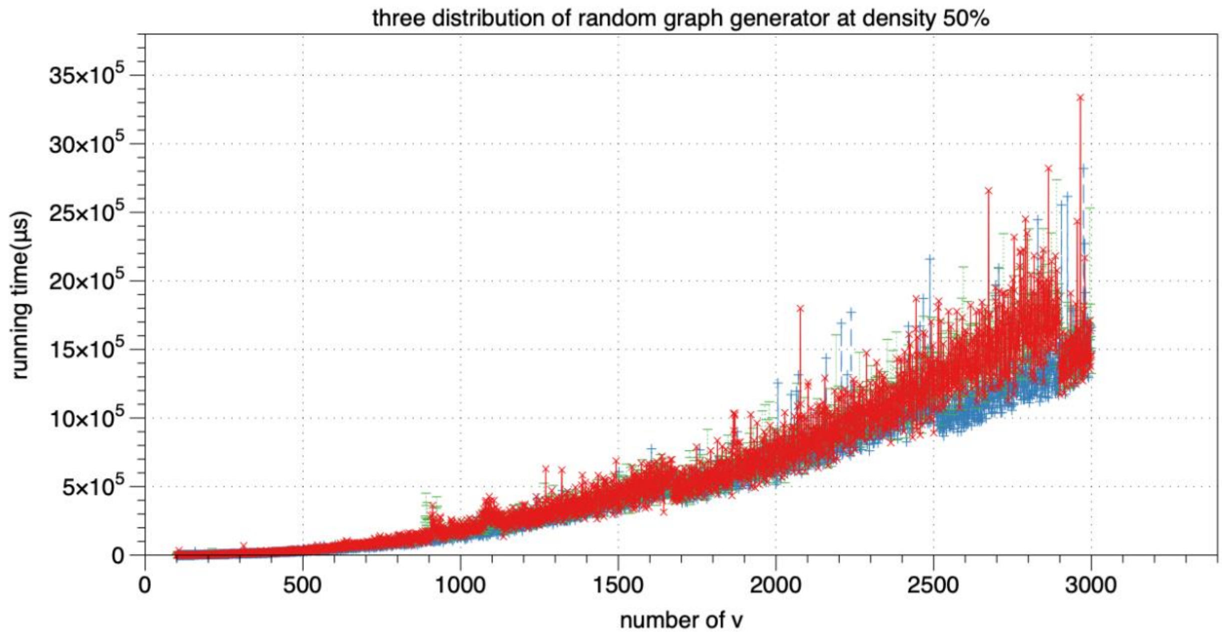
At low density, the performances of these three distribution random graph generators are roughly the same. They are all O(v^2+e), but since e is too low here, it's O(v^2).

comparison of three distribution random graph generator at density of 25%

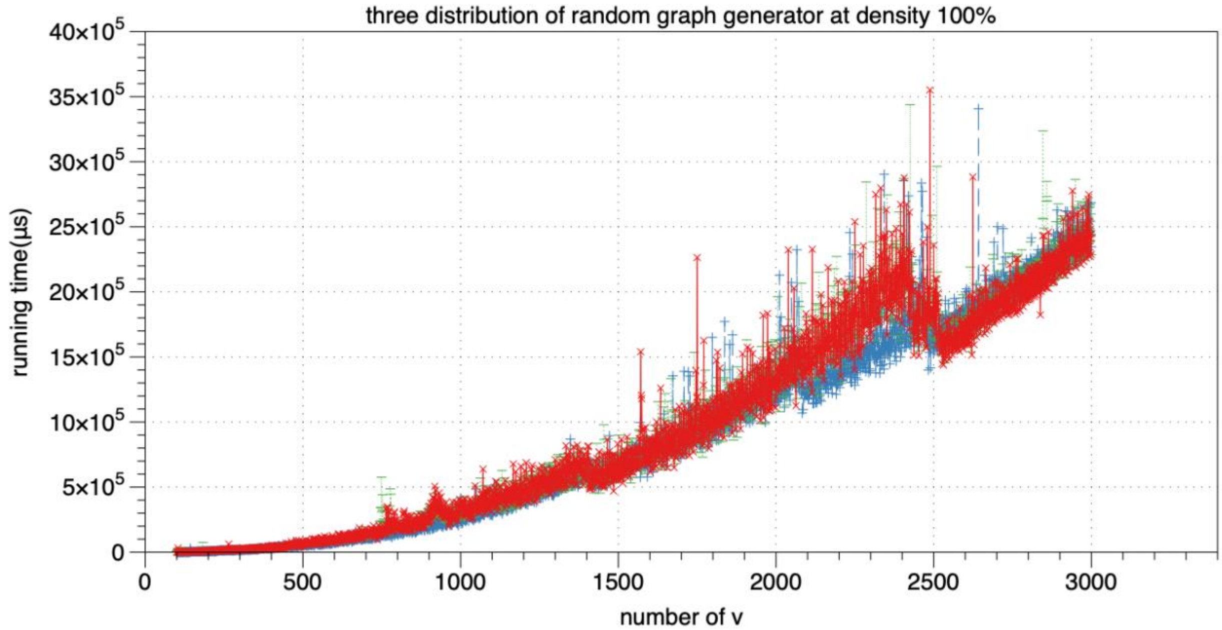


uniform blue + skewed green - my own red x

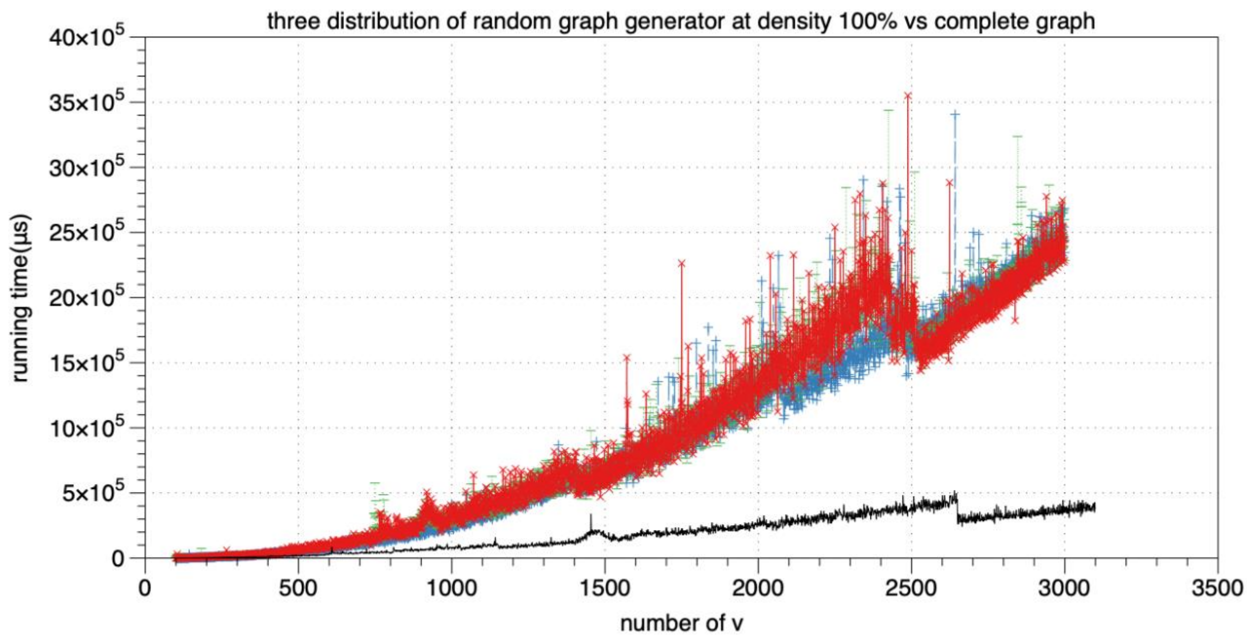
comparison of three distribution random graph generator at density of 50%



uniform blue + skewed green - my own red x comparison of three distribution random graph generator at density of 100%



uniform blue + skewed green - my own red x



uniform blue + skewed green - my own red x complete graph generator black line

Conclusion:

1.At density of 2%, these three generators are roughly the same(Maybe the number of vertices are too low to give an impact on the performance) since they are O(v^2)

2.At density of 25%, the uniform one would be better since it didn't do extra loop to avoid duplicate random number. Besides, the skewed one and my own one are not O(v^2)
3.At density of 50%, these are roughly the same because they are all uniform generators with different dices.(I generate an array of x, then throw the dice and get the number as index, then shift it with the last element of the array. Actually, I think it's shuffling in some sense)
4.Explanation for the dropping at density 100% graph: I ran the same generator concurrently among different densities, since 2% and 25% ran faster than 50% and 100%, thus the later two got extra resource from cpu and improved their performance.
5.The performance of complete graph generator is more better than other three random graph generator if I want the complete graph.
6. Since the complete graph and cycle has talked in Hw3, I would not repeat here.

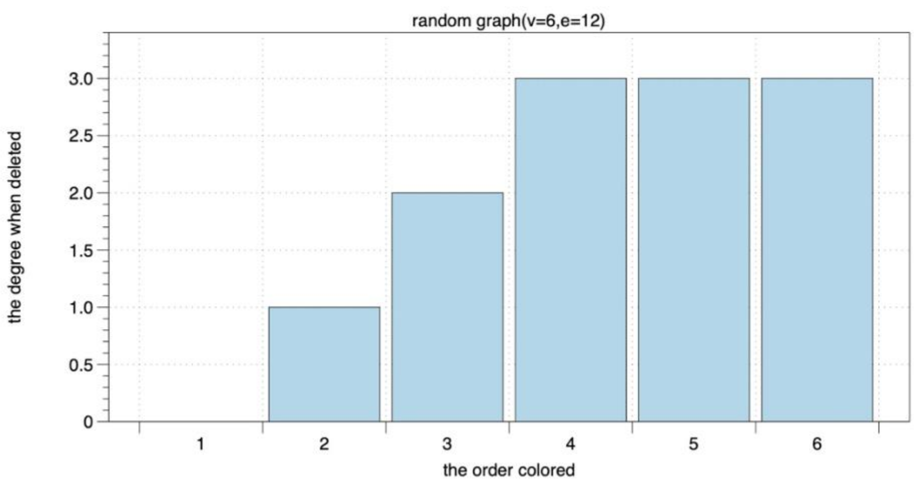
Project Part2.

1. Two examples from coloring random graph and complete graph by the smallest last vertex ordering and required additional output

Output an example of a random graph(v=6,e=12) with uniform distribution
6 #0th value=Number of vertices
7 #1th value=starting location for vertex 1's edges
10 #2th value=starting location for vertex 2's edges
14 #3th value=starting location for vertex 3's edges
18 #4th value=starting location for vertex 4's edges
22 #5th value=starting location for vertex 5's edges
26 #6th value=starting location for vertex 6's edges
2 #7th value=Vertex 1 is adjacent to Vertex 2
3 #8th value=Vertex 1 is adjacent to Vertex 3
6 #9th value=Vertex 1 is adjacent to Vertex 6
6 #10th value=Vertex 2 is adjacent to Vertex 6
4 #11th value=Vertex 2 is adjacent to Vertex 4
1 #12th value=Vertex 2 is adjacent to Vertex 1
5 #13th value=Vertex 2 is adjacent to Vertex 5
1 #14th value=Vertex 3 is adjacent to Vertex 1
5 #15th value=Vertex 3 is adjacent to Vertex 5
4 #16th value=Vertex 3 is adjacent to Vertex 4
6 #17th value=Vertex 3 is adjacent to Vertex 6
5 #18th value=Vertex 4 is adjacent to Vertex 5
2 #19th value=Vertex 4 is adjacent to Vertex 2
3 #20th value=Vertex 4 is adjacent to Vertex 3
6 #21th value=Vertex 4 is adjacent to Vertex 6
4 #22th value=Vertex 5 is adjacent to Vertex 4

2 #23th value=Vertex 5 is adjacent to Vertex 2
3 #24th value=Vertex 5 is adjacent to Vertex 3
6 #25th value=Vertex 5 is adjacent to Vertex 6
2 #26th value=Vertex 6 is adjacent to Vertex 2
5 #27th value=Vertex 6 is adjacent to Vertex 5
1 #28th value=Vertex 6 is adjacent to Vertex 1
4 #29th value=Vertex 6 is adjacent to Vertex 4
3 #30th value=Vertex 6 is adjacent to Vertex 3

```
➜  SourceCode git:(master) ✗ go run main.go
vertex 2 orginal degree 4 degree when deleted 0 color 1
vertex 4 orginal degree 4 degree when deleted 1 color 2
vertex 5 orginal degree 4 degree when deleted 2 color 3
vertex 6 orginal degree 5 degree when deleted 3 color 4
vertex 3 orginal degree 4 degree when deleted 3 color 1
vertex 1 orginal degree 3 degree when deleted 3 color 2
number of color used: 4
the maximum degree when deleted: 3
the size of terminal clique: 4
```

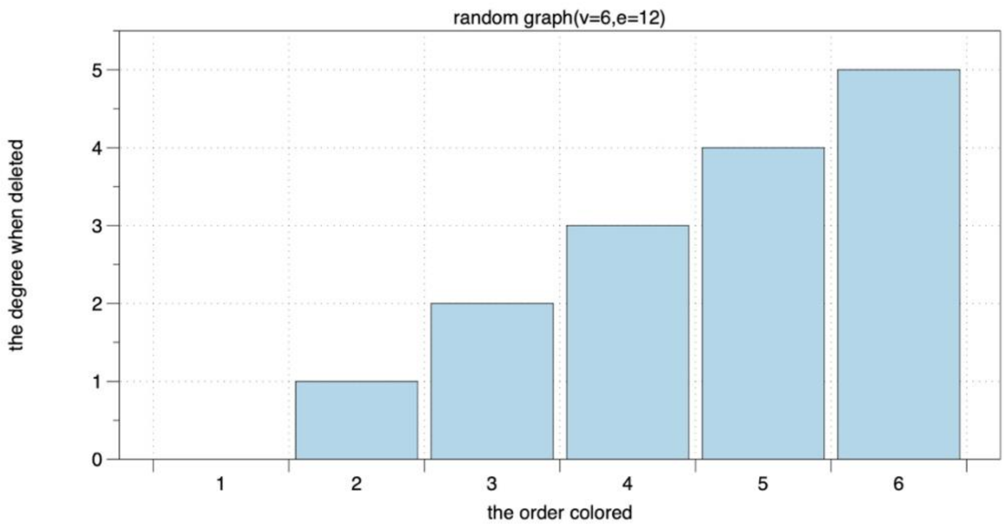


This is the second example, complete graph(v=6)

6 #0th value=Number of vertices
7 #1th value=starting location for vertex 1's edges
12 #2th value=starting location for vertex 2's edges
17 #3th value=starting location for vertex 3's edges
22 #4th value=starting location for vertex 4's edges
27 #5th value=starting location for vertex 5's edges
32 #6th value=starting location for vertex 6's edges
2 #7th value=Vertex 1 is adjacent to Vertex 2
3 #8th value=Vertex 1 is adjacent to Vertex 3

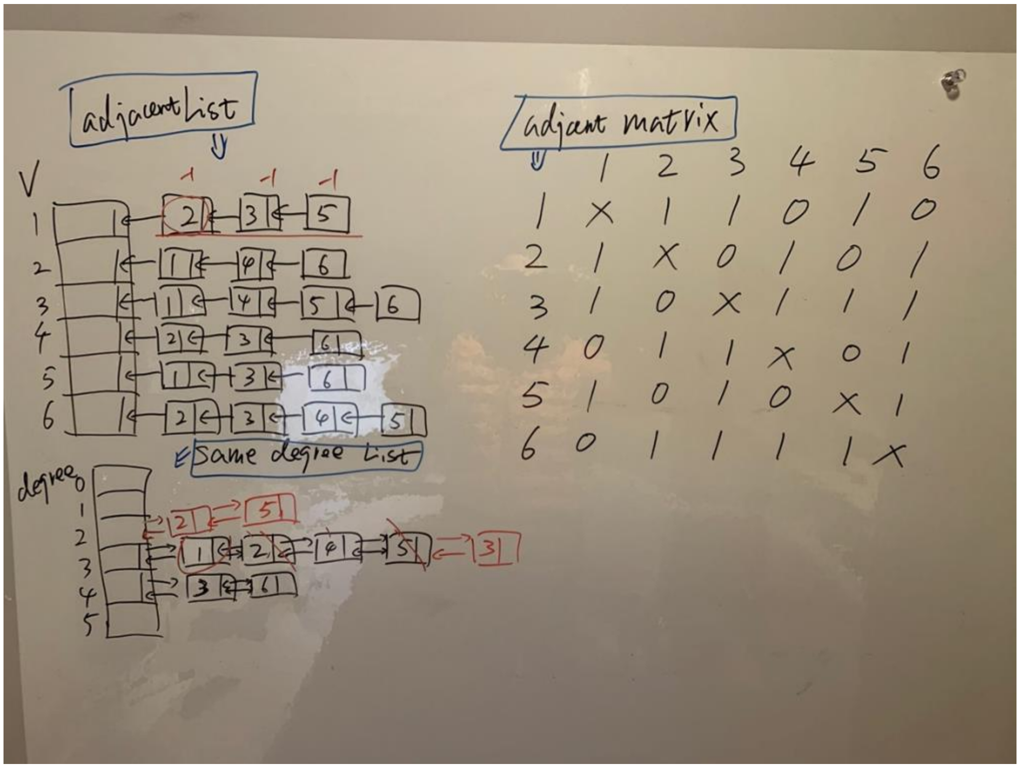
4 #9th value=Vertex 1 is adjacent to Vertex 4
5 #10th value=Vertex 1 is adjacent to Vertex 5
6 #11th value=Vertex 1 is adjacent to Vertex 6
1 #12th value=Vertex 2 is adjacent to Vertex 1
3 #13th value=Vertex 2 is adjacent to Vertex 3
4 #14th value=Vertex 2 is adjacent to Vertex 4
5 #15th value=Vertex 2 is adjacent to Vertex 5
6 #16th value=Vertex 2 is adjacent to Vertex 6
1 #17th value=Vertex 3 is adjacent to Vertex 1
2 #18th value=Vertex 3 is adjacent to Vertex 2
4 #19th value=Vertex 3 is adjacent to Vertex 4
5 #20th value=Vertex 3 is adjacent to Vertex 5
6 #21th value=Vertex 3 is adjacent to Vertex 6
1 #22th value=Vertex 4 is adjacent to Vertex 1
2 #23th value=Vertex 4 is adjacent to Vertex 2
3 #24th value=Vertex 4 is adjacent to Vertex 3
5 #25th value=Vertex 4 is adjacent to Vertex 5
6 #26th value=Vertex 4 is adjacent to Vertex 6
1 #27th value=Vertex 5 is adjacent to Vertex 1
2 #28th value=Vertex 5 is adjacent to Vertex 2
3 #29th value=Vertex 5 is adjacent to Vertex 3
4 #30th value=Vertex 5 is adjacent to Vertex 4
6 #31th value=Vertex 5 is adjacent to Vertex 6
1 #32th value=Vertex 6 is adjacent to Vertex 1
2 #33th value=Vertex 6 is adjacent to Vertex 2
3 #34th value=Vertex 6 is adjacent to Vertex 3
4 #35th value=Vertex 6 is adjacent to Vertex 4
5 #36th value=Vertex 6 is adjacent to Vertex 5

```
➜  SourceCode git:(master) ✗ go run main.go
vertex 1 orginal degree 5 degree when deleted 0 color 1
vertex 2 orginal degree 5 degree when deleted 1 color 2
vertex 3 orginal degree 5 degree when deleted 2 color 3
vertex 4 orginal degree 5 degree when deleted 3 color 4
vertex 5 orginal degree 5 degree when deleted 4 color 5
vertex 6 orginal degree 5 degree when deleted 5 color 6
number of color used: 6
the maximum degree when deleted: 5
the size of terminal clique: 6
```

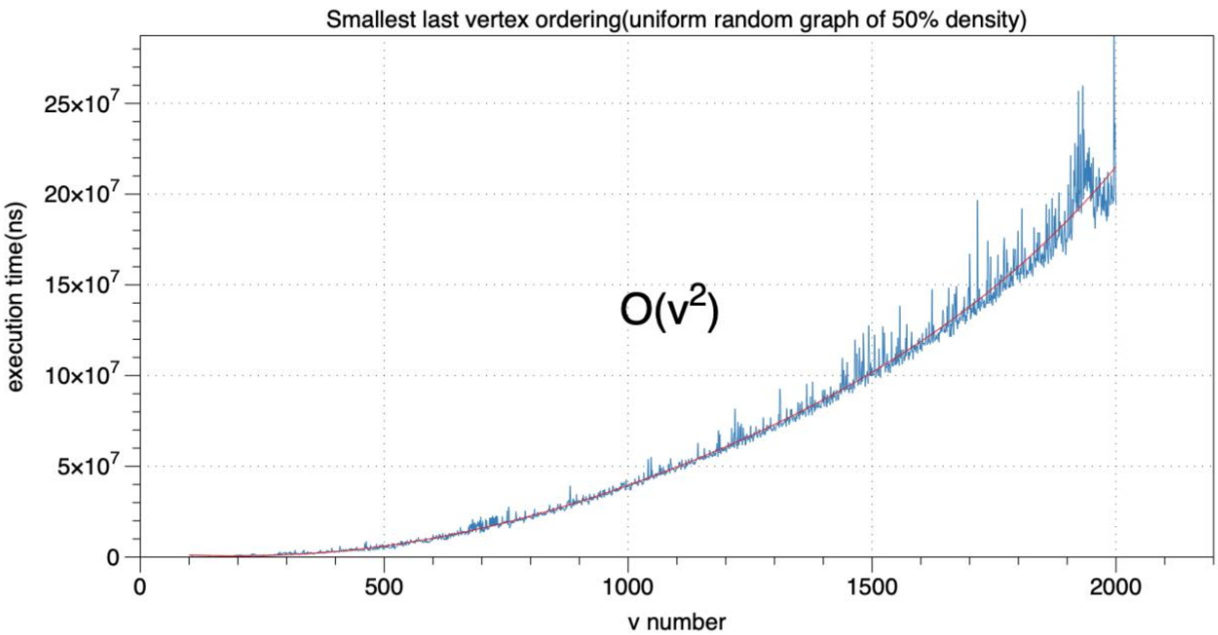


A discussion of how these bound the colors needed:
The size of terminal clique determines the lower bound of the colors needed. The maximum degree when deleted will help to find the size of the terminal clique.

2. Description of vertex ordering
For graph coloring part, I used three structures to help me finishing the algorithm.
Since there is not O(1) operation in the adjacent list to determine whether two points are adjacent, I use the adjacent matrix, which is a two dimensional array.
Same degree list is an array struct contained double linked list for each vertex of the same degree



1)Smallest last vertex ordering: In order to gain the vertex of minimum degree, I need to do the loop from degree of 0 to degree of (v-1) to pick the vertex contained smallest degree to be removed. Removing the vertex picked, and marking it deleted is O(1). Decreasing the degree of its adjacent vertices is not O(1), also adjusting the position in the same degree DLLs for its adjacent vertices. Decreasing the degree of its adjacent vertices is O(e) because I need to do the loop in every edge to determine which vertex should do degree decrement. Moving adjacent vertices in same degree DLLs is not O(e) since lacking the index to find the vertex in the same degree DLLs, thus I rebuild the DLLs, which cost O(v). **Consequently, I got this O(v^2) instead of O(v+e)**. **I try some ways, but failed, since I need to find adjacent vertices in the DLLs of different degree and delete them, which I couldn't find a O(e) approach to do this.** Moving them to the right position in the DLLs is O(1).



v	e	time(ns)
100	2,475	130,526
200	9,950	1,058,369
300	22,425	2,054,157
400	39,900	3,001,311
500	62,375	5,794,804
600	89,850	12,295,459
700	122,325	18,281,088
800	159,800	21,660,084
900	202,275	30,170,286
1,000	249,750	39,536,847
1,001	250,250	39,775,881
1,002	250,750	40,270,259
1,003	251,251	39,306,615
1,004	251,753	36,784,438
1,005	252,255	38,638,011
1,006	252,757	37,209,540
1,007	253,260	41,077,516
1,008	253,764	39,332,790
1,009	254,268	40,491,418
1,100	302,225	49,034,540
1,200	359,700	60,376,464
1,300	422,175	70,699,767
1,400	489,650	85,022,820

1,500	562,125	104,034,146
1,600	639,600	116,858,781
1,700	722,075	167,051,637
1,800	809,550	155,737,020
1,900	902,025	173,775,322
2,000	999,500	193,769,399

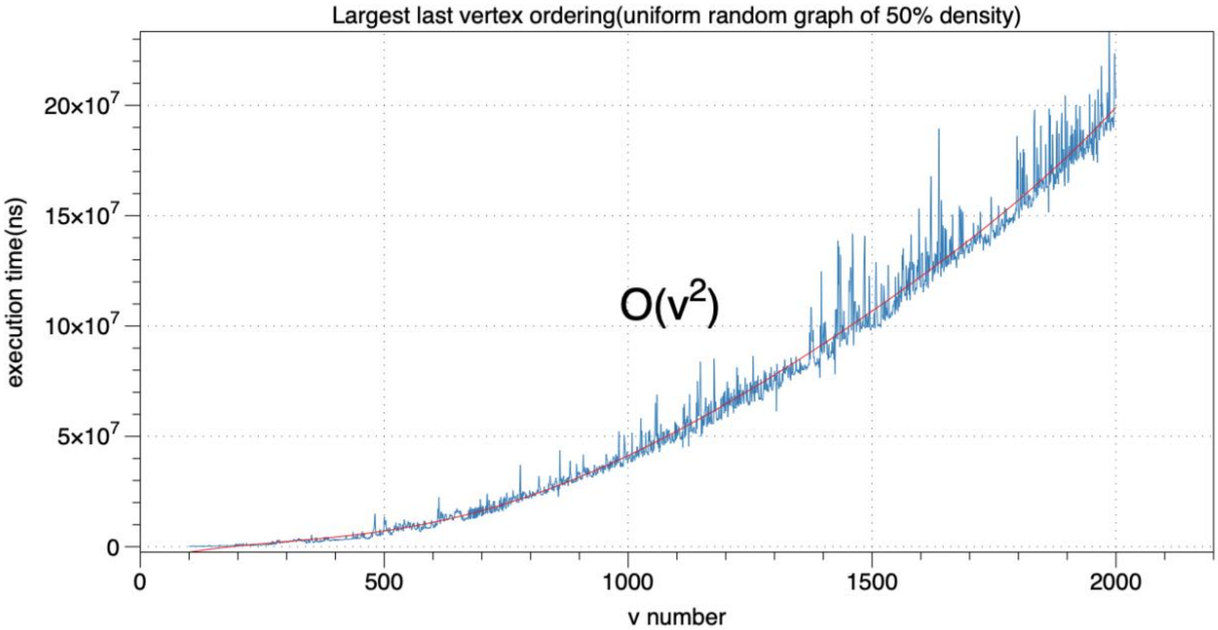
116858781(when v=1600)/21660084(when v=800) = 5.3

```
//Smallest Last Vertex Ordering

func coloringGraphWithTheSmallestLastOrdering(verticesList []*Vertex, adjacentMatrix [][]int, sameCurrentDegreeList

[]*Vertex) ([]*Vertex, [][]int, []int) {

	oLength := len(verticesList)

	orderList := make([]int, oLength)

	count := oLength - 1

	i := 0

	for count >= 0 {

		// fmt.Printf("count is %d the %d loop\n", count, i)

		if sameCurrentDegreeList[i] != nil {

			//pick a vetex to be removed

			pickVertexIndex := sameCurrentDegreeList[i].value - 1

			//push on stack

			orderList[count] = verticesList[pickVertexIndex].value

			//remove from dll

			if nil != sameCurrentDegreeList[i].next {

				sameCurrentDegreeList[i] = sameCurrentDegreeList[i].next
```

```
                sameCurrentDegreeList[i].last = nil

            } else {

                sameCurrentDegreeList[i] = nil

            }

            //record when degree deleted

            verticesList[pickVertexIndex].markDelete()

            //decrement degrees from its adjacent vertices

            currNode := verticesList[pickVertexIndex].head

            for currNode != nil {

                verticesList[currNode.value-1].degree--

                currNode = currNode.next

            }

            //move adjacent vertices in the dll

            count--

            sameCurrentDegreeList = initSDgreeList(verticesList)

            i = 0

        } else {

            i++

        }

    }
```

```
    return verticesList, adjacentMatrix, orderList

}
```

2)Largest last vertex ordering: the only difference between it and smallest last vertex ordering is that it pick the vertex of largest degree every time I remove the vertex from the graph. It's asymptotic time is O(v^2)



v	e	time(ns)
100	2,475	157,697
200	9,950	486,951
300	22,425	2,007,087
400	39,900	2,810,186
500	62,375	13,601,373
600	89,850	9,785,658
700	122,325	13,724,795
800	159,800	24,281,379
900	202,275	33,806,913
1,000	249,750	39,502,367
1,001	250,250	40,740,932
1,002	250,750	39,918,408
1,003	251,251	40,890,083
1,004	251,753	41,587,570
1,005	252,255	40,105,695
1,006	252,757	38,624,095

1,007	253,260	38,435,898
1,008	253,764	51,676,437
1,009	254,268	38,625,682
1,100	302,225	55,189,548
1,200	359,700	61,504,740
1,300	422,175	79,580,917
1,400	489,650	96,778,887
1,500	562,125	100,645,731
1,600	639,600	117,585,998
1,700	722,075	137,242,665
1,800	809,550	153,551,438
1,900	902,025	192,834,675
2,000	999,500	203,130,035

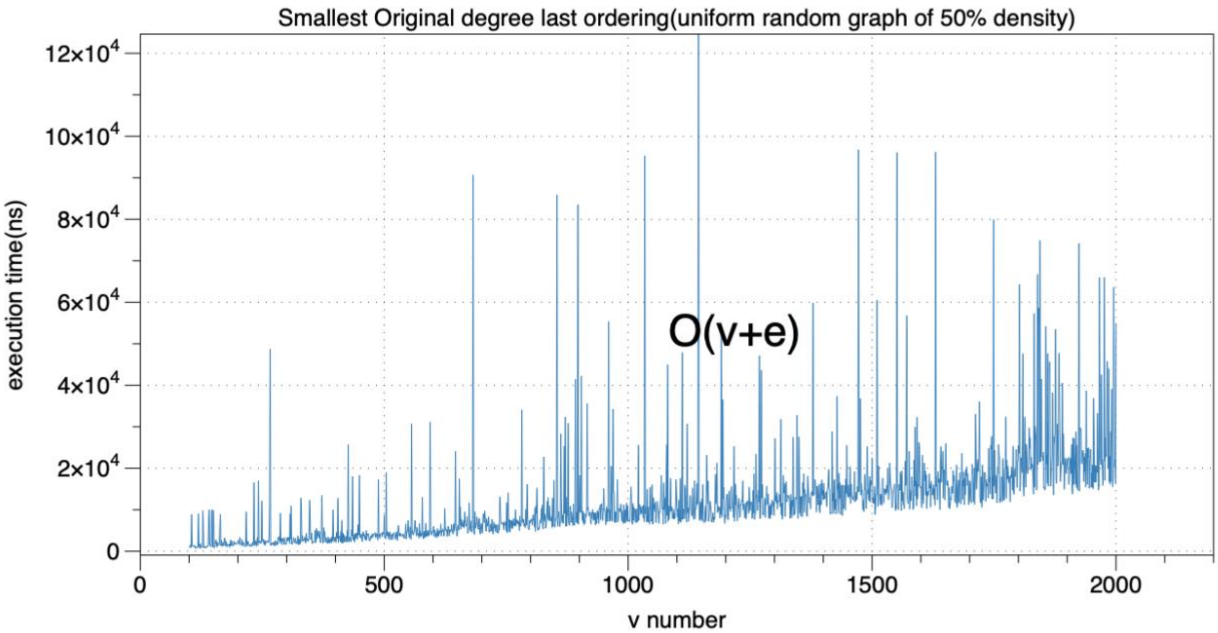
153551438(when v=900)/33806913(when v=1800) = 4.5

```
//Largest Last Vertex Ordering

func coloringGraphWithTheLargestLastOrdering(verticesList []*Vertex, adjacentMatrix [][]int, sameCurrentDegreeList

[]*Vertex) ([]*Vertex, [][]int, []int) {

	oLength := len(verticesList)

	orderList := make([]int, oLength)

	count := oLength - 1

	for i := oLength - 1; i >= 0; i-- {

		if sameCurrentDegreeList[i] != nil {

			//pick a vetex to be removed

			pickVertexIndex := sameCurrentDegreeList[i].value - 1

			//push on stack

			// reservedList[count] = verticesList[pickVertexIndex]

			orderList[count] = verticesList[pickVertexIndex].value
```

```
//remove from dll
if nil != sameCurrentDegreeList[i].next {
	sameCurrentDegreeList[i] = sameCurrentDegreeList[i].next
	sameCurrentDegreeList[i].last = nil
} else {
	sameCurrentDegreeList[i] = nil
}
//record when degree deleted
// reservedList[count-1].markDelete()
verticesList[pickVertexIndex].markDelete()
// reservedList[count-1].degreeWhenDelete = reservedList[count-1].degree
currNode := verticesList[pickVertexIndex].head
for currNode != nil {
	//decrease the degree of its adjacent vertices
	verticesList[currNode.value-1].degree--
	currNode = currNode.next
}
//move adjacent vertices in the dll
count--
if count < 0 {
```

```
                        break
                  }
                  sameCurrentDegreeList = initSDgreeList(verticesList)
                  i = oLength
            }
      }
      //graph coloring part
      //TODO need to optimize the loop to reduce the times it looped
      // startColoring(verticesList, adjacentMatrix, orderList)
      return verticesList, adjacentMatrix, orderList
}
```

3)Smallest Original degree last: Every time I pick the vertex contained smallest original degree from the graph and push it into the stack. It is O(v+e)



v	e	time(ns)

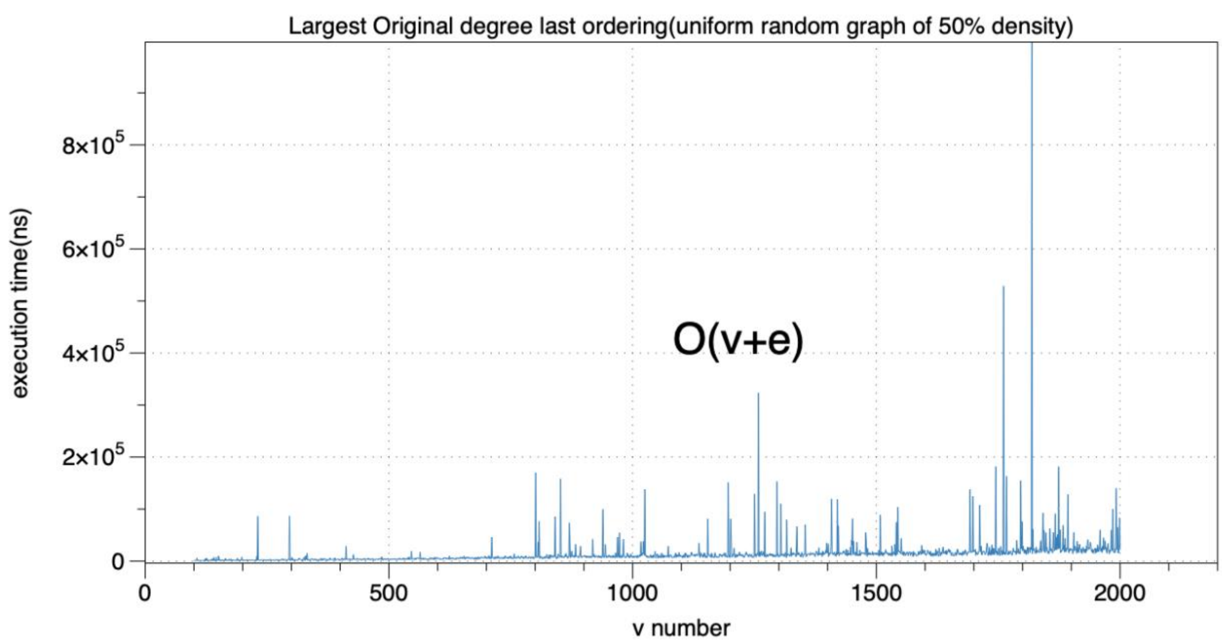
100	2,475	912
200	9,950	2,102
300	22,425	2,569
400	39,900	2,912
500	62,375	8,675
600	89,850	4,793
700	122,325	5,647
800	159,800	6,853
900	202,275	10,130
1,000	249,750	7,745
1,001	250,250	7,406
1,003	251,251	7,134
1,004	251,753	7,661
1,005	252,255	11,911
1,006	252,757	15,491
1,007	253,260	10,785
1,008	253,764	6,893
1,009	254,268	6,451
1,200	359,700	8,037
1,300	422,175	13,024
1,400	489,650	12,434
1,500	562,125	22,504
1,600	639,600	15,645
1,700	722,075	16,967
1,800	809,550	25,456
1,900	902,025	22,234
2,000	999,500	55,015

25456(when v=1800)/10130(when v=900)=2.5

```
func coloringGraphWithLargestOriginalDegreeLast(verticesList []*Vertex, adjacentMatrix [][]int,

sameCurrentDegreeList []*Vertex) ([]*Vertex, [][]int, []int) {

    oLength := len(verticesList)

    orderList := make([]int, oLength)

    count := oLength - 1

    for i := oLength - 1; i >= 0; i-- {
```

```
        currNode := sameCurrentDegreeList[i]

        for currNode != nil {

            orderList[count] = currNode.value

            count--

            currNode = currNode.next

        }

    }

    return verticesList, adjacentMatrix, orderList

    // startColoring(verticesList, adjacentMatrix, orderList)

}
```

4)Largest Original degree last : Every time I pick the vertex contained largest original degree from the graph and push it into the stack. It is O(v+e)



v	e	time(ns)
100	2,475	819
200	9,950	1,544

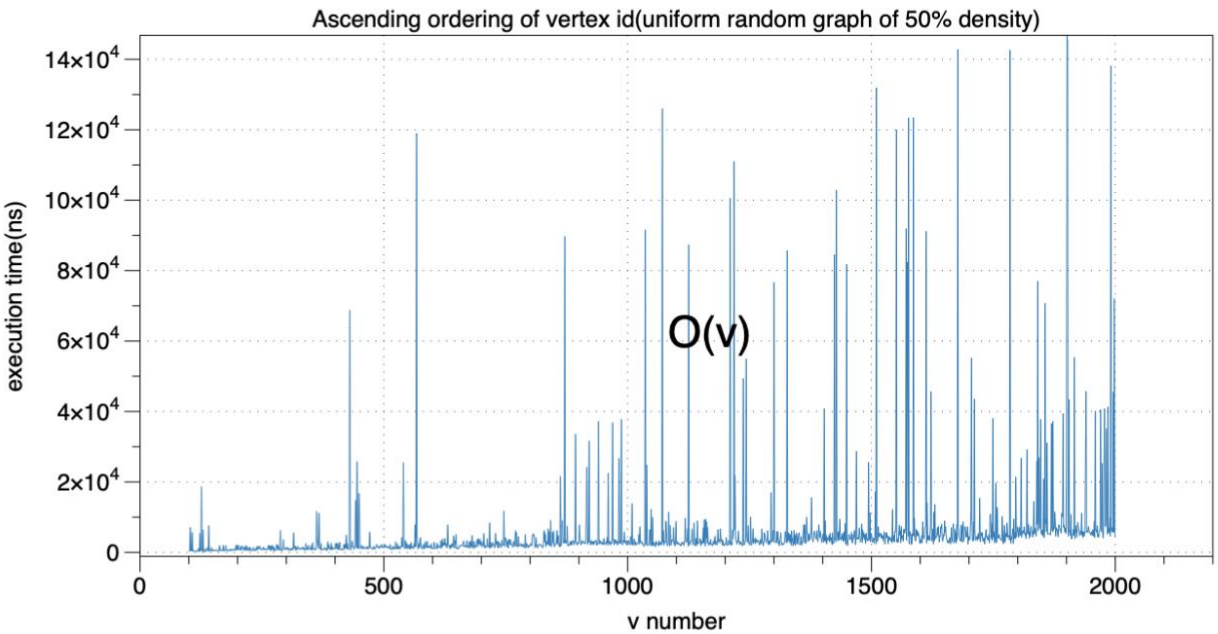
300	22,425	2,629
400	39,900	2,393
500	62,375	3,227
600	89,850	5,226
700	122,325	6,421
800	159,800	8,677
900	202,275	9,326
1,000	249,750	7,114
1,001	250,250	7,762
1,002	250,750	10,292
1,003	251,251	7,327
1,004	251,753	7,088
1,005	252,255	9,347
1,006	252,757	9,460
1,007	253,260	10,414
1,008	253,764	6,973
1,009	254,268	7,072
1,100	302,225	8,278
1,200	359,700	7,341
1,300	422,175	7,964
1,400	489,650	16,392
1,500	562,125	13,415
1,600	639,600	12,162
1,700	722,075	20,206
1,800	809,550	20,160
1,900	902,025	23,232
2,000	999,500	21,879

20160(time when v=1800)/9326(time when v=900) = 2.1

```
func coloringGraphWithLargestOriginalDegreeLast(verticesList []*Vertex, adjacentMatrix [][]int,
sameCurrentDegreeList []*Vertex) ([]*Vertex, [][]int, []int) {
	oLength := len(verticesList)
	orderList := make([]int, oLength)
	count := oLength - 1
	for i := oLength - 1; i >= 0; i-- {
```

```
        currNode := sameCurrentDegreeList[i]

        for currNode != nil {

            orderList[count] = currNode.value

            count--

            currNode = currNode.next

        }

    }

    return verticesList, adjacentMatrix, orderList

    // startColoring(verticesList, adjacentMatrix, orderList)

}
```

5)Ascending ordering of vertex id: just use its original order as comparison to other ordering. I know it should be O(1) since I already have the order, but I make it O(n) for better maintainable code.



v	e	time(ns)
100	2,475	463

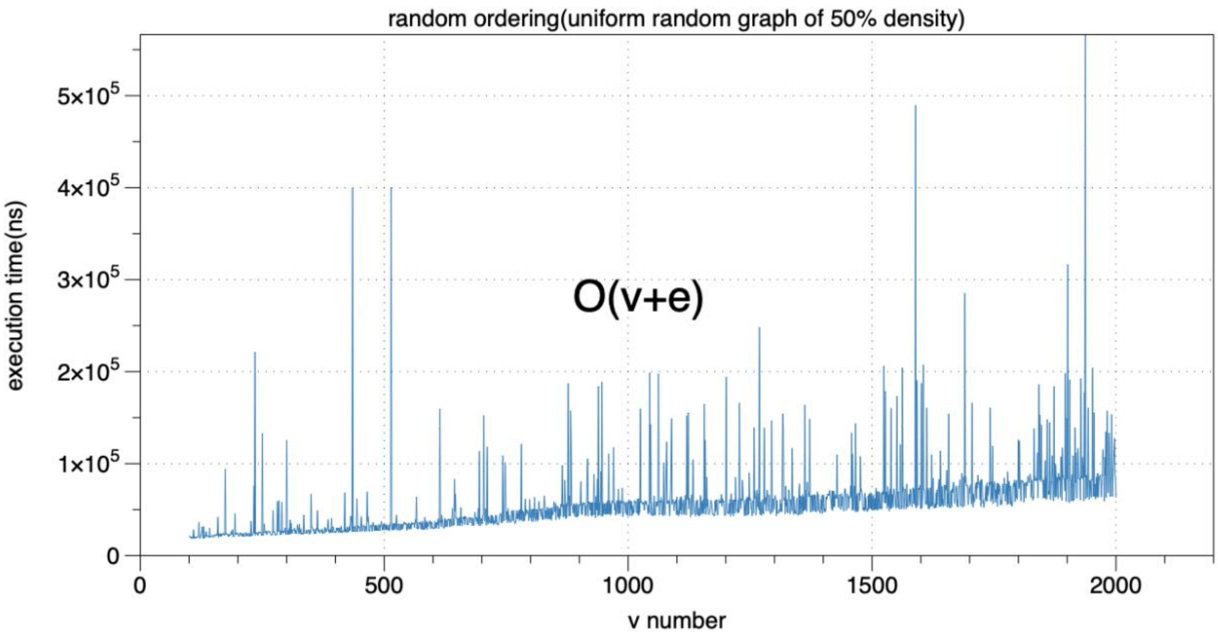
200	9,950	794
300	22,425	963
400	39,900	994
500	62,375	1,295
600	89,850	1,184
700	122,325	3,133
800	159,800	1,804
900	202,275	2,953
1,000	249,750	2,427
1,001	250,250	3,391
1,002	250,750	3,052
1,003	251,251	2,700
1,004	251,753	2,132
1,005	252,255	3,446
1,006	252,757	4,510
1,007	253,260	2,355
1,008	253,764	2,353
1,009	254,268	13,821
1,100	302,225	3,000
1,200	359,700	2,408
1,300	422,175	76,632
1,400	489,650	4,603
1,500	562,125	5,227
1,600	639,600	3,758
1,700	722,075	3,572
1,800	809,550	7,212
1,900	902,025	25,940
2,000	999,500	4,396

6)uniform random ordering : Given original order, just do the shuffle. For generating the sequence, it's similar to non-repeat random number I have done in the HW2.

```
func coloringGraphWithRandomOrdering(verticesList []*Vertex, adjacentMatrix [][]int) ([]*Vertex, [][]int, []int) {

    oLength := len(verticesList)

    orderList := randomNumberSequenceGenerator(oLength, oLength, Uniform)

    return verticesList, adjacentMatrix, orderList
```

```
}

func randomNumberSequenceGenerator(needNumber int, numberRange int, distributedType int) []int {

	if needNumber < 1 || numberRange < 1 {

		panic("needNumber or numberRange should be greater than 1")

	}

	if distributedType > 3 || distributedType < 1 {

		panic("distributedType should be between 1 and 3")

	}

	rand.Seed(time.Now().UnixNano())

	list := make([]int, needNumber)

	rand.Seed(time.Now().UnixNano())

	var randList = make([]int, numberRange)

	for i := 0; i < numberRange; i++ {

		randList[i] = i + 1

	}

	var randIndex int

	for i := 0; i < needNumber; i++ {

		if numberRange-i == 0 {

			randIndex = 0

		} else if 1 == distributedType {
```

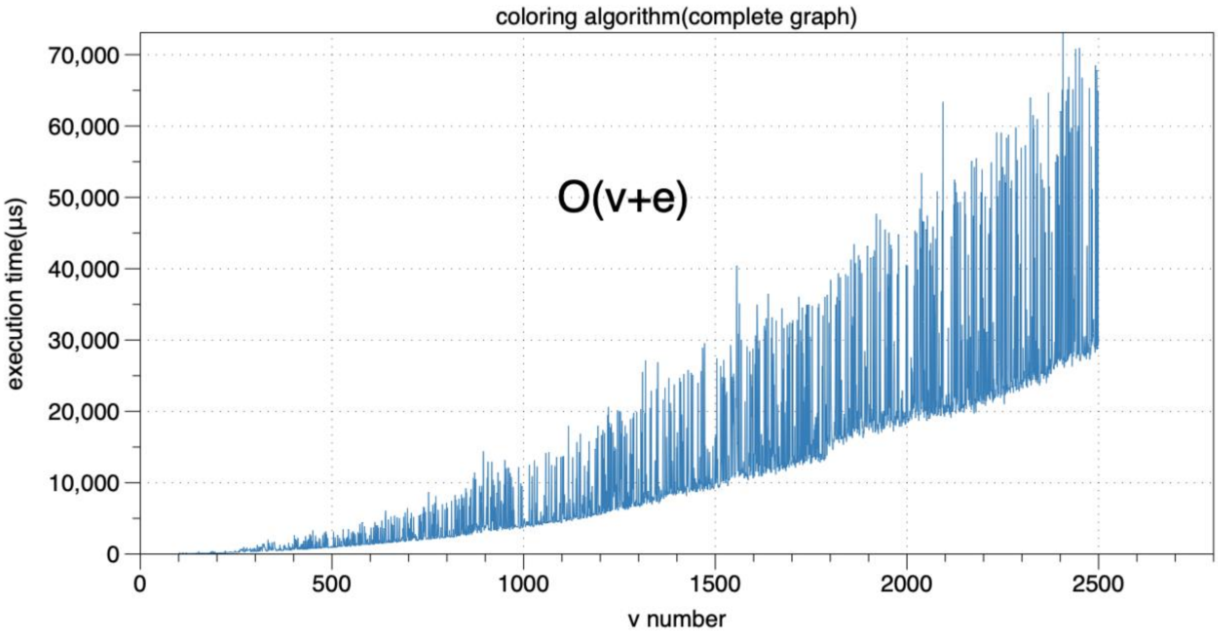
```
                randIndex = uniformRandomNumber(numberRange-i) - 1

            } else if 2 == distributedType {

                randIndex = skewedRandomNumber(numberRange-i) - 1

            } else {

                randIndex = amorRandomNumber(numberRange-i) - 1

            }

            list[i] = randList[randIndex]

            randList[randIndex], randList[numberRange-i-1] = randList[numberRange-i-1], randList[randIndex]

        }

        return list

}
```



v	e	time(ns)
100	2,475	21,183
200	9,950	23,601

300	22,425	125,226
400	39,900	25,808
500	62,375	35,506
600	89,850	29,886
700	122,325	37,567
800	159,800	40,842
900	202,275	55,531
1,000	249,750	43,506
1,001	250,250	57,343
1,002	250,750	60,661
1,003	251,251	57,421
1,004	251,753	43,336
1,005	252,255	59,482
1,006	252,757	51,103
1,007	253,260	58,407
1,008	253,764	43,411
1,009	254,268	54,068
1,100	302,225	63,714
1,200	359,700	45,789
1,300	422,175	59,379
1,400	489,650	66,040
1,500	562,125	50,633
1,600	639,600	50,983
1,700	722,075	68,188
1,800	809,550	125,981
1,900	902,025	83,835
2,000	999,500	62,900

3.Description of coloring algorithm: given the sequence of ordered vertices, assign color by its order. For vertex picked to be coloring, I used an auxiliary array to remember the color used by its adjacent vertices. Marking the colors of adjacent vertices is O(e), and ensuring the color of the vertex picked unique to its adjacent vertices is O(e). Thus its asymptotic time is O(v+2*e) or O(v+e)



v	e	time(us)
100	4,950	88
200	19,900	139
300	44,850	687
400	79,800	738
500	124,750	930
600	179,700	1,350
700	244,650	1,894
800	319,600	2,372
900	404,550	3,985
1,000	499,500	4,035
1,100	604,450	4,500
1,200	719,400	5,506
1,300	844,350	11,924
1,400	979,300	8,429
1,500	1,124,250	16,339
1,600	1,279,200	14,058
1,700	1,444,150	32,322
1,800	1,619,100	17,354
1,900	1,804,050	18,189
2,000	1,999,000	19,135
2,100	2,203,950	20,842
2,200	2,418,900	21,059
2,300	2,643,850	23,840

2,400	2,878,800	27,042
2,500	3,123,750	30,603

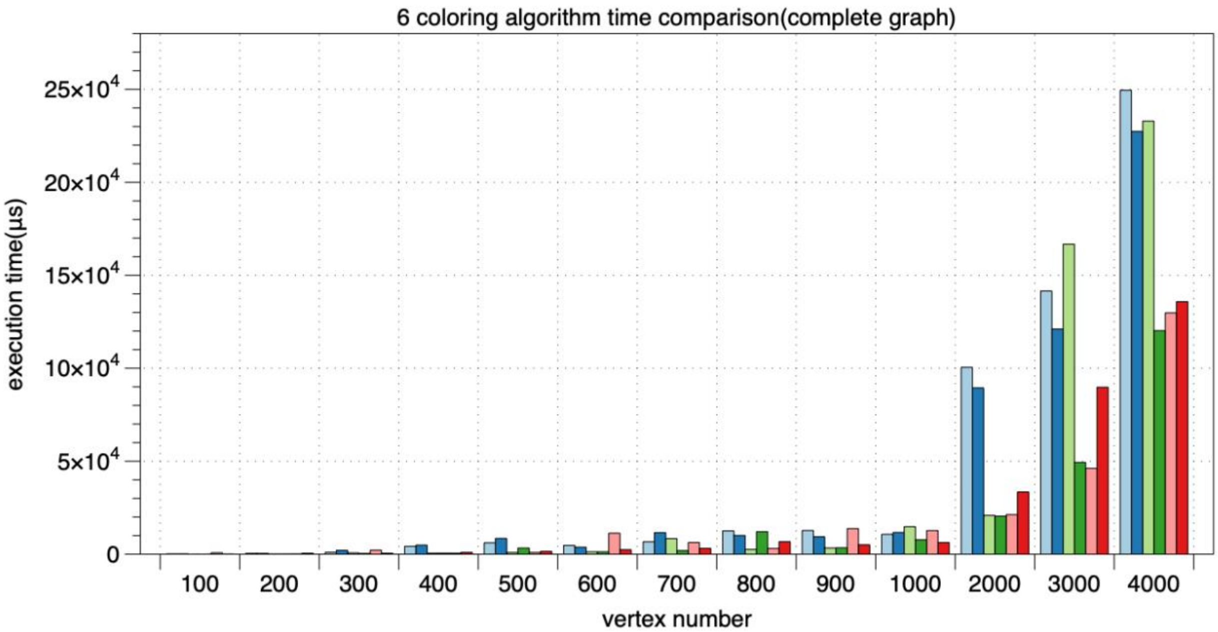
For example, comparing (v=2500,e=3123750) with (v=1800,e=1619100)
3123750/161900=1.93
30603/17354=1.7, which is pretty close to 1.9.

```
func startColoring(verticesList []*Vertex, adjacentMatrix [][]int, orderList []int) {

    oLength := len(verticesList)

    for i := 0; i < oLength; i++ {

        currNode := verticesList[orderList[i]-1].head

        colorSlot := make([]int, oLength) //to guarantee every vertex has different color

        for currNode != nil {

            if 0 != verticesList[currNode.value-1].color {

                colorSlot[verticesList[currNode.value-1].color-1] = 1

            }

            currNode = currNode.next

        }

        for k := 0; k < oLength; k++ {

            if colorSlot[k] != 1 {

                verticesList[orderList[i]-1].color = k + 1

                break

            }

        }

    }
```

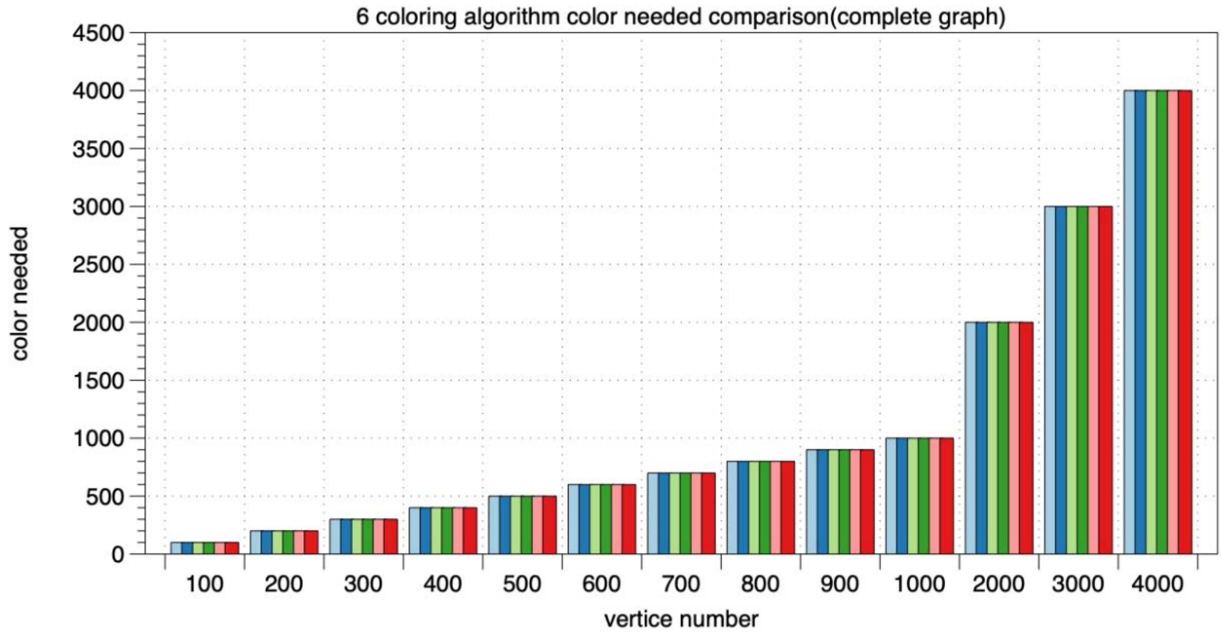
```
}
```

4.Vertex Ordering Capabilities

1)For coloring the complete graph, all 6 algorithms have the same color needed. However, the largest original degree last and ascending vertex id have better performance than others.



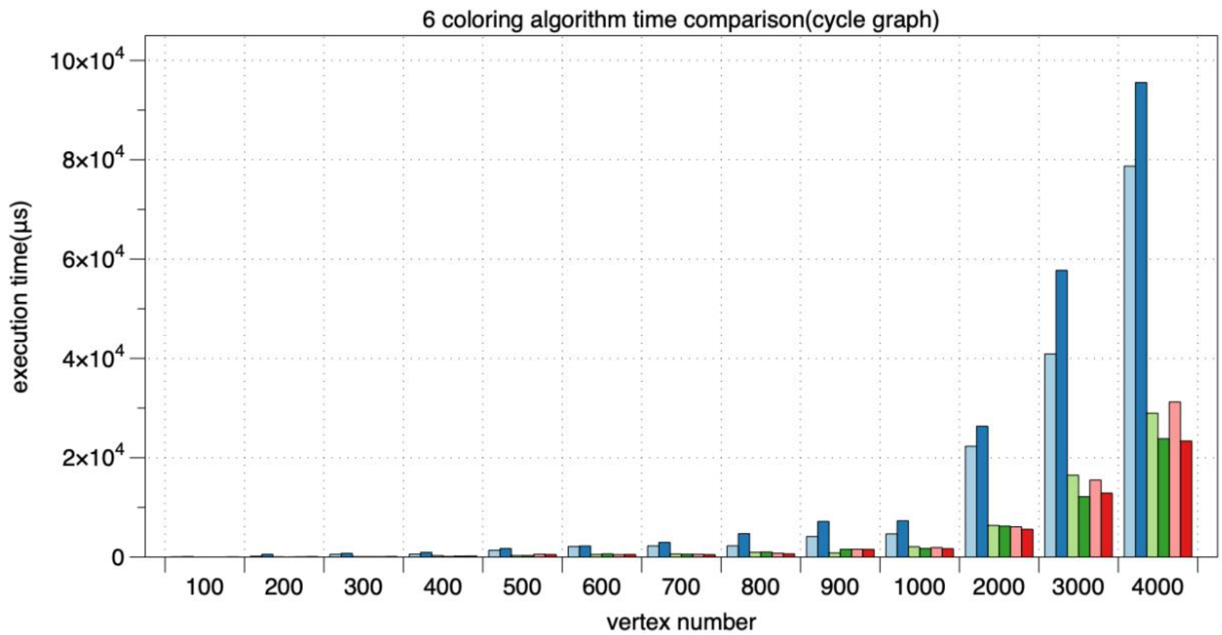
smallest last vertex ordering largest last vertex ordering smallest original degree last
largest original degree last ascending vertex ID ordering random ordering



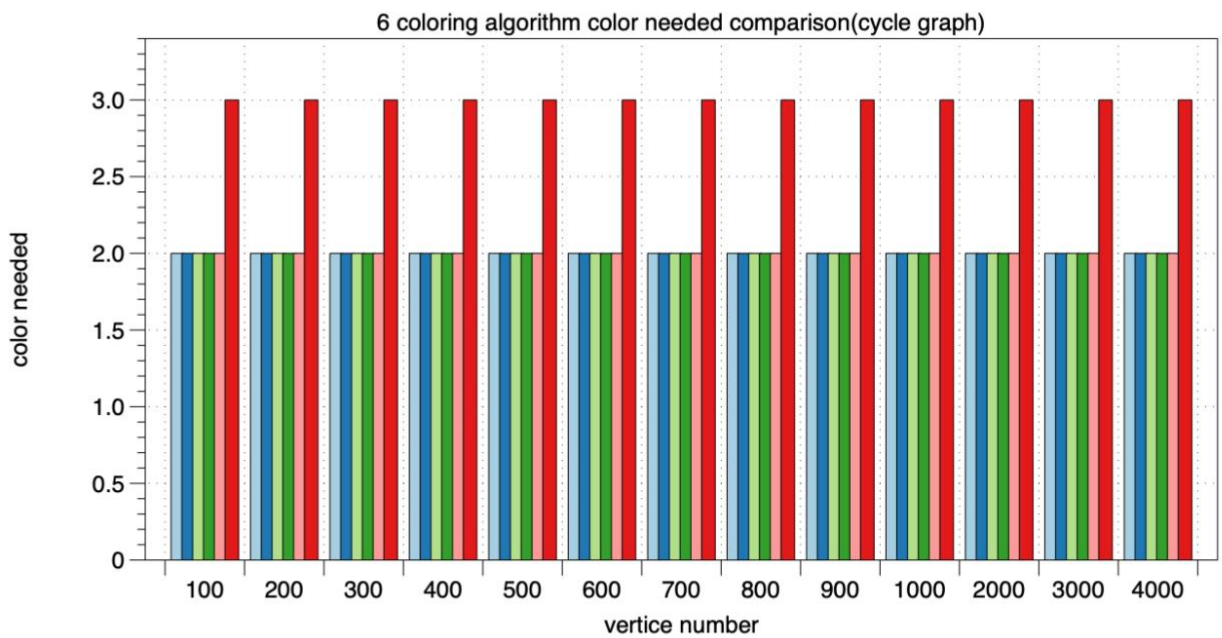
smallest last vertex ordering largest last vertex ordering smallest original degree last
largest original degree last ascending vertex ID ordering random ordering

v	e	average degree	max degree	color needed of smallest last vertex ordering	execution time of smallest last vertex ordering(μs)	color needed of largest last vertex ordering	execution time of largest last vertex ordering(μs)	color needed of smallest original degree last	execution time of smallest original degree last(μs)	color needed of largest original degree last	execution time of largest original degree last(μs)	color needed of ascending vertex ID ordering	execution time of ascending vertex ID ordering(μs)	color needed of random ordering	execution time of random ordering(μs)
100	4,950	99	99	100	170	100	181	100	76	100	82	100	785	100	106
200	19,900	199	199	200	425	200	398	200	137	200	144	200	172	200	504
300	44,850	299	299	300	1,064	300	2,070	300	724	300	374	300	2,185	300	532
400	79,800	399	399	400	4,229	400	4,899	400	580	400	584	400	574	400	960
500	124,750	499	499	500	6,121	500	8,492	500	926	500	3,319	500	921	500	1,512
600	179,700	599	599	600	4,725	600	3,781	600	1,291	600	1,282	600	11,309	600	2,417
700	244,650	699	699	700	6,679	700	11,611	700	8,381	700	1,922	700	6,292	700	3,153
800	319,600	799	799	800	12,520	800	10,092	800	2,569	800	12,121	800	3,064	800	6,710
900	404,550	899	899	900	12,754	900	9,375	900	3,343	900	3,450	900	13,769	900	5,054
1,000	499,500	999	999	1,000	10,704	1,000	11,703	1,000	14,811	1,000	7,740	1,000	12,707	1,000	6,259
2,000	1,999,000	1,999	1,999	2,000	100,510	2,000	89,455	2,000	20,861	2,000	20,484	2,000	21,280	2,000	33,477
3,000	4,498,500	2,999	2,999	3,000	141,558	3,000	121,081	3,000	166,713	3,000	49,369	3,000	46,199	3,000	89,811
4,000	7,998,000	3,999	3,999	4,000	249,521	4,000	227,373	4,000	232,925	4,000	120,247	4,000	129,802	4,000	135,831
5,000	12,497,500	4,999	4,999	5,000	749,331	5,000	1,040,856	5,000	162,476	5,000	149,265	5,000	162,500	5,000	178,526
8,000	31,996,000	7,999	7,999	8,000	25,959,829	8,000	57,890,862	8,000	112,054,938	8,000	121,888,016	8,000	105,081,252	8,000	110,117,381
10,000	49,995,000	9,999	9,999	10,000	155,419,147	10,000	330,375,799	10,000	156,527,869	10,000	22,654,092	10,000	48,251,530	10,000	49,501,152

2)For coloring the cycle graph, random ordering need one more color than other algorithms. For comparison of performance, largest original degree last and random algorithm would be better.

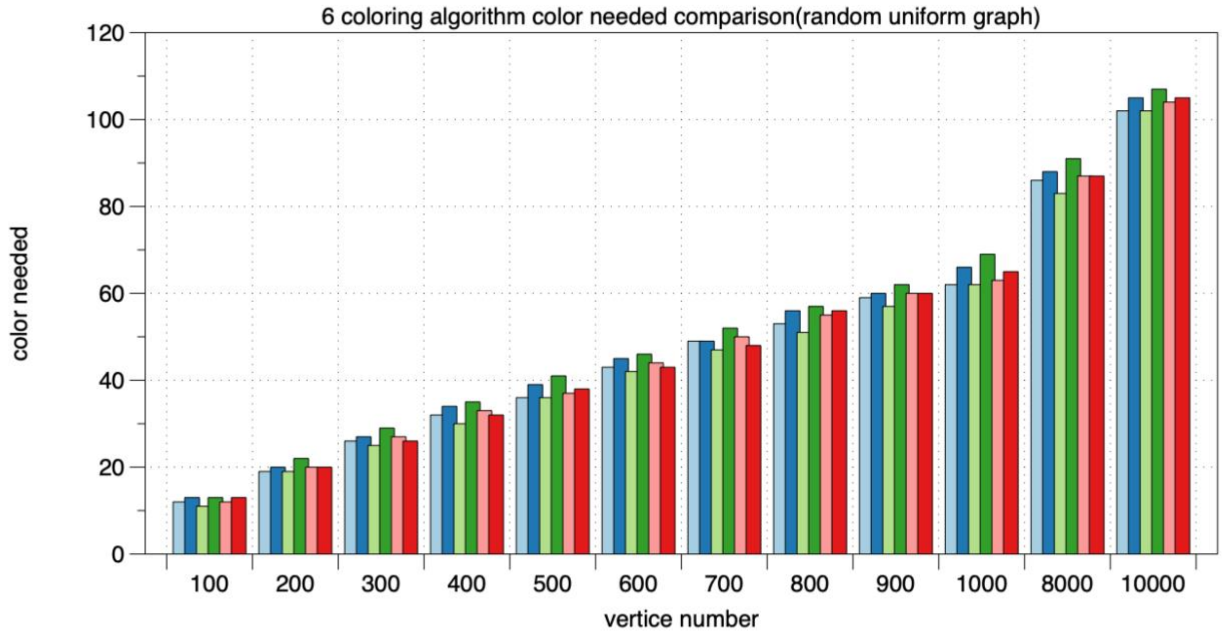


smallest last vertex ordering largest last vertex ordering smallest original degree last
largest original degree last ascending vertex ID ordering random ordering

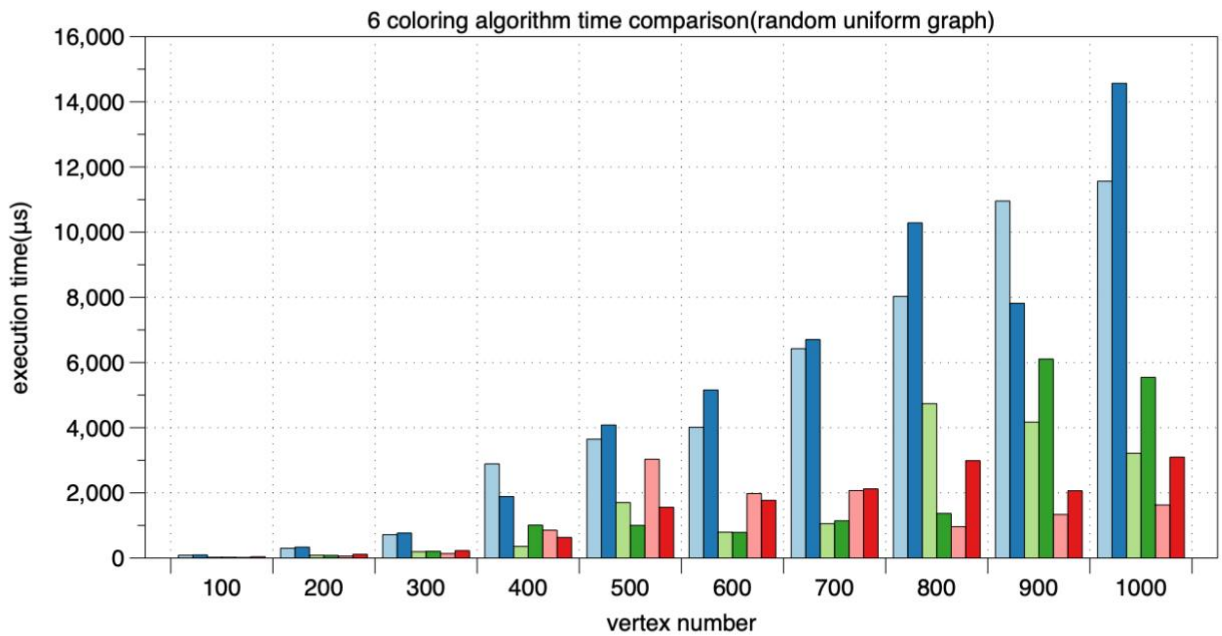


smallest last vertex ordering largest last vertex ordering smallest original degree last
largest original degree last ascending vertex ID ordering random ordering

3)For coloring random graph of uniform distribution, the smallest original degree last would be better than other algorithms. However, ascending vertex ID and random would be better in performance.



smallest last vertex ordering largest last vertex ordering smallest original degree last
largest original degree last ascending vertex ID ordering random ordering

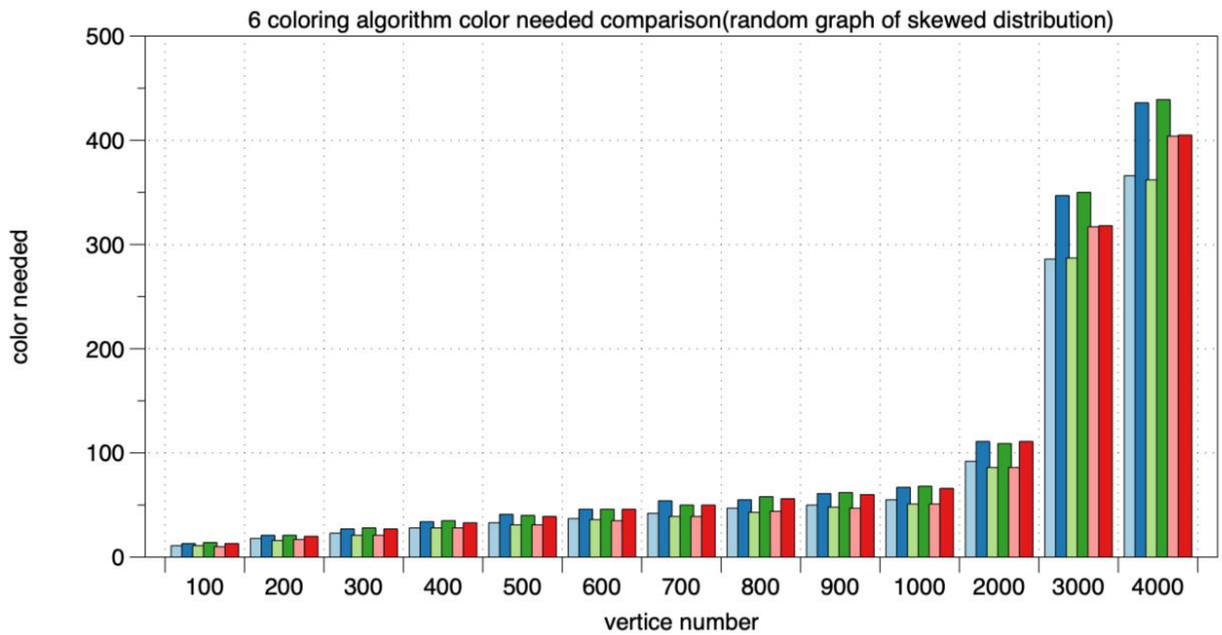


smallest last vertex ordering largest last vertex ordering smallest original degree last
largest original degree last ascending vertex ID ordering random ordering

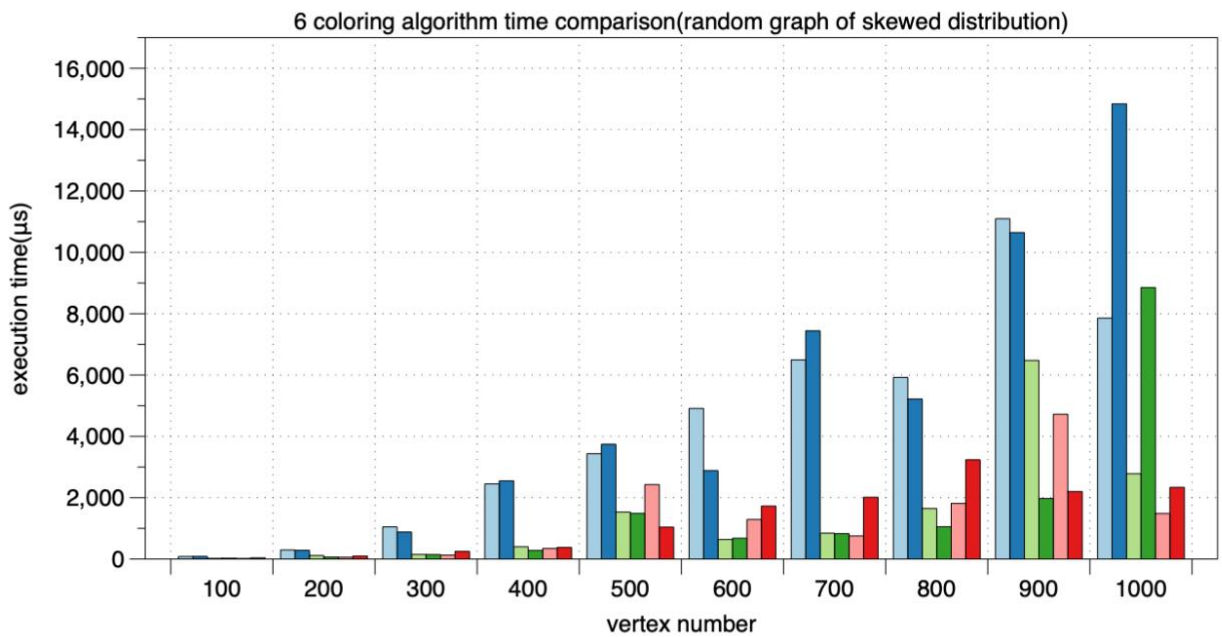
v	e	average degree	max degree	color needed of smallest last vertex ordering	execution time of smallest last vertex ordering(µs)	color needed of largest last vertex ordering	execution time of largest last vertex ordering(µs)	color needed of smallest original degree last	execution time of smallest original degree last(µs)	color needed of largest original degree last	execution time of largest original degree last(µs)	color needed of ascending vertex ID ordering	execution time of ascending vertex ID ordering(µs)	color needed of random ordering	execution time of random ordering(µs)
100	1,237	24	34	12	85	13	95	11	22	13	22	12	17	13	40
200	4,975	49	69	19	300	20	335	19	84	22	79	20	59	20	113
300	11,212	74	98	26	717	27	768	25	195	29	205	27	137	26	227
400	19,950	99	128	32	2,891	34	1,886	30	357	35	1,009	33	855	32	634
500	31,187	124	153	36	3,647	39	4,083	36	1,703	41	1,002	37	3,033	38	1,557
600	44,925	149	188	43	4,012	45	5,157	42	794	46	789	44	1,980	43	1,771
700	61,162	174	204	49	6,426	49	6,704	47	1,055	52	1,144	50	2,070	48	2,121
800	79,900	199	242	53	8,029	56	10,287	51	4,739	57	1,368	55	963	56	2,988
900	101,137	224	259	59	10,956	60	7,816	57	4,172	62	6,106	60	1,336	60	2,063
1,000	124,875	249	288	62	11,563	66	14,568	62	3,216	69	5,546	63	1,629	65	3,097

8,000	1,599,800	399	474	86	90,212,908	88	45,236,049	83	31,386,333	91	17,348,460	87	4,623,103	87	4,467,287
10,000	2,499,750	499	577	102	15,635,739	105	2,190,457	102	173,263	107	180,586	104	171,220	105	171,285

4)For coloring the random graph of skewed distribution, it seems that smallest last vertex ordering and smallest original vertex degree would need less color than other algorithms. In performance comparison, smallest last vertex ordering and largest last vertex ordering still fall behind other algorithms.



smallest last vertex ordering largest last vertex ordering smallest original degree last
largest original degree last ascending vertex ID ordering random ordering

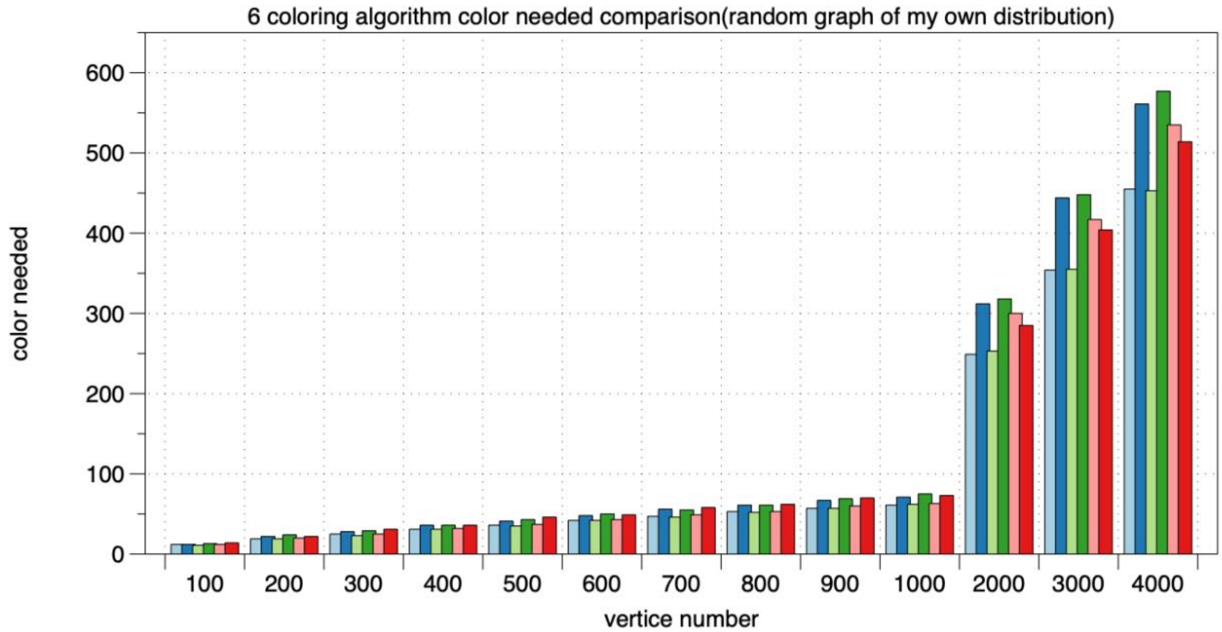


smallest last vertex ordering largest last vertex ordering smallest original degree last
largest original degree last ascending vertex ID ordering random ordering

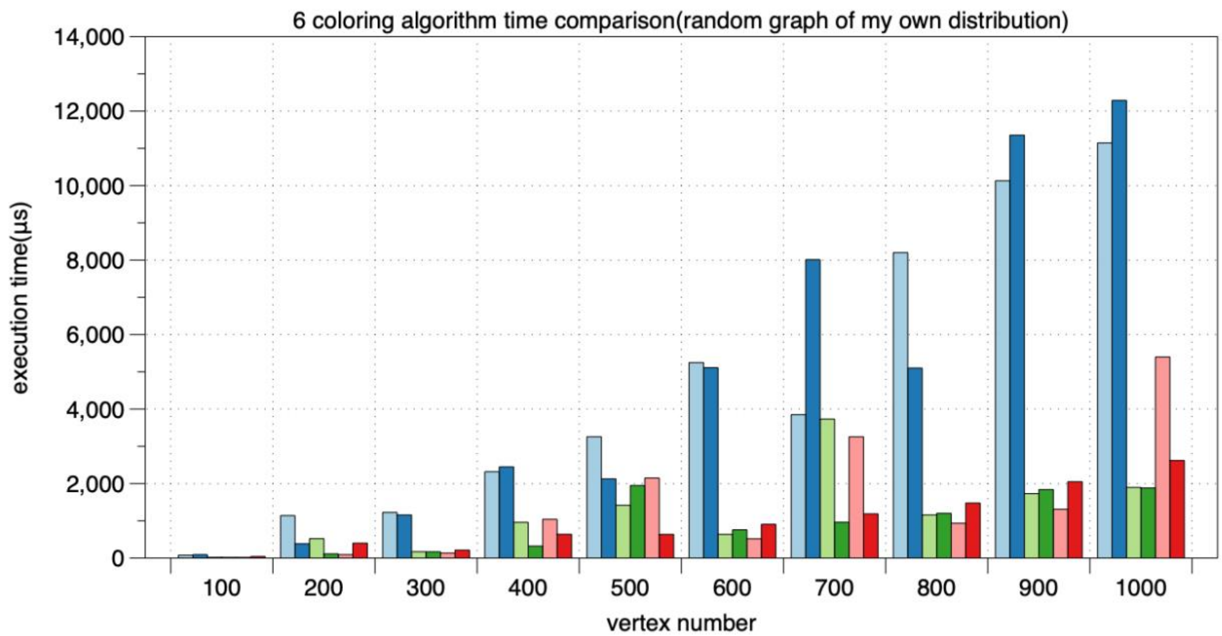
v	e	average degree	max degree	color needed of smallest last vertex ordering	execution time of smallest last vertex ordering(µs)	color needed of largest last vertex ordering	execution time of largest last vertex ordering(µs)	color needed of smallest original degree last	execution time of smallest original degree last(µs)	color needed of largest original degree last	execution time of largest original degree last(µs)	color needed of ascending vertex ID ordering	execution time of ascending vertex ID ordering(µs)	color needed of random ordering	execution time of random ordering(µs)
100	1237	24	47	11	82	13	85	11	20	14	26	10	17	13	40
200	4975	49	94	18	296	21	285	16	114	21	66	17	58	20	97
300	11212	74	157	23	1048	27	880	21	149	28	145	21	127	27	248

400	19950	99	196	28	2450	34	2549	28	400	35	279	28	344	33	377
500	31187	124	238	33	3434	41	3743	31	1532	40	1487	31	2428	39	1039
600	44925	149	285	37	4911	46	2883	36	638	46	680	35	1290	46	1728
700	61162	174	344	42	6492	54	7441	39	844	50	830	39	752	50	2015
800	79900	199	381	47	5922	55	5217	43	1647	58	1054	44	1813	56	3236
900	101137	224	438	50	11097	61	10645	48	6475	62	1975	47	4721	60	2202
1000	124875	249	479	55	7855	67	14840	51	2783	68	8853	51	1484	66	2338
2000	499750	499	951	92	46093	111	46463	86	8147	109	8524	86	9042	111	10843
3000	2249250	1499	2281	286	150683	347	184480	287	73611	350	53240	317	32035	318	39245
4000	3999000	1999	3034	366	264602	436	404089	362	166845	439	150792	404	147102	405	115130

5) For coloring random graph of my own distribution, smallest last vertex ordering and smallest original degree last need less color than other algorithms. However, smallest original degree last would be better in performance comparison.



smallest last vertex ordering largest last vertex ordering smallest original degree last
largest original degree last ascending vertex ID ordering random ordering



smallest last vertex ordering largest last vertex ordering smallest original degree last
largest original degree last ascending vertex ID ordering random ordering

v	e	average degree	max degree	color needed of smallest last vertex ordering	execution time of smallest last vertex ordering(μs)	color needed of largest last vertex ordering	execution time of largest last vertex ordering(μs)	color needed of smallest original degree last	execution time of smallest original degree last(μs)	color needed of largest original degree last	execution time of largest original degree last(μs)	color needed of ascending vertex ID ordering	execution time of ascending vertex ID ordering(μs)	color needed of random ordering	execution time of random ordering(μs)
100	1237	24	87	12	76	12	92	11	19	13	17	12	17	14	41
200	4975	49	178	19	1141	22	384	19	521	24	114	20	92	22	401
300	11212	74	262	25	1225	28	1158	23	171	29	169	25	135	31	214
400	19950	99	348	31	2320	36	2449	31	959	36	322	32	1040	36	639
500	31187	124	429	36	3257	41	2127	35	1418	43	1949	37	2148	46	637
600	44925	149	519	42	5248	48	5112	42	637	50	758	43	517	49	907
700	61162	174	599	47	3849	56	8012	46	3732	55	961	49	3256	58	1188
800	79900	199	698	53	8200	61	5101	52	1158	61	1200	53	934	62	1478
900	101137	224	774	57	10130	67	11355	57	1730	69	1839	60	1310	70	2051
1000	124875	249	852	61	11146	71	12287	62	1897	75	1881	63	5398	73	2619
2000	999500	999	1897	249	60233	312	67537	253	14144	318	17882	300	10819	285	33620

3000	2249250	1499	2851	354	134696	444	159579	355	35335	448	79064	417	81706	404	67631
4000	3999000	1999	3788	455	165494	561	278161	453	156338	577	78035	535	53616	514	71010

5.An interesting problem:

This is a random graph I found on the way working with the project. By the smallest last vertex ordering, its order colored is 2 4 6 3 1 5. When I was coloring vertex 1 here, I had two choice : 2 or 3. If I chose 2, then 5 had to be colored with 4, thus the total color used is 4. But if I chose 3, then 5 could be 2, thus the total color used is 3. Maybe it's the tradeoff of algorithm. If I want the minimum color used, I should try branching different situations and calculate the color used to gain a minimum color, but the efficiency of algorithm is worse than the algorithm now I used