CS7350 Final Report

Name BanBo
Id: 48358717

Computing environment in detail
Computer system: macOS Big Sur;
Running Software: Terminal

Gcc p.c
./a.out
(windows & linux could have some mistakes because C compiler, line 31 in program is the main error. Could Windows is C99 or older version ? MacOS is fine run and no errors, if you need, I can share my screen to show the program)
Program Language: C

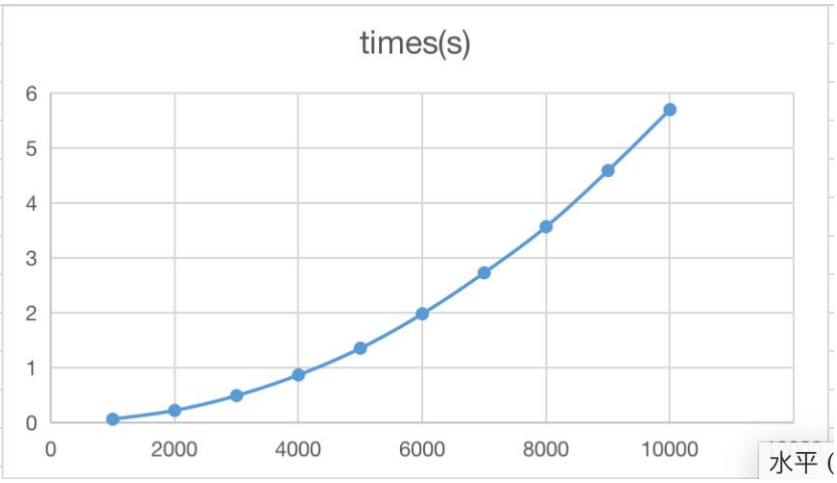
1. A description and analysis of your algorithms for generating the conflict graphs

<Complete Graph>Θ(n^2)

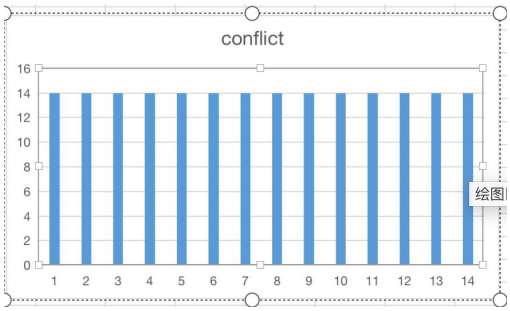
Method: In my code, I define 2 functions: 1. AddEdge 2.Complete Graph.
The first one contain all pointer operation, the second one is contain 2 “for” loop to contain each vertex to each other left behind of it.

A function representing time = N*N*C.
f(n) is Θ(n^2)

vex	times(s)
1000	0.066197
2000	0.226326
3000	0.496276
4000	0.872087
5000	1.358128
6000	1.984556
7000	2.731374
8000	3.57073
9000	4.592998
10000	5.701563



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



<Circle Graph>Θ(n)

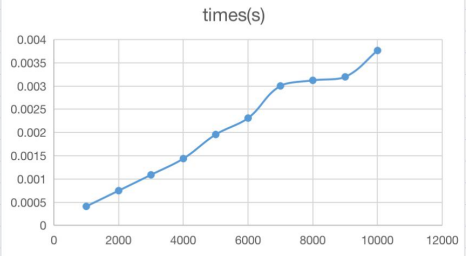
Method: In my code, use earlier function “addedge”. For each vertex add with its rear vertex and final vertex add it with the beginning vertex.

A function representing time = C*N.

f(n) is Θ(n)

Because all node haven’t connected with other vertex and edges so, “addedge” functions check always use fixed time, so it means it equals to C.

n	times(s)
1000	0.000411
2000	0.000749
3000	0.00109
4000	0.001439
5000	0.001958
6000	0.002311
7000	0.003003
8000	0.003123
9000	0.003198
10000	0.003765



<Random Uniform Graph>Θ(n^2)

Method: In this function, I randomly generate 2 numbers, so each vertex is equally likely to be chosen for a conflict And use “isedge” function to traverse the Vertex pointer information to detect is there already have edge? Finally adding a new edge to it. During the beginning of the running, there is have a little chance to fail or crash. In the middle of time, the chance of conflict is increasing. Finally, have the highest chance to fail, so it have to re-random a new edge between different vertex.

A function representing time = N*N*C+C*N.

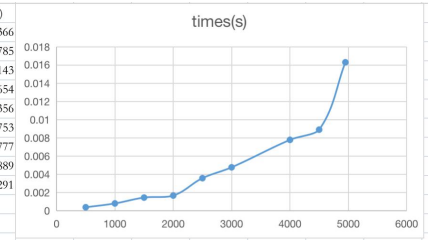
Also the more input’s edges number relative to the same vertex, the higher chance to fail and re-random. If the number of edges is equal relatively to the number of edges of complete graph with the same vertex, there will be a worst case. The function represent time could be Θ(n^2). Because it have to random n times to get the right answer because it need both right “from” pointer and “to” pointer. At the same time, it has to detect whether there is already an edge existing that there will be pay n times. Finally add edge that change the pointer and other operation will pay C times.

So the function represent time could be Θ(n^2) and Histograms should be a horizontal, because each vertex have same chance to be chosen.

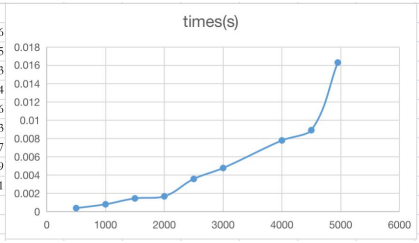
Finally I test 2 times for different data set:

1. # of vertex = 100. And every time its edges add 500 until it become a complete graph that the number is 4950;
2. # of vertex = 200.And every time its edges add 2000 until it become a complete graph that the number is 19900;

edges	times(s)
500	0.000366
1000	0.000785
1500	0.00143
2000	0.001654
2500	0.00356
3000	0.004753
4000	0.00777
4500	0.00889
4950	0.016291



edges	times(s)
500	0.000366
1000	0.000785
1500	0.00143
2000	0.001654
2500	0.00356
3000	0.004753
4000	0.00777
4500	0.00889
4950	0.016291



So we can see these two pictures that indicate at beginning it could have the best case that could lower than Θ(n), and with edges number increasing, the function represent time could be Θ(n^2).

<Random Skewed Graph>Θ(n^2)

Method: I use Quadratic system of one variable method to simulate the real possibility for the chance of each vertex that could be chosen. The lower numbered vertices are linearly more likely than higher numbered vertices. For example, I assume the sum of possibilities of each vertex is 100%. If we have 5 vertex, y = each ones possibilities, x means vertex id, a means linear line's slope and b is a weight.

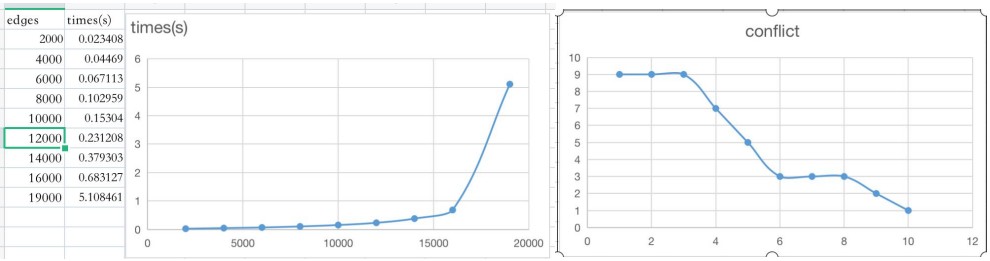
The formula is $y_1 = a*x_1 +b; y_2 = a*x_2 +b; y_3 = a*x_3 +b; y_4 = a*x_4 +b; y_5 = a*x_5 +b;$

And $y_1+y_2 +y_3 +y_4+y_5=1$; And when x = 0; we can get the the value of b.

By using this method, I defined a function named "skewvertex". It will read value of total number to create a linear line. And each vertex's possibilities can use above formula to calculate. And then giving a million numbers to assign according to the probability of each vertex. Finally generating a random number between 0-1000000,to determine which vertex this round has been chose. It also seem as Random Uniform Method. For each edges we will generate a pair of vertex and use "isedges" function to check whether there is existing an edge.

A function representing time = N*C*N+C*N.

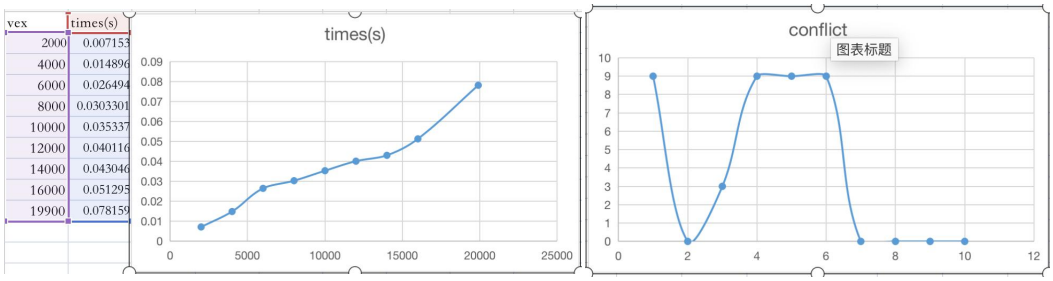
So in worst case it could be Θ(n^2)(ideally condition), in best case lower than Θ(n).



<Random normal>Θ(n)

Method: This method is user designed. The way is that choosing any vertex is first pointer, adding edges with other vertex until it cannot add any edges, it break the while loop. And then choosing another vertex until it become a complete graph. For each vertex, it need to add edges c times. And traverse n times for randomly choosing vertex. This method I used “edgenum” function that will pay c times for check whether this vertex has the maximum edges. So the function represent time could be Θ(n).

A function representing time = C*N.



2. Vertex Ordering

<Smallest Last Ordering>Θ(V+E) = Θ(N)

Method: The basic idea is to pick the vertex with the least degree and then place it at the end of the array. Repeat this operation until all vertices are deleted. So I make many (double)Link List, just like vertex List called Degree List. Each degree List node has its own next pointer to point to the vertex that has the same degree with the certain degree List node.

The preparatory work： My first step is building the data structure for the DegreeList. After that I will traverse from Graph, use “void writeList(PGraph g, DegreeList l) and InsertNextDNode(l, g->p[i].degree, g->p[i].vertex, g)” to build a complete Degree List for next step to used.

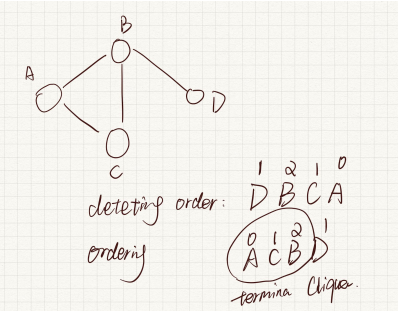
STEP：The method Pseudocode for Smallest Last Ordering is showing:

```
while(v){
        while (degree have not vertex)
        {
            degree++;
        }
        delelte;
        degree--;
}
```

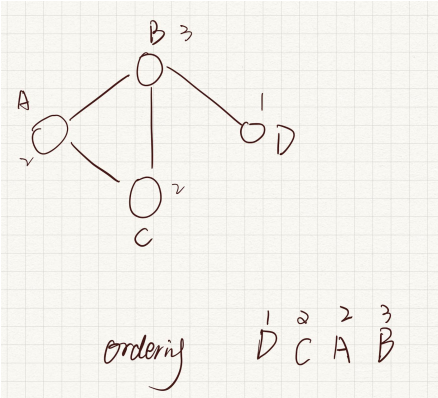
The first while means we need to totally delete n times(that is the number of vertex). And it begin with degree = 0 that ensure we delete vertex from the smallest ordering. After deleting one vertex, we return back to the previous step, because deleting any vertex, all other vertex's degree decreased by at most 1 degree. Finally all vertex has been delete will break out the while loop, the out condition is that the number of deletes is equal to the number of vertices. The following is a detailed deletion method:

1. On the degree to which we are currently allowed to delete, deleting the first node connected with Degree Node LinkList by modifying the both Node's prior and next pointers. And record the vertex's id.
2. According the id, make the graph vertex Link Node' s "isdelete" to -1. That mark this vertex has been deleted.
3. Check all connected vertex with current id's vertex by traverse certain row of Graph Link List and record those vertex until the pointer next == NULL.
4. Traversing the vertex Link List(column) because each structure has a pointer to point to the vertex in the Degree List. And based on those pointer to modify each vertex in the Degree List (everyone's degree --). Modifying those vertex's rear and prior pointer in the Degree List.
5.Repeat Step 4, exit when all node connected in the Graph Node List have been modified.
6. Finally Places the sort queue in the reverse order of vertex deletion.

Walkthrough：



Walkthrough：



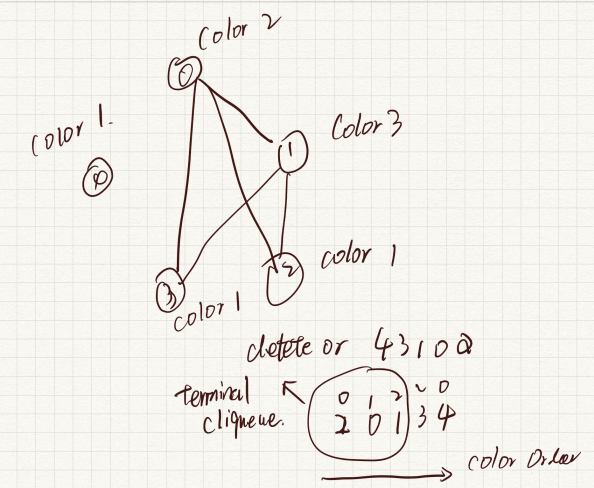
For example:

We assume there are 5 vertex and 5 edges, using skew to generate the graph:

```
Please input the Number of vertices. (MAX = 10,000)
5
Please input the Number of Edges that show conflict between 2 courses. (MAX = 2,000,000)(Complete graph and circle you can input any number)
5
Please choose 1 way to creat the graph you can enter number G : 1.COMPLETE | 2.CYCLE | 3.RANDOM (with DIST below)
3
Which random way you expected: 1. uniform random 2. skew distribution 3. normal
2
5               # 0th value = Number of vertices
0               # 1th value = Vertex 0 is adjacent to Vertex 3
0               # 2th value = Vertex 0 is adjacent to Vertex 2
0               # 3th value = Vertex 0 is adjacent to Vertex 1
1               # 4th value = Vertex 1 is adjacent to Vertex 2
1               # 5th value = Vertex 1 is adjacent to Vertex 3
1               # 6th value = Vertex 1 is adjacent to Vertex 0
2               # 7th value = Vertex 2 is adjacent to Vertex 1
2               # 8th value = Vertex 2 is adjacent to Vertex 0
3               # 9th value = Vertex 3 is adjacent to Vertex 0
3               # 10th value = Vertex 3 is adjacent to Vertex 1
1               # 11th value = starting location for vertex 0's edges
4               # 12th value = starting location for vertex 1's edges
7               # 13th value = starting location for vertex 2's edges
9               # 14th value = starting location for vertex 3's edges
```

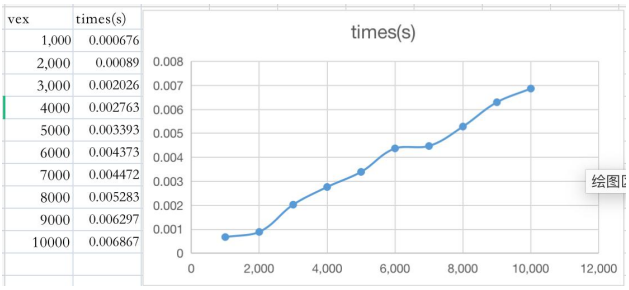
```
NOW degree 0 contains: 4
NOW degree 1 contains: None
NOW degree 2 contains: 3 2
NOW degree 3 contains: 1 0
Which way to ordering the graph you expected: 1. Smallest Last Ordering 2. Smallest Original Degree Last 3. Random Ordering
4. DFS 5. BFS 6.Medium Degree First
1
when deleting vetex 4, it degree is 0
when deleting vetex 3, it degree is 2
when deleting vetex 1, it degree is 2
when deleting vetex 0, it degree is 1
when deleting vetex 2, it degree is 0
 VERTEX_NUMBER <2> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <0> COLOR_NUMBER <2> and ORIGINAL_NUMBER is <3>
 VERTEX_NUMBER <1> COLOR_NUMBER <3> and ORIGINAL_NUMBER is <3>
 VERTEX_NUMBER <3> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <4> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <0>
total use 3 color
the average original degree is 2
the maximum degree when deleted" value for the smallest last ordering is 2
the size of terminal clique = 3
totaltime_for graph=0.000006s
the delete order is:4 3 1 0 2
the ordering is :2 0 1 3 4
finished
```

The left is graph information and the right is all step to delete and colored.



A function representing time：V+E

It will traverse V times to delete V numbers of vertex; For each vertex it will modified its connected E(edges) times to ensure the Degree List is correct. And for each modified time is C time because of Double-Link List.
And I will test the number is double of vertex. For example: 1000 vertex has 2000 edges used uniform random method to generate graph.



vex	times(s)
1,000	0.000676
2,000	0.00089
3,000	0.002026
4000	0.002763
5000	0.003393
6000	0.004373
7000	0.004472
8000	0.005283
9000	0.006297
10000	0.006867

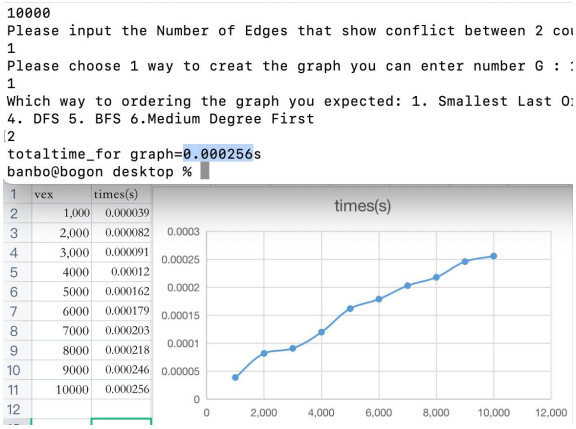
From this graph, we double both the vertex and edges at same time, the time consuming also doubled. When I double it at the same time, it's essentially Θ(N).

<Smallest Original Ordering>Θ(V) = Θ(N)

Method: The basic idea is to pick the vertex with the least degree and then place it at the the array without deleting.

STEP:
1. while loop to traverse each degree's vertex.
2. Find there(like degree 1) exist vertex, record the vertex id, and put it into the array
3. Repeat step 3 until all vertex has been traverse.

A function representing time：V
It will traverse V times and record the vertex's ID. And test condition the graph is complete graph, the edges will not effect the time.



```
10000
Please input the Number of Edges that show conflict between 2 co
1
Please choose 1 way to creat the graph you can enter number G :
1
Which way to ordering the graph you expected: 1. Smallest Last O
4. DFS 5. BFS 6.Medium Degree First
2
totaltime_for graph=0.000256s
banbo@bogon desktop %
```

vex	times(s)
1,000	0.000039
2,000	0.000082
3,000	0.000091
4000	0.00012
5000	0.000162
6000	0.000179
7000	0.000203
8000	0.000218
9000	0.000246
10000	0.000256

<Random Ordering Ordering>Θ(V) = Θ(N)
Method: The basic idea is to pick the vertex randomly and put it into the array for

ordering.
Step:
1.for loop v times.
2.In for loop randomly generate a number as vertex id. Put it in the array. If this vertex's "isdelete" == -1. Repeat step 2 until all vertex has been chose one time.
<DFS> Θ(V+E) = Θ(N)

Method: The basic idea is using DFS algorithm to traverse all vertex, if that vertex first meet, pick it into the array.

Step:(combine two function dfs and dfs traverse,all vertex initialize to white color)
1. for loop to ensure all vertex (especially those vertex have 0 degree have chance to be find)
2. Pick one vertex make its color black.
3. Checking this vertex related Graph Vertex LinkList if there have some vertex color white. Repeat Step 2,3 until haven't sub-vertex can colored and exit.
4. According to the order of first finding white vertex put it into the array.

A function representing time：V+E

The time required to find the adjacency points of all vertices is O(E). And the time taken to access the adjacency points of vertices is O(V)

<BFS> Θ(V+E) = Θ(N)
Method: The basic idea is using BFS algorithm to traverse all vertex, if that vertex first meet, pick it into the array.

Step:(all vertex initialize to visited array and value is 0 means it haven't been traversed)
1. for loop to ensure all vertex (especially those vertex have 0 degree have chance to be find)
2. Pick one vertex that is the first vertex from the queue make its visited[id] = 1 means it already have been find. (if there queue is empty, we can make for loop continue and pick for loop's vertex)
3. Checking this vertex related Graph Vertex LinkList if there have some vertex haven't been find put these vertex to the queue. Repeat Step 2,3 until haven't sub-vertex can colored and exit.
4. According to the order of first finding vertex put it into the array.

A function representing time：V+E

The time required to find the adjacency points of all vertices is O(E). And the time taken to access the adjacency points of vertices is O(V)

<Normal (my designed)> Θ(V) = Θ(N)

Method: The basic idea is choosing the medium degree position's vertex id, and pick it into the array.

Step:

1. define two array, front array and rear array
2. From the medium position to the left and right direction to pick vertex id and put it into array.

A function representing time：V

The time required to taken to access the Vertex Link List points of vertices is O(V)

3. Coloring Algorithm Θ(N)~Θ(N^3)

Method: The basic idea is traverse every vertex, and check its connected vertex color, finally choose a color that number is smallest.

Step:

1. First for loop to traverse each vertex set all color is 0.
2. For each vertex, check the vertex's color that connected to determined what color we need, Default set this vertex is 1,and compare with each vertex connect with it. If both colors are the same. Color ++; Repeat check the Color from the beginning. IF all colors haven't been used. Let this vertex equals to the color value.

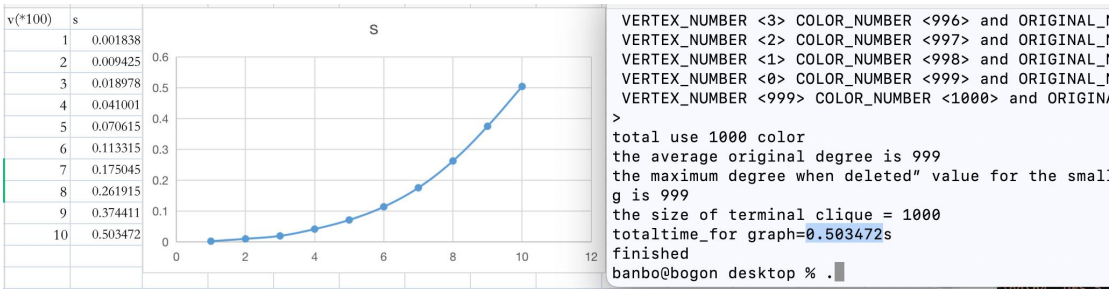
A function representing time： N~N^3 (depend on the kind of graph)

F(n) = 1!+2!+3!----N! (nearly equal to N^3)

In best case, A function representing time: Θ(V);For example: there have 100 vertex,and 0 edges. Step 2's time consuming is equals to 0; so we needn't consider it.

In worst case, A function representing time: Θ(N^3) = Θ(N^3)(there N is represent V); Like complete graph, 100vetex, each one has to check 1!+2!+3!----N! times, so the time consuming is really important.

For example： (complete graph)(Smallest Last Ordering)

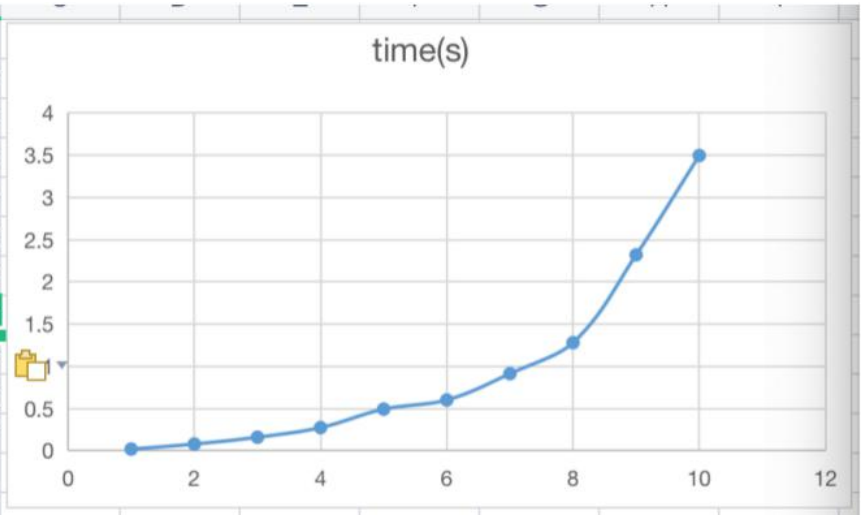


v(*100)	s
1	0.001838
2	0.009425
3	0.018978
4	0.041001
5	0.070615
6	0.113315
7	0.175045
8	0.261915
9	0.374411
10	0.503472


```
 VERTEX_NUMBER <3> COLOR_NUMBER <996> and ORIGINAL_
 VERTEX_NUMBER <2> COLOR_NUMBER <997> and ORIGINAL_
 VERTEX_NUMBER <1> COLOR_NUMBER <998> and ORIGINAL_
 VERTEX_NUMBER <0> COLOR_NUMBER <999> and ORIGINAL_
 VERTEX_NUMBER <999> COLOR_NUMBER <1000> and ORIGIN
>
total use 1000 color
the average original degree is 999
the maximum degree when deleted" value for the smal
g is 999
the size of terminal clique = 1000
totaltime_for graph=0.503472s
finished
banbo@bogon desktop % .
```

(complete graph)(Smallest Original Degree Last)

vertex(k)	time(s)
1	0.021257
2	0.081226
3	0.161555
4	0.275715
5	0.494232
6	0.602748
7	0.913947
8	1.280678
9	2.320323
10	3.495839

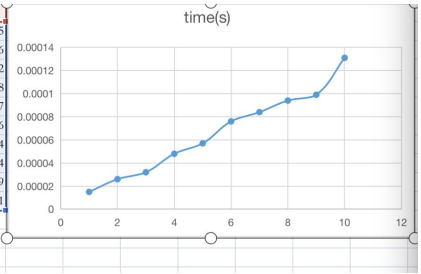


Also $\Theta(N^3)$;
Other ordering method I also test, it cannot affect the coloring algorithm's function representing time.

Now we consider the best case, that is we do not need to enter the step2, each vertex contain 0 edges. So it only pay n times to traverse the vertex.

For example：(graph only have vertex)

vertex(k)	time(s)
1	0.000015
2	0.000026
3	0.000032
4	0.000048
5	0.000057
6	0.000076
7	0.000084
8	0.000094
9	0.000099
10	0.000131

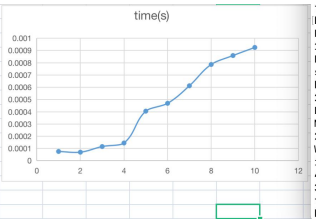



```
10000
Please input the Number of Edges that show conflict between 2 course
s. (MAX = 2,000,000)(Complete graph and circle you can input any num
ber)
0
Please choose 1 way to creat the graph you can enter number G : 1.CO
MPLETE | 2.CYCLE | 3.RANDOM (with DIST below)
3
Which random way you expected: 1. uniform random 2. skew distributio
n 3. normal
1
Which way to ordering the graph you expected: 1. Smallest Last Order
ing 2. Smallest Original Degree Last 3. Random Ordering
4. DFS 5. BFS 6.Medium Degree First
1
totaltime_for graph=0.000131s
```

And we can consider the middle condition. Because the actually Θ(V). So we just stimulate the middle condition for representing time. We assume V = N and we test it. (That condition like circle, so I just use circle graph to test) for each vertex it only have 2 vertex, so the representing time function is like: f(n) = C*N Θ(V)

For example：(circle graph)

vertex(k)	time(s)
1	0.000076
2	0.00007
3	0.000116
4	0.000145
5	0.000405
6	0.000469
7	0.000612
8	0.000786
9	0.000859
10	0.000925




```
totaltime_for graph=0.000859s
banbo@bogon desktop % ./a.out
Please input the Number of vertices. (MAX = 10,000)
10000
Please input the Number of Edges that show conflict between 2 course
s. (MAX = 2,000,000)(Complete graph and circle you can input any num
ber)
2
Please choose 1 way to creat the graph you can enter number G : 1.CO
MPLETE | 2.CYCLE | 3.RANDOM (with DIST below)
2
Which way to ordering the graph you expected: 1. Smallest Last Order
ing 2. Smallest Original Degree Last 3. Random Ordering
4. DFS 5. BFS 6.Medium Degree First
3
totaltime_for graph=0.000925s
banbo@bogon desktop %
```

4. Vertex Ordering Capabilities

I set input V = 2000; E = 10000;

<Smallest Original Degree Last>

1. These for v=2000 and E = 10000 to compare with other method.

Complete graph:

```
total use 2000 color
the average original degree is 1999
the maximum degree when deleted" value for the smallest last orderin
g is 1999
the size of terminal clique = 2000
```

Circle:

```
total use 2 color
the average original degree is 2
the maximum degree when deleted" value for the smallest last orderin
g is 2
the size of terminal clique = 2
totaltime_for graph=0.000070s
banbo@bogon desktop %
```

Uniform:

```
total use 7 color
the average original degree is 10
the maximum degree when deleted" value for the smallest last orderin
g is 7
the size of terminal clique = 3
totaltime_for graph=0.000631s
```

Skewed:

```
total use 8 color
the average original degree is 10
the maximum degree when deleted" value for the smallest last orderin
g is 8
the size of terminal clique = 3
totaltime_for graph=0.000898s
```

Normal:

```
total use 7 color
the average original degree is 10
the maximum degree when deleted" value for the smallest last orderin
g is 6
the size of terminal clique = 7
totaltime_for graph=0.000294s
```

2. These are for A graph indicating the degree when deleted on the vertical axes and the order colored on the x-axis.(Out put contain the information about Maximum when degree,and terminal clique)

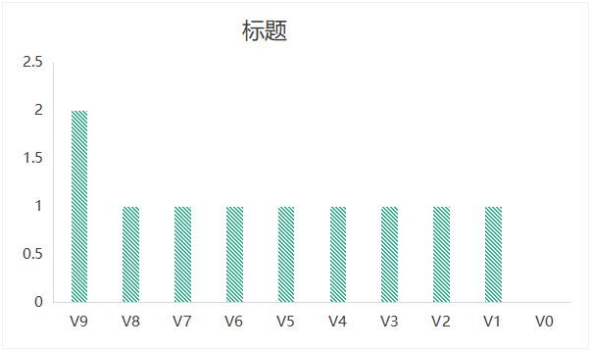
For each graph I draw a bar graph, but the actually coloring order is reverse with the bar order. And the terminal clique can determine the lower bound for the color combine with when V = 2000 E =10000, the color need 7,and the terminal clique is 3. And I think the upper bound is the max value of degree when deleting because of worst case it is a complete graph.

Complete graph:

```
when deleting vetex 9, it degree is 9
when deleting vetex 0, it degree is 8
when deleting vetex 1, it degree is 7
when deleting vetex 2, it degree is 6
when deleting vetex 3, it degree is 5
when deleting vetex 4, it degree is 4
when deleting vetex 5, it degree is 3
when deleting vetex 6, it degree is 2
when deleting vetex 7, it degree is 1
when deleting vetex 8, it degree is 0
 VERTEX_NUMBER <8> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <9>
 VERTEX_NUMBER <7> COLOR_NUMBER <2> and ORIGINAL_NUMBER is <9>
 VERTEX_NUMBER <6> COLOR_NUMBER <3> and ORIGINAL_NUMBER is <9>
 VERTEX_NUMBER <5> COLOR_NUMBER <4> and ORIGINAL_NUMBER is <9>
 VERTEX_NUMBER <4> COLOR_NUMBER <5> and ORIGINAL_NUMBER is <9>
 VERTEX_NUMBER <3> COLOR_NUMBER <6> and ORIGINAL_NUMBER is <9>
 VERTEX_NUMBER <2> COLOR_NUMBER <7> and ORIGINAL_NUMBER is <9>
 VERTEX_NUMBER <1> COLOR_NUMBER <8> and ORIGINAL_NUMBER is <9>
 VERTEX_NUMBER <0> COLOR_NUMBER <9> and ORIGINAL_NUMBER is <9>
 VERTEX_NUMBER <9> COLOR_NUMBER <10> and ORIGINAL_NUMBER is <9>
total use 10 color
the average original degree is 9
the maximum degree when deleted" value for the smallest last orderin
g is 9
the size of terminal clique = 10
the delete order is:9 0 1 2 3 4 5 6 7 8
the ordering is :8 7 6 5 4 3 2 1 0 9
```

Circle:

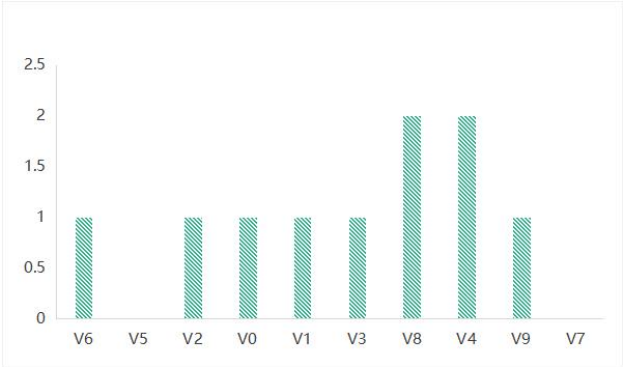
```
when deleting vetex 9, it degree is 2
when deleting vetex 8, it degree is 1
when deleting vetex 7, it degree is 1
when deleting vetex 6, it degree is 1
when deleting vetex 5, it degree is 1
when deleting vetex 4, it degree is 1
when deleting vetex 3, it degree is 1
when deleting vetex 2, it degree is 1
when deleting vetex 1, it degree is 1
when deleting vetex 0, it degree is 0
 VERTEX_NUMBER <0> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <1> COLOR_NUMBER <2> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <2> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <3> COLOR_NUMBER <2> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <4> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <5> COLOR_NUMBER <2> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <6> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <7> COLOR_NUMBER <2> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <8> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <9> COLOR_NUMBER <2> and ORIGINAL_NUMBER is <2>
total use 2 color
the average original degree is 2
the maximum degree when deleted" value for the smallest last orderin
g is 2
the size of terminal clique = 2
the delete order is:9 8 7 6 5 4 3 2 1 0
the ordering is :0 1 2 3 4 5 6 7 8 9
finished
banbo@bogon desktop %
```



Uniform:

```
1
10              # 0th value = Number of vertices
0               # 1th value = Vertex 0 is adjacent to Vertex 1
1               # 2th value = Vertex 1 is adjacent to Vertex 3
1               # 3th value = Vertex 1 is adjacent to Vertex 0
2               # 4th value = Vertex 2 is adjacent to Vertex 4
3               # 5th value = Vertex 3 is adjacent to Vertex 4
3               # 6th value = Vertex 3 is adjacent to Vertex 1
4               # 7th value = Vertex 4 is adjacent to Vertex 3
4               # 8th value = Vertex 4 is adjacent to Vertex 7
4               # 9th value = Vertex 4 is adjacent to Vertex 9
4               # 10th value = Vertex 4 is adjacent to Vertex 8
4               # 11th value = Vertex 4 is adjacent to Vertex 2
5               # 12th value = Vertex 5 is adjacent to Vertex 6
6               # 13th value = Vertex 6 is adjacent to Vertex 5
7               # 14th value = Vertex 7 is adjacent to Vertex 9
7               # 15th value = Vertex 7 is adjacent to Vertex 4
8               # 16th value = Vertex 8 is adjacent to Vertex 9
8               # 17th value = Vertex 8 is adjacent to Vertex 4
9               # 18th value = Vertex 9 is adjacent to Vertex 7
9               # 19th value = Vertex 9 is adjacent to Vertex 4
9               # 20th value = Vertex 9 is adjacent to Vertex 8
```

```
when deleting vetex 6, it degree is 1
when deleting vetex 5, it degree is 0
when deleting vetex 2, it degree is 1
when deleting vetex 0, it degree is 1
when deleting vetex 1, it degree is 1
when deleting vetex 3, it degree is 1
when deleting vetex 8, it degree is 2
when deleting vetex 4, it degree is 2
when deleting vetex 9, it degree is 1
when deleting vetex 7, it degree is 0
 VERTEX_NUMBER <7> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <9> COLOR_NUMBER <2> and ORIGINAL_NUMBER is <3>
 VERTEX_NUMBER <4> COLOR_NUMBER <3> and ORIGINAL_NUMBER is <5>
 VERTEX_NUMBER <8> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <3> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <1> COLOR_NUMBER <2> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <0> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <1>
 VERTEX_NUMBER <2> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <1>
 VERTEX_NUMBER <5> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <1>
 VERTEX_NUMBER <6> COLOR_NUMBER <2> and ORIGINAL_NUMBER is <1>
total use 3 color
the average original degree is 2
the maximum degree when deleted" value for the smallest last orderin
g is 2
the size of terminal clique = 3
the delete order is:6 5 2 0 1 3 8 4 9 7
the ordering is :7 9 4 8 3 1 0 2 5 6
finished
banbo@bogon desktop %
```



Skewed:

```
10              # 0th value = Number of vertices
0               # 1th value = Vertex 0 is adjacent to Vertex 2
0               # 2th value = Vertex 0 is adjacent to Vertex 3
0               # 3th value = Vertex 0 is adjacent to Vertex 4
1               # 4th value = Vertex 1 is adjacent to Vertex 6
1               # 5th value = Vertex 1 is adjacent to Vertex 5
1               # 6th value = Vertex 1 is adjacent to Vertex 4
2               # 7th value = Vertex 2 is adjacent to Vertex 0
2               # 8th value = Vertex 2 is adjacent to Vertex 3
3               # 9th value = Vertex 3 is adjacent to Vertex 8
3               # 10th value = Vertex 3 is adjacent to Vertex 0
3               # 11th value = Vertex 3 is adjacent to Vertex 5
3               # 12th value = Vertex 3 is adjacent to Vertex 2
4               # 13th value = Vertex 4 is adjacent to Vertex 8
4               # 14th value = Vertex 4 is adjacent to Vertex 1
4               # 15th value = Vertex 4 is adjacent to Vertex 0
5               # 16th value = Vertex 5 is adjacent to Vertex 1
5               # 17th value = Vertex 5 is adjacent to Vertex 3
6               # 18th value = Vertex 6 is adjacent to Vertex 1
8               # 19th value = Vertex 8 is adjacent to Vertex 3
8               # 20th value = Vertex 8 is adjacent to Vertex 4
```

```
when deleting vetex 9, it degree is 0
when deleting vetex 7, it degree is 0
when deleting vetex 6, it degree is 1
when deleting vetex 1, it degree is 2
when deleting vetex 5, it degree is 1
when deleting vetex 4, it degree is 2
when deleting vetex 8, it degree is 1
when deleting vetex 3, it degree is 2
when deleting vetex 2, it degree is 1
when deleting vetex 0, it degree is 0
 VERTEX_NUMBER <0> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <3>
 VERTEX_NUMBER <2> COLOR_NUMBER <2> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <3> COLOR_NUMBER <3> and ORIGINAL_NUMBER is <4>
 VERTEX_NUMBER <8> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <4> COLOR_NUMBER <2> and ORIGINAL_NUMBER is <3>
 VERTEX_NUMBER <5> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <1> COLOR_NUMBER <3> and ORIGINAL_NUMBER is <3>
 VERTEX_NUMBER <6> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <1>
 VERTEX_NUMBER <7> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <0>
 VERTEX_NUMBER <9> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <0>
total use 3 color
the average original degree is 2
the maximum degree when deleted" value for the smallest last orderin
g is 2
the size of terminal clique = 3
the delete order is:9 7 6 1 5 4 8 3 2 0
the ordering is :0 2 3 8 4 5 1 6 7 9
finished
banbo@bogon desktop %
```

Normal:

```
10              # 0th value = Number of vertices
0               # 1th value = Vertex 0 is adjacent to Vertex 3
1               # 2th value = Vertex 1 is adjacent to Vertex 3
2               # 3th value = Vertex 2 is adjacent to Vertex 3
3               # 4th value = Vertex 3 is adjacent to Vertex 4
3               # 5th value = Vertex 3 is adjacent to Vertex 1
3               # 6th value = Vertex 3 is adjacent to Vertex 5
3               # 7th value = Vertex 3 is adjacent to Vertex 9
3               # 8th value = Vertex 3 is adjacent to Vertex 2
3               # 9th value = Vertex 3 is adjacent to Vertex 7
3               # 10th value = Vertex 3 is adjacent to Vertex 0
3               # 11th value = Vertex 3 is adjacent to Vertex 8
3               # 12th value = Vertex 3 is adjacent to Vertex 6
4               # 13th value = Vertex 4 is adjacent to Vertex 8
4               # 14th value = Vertex 4 is adjacent to Vertex 3
5               # 15th value = Vertex 5 is adjacent to Vertex 3
6               # 16th value = Vertex 6 is adjacent to Vertex 3
7               # 17th value = Vertex 7 is adjacent to Vertex 3
8               # 18th value = Vertex 8 is adjacent to Vertex 4
8               # 19th value = Vertex 8 is adjacent to Vertex 3
9               # 20th value = Vertex 9 is adjacent to Vertex 3
```

```
when deleting vetex 9, it degree is 1
when deleting vetex 7, it degree is 1
when deleting vetex 6, it degree is 1
when deleting vetex 5, it degree is 1
when deleting vetex 2, it degree is 1
when deleting vetex 1, it degree is 1
when deleting vetex 0, it degree is 1
when deleting vetex 3, it degree is 2
when deleting vetex 8, it degree is 1
when deleting vetex 4, it degree is 0
 VERTEX_NUMBER <4> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <8> COLOR_NUMBER <2> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <3> COLOR_NUMBER <3> and ORIGINAL_NUMBER is <9>
 VERTEX_NUMBER <0> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <1>
 VERTEX_NUMBER <1> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <1>
 VERTEX_NUMBER <2> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <1>
 VERTEX_NUMBER <5> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <1>
 VERTEX_NUMBER <6> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <1>
 VERTEX_NUMBER <7> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <1>
 VERTEX_NUMBER <9> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <1>
total use 3 color
the average original degree is 2
the maximum degree when deleted" value for the smallest last orderin
g is 2
the size of terminal clique = 3
the delete order is:9 7 6 5 2 1 0 3 8 4
the ordering is :4 8 3 0 1 2 5 6 7 9
finished
banbo@bogon desktop %
```

<Smallest Original Degree Last>

Complete graph:

```
 VERTEX_NUMBER <1998> COLOR_NUMBER <1999>
99>
 VERTEX_NUMBER <1999> COLOR_NUMBER <2000>
99>
total use 2000 color
the average original degree is 1999
```

Circle:

```
total use 2 color
the average original degree is 2
```

Uniform:

```
total use 7 color
the average original degree is 10
```

Skewed:

```
total use 8 color
the average original degree is 10
```

Normal:

```
total use 7 color
the average original degree is 10
```

<Random Ordering>

Complete graph:

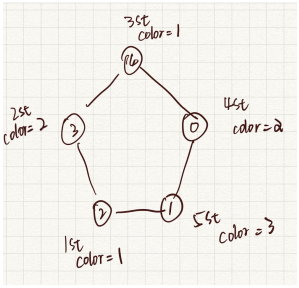
```
 VERTEX_NUMBER <469> COLOR_NUMBER <2000>
9>
total use 2000 color
the average original degree is 1999
```

Circle:

```
 VERTEX_NUMBER <128> COLOR_NUMBER <3> and
total use 3 color
the average original degree is 2
```

we can see their use 3 colors, and it is not the optimal solution, because their order for list is disorder, could happen some problem like this:

```
 VERTEX_NUMBER <2> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <3> COLOR_NUMBER <2> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <4> COLOR_NUMBER <1> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <0> COLOR_NUMBER <2> and ORIGINAL_NUMBER is <2>
 VERTEX_NUMBER <1> COLOR_NUMBER <3> and ORIGINAL_NUMBER is <2>
total use 3 color
the average original degree is 2
totaltime_for graph=0.000006s
```



Obviously, it can generate the not optimal option.

Uniform:

```
 VERTEX_NUMBER <337> COLOR_NUMBER <4>
total use 8 color
the average original degree is 10
```

Skewed:

```
total use 9 color
the average original degree is 10
```

Normal:

```
total use 7 color
the average original degree is 10
```

<DFS>

Complete graph:

```
total use 2000 color
the average original degree is 1999
```

Circle:

```
total use 2 color
the average original degree is 2
```

Uniform:

```
total use 8 color
the average original degree is 10
```

Skewed:

```
total use 9 color
the average original degree is 10
```

Normal:

```
total use 7 color
the average original degree is 10
```

<BFS>

Complete graph:

```
total use 2000 color
the average original degree is 1999
```

Circle:

```
total use 2 color
the average original degree is 2
```

Uniform:

```
total use 8 color
the average original degree is 10
```

Skewed:

```
total use 9 color
the average original degree is 10
```

Normal:

```
total use 7 color
the average original degree is 10
```

<Medium degree first >

Complete graph:

```
total use 2000 color
the average original degree is 1999
```

Circle:

```
total use 2 color
the average original degree is 2
```

Uniform:

```
total use 9 color
the average original degree is 10
```

Skewed:

```
total use 10 color
the average original degree is 10
```

Normal:

```
total use 7 color
the average original degree is 10
```

An in-depth analysis of the capabilities different orderings based on my results:
Complete graph for each method is the same, it has to use V numbers color to color the graph.
Circle: All methods except random ordering have the same capabilities. The specific reason I write in the Random Ordering part.
Uniform random graph: Smallest Last Ordering and Smallest Original Degree Last are better than other methods. The color order according to the method sequence is: 7 7 8 8 8 9;
Skewed Random Graph： The color order according to the method sequence is: 8 8 9 9 10.Smallest Last Ordering and Smallest Original Degree Last are better than other methods.
Normal graph： The color order according to the method sequence is: 7 7 7 7 7 7; All 6 method have the same capabilities, because this kind of my defined way to generate graph is prioritizing all edges assigned to a vertex, when this vertex cannot add edges, program will chose next vertex to add edges. So it can generate more sub-complete graphs. The sub-complete that has the max number of vertex will determine the color number. Like Complete graph, ordering methods are not important to affect the color number.
The first two ways of random shows Smallest Last Ordering and Smallest Original Degree Last are better than other methods. In fact, more larger vertex number and testing, Smallest Last Ordering is better than Smallest Original Degree Last. Because those two ways starting from the minimum reduces color waste;

Tips: all time measure function I mark in code you can cancel the “//”to test it.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
int *s;
unsigned short xsub1[3];
// there three variable is used to build the graph's dfs
#define gray -1
#define white 0
#define black 1
int *gcolor;

/*----------------------------------------------------------------DLinkList-----------------------------
------------------------------*/
typedef struct DNode
{
	int id;
	int degree;
	struct DNode *prior, *next;
} DNode, *DLinkList;

typedef struct DegreeNode
{
	DLinkList a;
	int maxdegree;
} DegreeNode, *DegreeList;

void creatDegList(DegreeList l, int d)
{
	l->a = (DNode *)calloc((d + 1), sizeof(DNode));
	for (int i = 0; i <= d; i++)
	{
		l->a[i].degree = i;
		l->a[i].prior = NULL;
		l->a[i].next = NULL;
		l->a[i].id = 0;
	}
```

```
	l->maxdegree = d;
}
/*-----------------------------------------------------------Graph
LinkList-----------------------------------------------------------*/
typedef struct ANode
{
	int vertex;
	struct ANode *next;
} ANode, *PANode; //define Edge connected vertex

typedef struct VNode
{
	int vertex;
	int degree;
	int original_degree;
	int isdelete;
	int color;
	PANode first;
	DLinkList map;
} VNode, *PVNode; //difine vertex

typedef struct Graph
{
	PVNode p;
	int vexnum;
	int edgenum;
} Graph, *PGraph; // graph

void creatgraph(PGraph g, int v)
{
	g->p = (VNode *)calloc(v, sizeof(VNode));
	for (int i = 0; i < v; i++)
	{
		g->p[i].vertex = i;
		g->p[i].first = NULL;
		g->p[i].degree = 0;
		g->p[i].original_degree = 0;
		g->p[i].isdelete = 0;
		g->p[i].color = 0;
		g->p[i].map = NULL;
	}
	g->vexnum = v;
	g->edgenum = 0;
}
```

```
//create    and initialize the graph

void addedge(PGraph g, int from, int to)
{
	PANode a, b;
	int succ = 0;
	b = g->p[from].first;
	if (from != to)
	{
		a = (PANode)malloc(sizeof(ANode));
		a->next = NULL;
		a->vertex = to;
		a->next = g->p[from].first;
		g->p[from].first = a;
		g->p[from].degree++;
		g->p[from].original_degree++;
		//printf("插入---<%d><%d>----------", from, to);
		succ++;


		a = (PANode)malloc(sizeof(ANode));
		a->next = NULL;
		a->vertex = from;
		a->next = g->p[to].first;
		g->p[to].first = a;
		g->p[to].degree++;
		g->p[to].original_degree++;
		//printf("---------<%d><%d>---成功\n", to, from);
		succ++;

	    if (succ == 2)
	    {
		g->edgenum++;
	    }
	}
	else
	{
	    // printf("FALSE\n");
	}
}
/*------------------------------------------------------------create graph with different methods -------------------------------------------------------------*/
void completegraph(PGraph g)
{
```

```
    for (int i = 0; i < g->vexnum; i++)
    {
        for (int j = i+1; j < g->vexnum; j++)
        {
            addedge(g, i, j);
        }
    }
}

void out(PGraph g)
{
    printf("%d\t\t# 0th value = Number of vertices\n", g->vexnum);
    int count = 1;
    for (int i = 0; i < g->vexnum; i++)
    {
        PANode a = g->p[i].first;
        if (a != NULL)
        {
            s[i] = count;
        }
        while (a != NULL)
        {

            printf("%d\t\t# %dth value = Vertex %d is adjacent to Vertex %d\n",
g->p[i].vertex, count, i, a->vertex);
            a = a->next;
            count++;
        }
    }
    for (int i = 0; i < g->vexnum; i++)
    {
        if (s[i] != 0)
        {
            printf("%d\t\t# %dth value = starting location for vertex %d' s edges\n", s[i],
count, i);
            count++;
        }
    }
}

void circle(PGraph g)
{
    for (int i = 0; i < g->vexnum; i++)
    {
```

```
            int j = i + 1;
            if (j == (g->vexnum))
            {
                j = 0;
            }
            addedge(g, i, j);
        }
    }

    int isedge(PGraph g, int from, int to)
    {
        PANode a, b;
        b = g->p[from].first;
        while (b)
        {
            if (b->vertex == to)
            {
                //printf("边存在  %d %d \n", from, to);
                return 1;
            }
            b = b->next;
        }
        //printf("边不存在  %d %d \n", from, to);
        return 0;
    }
    int edgenum(PGraph g, int from)
    {
        //method 1
        return g->p[from].original_degree;
        //method 2 not efficient
        // PANode a, b;
        // b = g->p[from].first;
        // int number = 0;
        // while (b)
        // {
        //     if (b != NULL)
        //     {
        //         number++;
        //     }
        //     b = b->next;
        // }
        // return number;
    }
```

```
void uniform(PGraph g, int e)
{
	int i;
	int p;
	for (int j = 0; j < e; j++)
	{
		while (1)
		{
			i = rand() % (g->vexnum);
			while (1)
			{
				p = rand() % (g->vexnum);
				if (i != p)
				{
					break;
				}
			}
			if (isedge(g, i, p) == 0)
			{
				break;
			}
			//sleep(1);
		}
		addedge(g, i, p);
		//sleep(1);
	}
}

int skewvertex(int a)
{
	float x;
	float slope;
	float random = 0;
	int number;
	float v = a;
	int choosed;
	int value = 0;
	float *possib = (float *)calloc(a, sizeof(float));

	x = 2 / (v - 1);
	slope = -(x) / v;

	for (int i = 0; i < v; i++)
	{
```

```
            possib[i] = (i + 1) * slope + x;
            random = random + possib[i];
        }
        float count = 0;
        for (int i = 0; i < a; i++)
        {
            possib[i] = count + possib[i] * 1000000;
            count = possib[i];
        }
        possib[a - 1] = possib[a - 1] + 100;

        number = rand() % 1000100;
        for (int j = 0; j < a; j++)
        {
            if (number <= possib[j])
            {
                value = j;
                break;
            }
        }
        return value;
}

void skew(PGraph g, int e)
{
        int first;
        int second;
        int isok;
        for (int i = 0; i < e; i++)
        {
            while (1)
            {
                first = skewvertex(g->vexnum);
                while (1)
                {
                    second = skewvertex(g->vexnum);
                    if (second != first)
                    {
                        break;
                    }
                }
                isok = isedge(g, first, second);
                if (isok != 1)
                {
```

```
                addedge(g, first, second);
                break;
            }
        }
    }
}

void normal(PGraph g, int e)
{
    int first;
    int second;
    int isok;
    int new;
    int count = 0;
    first = rand() % g->vexnum;
    while (1)
    {

        if (edgenum(g, first) == (g->vexnum - 1))
        {
            while (1)
            {
                first = rand() % g->vexnum;
                if (edgenum(g, first) < (g->vexnum - 1))
                {
                    break;
                }
            }
        }

        while (1)
        {
            while (1)
            {
                second = rand() % g->vexnum;
                if (second != first)
                {
                    break;
                }
            }
            isok = isedge(g, first, second);
            if (isok != 1)
            {
                addedge(g, first, second);
```

```
                    count++;
                    break;
                }
            }
            if (count >= e)
            {
                break;
            }
        }
}

void InsertNextDNode(DegreeList l, int d, int id, PGraph g) //d = degree id = vertex id
{
    DLinkList s;
    s = (DNode *)calloc(1, sizeof(DNode));
    s->next = l->a[d].next;
    s->id = id;
    if (l->a[d].next != NULL)
    {
        l->a[d].next->prior = s;
    }
    s->prior = &(l->a[d]);
    l->a[d].next = s;
    g->p[id].map = s;
}


int RemoveAnyVertex(PGraph g, DegreeList l, int deg)
{

    //delete any vertex that the value of deg
    int id = 0;
    DNode *q = l->a[deg].next;
    id = l->a[deg].next->id;
    l->a[deg].next = q->next;
    if (q->next != NULL)
    {
        q->next->prior = &(l->a[deg]);
    }
    free(q);
    g->p[id].isdelete = -1;
    g->p[id].map = NULL;
    g->p[id].degree = -1;
```

```
    //change other connected vertex degree;
    PANode a;
    a = g->p[id].first;
    int record = 0;

    int change_degree = 0;

    while (a != NULL)
    {

        record = (int)a->vertex;

        if (g->p[record].isdelete != -1)
        {
            change_degree = (int)(g->p[record].degree) - 1;
            g->p[record].degree--;

            DLinkList temp = (DNode *)(g->p[record].map);
            temp->prior->next = temp->next;

            if (temp->next != NULL)
            {
                temp->next->prior = temp->prior;
            }

            temp->next = l->a[change_degree].next;

            if (l->a[change_degree].next != NULL)
            {
                l->a[change_degree].next->prior = temp;
            }
            temp->prior = &(l->a[change_degree]);
            l->a[change_degree].next = temp;
        }
        a = a->next;
    }
    return id;
}

int maxdegree(PGraph g)
{
    int max = 0;
    for (int i = 0; i < g->vexnum; i++)
    {
```

```
            if (g->p[i].degree >= max)
            {
                max = g->p[i].degree;
            }
        }
        return max;
}
void writeList(PGraph g, DegreeList l)
{
        for (int i = 0; i < g->vexnum; i++)
        {
            InsertNextDNode(l, g->p[i].degree, g->p[i].vertex, g);
        }
}
void outList(DegreeList l)
{
        DLinkList s;
        for (int i = 0; i <= l->maxdegree; i++)
        {
            printf("NOW degree %d contains： ", i);
            if (l->a[i].next != NULL)
            {
                s = l->a[i].next;
                while (1)
                {
                    printf("%d ", s->id);
                    s = s->next;
                    if (s == NULL)
                    {
                        printf("\n");
                        break;
                    }
                }
            }
            else
            {
                printf("None\n");
            }
        }
}
/*-------------------------------------------------------------Smallest                                Last
Ordering-------------------------------------------------------------*/
int *orderlist;
int *deleteorder;
```

```
int record_degree;
int max_degree = 0;
void SmallestLastOrdering(PGraph g, DegreeList l)
{
	record_degree = 0;
	int last_degree = 0;
	int counter = g->vexnum - 1;
	int id = 0;
	int degree = 0;
	int counter1 = 0;
	while (counter1 < (g->vexnum))
	{
		while (l->a[degree].next == NULL)
		{
			degree++;
		}

		id = RemoveAnyVertex(g, l, degree);
		 //this line for out put degree for graph
		printf("when deleting vetex %d, it degree is %d\n",id,degree);


		if (degree > last_degree)
		{
			record_degree = degree;
		}
		if (degree < last_degree && degree != (last_degree - 1))
		{
			record_degree = degree;
		}

		last_degree = degree;

		if(degree>max_degree){
			max_degree = degree;
		}
		deleteorder[counter1] = id;
		orderlist[counter] = id;
		counter--;
		degree--;
		counter1++;
		if (degree == -1)
		{
			degree = 0;
```

```
            }

            // outList(l);
            // printf("--------------------\n");
        }
}

// while(v){
//              while (degree have not vertex)
//              {
//                      degree++;
//              }
//              delelte;
//              degree--;
// }

/*-------------------------------------------------------------Smallest Original Degree Last-------------------------------------------------------------*/

void SmallestOriginalDegreeLast(DegreeList l, PGraph g)
{
      DNode *q;
      int count = g->vexnum - 1;
      for (int i = 0; i <= (l->maxdegree); i++)
      {
            q = l->a[i].next;
            while (q != NULL)
            {
                  orderlist[count] = q->id;
                  count--;
                  q = q->next;
            }
      }
}
/*-------------------------------------------------------------  RandomOrdering  -------------------------------------------------------------*/
void RandomOrdering(DegreeList l, PGraph g)
{
      int random_number = 0;
      int counter = 0;
      for (int i = 0; i < g->vexnum; i++)
      {
            while (1)
            {
```

```
                random_number = rand() % g->vexnum;
                if (g->p[random_number].isdelete != -1)
                {
                    break;
                }
            }
            g->p[random_number].isdelete = -1;
            orderlist[counter] = random_number;
            counter++;
        }
}

/*-----------------------------------------------------------dfs-----------------------------------
-----------------------*/

int counter_dfs = 0;
void dfs(Graph g, int i)
{
    PANode p = NULL;
    gcolor[i] = black;
    orderlist[counter_dfs] = i;
    counter_dfs++;
    p = g.p[i].first;
    while (p != NULL)
    {
        if (gcolor[p->vertex] == white)
        {
            dfs(g, p->vertex);
        }
        p = p->next;
    }
}

void DFSTraverse(Graph g)
{
    int i;
    for (i = 0; i < g.vexnum; i++)
    {
        gcolor[i] = white;
    }

    for (i = 0; i < g.vexnum; i++)
    {
        if (gcolor[i] == white)
```

```
            {
                dfs(g, i);
            }
        }
}

/*-------------------------------------------------------------BFS-----------------------------------
-------------------------*/
typedef struct QNode
{
    int vertex;
    struct QNode *next;
} QNode, *PQNode;

typedef struct Queue
{
    PQNode front, rear;
} Queue, *PQueue;

PQueue initQueue()
{
    PQueue q = (PQueue)malloc(sizeof(Queue));
    PQNode head = (PQNode)malloc(sizeof(QNode));
    q->front = q->rear = head;
    head->next = NULL;
    return q;
}

void enQueue(PQueue q, int i)
{
    PQNode p = (PQNode)calloc(1, sizeof(QNode));
    p->vertex = i;
    p->next = NULL;
    q->rear->next = p;
    q->rear = p;
}
int isEmpty(PQueue q)
{
    if (q->front == q->rear)
    {

        return 1;
    }
    else
```

```
    {
        return 0;
    }
}
int deQueue(PQueue q)
{
    if (isEmpty(q))
    {
        printf("the queue is empty\n");
        exit(-1);
    }

    PQNode p = q->front->next;
    q->front->next = p->next;
    if (p == q->rear)
        q->rear = q->front;
    int data = p->vertex;
    free(p);
    return data;
}

void BFSTraverse(Graph g)
{

    PQueue q = initQueue();
    PANode p;
    int *visited = (int *)calloc((g.vexnum), sizeof(int));
    int v;
    int counter_bfs = 0;
    for (int i = 0; i < g.vexnum; i++)
        visited[i] = -1;
    for (int i = 0; i < g.vexnum; i++)
    {
        if (visited[i] == -1)
        {
            visited[i] = 1;
            orderlist[counter_bfs] = i;
            counter_bfs++;
            enQueue(q, i);
            while (isEmpty(q) == 0)
            {
                v = deQueue(q);
                for (p = g.p[v].first; p != NULL; p = p->next)
                {
```

```
                        if (visited[p->vertex] == -1)
                        {
                            orderlist[counter_bfs] = p->vertex;
                            counter_bfs++;
                            visited[p->vertex] = 1;
                            enQueue(q, p->vertex);
                        }
                    }
                }
            }
        }
}

void MediumDegreeFirst(PGraph g, DegreeList l)
{
    int *orderlist_1;
    int counter = 0;
    orderlist_1 = (int *)calloc(g->vexnum, sizeof(int));

    DNode *c;

    for (int i = 0; i <= l->maxdegree; i++)
    {
        c = l->a[i].next;
        if (c == NULL)
        {
            continue;
        }
        else
        {
            while (c != NULL)
            {
                orderlist_1[counter] = c->id;
                counter++;
                c = c->next;
            }
        }
    }
    int a = 0;
    int b = 0;
    b = g->vexnum % 2;
    a = g->vexnum / 2;
    int counter_1 = 0;
    if (b > 0)
```

```
    {
        orderlist[counter_1] = orderlist_1[a];
        counter_1++;
        int i = 1;
        while (1)
        {
            if (a == (g->vexnum - 1))
            {
                break;
            }
            a = a + 1;
            orderlist[counter_1] = orderlist_1[a];
            counter_1++;
            orderlist[counter_1] = orderlist_1[a - 2 * i];
            counter_1++;
            i++;
        }
    }
    else
    {
        int i = 0;
        orderlist[counter_1] = orderlist_1[a];
        counter_1++;
        orderlist[counter_1] = orderlist_1[a - (2 * i + 1)];
        counter_1++;
        i++;
        while (1)
        {
            if (a == (g->vexnum - 1))
            {
                break;
            }
            a = a + 1;
            orderlist[counter_1] = orderlist_1[a];
            counter_1++;
            orderlist[counter_1] = orderlist_1[a - (2 * i + 1)];
            counter_1++;
            i++;
        }
    }
}

/*-------------------------------------------------------color-----------------------------------
-------------------------*/
```

```
void greedcoloring(PGraph g)
{
	int color = 1;
	PANode a;
	for (int i = 0; i < g->vexnum; i++)
	{
		color = 1;
		int coloring_item = orderlist[i];
		a = g->p[coloring_item].first;

		while (a != NULL)
		{
			if (g->p[a->vertex].color == color)
			{
				color++;
				a = g->p[coloring_item].first;
			}
			else
			{
				a = a->next;
			}
		}
		g->p[coloring_item].color = color;
	}
}
int isused_sllo;
void outcolor(PGraph g)
{
	int maxcolor = 0;
	int counter = 0;
	for (int i = 0; i < g->vexnum; i++)
	{
		printf(" VERTEX_NUMBER <%d> COLOR_NUMBER <%d> and
ORIGINAL_NUMBER is <%d>\n", orderlist[i],
g->p[orderlist[i]].color,g->p[orderlist[i]].original_degree);
		counter = counter + g->p[orderlist[i]].original_degree;
		if (g->p[i].color >= maxcolor)
		{
			maxcolor = g->p[i].color;
		}
	}
	printf("total use %d color\n", maxcolor);
	printf("the average original degree is %d\n", (counter/g->vexnum));
	if(isused_sllo == 1){
```

```
        printf("the maximum degree when deleted”  value for the smallest last ordering is %d\n", max_degree);
        printf("the size of terminal clique = %d\n", (record_degree + 1));
        }

}

int main()
{
        srand((unsigned)time(NULL));
        DegreeList l;
        clock_t start_complete,finish_complete;
        double totaltime;
        int v;
        int edge;
        Graph g;
        while (1)
        {
                printf("Please input the Number of vertices. (MAX = 10,000)\n");
                fflush(stdin);
                scanf("%d", &v);
                fflush(stdin);
                if (v > 10000)
                {
                        printf("the value is invalid\n");
                }
                else
                {
                        break;
                }
        }

        while (1)
        {
                printf("Please input the Number of Edges that show conflict between 2 courses. (MAX = 2,000,000)(Complete graph and circle you can input any number)\n");
                fflush(stdin);
                scanf("%d", &edge);
                fflush(stdin);
                if (v > 2000000)
                {
                        printf("the value is invalid\n");
                }
                else
```

```
        {
            break;
        }
    }
    gcolor = (int *)calloc(v, sizeof(int));
    orderlist = (int *)calloc(v, sizeof(int));
    s = (int *)calloc(v, sizeof(int));
    deleteorder = (int *)calloc(v, sizeof(int));
    creatgraph(&g, v);
    int model;
    while (1)
    {
        printf("Please choose 1 way to creat the graph you can enter number G : 1.COMPLETE | 2.CYCLE | 3.RANDOM (with DIST below)\n");
        fflush(stdin);
        scanf("%d", &model);
        fflush(stdin);
        if (model >= 4 || model <= 0)
        {
            printf("the value is invalid\n");
        }
        else
        {
            break;
        }
    }
    int model_2;
    switch(model)
    {
    case 1:
        //start_complete=clock();
        completegraph(&g);
        //finish_complete=clock();
        break;
    case 2:
        //start_complete=clock();
        circle(&g);
        //finish_complete=clock();
        break;
    case 3:
        while (1)
        {

            printf("Which random way you expected: 1. uniform random 2. skew
```

```
distribution 3. normal\n");
                fflush(stdin);
                scanf("%d", &model_2);
                fflush(stdin);
                if (model_2 >= 4 || model_2 <= 0)
                {
                    printf("the value is invalid\n");
                }
                else
                {
                    break;
                }
            }
        }

        switch(model_2)
        {
        case 1:
            //start_complete=clock();
            uniform(&g, edge);
            //finish_complete=clock();
            break;
        case 2:
            //start_complete=clock();
            skew(&g, edge);
            //finish_complete=clock();
            break;
        case 3:
            //start_complete=clock();
            normal(&g, edge);
            //finish_complete=clock();
            break;
        }
        out(&g);
        //totaltime=(double)(finish_complete-start_complete)/CLOCKS_PER_SEC;
        //printf("totaltime_for graph=%fs\n",totaltime);
        int max = maxdegree(&g);
        creatDegList(l, max);
        writeList(&g, l);
        outList(l);
        int model_3;
        while (1)
        {
            printf("Which way to ordering the graph you expected: 1. Smallest Last Ordering
```

```
2. Smallest Original Degree Last 3. Random Ordering\n4. DFS 5. BFS 6.Medium Degree
First\n");
            fflush(stdin);
            scanf("%d", &model_3);
            fflush(stdin);
            if (model_3 >= 7 || model_3 <= 0)
            {
                  printf("the value is invalid\n");
            }
            else
            {
                  break;
            }
      }
      switch(model_3)
      {
      case 1:
            //start_complete=clock();
            SmallestLastOrdering(&g, l);
            //finish_complete=clock();

            isused_sllo = 1;
            break;
      case 2:
            //start_complete=clock();
            SmallestOriginalDegreeLast(l, &g);
            //finish_complete=clock();

            break;
      case 3:
            RandomOrdering(l, &g);
            break;
      case 4:
            DFSTraverse(g);
            break;
      case 5:
            BFSTraverse(g);
            break;
      case 6:
            MediumDegreeFirst(&g,l);
            break;
      }

      start_complete=clock();
```

```
    greedcoloring(&g);
    finish_complete=clock();


    totaltime=(double)(finish_complete-start_complete)/CLOCKS_PER_SEC;
    outcolor(&g);
    printf("totaltime_for graph=%fs\n",totaltime);

    if(model_3 == 1){
    printf("the delete order is:");
    for(int i = 0;i< g.vexnum;i++){
        printf("%d ",deleteorder[i]);
    }
    printf("\n");
    }
    printf("the ordering is :");
    for (int i = 0; i < g.vexnum; i++)
    {
        printf("%d ", orderlist[i]);
    }
    printf("\n");

    printf("finished\n");

    return 1;
}
```