**Algorithms Final Project**

*Graph Coloring Analysis*

**Bryan Putnam**

49235478

CS 7350

**Computer Environment Details**

Operating System: macOS Monterey (version 12.5)

Processor: 2.4 Ghz 8-core Intel Core i9

Memory: 32 GB 2667 MHz DDR4

Graphics: AMD Radeon Pro 5500M 8 GB

IDE: Visual Studio Code

Version: 1.77.3

**Conflict Graph Analysis**

**Complete Graph:**

***Analysis***:

Based on the code written to create the complete graph, I expected running time to be given that I have nested for loops that create a symmetric adjacency list.

Chart, line chart, scatter chart

Description automatically generated***Runtime Tables / Graph:***

***Table

Description automatically generated***

***Runtime Tables / Graph Analysis:***

***Text

Description automatically generated***My previous analysis of my code seems to close to that of the running times table. Based on my code, I should expect to see a running time that quadruples as n doubles, and based on the table it seems that running time almost quadruples while n doubles. The function addEdge() contains code that only assigns values in a negligible time, so that should not be effecting my running time. While my code does not display my running time calculations, I calculated these times using System.currentTimeMillis() in which I stored the time at the beginning and end of completeGraph(), and subtracted the two.

**Cycle Graph**

***Analysis***:

Based on the code written to create the cycle graph, I expected running time to be

given that I have one for loop that only iterates from 0 through n. Within this loop, I also call the addEdge() method, which as mentioned above, adds a negligible amount of time given that .

Chart, scatter chart

Description automatically generatedTable

Description automatically generated***Runtime Tables / Graph:***

***Runtime Tables / Graph Analysis:***

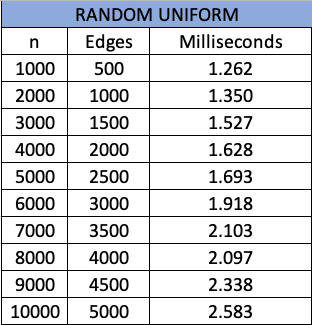
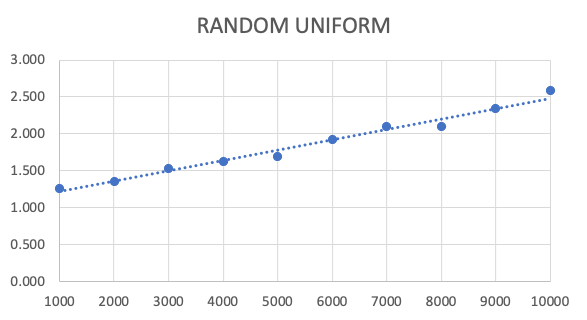
Text

Description automatically generatedMy analysis for the running time of creating a cycle graph seems to be vastly far off my prediction. Based on my code, I expected to see a running time that doubles as n doubles, but according to my timing chart, the data seems to be displaying a logarithmic increase in running time as n increases. I’m not sure why this is because as I explained in my analysis, my code contains a for loop that iterates 0 – 2 and calls another function that is , so my results should match an time, although they clearly do not.

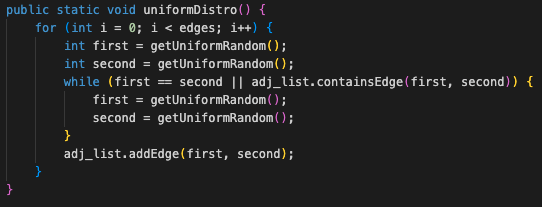
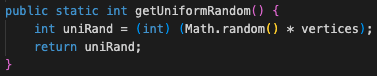
**Random Graph (Uniform, Skewed, Normal)**

***Uniform Analysis:***

Based on my code, I should expect to see the creation of my uniform random have a running time of as I iterate from 0 through n (edges) in a for loop to create two uniform random numbers to add an edge between.

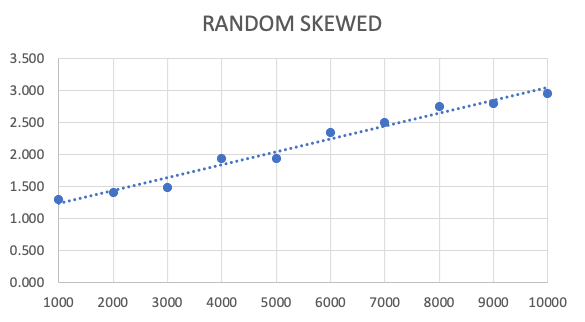
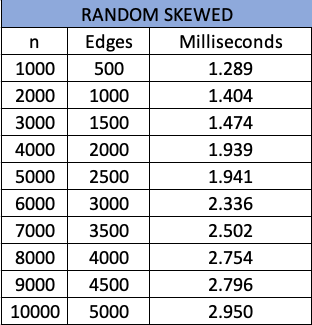
***Runtime Tables / Graph:***

***Runtime Tables / Graph Analysis:***

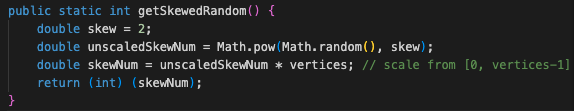
I expected to see a running time of for the creation of the uniform random graph, however, my timing table seems to suggest that my running time was much less. If we compare the doubling of n to the respective running times, we see that the doubling of n does not double the running time. In fact, the resulting time comparison of doubling n shows that our running time is much less than doubled. The table shows that we only just manage to double the running time when multiplying the value of n times 10. Looking at the code on the right, we can see that for every edge that we decide to create, we iterate in the for loop that many times. Inside the for loop we create two uniform random numbers and then go to a while loop to ensure those numbers are not the same. From there, all we do is add an edge using those two uniform random numbers. Below, I have also displayed the code used to create the random uniform number (getUniformRandom()). 

***Skewed Analysis:***

My code for creating a skewed conflict graph was nearly identical to the uniformDistro() code above, with the only difference being a function call to getSkewedRandom() instead of getUniformRandom(). Given this, I expected my running time to be just as it was for uniformDistro().

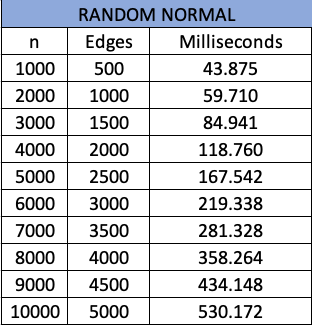
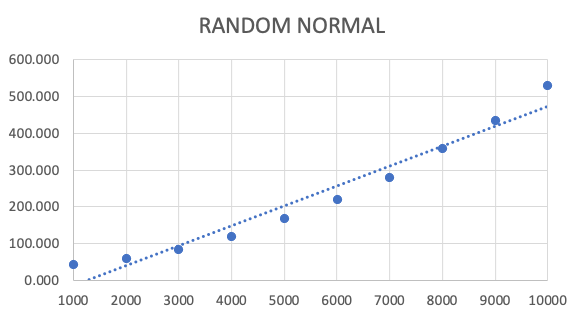
***Runtime Tables / Graph:***

***Runtime Tables / Graph Analysis:***

As mentioned above, my code seems to be written in a way that creates skewed conflict graphs in time. However, just as we got with the random uniform graph, my table seems to tell a different story. When looking at the doubling of n, while I would expect to see the running times doubling, they seem to be much less than double. In fact, like the uniform random graph, we only seem to double our running time of n is multiplied by around 10. Even though the code that create graph is predicted to be , the getSkewedRandom() function seems to only have operations being used.

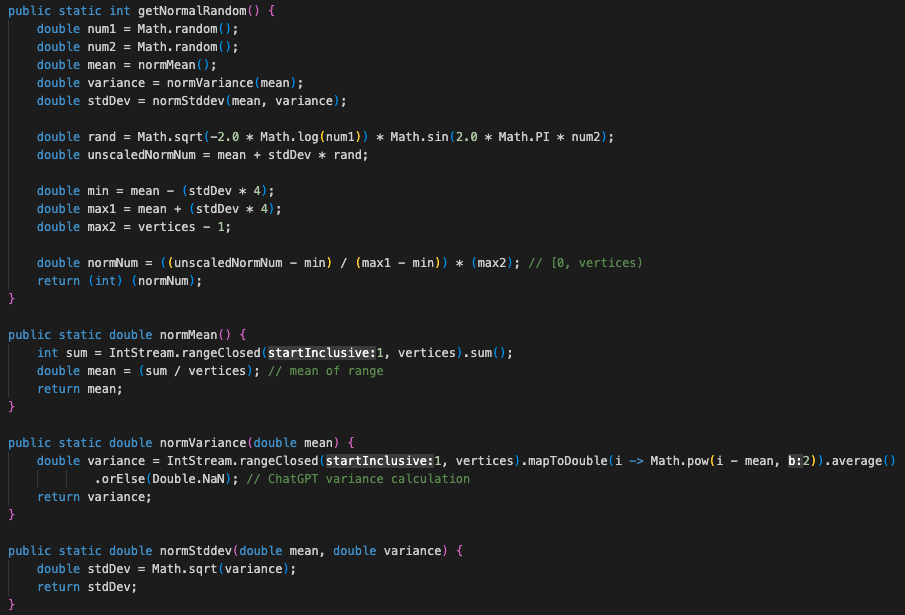
***Normal Analysis:***

My code for creating a normal conflict graph was nearly identical to the uniformDistro() and skewedDistro() code above, with the only difference being a function call to getNormalRandom(). Given this, I expected my running time to be just as it was for uniformDistro() and skewedDistro().

***Runtime Tables / Graphs:***

***Runtime Tables / Graph Analysis:***

Out of the other random graphs, random normal seemed to be the one that fit the runtime predictions to a closer degree. I expected to see , and for the most part, that’s what the table tells as well. Looking at the table we can see that the doubling of n roughly translates to the doubling of running time, with the exception of going from n vales 1000 to 2000. The data that I got for this seemed to initially be inconsistent, not only with my predicted running time, but even with its own linearity. I ran these timing tests multiple times but was never able to get a very clear linear line (with the exception of the trendline in the graph). This could be because many functions are called in the making of this graph. Out of all the other graphs, this graph needed a lot of mathematical manipulation to be considered a normal distribution. Some of those manipulations came in scaling the random numbers, and using the mean, and standard deviation of those numbers. The code for those functions are provided below. The code that references chatGPT was later found to be an equation that could be found on Stack Overflow.



**Vertex Ordering Analysis**

**Smallest Last Ordering**

***Description:***

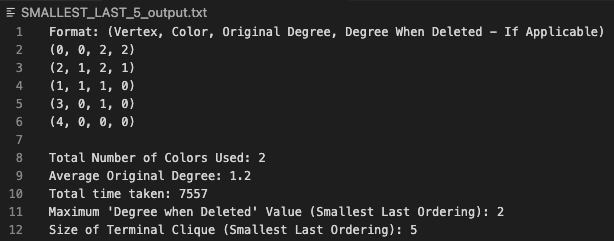
The purpose of the Smallest Last Vertex Ordering algorithm is to computer the degree of all vertices and then order the vertices by decreasing degree (start with highest degree vertex). To do this, the algorithm will find the degree of all vertices in the graph, and then start with the vertex that has the highest degree. Once this vertex is found, it is deleted from the graph, added to an order array, and the degrees of all over vertices in the graph are updated respectively. This process will repeat until every vertex has been deleted from the graph and subsequently added to the ordered array. The resulting array will have vertices ordered from largest degree to least degree (hence smallest LAST).

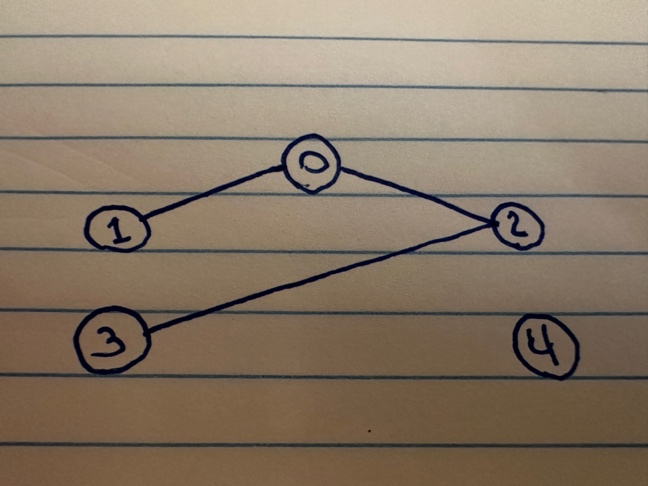
***Implementation:***

In my Smallest Last Vertex Ordering algorithm, I initialize an empty array named ‘order’ of size ‘vertices’ (n) that will store the sorted vertices. I then initialize a new array of size ‘vertices’ (n) named ‘degreeOnDelete’ which will store the degree of a vertex upon deletion. Next, I make variables ‘cliqueSize’ and ‘numNotDeleted’ that are both initialized to ‘vertices’ (being the number of vertices in the graph). I then iterate through a for loop that checks to see if the number of non-deleted vertices is less than the clique size and if the number of edges is equal to the number of edges you would expect for a complete graph. If that statement is true, I update the clique size to be equal to the number of vertices not deleted.

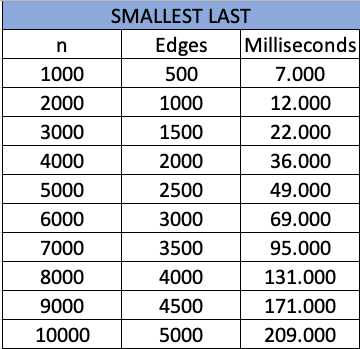
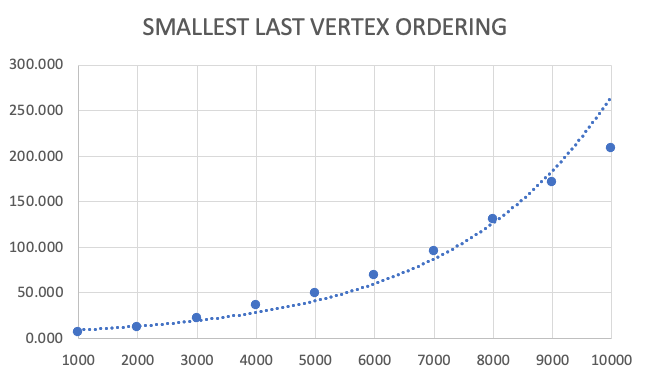
Then I initialize the ‘maxVertex’ and ‘maxDegree’ to -1 so that I have a baseline for error checking (I chose -1 because its possible for maxVertex and maxDegree to be 0). I then iterate through a for loop and check to see if the value at index j of the degree list is not equal to -1 and that the value at index j of the degree list is greater than the max degree. If this statement is true, I assign maxDegree to be the value at index j of the degree list and assign maxVertex to be j. To keep track of the max degree when deleted, I used an if statement that says if the max degree is greater than the max degree when deleted, then maxDegreeWhenDeleted is assigned to the value of maxDegree. Finally, I add i to order with the value of maxVertex, subtract the number of edges, number of vertices not deleted, and assign values to specify the degree of the vertex upon.

***Smallest Last Vertex Ordering Walkthrough (On Graph)***

Given the output file below, I’ve drawn out a graph that follows the output files specification for what vertices are connected and have what degree.



Upon creation, the graph will look like the depiction on the right. Vertices 0 and 2 have degree 2, vertex 3 has degree 1, and vertex 4 has degree 0. From this point we will start at vertex 0 and delete it, add it to order array, and take away the edges that connect to vertices 1 and 2. Upon deletion, vertex 0 is of degree 2. Next, we move to vertex 2 which is now of degree 1, delete vertex 2, add to order array, and delete its edge to vertex 3. Vertex 2 has degree 1 upon deletion. Now there are no more edges in the graph so we can now do vertices 1, 3, and 4. We deleted them from the graph, add them to order array, and there are no edges to delete. Vertices 1, 3, and 4 were of degree 0 upon deletion.

***Runtime Tables / Graph:***

***Runtime Tables / Graph Analysis:***

Based on my code that will be shown below. I was expecting to see quadratic running time due to the use of nested for loops in my ordering function. While there were many other operations that were either or , these would be negligible compared to . It is safe the say that, for the most part, for every doubling of n, the running time roughly quadruples. The only significant exception is going from n=1000 to n=2000 in which the running time less than doubles. However, when we look at n=4000 to n=8000 and n=5000 to n=10000, the running time is nearly exactly quadrupled when the n value is doubled. As a result of this, we have a clean graph showing increasing quadratic time in comparison to n, with an exponential trendline.

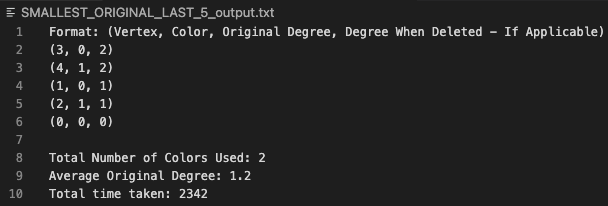
**Smallest Original Degree Ordering**

***Description:***

Smallest Original Degree Ordering is similar to Smallest Last Vertex Ordering in that the ordering of the vertices in the ordered array has the vertices with the highest degrees coming first in the list and the vertices with the least degrees coming at the end of the list. The only difference is that while Smallest Last Vertex Ordering uses the original degree of the vertices before vertex deletion, whereas the degree of each vertex is re-calculated in Smallest Last Vertex Ordering.

***Implementation:***

For my implementation, I created a new array of size ‘vertices’ (number of vertices) with the variable name ‘order’, and then I made a copy of my degree list and named it ‘degreeListCopy’.

Next, I step into a for loop that iterates from 0 through ‘vertices’ and initializes ‘max’ and ‘maxVertex’ to -1. Then I step into another for loop that is nested within the first, and this loop also iterates from 0 through ‘vertices’. In this for loop I have a conditional that will check to see if that value in ‘degreeListCopy’ at index j is not equal to -1 and that the value at index j of ‘degreeListCopy’ is greater than ‘max’. If this condition is true, I will assign the value of ‘degreeListCopy’ at index j to ‘max’ and set ‘maxVertex’ equal to j. Then I update order at index i with the value of ‘maxVertex’ and I make the value of degreeListCopy at index ‘maxVertex’ to -1 to symbolize that the vertex at that index is deleted. The below image is the result of running this algorithm with 5 vertices and 3 edges.

**Uniform Random Ordering**

***Description:***

Uniform Random Ordering is an ordering algorithm that orders vertices based on random numbers. First, a random number is given to each vertex in the graph, then the vertices are ordered in ascending order based off of their given random numbers. This is a very simple algorithm and can easily be done in time.

***Implementation:***

To create the Uniform Random Ordering algorithm, I first started by creating an integer array ‘order’ of length ‘vertices’. I then used Arrays.fill() to fill that empty array with -1’s so I could later check to see which indices had not been given a random number. I then created a for loop that iterates from 0 through ‘vertices’ and called the getUniformRandom() function and assigned its value to ‘index’. Then I used a while loop to assign getUniformRandom() to index while the value at index ‘index’ does not equal 1. Finally, I added i to the order array at index ‘index’. The following image is the result of running this algorithm with 5 vertices and 3 edges.

Text

Description automatically generated

**Largest Last Vertex Ordering**

***Description:***

Largest Last Vertex Ordering is the reverse of Smallest Last Vertex Ordering with the same basic principles. All the original vertex degrees are calculated, but instead of starting with the largest vertex degree, you start with the smallest vertex degree so that the last vertex in the list has the largest degree. To do this, you check for the smallest degree, delete edges if possible, recalculate all vertex degrees, and repeat the process until you have deleted and stored the last vertex in the order array.

***Implementation:***

The first steps I made to complete the Largest Last Vertex Ordering was to create an array integer array ‘order’ of length ‘vertices’, and then create a copy of ‘degreeList’ named ‘degreeListCopy’. From here, I created a for loop that iterates from 0 through ‘vertices’. Inside this for loop, I initialized ‘minDegree’ to the largest possible integer value, and ‘minIndex’ to -1. I did this so that it was impossible for me to start out with a ‘minDegree’ that was smaller than any possible integer degree value, and so that ‘minIndex’ was smaller than any possible index before entering another for loop. Then I created a nested for loop and conditional so that if the value at index j of ‘degreeListCopy’ was not equal to -1 and the same value was less than ‘minDegree’, the value at index j of ‘degreeListCopy’ was assigned to ‘minDegree’ and j was assigned to ‘minIndex’. After the inner for loop, I assign ‘minIndex’ to index i of order array and then delete the necessary degrees. The following image is the result of running this algorithm with 5 vertices and 3 edges.

Text

Description automatically generated

**Largest Last Original Degree Ordering**

***Description:***

The Largest Last Original Degree Ordering algorithm is the same as Largest Last Vertex Ordering with the difference being the original degrees of the vertices are kept even after deletion. This the exact same technique that is used in Smallest Original Degree Ordering, just a reverse order. You start by calculating the degree of all vertices and then delete and subsequently add to the order array, the smallest degree vertex. Instead of recalculating the degree of each vertex after a deletion, each vertex keeps its original degree and you repeat the process until the last deleted vertex is the one of the largest degree.

***Implementation:***

The implementation for this algorithm uses the exact same code as Largest Last Ordering Vertex with the exception that this algorithm does not call the ‘deleteDegrees()’ function. This ensures that when iterating back through the for loop ‘vertices’ number of times, no degrees are deleted and every vertex keeps its original degree. At the end of the for loop, minIndex is assigned to index i of the order array and the final result will have the vertices ordered in increasing order with the final vertex having the largest original degree. For the sake of redundancy, because the code is the same with the exception of one function call, I do not feel the need to reiterate the entire function. (For more information reference “Implementation” for Largest Last Vertex Ordering. Below is the result of running this function with 5 vertices and 3 edges.

Text

Description automatically generated

**Outside to Inside Ordering**

***Description:***

This algorithm is quite unique and is nothing like what I’ve done before so far. The goal of this algorithm is to order the vertices from the smallest and largest towards the middle of the array by assigning the low values towards the front of the array and the high values towards the end of the array going towards the middle.

***Implementation:***

To create this unique algorithm, I started by initializing ‘low’, ‘high’, and ‘index’ to the values 0, ‘vertices’ -1, and 0, respectively. Then I created a while loop in which the condition is if ‘low’ is less than or equal to ‘high’, then you enter a conditional if statement. If ‘index’ modulo 2 is equal to 0, then assign the value of low to index ‘index’ if order and then increment ‘low’. If not, assign the value of high to index ‘index’ of order. After this process increment index and continue until the while condition is not met. Below I have included the result of running this algorithm with 5 vertices and 3 edges.

Text

Description automatically generated

**Coloring Algorithm Analysis**

***Description:***

Coloring algorithms are used to find the minimum number of colors needed to color a graph without two adjacent vertices sharing the same color. To create my coloring algorithm, I created a function named color() that is called after my ordering functions. My ordering functions use global variables so it is not necessary to pass any variables into the color method as each used global variable in color() will have the updated values from the desired ordering algorithm. I start by creating an integer array and a Boolean array named ‘colors’ and ‘availableColors’ respectively. I then fill the ‘color’ array with -1’s and the ‘availableColors’ array with true values. Next, I enter a for each loop that states, for each i in ‘order’, ‘destinationNode’ is equal to i in the node list. While the destination node is not equal to null, if ‘colors’ at index vertex is not equal to -1, then a color has been assigned. I will then change the Boolean value at that vertex index in ‘availableColors’ to false because it is no longer available. Then I set the new destination node to be equal to the next destination node pointer. Next, I need to find the smallest available color, so I initialize variable ‘currColor’ to 0 and while index ‘currColor’ in ‘availableColors’ is not equal to 0 and ‘currColor’ is less than ‘vertices’, increment currColor to use another color. Next, I use colors[i] = currColor to assign the new color to ‘colors’. Finally, I set all values in ‘availableColors’ back to true for the next coloring iteration.

For reference, I will put an image of the color() function on the next page.

Text

Description automatically generated

**Vertex Ordering Capabilities**

In the following pages, I will be showing some charts I made that display the number of colors needed for all graphs will distributions. I ran multiples tests for both sparse and dense graph representations. For sparse graphs, I made edges equal to ¼ vertices, and for dense graphs, I made edges equal to ¾ vertices.

I used chart coloring to depict which graphs/distributions did the best, neutral, and worst. For each coloring number I did the same thing with the least number of colors being good, the most being bad, and the middle being neutral. To decide which orderings did better as a whole, I did a rough average of how good the colorings did for each test and gave the ordering either a good, neutral, or bad color score.

Table

Description automatically generatedTable

Description automatically generatedTable

Description automatically generatedTable

Description automatically generated

Table

Description automatically generatedBased on my tests, it seems that SLO and SOD did the best overall compared to any other ordering algorithm. It is also notable to mention that across all ordering methods, the denser the graph became, the more colors were needed to solve. The exception to this is the complete graph, where every vertex is connected to all other vertex, meaning that the number of colors necessary can never be less than the number of vertices in the graph.

Table

Description automatically generated**Smallest Last Vertex Ordering**

***Random Graph, Uniform Distribution, v=10, e=9:***

Maximum degree when deleted: 5

Size of terminal clique: 1

Upper Bound: 5

Lower Bound: 4

Table

Description automatically generated

***Random Graph, Skewed Distribution, v=10, e=5***

Maximum degree when deleted: 2

Size of terminal clique: 10

Upper Bound: 10

Lower Bound: 9

***Table

Description automatically generated***

***Random Graph, Normal Distribution, v=10, e=7***

Maximum degree when deleted: 4

Size of terminal clique: 10

Upper Bound: 10

Lower Bound: 9

Table

Description automatically generated

***Cycle Graph, v=10***

Maximum degree when deleted: 2

Size of terminal clique: 3

Upper Bound: 3

Lower Bound: 2

Table

Description automatically generated

***Complete Graph, v=10***

Maximum degree when deleted: 9

Size of terminal clique: 10

Upper Bound: 10

Lower Bound: 8

***Bounds***

I can calculate the upper and lower bounds for the number of colors deleted by using the following formulas:

Upper Bound = max(max degree, terminal clique)

Lower Bound = max(max degree -1, terminal clique -1, 1)

The terminal clique is a maximal clique of a graph in which every vertex outside of the clique is adjacent to at least one vertex in the clique. Because of this, all vertices within the terminal clique must be colored differently as well as adjacent vertices outside the clique. Therefore, the lower bound must at least be the size of the terminal clique, and the upper bound can be no greater than the max degree plus one.