**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:***

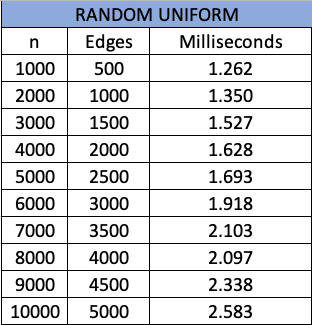
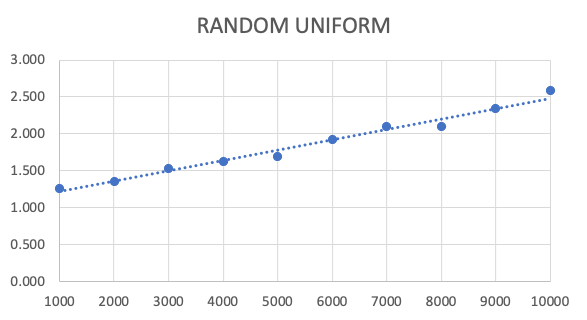
Text

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.

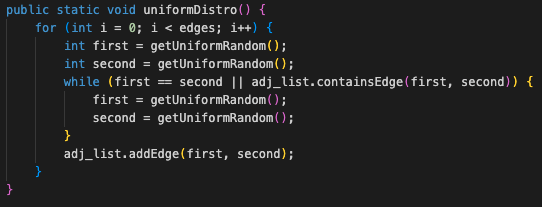
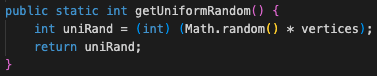
**Random Graph (Uniform, Skewed, Normal)**

***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.

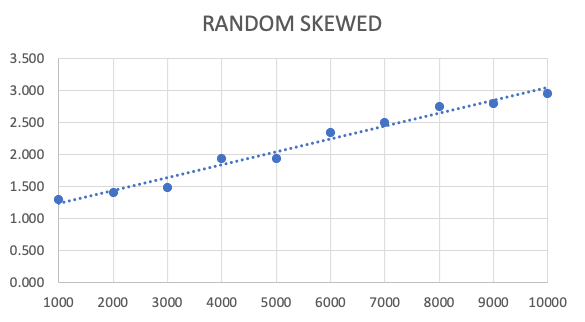
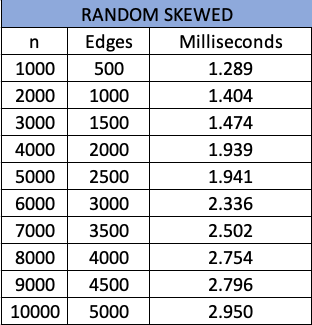
***Runtime Tables / Graph:***

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

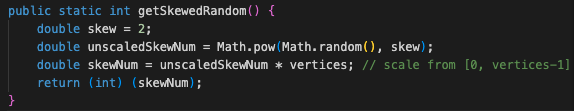
I expected to see a running time of for the creation of the uniform random graph, however, my timing table seems to suggest that my running time was much less. If we compare the doubling of n to the respective running times, we see that the doubling of n does not double the running time. In fact, the 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().

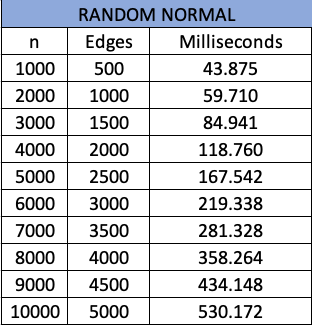
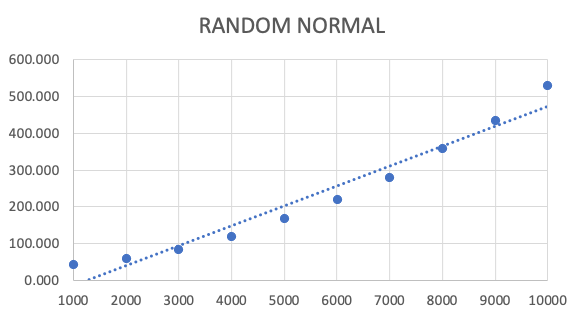
***Runtime Tables / Graph:***

***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.

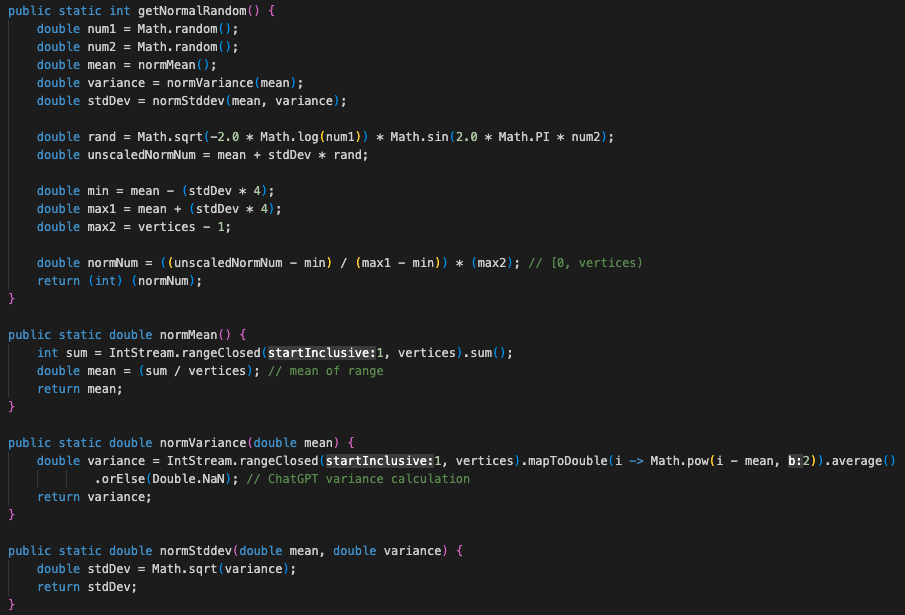
***Normal Analysis:***

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**

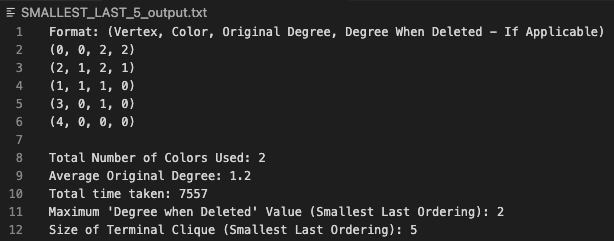
***Description:***

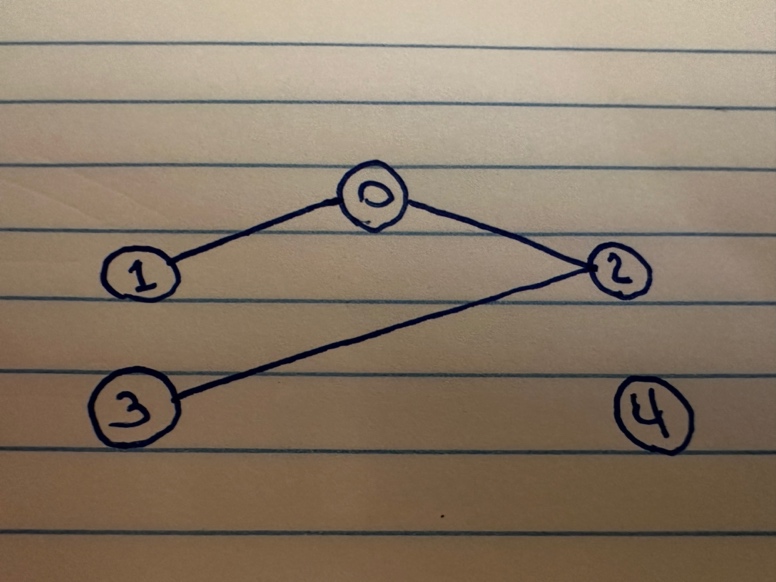
The purpose of the Smallest Last Vertex Ordering algorithm is to computer the degree of all vertices and then order the vertices by decreasing degree (start with highest degree vertex). To do this, the algorithm will find the degree of all vertices in the graph, and then start with the vertex that has 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).

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.

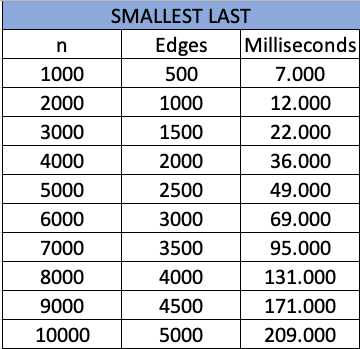
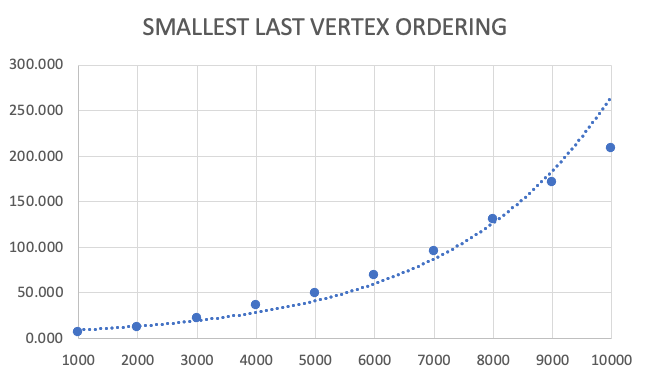
Then I initialize the ‘maxVertex’ and ‘maxDegree’ to -1 so that I have a baseline for error checking (I chose -1 because its possible for maxVertex and maxDegree to be 0). I then iterate through a for loop and check to see if the value at index j of the degree list is not equal to -1 and that the value 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**