**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:***

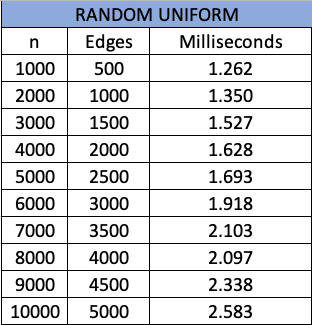
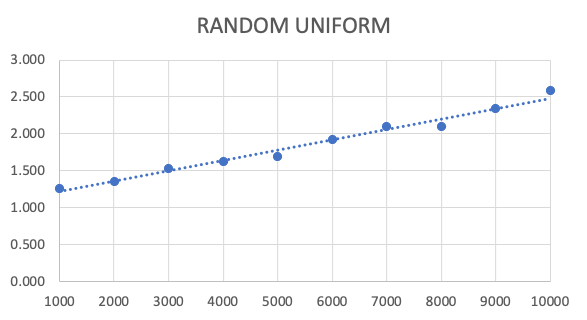
Text

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.

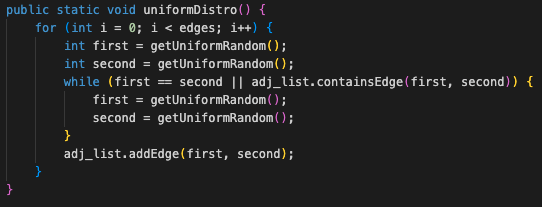
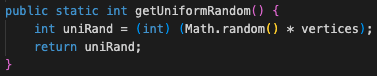
**Random Graph (Uniform, Skewed, Normal)**

***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.

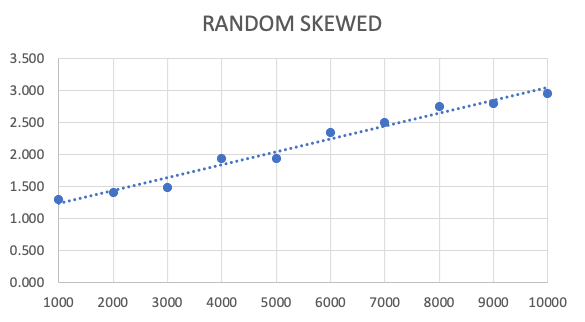
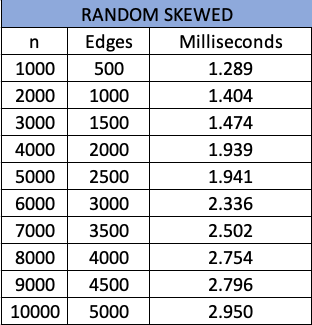
***Runtime Tables / Graph:***

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

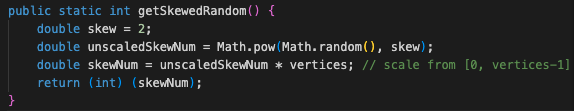
I expected to see a running time of for the creation of the uniform random graph, however, my timing table seems to suggest that my running time was much less. If we compare the doubling of n to the respective running times, we see that the doubling of n does not double the running time. In fact, the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

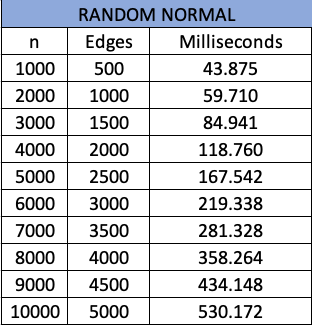
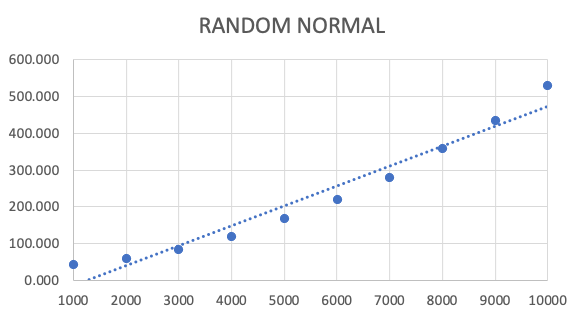
***Runtime Tables / Graph:***

***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.

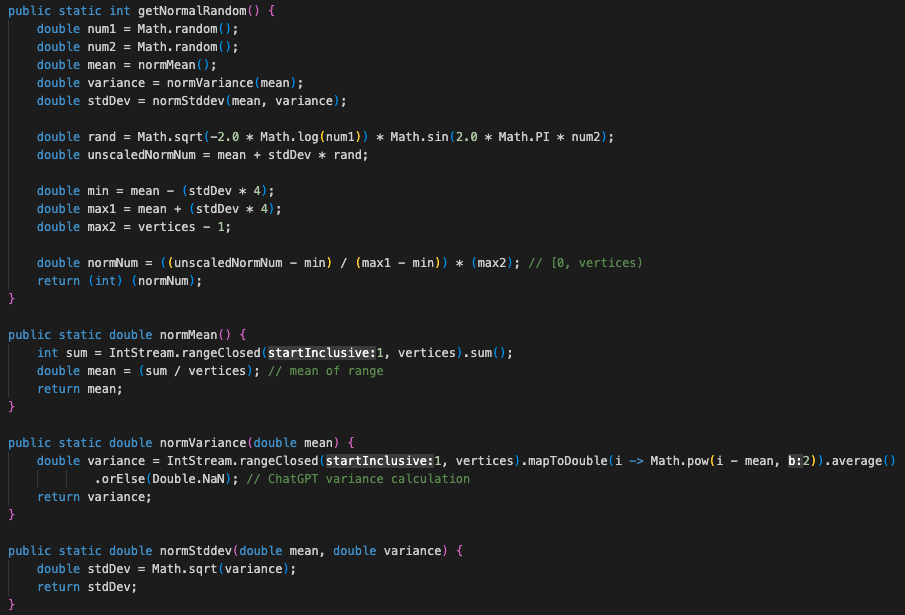
***Normal Analysis:***

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**