SOUTHERN METHODIST UNIVERSITY

Algorithm Engineering Final Project

Spring 2023

By Cameron Rosenberger

Abstract

This study examines the runtime complexity of various graph generation methods combined with four different coloring algorithms. Graph generation will be examined first, and then smallest last vertex ordering, coloring algorithm, and finally the ordering capabilities will be examined.

Table of Contents

List of Tables and Figures

Introduction

To complete this project, I used an Apple MacBook Air M1 running C++ 14 through the IDE Clion. The source code makes use of two classes, Main.cpp and Graph.cpp. Graph.cpp houses all the relevant information and computational methods for this project including graph creation, smallest last vertex ordering, greatest last vertex ordering, and the coloring algorithm. There are five types of graphs able to be generated: cycle, complete, random with uniform distribution, random with skewed lower distribution, and random with skewed higher distribution. The random skewed lower graph refers to the distribution given in the instructions where random numbered vertices are more likely to have an edge than higher vertices. The same goes for the random skewed higher graph, where the higher numbered vertices are more likely to have an edge than the lower vertices. There are four vertex orderings implemented: smallest last vertex, smallest original degree, random, and greatest last vertex ordering. Smallest last vertex ordering and smallest original degree orderings are based off of the information provided in the instructions. The random ordering inserts vertices into a stack in random order. The greatest last vertex ordering is the opposite of smallest last vertex ordering, but there is no code reuse. I could just take the output from smallest last vertex ordering and reverse it, but that would not give accurate results for timing the orderings, so I implemented an entirely separate method that is very similar to smallest last vertex ordering, mainly with the equality tests being flipped from less than to greater than. There is one coloring algorithm in place. This coloring algorithm is modified from source code given by geeksforgeeks. [1]

Conflict Graphs

There are five graph types to be analyzed: cycle, complete, random uniform, random skewed low, and random skewed high.

Cycle

The first graph to analyze is the cycle graph. This method connects two vertices together and places an edge between a->b and then b->a. This method results in vertex edge having two edges no matter the input size. This relationship is graphed below.

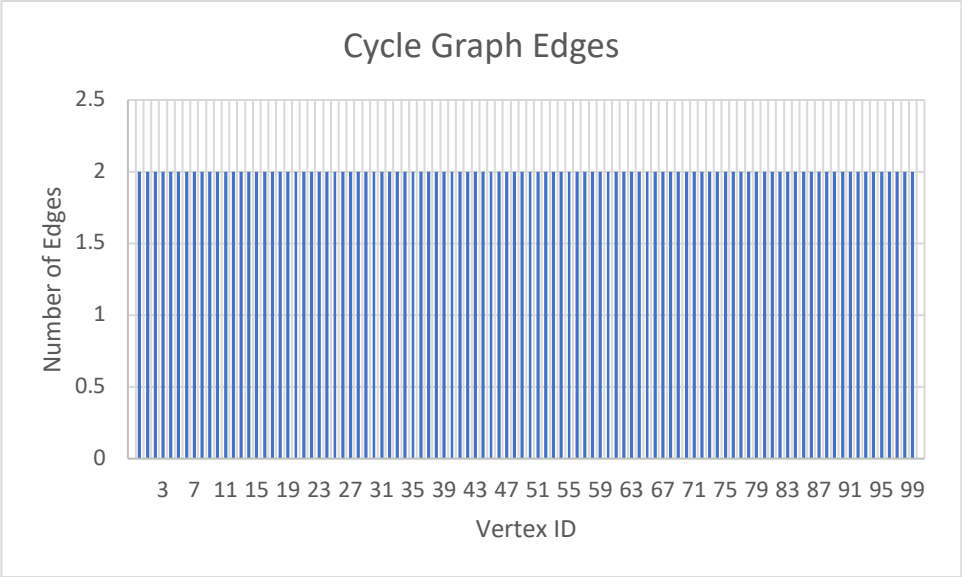


Figure 1 Number of Edges vs Vertex for Cycle Graph

As seen in the graph, each vertex has 2 edges. The following runtimes were reported for the method.

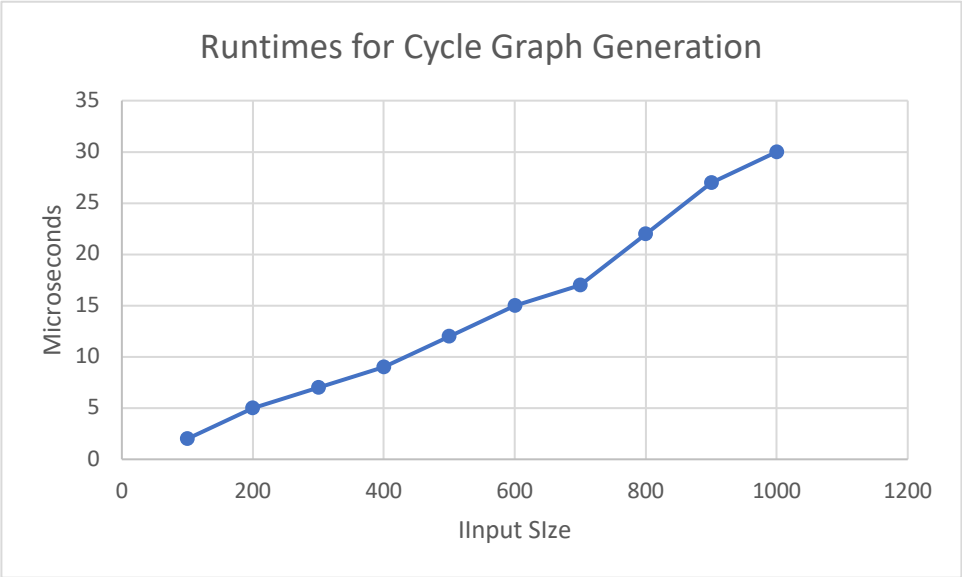


Figure 2 Runtimes for Cycle Graph Generation

This graph is as expected. The line rises linearly across all points. When the input size doubles, the time approximately doubles as seen from input sizes 200 and 400. The same is true for input pairs 100 and 200, 200 and 400, and 400 and 800. Each time the input size doubles, the runtime approximately doubles. It can be concluded that this method has an asymptotic runtime of O(V).

Complete

The complete graph generation method runs in two for loops. The outer loop goes from 0 to V - 1 and the inner loop goes from 0 to V-2. This is done because for each vertex, it needs an edge to every other vertex. We can assume here that the runtime is in O(V^2), but that will be confirmed by graphing the times recorded.

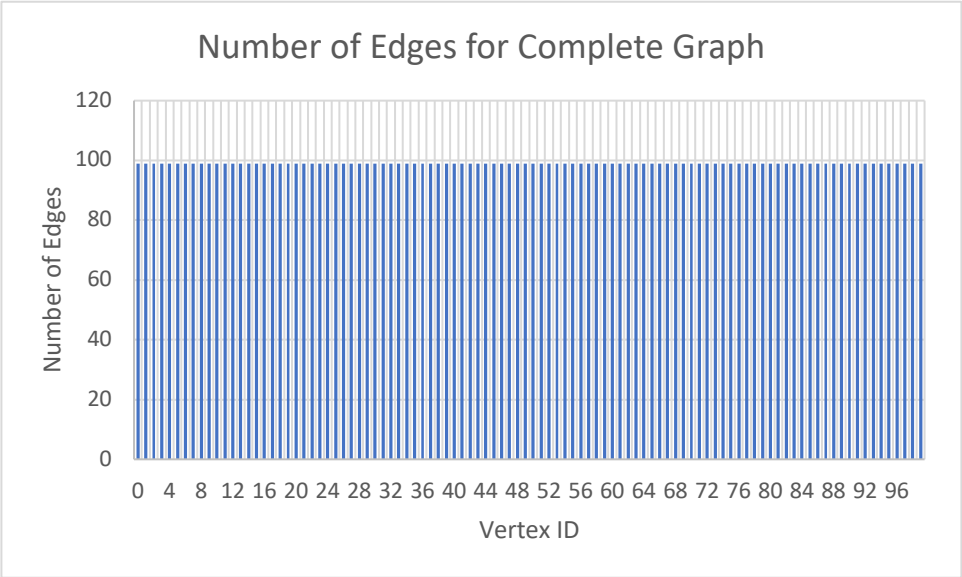


Figure 3 Number of Edges per Vertex for Complete Graph

As seen in the graph, no matter the vertex number, each vertex has 99 edges for an input size of 100.

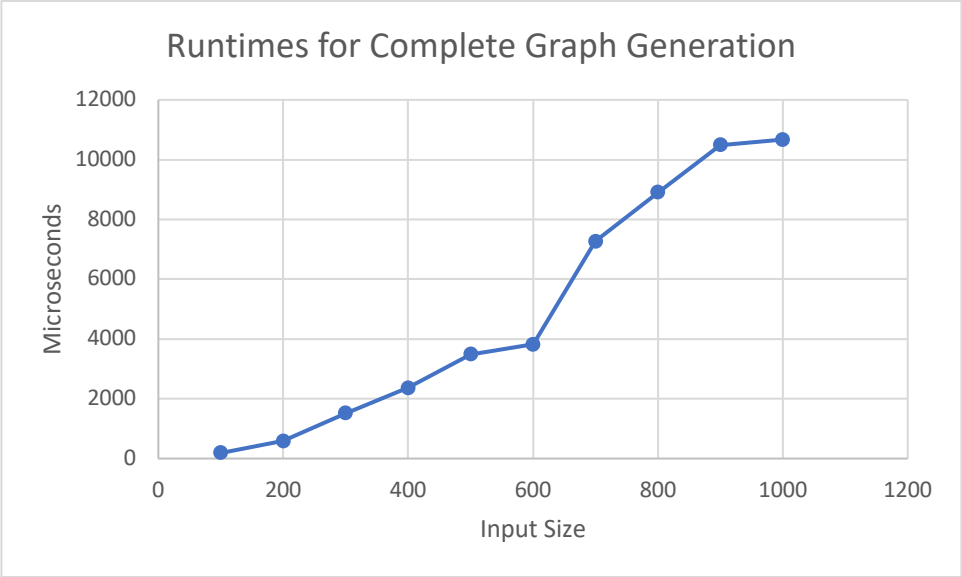


Figure 4 Runtimes for Complete Graph Generation

As seen from the plotted points, when input size doubles, the time taken approximately quadruples. When input equals 200, the associated time is 581ms. When input equals 400, the time is 2362ms. 2362/581 is approximately 4.07. This confirms the previous estimate that the generation of a complete graph has a runtime of O(V^2).

Random Uniform

The random uniform graph generation method runs in a for loop from 0 to E-1 and picks two random numbers between 0 and V-1 and if they are not already connected, then an edge is placed. This runs until E edges have been placed. From this method, we should see a flat distribution of edges.

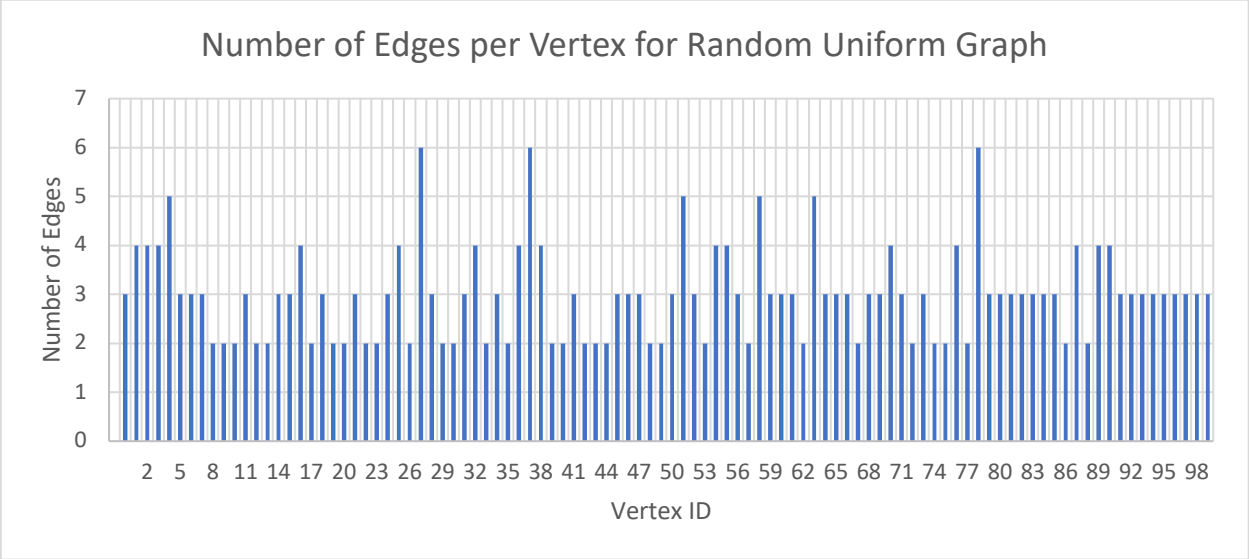


Figure 5 Number of Edges per Vertex for Uniform Graph

As we can see, no portions of the graph have significantly more edges than another. There are a few outliers, but those are due to the uniform distribution random number generator used.

Random is very difficult to produce, and even RNGs have a bit of pattern to them. We should expect this method to run approximately linearly because of the single for loop.

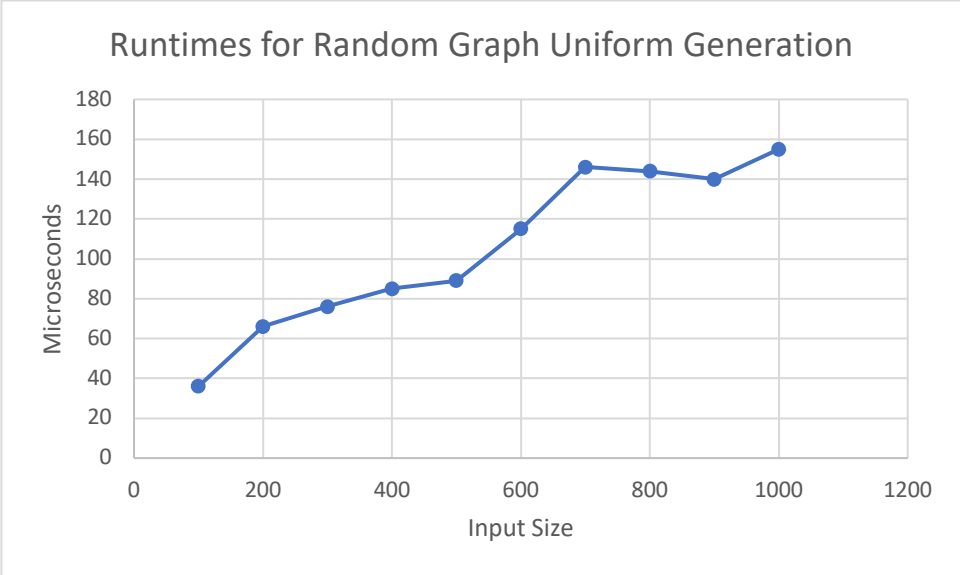


Figure 6 Runtimes for Random Graph Uniform Generation

In this procedure for each input, the number of edges requested is 1.5*V. From the graph, we can see that when input size doubles, the time does not. It increases with respect to input size, but in a linear fashion. From an input size of 200 to 400 we see a jump in time from 66ms to 85ms. This supports the assertion that the generation of a random graph as implemented here has a runtime complexity of O(E).

Random Graph Skewed Low

The random uniform graph generation method runs almost identically top the uniform generation. The difference is that the random numbers generated are not from 0 to V-1, but rather 0- to .25*V so that the lowest 25% of vertices have more edges placed on them. This runs

until E edges have been placed. From this method, we should see a two level distribution of edges, where the lower numbered vertices have more edges.

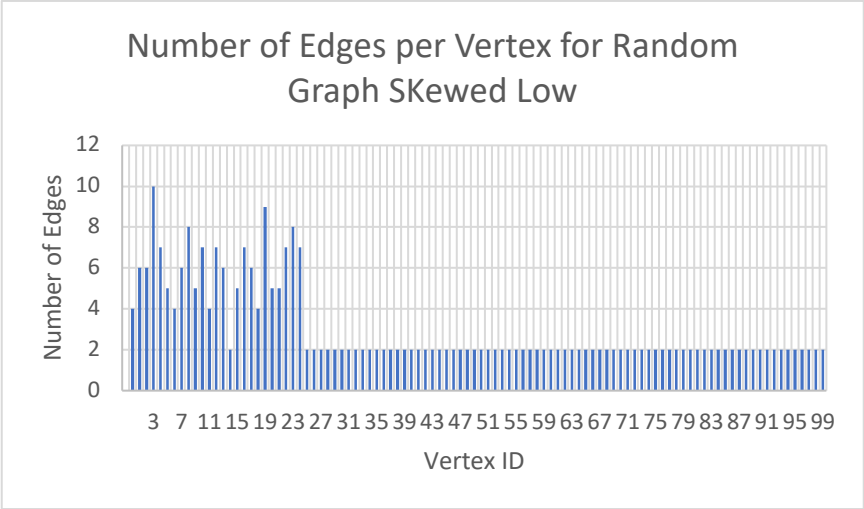


Figure 7 Number of Edges per Vertex for Random Skewed Low Graph

As seen from the graph, it is the case that the lower numbered edges have more vertices.

The runtime of this graph generation should be similar to the uniform graph generation runtime at O(E).

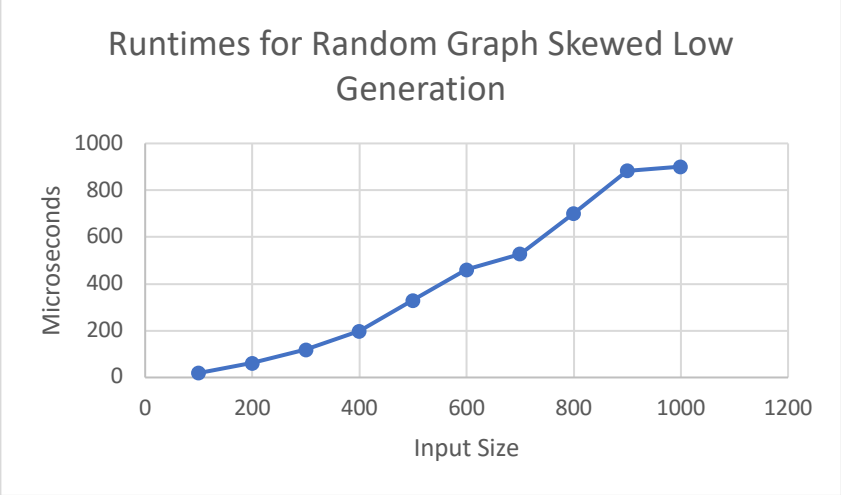


Figure 8 Runtimes for Random Graph Skewed Low Generation

As expected, the times appear to follow a similar path to that in the uniform generation graph. The runtime here is concluded as O(E).

Random Graph Skewed High

As with the random graph skewed low, this method is almost identical. But instead of choosing a random number between 0-25%V, it chooses two random numbers between 75%V and V for the vertex to place an edge to. The number of edges per vertex should be flipped form that of the random skewed low graph.

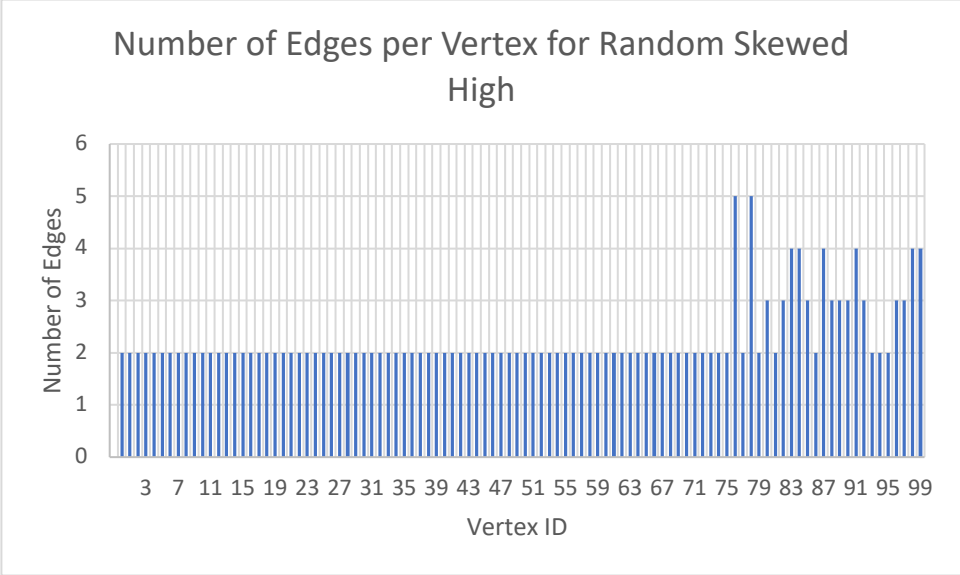


Figure 9 Number of Edges per Vertex Random Skewed High

The runtime complexity should be the same as the uniform and the skewed low graph generation at O(E).

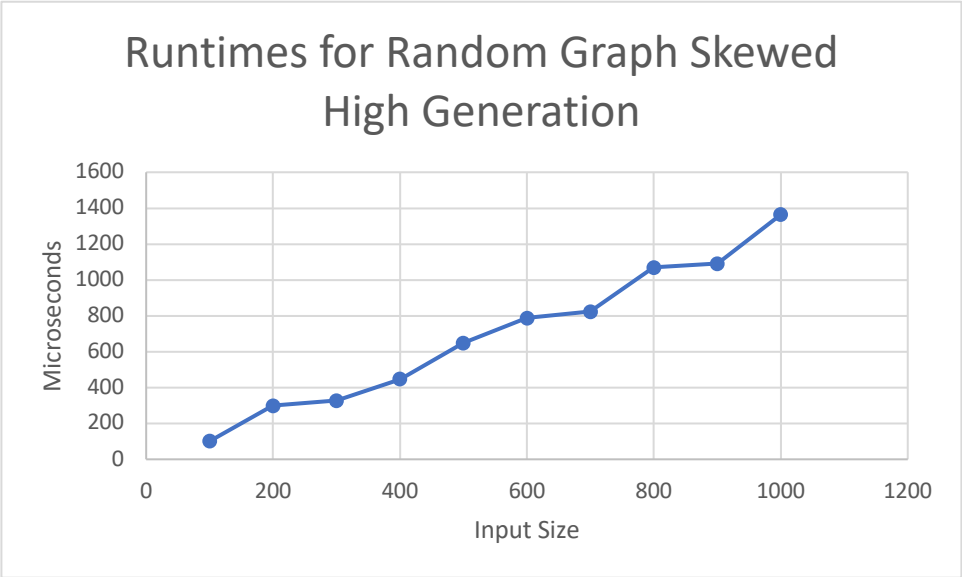


Figure 10 Runtimes for Random Graph Skewed High Generation

The assertion of linear growth in the runtime is confirmed by this graph. As we can see, when input size doubles, the time taken does not double. The growth is consistently less than double the input increase, so it can be concluded that this runtime complexity is O(E).

Vertex Orderings

My implementation of smallest last vertex ordering strives to be as close to the demonstration from class. I make use of an adjacency list, array of linked lists, and a degree list. The degree list is an array of linked lists where the index of the array is the degree size. For example, my degree container is called 'deg', so deg[1] returns all vertices with 1 edge to/from them. The ordering runs in a for loop starting at 0 and ending at V, the number of vertices. The algorithm selects the degree list, deg, at index i. If there are any vertices of that degree, it selects the first vertex. It sends the vertex number as well as the degree, i, to method deleteVertex(). This

method selects the adjacency list, adj, at the vertex number given. For example, if vertex 0 is selected as the first vertex from the smallest last vertex ordering, then deleteVertex() would be called with '0' as the parameter. deleteVertex would then select adj[0] to get the edges connected to 0 and then call adjustGraph(i,0), where i is the connected vertex to 0. adjustGraph() then removes 0 from the given vertex I and counts the remaining vertices and returns that number. This number is then used in deleteVertex() to move the edge connected to 0 from wherever it is on the degree list to the length provided by adjustGraph(), that being 1 less than its current degree. Once the edges connected to 0 are adjusted on the degrees list and the adjacency list, 0 is removed from both the adjacency list and the degree list and pushed into a stack. smallestLast() then subtracts 1 from i because often the deletion of a vertex results in a vertex having a degree less than the current lowest degree. The process then repeats until i==V, ensuring all vertices are in the stack. The only place of uncertainty in this algorithm is in smallestLast() where the for loop runs until i==V. The last removed vertex will have a size 0, so the loop will make V-1 unnecessary comparisons. This implementation ideally runs in O(|E| + |V|) time with E being number of edges and V being number of vertices. Below is a graph of the running times recorded from timing just the smallest last vertex ordering on a complete graph.

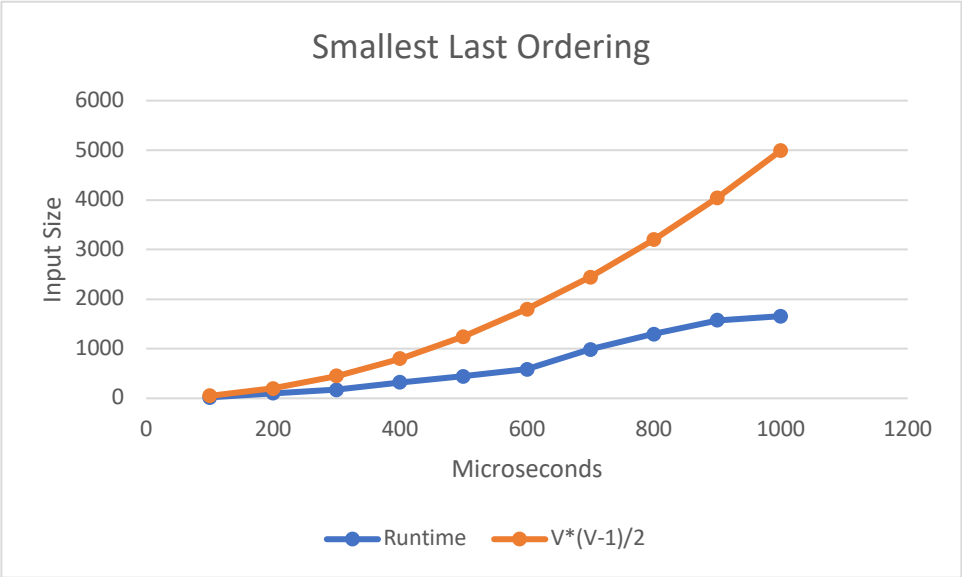


Figure 11 Smallest Last Vertex Ordering Time vs f(x) = V+E

This graph shows two lines, the recorded times from smallest last vertex ordering, and the function f(x) = V+E. This function is used as a benchmark for the runtime because it is the expected runtime for the smallest last vertex ordering. As we can see, the lines are very similar and appear to rise at the same rate. If my implementation were to run in Θ(V+E), we would expect the following scenario similar to a homework/test question we have used. "A process takes 25ms to process an input size of 100, how long would it take to process an input size of 200 with a complexity of Θ(V+E)." For complete graphs, E = V*(V-1)/2. We would expect the runtime for an input of 200 to be 25* {200+(200*199/2)}/{100+(100*99/2)} which equals about 99ms. The recorded time for my implementation is 100ms. The same comparison holds for all of the plotted points. For each timing, it is important to remember the units used are microseconds, so when the function is timed, there is a good bit of variability in the smaller inputs. The points I plotted are the lower bound of three times recorded at each input size. Below is the sample of datapoints used to make the above graph.

Table 1 Points for Smallest Last Vertex Ordering Graph

100	20	49
200	100	199
300	175	448
400	319	798
500	444	1247
600	585	1797
700	983	2446
800	1293	3196
900	1571	4045
1000	1656	4995

Coloring Algorithm

The coloring algorithm used works by taking in a stack of vertices. It assigns a color to the first vertex in the stack, then for the remaining vertices it checks to see if the lowest color number is available to assign to this vertex, if not it checks the next color number and so on until a number is assigned. The runtime for this algorithm should be approximately O(V^2+E) because it runs a for loop inside a for loop. As previously mentioned, my implementation is a modified version of the algorithm found online. [1]

Vertex Ordering Capabilities

The number of colors used remains the same throughout each ordering method on the same graph. The graph type is what changes the number of colors required.

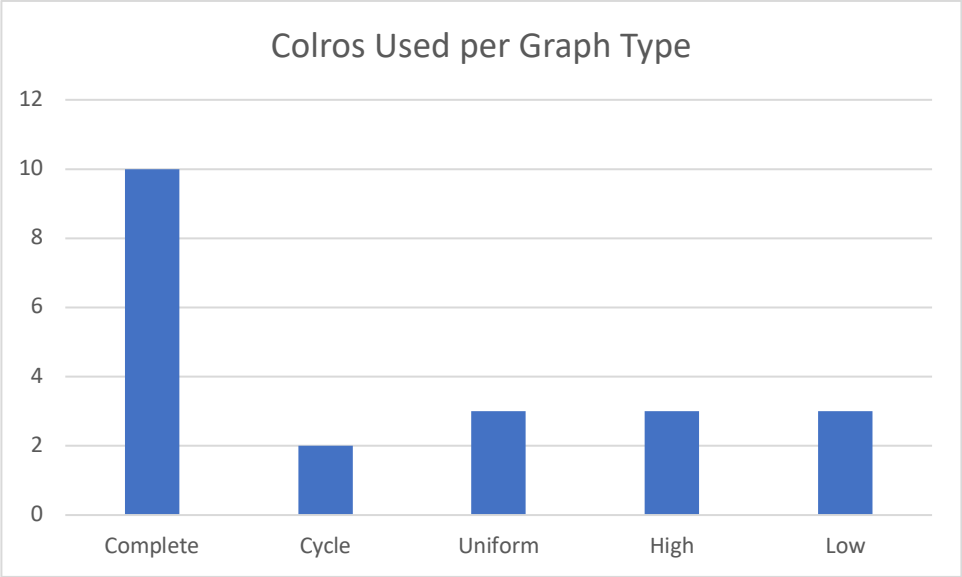


Figure 12 Colors User per Graph Type

As seen here, complete graphs take V number of colors. Cycle graphs either take 2 or 3 colors depending on if V is odd or even. The random graphs can take any number of colors, usually a low number though. In generating this graph, the average of three colorings were used to get 3 colors per graph for all the random graph generators.

The different ordering methods behave differently based on the graph given. When the graph is a cycle, it doesn't really matter which ordering method is used because at most each vertex will

have 2 edges.

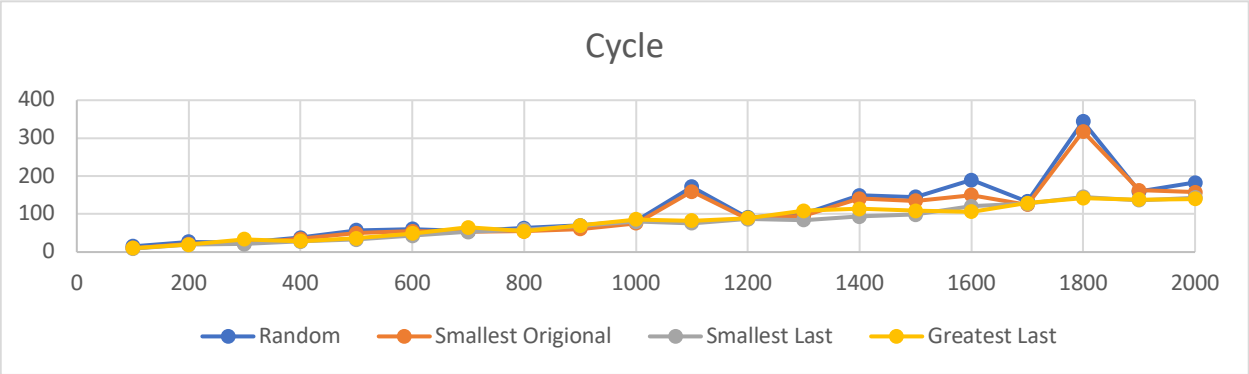


Figure 13 Runtimes of Orderings

This graph depicts the runtimes of coloring a cycle graph with the four ordering methods. One interesting note is that the only outliers come from the random and smallest original degree orderings. Smallest last vertex ordering and greatest last vertex ordering have very steady increases in time and do not have major outliers. This is due to the methodical ordering which provides the easiest coloring order for the coloring algorithm.

One interesting graph generated is from coloring a random graph skewed low. The smallest original degree competes with smallest last vertex ordering in terms of faster runtime. This was the only graph I observed this behavior on. All others had smallest last vertex ordering being the best overall.

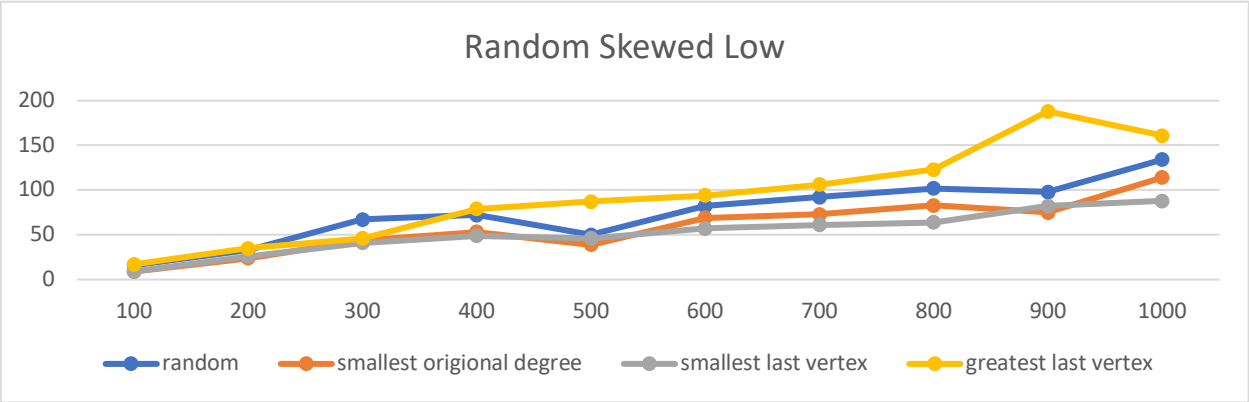
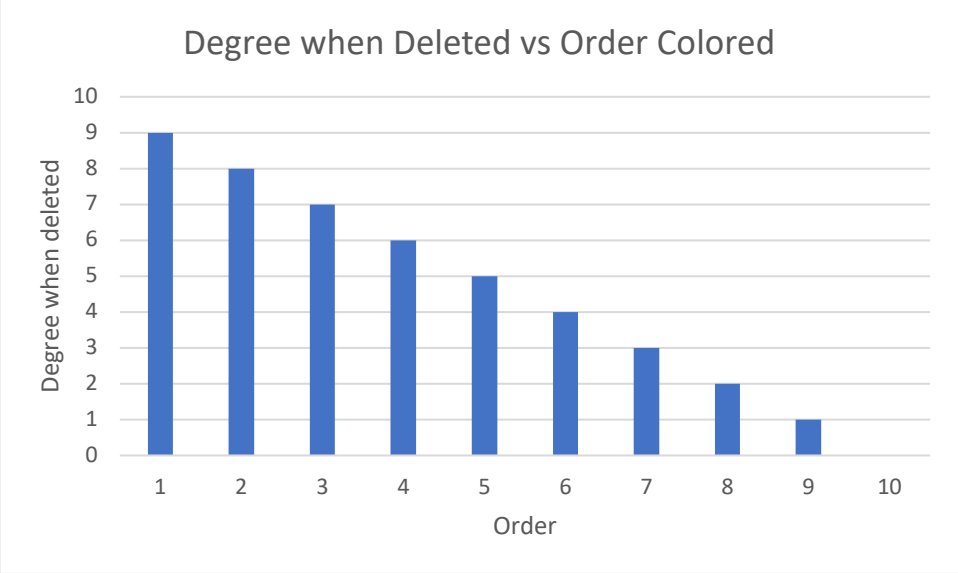


Figure 14 Runtimes of Coloring Random Graph Skewed Low

The greatest last vertex ordering did the worst, with random being just slightly better. As expected, smallest last vertex ordering did the best. All the ordering will perform somewhat similarly with only marginal benefit of using smallest last vertex ordering because the coloring algorithm still has to loop through every vertex to see if it can assign the color number to the vertex being considered. Even with complete graphs, smallest last vertex ordering performed marginally better than the others. The ordering is consistently the best and fastest, but not by much.

Figure 15 Degree When Deleted vs Color Order



For an input size of 10 on a complete graph, there were 10 colors used. The graph above shows the order of coloring versus the degree when deleted. The first colored vertex had a degree of 9, the second had a degree of 8, and so on. The maximum degree when deleted for a complete graph will always be V-1 because every vertex is connected to every other vertex. The terminal clique for any complete graph is V because every time a vertex is removed, all the other vertices are still connected to all other remaining vertices. For a cycle graph, the terminal clique is 2 for V>3. For the random graphs, the terminal clique is usually less than 5 as seen from examining the output of multiple input sizes. The terminal clique will always determine the number of colors needed in smallest last vertex ordering. The maximum degree when deleted + 1 will always be an upper bound on the number of colors needed, no matter the graph type.

Conclusion

In conclusion, graph coloring is a NP-complete problem meaning that no one currently knows how to efficiently determine the number of colors needed. This study has examined four ordering methods combined with 5 different graph types being colored by a single coloring algorithm. It can be concluded that smallest last vertex ordering can consistently perform the best over any graph type, even if by a marginal amount.

References

[1] "Graph Coloring (Greedy)," 2023. [Online]. [Accessed 2023].