Graph Coloring Analysis Project Report

Kylie Jordan

Introduction:

Graph creation and coloring can be done many ways where each has advantages and disadvantages for computing efficiency. For this paper, 3 different types of graphs will be created: cyclic, complete, and random with 3 distributions (uniform, left skewed, and normal). 3 different coloring methods will be used on the outputs of the graph creations including smallest last ordering, smallest original ordering, and random ordering. These methods all have different asymptotic running times based on the difficulty of their execution. The sections below will discuss the code base for this project, the implementations for each graph creation and coloring method, as well as provide a comparison between all of them.

Code base:

This project was created in C++ and tests were ran on a MacBook Pro, dual Intel-Core, version 11.6.5 Big Sur. The code itself has 3 major classes: Graph, Creator, and Analyzer. Their core functions and subclasses are listed below.

Graph:
- -graph : type LinkedList<Edge>
- +degrees : type array[20000]
- +getEdge(edge name)
- +addEdge(Edge)

Edge:
- -edge : type LinkedList<Vertex>
- -head : type Vertex

Vertex:
- -id : type int
- -degree : type int
- -color : type int

Creator:

-g : type Graph<Vertex>
+completeGraph(vertices)
+cycleGraph(vertices)
+randGraph(vertices, conflicts, distribution number)
+runCreations()
+analyzeTimes(vertices, conflicts)

Analyzer:

-g : type Graph<Vertex>
-order : type vector<int>
+read()
+compDegree()
+smallLastOrder()
+smallOrgOrder()
+randOrder()
+color()

Graph Class:

The Graph class has two subclasses used in the graph: Vertex and Edge. A Graph is of type Vertex. A Graph is a list of Edges of a list of Vertex's. Essentially, this creates an Adjacency List of Vertex's. Each list in the graph is the conflicts for that lists head Vertex. An example of what this looks like is pictured below.

1 -> 6 -> 2
2 -> 5 -> 1 -> 6
3 -> 4
4 -> 6 -> 3 -> 5
5 -> 2 -> 4
6 -> 1 -> 4 -> 2

In the example above, each highlighted number is a vertex in the graph. The numbers connected to the right are each of the vertices it is connected to. This is a doubly linked list so if vertex 2 appears in the conflict list of 6, then 6 must appear in the conflict list of 2. Each Vertex has identification information including number, degree, and color. These class members are important to the ordering and coloring algorithms to be discussed.

Creator Class:

The Creator class contains functions that create Complete, Cycle, and Random graphs. The two other functions labeled runCreations() and analyzeTimes() do timing analysis on the creation algorithms and output them to the screen.

A complete graph is one where all vertices are connected to each other. Each vertex has the same degree of # of Vertices-1. An example is shown below.

1 -> 2 -> 3
2 -> 1 -> 3
3 -> 1 -> 2

A cyclic graph is one where each vertex is connected to only two other vertices and the whole graph makes a cycle: the end wraps around to the beginning. An example is shown below.

1 -> 2 -> 5
2 -> 1 -> 3
3 -> 2 -> 4
4 -> 3 -> 5
5 -> 4 -> 1

A random graph is created by randomly choosing a vertex and conflict. However, the choice is based on a probability distribution. For this code base there are 3 probability distributions: uniform, left skewed, and normal. Uniform distribution gives every vertex an equal probability of being chosen. Left skewed distribution gives lower numbered vertices a higher probability of being chosen than higher numbered. Normal distribution gives higher probability of being chosen to the mean value of the set. Figures 1 through 3 respectively show these distributions.

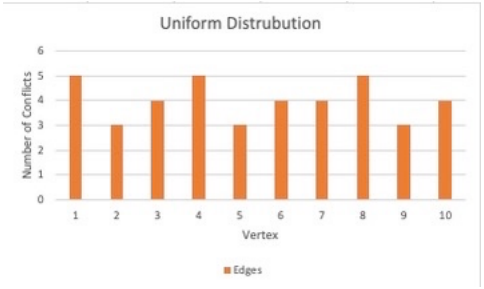

Figure 1: Uniform Distribution

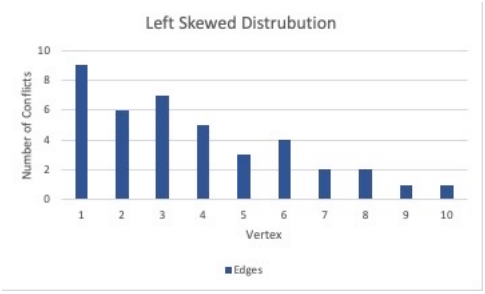

Figure 2: Left Skewed Distribution

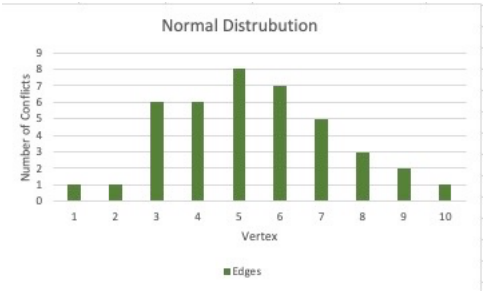

Figure 3: Normal Distribution

Each of these graph creations were analyzed for their runtime efficiency. Figure 4 shows runtimes for random graph creations for all 3 distributions as well as the cycle graph creation. The number of vertices increased by 100 for sets of 100-1000. The number of conflicts increased by 100 for sets of 200-1100. The complete graph creation became exponentially larger than the other algorithms for the same vertex numbers, so it was necessary to test with smaller data sets as seen in Figure 5. The complete graph creation was tested with numbers of vertices ranging from 10-100. The full comparison of all algorithms can be seen in Figure 6.

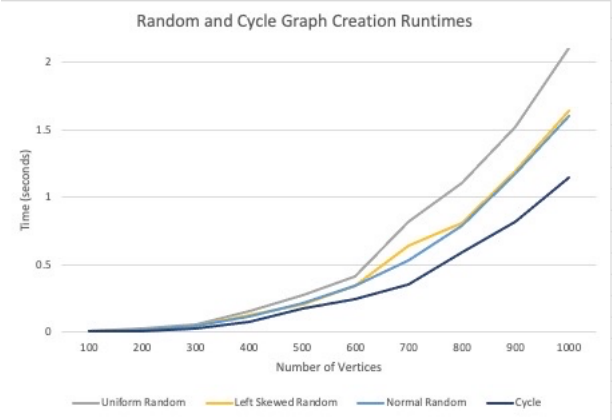

Figure 4: Random and Cycle Graph Creation Runtimes

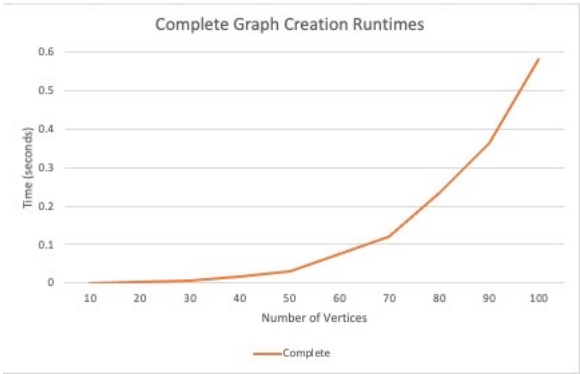

Figure 5: Complete Graphs with Smaller # of Vertices

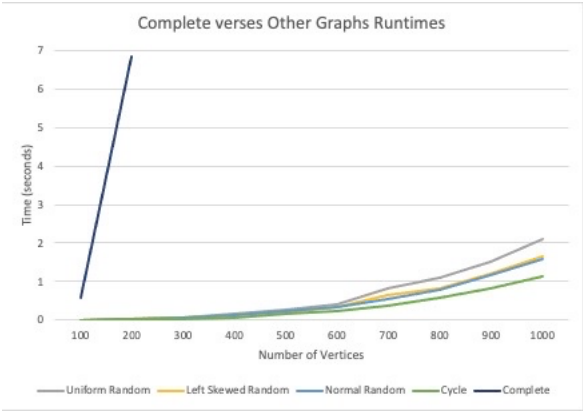

Figure 6: Comparison of Complete Verses Other Graph Runtimes

By analyzing the graphs in Figures 4-6, the approximate running times for the implementations can be divulged. The random graph creation algorithms all run approximately within the same timing. Uniform distribution may overtake the other two distributions overt time but as seen in the graph, the times increase by 4x as the number of vertices increase by 2x. The cycle graph creation is more efficient than the random implementations and increases by 3x

every time the number of vertices doubles. The complete graph increases by 8x every time the number of vertices doubles making it much more inefficient than the rest. The following are the determined asymptotic running times for the graph creation algorithms:

Random:

Uniform: $\Theta(n^2)$

Left Skewed: $\Theta(n^2)$

Normal: $\Theta(n^2)$

Cycle: $\Theta(lgn)$

Complete: $\Theta(n^3)$

Analyzer Class:

This class has the capability to read and compute graph colorings for 3 different orderings: Smallest Last Vertex, Smallest Original Degree Last, and Random. Smallest Last ordering computes the following algorithm:

1. While there are still vertices in graph
    a. Find vertex with smallest degree
    b. Add vertex to back of order
    c. Delete vertex from graph
2. Update vertices and their degree to not include the deleted vertex
3. Repeat and check step 1

At the end of the algorithm, the order will be in reverse from when each vertex was deleted.

Example walkthrough:

Original graph:
1 -> 4 -> 3
2 -> 3 -> 6 -> 5
3 -> 1 -> 2 -> 4 ->5
4 -> 1 -> 3
5 -> 3 -> 2
6 -> 2

First delete:
1 -> 4 -> 3
2 -> 3 -> 6 -> 5
3 -> 1 -> 2 -> 4 ->5
4 -> 1 -> 3
5 -> 3 -> 2
6 -> 2

1 -> 4 -> 3
2 -> 3 -> 5
3 -> 1 -> 2 -> 4 ->5
4 -> 1 -> 3
5 -> 3 -> 2

Order: { , , , , , 6}

Next delete:
1 -> 4 -> 3
2 -> 3 -> 5
3 -> 1 -> 2 -> 4 ->5
4 -> 1 -> 3
5 -> 3 -> 2

2 -> 3 -> 5
3 -> 2 -> 4 ->5
4 -> 3
5 -> 3 -> 2

Order: { , , , , 1, 6}

Next delete:
2 -> 3 -> 5
3 -> 2 -> 4 ->5

4 -> 3
5 -> 3 -> 2

2 -> 3 -> 5
3 -> 2 -> 5
5 -> 3 -> 2

Order: { , , , 4, 1, 6}

Next delete:
2 -> 3 -> 5
3 -> 2 -> 5
5 -> 3 -> 2

3 -> 5
5 -> 3

Order: { , , 2, 4, 1, 6}

Next delete:
3 -> 5
5 -> 3

5 -> 3

Order: { , 3, 2, 4, 1, 6}

Next delete:
5 -> 3

Final Order for Smallest Last Vertex Ordering: { 5, 3, 2, 4, 1, 6}

Smallest Original ordering computes the following algorithm:

1. For all vertices in graph
    a. Find smallest degree
    b. Add vertex to back of order
    c. Change vertex degree to -1
2. Repeat step 1 until all vertices are in the ordering

At the end of the algorithm, the order will be in reverse from largest to smallest vertex degree from the original graph.

Random ordering computes the following algorithm:

1. While there has been less than 1.5x # vertices of swaps
    a. Choose two random vertices
    b. Swap them
2. Repeat step 1 until 1.5x # vertices swaps have been made

At the end of the algorithm, the order will be random from the original graph.

Figures 7-9 show how these ordering algorithms performed on cyclic, complete, and uniform random graphs.

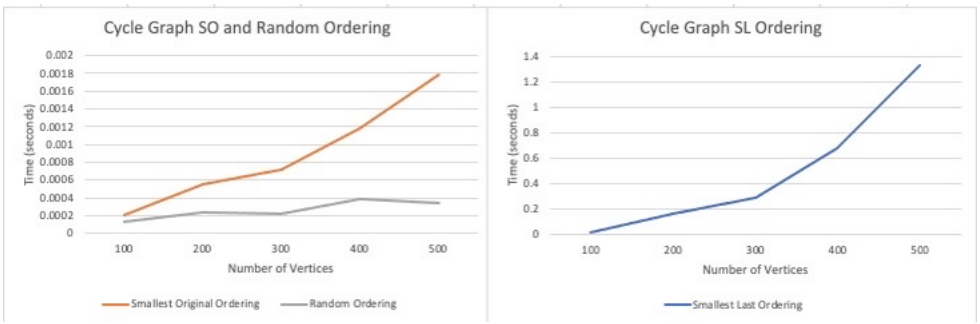

Figure 7: Cycle Graph Ordering Runtimes

Figure 8: Complete Graph Ordering Runtimes

Figure 9: Uniform Random Graph Runtimes

Table 1 shows the data in the Figure 7-9 more precisely which gives a better estimate for each algorithm's asymptotic runtimes. While the running times differ slightly based on the type of graph it is ordering, on average they tend to behave similarly. For Smallest Last Ordering, the runtimes tend to increase by 4x as the number of vertices doubles. For Random Ordering, the running time increases by approximately 0.0001 as the number of vertices increase by 100. The final determinations are listed below:

Smallest Last Ordering: $\Theta(n^2)$

Random Ordering: $\Theta(n)$

Table 1: Ordering and Coloring runtimes

Cycle						
Vertices	**Smallest Last Ordering**	**SL Coloring**	**Smallest Original Ordering**	**SO Coloring**	**Random Ordering**	**Rand Coloring**
100	0.015511	0.038821	0.000202	0.041157	0.000126	0.057134
200	0.159029	0.457405	0.000554	0.577131	0.00023	0.301278
300	0.285857	0.862088	0.000722	0.863095	0.000214	0.861897
400	0.679501	2.02745	0.001183	2.02141	0.000382	2.04007
500	1.33033	4.16608	0.001782	4.15388	0.000337	4.16357
Uniform						
Vertices	**Smallest Last Ordering**	**SL Coloring**	**Smallest Original Ordering**	**SO Coloring**	**Random Ordering**	**Rand Coloring**
100	0.045669	0.138251	0.000263	0.057023	0.000139	0.055502
200	0.114468	0.349773	0.000482	0.34727	0.000187	0.347265
300	0.337147	1.10778	0.000969	1.17194	0.000269	1.16761
400	0.958382	3.01534	0.001896	2.89248	0.000326	2.81471
500	2.11377	6.39013	0.002322	6.38708	0.000393	6.39204
Complete						
Vertices	**Smallest Last Ordering**	**SL Coloring**	**Smallest Original Ordering**	**SO Coloring**	**Random Ordering**	**Rand Coloring**
10	0.000105	0.000212	3.30E-05	0.000213	3.10E-05	0.000225
20	0.000682	0.00206	7.30E-05	0.002079	8.70E-05	0.001916
30	0.002701	0.008738	0.000134	0.008722	0.000126	0.008723
40	0.008056	0.027078	0.000232	0.027113	0.000224	0.026989
50	0.019531	0.064899	0.000359	0.065527	0.000355	0.06828

Coloring Algorithm:

Graph coloring is a way to represent the edges in a graph by giving different values to vertices on the same edge. Vertices with the same color do not have an edge, or conflict. The coloring algorithm implemented in the code based is outlined below:

1. For vertices in the order
    a. If all of its edges do not have the current color value
    b. Set the vertex's color to the current color value
2. Repeat step 1

Table 1 shows the data for the coloring algorithm for each ordering. Figure 10 shows how the coloring algorithm performed on different graph orderings and types. The coloring algorithm performed approximately the same for all orderings for all graph types. The runtime tends to increase by 4x as the number of vertices doubles. The final asymptotic runtime determination is listed below:

Coloring algorithm: $\Theta(n^2)$

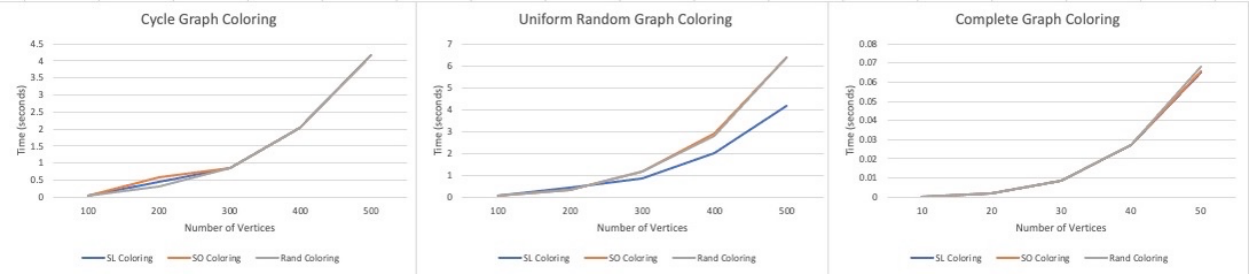

Figure 10: Graphs for Coloring Algorithm on Different Orderings

Comparison and Applicability:

This code base allows for 5 different graph creation types and 3 different graph orderings for a total of 15 different combinations of graph representation.

In terms of creation, this code base is efficient with the fastest algorithm being cycle graphs and the slowest being the complete graphs. The reason for this is the number of edges that need to be created and inserted into the graph. The complete graph requires more edges which will make it take more time overall. All random graph creations for the 3 distributions perform similarly with little difference.

In terms of ordering and coloring, random ordering is the fastest algorithm taking time relational to the number of vertices. This is much faster than the smallest last vertex ordering which takes $(number\ of\ vertices)^2$ time. This is exponentially slower than the random ordering. The Smallest Last Ordering performs best on Cycle graphs as seen in Figures 7-9. The Random ordering performs best on uniform random graphs also seen in Figures 7-9.

Overall, the type of graph and ordering has little effect on the efficiency of the coloring algorithm. However, as seen in Figure 10, the coloring algorithm on Smallest last ordered, uniform, random graphs tends to be more efficient than with the other two orderings and graph types.

This code base can be used in many applications including CPU process allocation and network benchmark scheduling. Graph coloring is ultimately used to find vertices that do not have conflicts. When a CPU is attempting to allocate processes into registers, the processes must not require resources at the same time. If they do, this would be a conflict. So the CPU can use this code base to find the best way to schedule processes such that they will not need the same resources at the same time. In network benchmark scheduling, this can be used to find positive conflicts. If a benchmark needs to be run on multiple VMs, this is a conflict. Instead of creating

and tearing down provisions for this benchmark for every VM, they can be scheduled back-to-back so that every benchmark run uses the same provisions. The code base can be used in both these applications to find and work around or with conflicts.

Limitations:

As with any algorithm implementation, it can always be made more efficient. While all algorithms in this code base run in under $\Theta(n^3)$ time, this is inefficient for large datasets. In the future, the graph creation algorithm for complete graphs should be analyzed to be made faster. The Graph class in this code base can also be made faster with a focus on the function to add edges. The addEdge() function in the graph class takes the time to ensure there are no repeating edges or vertices which takes a lot of time. This can be made more efficient by using data structures that do not allow duplicates like the std::set in C++.

Another limitation to this code base is the format required for input to graph ordering and coloring. Examples have been provided in the code base and follow this format:

- P[] = Pointer for each course I, 1 <= I <= N denoting the starting point in E[] of the list of courses in conflict with course I. That is, the conflicts for course I are indicated in locations E[P[I]], E[P[I]+1], ..., E[P[I+1]-1].
- E[] = adjacency list of distinct course conflicts (length = 2M)

This is very specific to this code base and not very versatile. The ordering and coloring algorithms require this code bases graph creation for the input since it is so specific.

Conclusion:

This report outlines the implementation and uses of a code base with graph creation, ordering, and coloring algorithms. The important parts of the code base have been discussed as well as the algorithms for the creation of cycle, complete, and random graphs with 3 distributions (uniform, left skewed, and normal). Smallest Last Vertex, Smallest Original Last, and Random graph orderings have been described and analyzed. Lastly, a coloring algorithm was discussed as well as how it performs with different ordering and graph types. Finally, comparisons and applications with the code bases limitations were also discussed with possible solutions.