Graph Coloring Analysis Report

John Park

CS 5350

Professor Lee McFearin

16 December 2020

Table of Contents

1. Introduction

1.1 Overview

This report seeks to provide empirical analysis on the runtime complexities of various vertex ordering and coloring algorithms. For this report, 6 vertex ordering algorithms were implemented and their performance measured:

<u>Vertex Ordering Algorithms</u>
1. Smallest-Last
2. Smallest Original Degree Last
3. Random
4. Incremental
5. Breadth First Search
6. Depth First Search

For coloring, 1 vertex coloring algorithm was implemented and its runtime complexity measured:

<u>Vertex Coloring Algorithm</u>
1. Greedy Coloring

For testing, these implementations were applied to graphs of varying type, density, and vertex distributions.

The graphs on which our vertex ordering and coloring implementations were tested fall under 3 types:

<u>Graph Types</u>
1. Cycle
2. Complete
3. Randomly-Generated

The randomly-generated graphs on which the vertex ordering and coloring algorithms' performance were measured range within 9 levels of density:

<u>Randomly-Generated Graph Density Levels:</u>
1. 10% of maximum possible number of edges
2. 20% of maximum possible number of edges
3. 30% of maximum possible number of edges
4. 40% of maximum possible number of edges
5. 50% of maximum possible number of edges
6. 60% of maximum possible number of edges

7. 70% of maximum possible number of edges
8. 80% of maximum possible number of edges
9. 90% of maximum possible number of edges

To further diversity the graphs by which our implementations were measured, the cycle, complete, and randomly-generated graphs' edges were uniformly randomly selected from a collection of vertices that fall within 3 different types of distributions:

Vertex Distribution Types:
1. Uniform
2. Skewed
3. Normal

In sum, to gain insight into the runtime complexities of graph ordering and coloring implementations, 6 vertex ordering and 1 vertex coloring algorithms were implemented and their performances measured. To thoroughly test their runtime performance, the implementations of these algorithms were applied to graphs of varying type, density, and vertex distributions.

1.2 Implementations

1.2.1 Data Structures

For this project, various data structures were implemented and used to exploit their unique characteristics that suit particular use cases:

Data Structure	Use	Justification
Vector	to keep track of the collection of vertices that obey a particular vertex distribution	- fast read O(1) - fast random access O(1) - fast write O(1) (amortized) - can contain non-unique elements - best when capacity is known before initialization
AVL Tree	to implement a Set	- only allows unique elements - fast read O(log(n)) - fast write O(log(n))
Set	to keep track of the collection of unique vertices inserted into a graph	- simpler interface for storing data into AVL Tree - only allows unique elements - fast read O(log(n)) - fast write O(log(n))
Doubly Linked List	to implement a Stack, Queue, and Adjacency List	- suitable when capacity remains unknown - uncommon random access (slow random access O(n)) - fast read front or back O(1) - fast write O(1)
Stack	to keep track of the order of vertices generated from vertex ordering algorithms	- simpler interface than Linked List - fast read O(1) - fast write O(1)

	to keep track of vertices to traverse in Depth First Search vertex ordering algorithm	
Queue	to keep track of vertices to traverse in Bread First Search vertex ordering algorithm	- simpler interface than Linked List - fast read O(1) - fast write O(1)
Adjacency List	to track the edges inserted and removed from an undirected graph	- efficient memory use O(V+E) - fast edge addition O(1) - marking a vertex as deleted O(1) - updating a vertex's degree O(1) - finding a vertex of particular degree O(1)

1.3 Hypothesis

In accordance with Dr. Matula's research [1], I hypothesize that my implementation of Smallest-Last vertex ordering will exhibit a runtime complexity consistent with his findings as well as with our discussion about this algorithm during our lecture on September 30th. In sum, I anticipate the following lower-bound time complexities for our vertex ordering and coloring implementations:

Algorithm Implemented	Time Complexity
Smallest-Last	Ω(V+E)
Smallest Original Degree Last	Ω(V)
Random	Ω(V)
Incremental	Ω(V)
Breadth First Search	Ω(V+E)
Depth First Search	Ω(V+E)
Greedy Coloring	Ω(V+E)

2. Computing Environment

The analyses presented in this report were performed under the following specifications:

2.1 Hardware

Hardware	
Brand	Apple
Model	13-inch MacBook Pro
Year	2019
Processor	2.4 GHz Quad-Core Intel Core i5 64-bit (Turbo Boost up to 3.9GHz)
Memory	16 GB 2133MHz LPDDR3
Graphics	Intel Iris Plus Graphics 655
Storage	512 GB 2.7 GHz PCIe

During all of the tasks, the computer's power was plugged into an electric outlet at all times, meaning the machine's CPU and GPU were able to operate at peak performance without having to oncern themselves with power conservation.

2.2 Software

Software	
Operating System	macOS Catalina Version 10.15.7
Language	C++14
Compiler	Clang Version 11.0.3

While the C++ programs related to this project were running, all of the compilation and execution processes took place locally within the machine. Other than the operating system's core functions, no other application was running besides this report's C++ programs.

3. Vertices

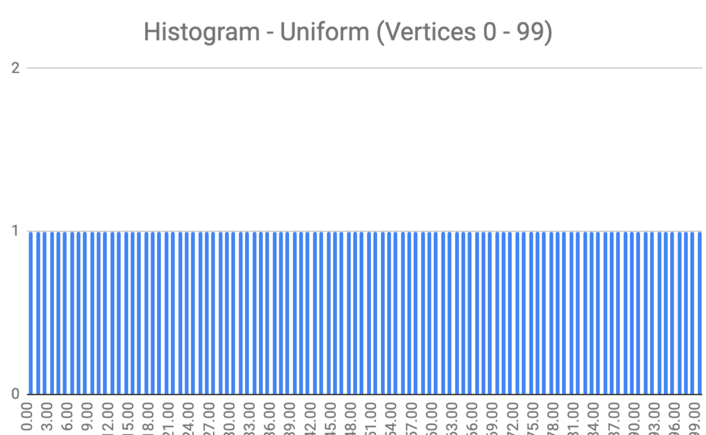
Vertices were generated according to the following distributions:

1. Uniform
2. Skewed
3. Normal

These distributions were chosen since they follow distinct probability curves. The collection of vertices generated from these distributions were then used to generate different graphs.

3.1 Uniform

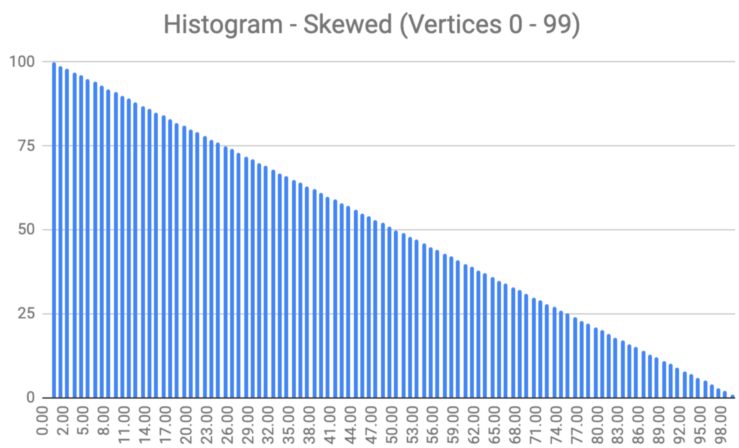
Since uniform distribution entails that all elements are equally likely to be chosen, its probability curve takes the shape of a horizontal line whose y-value represents the equal probability that any of the elements in a collection of vertices will be chosen. For instance, if there are 100 vertices stored in a vector ranging from values 0 to 99, if each vertex is present in this vector once, each vertex value has the equal 1 out of 100 chance of being chosen. Thus, in this case, the probability curve of uniform distribution would exhibit a y = 1/100 curve while its histogram would follow a y = 1 curve. In accordance with this, when 100 vertices whose values ranging from 0 to 99 are generated using our implemented uniform distribution generator function `uniform` (Appendix B), the following histogram is generated.



Since this histogram follows a y = 1 curve, the histogram indicates that `uniform` generates a vector of vertices whose distribution follows a uniform distribution.

3.2 Skewed

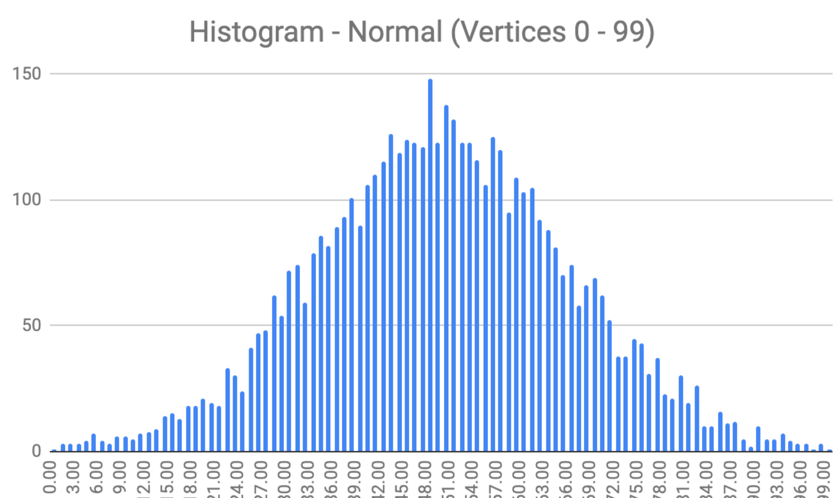
Since a linearly skewed distribution entails that the probability of a later element being chosen than its former decreases linearly, its probability curve manifests as a line with a negative slope whose y-values are on or above the x-axis. Given that 100 different vertices (Vertices 0 to 99) are stored in a vector, a histogram made from the collection of vertices that follows this distribution should exhibit a line with a negative slope whose y-values are on or above the x-axis. In accordance with this, when 100 vertices whose values ranging from 0 to 99 are generated using our implemented skewed distribution generator function `linear` (Appendix B), the following histogram is created.



Since this histogram follows a line with a negative slope whose y-values are on or above the x-axis, it indicates that `linear` generates a vector of vertices whose distribution follows a skewed distribution.

3.3 Normal

Since a normal distribution follows a Gaussian curve whose mean is approximately at the center of the curve and is also the curve's highest point, we expect our histogram to follow the characteristics of a Gaussian curve given that the histogram is generated from a collection of normally distributed vertices. Consistent with this, when vertices ranging from values 0 to 99 are generated using our implemented normal distribution generator function `normal`, the following histogram is produced (Appendix B).



Since this histogram follows a Gaussian curve, it indicates that `normal` generates a vector of vertices whose distribution follows a normal distribution.

A collection of vertices that follow a certain distribution should generate a graph whose edges differ from those generated from a collection of vertices that follow another distribution. For example, since vertex of values between 0 and 30 are more common in a collection of vertices of skewed distribution than in a collection of vertices of normal distribution whose most common values hover between 35 and 65 (as seen in the earlier skewed and normal distribution histograms), if a graph were to be generated from collection of vertices that follow a skewed distribution, there is a higher likelihood that the edges will consist of values between 0 and 30 more so than consisting of 35 and 65. Conversely, if a graph were to be generated from a collection of vertices following a normal distribution of 100 vertices between the values 0 and 99, it is more likely that its vertices will consist of values between 35 and 65 than between 0 and 30. Thus, the type of distribution of the vertices from which a graph is generated can impact the composition of its edges. In sum, the different distribution types serve as a means to generate distinct graphs that can be used to assess the runtime performance of our vertex ordering and coloring implementations.

4. Graphs

Other than using distinct vertex distributions, we can further construct distinct graphs by creating those of different types. For this report, 3 types of graphs were implemented:

1. Cycles
2. Complete Graphs
3. Randomly-Generated Graphs

4.1 Cycles

Cycles are the most sparse type of graphs analyzed in this report whose number of vertices and edges share the following relation:

# of edges in a cycle= # of vertices

4.2 Complete

Complete graphs, on the other hand, are the most dense possible graphs whose vertices connect to all other vertices adjacently. In a complete graph, the number of vertices and edges share the following relation:

*# of edges in a complete graph = # of vertices * (# of vertices - 1) / 2*

Moreover, since all the vertices in a complete graph are connected to one another adjacently, the number of edges in a complete graph also equals the maximum possible number of edges that can be inserted into a graph:

*max # of edges in a graph = # of vertices * (# of vertices - 1) / 2*

4.3 Random

Through our randomly-generated graphs, we seek to fill in the intermediary density levels between that of a cycle and a complete graph. Given that we have V distinct vertices, the number of edges that can be inserted into our randomly-generated graph fall within the following range:

$$V < \text{\# of edges inserted in a random graph} < V * (V - 1) /2$$

To fill this range, we will take the maximum number of edges possible for a graph and multiply this by a density-level factor ranging from .1 (10% of the maximum number of edges possible) to .9 (90% of the maximum number of edges possible) by a fixed increment of .1 (+10% of the maximum number of edges possible). Thus, the random graphs generated will span the following densities:

1. 10% of max # of edges
2. 20% of max # of edges
3. 30% of max # of edges

4. 40% of max # of edges
5. 50% of max # of edges
6. 60% of max # of edges
7. 70% of max # of edges
8. 80% of max # of edges
9. 90% of max # of edges

Since the maximum number of edges possible for a graph is equal to the number of edges in a complete graph, the number of edges in our randomly-generated graph shares the following relation with the number of edges in a complete graph:

*# of edges in a randomly-generated graph = density factor * # of edges in a complete graph*

Applying our earlier relation between the number of edges we have in a complete graph and the graph's number of vertices:

*# of edges in a randomly-generated graph = density factor * # of vertices * (# of vertices - 1) / 2*

where .1 $\leq$ density factor $\leq$ .9 and increments by .1

5. Ordering Predictions

Now that we have established a means to obtain graphs whose vertex distribution, graph type, and density levels vary, we will investigate the different methods of vertex ordering implemented in this report:

1. Smallest-Last
2. Smallest Original Degree Last
3. Random
4. Incremental
5. Breadth First Search
6. Depth First Search

Consistent with the hypothesis mentioned at this report's introduction, I predict that all of these vertex ordering algorithms will exhibit the following lower-bound complexities:

Algorithm	Time Complexity
Smallest-Last	Ω(V+E)
Smallest Original Degree Last	Ω(V)
Random	Ω(V)
Incremental	Ω(V)
Breadth First Search	Ω(V+E)
Depth First Search	Ω(V+E)

5.1 Smallest-Last

The Smallest-Last vertex ordering algorithm works as follows:

```
While graph is NOT empty:
     Find vertex with smallest degree from graph;
     Push removed vertex with smallest degree to stack;
     Remove vertex with smallest-degree from graph;
     Update graph to find next smallest degree;
```

Since we successively push the vertex with minimum degree `m` onto our stack, the vertices with smallest degrees will be further and further down the stack than the ones that did not have the smallest degrees at the point of `m`'s deletion. Because Smallest-Last algorithm as discussed during September 30th's lecture finds the vertex with smallest degree in the graph, pushes this vertex to the stack, and removes it from the graph in constant time and these steps are repeated for every vertex, I anticipate my implementation of this algorithm to exhibit a time complexity that is at least linearly proportional to a graph's number of vertices. Furthermore, because updating a graph after a vertex's deletion involves updating the degrees of every vertex adjacent to the deleted vertex, updating adjacent vertices involves making a number of updates that grow linearly with the number of edges in a graph. Thus, I anticipate my implementation of the Smallest-Last algorithm to exhibit runtime complexity that at least grows linearly with the number of edges in a graph.

Thus, I expect the implemented Smallest-Last vertex ordering function `smallestLast` (Appendix D) to exhibit a time complexity that is at least linearly proportional to `V` (the number of vertices) and `E` (the number of edges):

1. From an algorithmic standpoint, finding the node that contains the vertex with the minimum degree is a constant time operation that repeats for every vertex. Thus, my implementation of finding this node has to be at least linearly proportional to the number of vertices in a graph.
2. From an algorithmic perspective, accessing the vertex value of a node with a minimum degree is a constant time operation, and this vertex-value accessing iterates for every node visited. Thus, my implementation of this vertex-value accessing has to be at least linearly proportional to the number of vertices in a graph.
3. From an algorithmic viewpoint, marking a vertex as deleted is a constant time operation that repeats for every vertex. Thus, my implementation of this has to be at least linearly proportional to the number of vertices.
4. Since Smallest-Last algorithm visits every vertex and updates the degrees of a visited vertex's every adjacent vertex, updating adjacent vertex's degrees and its location will iterate `E` times. Thus, my implementation of updating every vertices' adjacent vertices has to be at least linearly proportional to the number of edges.

For the reasons stated above, I expect my Smallest-Last vertex ordering implementation to exhibit a lower-bound runtime complexity of $\Omega(E + V)$.

5.2 Smallest Original Degree Last

Since the Smallest Original Degree Last vertex ordering algorithm reproduces every step from Smallest-Last algorithm except marking the visited vertices as deleted and updating every visited vertices' adjacent vertices' degrees, I predict that my implementation of the Smallest Original Degree Last vertex ordering algorithm will have a lower bound running time of $\Omega(V)$ for reasons 1 and 2 mentioned in the previous subsection.

5.3 Random

Since random vertex ordering algorithm involves:

1. Copying every vertex value from an array of V vertices
2. Shuffling an array of V integers using Fisher-Yates shuffling algorithm, whose implmentation exhibits $\Omega(V)$ time complexity (additional details provided below).
3. Iterating through a shuffled array of V integers and pushing every integer on to the stack involves visiting every vertices in a graph and thus iterates V times

I expect my implementation of my random vertex ordering algorithm to exhibit lower-bound time complexity of $\Omega(V)$. To go further into why I believe this to be the case, I will now explore in-depth one of the Random vertex ordering algorithm's essential functions: randomizing the vertex ordering by shuffling an array of vertices.

5.3.1 Fisher-Yates Shuffling

The vertex shuffling implementation used in my random vertex ordering algorithm exhibits Ω(V) time complexity since it utilizes Fisher-Yates shuffling algorithm, which does the following:

```
for(int i = V - 1; i > 0; --i):
     j = random integer 0 <= j <= i;
     arr[i] = arr[j];
```

Thus, since the algorithm:

1. Iterates through a for-loop with the iteration counter `i` that is initialized to `V - 1` and is then decremented `V - 1` times
2. Generates an integer `j` whose randomly-generated value lies between `0` and `i` inclusively (iterated `V - 1` times)
3. Swaps the `arr`'s elements (from its last element to its second element) with the randomly selected element at index `j` (iterated `V - 1` times)

Its constant-time operations of decrementing `i`, generating a random number between `0` and `i`, and swapping `arr`'s elements are iterated `V - 1` times. Since the number of times these operations are iterated grows linearly with V, the number of vertices stored in a given array, my implementation of this algorithm will exhibit a runtime that grows at least linearly with `arr`'s given number of elements, which equals the number of vertices in a given graph. Thus, I anticipate my implemented vertex shuffling method used in my random vertex ordering algorithm to exhibit a lower-bound time complexity of Ω(V).

5.4 Incremental

Since incremental vertex ordering algorithm, involves:

1. Iterating through every vertices
2. Pushing every vertex on to the stack

I expect my implementation of this algorithm to exhibit a lower-bound time complexity of Ω(V).

5.5 Breadth First Search

Since breadth first search vertex ordering algorithm, involves:

1. Visiting every vertex in a graph
2. Visiting every visited vertex's every adjacent vertices

I expect my implementation of this algorithm to exhibit a lower-bound runtime complexity of Ω(V+E).

5.6 Depth First Search

Since depth first search vertex ordering algorithm involves:

1. Visiting every vertex in a graph
2. Backtracking through every visited vertex's adjacent vertices

I expect my implementation of this algorithm to exhibit lower-bound complexity of $\Omega(V+E)$.

5.7 Ordering Predictions Overview

In sum, I predict the following time complexities for my vertex ordering implementations:

Algorithm Implemented	Time Complexity
Smallest-Last	$\Omega(V+E)$
Smallest Original Degree Last	$\Omega(V)$
Random	$\Omega(V)$
Incremental	$\Omega(V)$
Breadth First Search	$\Omega(V+E)$
Depth First Search	$\Omega(V+E)$

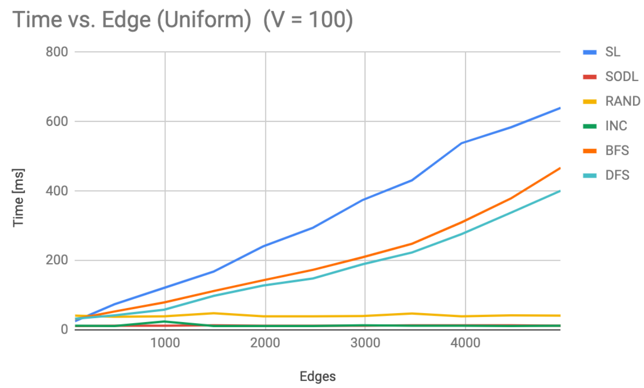
6. Ordering Empirical Results

6.1 Time vs. Edges

6.1.1 Uniform

In accordance with the predictions made, for a graph consisting of 100 vertices whose edges are generated from a collection of uniformly distributed vertices, the runtime complexities for Smallest Original Degree Last, Random, and Incremental implementations remain constant in relation to the growing number of edges:

Time [ms] vs Edges (Uniform) (V = 100)						
Edges	SL	SODL	RAND	INC	BFS	DFS
100	25	11	41	12	31	32
495	74	12	38	11	53	42
990	121	12	39	24	79	58
1485	168	13	48	11	112	98
1980	241	12	39	11	143	128
2475	294	12	39	11	173	148
2970	374	12	40	13	209	189
3465	431	13	47	12	248	223
3960	538	13	39	12	310	276
4455	584	13	42	11	379	338
4950	640	12	41	12	467	401



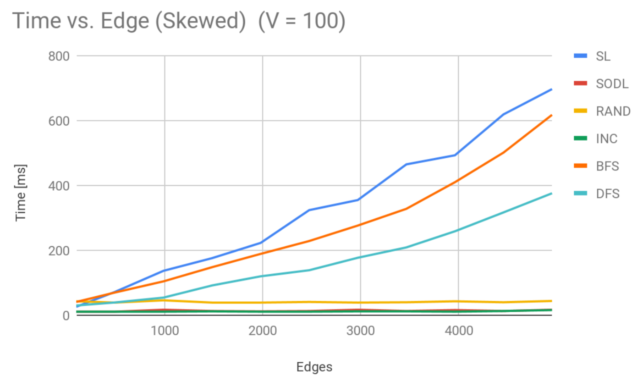
For Smallest-Last, Breadth First Search, and Depth First Search implementations, however, their runtimes grow linearly as the number of edges increases. The following table highlights the linear growth in runtime for these 3 algorithms' implementations.

N: Factor Increase in Edges; T [ms]: Factor Increase in Time							
Edges	N	SL	T_SL	BFS	T_BFS	DFS	T_DFS
495		74		53		42	
990	2	121	1.835	79	1.791	58	1.681
1980	2	241	1.992	143	1.810	128	2.207
3960	2	538	2.232	310	2.168	276	2.156

6.1.2 Skewed

As for a graph consisting of 100 vertices whose edges are generated from a collection of skewed distribution of vertices, as with uniform distribution, the time complexities for Smallest Original Degree Last, Random, and Incremental implementations remain independent from the growing number of edges.

Time [ms] vs Edges (Skewed) (V = 100)						
Edges	SL	SODL	RAND	INC	BFS	DFS
100	26	11	43	11	41	30
495	72	11	39	11	70	39
990	137	17	46	11	104	54
1485	176	13	39	12	148	92
1980	223	12	39	11	189	120
2475	324	13	41	11	229	139
2970	355	17	39	12	276	177
3465	465	13	40	12	328	209
3960	493	16	43	11	410	259
4455	619	13	40	13	501	317
4950	697	17	44	16	618	376



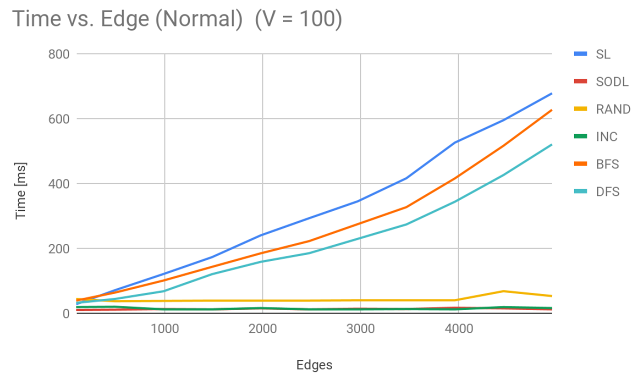
For Smallest-Last, Breadth First Search, and Depth First Search algorithms, however, as the number of edges increases, their runtimes increase linearly. The following table highlights this relationship:

N: Factor Increase in Edges; T [ms]: Factor Increase in Time							
Edges	N	SL	T_SL	BFS	T_BFS	DFS	T_DFS
495		72		70		39	
990	2	137	2.103	104	1.791	54	1.681
1980	2	223	1.628	189	1.810	120	2.207
3960	2	493	2.211	410	2.168	259	2.156

6.1.3 Normal

As for a graph consisting of 100 vertices whose edges are generated from a collection of normal distribution of vertices, the time complexities for Smallest Original Degree Last, Random, and Incremental implementations remain the same as the number of edges increases.

Time [ms] vs Edges (Normal) (V = 100)						
Edges	SL	SODL	RAND	INC	BFS	DFS
100	28	10	44	19	39	32
495	71	11	37	20	67	46
990	121	13	38	12	92	63
1485	173	12	39	12	131	100
1980	240	15	39	16	169	129
2475	293	12	39	12	200	144
2970	345	14	40	12	250	190
3465	416	13	40	13	292	224
3960	526	17	40	12	358	284
4455	595	15	68	19	443	343
4950	678	12	53	16	548	411



As the number of edges increases, the runtimes for Smallest-Last, Breadth First Search, and Depth First Search implementations increase linearly. The following table highlights this linear growth.

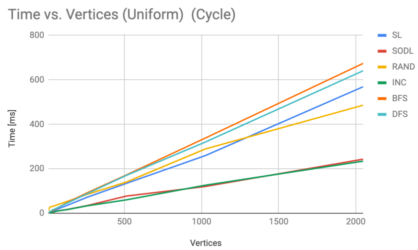
N: Factor Increase in Edges; T [ms]: Factor Increase in Time							
Edges	N	SL	T_SL	BFS	T_BFS	DFS	T_DFS
495		71		75		37	
990	2	121	1.904	108	1.750	43	1.465
1980	2	240	1.983	204	1.879	101	2.351
3960	2	526	2.192	449	2.205	215	2.119

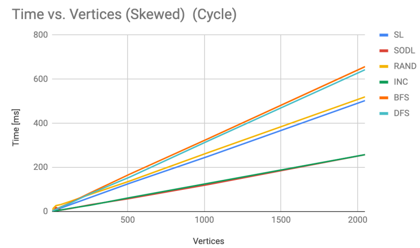
In sum, the vertex distributions had little effect on the vertex ordering algorithms' implementations' runtime performance. Whereas the runtime complexities for Smallest Original Degree Last, Random, and Incremental implementations remained constant in relation to the growing number of edges, the runtimes for Smallest-Last, Breadth First Search, and Depth First Search implementations grew linearly with the growing number of edges in a graph. These findings remain consistent with the predictions made in *5.7 Ordering Predictions Overview*.

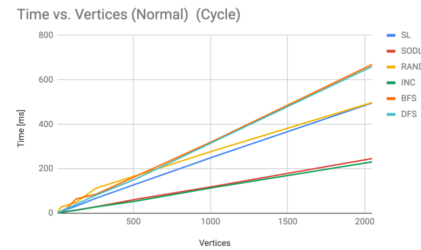
6.2 Time vs. Vertices

6.2.1 Cycle

As the number of vertices in a cycle increases, all 6 vertex ordering implementations' runtimes grow linearly as seen below:







The following table further highlights the linear growth in runtime for these vertex ordering implementations. As the number of vertices grows by a factor of 2, the runtimes for the 6 vertex ordering implementations grow at a relatively constant rate:

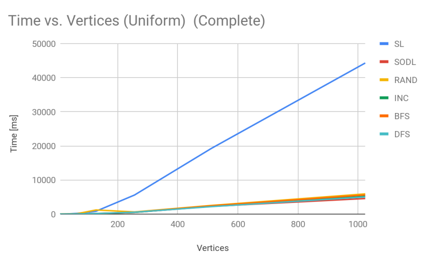
T [ms] vs Vertices; N: Factor Increase in V; T [ms]: Factor Increase in Time (Cycle)													
Uniform													
Edges	N	SL	T_SL	SODL	T_SODL	RAND	T_RAND	INC	T_INC	BFS	T_BFS	DFS	T_DFS
8		3		1		12		1		3		3	
16	2	5	1.67	2	2.00	28	2.33	2	2.00	7	2.33	6	2.00
32	2	9	1.80	6	3.00	30	1.07	4	2.00	12	1.71	14	2.33
64	2	18	2.00	10	1.67	38	1.27	10	2.50	23	1.92	24	1.71
128	2	35	1.94	17	1.70	53	1.39	16	1.60	48	2.09	45	1.88
256	2	71	2.03	34	2.00	84	1.58	33	2.06	91	1.90	87	1.93
512	2	134	1.89	77	2.26	140	1.67	60	1.82	172	1.89	170	1.95
1024	2	259	1.93	120	1.56	289	2.06	126	2.10	338	1.97	320	1.88
2048	2	569	2.20	243	2.03	486	1.68	234	1.86	674	1.99	641	2.00
Skewed													
8		2		1		7		1		3		3	
16	2	4	2.00	2	2.00	16	2.29	1	1.00	5	1.67	5	1.67
32	2	8	2.00	5	2.50	27	1.69	3	3.00	11	2.20	10	2.00
64	2	16	2.00	7	1.40	32	1.19	7	2.33	21	1.91	20	2.00
128	2	32	2.00	14	2.00	47	1.47	15	2.14	41	1.95	39	1.95
256	2	64	2.00	31	2.21	78	1.66	29	1.93	85	2.07	81	2.08
512	2	128	2.00	59	1.90	138	1.77	63	2.17	169	1.99	156	1.93
1024	2	250	1.95	122	2.07	267	1.93	128	2.03	330	1.95	320	2.05
2048	2	503	2.01	258	2.11	520	1.95	258	2.02	657	1.99	643	2.01
Normal													
8		2		1		4		1		3		2	
16	2	4	2.00	1	1.00	11	2.75	2	2.00	5	1.67	5	2.50
32	2	8	2.00	3	3.00	28	2.55	3	1.50	10	2.00	10	2.00
64	2	16	2.00	7	2.33	35	1.25	7	2.33	21	2.10	20	2.00
128	2	32	2.00	14	2.00	49	1.40	13	1.86	64	3.05	40	2.00
256	2	67	2.09	29	2.07	112	2.29	28	2.15	85	1.33	81	2.03

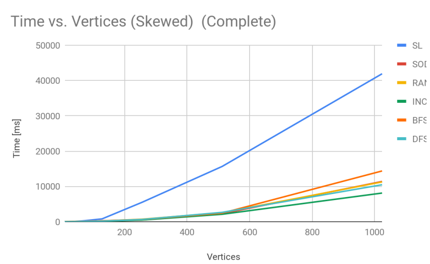
512	2	130	1.94	62	2.14	168	1.50	54	1.93	165	1.94	153	1.89
1024	2	255	1.96	121	1.95	282	1.68	116	2.15	327	1.98	323	2.11
2048	2	496	1.95	246	2.03	498	1.77	231	1.99	669	2.05	660	2.04

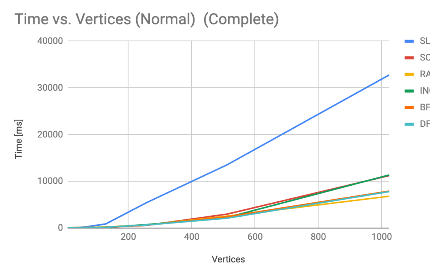
The table also shows that distribution of vertices plays little part in affecting the vertex ordering implementations' runtimes for cycles.

6.2.2 Complete

As the number of vertices in a complete graph increases, all 6 vertex ordering implementations' runtimes grow linearly as seen below:







The following table further highlights the linear growth in runtime for these vertex ordering implementations. As the number of vertices grows by a factor of 2, the runtimes for the 6 vertex ordering implementations grow at a relatively constant rate:

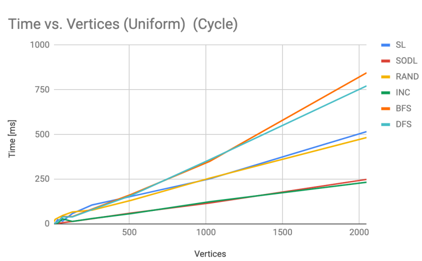
T [ms] vs Vertices; N: Factor Increase in V; T [ms]: Factor Increase in Time (Complete)													
Uniform													
Edges	N	SL	T_SL	SODL	T_SODL	RAND	T_RAND	INC	T_INC	BFS	T_BFS	DFS	T_DFS
8		6		3		23		3		5		4	
16	2	21	3.50	6	2.00	44	1.91	5	1.67	9	1.80	9	2.25
32	2	62	2.95	17	2.83	48	1.09	16	3.20	23	2.56	23	2.56
64	2	213	3.44	45	2.65	116	2.42	44	2.75	57	2.48	56	2.43
128	2	850	3.99	148	3.29	317	2.73	146	3.32	183	3.21	183	3.27
256	2	5595	6.58	506	3.42	643	2.03	503	3.45	550	3.01	558	3.05
512	2	19275	3.45	2272	4.49	2548	3.96	2353	4.68	2433	4.42	2197	3.94
1024	2	44310	2.30	4570	2.01	5935	2.33	5221	2.22	5624	2.31	5033	2.29
Skewed													
8		6		3		24		3		4		4	
16	2	17	2.83	6	2.00	30	1.25	5	1.67	10	2.50	11	2.75
32	2	59	3.47	24	4.00	47	1.57	17	3.40	25	2.50	23	2.09
64	2	219	3.71	46	1.92	103	2.19	43	2.53	56	2.24	56	2.43
128	2	813	3.71	266	5.78	210	2.04	147	3.42	264	4.71	186	3.32
256	2	4513	5.55	520	1.95	720	3.43	480	3.27	626	2.37	645	3.47

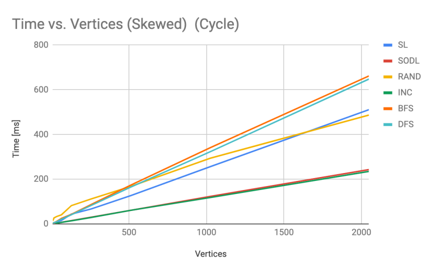
512	2	15658	3.47	2270	4.37	2561	3.56	2113	4.40	2441	3.90	2656	4.12
1024	2	41909	2.68	11410	5.03	11274	4.40	8127	3.85	14386	5.89	10480	3.95
Normal													
8		6		3		36		4		7		5	
16	2	28	4.67	6	2.00	56	1.56	13	3.25	9	1.29	9	1.80
32	2	98	3.50	17	2.83	110	1.96	36	2.77	30	3.33	25	2.78
64	2	233	2.38	46	2.71	117	1.06	44	1.22	59	1.97	60	2.40
128	2	864	3.71	150	3.26	244	2.09	144	3.27	183	3.10	171	2.85
256	2	5381	6.23	564	3.76	652	2.67	721	5.01	601	3.28	629	3.68
512	2	13548	2.52	2988	5.30	2550	3.91	2269	3.15	2407	4.00	2121	3.37
1024	2	32792	2.42	11244	3.76	6810	2.67	11364	5.01	7912	3.29	7812	3.68

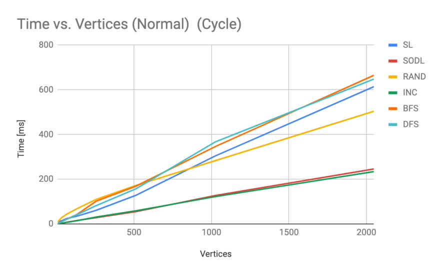
The table also shows that distribution of vertices plays little part in affecting the vertex ordering implementations' runtimes for complete graphs.

6.2.3 Random

As the number of vertices in a randomly-generated graph increases, all 6 vertex ordering implementations' runtimes grow linearly as seen below:







The following table further highlights the linear growth in runtime for these vertex ordering implementations. As the number of vertices grows by a factor of 2, the runtimes for the 6 vertex ordering implementations grow at a relatively constant rate:

T [ms] vs Vertices; N: Factor Increase in V; T [ms]: Factor Increase in Time (Random)													
Uniform													
Edges	N	SL	T_SL	SODL	T_SODL	RAND	T_RAND	INC	T_INC	BFS	T_BFS	DFS	T_DFS
8		3		1		13		1		3		3	
16	2	4	1.33	2	2.00	26	2.00	3	3.00	5	1.67	5	1.67
32	2	7	1.75	3	1.50	33	1.27	7	2.33	11	2.20	11	2.20
64	2	16	2.29	7	2.33	47	1.42	14	2.00	41	3.73	39	3.55
128	2	58	3.63	14	2.00	67	1.43	28	2.00	41	1.00	39	1.00
256	2	106	1.83	29	2.07	77	1.15	30	1.07	83	2.02	80	2.05
512	2	154	1.45	61	2.10	132	1.71	58	1.93	165	1.99	159	1.99

1024	2	252	1.64	117	1.92	256	1.94	124	2.14	350	2.12	359	2.26
2048	2	516	2.05	249	2.13	483	1.89	233	1.88	845	2.41	772	2.15
Skewed													
8		3		1		14		1		3		3	
16	2	4	1.33	2	2.00	27	1.93	2	2.00	5	1.67	5	1.67
32	2	8	2.00	3	1.50	33	1.22	3	1.50	11	2.20	10	2.00
64	2	16	2.00	7	2.33	41	1.24	7	2.33	25	2.27	21	2.10
128	2	44	2.75	15	2.14	82	2.00	14	2.00	43	1.72	41	1.95
256	2	67	1.52	30	2.00	112	1.37	29	2.07	87	2.02	83	2.02
512	2	127	1.90	61	2.03	168	1.50	61	2.10	174	2.00	165	1.99
1024	2	256	2.02	123	2.02	294	1.75	118	1.93	341	1.96	323	1.96
2048	2	511	2.00	243	1.98	487	1.66	235	1.99	662	1.94	648	2.01
Normal													
8		8		1		11		1		3		3	
16	2	11	1.38	2	2.00	15	1.36	2	2.00	5	1.67	5	1.67
32	2	14	1.27	3	1.50	28	1.87	3	1.50	10	2.00	10	2.00
64	2	24	1.71	7	2.33	43	1.54	7	2.33	21	2.10	20	2.00
128	2	34	1.42	15	2.14	67	1.56	14	2.00	41	1.95	40	2.00
256	2	61	1.79	28	1.87	110	1.64	31	2.21	102	2.49	82	2.05
512	2	128	2.10	55	1.96	172	1.56	59	1.90	169	1.66	157	1.91
1024	2	304	2.38	127	2.31	283	1.65	122	2.07	346	2.05	367	2.34
2048	2	614	2.02	246	1.94	504	1.78	234	1.92	665	1.92	648	1.77

The table also shows that distribution of vertices plays little part in affecting the vertex ordering implementations' runtimes for randomly-generated graphs.

In sum, my Smallest-Last, Smallest Original Degree Last, Random, Incremental, Breadth First Search, and Depth First Search vertex ordering implementations' runtimes grow linearly with the number of vertices for cyclic, complete, and randomly-generated graphs. This empirical finding remains consistent with the time complexity predictions made in *5.7 Ordering Predictions Overview.*

7. Coloring Runtime

7.1 Time Complexity Prediction

The greedy algorithm implemented for coloring a graph given the order of vertices determined by our 6 vertex ordering algorithms iterates through every vertex in a given graph and assigns the lowest possible color to each vertex by examining the colors assigned to that vertex's adjacent vertices and assigning the lowest unassigned color to that vertex. For example, if vertex v had 3 adjacent vertices that had the color values 1, 3, and 5, given that 1 is the lowest possible color, the lowest unassigned color value of 2 would be assigned to vertex v. Since our greedy coloring algorithm involves:

1. Visiting every vertex in a given graph
2. Visiting every vertex's adjacent vertices, meaning visiting every edges of a given graph

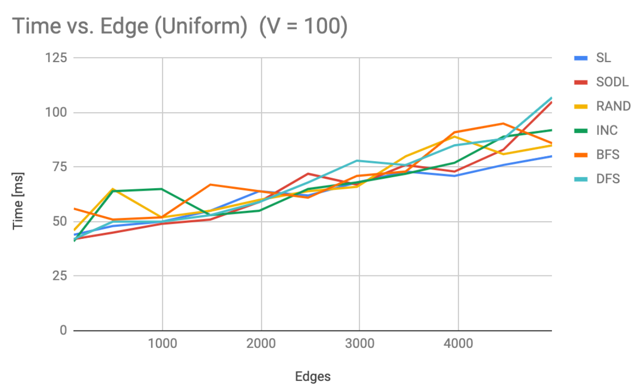
I expect my greedy coloring algorithm's implementation to exhibit a lower-bound running time of $\Omega(V+E)$.

7.2 Time vs. Edges

7.2.1 Uniform

In accordance with the above prediction, for a graph consisting of 100 vertices whose edges are generated from a collection of uniformly distributed vertices, the time complexity for our greedy coloring implementation increases linearly as the number of edges increases for all vertex ordering generated by our 6 vertex ordering algorithms.

Time [ms] vs Edges (Uniform) (V = 100)						
Edges	SL	SODL	RAND	INC	BFS	DFS
100	44	42	46	41	56	42
495	48	45	65	64	51	50
990	50	49	52	65	52	50
1485	55	51	55	53	67	53
1980	64	59	60	55	64	59
2475	62	72	64	65	61	68
2970	68	67	66	68	71	78
3465	73	76	80	72	73	76
3960	71	73	89	77	91	85
4455	76	83	81	89	95	88
4950	80	105	85	92	86	107



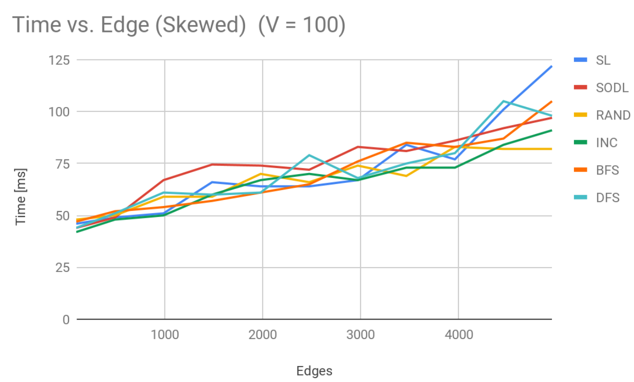
The following table highlights the greedy coloring implementations' linear runtime growth for these 6 algorithms:

N: Factor Increase in Edges; T [ms]: Factor Increase in Time (Uniform) (V = 100)													
Edges	N	SL	T_SL	SODL	T_SODL	RAND	T_RAND	INC	T_INC	BFS	T_BFS	DFS	T_DFS
495		48		45		65		64		51		50	
990	2	50	1.24	49	1.39	52	1.10	65	1.02	52	1.02	50	1.00
1980	2	64	1.28	59	1.20	60	1.15	55	0.85	64	1.23	59	1.18
3960	2	71	1.11	73	1.24	89	1.48	77	1.40	91	1.42	85	1.44

7.2.2 Skewed

For a graph consisting of 100 vertices whose edges are generated from a collection of skewed distribution of vertices, the time complexity for the greedy coloring algorithm increases linearly as the number of edges increases for all vertex orders generated by our 6 vertex ordering algorithms.

Time [ms] vs Edges (Skewed) (V = 100)						
Edges	SL	SODL	RAND	INC	BFS	DFS
100	46	44	48	42	47	44
495	49	49	50	48	52	51
990	51	67	59	50	54	61
1485	66	69.5	59	60	57	60
1980	64	74	70	67	61	61
2475	64	72	66	70	65	79
2970	67	83	74	67	76	68
3465	84	85	69	73	85	75
3960	77	86	83	73	83	80
4455	101	92	82	84	87	105
4950	122	97	82	91	105	98



The following table highlights this linear growth in runtime complexity. As the number of edges grows linearly, so does the coloring algorithm's runtime for all 6 vertex ordering algorithms:

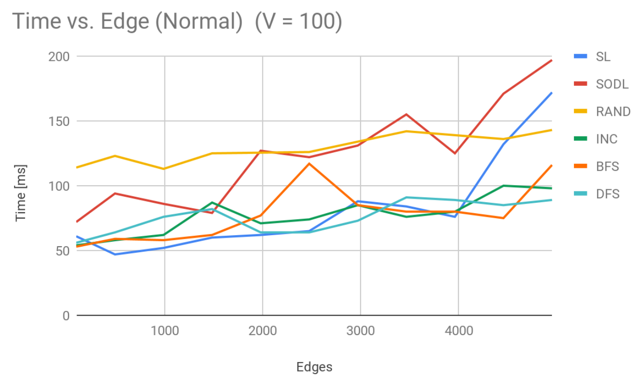
N: Factor Increase in Edges; T [ms]: Factor Increase in Time (Skewed) (V = 100)													
Edges	N	SL	T_SL	SODL	T_SODL	RAND	T_RAND	INC	T_INC	BFS	T_BFS	DFS	T_DFS
495		49		49		50		48		52		51	
990	2	51	1.24	67	1.67	59	1.48	50	1.04	54	1.04	61	1.20
1980	2	64	1.25	74	1.10	70	1.19	67	1.34	61	1.13	61	1.00

3960	2	77	1.20	86	1.16	83	1.19	73	1.09	83	1.36	80	1.31

7.2.3 Normal

For a graph consisting of 100 vertices whose edges are generated from a collection of normally distributed vertices, the time complexity for the greedy coloring algorithm increases linearly as the number of edges increases for all vertex ordering algorithms.

Time [ms] vs Edges (Normal) (V = 100)						
Edges	SL	SODL	RAND	INC	BFS	DFS
100	61	72	114	54	53	56
495	47	94	123	58	59	64
990	52	86	113	62	58	76
1485	60	79	125	87	62	82
1980	62	137	126	71	77	64
2475	65	122	126	74	117	64
2970	88	131	134	85	85	73
3465	84	155	142	76	80	91
3960	76	125	139	80	80	89
4455	132	171	136	100	75	85
4950	172	197	143	98	116	89



The following table highlights this linear growth in runtime. As the number of edges grows linearly, so does the coloring algorithm's runtime for all 6 vertex ordering algorithms:

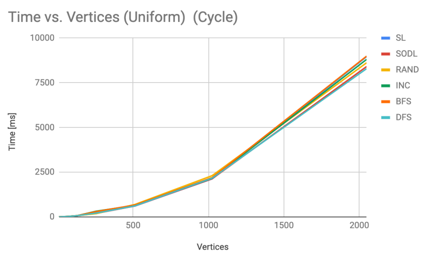
N: Factor Increase in Edges; T [ms]: Factor Increase in Time (Normal) (V = 100)													
Edges	N	SL	T_SL	SODL	T_SODL	RAND	T_RAND	INC	T_INC	BFS	T_BFS	DFS	T_DFS
495		47		94		123		58		59		64	
990	2	52	1.31	86	1.21	113	1.22	62	1.07	58	0.98	76	1.19
1980	2	62	1.19	127	1.48	126	1.11	71	1.15	77	1.33	64	0.84
3960	2	76	1.23	135	1.06	134	1.07	80	1.13	80	1.04	89	1.39

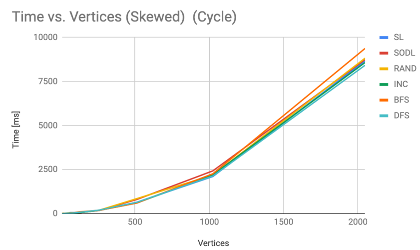
In sum, the vertex distributions had little effect on the greedy coloring implementation's runtime performance. The coloring runtimes for Smallest-Last, Smallest Original Degree Last, Random, Incremental, Breadth First Search, and Depth First Search implementations grew linearly with the growing number of edges in a graph. These findings remain consistent with the predictions made in the earlier section, *6.1 Time Complexity Prediction*.

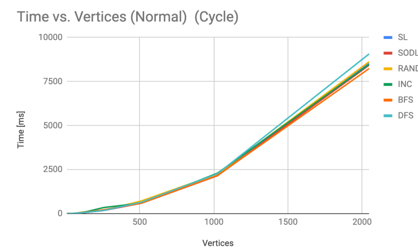
7.3 Time vs. Vertices

7.3.1 Cycle

As the number of vertices in a cycle graph increases, the coloring implementation's runtime increases quadratically as seen in the following graphs:







The table below further highlights this relation. As the number of vertices grow at a constant rate, the rate of increase of the coloring algorithm's runtime for a cycle graph steadily increases for all 6 vertex ordering algorithms:

T [ms] vs Vertices N: Factor Increase in V; T [ms]: Factor Increase in Time (Cycle)													
						Uniform							
Vertices	N	SL	T_SL	SODL	T_SODL	RAND	T_RAND	INC	T_INC	BFS	T_BFS	DFS	T_DFS
8		3		2		3		2		2		2	
16	2	4	1.33	4	2.00	5	1.67	4	2.00	4	2.00	4	2.00
32	2	9	2.25	12	3.00	14	2.80	11	2.75	11	2.75	11	2.75
64	2	25	2.78	25	2.08	26	1.86	25	2.27	25	2.27	25	2.27
128	2	66	2.64	66	2.64	68	2.62	67	2.68	67	2.68	66	2.64
256	2	230	3.48	201	3.05	186	2.74	320	4.78	301	4.49	214	3.24
512	2	614	2.67	614	3.05	698	3.75	638	1.99	667	2.22	618	2.89
1024	2	2130	3.47	2134	3.48	2308	3.31	2162	3.39	2196	3.29	2168	3.51
2048	2	8970	4.21	8410	3.94	8619	3.73	8810	4.07	8992	4.09	8292	3.82
						Skewed							
8		2		2		2		2		2		2	
16	2	4	2.00	4	2.00	5	2.50	4	2.00	4	2.00	3	1.50
32	2	10	2.50	10	2.50	16	3.20	9	2.25	11	2.75	11	3.67
64	2	63	6.30	38	3.80	51	3.19	24	2.67	23	2.09	24	2.18
128	2	62	0.98	62	1.63	63	1.24	63	2.63	102	4.43	78	3.25
256	2	180	2.90	185	2.98	187	2.97	185	2.94	185	1.81	197	2.53
512	2	647	3.59	806	4.36	851	4.55	606	3.28	604	3.26	643	3.26
1024	2	2111	3.26	2429	3.01	2199	2.58	2207	3.64	2264	3.75	2134	3.32

2048	2	8682	4.11	8746	3.60	8824	4.01	8580	3.89	9379	4.14	8423	3.95
Normal													
8		2		2		2		2		2		2	
16	2	4	2.00	4	2.00	4	2.00	4	2.00	4	2.00	3	1.50
32	2	10	2.50	10	2.50	11	2.75	10	2.50	11	2.75	10	3.33
64	2	24	2.40	24	2.40	31	2.82	24	2.40	24	2.18	24	2.40
128	2	65	2.71	63	2.63	103	3.32	63	2.63	67	2.79	64	2.67
256	2	181	2.78	181	2.87	184	1.79	178	2.83	222	3.31	180	2.81
512	2	604	3.34	595	3.29	727	3.95	597	3.35	597	2.69	652	3.62
1024	2	2158	3.57	2151	3.62	2250	3.09	2297	3.85	2163	3.62	2265	3.47
2048	2	8537	3.96	8435	3.92	8632	3.84	8479	3.69	8243	3.81	9075	4.01

The consistent growth in rate of change observed among the cycles generated from our uniform, skewed, and normal vertex distributions suggests that for a cycle, uniform, skewed, and normal vertex distributions do not play a significant role in affecting the runtime complexities of coloring a cycle using our greedy coloring and 6 vertex ordering implementations.

Our greedy coloring implementation's quadratic time complexity with the number of vertices for a cycle graph remains consistent with *6.1 Time Complexity Prediction*'s prediction of anticipating the graph coloring implementation to exhibit a lower-bound complexity of Ω(V+E).

For a cycle, the relation between the number of edges and vertices (stated in *4.1 Cycles*) is:

$$\#\ of\ edges\ in\ a\ cycle = \#\ of\ vertices$$

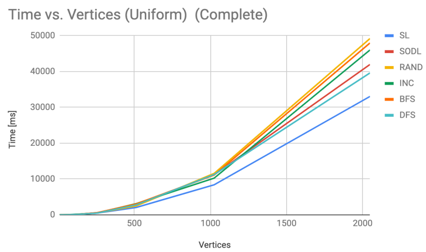
Thus, for a graph with a given number of vertices V and edges E, the relation between E and V is:

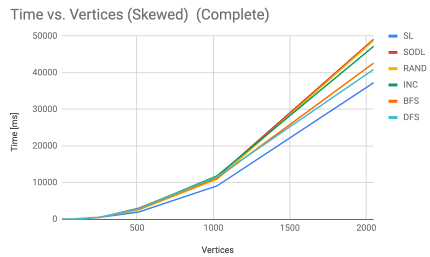
$$E = V$$

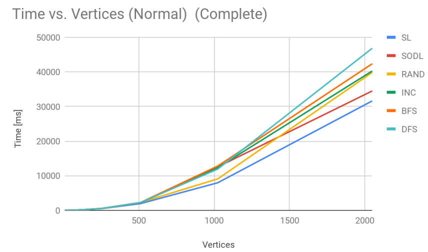
Since E and V share this linear relation, the predicted lower-bound complexity Ω(V+E) for a complete graph can be re-written as Ω(V+V), or simply Ω(V). Because the empirical findings show that our greedy coloring implementation's runtime is quadratic to the number of vertices in a cycle, having Ω(V) as the lower bound for our greedy coloring implementation's runtime complexity remains consistent with our empirical findings.

7.3.2 Complete

As the number of vertices in a complete graph increases, the coloring implementation's runtime increases quadratically as seen in the following graphs:







The table below further highlights this relation. As the number of vertices grow at a constant rate, the rate of increase of the coloring algorithm's runtime steadily increases for all 6 vertex ordering algorithms:

T [ms] vs Vertices; N: Factor Increase in V; T [ms]: Factor Increase in Time (Complete)													
Uniform													
Vertices	N	SL	T_SL	SODL	T_SODL	RAND	T_RAND	INC	T_INC	BFS	T_BFS	DFS	T_DFS
8		2		2		3		2		2		2	
16	2	4	2.00	4	2.00	5	1.67	4	2.00	6	3.00	4	2.00
32	2	13	3.25	14	3.50	13	2.60	16	4.00	14	2.33	13	3.25
64	2	39	3.00	38	2.71	39	3.00	63	3.94	38	2.71	38	2.92
128	2	123	3.15	124	3.79	124	3.18	185	2.94	126	3.32	127	3.34
256	2	488	3.97	491	3.96	512	4.13	513	2.77	503	3.99	488	3.84
512	2	1961	4.02	2026	4.13	2178	4.25	1829	3.57	2041	4.06	2079	4.26
1024	2	7980	4.07	9186	4.53	10062	4.62	7541	4.12	9280	4.55	9262	4.46
2048	2	33054	4.14	44969	4.90	49190	4.89	40015	5.31	46966	5.06	43605	4.71
Skewed													
8		5		4		5		4		4		4	
16	2	7	1.40	7	1.75	7	1.40	7	1.75	7	1.75	8	2.00
32	2	23	3.29	22	3.14	23	3.29	21	3.00	23	3.29	23	2.88
64	2	41	1.78	43	1.95	65	2.83	42	2.00	64	2.78	60	2.61
128	2	131	3.20	153	3.56	156	2.40	161	3.83	194	3.03	172	2.87
256	2	511	3.90	495	3.24	542	3.47	514	3.19	461	2.38	513	2.98
512	2	1987	3.89	1927	3.89	2296	4.24	1887	3.67	1959	4.25	1858	3.62
1024	2	7324	4.59	9624	4.99	9948	4.33	9149	4.85	9260	4.73	7607	4.09
2048	2	37283	5.09	49150	5.11	49827	5.01	47235	5.16	47642	5.14	38911	5.12
Normal													
8		5		4		5		13		4		4	
16	2	8	1.60	5	1.25	6	1.20	5	0.38	5	1.25	5	1.25
32	2	16	2.00	16	3.20	17	2.83	17	3.40	16	3.20	16	3.20

64	2	44	2.75	45	2.81	53	3.12	45	2.65	44	2.75	44	2.75
128	2	153	3.48	162	3.60	172	3.25	176	3.91	165	3.75	149	3.39
256	2	504	3.29	531	3.28	560	3.26	529	3.01	534	3.24	551	3.70
512	2	1948	3.87	1802	3.39	2272	4.06	1796	3.40	2265	4.24	2260	4.10
1024	2	7970	4.09	7049	3.91	9584	4.22	7325	4.08	9803	4.33	9930	4.39
2048	2	33611	4.22	34490	4.89	40826	4.26	40253	5.50	44348	4.52	46816	4.71

The consistent growth in rate of change observed among the complete graphs generated from our uniform, skewed, and normal vertex distributions suggests that for a complete graph, uniform, skewed, and normal vertex distributions do not play a significant role in affecting the runtime complexities of coloring a complete graph using our greedy coloring and 6 vertex ordering implementations. This observation remains consistent with our understanding of a complete graph. Given that the set of unique vertices generated from our vertex distributions are the same, a complete graph generated from one distribution should be identical a graph generated from another since, regardless of the distributions, all vertices in a complete graph are directly adjacent to one another, meaning as long as the set of unique vertices between one complete graph and another remain the same, the two graphs necessarily must be the same.

Our greedy coloring implementation's quadratic time complexity with the number of vertices for a complete graph remains consistent with *6.1 Time Complexity Prediction*'s prediction of anticipating the graph coloring implementation to exhibit a lower-bound complexity of Ω(V+E).

For a complete graph, the relation between the number of edges and vertices (stated in *4.2 Complete*) is:

*# of edges in a complete graph = # of vertices * (# of vertices - 1) / 2*

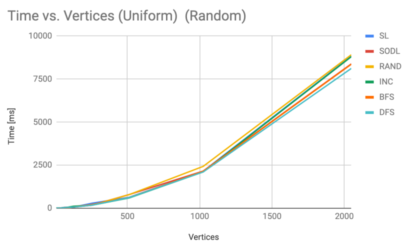
Thus, for a graph with a given number of vertices V and edges E, the relation between E and V stands as:

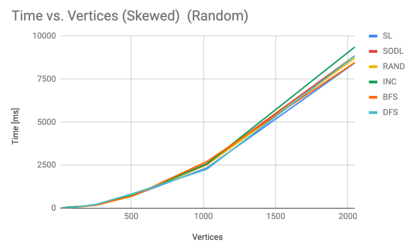
$$E = V * (V - 1) / 2$$

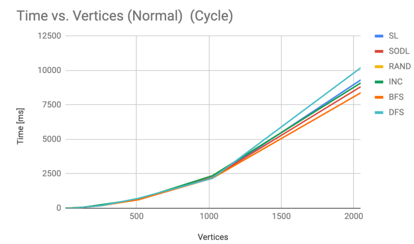
Since E and V share this quadratic relation, the predicted lower-bound complexity Ω(V+E) for a complete graph can be re-written as Ω(V+V^2), or simply Ω(V^2). Thus, the quadratic relation found in our empirical measurements between our greedy coloring implementation's runtime and the number of vertices remains consistent with our expected lower-bound complexity for this implementation.

7.3.3 Random

As the number of vertices in a randomly-generated graph increases, the coloring implementation's runtime increases quadratically as seen in the following graphs:







The table below further highlights this relation. As the number of vertices grow at a constant rate, the rate of increase of the coloring algorithm's runtime steadily increases for all 6 vertex ordering algorithms:

T [ms] vs Vertices N: Factor Increase in V; T [ms]: Factor Increase in Time (Random)													
Uniform													
Vertices	N	SL	T_SL	SODL	T_SODL	RAND	T_RAND	INC	T_INC	BFS	T_BFS	DFS	T_DFS
8		2		2		2		2		2		2	
16	2	4	2.00	4	2.00	4	2.00	3	1.50	3	1.50	17	8.50
32	2	10	2.50	10	2.50	10	2.50	10	3.33	10	3.33	10	0.59
64	2	23	2.30	51	5.10	24	2.40	24	2.40	23	2.30	44	4.40
128	2	62	2.70	63	1.24	66	2.75	116	4.83	64	2.78	64	1.45
256	2	190	3.06	179	2.84	182	2.76	180	1.55	188	2.94	180	2.81
512	2	633	3.33	492	2.75	587	3.23	597	3.32	622	3.31	614	3.41
1024	2	2150	3.40	1858	3.78	2430	4.14	2116	3.54	2147	3.45	2108	3.43
2048	2	8832	4.11	8381	4.51	8912	3.67	8806	4.16	8362	3.89	8121	3.85
Skewed													
8		3		2		2		2		2		6	
16	2	4	1.33	4	2.00	4	2.00	3	1.50	4	2.00	5	0.83
32	2	14	3.50	31	7.75	29	7.25	28	9.33	10	2.50	10	2.00
64	2	23	1.64	64	2.06	62	2.14	64	2.29	24	2.40	23	2.30
128	2	94	4.09	93	1.45	95	1.53	94	1.47	71	2.96	79	3.43
256	2	176	1.87	180	1.94	183	1.93	180	1.91	181	2.55	221	2.80
512	2	627	3.56	625	3.47	640	3.50	657	3.65	718	3.97	654	2.96
1024	2	2346	3.74	2601	4.16	2514	3.93	2566	3.91	2720	3.79	2266	3.46
2048	2	8447	3.60	8856	3.40	8728	3.47	9370	3.65	8466	3.11	8855	3.91
Normal													
8		4		2		3		2		2		3	
16	2	4	1.00	6	3.00	5	1.67	4	2.00	17	8.50	4	1.33
32	2	11	2.75	10	1.67	11	2.20	10	2.50	10	0.59	10	2.50

64	2	33	3.00	29	2.90	46	4.18	24	2.40	24	2.40	24	2.40
128	2	65	1.97	64	2.21	67	1.46	64	2.67	63	2.63	65	2.71
256	2	185	2.85	188	2.94	182	2.72	199	3.11	186	2.95	186	2.86
512	2	598	3.23	565	3.01	633	3.48	652	3.28	614	3.30	617	3.32
1024	2	2186	3.66	2227	3.94	2364	3.73	2357	3.62	2180	3.55	2177	3.53
2048	2	9314	4.26	8825	3.96	9092	3.85	9098	3.86	8379	3.84	10195	4.68

The consistent growth in rate of change observed among the randomly-generated graphs created from our uniform, skewed, and normal vertex distributions suggests that for a randomly-generated graph, uniform, skewed, and normal vertex distributions do not play a significant role in affecting the runtime complexities of coloring a randomly-generated graph using our greedy coloring and 6 vertex ordering implementations.

Our greedy coloring implementation's quadratic time complexity with the number of vertices for a randomly-generated graph remains consistent with *6.1 Time Complexity Prediction*'s prediction of anticipating the graph coloring implementation to exhibit a lower-bound complexity of Ω(V+E).

For a randomly-generated graph, the relation between the nuber of edges and vertices (stated in *4.3 Random*) is:

*# of edges in a randomly-generated graph = density factor * # of vertices * (# of vertices - 1) / 2*

where .1 ≤ density factor ≤ .9 and increments by .1

Thus, for a graph with a given number of vertices V and edges E, the relation between E and V stands as:

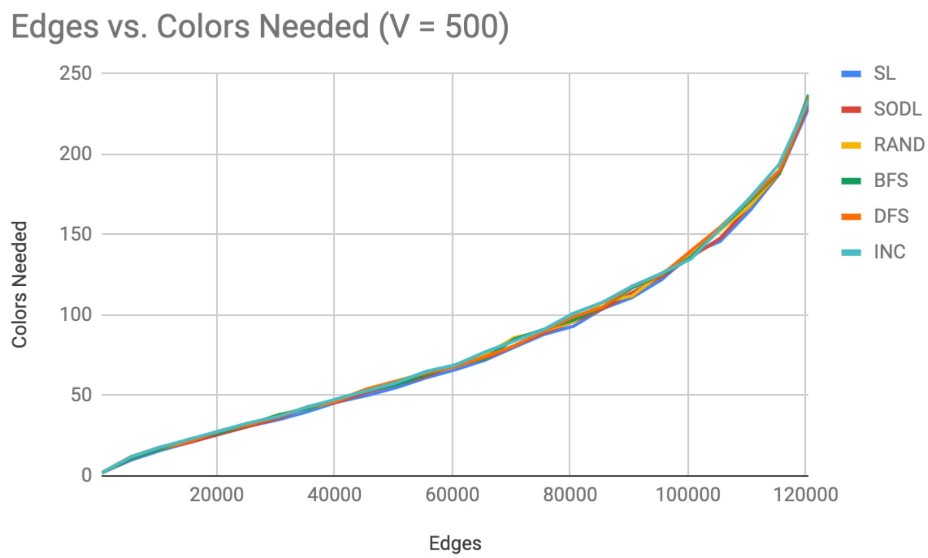
$$E = \text{density factor} * V * (V - 1) / 2$$

Since E and V share this quadratic relation, the predicted lower-bound complexity Ω(V+E) for a randomly-generated graph can be re-written as Ω(V+V^2), or simply Ω(V^2). Thus, the quadratic relation found in our empirical measurements between our greedy coloring implementation's runtime and the number of vertices remains consistent with our expected lower-bound complexity for this implementation.

In sum, our empirical findings show that our greedy graph coloring implementation demonstrates quadratic runtime complexity in relation to the number of vertices in cycles, complete graphs, and randomly-generated graphs generated from a collection of vertices that follow uniform, skewed, and normal distributions. This finding remains consistent with *6.1 Time Complexity Prediction*'s expectation for the greedy coloring implementation to exhibit a lower-bound running time of Ω(V+E).

8. Colors Needed

Given a graph with 500 vertices, the following plot depicts the relation between the number of edges and the number of colors needed to color a graph in accordance to our 6 vertex ordering implementations.



Plot-wise, the 6 vertex ordering implementations perform very similarly in terms of generating a vertex ordering that minimizes the number of distinct colors needed to color a graph. To pinpoint their differences, the following table shows the number of distinct colors needed to color a randomly-generated 500-vertices graph with a given amount of edges according to our 6 vertex ordering implementations. The final row depicts the averaged number of colors needed for each vertex ordering implementation.

Density vs. Colors Needed (V = 500)							
# of Edges	Density (# of Edges / Max # of Edges)	SL	SODL	RAND	BFS	DFS	INC
500	0.40%	2	2	2	2	2	2
5500	4.41%	10	11	12	11	12	12
10500	8.42%	16	17	17	17	18	18
15500	12.42%	21	22	22	23	22	23
20500	16.43%	26	26	28	27	28	28
25500	20.44%	31	31	32	32	32	33
30500	24.45%	35	36	38	38	37	37
35500	28.46%	40	43	42	42	43	43
40500	32.46%	46	46	47	48	47	48
45500	36.47%	50	52	54	53	54	53

50500	40.48%	55	57	59	57	59	58
55500	44.49%	61	62	64	63	64	65
60500	48.50%	66	69	68	69	68	69
65500	52.51%	72	73	75	73	74	77
70500	56.51%	80	81	86	85	81	84
75500	60.52%	88	89	90	91	90	91
80500	64.53%	93	96	96	97	99	101
85500	68.54%	104	104	108	105	105	108
90500	72.55%	111	114	112	117	118	118
95500	76.55%	122	124	126	125	125	126
100500	80.56%	138	136	137	136	140	135
105500	84.57%	146	148	153	154	155	154
110500	88.58%	165	168	168	171	172	173
115500	92.59%	188	188	189	189	190	194
120500	96.59%	229	232	235	237	235	234
Avg # of Colors Needed		**79.8**	**81.08**	**82.4**	**82.48**	**82.8**	**83.36**

Although the difference in performance among the 6 vertex algorithms may be small, according to our empirical findings, Smallest-Last vertex ordering consistently creates an ordering that generates the fewest number of colors used (highlighted in orange). Thus, although the differences in performance among our 6 vertex algorithms may appear to be small, our empirical findings suggest that there are measurable differences in performance among our vertex algorithms' abilities to generate a vertex ordering that renders the fewest number of colors needed. Thus, we will use the average number of colors needed to rank the performance of our 6 vertex ordering implementations (from best to worst):

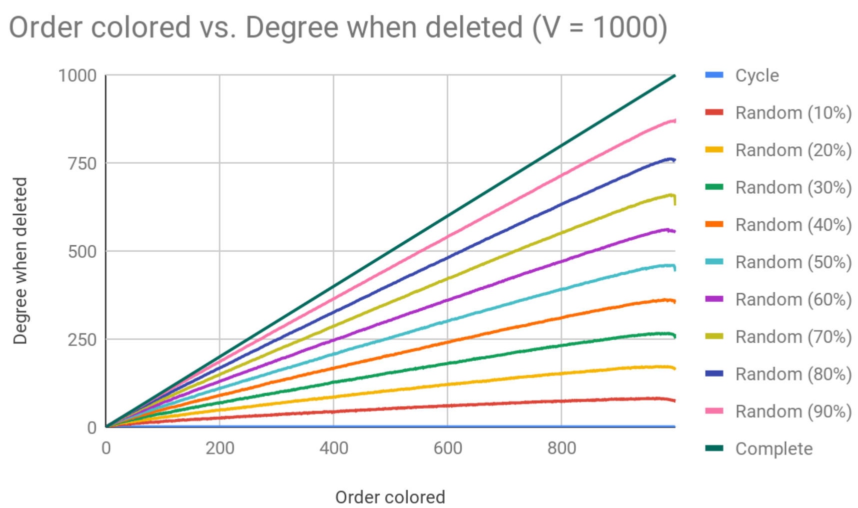
Average Colors Needed Ranking

1. Smallest-Last
2. Smallest Original Degree Last
3. Random
4. Breadth First Search
5. Depth First Search
6. Incremental

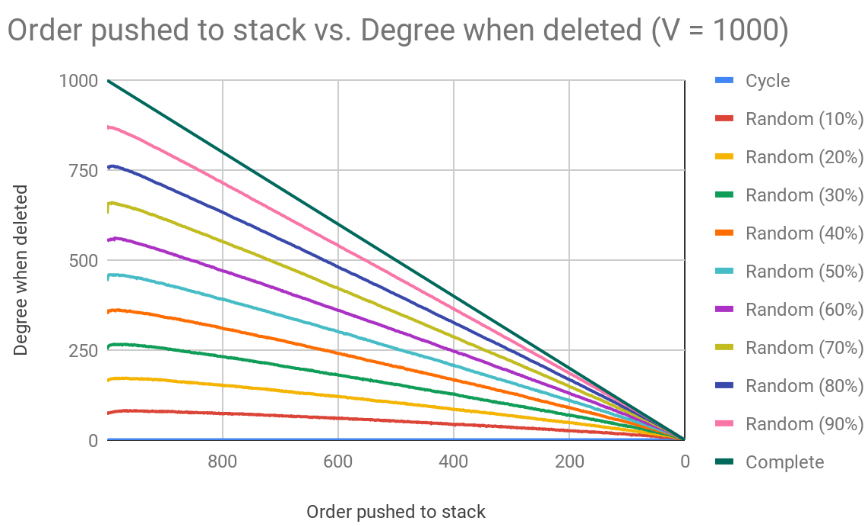
9. Smallest-Last In-Depth

9.1 Order vs. Degree When Deleted

The following plot depicts the relationship between order of vertices colored and their degrees at the time of deletion.



According to the Smallest-Last vertex ordering algorithm, the first vertex colored is the latest vertex pushed to the stack for ordering while the last vertex colored is the earliest vertex pushed onto the stack for vertex ordering. Since the ordering of vertices colored and the selection of vertices pushed to stack share an inverse relation, if we were to reverse the x-axis orientation of the above plot, we get the following relation:



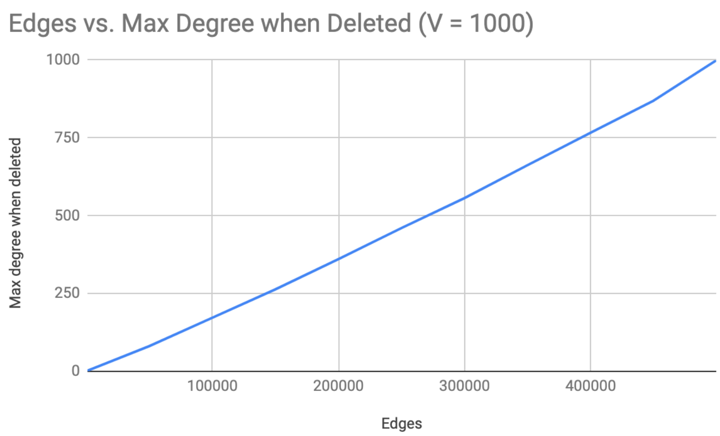
Notice that the degrees for a cycle's vertices at their time of deletion and being pushed to our stack for vertex ordering in *Order pushed to stack vs. Degree when deleted* plot is nearly flat. This empirical finding is consistent with our understanding of how the Smallest-Last vertex ordering algorithm works for a cycle since prior to any deletion, all degrees of a cycle's vertices are equal to 2. Yet as soon as one of its vertex of degree 2 is deleted and pushed to our stack, the cycle is then broken, at which point the graph becomes a single chain of vertices such that the vertices with minimum degrees are those at either ends of this chain with a degree of 1. Thus, as one of the either ends in our chain of vertices are iteratively deleted and pushed to our stack, the minimum degrees for all non-first and non-last vertices pushed to the stack are 1. Finally, when our graph is reduced to a single vertex. Since there are no other vertices remaining, it is pushed to our stack with a recorded degree of 0. Thus, the sequence of minimum degrees as a cycle's vertices are pushed to our stack should be 2, 1, 1, … 1, 0. Thus, the consistency between these nearly flat values, which we deduced from our understanding of how the Smallest-Last vertex ordering algorithm works for a cycle, and the nearly flat trend we observed for a cycle in the *Order pushed to stack vs. Degree when deleted* plot suggests that our Smallest-Last Vertex Ordering implementation is working as expected on a cyclic graph.

For a complete graph, in *Order pushed to stack vs. Degree when deleted* plot, notice that the degrees for its vertices at their time of deletion and being pushed to our stack for vertex ordering is identical to the relation: *degree when deleted = (# of vertices - 1) - order pushed to stack.* This empirical finding is consistent with our understanding of how the Smallest-Last vertex ordering algorithm works for a complete graph. Prior to any deletion, all degrees of a complete graph's vertices are equal to the number of vertices minus 1, which in our case of 1000 vertices is equal to 999. Yet as soon as one of its vertex of degree 999 is deleted and pushed to our stack, the vertex that has the minimum degree in our resultant graph is one of the vertices that was adjacent to the vertex that was deleted, meaning this adjacent vertex now has one less vertex it is connected to and thus has the degree 998. Thus, this iteration of deleting one adjacent vertex after another with their minimum degrees 997, 996, and so forth results in the sequence of minimum degrees 999, 998, 997, 996, … 3, 2, 1. Thus, the consistency between this sequence, which we deduced from our understanding of how the Smallest-Last vertex ordering algorithm works for a complete graph, and the *degree when deleted = (# of vertices - 1) - order pushed to stack* relation we observed for a complete graph in the *Order pushed to stack vs. Degree when deleted* plot suggests that our Smallest-Last Vertex Ordering implementation is working as expected on a complete graph.

Something interesting to note in *Order colored vs. Degree when deleted* plot is that as the density in our randomly-generated graph linearly increases, its slope becomes closer and closer to that of a complete graph and does so in a linear fashion that is proportional to the increase in density. Since the number of vertices in our graph is fixed at 1000, the only parameter that increases our graph's density is the number of edges. Thus, the linear relation observed between density and slope in our order colored vs degree when deleted relation suggests that, given that the number of vertices in a graph is fixed, the number of edges in a graph is directly proportional to the rate of change in the degrees of vertices when deleted as the order colored progresses for our Smallest-Last vertex ordering implementation.

9.2 Density vs. Degree When Deleted

To explore this relation between density and degrees further, the following plot depicts the relation between the number of edges in a 1000-vertices graph and the max degrees of vertices deleted from our Smallest-Last vertex ordering implementation:



Consistent to the findings from the previous section, the plot suggests that, given that the number of vertices remains fixed (which in our case is 1000), the number of edges and max degree when deleted from a graph share a linear relation.

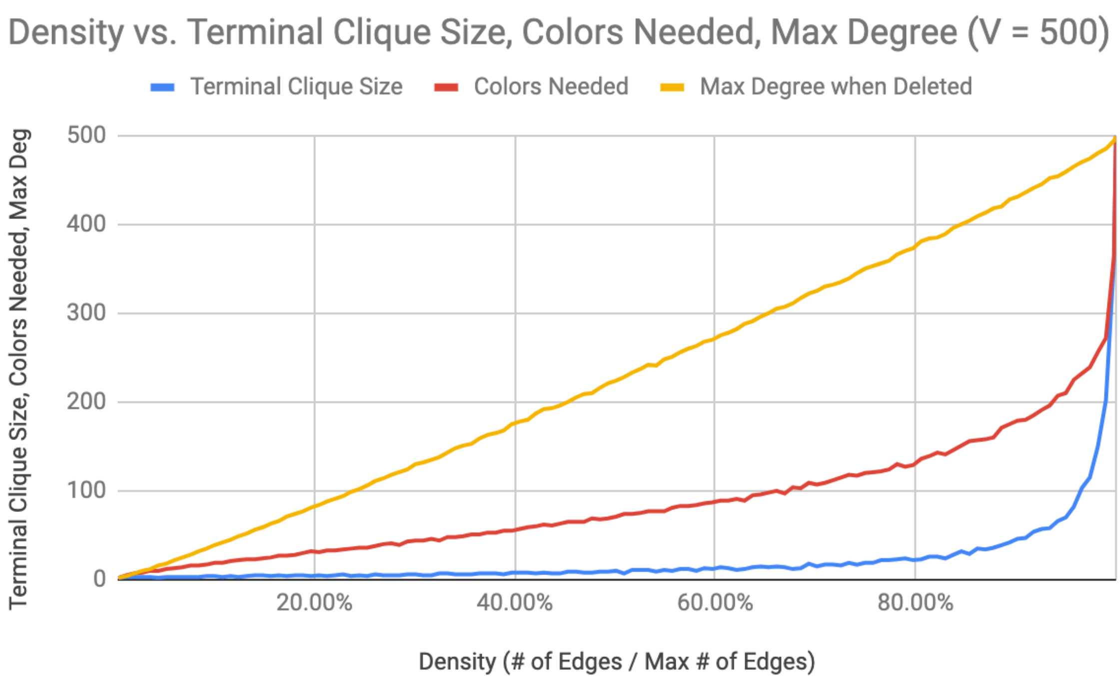
The table below depicts this more precisely:

Density vs. Max Degree When Deleted (V = 1000)					
Density Level	Graph Type	Edges	Factor E	Max Degree	Factor D
0	Cycle	1,000		2	
1	Random (10%)	49,950	49.95	81	40.50
2	Random (20%)	99,900	2.00	172	2.12
3	Random (30%)	149,850	1.50	263	1.53
4	Random (40%)	199,800	1.33	360	1.37
5	Random (50%)	249,750	1.25	460	1.28
6	Random (60%)	299,700	1.20	556	1.21
7	Random (70%)	349,650	1.17	662	1.19
8	Random (80%)	399,600	1.14	766	1.16
9	Random (90%)	449,550	1.13	869	1.13
10	Complete	499,500	1.11	999	1.15

As seen from this table, while the number of vertices in a graph remains constant, the factors by which the number of edges increase in our 1000-vertices graph nearly matches the factors by which the max degree increases. Thus, the density of a graph is directly proportional to the max degree that is deleted from a graph using our Smallest-Last vertex ordering implementation.

9.3 Density vs. Terminal Clique Size, Colors Needed, and Max Degree when Deleted

The following plot depicts the relation between graph density and its terminal clique size, number of colors needed, and max degree when deleted.



As seen from this plot, max degree when deleted appears to be greater than the number of colors needed, and the terminal clique size appears to be less than the number of colors needed.

The table below depicts these relations in greater detail:

Density vs. Max Degree When Deleted (V = 500)				
Density	Edges	Terminal Clique Size	Colors Needed	Max Degree when Deleted
0.40%	500	2	2	2
8.42%	10,500	3	16	32
16.43%	20,500	5	27	66
24.45%	30,500	5	36	102
32.46%	40,500	7	44	138
40.48%	50,500	8	57	178
48.50%	60,500	9	68	216
56.51%	70,500	12	83	256
64.53%	80,500	15	96	296

72.55%	90,500	16	115	335
80.56%	100,500	23	136	381
88.58%	110,500	39	171	420
96.59%	120,500	103	232	470
100.00%	124,750	500	500	499

As seen from this plot, max degree when deleted is greater than or equal to the number of colors needed for all numbers of edges except when the number of edges equals the maximum number of edges possible at which point the max degree when deleted is less than the terminal clique size and colors needed by 1. Thus, max degree when deleted + 1 is greater than or equal to the number of colors needed for all densities. Thus, by definition of upper bound, max degree when deleted + 1 serves as the upper bound for colors needed. Additionally, terminal clique size is less than or equal to the number of colors needed across all densities. Thus, by definition of lower bound, terminal clique size serves as the lower bound for colors needed.

10. Conclusion

10.1 Runtime Complexities

In conclusion, our 6 vertex ordering and 1 coloring implementations exhibit the following runtime complexities:

Algorithm Implemented	Time Complexity
Smallest-Last	Ω(V+E)
Smallest Original Degree Last	Ω(V)
Random	Ω(V)
Incremental	Ω(V)
Breadth First Search	Ω(V+E)
Depth First Search	Ω(V+E)
Greedy Coloring	Ω(V+E)

10.2 Vertex Distribution

My implemented vertex distributions – uniform, skewed, and normal – did not play a significant factor in affecting the runtime complexities of our vertex ordering and coloring implementations.

10.3 Colors Needed

Although the performance of our 6 vertex ordering implementations in terms of generating the minimum number of colors needed to color a graph were very similar, based on the average number of colors needed that was calculated from our empirical findings, we were able to rank from best to worst our vertex ordering implementations' ability to generate the fewest number of colors needed.

Average Colors Needed Ranking

7. Smallest-Last
8. Smallest Original Degree Last
9. Random
10. Breadth First Search
11. Depth First Search
12. Incremental

10.4 Smallest-Last Vertex Ordering In-Depth

10.4.1 Density vs. Degree when Deleted

A graph's density is directly proportional to the rate of change in degree when deleted as the order colored progresses. Furthermore, a graph's dentistry shares a linear relation with the max degrees of vertices when deleted.

10.4.1 Density vs. Terminal Clique Size, Colors Needed, and Max Degree when Deleted

While the terminal clique size serves as the lower bound for colors needed, max degree when deleted + 1 serves as the upper bound for colors needed.

Appendix A: Data Structures

A.1 Vector

```
struct Vector {
 int *arr, cap, length;
 Vector() { length = 0; cap = 10; arr = new int[cap]; }
 Vector(int sz) { length = 0; cap = sz + 2; arr = new int[cap]; }
 Vector(const Vector& src) { copy(src); }
 Vector& operator = (const Vector& rhs) { copy(rhs); return *this; }
 void copy(const Vector& src) {
   if (this == &src) return;
   clear();
   cap = src.cap;
   length = src.length;
   arr = new int[cap];
   for (int i = 0; i < length; ++i) arr[i] = src.arr[i];
 }
 void clear() {
   delete [] arr;
   length = 0;
   cap = 10;
 }
 ~Vector() { clear(); }
 int& operator [] (int i) { return arr[i]; }
 void pushBack(int n) {
   if (cap <= length + 1) {
     cap *= 2;
     int *newArr = new int[cap];
     for (int i = 0; i < length; ++i) newArr[i] = arr[i];
     delete [] arr;
     arr = newArr;
   }
   arr[length++] = n;
 }
};
```

A.2 List

```
struct List {
 struct Node {
   int data;
   Node *prev, *next;
   Node(int d) { data = d; prev = next = nullptr; }
 };
 Node *head, *tail;
 int length;
 List() { head = tail = nullptr; length = 0; }
 List(const List& src) { copy(src); }
 List& operator = (const List& rhs) { copy(rhs); return *this; }
 ~List() { clear(); }
 void copy(const List& src) {
   if (this == &src) return;
   clear();
   Node *curr = src.head;
   while (curr != nullptr) {
     pushBack(curr->data);
     curr = curr->next;
   }
 }
 Node *pushBack(int n) {
   Node *newNode = new Node(n);
   if (tail == nullptr) head = tail = newNode;
   else {
     tail->next = newNode;
     newNode->prev = tail;
     tail = newNode;
   }
   ++length;
   return newNode;
 }
```

```
Node *pushFront(int n) {
  Node *newNode = new Node(n);
  if (head == nullptr) head = tail = newNode;
  else {
    head->prev = newNode;
    newNode->next = head;
    head = newNode;
  }
  ++length;
  return newNode;
}
void removeByNode(Node *nodeToDel) {
  if (length == 0) return; // if length == 0, do nothing
  if (nodeToDel == head) { // delete head
    popFront();
    return;
  }
  if (nodeToDel == tail) { // delete tail
    popBack();
    return;
  }
  Node *curr = head;              // start from head
  while (curr != nullptr) {       // until end of list
    if (curr == nodeToDel) {      // if curr's adr == nodeToDel's adr
      Node *before = curr->prev; // get curr's before
      Node *after = curr->next;  // get curr's after
      before->next = after;      // link after and
      after->prev = before;      // after
      delete curr;               // delete curr
      --length;                  // update length
      break;                     // break
    }
    curr = curr->next; // move to next node
  }
}
void removeByData(int n) {
  if (length == 0) return;
  if (head->data == n) {
    popFront();
    return;
  }
  if (tail->data == n) {
    popBack();
    return;
  }
  Node *curr = head;
  while (curr != nullptr) {
    if (curr->data == n) {
      Node *before = curr->prev;
      Node *after = curr->next;
      before->next = after;
      after->prev = before;
      delete curr;
      --length;
      break;
    }
    curr = curr->next;
  }
}
int front() { return head->data; }
void popFront() {
  if (length == 0) return; // if length == 0, nothing to delete
  Node *nodeToDel = head;  // get node to delete
  head = head->next;       // update head
  delete nodeToDel;        // delete node to delete
  --length;                // update length
  if (length > 0) head->prev = nullptr; // if there's length, update new head's prev
  else tail = nullptr;     // if length == 0, update tail
}
void popBack() {
  if (length == 0) return;
  Node *nodeToDel = tail;
  tail = tail->prev;
  delete nodeToDel;
```

```
  --length;
  if (length > 0) tail->next = nullptr;
  else head = nullptr;
 }
 bool containsData(int n) { // does list contain node with certain data?
  Node *curr = head;
  while (curr != nullptr) {
   if (curr->data == n) return true;
   curr = curr->next;
  }
  return false;
 }
 bool containsNode(Node *node) {
  Node *curr = head;
  while (curr != nullptr) {
   if (curr == node) return true;
   curr = curr->next;
  }
  return false;
 }
 void clear() {
  while (head != nullptr) { // start from head and keep deleting head
   Node *headToDel = head;
   head = head->next;
   delete headToDel;
  }
  length = 0;     // update length
  tail = nullptr; // update tail
 }
 int& operator [] (int idx) {
  if (length == 0) { // corner case 1: list is empty
   cout << "List is empty so returning length" << endl;
   return length;
  }
  if (idx >= length) { // corner case 2: index out of range
   cout << "Index out of range. Returning length" << endl;
   return length;
  }
  Node *curr = head;
  for (int i = 0; i <= idx; ++i) {   // traverse nodes till node at desired index is reached
   if (i == idx) return curr->data; // return that node's data
   curr = curr->next;
  }
  cout << "Returning length" << endl; // corner case 3: for whatever reason,
  return length;                      // data access via index didn't work
 }
 Vector toVec() {
  Vector vec(length);
  Node *curr = head;
  while (curr != nullptr) {
   vec.pushBack(curr->data);
   curr = curr->next;
  }
  return vec;
 }
};
```

A.3 Stack

```
struct Stack {
 List l;
 Stack() { }
 Stack(const Stack& src) { l = src.l; }
 Stack& operator = (const Stack& rhs) { l = rhs.l; return *this; }
 ~Stack() { }
 List::Node *push(int n) { return l.pushFront(n); }
 void pop() { l.popFront(); }
 int front() { return l.front(); }
 bool empty() { return l.length == 0; }
 int length() { return l.length; }
 Vector toVec() { return l.toVec(); }
};
```

A.4 Queue

```
struct Queue {
 List l;
 Queue() { }
 Queue(const Queue& src) { l = src.l; }
 Queue& operator = (const Queue& rhs) { l = rhs.l; return *this; }
 ~Queue() { }
 void push(int n) { l.pushBack(n); }
 void pop() { l.popFront(); }
 int front() { return l.front(); }
 bool empty() { return l.length == 0; }
};
```

A.5 AVL Tree

```
struct Tree { // AVL Tree for fast read & write
 struct Node {
   int data, height;
   Node *left, *right;
   Node(const int& d, Node *l = nullptr, Node *r = nullptr, int h = 0) {
     data = d; left = l; right = r; height = h;
   }
   Node(int&& d, Node *l = nullptr, Node *r = nullptr, int h = 0) {
     data = move(d); left = l; right = r; height = h;
   }
 };
 Node *root;
 Tree() { root = nullptr; }
 Tree(const Tree& t) { root = copy(t.root); }
 Tree& operator = (const Tree& rhs) {
   Tree copy = rhs;
   std::swap(*this, copy);
   return *this;
 }
 ~Tree() { clear(); }
 void insert(int& d) { if (!contains(d)) insertHelper(d, root); }
 int& retrieve(int& d) { return contains(d) ? getNode(d)->data : d; }
 void update(int& d) { getNode(d)->data = d; }
 void remove(const int& d) { if (contains(d)) removeHelper(d, root); }
 void clear() { clearHelper(root); }
 bool empty() const { return root == nullptr; }
 bool contains(const int& d) { return containsHelper(d, root); }
 int numNodes() { return empty() ? 0 : numNodesHelper(root); }
 Vector toVec() { Vector v; toVecHelper(root, v); return v; }
private:
 Node *getNode(int& d) { return getNodeHelper(d, root); } // get node by val
 void insertHelper(const int& d, Node*& t) {
   if (t == nullptr) t = new Node(d);
   else if (t->data > d) insertHelper(d, t->left);
   else if (t->data < d) insertHelper(d, t->right);
   balance(t);
 }
 void insertHelper(int&& d, Node*& t) {
   if (t == nullptr) t = new Node(move(d));
   else if (d < t->data) insertHelper(move(d), t->left);
   else if (t->data < d) insertHelper(move(d), t->right);
   balance(t);
 }
 void removeHelper(const int& d, Node*& t) {
   if (t == nullptr) return;
   if (t->data > d) removeHelper(d, t->left);
   else if (t->data < d) removeHelper(d, t->right);
   else if (t->left != nullptr && t->right != nullptr) {
     t->data = privFindMin(t->right)->data;
     removeHelper(t->data, t->right);
   }
   else {
     Node *oldNode = t;
     t = t->left != nullptr ? t->left : t->right;
     delete oldNode;
   }
   balance(t);
 }
 void balance(Node*& t) {
   if (t == nullptr) return;
   else if (height(t->left) - height(t->right) == 2) {
     if (height(t->left->left) >= height(t->left->right)) rotateWithLeftChild(t);
     else doubleWithLeftChild(t);
   }
   else if (height(t->right) - height(t->left) == 2) {
     if (height(t->right->right) >= height(t->right->left)) rotateWithRightChild(t);
     else doubleWithRightChild(t);
   }
   t->height = max(height(t->left), height(t->right)) + 1;
 }
 bool containsHelper(const int& d, Node *t) {
   if (t == nullptr) return false;
```

```
  else if (t->data > d) return containsHelper(d, t->left);
  else if (t->data < d) return containsHelper(d, t->right);
  else return true;
}
void clearHelper(Node*& t) {
  if (t != nullptr) {
    clearHelper(t->left);
    clearHelper(t->right);
    delete t;
  }
  t = nullptr;
}
int height(Node *t) const { return t == nullptr ? -1 : t->height; }
int max(int lhs, int rhs) const { return lhs > rhs ? lhs : rhs; }
void rotateWithLeftChild(Node*& k2) {
  Node *k1 = k2->left;
  k2->left = k1->right;
  k1->right = k2;
  k2->height = max(height(k2->left), height(k2->right)) + 1;  // update
  k1->height = max(height(k1->left), k2->height) + 1;         // height
  k2 = k1;                                                    // root
}
void rotateWithRightChild(Node*& k1) {
  Node *k2 = k1->right;
  k1->right = k2->left;
  k2->left = k1;
  k1->height = max(height(k1->left), height(k1->right)) + 1;  // update
  k2->height = max(height(k2->right), k1->height) + 1;        // height &
  k1 = k2;                                                    // root
}
void doubleWithLeftChild(Node*& k3) {
  rotateWithRightChild(k3->left);
  rotateWithLeftChild(k3);
}
void doubleWithRightChild(Node*& k1) {
  rotateWithLeftChild(k1->right);
  rotateWithRightChild(k1);
}
Node *privFindMin(Node *t) const { // return node w/ min data
  if (t == nullptr) return nullptr;
  else if (t->left == nullptr) return t;
  else return privFindMin(t->left);
}
Node *privFindMax(Node *t) const { // return node w/ max data
  if (t != nullptr) while (t->right != nullptr) t = t->right;
  return t;
}
Node *copy(Node *t) const {
  return t == nullptr ? nullptr : new Node(t->data, copy(t->left), copy(t->right), t->height);
}
Node *getNodeHelper(int& d, Node*& t) {
  if (t->data == d) return t;
  return t->data > d ? getNodeHelper(d, t->left) : getNodeHelper(d, t->right);
}
int numNodesHelper(Node*& t) {
  int nodeCount = 1;
  if (t->left!=nullptr) nodeCount += numNodesHelper(t->left);
  if (t->right!=nullptr) nodeCount += numNodesHelper(t->right);
  return nodeCount;
}
void toVecHelper(Node *t, Vector& v) {
  if (t == nullptr) return;
  toVecHelper(t->left, v);
  v.pushBack(t->data);
  toVecHelper(t->right, v);
}
};
```

A.6 Set

```
struct Set {
 Tree t;
 Set() { }
 Set(Vector& v) { for (int i = 0; i < v.length; ++i) t.insert(v[i]); }
 Set(const Set& src) { copy(src); }
 Set& operator = (const Set& rhs) { copy(rhs); return *this; }
 ~Set() { clear(); }
 void copy(const Set& src) { if (this == &src) return; t = src.t; }
 void add(int n) { t.insert(n); }
 void remove(int n) { t.remove(n); }
 void clear() { t.clear(); }
 bool empty() { return t.empty(); }
 bool contains(int n) { return t.contains(n); }
 int length() { return t.numNodes(); }
 Vector toVec() { return t.toVec(); }
};
```

A.7 Adjacency List

```
struct AdjList {
 int length;
 List *arr; // len = # of vertices; vertices = 0 ~ (# of vertices - 1)
 AdjList(int len) { length = len; arr = new List[length]; }
 AdjList(const AdjList& src) { copy(src); }
 AdjList& operator = (const AdjList& src) { copy(src); return *this; }
 ~AdjList() { clear(); }
 void copy(const AdjList& src) {
   if (this == &src) return;
   clear();
   length = src.length;
   arr = new List[length];
   for (int i = 0; i < length; ++i) arr[i] = src.arr[i];
 }
 void clear() { delete [] arr; length = 0; }
 void add(int u, int v) { if (!arr[u].containsData(v)) arr[u].pushBack(v); }
 void remove(int u, int v) { if (arr[u].containsData(v)) arr[u].removeByData(v); }
 int getNumEl() { int tot = 0; for (int i = 0; i < length; ++i) tot += arr[i].length; return tot; }
};
```

A.8 Graph

```
struct Graph {
 AdjList *adj;
 Graph(int len) { adj = new AdjList(len); }
 Graph(const Graph& src) { copy(src); }
 Graph& operator = (const Graph& rhs) { copy(rhs); return *this; }
 ~Graph() { delete adj; }
 void copy(const Graph& src) { if (this != &src) *adj = *src.adj; }
 void addEdge(int u, int v) { adj->add(u, v); adj->add(v, u); }
 void removeEdge(int u, int v) { adj->remove(u, v); adj->remove(v, u); }
 int getNumVertices() { return adj->length; }
 int getNumEdges() { return adj->getNumEl() / 2; }
};
```

Appendix B: Vertex Distributions

```
#include <random>

enum DistributionType { Uniform, Skewed, Normal };

Vector initVertices(int distributionType, int numVertices) { // init vertices based on distribution type
 switch (distributionType) {
   case Uniform: return uniform(numVertices);
   case Skewed: return linear(numVertices);
   default: return normal(numVertices);
 }
}
```

B.1 Uniform

```
Vector uniform(int numVertices) {
 Vector vec(numVertices);
 for (int i = 0; i < numVertices; ++i) vec.pushBack(i);
 return vec;
}
```

B.2 Skewed

```
Vector linear(int numVertices) { // skewed
 int numRepeat = numVertices;
 int vertex = 0;
 Vector vec(getMaxNumEdges(numVertices)); // init vector w/ sum as cap
 for (int i = 0; i < numVertices; ++i) { // # of times to repeat
   for (int j = 0; j < numRepeat; ++j) vec.pushBack(vertex); // # of times to add each vertex
   ++vertex;
   --numRepeat;
 }
 return vec;
}
```

B.3 Normal

```
Vector normal(int numVertices) {
 double mean = numVertices / 2;            // prep normal random generator
 double stdDev = numVertices / 6;          // 99.7% of data is captured in 3 std dev
 default_random_engine generator(time(0)); // We initialize a generator
 normal_distribution<double> distribution(mean, stdDev); // And a distribution

 int numRepeat = getMaxNumEdges(numVertices); // arithmetic to sufficiently populate
 Vector vec(numRepeat);
 for (int i = 0; i < numVertices; ++i) vec.pushBack(i);
 for (int i = 0; i < numRepeat; ++i) {
   double n;
   do {
     n = distribution(generator);
   } while (n < 0 or n >= numVertices); // n can't be negative or >= numVertices
   vec.pushBack((int) n);
 }
 return vec;
}

int getMaxNumEdges(int numVertices) {
  return numVertices * (numVertices - 1) / 2;
}
```

Appendix C: Graphs

```
enum GraphType { Complete, Cycle, Random };

Graph initGraph(int graphType, Vector& vertices, int numEdges = -1) {
 if (graphType == Random && numEdges == -1) {
   cout << "For random graph, need to supply number of edges" << endl;
   exit(0);
 }
 switch (graphType) {
   case Complete: return complete(vertices);
   case Cycle: return cycle(vertices);
   default: return random(vertices, numEdges);
 }
}
```

C.1 Cycle

```
Graph cycle(Vector& vertices) {
 Set set(vertices);
 Vector uniqueVertices = set.toVec();
 int len = uniqueVertices.length;
 Graph g(len);
 for (int i = 0; i < len; ++i) g.addEdge(uniqueVertices[i], uniqueVertices[(i + 1) % len]);
 return g;
}
```

C.2 Complete

```
Graph complete(Vector& vertices) {
 Set set(vertices);
 Vector uniqueVertices = set.toVec();
 int len = uniqueVertices.length;
 Graph g(len);
 for (int i = 0; i < len; ++i) {
   for (int j = 0; j < len; ++j) {
     if (i != j) g.addEdge(uniqueVertices[i], uniqueVertices[j]);
   }
 }
 return g;
}
```

C.3 Random

```
Graph random(Vector& vertices, int numEdges) {
 Graph g = cycle(vertices);
 while(g.getNumEdges() < numEdges) {
   int u, v;
   do {
     u = vertices[randIndex(vertices.length - 1)];
     v = vertices[randIndex(vertices.length - 1)];
   } while (u == v);
   g.addEdge(u, v);
 }
 return g;
}

int randIndex(int maxIdx, int minIdx = 0) { // random index 0 ~ maxIdx; uniform random distr
 random_device rand_dev;
 mt19937 generator(rand_dev());
 uniform_int_distribution<int> distribution(minIdx, maxIdx);
 return distribution(generator);
}
```

Appendix D: Ordering

```
// Smallest Last, Smallest Original Degree Last, Random,
// Incremental, Breadth First Search, Depth First Search
enum OrderType { SL, SODL, RAND, INC, BFS, DFS };

Stack order(int orderType, int numVertices, VertexEntry *vertexTracker, List *degreeTracker, int&
terminalCliqueSize, int& maxDegreeWhenDeleted) {
 switch (orderType) {
   case SL  : return smallestLast(numVertices, vertexTracker, degreeTracker, terminalCliqueSize,
maxDegreeWhenDeleted);
   case SODL: return smallestOriginalDegreeLast(numVertices, vertexTracker, degreeTracker);
   case RAND: return randomVertexOrdering(numVertices);
   case INC : return incremental(numVertices);
   case BFS : return bfs(numVertices, vertexTracker);
   default  : return dfs(numVertices, vertexTracker);
 }
}

struct VertexEntry {
 List vertices;
 int degree = 0;
 int color = -1;
 bool deleted = false;
 List::Node *nodeInDegreeTracker = nullptr;
};
```

D.1 Smallest-Last

```
Stack smallestLast(int numVertices, VertexEntry *vertexTracker, List *degreeTracker, int&
terminalCliqueSize, int& maxDegreeWhenDeleted) {
 Stack orderedVertices;
 int idx = 0;
 while (orderedVertices.length() < numVertices) {
   List::Node *currNode = degreeTracker[idx].head; // init curr node
   if (currNode == nullptr) { ++idx; continue; }   // if curr node is a nullptr, move onto next index
   int currVertex = currNode->data;                // else init curr vertex

   // update terminal clique size
   if ((idx + 1) == (numVertices - orderedVertices.length()) && terminalCliqueSize == 0)
     terminalCliqueSize = idx + 1;

   orderedVertices.push(currVertex);          // update order
   degreeTracker[idx].popFront();             // update degree tracker
   vertexTracker[currVertex].deleted = true; // update vertex tracker
   if (vertexTracker[currVertex].degree > maxDegreeWhenDeleted)
     maxDegreeWhenDeleted = vertexTracker[currVertex].degree; // update max degree when deleted

   // update adjacent vertices in degree tracker
   List::Node *adjNode = vertexTracker[currVertex].vertices.head;
   while (adjNode != nullptr) {                // for every node adjacent to current vertex
     int adjVertex = adjNode->data;            // get adjacent node's vertex
     if (!vertexTracker[adjVertex].deleted) { // if adjacent vertex has not been deleted
       int currDegree = vertexTracker[adjVertex].degree; // get current degree
       degreeTracker[currDegree].removeByNode(vertexTracker[adjVertex].nodeInDegreeTracker); // remove
old node
       vertexTracker[adjVertex].nodeInDegreeTracker = degreeTracker[currDegree - 1].pushFront(adjVertex);
// update degree tracker w/ vertex at new location and update its degree tracker
       --vertexTracker[adjVertex].degree; // update degree in vertex tracker
     }
     adjNode = adjNode->next; // mode to next adj node
   }
   if (idx > 0) --idx; // update index
 }
 return orderedVertices;
}
```

D.2 Smallest Original Degree Last

```
Stack smallestOriginalDegreeLast(int numVertices, VertexEntry *vertexTracker, List *degreeTracker) {
 Stack orderedVertices;
 int idx = 0;
 List::Node *currNode = degreeTracker[0].head;
 while (orderedVertices.length() < numVertices) {
   if (currNode == nullptr) { currNode = degreeTracker[++idx].head; continue; } // update curr node &
index
   int currVertex = currNode->data;
   orderedVertices.push(currVertex);          // update ordered vertices
   vertexTracker[currVertex].deleted = true; // update vertex tracker
   currNode = currNode->next;                 // move to next node
 }
 return orderedVertices;
}
```

D.3 Random

```
Stack randomVertexOrdering(int numVertices) {
 int *shuffledVertices = new int[numVertices]; // init unshuffled vertices
 for (int i = 0; i < numVertices; ++i) shuffledVertices[i] = i;
 shuffle(shuffledVertices, numVertices);
 Stack orderedVertices; // push shuffled vertices to stack
 for (int i = 0; i < numVertices; ++i) orderedVertices.push(shuffledVertices[i]);
 delete [] shuffledVertices;
 return orderedVertices;
}

void swap (int *a, int *b) {
 int temp = *a;
 *a = *b;
 *b = temp;
}

// Fisher-Yates Shuffling
void shuffle(int *arr, int n) { for (int i = n - 1; i > 0; --i) swap(&arr[i], &arr[randIndex(i)]); }
```

D.4 Incremental

```
Stack incremental(int numVertices) {
 Stack orderedVertices;
 for (int i = numVertices; i > 0; --i) orderedVertices.push(i - 1);
 return orderedVertices;
}
```

D.5 Breadth First Search

```
Stack bfs(int numVertices, VertexEntry *vertexTracker) {
 Stack unvisited, orderedVertices;          // init unvisited & ordered vertices
 for (int i = 0; i < numVertices; ++i) vertexTracker[i].nodeInDegreeTracker = unvisited.push(i);

 int currVertex = unvisited.front();        // init current vertex
 orderedVertices.push(currVertex);          // update color order
 vertexTracker[currVertex].deleted = true; // update vertices tracker
 unvisited.pop();                           // update unvisited

 Queue queue;
 while (orderedVertices.length() < numVertices) {

   // iterate through current vertex's adjacent nodes and add unvisted nodes
   List::Node *adjNode = vertexTracker[currVertex].vertices.head; // init adjacent node
   while (adjNode != nullptr) {                  // while there are adjacent nodes
     int adjVertex = adjNode->data;              // get adjacent node's vertex
     if (!vertexTracker[adjVertex].deleted) {    // if vertex is not deleted
       vertexTracker[adjVertex].deleted = true; // update vertex tracker
       unvisited.l.removeByNode(vertexTracker[adjVertex].nodeInDegreeTracker);
       queue.push(adjVertex);                    // update queue
       orderedVertices.push(adjVertex);          // update color order
     }
```

```
    adjNode = adjNode->next; // move to next adj node
  }

  // if queue is not empty, update curr data & queue
  if (!queue.empty()) { currVertex = queue.front(); queue.pop(); }

  // if queue is empty but there are still unvisited nodes
  else if (!unvisited.empty()) {
    currVertex = unvisited.front();           // update current vertex
    vertexTracker[currVertex].deleted = true; // update vertices tracker
    orderedVertices.push(currVertex);         // update color order
    unvisited.pop();                          // update unvisited
  }
 }
 return orderedVertices;
}
```

D.6 Depth First Search

```
Stack dfs(int numVertices, VertexEntry *vertexTracker) {
 Stack unvisited, orderedVertices, stack; // init unvisited & ordered vertices, stack
 for (int i = 0; i < numVertices; ++i) vertexTracker[i].nodeInDegreeTracker = unvisited.push(i);

 int currVertex = unvisited.front();       // init current vertex
 orderedVertices.push(currVertex);         // update color order
 vertexTracker[currVertex].deleted = true; // update vertices tracker
 unvisited.pop();                          // update unvisited

 while (orderedVertices.length() < numVertices) {
  // iterate through current vertex's adjacent nodes and add unvisted nodes
  List::Node *adjNode = vertexTracker[currVertex].vertices.head; // init adjacent node
  while (adjNode != nullptr) {                  // while there are adjacent nodes
    int adjVertex = adjNode->data;              // get adjacent node's vertex
    if (!vertexTracker[adjVertex].deleted) {    // if vertex is not deleted
      vertexTracker[adjVertex].deleted = true; // update vertex tracker
      unvisited.l.removeByNode(vertexTracker[adjVertex].nodeInDegreeTracker);
      stack.push(adjVertex);                    // update stack
      orderedVertices.push(adjVertex);          // update color order
    }
    adjNode = adjNode->next; // move to next adj node
  }

  // if stack is not empty, update curr data & stack
  if (!stack.empty()) { currVertex = stack.front(); stack.pop(); }

  // if stack is empty but there are still unvisited nodes
  else if (!unvisited.empty()) {
    currVertex = unvisited.front();           // update current vertex
    vertexTracker[currVertex].deleted = true; // update vertices tracker
    orderedVertices.push(currVertex);         // update color order
    unvisited.pop();                          // update unvisited
  }
 }
 return orderedVertices;
}
```

Appendix E. Coloring

E.1 Greedy Coloring

```
// get # of colors used according to particular ordering
int color(int numVertices, Stack orderedVertices, VertexEntry *vertexTracker, Vector& degreesWhenDeleted)
{
 int maxColor = 0; // track highest color used

 while (!orderedVertices.empty()) {              // while there are vertices left on the stack
   int currVertex = orderedVertices.front();     // get vertex
   orderedVertices.pop();                        // update ordered vertices
   degreesWhenDeleted.pushBack(vertexTracker[currVertex].degree); // update degrees when colored
   int minColor = getMinColor(numVertices, currVertex, vertexTracker); // get min color
   if (minColor > maxColor) maxColor = minColor; // update max color
   vertexTracker[currVertex].color = minColor;   // update vertex tracker
 }
 return maxColor; // highest color used = # of colors used
}

int getMinColor(int numVertices, int currVertex, VertexEntry *vertexTracker) {
 bool *availableColors = new bool[numVertices + 1]; // init available colors
 for (int i = 1; i < numVertices + 1; ++i) availableColors[i] = true;

 // update available colors w/ colors taken by current vertex's adjacent nodes
 List::Node *adjNode = vertexTracker[currVertex].vertices.head;
 while (adjNode != nullptr) {
   int adjVertex = adjNode->data;
   int adjColor = vertexTracker[adjVertex].color;
   availableColors[adjColor] = false;
   adjNode = adjNode->next;
 }

 int minColor; // get min color from available colors
 for (int i = 1; i < numVertices + 1; ++i) if (availableColors[i]) { minColor = i; break; }
 delete [] availableColors;
 return minColor;
}
```

Appendix F: Analysis Functions

F.1 Write to File

```
#include <fstream>

enum orderingOrColoringMode { Coloring, Ordering };
enum edgesOrVerticesMode { Edges, Vertices };

const int NUM_REPEAT = 5;
const int NUM_VERTICES = 50;
const int EDGE_INCREMENT = 50;

void writeToFileHistogram(Vector& vertices, string fileName) {
 ofstream out(fileName + ".csv"); // create file w/ specified name
 out << "Vertex" << endl;
 for (int i = 0; i < vertices.length; ++i) out << vertices[i] << endl; // write vertices to file
 out.close();
}

void writeToFileRuntime(int **aoa, int rows, int cols, int distributionType, string name) {
 ofstream out(name + " (" + distributionToStr(distributionType) + ").csv");
 out << "Edges,SL,SODL,RAND,INC,BFS,DFS" << endl;
 for (int orderType = 0; orderType < rows; ++orderType) {
   string row = to_string(aoa[orderType][0]);
   for (int density = 1; density < cols; ++density) {
     row += ',' + to_string(aoa[orderType][density]);
   }
   out << row << endl;
 }
 out.close();
}

void writeToFileOrderVsDegree(int density, Vector& degreesWhenDeleted) {
 string fileName = "Order vs. Degree When Colored (V = " + to_string(degreesWhenDeleted.length) + ") ("
+ densityToStr(density) + ").csv";
 ofstream out(fileName);
 out << "Order,\"Degree When Colored\"" << endl;
 for (int i = 0; i < degreesWhenDeleted.length; ++i) out << i << ',' << degreesWhenDeleted[i] << endl;
 out.close();
}
```

F.2 To String Functions

```
string distributionToStr(int distributionType) {
 switch (distributionType) {        // int distribution index to
   case Uniform: return "Uniform"; // string distribution name
   case Skewed: return "Skewed";
   default: return "Normal";
 }
}

string graphTypeToStr(int graphType) { // Cycle, Random, Complete
 switch (graphType) {
   case Cycle: return "Cycle";
   case Random: return "Random";
   case Complete: return "Complete";
   default: return "Unknown Graph Type";
 }
}

string orderTypeToStr(int orderType) { // SL, SODL, RAND, INC, BFS, DFS
 switch (orderType) {
   case SL: return "SL";
   case SODL: return "SODL";
   case RAND: return "RAND";
   case INC: return "INC";
   case BFS: return "BFS";
   case DFS: return "DFS";
   default: return "Unknown Order Type";
 }
}

string densityToStr(int density) {
 switch (density) {
   case 0: return "Cycle";
   case 10: return "Complete";
   default: return "Random " + to_string(density * 10) + "%";
 }
}

string edgesOrVerticesToStr(int edgesOrVertices) { // Edges, Vertices
 switch (edgesOrVertices) {
   case Edges: return "Edges";
   case Vertices: return "Vertices";
   default: return "Unknown Edges or Vertices Mode";
 }
}

string orderingOrColoringToStr(int orderingOrColoring) { // Ordering, Coloring
 switch (orderingOrColoring) {
   case Ordering: return "Ordering";
   case Coloring: return "Coloring";
   default: return "Unknown Ordering or Coloring Mode";
 }
}

string getFileName(int ev, int oc, int gt = -1) {
 if (gt == -1) return "Time vs. " + edgesOrVerticesToStr(ev) + " on " + orderingOrColoringToStr(oc);
 return "Time vs. " + edgesOrVerticesToStr(ev) + " on " + orderingOrColoringToStr(oc) + " (" +
graphTypeToStr(gt) + ')';
}
```

F.3 Vertex Distribution (Histogram)

```
// generate histogram based on distribution types Uniform, Skewed, Normal
void generateHistogram(int numVertices = NUM_VERTICES) { // default # of vertices = 100
 cout << "Generating data Vertex Distribution for Histogram..." << endl;
 for (int distributionType = 0; distributionType < 3; ++distributionType) { // for each distribution
index
   string fileName = distributionToStr(distributionType) + " - " + to_string(numVertices); // name file
   Vector vertices = initVertices(distributionType, numVertices); // generate vertices for each
distribution
   writeToFileHistogram(vertices, fileName); // create histogram using generated vertices
 }
}
```

F4. Runtime (Edges vs. Ordering or Coloring)

```
#include <chrono>

// measure density vs ordering / coloring runtime performance
// for all 3 distribution types Uniform, Skewed, Normal
void timeEdges(int orderingOrColoring, int numVertices = NUM_VERTICES, int numRepeat = NUM_REPEAT) {
 cout << "Generating data for Edges vs. " << orderingOrColoringToStr(orderingOrColoring) << "..." <<
endl;

 int cols = 7;
 int rows = 11;

 int maxNumEdges = getMaxNumEdges(numVertices);

 // for each distribution
 for (int distributionType = 0; distributionType < 3; ++distributionType) { // for each distribution

   // init vertices
   Vector vertices = initVertices(distributionType, numVertices);

   // init record
   int **rec = new int*[rows];
   for (int i = 0; i < rows; ++i) rec[i] = new int[cols];
   for (int i = 0; i < 11; ++i) rec[i][0] = i == 0 ? numVertices : i * maxNumEdges / 10; // x-axis = #
of edges

   // for each order type
   for (int orderType = 0; orderType < 6; ++orderType) {

     // for each density level
     for (int density = 0; density < 11; ++density) {

       // init graph
       int graphType, numEdges;
       initGraphTypeAndNumEdges(graphType, numEdges, numVertices, density);
       Graph g = initGraph(graphType, vertices, numEdges);
       double t = 0.0;

       for (int i_rep = 0; i_rep < numRepeat; ++i_rep) {

         // init params
         VertexEntry *vertexTracker = new VertexEntry[numVertices];
         List *degreeTracker = new List[numVertices];
         for (int i = 0; i < numVertices; ++i) {
           vertexTracker[i].vertices = g.adj->arr[i];
           vertexTracker[i].degree = vertexTracker[i].vertices.length;
           vertexTracker[i].nodeInDegreeTracker = degreeTracker[vertexTracker[i].degree].pushFront(i);
         }
         int terminalCliqueSize = 0;
         int maxDegreeWhenDeleted = 0;
         Vector degreesWhenDeleted(numVertices);

         // order and color
         auto start = chrono::high_resolution_clock::now();
         auto end = start;
         if (orderingOrColoring == Ordering) {
           start = chrono::high_resolution_clock::now();
           order(orderType, numVertices, vertexTracker, degreeTracker, terminalCliqueSize,
maxDegreeWhenDeleted);
           end = chrono::high_resolution_clock::now();
         }
         else if (orderingOrColoring == Coloring) {
           Stack orderedVertices = order(orderType, numVertices, vertexTracker, degreeTracker,
terminalCliqueSize, maxDegreeWhenDeleted);
           start = chrono::high_resolution_clock::now();
           color(numVertices, orderedVertices, vertexTracker, degreesWhenDeleted);
           end = chrono::high_resolution_clock::now();
         }

         // get time
         t += chrono::duration_cast<chrono::microseconds>(end - start).count(); // update time

         delete [] vertexTracker;
```

```
        delete [] degreeTracker;
      }

      // record time
      rec[density][orderType + 1] = t / numRepeat;
    }
  }

  // write
  string fileName = getFileName(Edges, orderingOrColoring);
  writeToFileRuntime(rec, rows, cols, distributionType, fileName);

  for (int i = 0; i < rows; ++i) if (rec[i] != nullptr) delete [] rec[i];
  delete [] rec;
 }
}

// init parameters graphType and numEdges needed for graph initialization based on numVertices and 11 (0
~ 10) density levels
void initGraphTypeAndNumEdges(int& graphType, int& numEdges, int numVertices, int density) {
 /*
   0  Cycle
   1  Random  10%
   2  Random  20%
   3  Random  30%
   4  Random  40%
   5  Random  50%
   6  Random  60%
   7  Random  70%
   8  Random  80%
   9  Random  90%
   10 Complete
 */
 switch (density) {
   case 0:
     graphType = Cycle;
     numEdges = numVertices;
     break;
   case 10:
     graphType = Complete;
     numEdges = getMaxNumEdges(numVertices);
     break;
   default:
     graphType = Random;
     numEdges = density * getMaxNumEdges(numVertices) / 10;
 }
}
```

F5. Runtime (Vertices vs. Ordering or Coloring)

```
// measure vertices vs ordering / coloring runtime performance
// for all 3 graph types Complete, Cycle, Random
// for all 3 distribution types Uniform, Skewed, Normal
void timeVertices(int orderingOrColoring, int numRepeat = NUM_REPEAT) {
 cout << "Generating data for Vertices vs. " << orderingOrColoringToStr(orderingOrColoring) << "..." <<
endl;
  // set limit for max # of vertices
 int maxNumVertices = 256;
  // init rows
 int rows = 0;
 int temp = maxNumVertices;
 while (temp > 4) {
   ++rows;
   temp /= 2;
 }

 // for each graph type
 for (int graphType = 0; graphType < 3; ++graphType) {

   // for each distribution type
   for (int distributionType = 0; distributionType < 3; ++distributionType) {
     // init record
     int cols = 7;
     int **rec = new int*[rows];
     for (int i = 0; i < rows; ++i) rec[i] = new int[cols];

     int i_row = 0;
     for (int numVertices = 8; numVertices <= maxNumVertices; numVertices *= 2) { // num vertices grows
by factor of 2
       rec[i_row][0] = numVertices;
       Vector vertices = initVertices(distributionType, numVertices);
       for (int orderType = 0; orderType < 6; ++orderType) { // for each order type]
         int numEdges = getMaxNumEdges(numVertices) / 2;
         Graph g = initGraph(graphType, vertices, numEdges); // Complete, Cycle, Random (for random
graph, supply # of edges)
         double t = 0.0;
         for (int i_rep = 0; i_rep < numRepeat; ++i_rep) {

           // init params
           VertexEntry *vertexTracker = new VertexEntry[numVertices];
           List *degreeTracker = new List[numVertices];
           for (int i = 0; i < numVertices; ++i) {
             vertexTracker[i].vertices = g.adj->arr[i];
             vertexTracker[i].degree = vertexTracker[i].vertices.length;
             vertexTracker[i].nodeInDegreeTracker = degreeTracker[vertexTracker[i].degree].pushFront(i);
           }
           int terminalCliqueSize = 0;
           int maxDegreeWhenDeleted = 0;
           Vector degreesWhenDeleted(numVertices);
           auto start = chrono::high_resolution_clock::now();
           auto end = start;
           if (orderingOrColoring == Ordering) {
             start = chrono::high_resolution_clock::now();
             order(orderType, numVertices, vertexTracker, degreeTracker, terminalCliqueSize,
maxDegreeWhenDeleted);
             end = chrono::high_resolution_clock::now();
           }
           else if (orderingOrColoring == Coloring) {
             Stack orderedVertices = order(orderType, numVertices, vertexTracker, degreeTracker,
terminalCliqueSize, maxDegreeWhenDeleted);
             start = chrono::high_resolution_clock::now();
             color(numVertices, orderedVertices, vertexTracker, degreesWhenDeleted);
             end = chrono::high_resolution_clock::now();
           }
           t += chrono::duration_cast<chrono::microseconds>(end - start).count();

           delete [] vertexTracker;
           delete [] degreeTracker;
         }
         rec[i_row][orderType + 1] = t / numRepeat;
       }
       ++i_row;
```

```
      }
      string fileName = getFileName(Vertices, orderingOrColoring, graphType);
      writeToFileRuntime(rec, rows, cols, distributionType, fileName);
      for (int i = 0; i < rows; ++i) if (rec[i] != nullptr) delete [] rec[i];
      delete [] rec;
    }
  }
}
```

F6. Smallest-Last In-Depth (Order Colored vs. Degree when Deleted)

```
// order colored vs degree when deleted
void orderVsDegree(int orderType = SL, int numVertices = NUM_VERTICES) {
 cout << "Generating data for Order Colored vs. Degree when Deleted..." << endl;

 // generate vertices
 int distributionType = Uniform;
 Vector vertices = initVertices(distributionType, numVertices);

 for (int density = 0; density < 11; ++density) {

   // generate graph
   int graphType, numEdges;
   initGraphTypeAndNumEdges(graphType, numEdges, numVertices, density);
   Graph g = initGraph(graphType, vertices, numEdges);

   // init params
   VertexEntry *vertexTracker = new VertexEntry[numVertices];
   List *degreeTracker = new List[numVertices];
   for (int i = 0; i < numVertices; ++i) {
     vertexTracker[i].vertices = g.adj->arr[i];
     vertexTracker[i].degree = vertexTracker[i].vertices.length;
     vertexTracker[i].nodeInDegreeTracker = degreeTracker[vertexTracker[i].degree].pushFront(i);
   }
   int terminalCliqueSize = 0;
   int maxDegreeWhenDeleted = 0;
   Vector degreesWhenDeleted(numVertices);

   // order & color
   Stack orderedVertices = order(orderType, numVertices, vertexTracker, degreeTracker,
terminalCliqueSize, maxDegreeWhenDeleted);
   int colorsNeeded = color(numVertices, orderedVertices, vertexTracker, degreesWhenDeleted);

   // write
   writeToFileOrderVsDegree(density, degreesWhenDeleted);

   delete [] vertexTracker;
   delete [] degreeTracker;
 }
}
```

F7. Smallest-Last In-Depth (Density vs. Degree when Deleted)

```
// density vs max degree when deleted
void densityVsDegree(int orderType = SL, int numVertices = NUM_VERTICES) {
 cout << "Generating data for Density vs. Max Degree when Deleted & Terminal Clique Size..." << endl;

 // open file
 ofstream out("Density vs. Max Degree when Deleted & Terminal Clique Size (V = " + to_string(numVertices)
+ ").csv");
 out << "\"Density Level\",\"Graph Type\",\"Edges\",\"Max Degree when Deleted\",\"Terminal Clique Size\""
<< endl;

 // generate vertices
 int distributionType = Uniform;
 Vector vertices = initVertices(distributionType, numVertices);

 for (int density = 0; density < 11; ++density) {

   // generate graph
   int graphType, numEdges;
   initGraphTypeAndNumEdges(graphType, numEdges, numVertices, density);
   Graph g = initGraph(graphType, vertices, numEdges);

   // init params
   VertexEntry *vertexTracker = new VertexEntry[numVertices];
   List *degreeTracker = new List[numVertices];
   for (int i = 0; i < numVertices; ++i) {
     vertexTracker[i].vertices = g.adj->arr[i];
     vertexTracker[i].degree = vertexTracker[i].vertices.length;
     vertexTracker[i].nodeInDegreeTracker = degreeTracker[vertexTracker[i].degree].pushFront(i);
   }
   int terminalCliqueSize = 0;
   int maxDegreeWhenDeleted = 0;
   Vector degreesWhenDeleted(numVertices);

   // order & color
   Stack orderedVertices = order(orderType, numVertices, vertexTracker, degreeTracker,
terminalCliqueSize, maxDegreeWhenDeleted);
   int colorsNeeded = color(numVertices, orderedVertices, vertexTracker, degreesWhenDeleted);

   // write results
   out << density << ',' << densityToStr(density) << ',' << numEdges << ',' << maxDegreeWhenDeleted <<
',' << terminalCliqueSize << endl;

   delete [] vertexTracker;
   delete [] degreeTracker;
 }

 // close file
 out.close();
}
```

F8. Smallest-Last In-Depth (Density vs. Terminal Clique, Colors Needed, Max Deg when Del)

```
// density vs terminal clique size, colors needed, max degree
void densityVsCliqueColorsNeededMaxDeg(int orderType = SL, int numVertices = NUM_VERTICES, int
edgeIncrement = EDGE_INCREMENT) {
 cout << "Generating data for Density vs. Terminal Clique Size, Colors Needed, Max Degree when
Deleted..." << endl;

 // open file
 ofstream out("Density vs. Terminal Clique Size, Colors Needed, Max Degree when Deleted (V = " +
to_string(numVertices) + ").csv");
 out << "\"Edges\",\"Terminal Clique Size\",\"Colors Needed\",\"Max Degree when Deleted\"" << endl;

 // generate vertices
 int distributionType = Uniform;
 Vector vertices = initVertices(distributionType, numVertices);

 int graphType = Random;
 int maxNumEdges = getMaxNumEdges(numVertices);

 for (int numEdges = numVertices; numEdges <= maxNumEdges; numEdges += edgeIncrement) {

   // generate graph
   Graph g = initGraph(graphType, vertices, numEdges);

   // init params
   VertexEntry *vertexTracker = new VertexEntry[numVertices];
   List *degreeTracker = new List[numVertices];
   for (int i = 0; i < numVertices; ++i) {
     vertexTracker[i].vertices = g.adj->arr[i];
     vertexTracker[i].degree = vertexTracker[i].vertices.length;
     vertexTracker[i].nodeInDegreeTracker = degreeTracker[vertexTracker[i].degree].pushFront(i);
   }
   int terminalCliqueSize = 0;
   int maxDegreeWhenDeleted = 0;
   Vector degreesWhenDeleted(numVertices);

   // order & color
   Stack orderedVertices = order(orderType, numVertices, vertexTracker, degreeTracker,
terminalCliqueSize, maxDegreeWhenDeleted);
   int colorsNeeded = color(numVertices, orderedVertices, vertexTracker, degreesWhenDeleted);

   // write results
   out << numEdges << ',' << terminalCliqueSize << ',' <<colorsNeeded << ',' << maxDegreeWhenDeleted <<
endl;

   delete [] vertexTracker;
   delete [] degreeTracker;
 }

 // close file
 out.close();
}
```

F9. Compare Colors Needed Performance (Density vs Colors Needed)

```
// density vs colors needed for all 6 vertex ordering algorithms
void compareEdgesVsColorsNeeded(int numVertices = NUM_VERTICES, int edgeIncrement = EDGE_INCREMENT) {
 for (int orderType = 0; orderType < 6; ++orderType) edgesVsColorsNeeded(orderType, numVertices,
edgeIncrement);
}

// density vs colors needed for specific vertex ordering algorithm
void edgesVsColorsNeeded(int orderType, int numVertices = NUM_VERTICES, int edgeIncrement =
EDGE_INCREMENT) {
 string orderTypeStr = orderTypeToStr(orderType);
 cout << "Generating data for Edges vs. Colors Needed for " << orderTypeStr << "..." << endl;

 // open file
 ofstream out("Edges vs. Colors Needed (V = " + to_string(numVertices) + ") (" + orderTypeStr + ").csv");
 out << "\"Edges\",\"Colors Needed\"" << endl;

 // generate vertices
 int distributionType = Uniform;
 Vector vertices = initVertices(distributionType, numVertices);

 int graphType = Random;
 int maxNumEdges = getMaxNumEdges(numVertices);

 for (int numEdges = numVertices; numEdges <= maxNumEdges; numEdges += edgeIncrement) {

   // generate graph
   Graph g = initGraph(graphType, vertices, numEdges);

   // init params
   VertexEntry *vertexTracker = new VertexEntry[numVertices];
   List *degreeTracker = new List[numVertices];
   for (int i = 0; i < numVertices; ++i) {
     vertexTracker[i].vertices = g.adj->arr[i];
     vertexTracker[i].degree = vertexTracker[i].vertices.length;
     vertexTracker[i].nodeInDegreeTracker = degreeTracker[vertexTracker[i].degree].pushFront(i);
   }
   int terminalCliqueSize = 0;
   int maxDegreeWhenDeleted = 0;
   Vector degreesWhenDeleted(numVertices);

   // order & color
   Stack orderedVertices = order(orderType, numVertices, vertexTracker, degreeTracker,
terminalCliqueSize, maxDegreeWhenDeleted);
   int colorsNeeded = color(numVertices, orderedVertices, vertexTracker, degreesWhenDeleted);

   // write results
   out << numEdges << ',' << colorsNeeded << endl;

   delete [] vertexTracker;
   delete [] degreeTracker;
 }

 // close file
 out.close();
}
```

F10. Vertices, Graph, Order, Color (Vertex, Color, Original Deg, Deg When Del)

```
// distributionType Uniform, Skewed, Normal
// graphType Complete, Cycle, Random
// orderType SL, SODL, RAND, INC, BFS, DFS
// density = 1, 2, 3, 4, 5, 6, 7, 8, 9 (only needed when graphType is Random)
void graphColorOrderWrite(int distributionType, int graphType, int orderType, int density = -1, int
numVertices = NUM_VERTICES) {
 // generate vertices
 Vector vertices = initVertices(distributionType, numVertices);
  // generate graph
 int numEdges = density * getMaxNumEdges(numVertices) / 10;
 if (density == -1 && graphType == Random) {
   cout << "Need to specify density (1 ~ 10) for Random graph type." << endl;
   return;
 }
 Graph g = initGraph(graphType, vertices, numEdges); // numEdges only used for random graph

 // init params
 VertexEntry *vertexTracker = new VertexEntry[numVertices];
 List *degreeTracker = new List[numVertices];
 for (int i = 0; i < numVertices; ++i) {
   vertexTracker[i].vertices = g.adj->arr[i];
   vertexTracker[i].degree = vertexTracker[i].vertices.length;
   vertexTracker[i].nodeInDegreeTracker = degreeTracker[vertexTracker[i].degree].pushFront(i);
 }
 int terminalCliqueSize = 0;
 int maxDegreeWhenDeleted = 0;
 Vector degreesWhenDeleted(numVertices);

 // order & color
 Stack orderedVertices = order(orderType, numVertices, vertexTracker, degreeTracker, terminalCliqueSize,
maxDegreeWhenDeleted);
 int colorsNeeded = color(numVertices, orderedVertices, vertexTracker, degreesWhenDeleted);

 // init file name
 string fileName = "Vertex, Color, Original Degree, Degree When Deleted (";
 fileName += distributionToStr(distributionType) + ") (";
 fileName += graphTypeToStr(graphType) + ") (";
 if (graphType == Random) fileName += densityToStr(density) + ") (";
 fileName += "V = " + to_string(numVertices) + ") (";
 fileName += orderTypeToStr(orderType) + ").csv";

 // write
 ofstream out(fileName);
 out << "\"Vertex\",\"Color\",\"Original Degree\",\"Degree when Deleted\"" << endl;
 Vector orderToVertex = orderedVertices.toVec();
 for (int i = 0; i < numVertices; ++i) {   // order colored
   int currVertex = orderToVertex[i];
   int color = vertexTracker[currVertex].color;
   int originalDegree = g.adj->arr[currVertex].length;
   int degreeWhenDeleted = orderType == SL ? degreesWhenDeleted[i] : vertexTracker[currVertex].degree;
   out << currVertex << ',' << color << ',' << originalDegree << ',' << degreeWhenDeleted << endl;
 }
 out.close();
 delete [] vertexTracker;
 delete [] degreeTracker;
}
```

Appendix G. Driver Functions

```
// generates 46 csv data files used for report's analysis
void generateData() {
 generateHistogram();
 timeEdges(Ordering);
 timeEdges(Coloring);
 timeVertices(Ordering);
 timeVertices(Coloring);
 orderVsDegree();
 densityVsDegree();
 densityVsCliqueColorsNeededMaxDeg();
 compareEdgesVsColorsNeeded();
}

// generates 198 csv files of possible graphs and their coloring outcomes
// Vertex, Color, Original Degree, Degree when Deleted in colored order
void generateAllPossibleGraphs(int numVertices = NUM_VERTICES) {
 cout << "Generating all possible graphs..." << endl;
 for (int distributionType = 0; distributionType < 3; ++distributionType) {
   for (int density = 0; density < 11; ++density) {
     int graphType, numEdges;
     initGraphTypeAndNumEdges(graphType, numEdges, numVertices, density);
     for (int orderType = 0; orderType < 6; ++orderType) {
       graphColorOrderWrite(distributionType, graphType, orderType, density, numVertices);
     }
   }
 }
}

int main() {
 generateData();               // generates 46 csv files of data used for report's analysis
 generateAllPossibleGraphs(); // generate 198 csv files of all possible graphs & their coloring outcomes
 cout << "DONE" << endl;
 return 0;
}
```

Works Cited

1. Matula, David W., and Leland L. Beck. “Smallest-Last Ordering and Clustering and Graph Coloring Algorithms.” *Journal of the ACM* 30.3 (1983): 417-427.
2. Juniawan, Fransiskus. “Performance Comparison of Linear Congruent Method and Fisher-Yates Shuffle for Data Randomization.” *Journal of Physics* 1196 (2019).