Final Project Report: Graph Coloring Analysis

Anthony Wang

47733248

CS 7350

All source code is available at:

https://github.com/Djaenk/Classroom-Projects/tree/master/CS%207350%20Algorithm%20Engineering

Computing Environment

The program used for analysis is written in C++ and compiled using g++ 8.3.0 using compiler defaults. All compilation and execution was performed on a Windows 10 computer using the Windows Subsystem for Linux running Debian GNU/Linux 10 (buster). The relevant hardware specifications of the computer include an AMD Ryzen 5 1600 processor and 16 GB of RAM. In the interest of saving space on legends and axis titles, all runtime analyses, tables, and charts in this report will show time in microseconds as a function of graph vertex count.

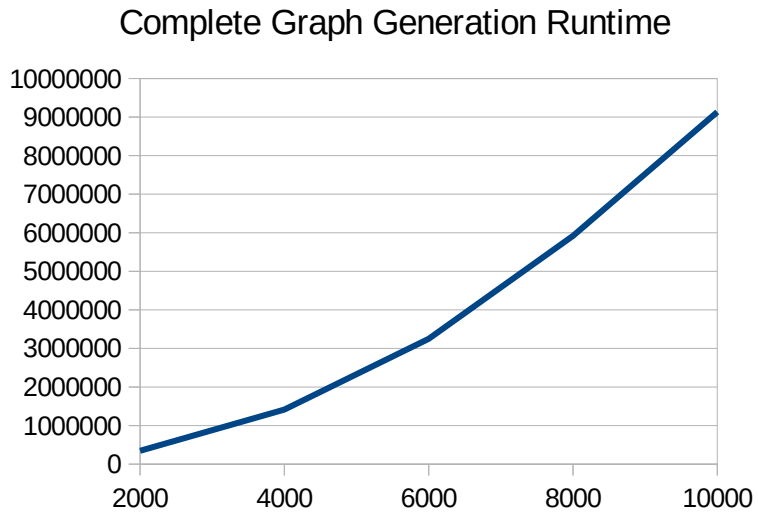
Graph Generation

The program stores graphs in memory as adjacency lists and represents each vertex with an integer. The data structure itself is a standard template library vector indexed directly by a vertex. Each entry within the vector is a standard template library list of integers which represent neighboring vertices.

Generation of a complete graph requires only one parameter: the number of vertices in the graph. For each vertex, the algorithm must create an edge to every other vertex by placing them into the former vertex's list. The nested loop structure of the implementation gives it a time complexity of $O(V^2)$, as supported by the quadratic curve in the runtime graph below.

Complete Graph

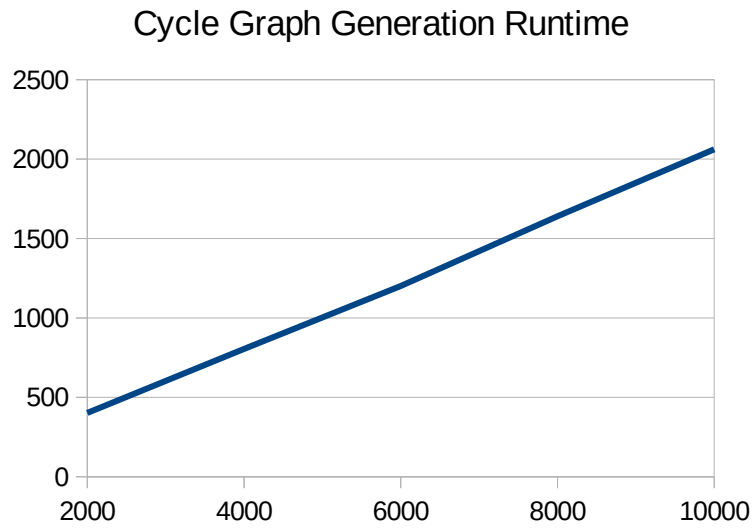
V	t
2000	343436
4000	1414489
6000	3249285
8000	5920880
10000	9130536



Generating a cycle graph also only requires the number of vertices as a parameter. The implementation loops over each vertex once and assigns the vertices immediately preceding and proceeding it as neighbors. Before completing, constant time operations are performed to set the first and last vertices as neighbors to complete a cycle. Because each vertex is iterated over once, the time complexity is O(V).

Cycle Graph

V	t
2000	402
4000	805
6000	1201
8000	1640
10000	2061

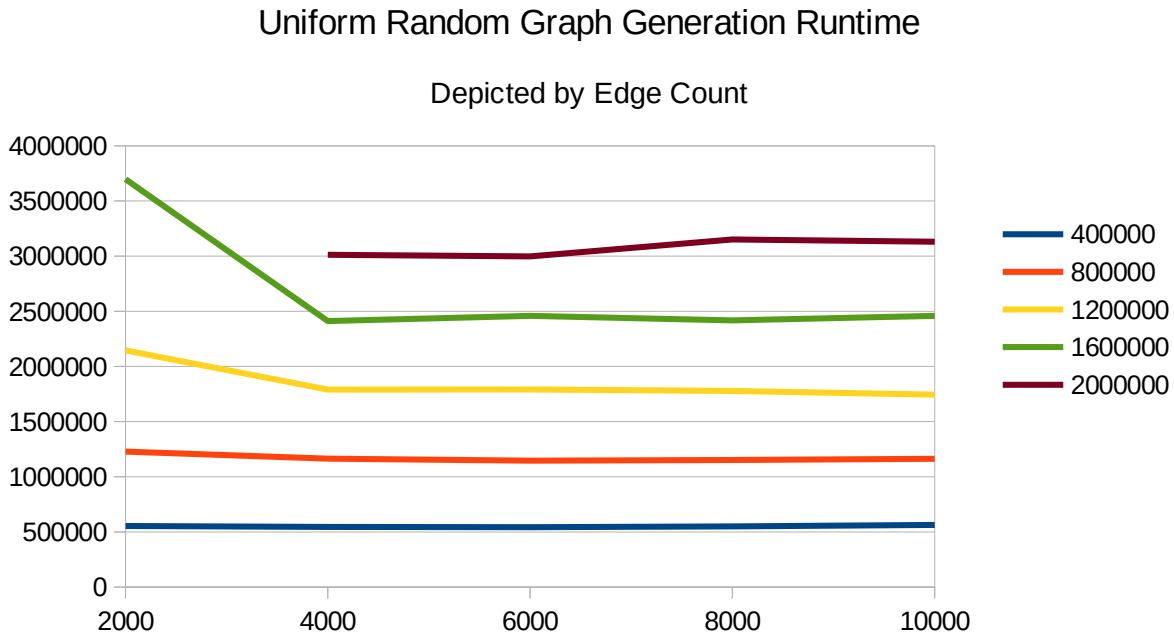


Generation of a random graph leverages the standard random library. The uniform, skewed, and normal variants of the generator have similar structures. Each generates a random integer edge which is converted into two vertices via Matthew Szudzik's elegant pairing function (http://szudzik.com/ElegantPairing.pdf) and placed into each other's lists to denote them as neighbors. This process is repeated for as many edges as are specified in the parameters. In the best case, no edge is ever randomly generated twice so the implementation runs in Ω(E). However, as a graph is more heavily populated by edges, the chance of randomly generating the same edges increases, and the performance of the implementation may degenerate to as bad as $O(E^2)$ while the implementation tries to select from the remaining edges. The below charts

exhibit this behavior. There is a large decrease in runtime when increasing the number of vertices from 2000 to 4000 because the number of edges to randomly select from is quadrupled. Afterwards, an increase in the number of vertices does not consistently affect the time to generate a random graph.

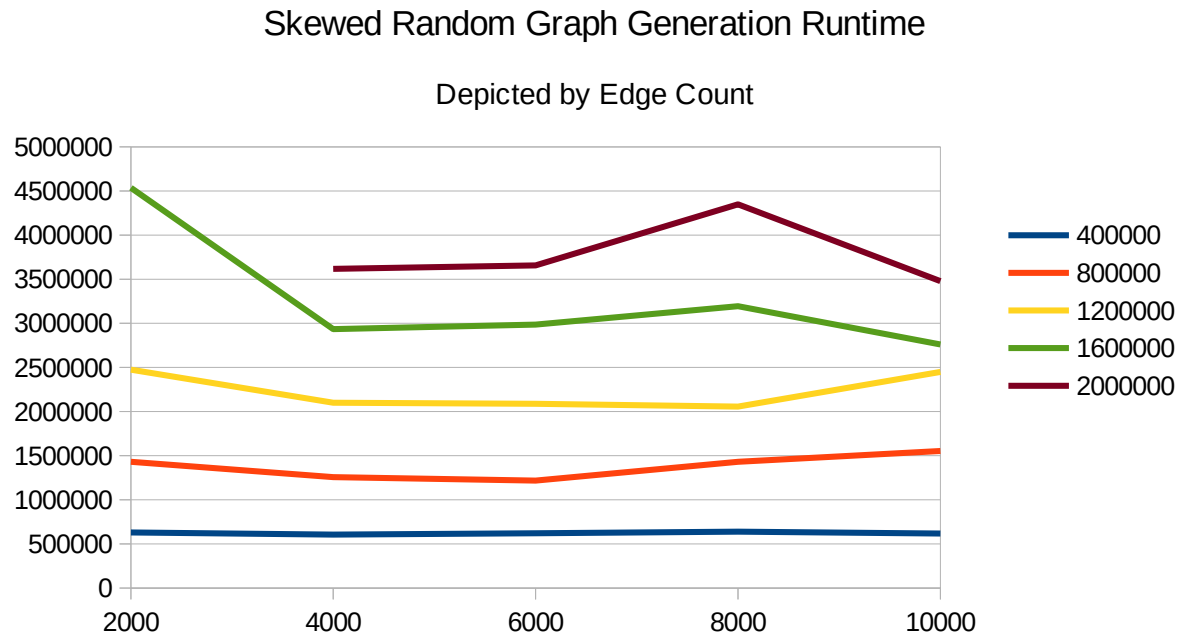
Uniform Random Graph Runtime

V\E	400000	800000	1200000	1600000	2000000
2000	553493	1228651	2146141	3697604	-
4000	545601	1163547	1789903	2411952	3011990
6000	542222	1146143	1790869	2459300	2997901
8000	549410	1152784	1778396	2417452	3151575
10000	563173	1162781	1743991	2458022	3131141



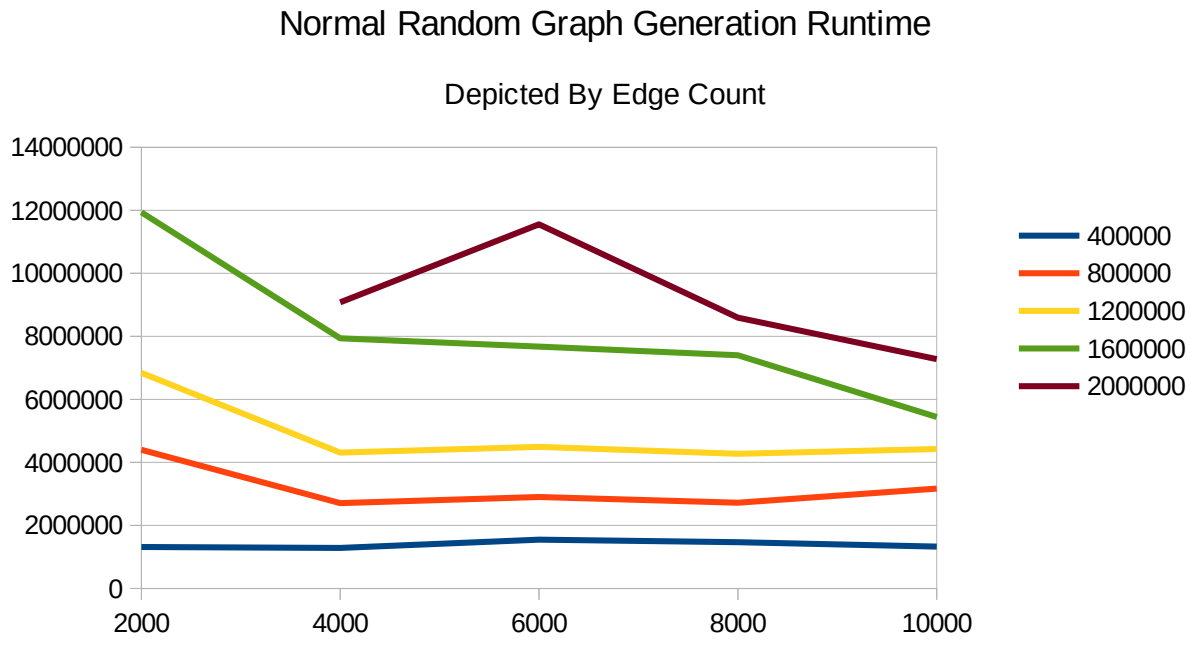
Skewed Random Graph Runtime

V\E	400000	800000	1200000	1600000	2000000
2000	630265	1430099	2474976	4535911	-
4000	606754	1256449	2100796	2935371	3616536
6000	621591	1217891	2088418	2985307	3657179
8000	639504	1430865	2056276	3194889	4349035
10000	618258	1553552	2449297	2759956	3479101

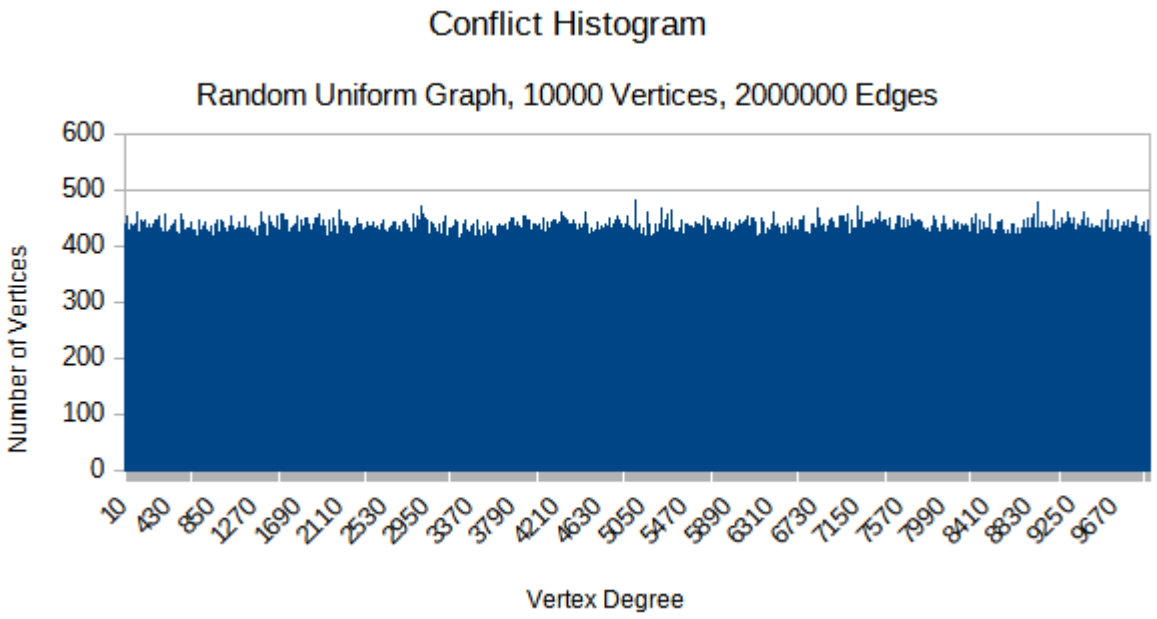


Normal Random Graph Runtime

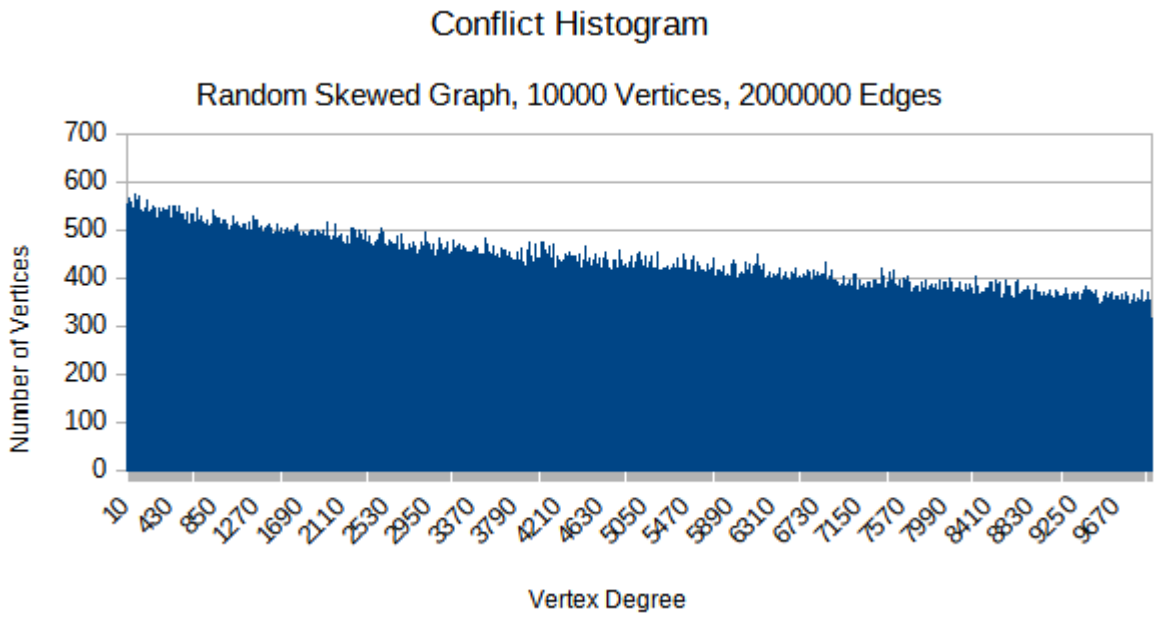
V\E	400000	800000	1200000	1600000	2000000
2000	1314677	4398411	6841422	11935139	-
4000	1286156	2705151	4310348	7936635	9079760
6000	1547009	2901705	4491400	7675941	11556130
8000	1466136	2721105	4272209	7398700	8594159



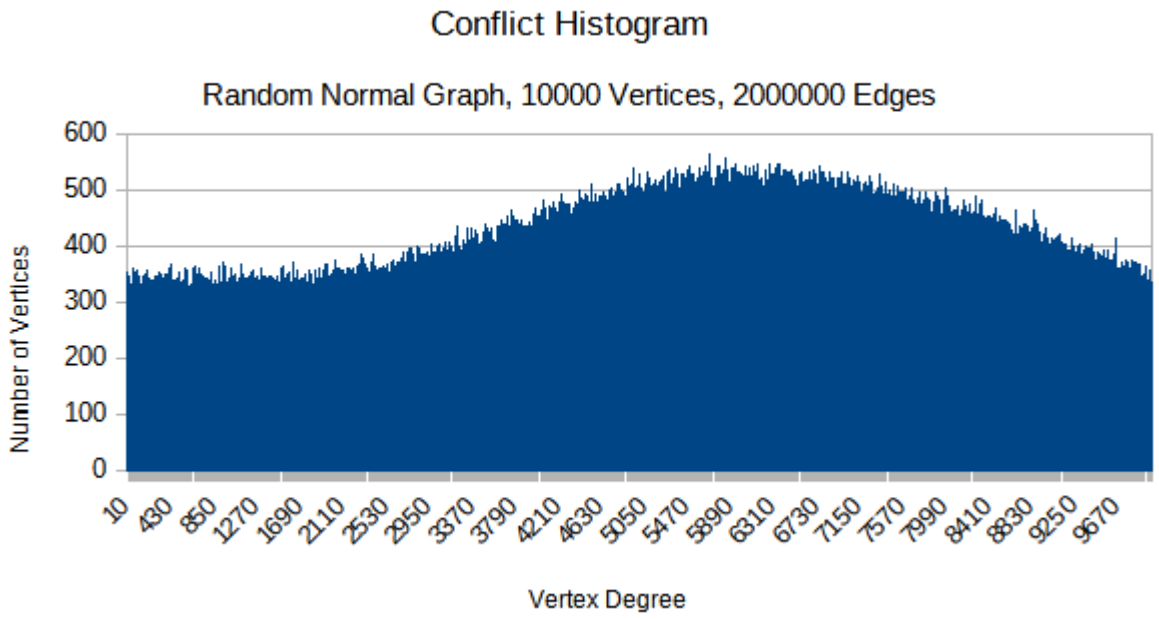
The difference between each random graph generator lies in edge generation methods. The uniform generator uses the standard library uniform_int_distribution to generate integers in the range from one to the square of the specified vertex count.



The skewed generator uses a uniform_real_distribution to randomly generate floating point values from zero to one. This double is raised to the 1.25th power then multiplied by the square of the vertex count and floored to generate at an edge to be unpaired.



The normal generator uses the standard library lognormal_distribution to generate floating point numbers in the range from one to the squared vertex count. This floating point value is multiplied by the square of the vertex count to create an edge.



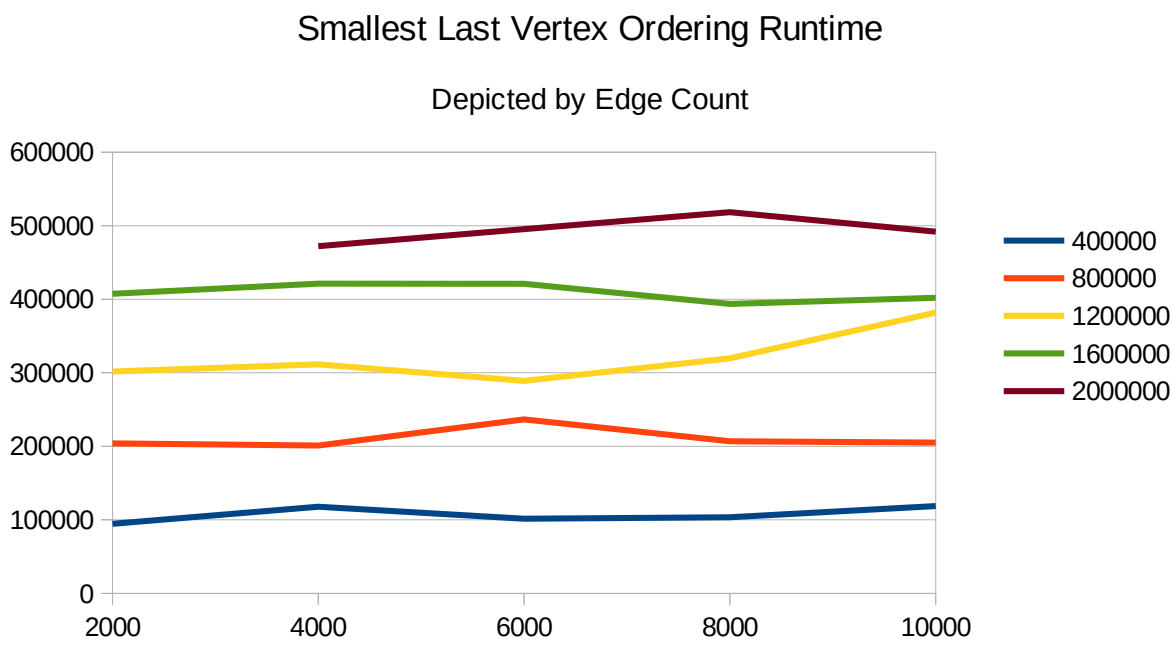
Vertex Ordering

The smallest last vertex ordering implementation first prepares auxiliary data structures to track the degrees of vertices as the algorithm runs. A vector of integers stores the current degree of a vertex, a vector of lists stores what vertices are of a certain degree, and a final vector of list iterators stores pointers to each vertex's location in the previous data structure. Finally, a boolean vector tracks which vertices have been removed. These data structures require time linear in the number of vertices to initialize. The implementation selects and removes a vertex of lowest degree in constant time until all vertices have been removed. After each removal, the implementation iterates through that vertex's neighbors to reduce the degree of each by one with constant time operations.

The outermost loop of the implementation iterates once for every vertex of the graph. The next two nested loops handle the operations to select the first vertex of lowest degree. The innermost loop iterates twice for each edge in the graph. Thus, the time complexity of this algorithm should be O(V+E). The chart below appears to support linear complexity with respect to the number of edges in a graph. However, effect of vertex count on runtime is lost to fluctuations in the execution of the program.

Smallest Last Vertex Ordering Runtime

V\E	400000	800000	1200000	1600000	2000000
2000	94617	203976	301659	407463	-
4000	117797	201000	311514	421297	472167
6000	101475	236616	289147	421014	495376
8000	103577	206830	319757	393668	518480
10000	118806	205203	381791	401984	491833



The smallest original degree ordering selects vertices in ascending order of degree. A vector of lists is generated in linear time to store the vertices of every possible degree. The

implementation then selects and removes vertices from lowest to highest degree until the data structure is empty. This implementation iterates twice for each vertex; its runtime is O(V).

The uniform random ordering places every vertex of the graph into a standard template unordered_set, which requires linear time to do so. The implementation then selects and removes the first element of this set until it is empty. The selection process, while deterministic, is effectively pseudo-random because the C++ standard template unordered_set is implemented internally using hash tables.

The largest original degree ordering selects vertices in decreasing order of degree. A vector of lists is generated in time linear to the number of vertices to store the vertices of every possible degree. The implementation then selects and removes vertices from highest to lowest degree until the data structure is empty. Given two separate loops that each iterate over the vertices of the graph, the runtime complexity is O(V).

Breadth first search ordering and depth first search ordering both select vertices in the order that they are traversed by graph search algorithms. The former prepares a queue and boolean vector to track explored vertices then performs breadth first search. The latter prepares a stack and boolean vector then performs depth first search. Both implementations run in O(V+E) time complexity.

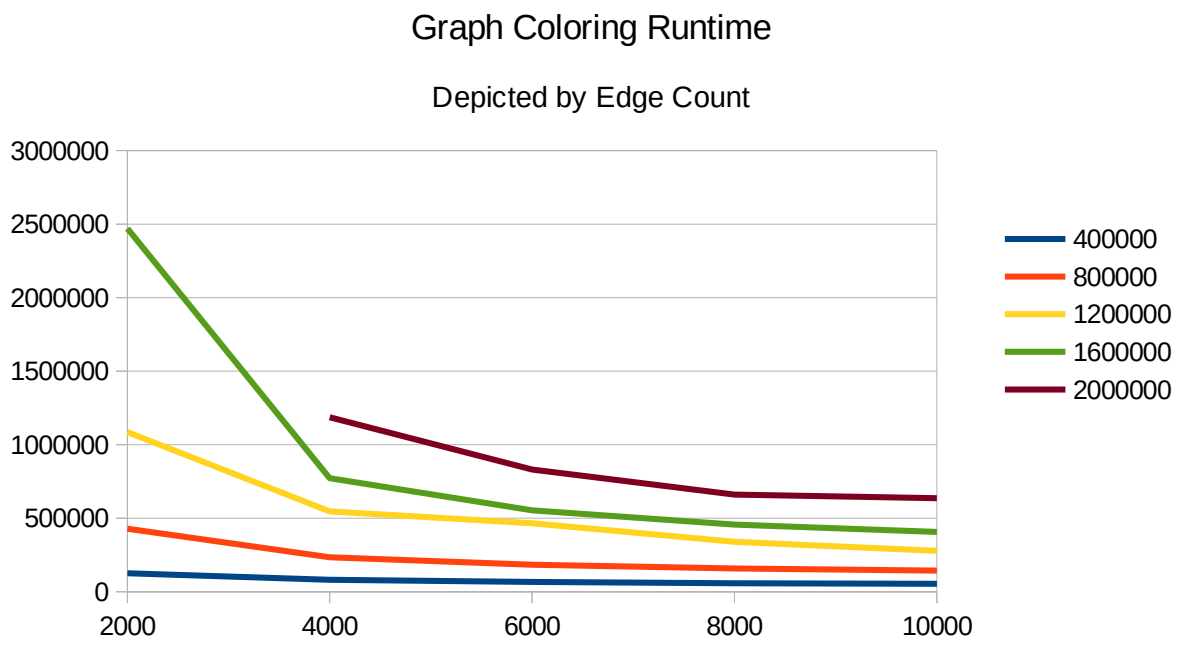
Coloring Algorithm

The greedy coloring implementation iterates through every single vertex of the ordering on the outermost loop. The next nested loop iterates through the colors to find a valid color to assign. The innermost nested loop iterates through the edges of the vertex to check if any neighboring vertices have already been assigned the color. The time complexity of this

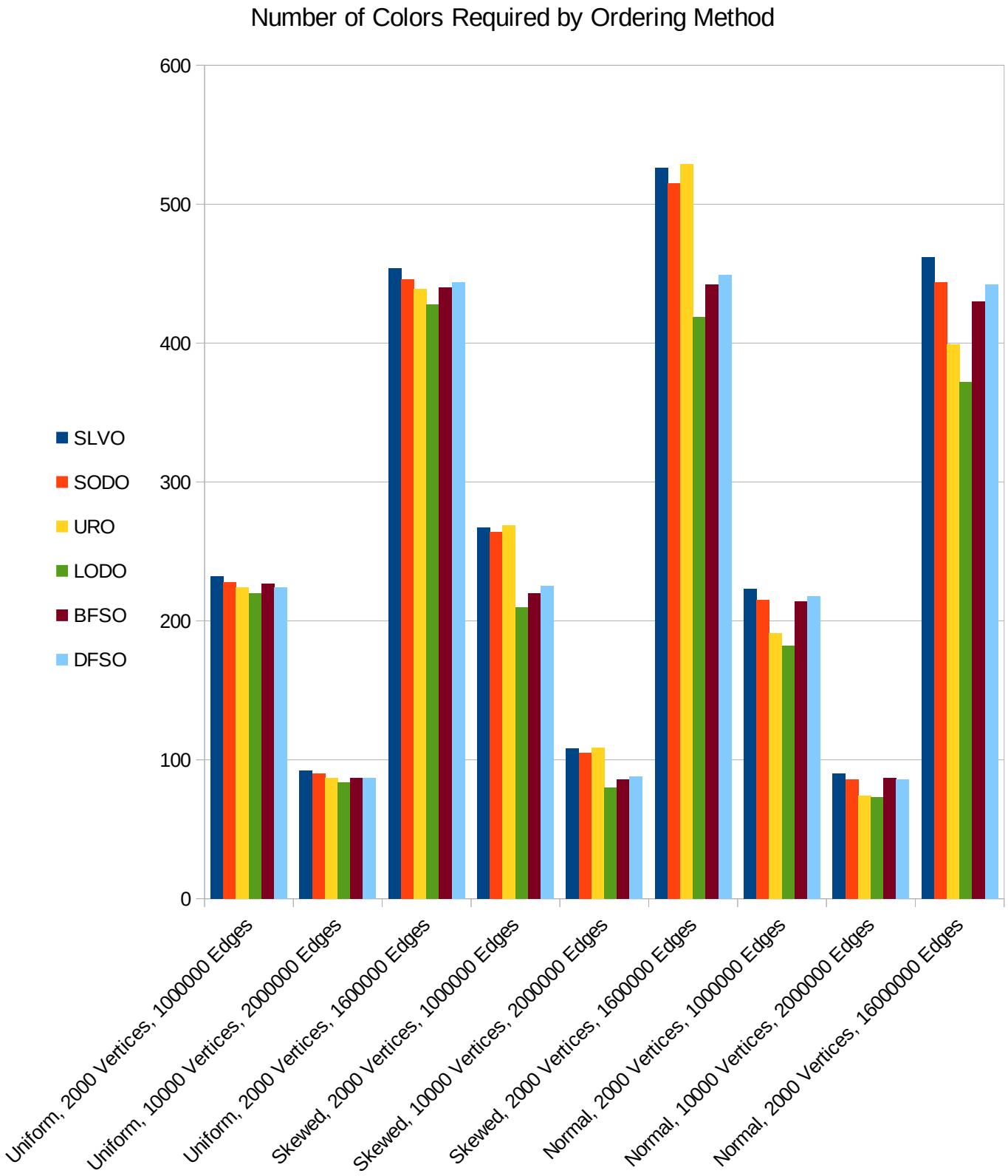
implementation is then O(V + C * E)) where C is the number of colors required to color the graph. In the worst case of complete graphs, the number of colors is equivalent to the number of vertices and the time complexity degenerates to O(V+VE). But for less densely connected graphs, fewer colors are iterated through for each vertex. At best, a graph with no edges will only require one color. Thus the best case asymptotic time complexity of this implementation is O(V+E). The chart below shows how the runtime suffers greatly for the graphs with the least vertices relative to the number of edges because more colors must be iterated through for each vertex.

Graph Coloring Runtime

V\E	400000	800000	1200000	1600000	2000000
2000	125952	429376	1085784	2470535	-
4000	81725	235022	546313	772437	1186835
6000	67690	183911	466136	553697	831723
8000	58195	158486	340347	457027	661026
10000	54135	143982	278301	406863	636121



Vertex Ordering Capabilities



Colors Required by Ordering

Graph\Order	SLVO	SODO	URO	LODO	BFSO	DFSO
Complete, 2000 Vertices	2000	2000	2000	2000	2000	2000
Cycle, 2000 Vertices	2	2	2	2	2	2
Uniform, 2000 Vertices, 1000000 Edges	232	228	224	220	227	224
Uniform, 10000 Vertices, 2000000 Edges	92	90	87	84	87	87
Uniform, 2000 Vertices, 1600000 Edges	454	446	439	428	440	444
Skewed, 2000 Vertices, 1000000 Edges	267	264	269	210	220	225
Skewed, 10000 Vertices, 2000000 Edges	108	105	109	80	86	88
Skewed, 2000 Vertices, 1600000 Edges	526	515	529	419	442	449
Normal, 2000 Vertices, 1000000 Edges	223	215	191	182	214	218
Normal, 10000 Vertices, 2000000 Edges	90	86	74	73	87	86
Normal, 2000 Vertices, 16000000 Edges	462	444	399	372	430	442

The ordering methods were used to color a variety of graphs and compared to each other on the metric of colors used. As expected, every ordering method performs poorly on a complete graph and requires as many colors as there are vertices. Also expect was the use of just two colors for a cycle for a cycle graph. The remainder of the test graphs were randomly generated, among which the largest original degree ordering consistently outperformed the other methods.

Of note is how smallest last vertex ordering performed poorest among the algorithms. More detailed data was generated when performing smallest last vertex ordering on the most densely connected skewed graph above. Below is a chart indicating the degree of a vertex when it is deleted by smallest last vertex ordering. The maximum degree of a deleted vertex was 1449, which is an upper bound on the number of colors required. At the point of its removal, the vertex of degree 1449 possesses the most neighboring vertices and sets the limit on how many colors are eliminated due to adjacency. Any previously removed vertex would not increase the limit, otherwise it would have been the vertex of maximum degree. The terminal clique

contained 100 vertices. Because the terminal clique is a maximally connected subgraph, every vertex of the terminal clique requires a different color from the other vertices in the clique. Therefore, 100 is a lower bound on the colors required.