Tim Singel

CS-5350

5350 Final Report

Introduction:

In this final project, we have been tasked with building, serializing, ordering, and coloring several sets of graphs. This will all be done to analyze our runtimes, and hopefully come out with a clear understanding of how it scales compared to input size.

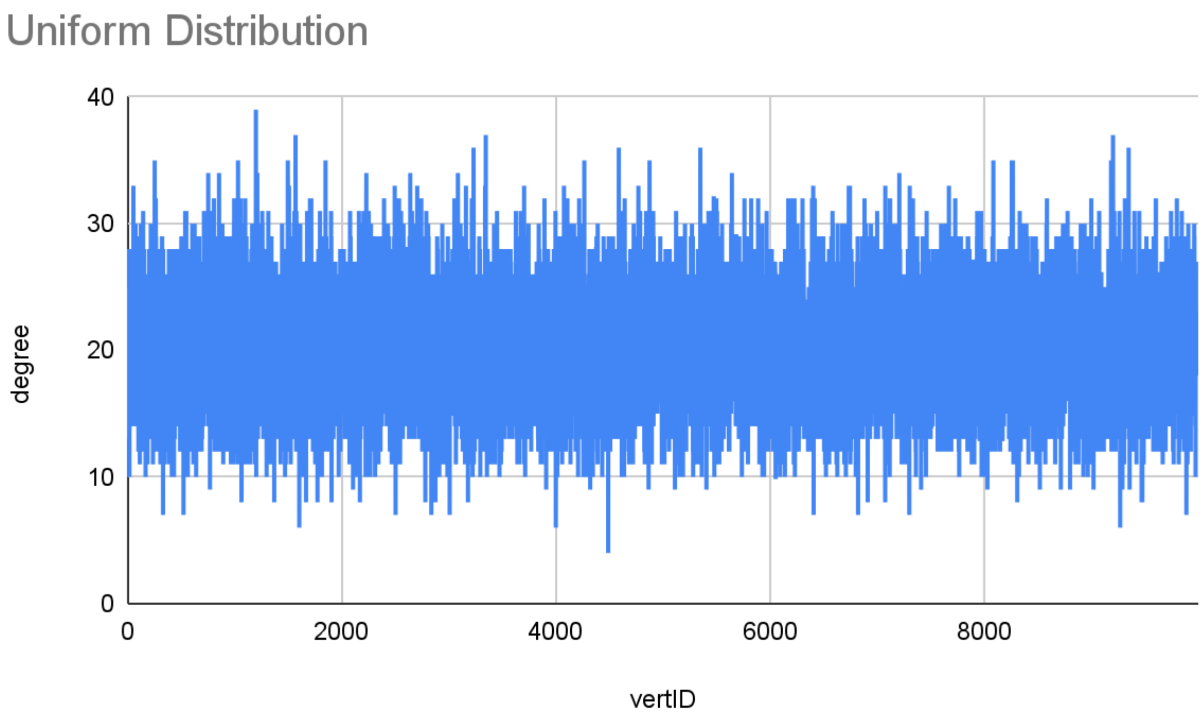
Environment:

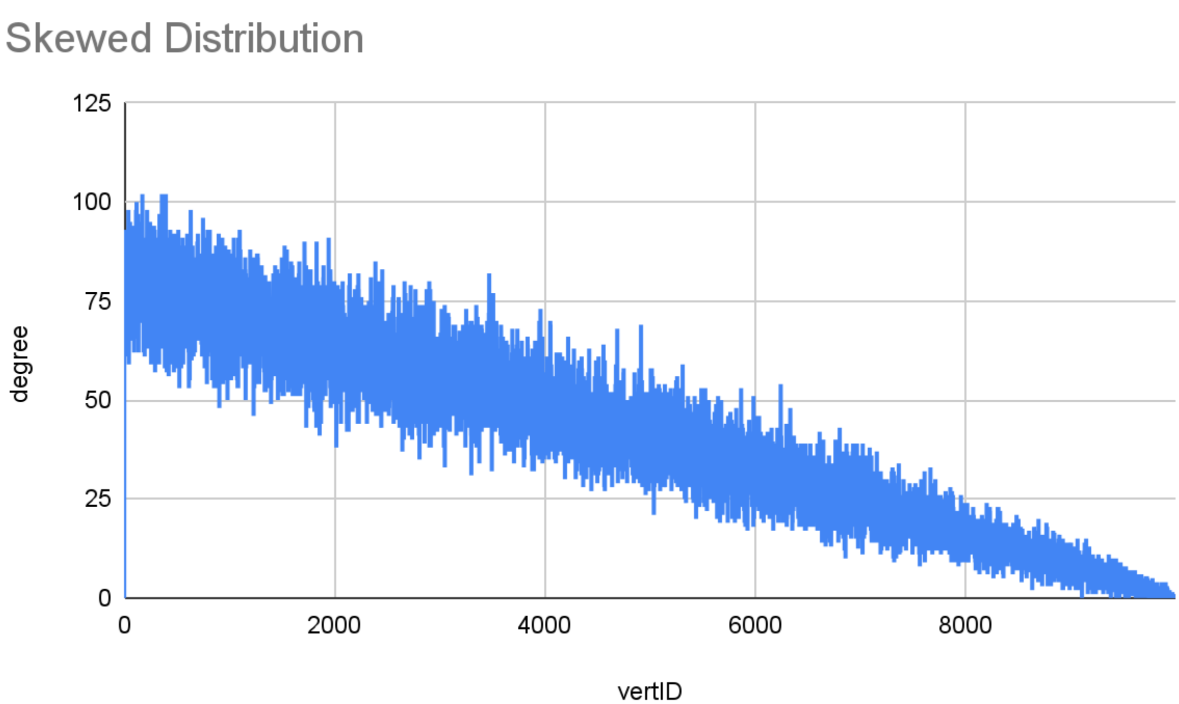
I made the crucial decision of language early on, after spending some serious time thinking about it. I decided on using python, as I am still very new to it myself, and felt I would take the plunge, and try to make my code as fast as it could be in this language.

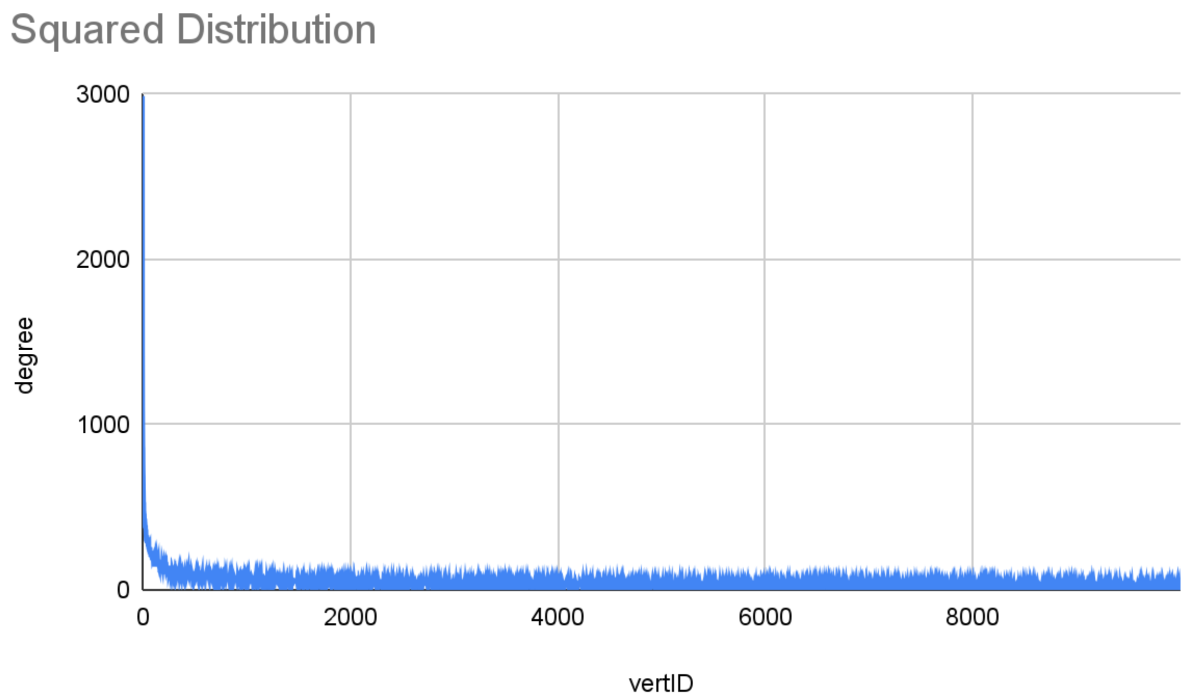
I coded it across several devices, using git to track my process and changes. I made sure that any change I made had no cascading effects, and would ensure any of my code would allow for modularity with any of my coloring and ordering techniques down the line.

Random Distribution Types:

On the following pages are the random distributions supported by my graph generation techniques.


Uniform Distribution
degree
40
30
20
10
0
0
2000
4000
6000
8000
vertID


Skewed Distribution
degree
125
100
75
50
25
0
0
2000
4000
6000
8000
vertID

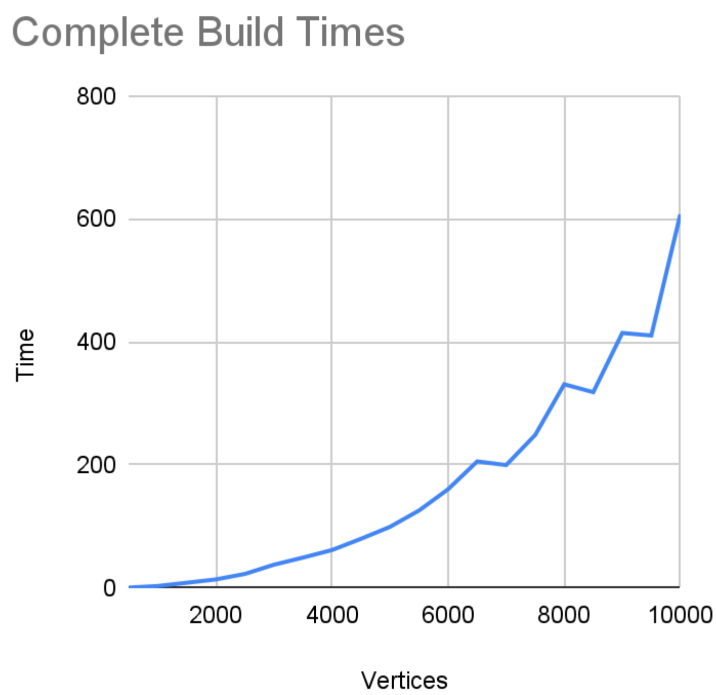


Graph Generation Runtimes:

The following tables and graphs were generated using a shell script to run them in order using > to inject it to a csv. I would then loop this with several different V or E values given whichever output I was going for.

Complete Graphs:

Vertices	Time
500	0.565956831
1000	3.141708374
1500	8.448401451
2000	13.82903528
2500	22.78555989
3000	37.7728467
3500	49.33704376
4000	61.59974337
4500	79.92107153
5000	99.35481358
5500	125.9474072
6000	160.4970536
6500	205.5820093
7000	199.7333093
7500	248.7258615
8000	330.9938192
8500	318.2370021
9000	414.4258409
9500	410.3291762
10000	607.2947783

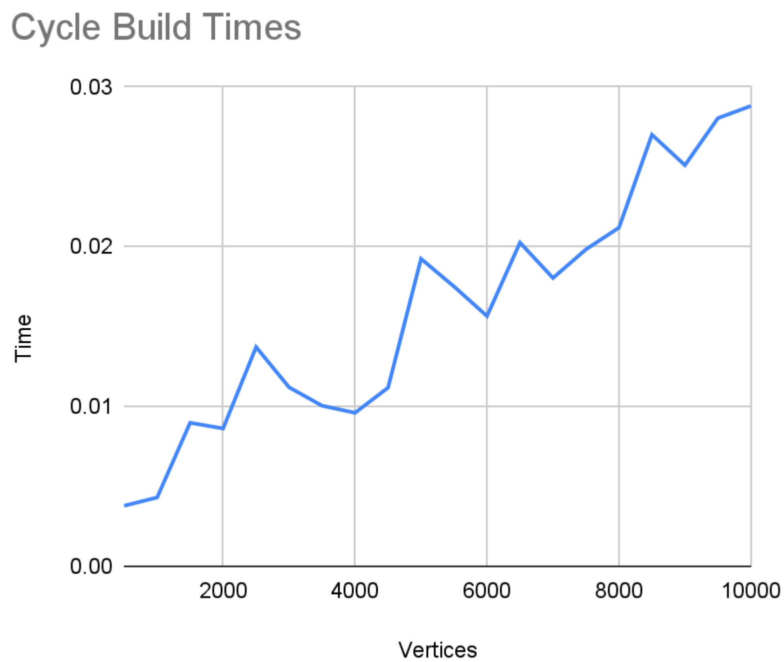


Analysis:

Based on my tabled runtimes and the graph, I have a feeling that my complete generation is among the slowest. My sincere guess is O(V^2), as the most extreme cases have extreme runtimes. Even running on the schools server with an intel xeon, 10000 which is around 50 million edges took nearly 10 minutes. I would say this is the best run I got, as at times the server froze up the python instance. This graph shows the change of Vertices by the runtime. I ended up with O(V^2)

Cycle Graphs:

Vertices	Time
500	0.003776311874
1000	0.00429391861
1500	0.008969545364
2000	0.008611440659
2500	0.01370668411
3000	0.01118707657
3500	0.01003718376
4000	0.009592533112
4500	0.01116490364
5000	0.01922893524
5500	0.01749515533
6000	0.015645504
6500	0.02024102211
7000	0.0180284977
7500	0.01981878281
8000	0.02117800713
8500	0.02699398994
9000	0.02508664131
9500	0.02802944183
10000	0.02879357338

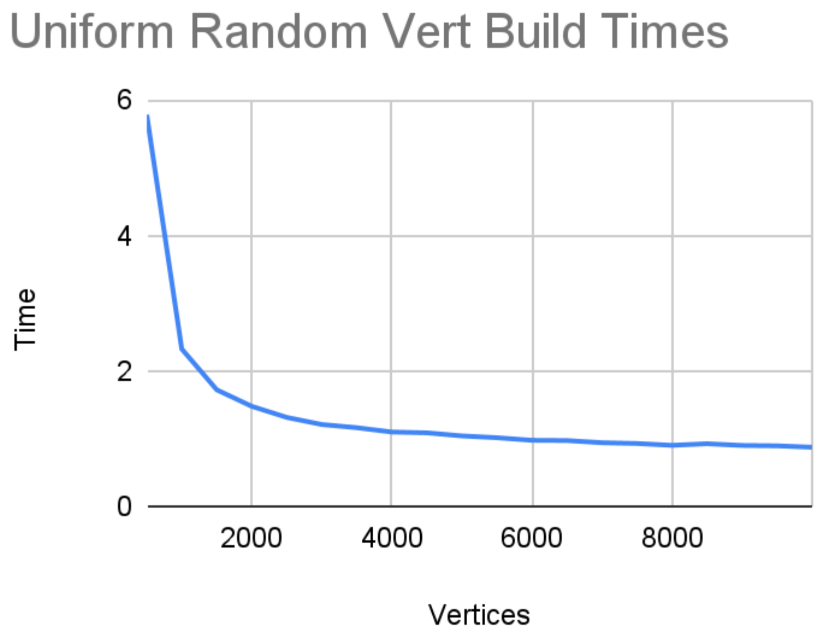


Analysis:

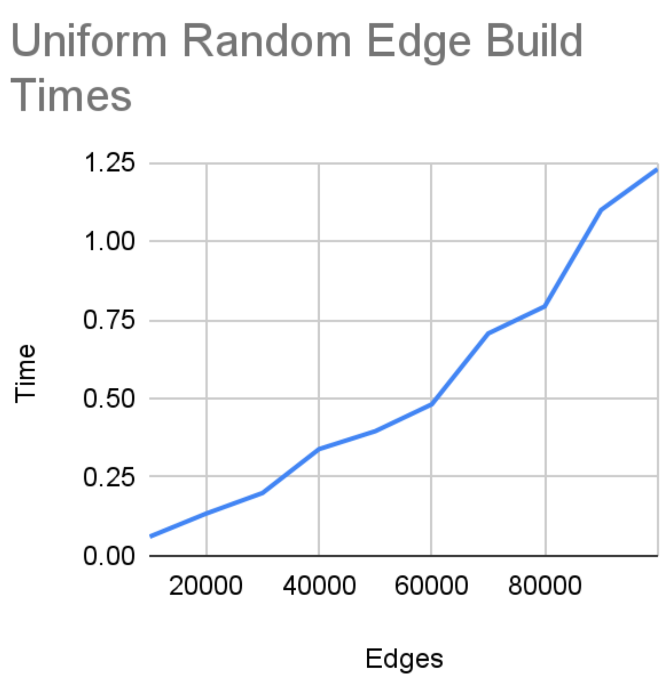
From my graphs and data outputs, the performance gain is clear between this and complete, and I would say that immediately this is O(V) runtime, as it goes to each vertice in order, and adds to cycle in what I would say is about linear time, meaning the main attribute scaling time is the vertices count. The strange up down variance is most likely from variance in the computer itself, not a programmatic error.

Uniform Random Graphs (Vert/Edge):

Vertices	Time
500	5.784392118
1000	2.325575113f
1500	1.722511053
2000	1.478859663
2500	1.315148115
3000	1.210270882
3500	1.163339376
4000	1.099345684
4500	1.087580919
5000	1.042124271
5500	1.01512599
6000	0.9773306847
6500	0.9734556675
7000	0.9415352345
7500	0.9298424721
8000	0.9031300545
8500	0.9264605045
9000	0.9013941288
9500	0.895537138
10000	0.8748147488

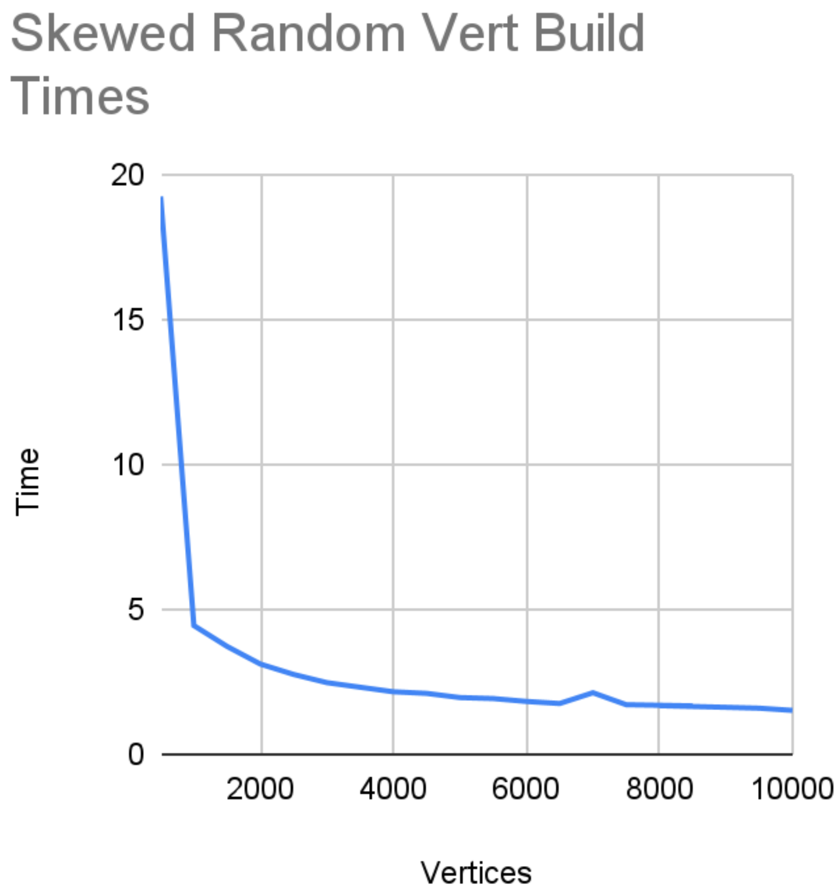


Edge	Time
10000	0.0600066185
20000	0.1334102154
30000	0.1989719868
40000	0.3384768963
50000	0.3957509995
60000	0.4812150002
70000	0.7065689564
80000	0.7925348282
90000	1.099574089
100000	1.229662895

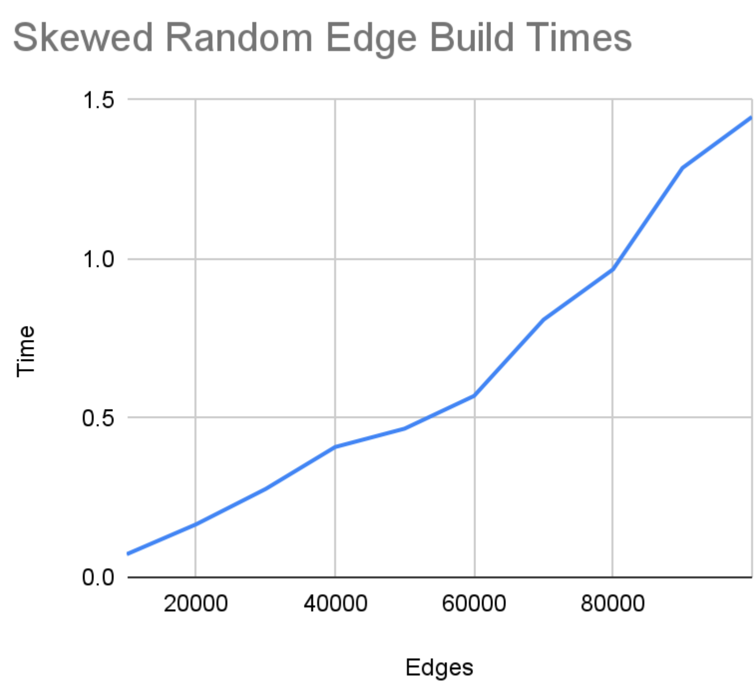


Skewed Random Graphs:

Vertices	Time
10000	1.5178895
9500	1.594171047
9000	1.624655008
8000	1.681574821
8500	1.674163818
7500	1.71496892
7000	2.12772274
6500	1.755957127
6000	1.826788425
5500	1.926962852
5000	1.960597754
4500	2.105140209
4000	2.16060257
3500	2.317932129
3000	2.475925922
2500	2.759656906
2000	3.117156506
1500	3.726086378
1000	4.446886778
500	19.26128769

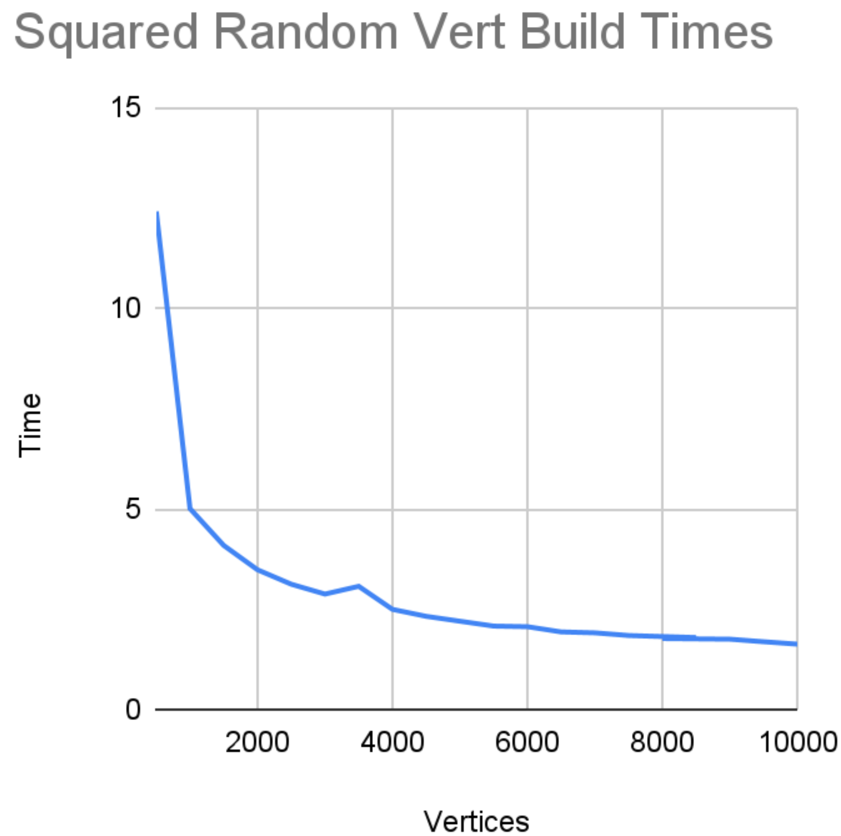


Edges	Time
10000	0.07255530357
20000	0.1665816307
30000	0.2775835991
40000	0.4093325138
50000	0.4670917988
60000	0.5698983669
70000	0.8094027042
80000	0.9674723148
90000	1.286313295
100000	1.446612835

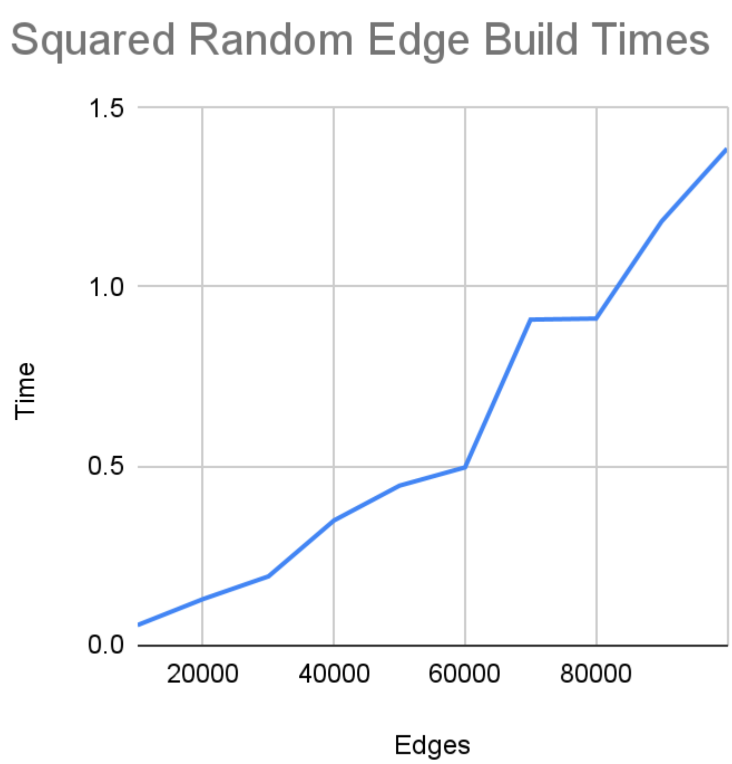


Squared Random Graphs:

Vertices	Time
10000	1.633201361
9500	1.693156242
9000	1.756319761
8000	1.772512197
8500	1.79771328
7500	1.848349094
7000	1.91458869
6500	1.937674046
6000	2.067599773
5500	2.08323288
5000	2.204091072
4500	2.32565093
4000	2.499406338
3500	3.073340178
3000	2.880060911
2500	3.125814438
2000	3.485306501
1500	4.09101367
1000	5.008760929
500	12.40857577



Edges	Time
10000	0.05501270294
20000	0.1276676655
30000	0.191190958
40000	0.3474667072
50000	0.4446377754
60000	0.4957227707
70000	0.9085805416
80000	0.9114742279
90000	1.182811499
100000	1.386120319



Analysis:

From my analysis and limited understanding of random generation complexity, my guess is that all of these random techniques will follow an O(E/V^2), as the edges are very impactful to the performance given there are little vertices, but given there is a sufficient vertex count, the runtime should scale down even with the same high edge counts. This equation may not be exact, but from my guess, E should be the heavy time sink, and V should in some way affect the weight of E given there's enough Vertices to spread them across

Ordering Techniques:

Smallest Last Vertex Ordering:

The small last vertex order(or SLVO) is created by using a degreeList which sorts references to the vertice objects in an adjacency list in order of degree. This is fed in and parsed, adding each vertex to the ordering, and adjusting the degrees of its connected partners. At each removal and addition to the ordering, the vertex saves its degree when removed, and goes on.

Test Walkthrough on Small Set:

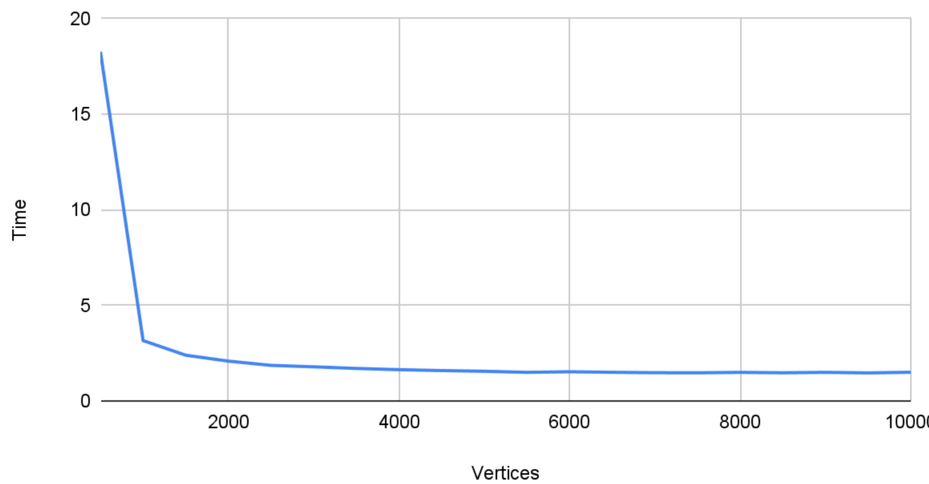
The example below starts with 5 vertices and 10 edges, and for each step it prints the vertices info, and if it has been removed or not. At the end I demonstrate what the SLVO coloring would be for the same graph.

```
VertID: 0, Degree: 4, Removed: False }->3->4->2->1
VertID: 1, Degree: 4, Removed: False }->4->3->0->2
VertID: 2, Degree: 4, Removed: False }->0->4->3->1
VertID: 3, Degree: 4, Removed: False }->0->4->1->2
VertID: 4, Degree: 4, Removed: False }->0->1->3->2
Step #:  1
VertID: 0, Degree: 3, Removed: False }->3->4->2->1
VertID: 1, Degree: 3, Removed: False }->4->3->0->2
VertID: 2, Degree: 3, Removed: False }->0->4->3->1
VertID: 3, Degree: 3, Removed: False }->0->4->1->2
VertID: 4, Degree: 4, Removed: True }->0->1->3->2
Step #:  2
VertID: 0, Degree: 2, Removed: False }->3->4->2->1
VertID: 1, Degree: 2, Removed: False }->4->3->0->2
VertID: 2, Degree: 3, Removed: True }->0->4->3->1
VertID: 3, Degree: 2, Removed: False }->0->4->1->2
VertID: 4, Degree: 4, Removed: True }->0->1->3->2
Step #:  3
VertID: 0, Degree: 1, Removed: False }->3->4->2->1
VertID: 1, Degree: 2, Removed: True }->4->3->0->2
VertID: 2, Degree: 3, Removed: True }->0->4->3->1
VertID: 3, Degree: 1, Removed: False }->0->4->1->2
VertID: 4, Degree: 4, Removed: True }->0->1->3->2
Step #:  4
VertID: 0, Degree: 1, Removed: True }->3->4->2->1
VertID: 1, Degree: 2, Removed: True }->4->3->0->2
VertID: 2, Degree: 3, Removed: True }->0->4->3->1
VertID: 3, Degree: 0, Removed: False }->0->4->1->2
VertID: 4, Degree: 4, Removed: True }->0->1->3->2
Step #:  5
VertID: 0, Degree: 1, Removed: True }->3->4->2->1
VertID: 1, Degree: 2, Removed: True }->4->3->0->2
VertID: 2, Degree: 3, Removed: True }->0->4->3->1
VertID: 3, Degree: 0, Removed: True }->0->4->1->2
VertID: 4, Degree: 4, Removed: True }->0->1->3->2
Vert#:  0  Color:  3
Vert#:  1  Color:  2
Vert#:  2  Color:  1
Vert#:  3  Color:  4
Vert#:  4  Color:  0
```

SLVO Runtime Examination:

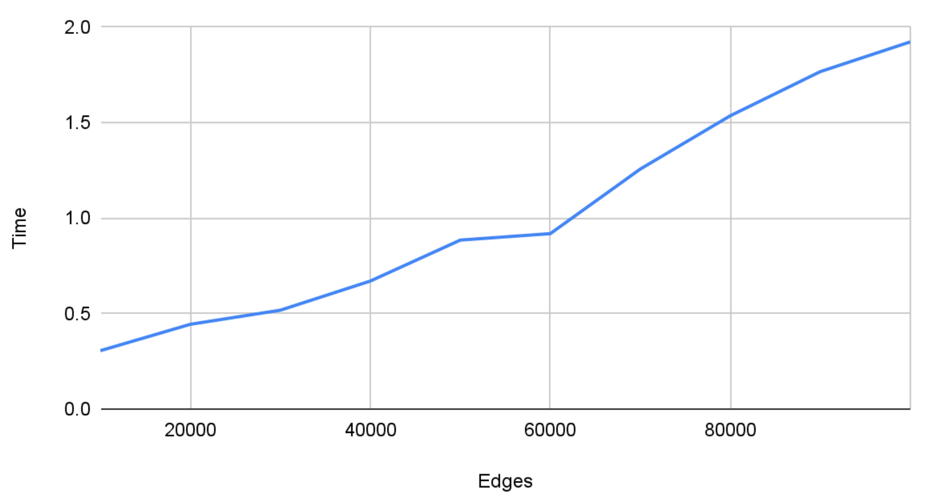
Vertices	Time
500	18.26828384
1000	3.159342527
1500	2.393642187
2000	2.08405304
2500	1.86239171
3000	1.786045551
3500	1.701081514
4000	1.636190891
4500	1.58955574
5000	1.552648783
5500	1.496366739
6000	1.527740002
6500	1.499153614
7000	1.477698326
7500	1.472055435
8000	1.495309591
8500	1.4739573
9000	1.496223927
9500	1.467093468
10000	1.501807451

SLVO Runtime with 100K Conflicts



Edges	Time
10000	0.3057024479
20000	0.4438335896
30000	0.5176608562
40000	0.6704156399
50000	0.8849031925
60000	0.9186694622
70000	1.256967545
80000	1.536068678
90000	1.767205954
100000	1.922997475

SLVO Runtime with 10K Vertices



SLVO Analysis:

From the document, I know the algorithm if done correctly is in O(V+E), and I am fairly certain that my graphs and tables support this. The second graph with scaling Edge counts shows a somewhat O(n) style graph, and the program only really has issues when trying to add edges that are near the max conflict for a certain number of vertices. The edge increase is clear to me, as when you scale that alone in a fairly large graph, its runtime scales linearly. SLVO is unique from my other orderings, as it does more work when removing vertices, going to each neighbor to update the conditions there. This I believe is O(V+E) because of the linear increase shown in my second table and graph.

Smallest Original Vertex Ordering:

This as mentioned in the document is a subset of SLVO, and operates in a similar way at first. It uses the degree list to know the order of the vertices based on degree, but differently to SLVO when removing a vertice, it does not go and change the degrees of neighboring nodes, and instead is essentially the degree list transitioned to a 1 dimensional array, going through every list at each degree and pushing them. This makes it rather fast as it is more of a parse. Still keeps a rather low amount of colors, but not exactly the same as SLVO

Smallest Original Inverted Ordering:

I originally intended on flipping SLVO ordering, but when I got around to it, it was extremely slow in a cascading sense. My guess is that it has much more difficulty working with the high degree nodes first, and will then have difficulty with the low degree that should have been easy. I instead made an invert of the Smallest Original, as all I had to ensure was that the parser could be reversed. It is still rather slow, as the cascading effect still happens when coloring high degree nodes first, but it is a fun twist that I came up with for my 4th coloring method.

Uniform Random Vertex Ordering:

This was the simplest for me to implement, given the fantastic python libraries that exist. All that you want here is to make a deep copy of your adjacency list object, holding your Vertices, and then shuffle the contents and simply return that as your ordering. It had the most variance in runtime, and at times it could be pretty quick. For me this seems to be efficient with its copy, as it does not parse in any way, so my guess is that I have it randomizing about as efficiently as possible in my environment. This will often produce high color counts in general, but at times it can get lucky and produce near the optimized amount.

Coloring Technique:

All of the ordering techniques are fed into the greedy coloring algorithm as an input. Then the greedy algorithm parses the ordered vertices, examining every edge for used colors nearby, then choosing from the nodes that were not found in neighboring nodes. It is rather fast, and allows for the order to express itself in the colors chosen.

Vertex Ordering Capabilities:

Complete Graphs:

When being colored, complete are slow, but for good reason, as in any case where you color a complete graph, you will end with the same amount of colors as your vertices count. All the orderings produce different stuff, but it has no effect, as in coloring it's all about distinguishing from your neighbor, but if every single node is your neighbor, every single node will have to be different from each other.

Cycle Graphs:

When testing the color counts on Cycle graphs, I found that there were obvious patterns presenting themselves across the board.

When coloring graphs with even vertices count, I found that my non random ordering techniques only colored with 2 colors for all even cycles. But my Uniform Random Ordering almost always produced 3 colors for an even vertex count. My assumption is that it could produce 2, but it would be random producing one of the other 3 orders.

When coloring the odd vertices count, all methods of ordering always had 3 colors.

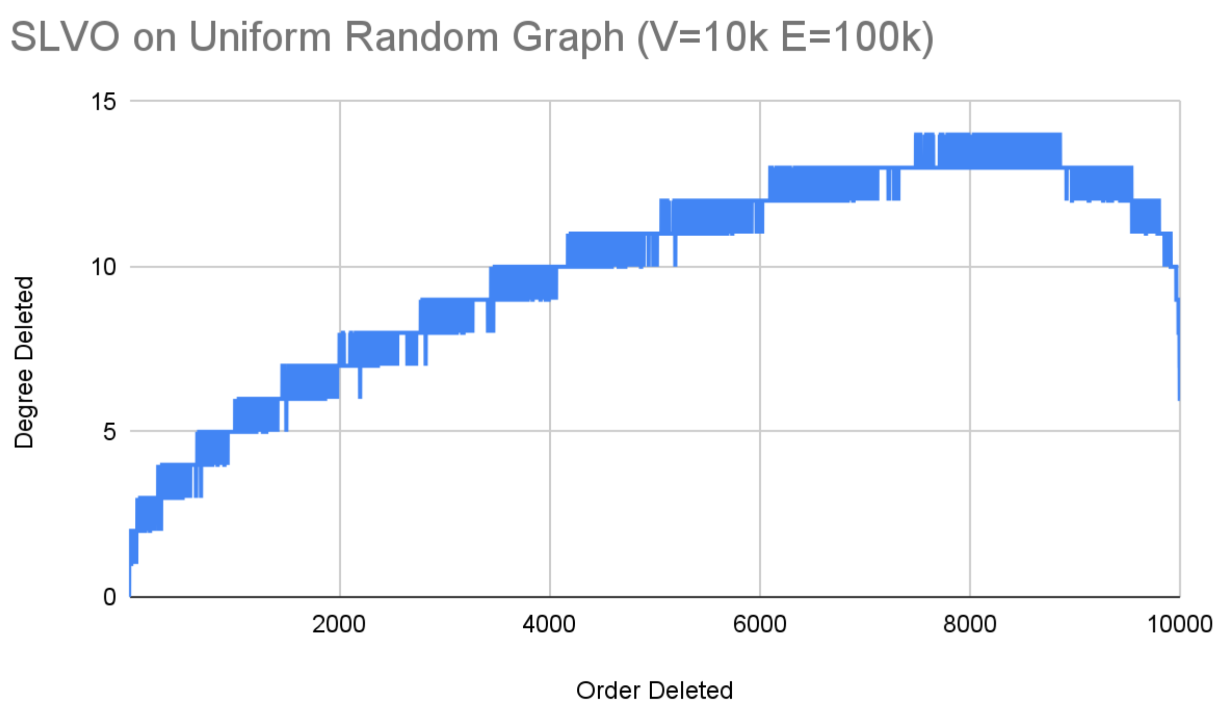
Random Graphs:

I started this part by serializing a file for all 3 random distributions with 10k vertices and 100k edges. I then tested all the coloring techniques on each of the graphs, so that I could compare the color counts given for one particular graph.

Uniform Graph:

Smallest Last Vertex Ordering:

Number of colors: 11



Max Degree Deleted: 14

Terminal Clique Size: 2

Smallest Original Vertex Ordering:

Number of Colors: 11

Smallest Original Vertex Ordering Inverted:

Number of Colors: 13

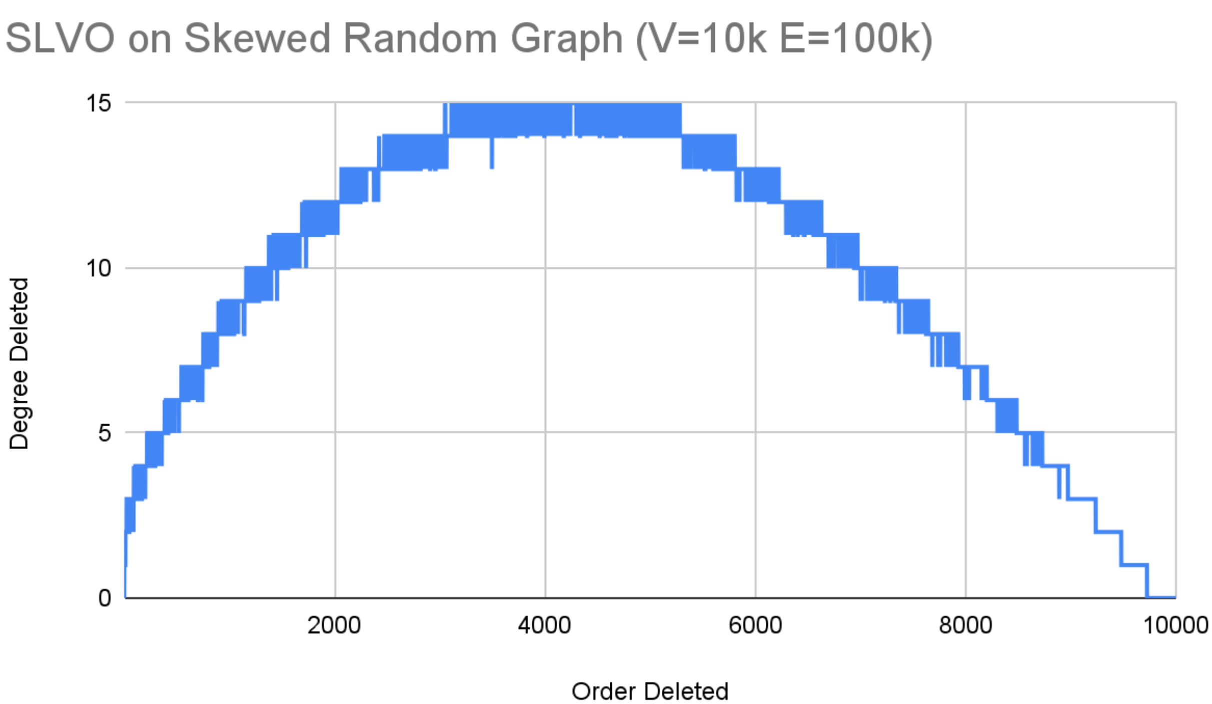
Uniform Random Ordering:

Number of Colors: 12

Skewed Graph:

Smallest Last Vertex Ordering:

Number of Colors: 11



Max Degree Deleted: 15

Terminal Clique Size: 3

Smallest Original Vertex Ordering:

Number of Colors: 11

Smallest Original Vertex Ordering Inverted:

Number of Colors: 17

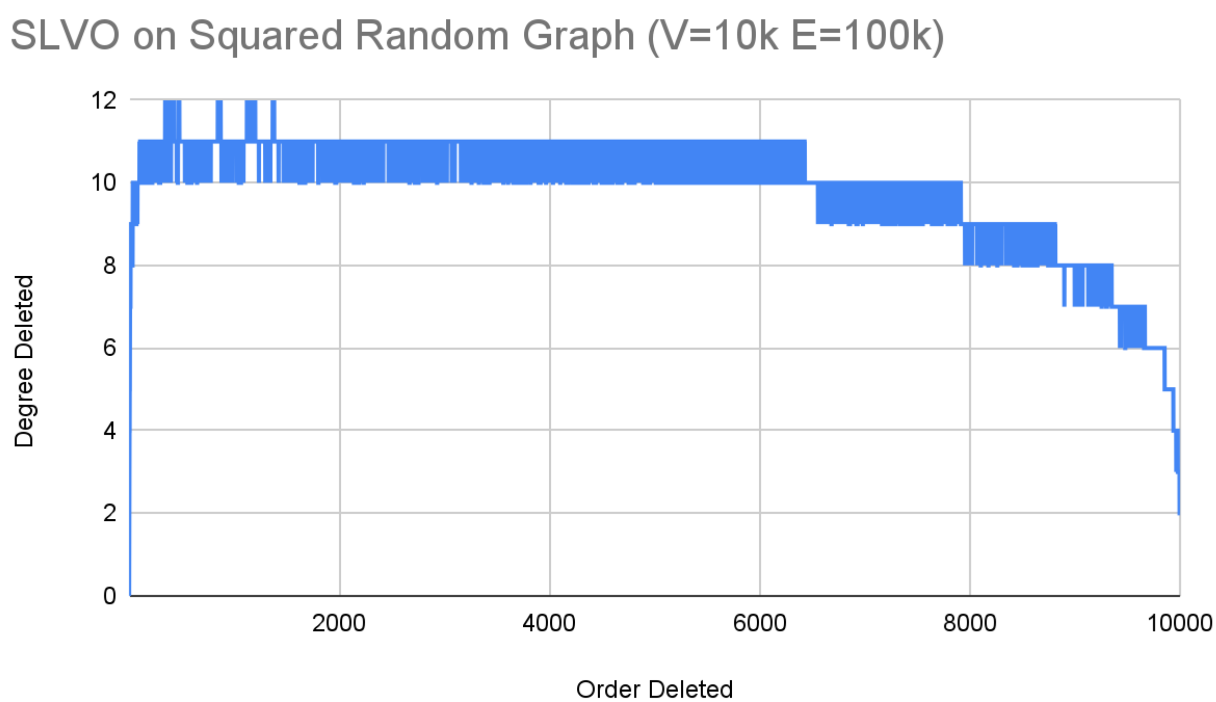
Uniform Random Ordering:

Number of Colors: 14

Squared Graph:

Smallest Last Vertex Ordering:

Number of Colors: 11



Max Degree Deleted: 12

Terminal Clique Size: 5

Smallest Original Vertex Ordering:

Number of Colors: 11

Smallest Original Vertex Ordering Inverted:

Number of Colors: 26

Uniform Random Ordering:

Number of Colors: 19

Ordering Results:

Smallest Last Vertex Ordering: Clearly the best, and if this was truly a tool I was selling, this would be the default. It clearly outperforms the others(except Smallest Original at times), and orders it with a more complicated technique, to allow later decisions to be more informed than the other orderings. It does have 1 tiny drawback, and that is complexity, as removing a vertex requires more memory reading that just ordering them randomly, but it does not matter if you want to optimize your colors, which random does not.

Smallest Original Vertex Ordering: This is my runner up to SLVO, as they both produced the same color set on the graphs I created above. It should also be faster than SLVO, but my guess is its a scaling issue for much larger graphs. I do not see the color count matching SLVO for all possible graphs, but I cannot prove that. In concept SLVO should be more optimal because of its choice to edit connected nodes.

Smallest Original Vertex Ordering INVERTED: This is my personal coloring method, and by far the worst. In every case I have tested above or during development, this always produced the highest color count. In a sense if SOVO is an optimized order, this is the intentionally unoptimized version. This is mainly here to show how important ordering is when you are coloring a graph. If this was a sales pitch, I would only use this as an example of orders importance.

<u>Uniform Random Vertex Ordering</u>: This was a surprise hit in my opinion, and did not result in the worst color set by a long shot. It does obviously produce a less optimized set, but I feel that it isn't sabotage. The inverted ordering always performed worse than this, showing me that greedy coloring alone can overcome a somewhat unoptimized ordering, but not an intentionally bad one. Its main benefit is its speed, as it simply just shuffles your adjacency list and gives the original and the shuffle to the greedy coloring.

Conclusion:

I would say that if this was a product for a ~10000 node system, that Smallest Last Vertex Ordering and Smallest Original Vertex Ordering would be the best options that a user could make. Smallest Last Vertex Ordering is obviously the best, but given you do want a little bit less read/write, Smallest Original takes a very simplified approach. The random is not unusable, but I'd advise against it, as it could produce (in theory) the absolute worst order for a graph. Finally I would say to not use the inverted Smallest Original method, as it is the worst and least effective method for doing this kind of work. If you want the optimal output, use SLVO, but if you are concerned about sheer processing power, Smallest Originals simple parse approach will help lower the complexity of the program overall.