Graph Coloring Analysis Project

Mark M. Brubaker

Southern Methodist University

Graph Coloring Analysis Project

This project looks at implementing an algorithm in multiple ways to solve graph coloring, analyzing the algorithms' implementations for running time, and testing and the implementations to show they match the analysis. The project also looks at testing to determine how well the multiple implementations solve the problem. The first part of the project is creating different conflict graphs which can represent real-world problems. For the second part of the problem, various coloring orders are analyzed for runtime efficiency and coloring efficiency. These efficiencies are intimately related to the data structures used for both the input graph data and the intermediate data necessary for the ordering and coloring algorithms.

Environment

All data was collected on a CPU Intel(R) Core(TM) i5-2500K CPU @ 3.30GHz desktop with 16GB of RAM. All code was compiled in x86 release mode in Visual Studio 2019 and times with custom functions using the C++ chrono library.

All code included in this report is taken from the source code, which can be found in appendix B. The code in the report may not exactly match the source code, as code that was not directly responsible for the function of the excerpt was removed. All removed lines were constant in time and do not significantly affect the runtime of the program. Code that was cut down was done with the intent of improving conciseness, readability, and clarity. The core functionality of each excerpt is preserved in the edits.

Conflict Graphs

Five unique strategies were used to generate the conflict graphs. All of the needed space used by these functions was allocated before the functions called except the memory for the

edges themselves. The vertices were also created beforehand, each with an ID one greater than the one before, starting at zero. Each edge is directed and dynamically allocated when it is created. The following function was used to create the edges:

```
void Graph::AddEdge(const int vertex1, const int vertex2) {
	Edge* e = new Edge(vertex2, edges[vertex1]);
	edges[vertex1] = e;
	vertices[vertex1].degree++;
	vertices[vertex1].edges[vertex2] = true;
}
```

This function runs in constant time and is used by every conflict graph generation function. It first creates the new edge by assigning the vertex it points to and the next edge in the adjacency list. Each edge only stores the id of the vertex it is directed towards and a pointer to the next edge in the list. The vertex it comes from is determined by which list it is in. Finally, it updates some information stored by its origin vertex. Because this function creates directed edges it is always called twice to make the edge in each direction.

As can be seen in the above function each vertex has its own list of which edges it is connected to in the form of a vector of bools. When an edge is created the bool at the index of the neighbor is set to true. This vector is only used so that the conflict creation functions can check if an edge already exists in constant time. Originally this check was done by walking the

adjacency list, but this change increased the speed of the whole program by an order of magnitude.

The following functions also make use of random numbers. These numbers are generated by c++'s Mersenne Twister engine on a uniform real distribution. The number is generated in constant time. A custom class wraps the engine and the class has a function that returns a number in a specified range. The range is inclusive of the lower bound but not the upper bound.

For the following generation strategies, all data will be from an example graph with 200 times as many edges as vertices. Datapoints were sampled in increments of 500 vertices. When timed, ten new graphs of each size were generated and the minimum time was plotted. The histograms are generated from a graph with 10,000 vertices and 2,000,000 edges to get the best representation of each method. All related tables and charts can be found in appendix A and the links to all raw data collected can be found in appendix B.

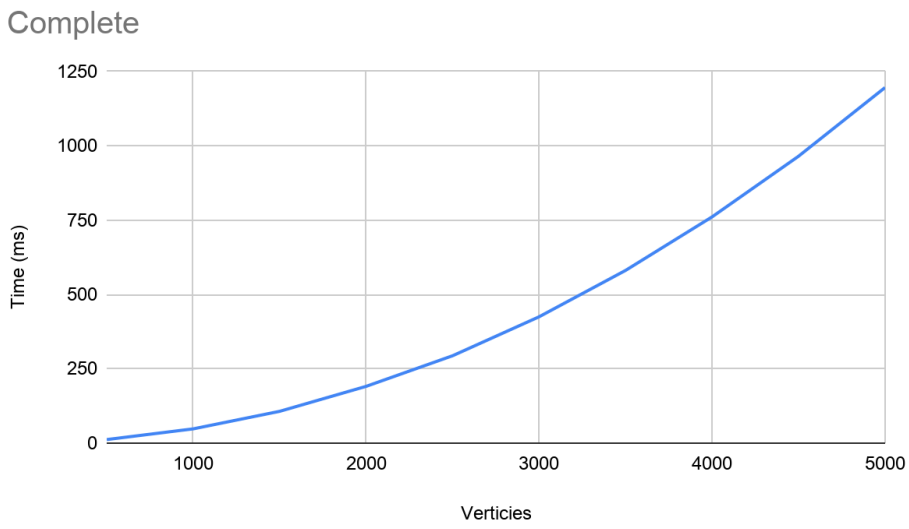
Complete Graph

The complete graph has an edge from every vertex to every other vertex. This results in every vertex having exactly n - 1 neighbors.

```
void Graph::CreateCompleteGraph() {
     for(int x = 0; x < size; x++) {
          for(int y = 0; y < size; y++) {
               if(x == y) continue;
               AddEdge(x, y);
          }
```

```
    }
}
```

This code runs in $O(n^2)$ as it needs to create a connection to every vertex for each vertex in the graph. This can be seen in the following graph where the time taken quadruples every time the number of vertices doubles.



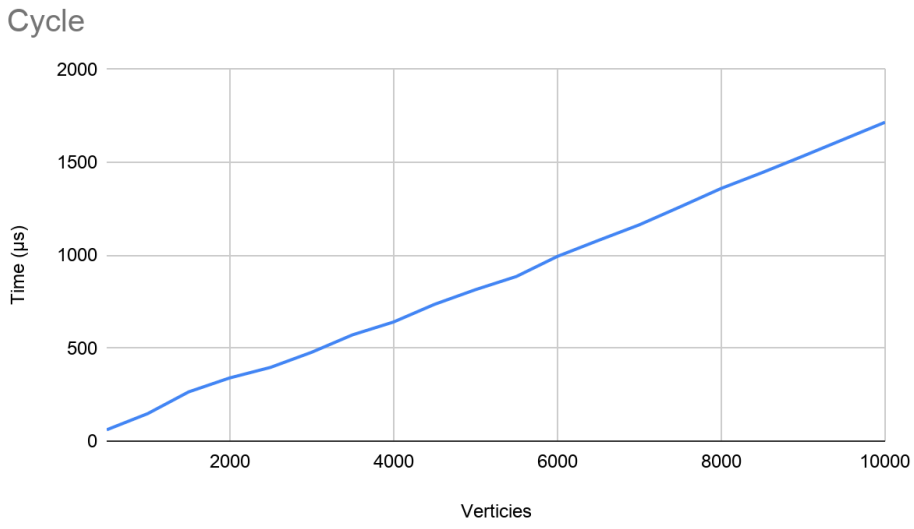
Cycle

The cycle graph is formed of a single cycle that contains every vertex in the graph. This is formed by iterating over every vertex and connecting it to the next one. Finally, the last vertex is connected back to the start.

```
void Graph::CreateCycle() {
    for(int x = 0; x < size - 1; x++) {
        AddEdge(x, x + 1);
```

```
            AddEdge(x + 1, x);
        }
        AddEdge(size - 1, 0);
        AddEdge(0, size - 1);
}
```

This function runs in O(n) because it iterates over every vertex exactly once. This is represented in the following graph where the time increases proportionally to the number of vertices linearly. It's also worth noting that this is the only distribution recorded in microseconds, opposed to milliseconds.



Even Distribution

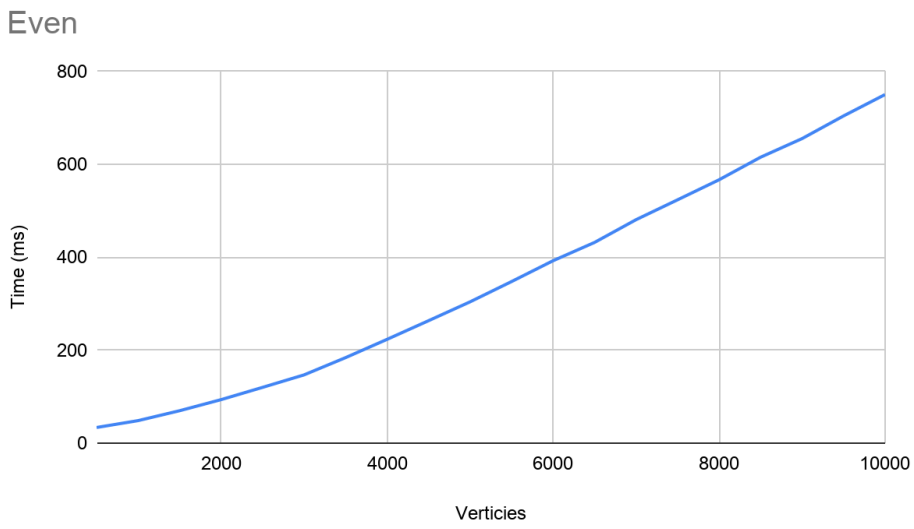
The even distribution takes a set number of edges and randomly distributes them among all vertices. This is done by selecting two random edges and checking if they already have an

edge and adding one if they don't. This evenly distributes the edges so that the degree of each vertex is roughly the same.

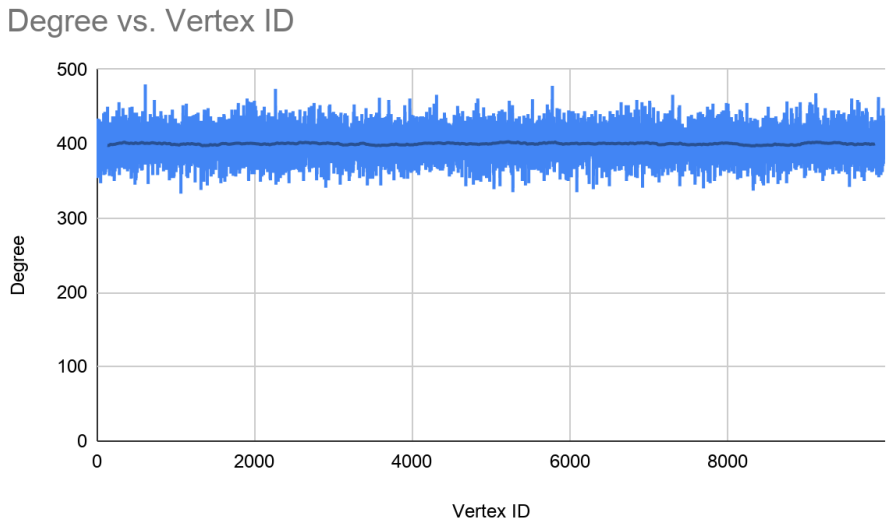
```
void Graph::CreateEvenDistribution(const int E) {
	for(int x = 0; x < E; x++) {
		int v1 = Random::getRandomNumber(0, size);
		int v2 = Random::getRandomNumber(0, size);
		//Edge already exists and a new one need to be chosen
		if(v1 == v2 || vertices[v1].edges[v2]) {
			x--;
			continue;
		}
		AddEdge(v1, v2);
		AddEdge(v2, v1);
	}
}
```

The worst case for this code is O(∞) which happens when the function only tries to make edges between a vertex and itself or edges that already exist. The average runtime approaches Θ(n) for very large values of n as collision get become less and less likely. This is because the number of edges for all tests is exactly 200 times the number of vertices and so it scales linearly

where the number of possible edges or the number of edges in a complete graph scales quadratically.



An expected distribution will result in each vertex having a similar degree. This can be seen below in an example graph generated this way. The thin line is a trendline based on a centered average of the surrounding 250 vertices.



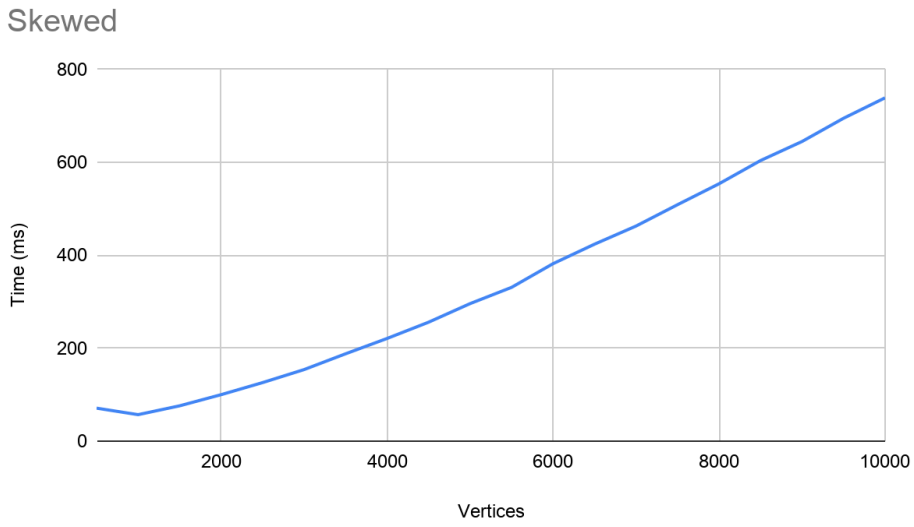
Skewed Distribution

The skewed distribution takes a set number of edges and randomly distributes them among all vertices on a skewed distribution where vertices with a lower ID have a higher chance of being selected. This is done by selecting two random edges on the skewed distribution and checking if they already have an edge and adding one if they don't. This distributes the edges so that the degree of each vertex is linearly related to its ID. The skew is achieved by passing a random number into the cumulative distribution function of downward sloping line and then scaling it to the size of the graph.

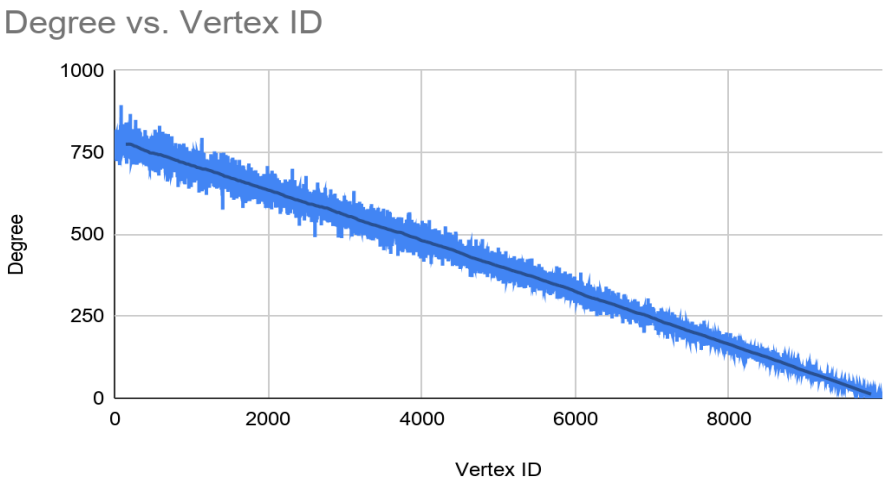
```
void Graph::CreateSkewedDistribution(const int E) {
	for(int x = 0; x < E; x++) {
		int v1 = size * (1 - sqrt(Random::getRandomNumber(0, 1)));
		int v2 = size * (1 - sqrt(Random::getRandomNumber(0, 1)));
		//Edge already exist and a new one need to be chosen
		if(v1 == v2 || vertices[v1].edges[v2]) {
			x--;
			continue;
		}
		AddEdge(v1, v2);
		AddEdge(v2, v1);
	}
```

```
}
```

Similarly to the even distribution, the worst case for this code is O(∞) which happens when the function only tries to make edges between a vertex and itself or edges that already exist. Likewise, the average runtime approaches Θ(n) for very large values of n as collision get become less and less likely for the same reason as before.



As expected, an average result of this function is a graph where low numbered vertices have the highest degrees and high numbered vertices have a degree close to zero.

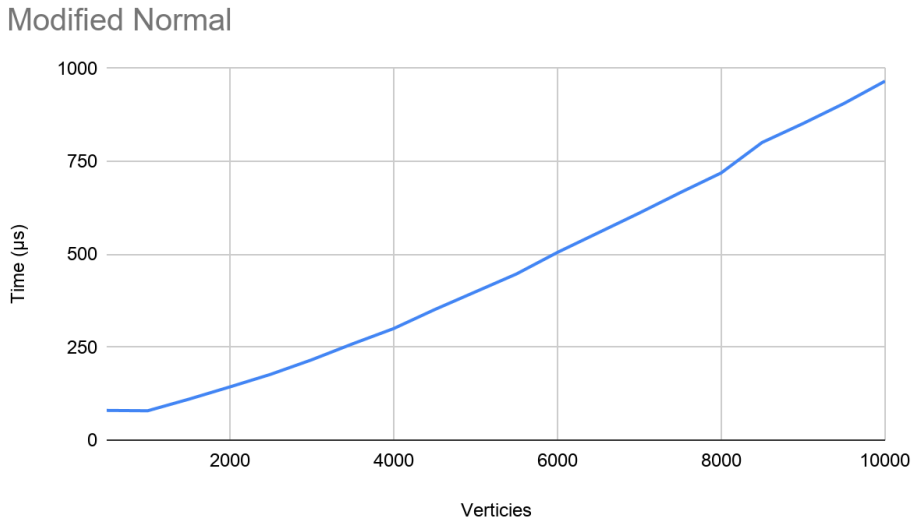


Modified Normal Distribution

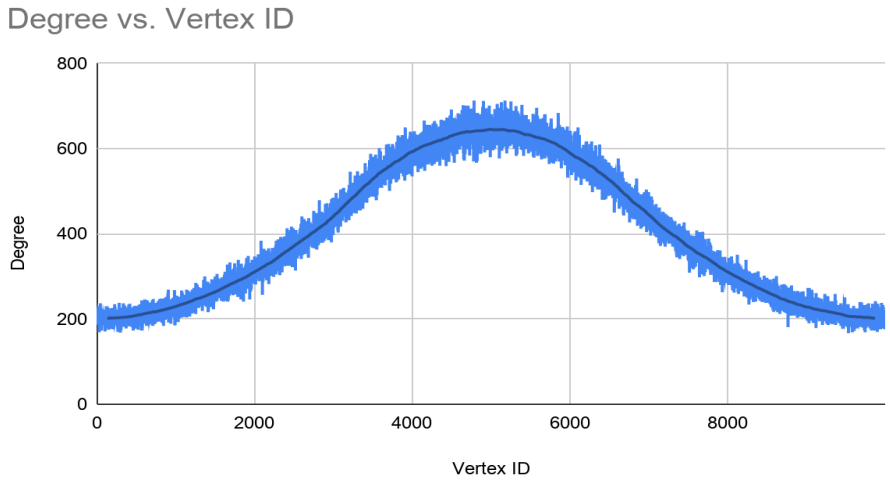
The modified normal distribution takes a set number of edges and randomly distributes them among all vertices on a modified normal distribution where one vertex is selected at random and the second is selected on a normal distribution. This results in each vertex having a minimum degree, and some vertices having a significantly higher degree. Originally both vertices were selected on a normal distribution but this resulted in very long graph creation times because collisions with existing edges were very likely when attempting to add a new edge. The modified normal distribution approaches linear run time much faster than a pure normal distribution allowing more reasonable run times for graphs of the expected size for this program. The normal distribution is achieved by generating three random numbers and taking their average. Taking the average of more numbers results in numbers close to the mean being selected more often. Three was selected because a relatively wide distribution was desired to minimize collisions and risk running into the same problem as a pure normal distribution. The width of the distribution can be seen in the distribution histogram below.

```
void Graph::CreateModifiedNormalDistribution(const int E) {
	for(int x = 0; x < E; x++) {
		int v1 = (int)Random::getNormalNumber(size);
		int v2 = (int)Random::getRandomNumber(0, size);
		//Edge already exist and a new one need to be chosen
		if(v1 == v2 || vertices[v1].edges[v2]) {
			x--;
			continue;
		}
		AddEdge(v1, v2);
		AddEdge(v2, v1);
	}
}
```

Similarly to the previous two distributions, the worst case for this code is O(∞) which happens when the function only tries to make edges between a vertex and itself or edges that already exist. Likewise, the average runtime approaches Θ(n) for very large values of n as collision get become less and less likely for the same reason as before. As compared to the skewed distribution, generating these graphs took slightly longer as the chance for a collision is higher.



Because one vertex is chosen at random and the other is chosen on a normal distribution, an average result will look like a normal distribution stacked on top of an even distribution. This can be seen in the graph below.



Vertex Ordering

Six vertex ordering algorithms were implemented, each with a different goal in mind. These algorithms are: random, smallest last vertex, smallest original degree, largest last vertex, outside in, and breadth first search.

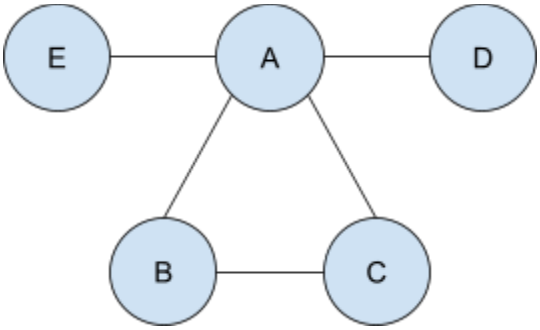
Random

The random ordering aims to provide a baseline ordering that the rest can be compared to. Theoretically, nothing needs to be done to the graph after it is generated to achieve this but a linear time shuffle is used to guarantee uniform randomness. If an approach to ordering performs worse than the random ordering, any time spent creating that ordering is wasted effort, as doing nothing would be better.

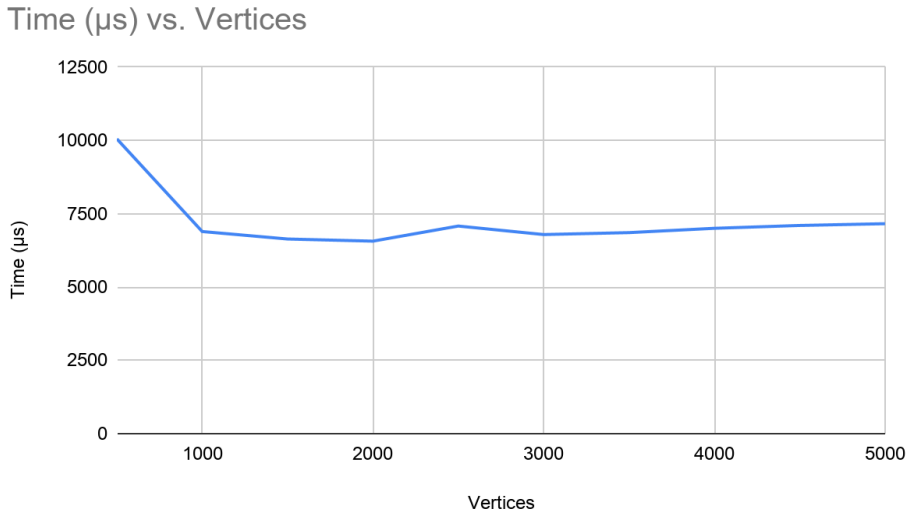
Smallest Last Vertex

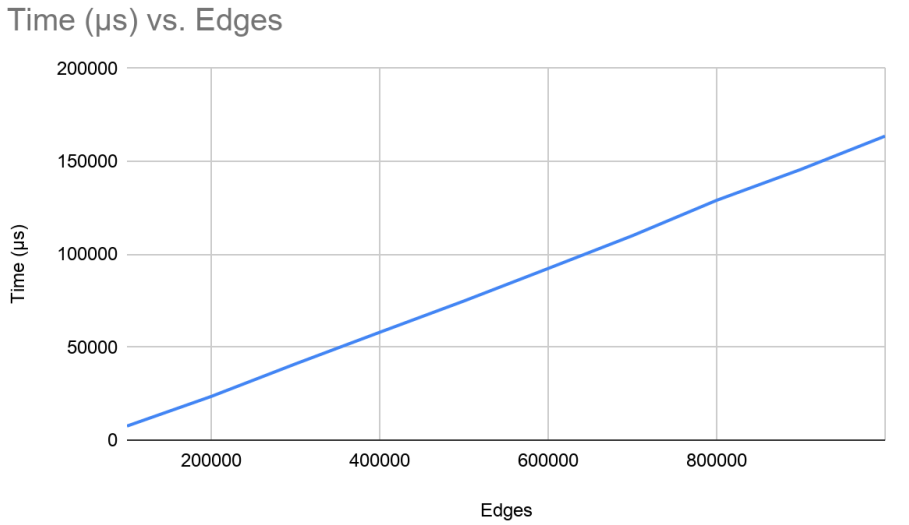
This ordering utilizes an adjacency list and a separate data structure where vertices are grouped by current degree to order the vertices referred to as a degree list. It starts by selecting the vertex with the lowest degree and removes it from the graph. Then for each of its neighbors it subtracts one from their degree and updates their place in the degree list. It then repeats this process until all vertices have been removed. The degree of the last vertex removed is kept track of so that the search for the next lowest degree vertex can be started at the previous degree minus on. This minimizes the time spent searching for the smallest degree vertex. Finally the graph is colored in the reverse order that the vertices were removed. Each vertex is moved down the degree list at maximum its degree times. The average degree can be represented as $2(E / V)$. This is done once for each vertex resulting in $O(V * (E / V))$. This simplifies to $O(E)$. An extra V term needs to be added to account for graphs with less edges than vertices as every vertex still

needs to be traversed at least once even if it has a degree of zero. Largely this V term does not have a significant impact on run time. This results in a $\Theta(V + E)$ run time. The idea behind this method is that vertices that affect the most other vertices should be colored first. A vertex in a complete graph will affect the color of every vertex in the graph, while a vertex that is disconnected from the rest of a graph will only affect itself. On the following graph SLV produces the ordering DECBA which is expected.

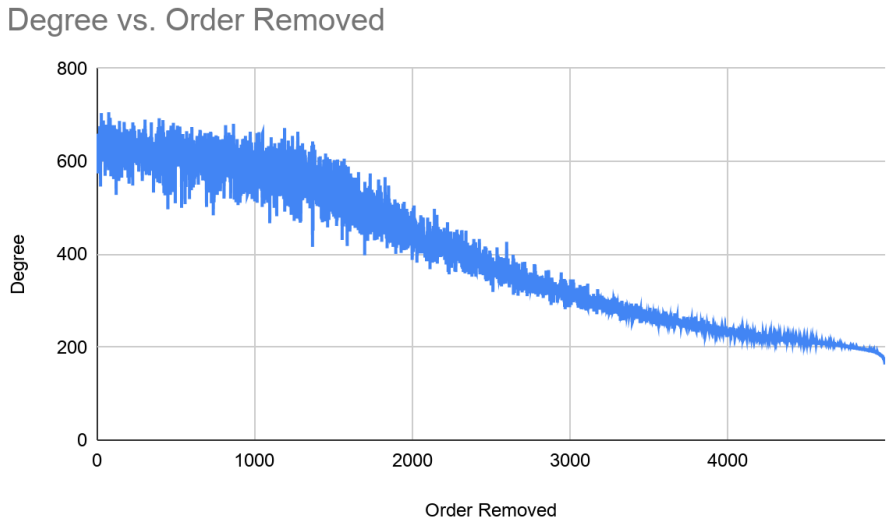


It will first remove E and D as they have the smallest degree. Then the remaining degree for every vertex will be three and they can be removed in any order as their degrees will all decrease together. The $\Theta(V + E)$ run time can be seen represented in the following graphs generated based on the minimum run time over ten trials for a modified normal distribution. The first graph linearly scales the number of vertices while holding the number of edges steady and the second graph does the reverse. The first graph shows how the V term is inconsequential and the second graph shows how run time scales on the number of edges.





The next graph shows the degree of each vertex in the order that they were removed for a 5,000 vertex 100,000 edge graph.



Smallest Original Degree Last

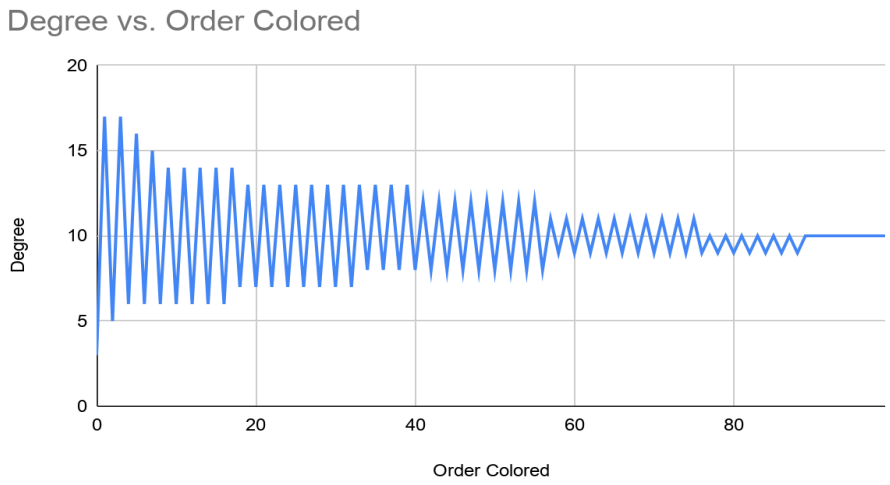
This ordering sorts the vertices from highest degree to lowest degree. This can be done in Θ(V) because it will only look at each vertex once and it keeps track of the degree of the last vertex removed like SLV. In practice, this operates like a bucket sort, which helps to see the liner run time on V.

Largest Last Vertex

This works the same as the smallest last vertex ordering, but instead of coloring the vertices in the opposite order that they were removed from the graph it colors them in order. This results in the least impactful vertices being colored first. Following that smallest last vertex is supposed to be a good ordering this is supposed to be a bad one. It is expected to do worse than random on most graphs. In theory it should do worse than random by the same amount that the smallest last vertex ordering does better than random. Additionally it should run very slightly faster than the smallest last vertex ordering because it doesn't need to reverse the order at the end.

Outside in

This is an offshoot of the smallest last original degree. It starts by constructing the same degree list but switches between removing the highest and lowest degree vertex. To better visualize this, here is a graph of degree and order colored for a simple 100 vertex 500 edge graph:



Breadth First Search

This ordering performs a breadth first search on the highest degree vertex and colors them in the order found. If a portion of the graph is fully explored but there are still undiscovered vertices, the highest remaining degree vertex is selected and a new search is performed. This repeats until all vertices have been accounted for. This tends to concentrate on coloring more dense areas of the graph first. This can break down as a graph becomes more connected as the order that neighbors are selected is arbitrary. A fully connected graph will be colored in a completely random order as every vertex will be counted after marking the neighbors of the initial vertex. Subgraphs that are long chains of vertices are guaranteed to be colored optimally

because they will be colored from one side to the other. Some of these ideas can be seen when comparing coloring order and degree in dense versus sparse graphs. The trendline in the following graphs is a centered running average across 250 vertices.

Coloring Algorithm

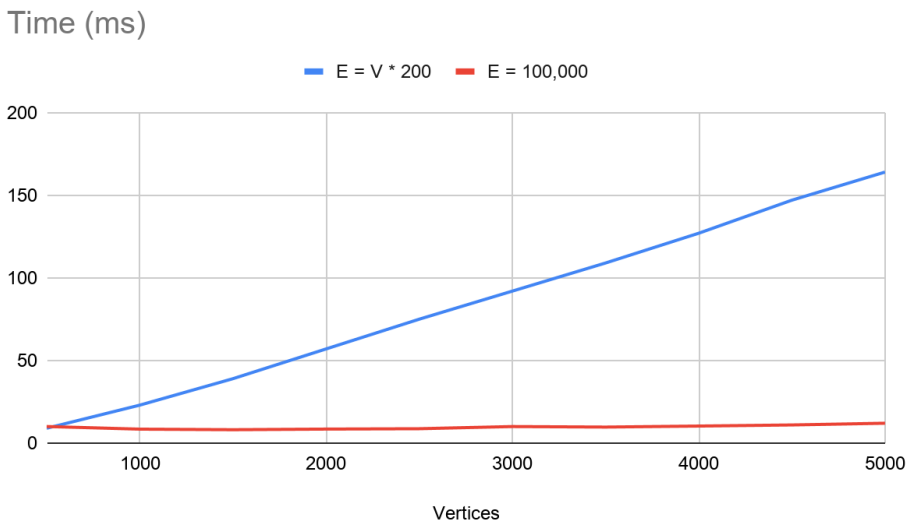
The coloring algorithm applies a color to each vertex in an order specified by an ordering algorithm. The color selected is the lowest possible color given each vertex's neighbors. This boils down to finding the smallest missing positive integer given a list of a vertex's neighbors's colors.

```
void Graph::ColorGraph() {
	for(Vertex* v : ordering) {
		v->color = Color(v);
		if(v->color > maxColor) {
			maxColor = v->color;
		}
	}
}

int Graph::Color(Vertex* v) {
	int minNeighborColor = 0;
	std::vector<int> neighborColors;

	Edge* curr = edges[v->id];
	//linear on the degree of v
	while(curr != nullptr) {
		neighborColors.emplace_back(vertices[curr->dest].color);
		curr = curr->next;
	}

	unsigned int num = 0;
	for(unsigned int x = 0; x < neighborColors.size(); x++) {
		num = neighborColors[x];
		while(0 < num && num <= neighborColors.size() &&
			neighborColors[num - 1] != num) {
			std::swap(neighborColors[x], neighborColors[num - 1]);
			num = neighborColors[x];
		}
	}
```

```
    for(unsigned int x = 0; x < neighborColors.size(); x++) {
        if(neighborColors[x] != x + 1) {
            return x + 1;
        }
    }
    return neighborColors.size() + 1;
}
```

This code runs in Θ(E) time, linear on the number of edges in the graph. The Color function runs once for each vertex and runs in linear time on the degree of the vertex. This can be seen when expanded to Θ(V * (E / V)) where the first V is running once for each vertex and (E / V) is the mean degree. In reality (E / V) is double the mean degree, but this doesn't matter for finding the asymptotic run time. In order to encompass all possible graphs the runtime needs to be Θ(V + E), for example, a graph with no edges will still traverse every vertex once. For most graphs the V term has no significant impact on the final run time. The Color function can be split into three parts. First, it constructs a list of the colors of all neighboring vertices. Next, it traverses the list looking at the value of each color and places it at the index matching the value of the color. It then does this for the color that was overwritten until the overwritten color does not belong. This reorders the list so that if an index has already been overwritten one it will be skipped over when found again. This will run in O(2n) time on the number of neighbors. Finally, the rearranged list is traversed and the function returns the value of the first index that does not match the color stored there. In total this function runs in O(4n) time. The Θ(E) can be seen in the following graph generated based on the minimum run time over ten trials for a modified normal distribution. The blue line displays a clear linear relationship between time and number of edges, which scales linearly with vertices. For the red line, the number of edges is held steady

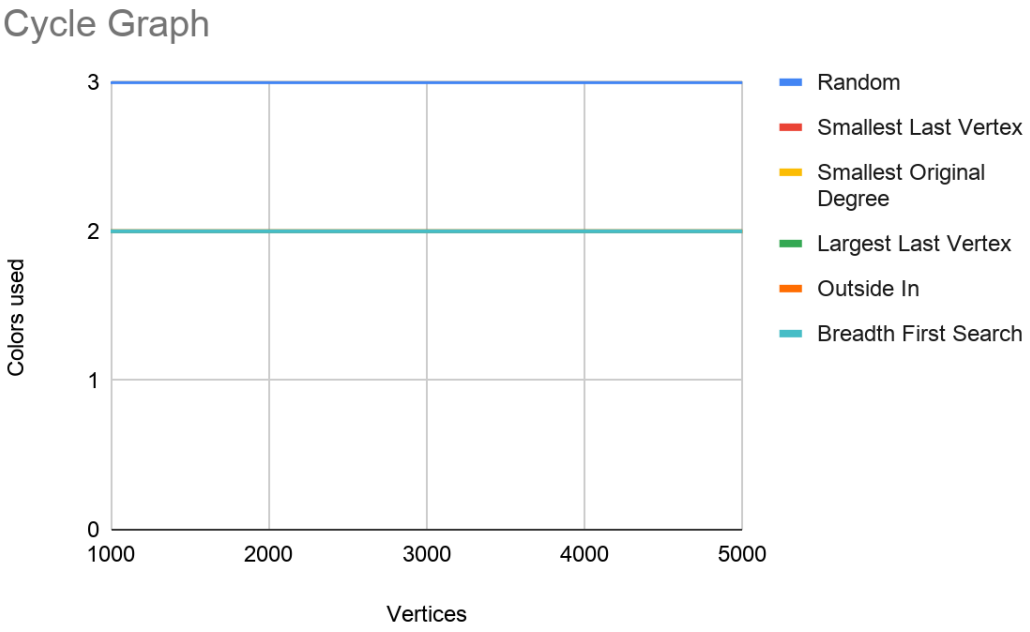
at 100,000. There is still a slight upward trend likely due to hardware constraints like different caches being hit for different sized graphs. This trend is negligible when compared to the first line.

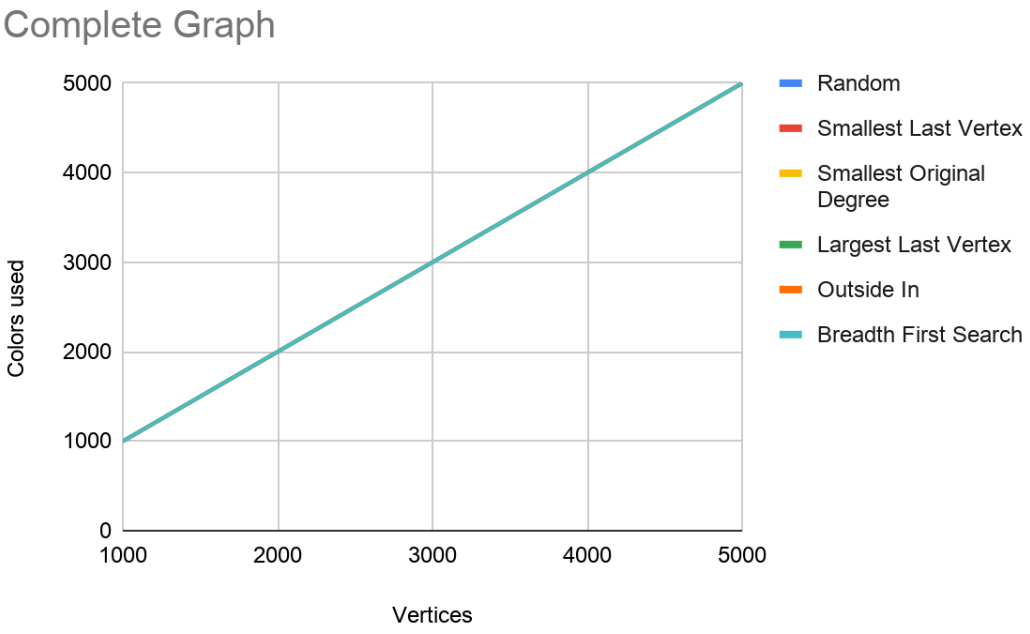


Vertex Ordering Capabilities

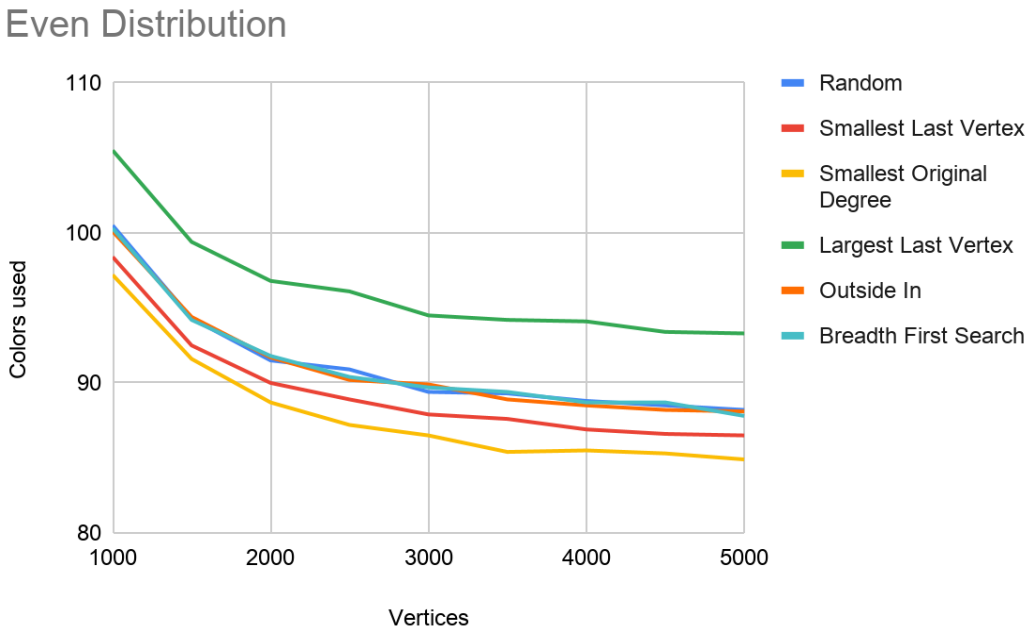
Each of the following graphs shows the average clored used by each ordering on each distribution over 10 trials. All tests were run on a graph with 200 times as many edges as vertices.

The first two graphs show complete graphs and cycle graphs. The color is equal to the number of vertices for all algorithms with a complete graph as expected. As all the values for vertices tested were even, all cycle graphs could be colored in two colors. Every ordering except random was always able to do this.

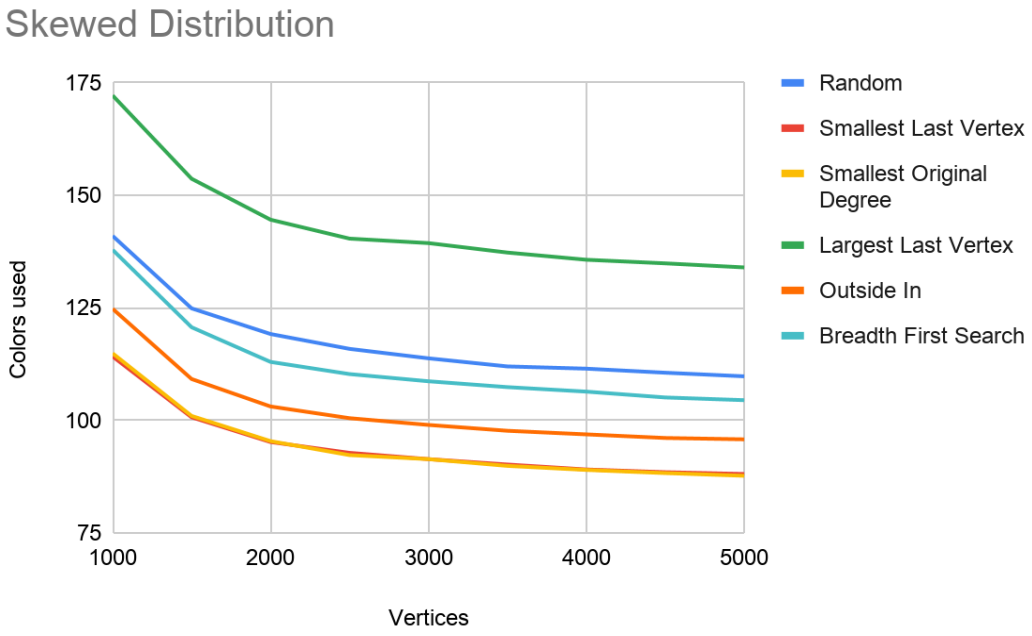

Cycle Graph
3
2
1
0
Colors used
1000
2000
3000
4000
5000
Vertices
Random
Smallest Last Vertex
Smallest Original Degree
Largest Last Vertex
Outside In
Breadth First Search

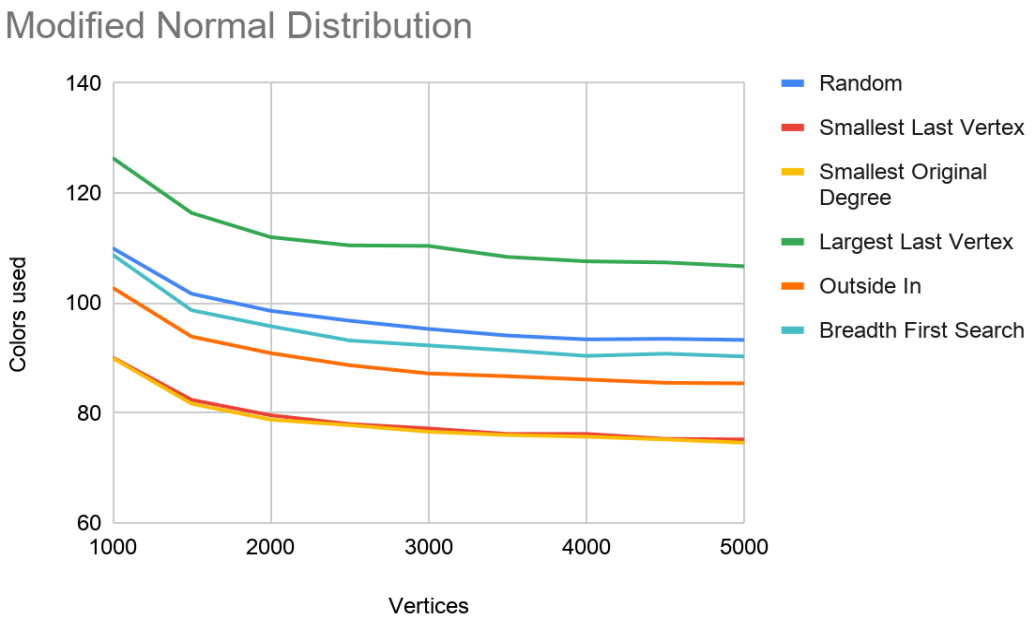

Complete Graph
5000
4000
3000
2000
1000
0
Colors used
1000
2000
3000
4000
5000
Vertices
Random
Smallest Last Vertex
Smallest Original Degree
Largest Last Vertex
Outside In
Breadth First Search

For the even distribution, the largest last vertex always did the worst and smallest original degree always did the best. Smallest last vertex did consistently slightly worse than smallest original degree. Random, outside in, breadth first search consistently tied.



The analysis for the skewed distribution and the modified normal distribution is the same except the number of colors used in the modified normal distribution is uniformly less. Largest last vertex did the worst followed by random, breadth first search, and outside in. Smallest last vertex and smallest original degree both provided the same results.





For all three distributions the colors used decreased at the same rate across orderings as the graphs got more sparse.

APPENDIX A: EXTERNAL LINKS

All spreadsheets have the accompanying tables that the graphs used were generated from.

1. Code:

   https://github.com/MarkBrubaker/CS5350-Project

   This can be opened as solution with one project inside using visual studio 2019

2. Distribution times:

   https://docs.google.com/spreadsheets/d/14aV3dgcIZ_Q3NGN8i2oE2r-Iq0kf8iSMCy5Ek9uIzPg/edit?usp=sharing

3. Distribution histograms:

   https://docs.google.com/spreadsheets/d/1HP_5EglW8IyYewl7fTNAr22Ie0VnltZ7_1urST_Z5Fk/edit?usp=sharing

4. Ordering data:

   https://docs.google.com/spreadsheets/d/12WJRUG7XcwSo2AvH9qEoVZ7IWjU2lqY5zokoa5aVObs/edit?usp=sharing

5. Coloring times:

   https://docs.google.com/spreadsheets/d/13idKHUETwWqbYHCcpLveN7SCdzW-ArHrE99TOBCSEpU/edit?usp=sharing

6. Coloring data:

   https://docs.google.com/spreadsheets/d/1MlloZNl1Nojkjm8yN_EpEY49RtcIpQR_VKIagBrKlFk/edit?usp=sharing