Branch and Bound Algorithms in Graph Coloring

CSC591: Experimental Algorithms

Anna Dubovitskaya

December 16th, 2015

1 Introduction

In this paper, multiple branch and bound algorithms for minimizing the k value in graph coloring will be discussed, as well as the experimental methods and justifications used in the creation of such algorithms.

1.1 Graph Coloring

Graph "coloring" is the act coloring each vertex of a network graph G consisting of edges E and vertices V such that no two vertices v_1 and v_2 sharing an edge in G are assigned the same color. It is a form of graph labeling. There are a few definitions that are frequently used in graph theory, and will be used throughout this paper. The terms are as follows:

- $k - coloring$: a graph that uses at most k colors is called a k-colored graph.
- $\chi(G)$: the chromatic number, or the smallest k-coloring a graph G can have and still be considered valid.

1.1.1 Graph Coloring Algorithms

The goal of many graph coloring algorithms is to minimize k. The trivial solution to a graph coloring algorithm is $k = |V|$, but this result can often be improved upon for any graph that is not a complete clique. For example, consider two different colorings for a graph, shown below:

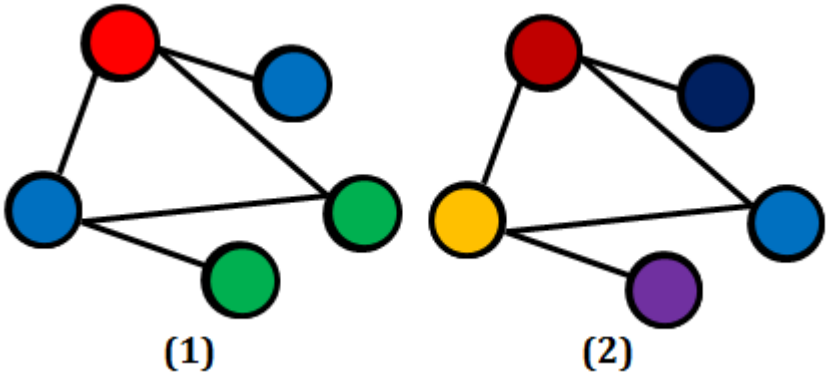

Figure 1: Two Valid Colorings of a Graph

The first graph G_1 is 3-colored, while the second graph G_2 is 5-colored. Both graphs are the same input graph G, and both colorings are valid, but G_1 uses less colors to achieve a full coloring. This k minimization problem can also be phrased as "finding the chromatic number of a graph", although the minimum k that can be found for a graph using experimental algorithms may not necessarily be equal to the chromatic number, and for some types of graphs a chromatic number has not yet been proven.

1.2 Motivation

The most commonly cited use for graph coloring is the coloring of maps, such as the map of the United States. In such a map, no adjacent countries can be colored the same, otherwise the map does not convey the intended information properly. However, it has already been proven that planar graphs - the type of graph a map becomes - have an upper bound of $k = 4$. As a result, recent research applications of graph coloring algorithms have concentrated on their impact on efficient scheduling and improving algorithms such as Google's PageRank.

Thus, the question of how to efficiently minimize k for non-planar graph becomes a question that can be addressed using experimental algorithms. It is a multifaceted problem, since even if we find the most efficient algorithm for one graph, it may not perform well on another graph. As a result, it becomes necessary to explore and develop different methods to evaluate and improve the effectiveness of the algorithm used. In this project, one algorithm was used and modified - the branch and bound algorithm.

1.3 Branch and Bound

A *branch and bound algorithm* is an algorithm that recursively generates problem instances ("branches"), some of which are eliminated based on some metric ("bounding") in order to

minimize or maximize the value of interest. With graph coloring, a branch and bound algorithm can be used in order to minimize the number of colors. For this purpose, the following metrics and definitions are used:

- *node*: either the initial node, or a recursively generated child node. The node consists of a graph to be colored, a lower bound, and an upper bound. *Note*: in this paper, the term *node* is used to specify a node in the branch and bound algorithm, and the term *vertex* is used to describe a vertex of the given graph.
- *lower bound*: this is the smallest k value a given graph G can have. The trivial case is $k = 1$, as any graph with $V > 0$ must be colored using at least one color.
- *upper bound*: this is the largest k value the given graph G can have. The trivial case is $k = V$, as every vertex in a graph can be colored a different color.
- *global upper bound*: this is the upper bound that is representative of the current, best minimized coloring at a given stage of the branch and bound algorithm.

An algorithm is created using the above definitions as follows:

1. Initialize an initial node containing the graph G for which we are trying to determine the minimized k-coloring.
2. Determine the upper and lower bound of the node using the process described in the Composition section.
3. If the upper bound is smaller than the global upper, set the global upper bound to be equal to the smaller upper bound.
4. If the lower bound is higher than the upper bounds, the node is killed.
5. Recursively generate child nodes by choosing two vertices in graph G that are not connected by an edge, and:
   (a) draw an edge between them in a new graph G', forcing the two vertices to be different colors.
   (b) collapse them in a new graph G', forcing the two vertices to be the same color.
6. Place the generated nodes onto the priority queue.
7. Pop the first node off of the queue and evaluate using steps 2-7.

Ideally, the lower and global upper bounds will converge to result in a minimized k. The branch and bound algorithm will explore and generate nodes recursively, updating the global upper bound.

2 Experimental Question

Using the information outlined above, an experimental question was formed: *What adjustments or decisions can we make to the branch and bound algorithm in order to improve its performance and efficiency in minimizing the k value for some given graph G?*

3 Algorithms and Implementations

In this section, the different branch and bound algorithms used in evaluating the experimental question will be discussed, as well as any justifications for the decisions made.

3.1 Technical Details

The branch and bound algorithms used were programmed in Python, primarily using the *networkx* module for graph/networking functions. This module is a pure Python implementation, meaning the runtimes for any functions called are higher than those in modules implemented in C/C++, such as graph-tool. It has been stated that networkx has "running times in the order of 20 to 170 times slower than graph-tool". In order to combat this decreased runtime, pypy was used to run the evaluation programs rather than Python. The time was reported using the command line *time* argument. The graphs, as well as their predetermined ideal k values if any were available, for testing the algorithm were obtained from two sources. For the first part of the experimental algorithm evaluation, the graphs were selected from a list of DIMACS Graph Coloring Instances, which can be found in Section 6 of this paper.

For the second part, graphs were generated with a provided algorithm. Two methods of graph generation were used: geometric and random. Random graphs were generated by drawing edges between two randomly selected vertices, until the amount of edges desired was reached. Geometric graphs by nature require a bit more work to generate. First, vertices were placed as points on a plane. This allowed calculation of distance values between each vertex. An edge was draw between two vertices only if their distance was within a particular value. In the graph generation algorithm used, the distance between two vertices (x_1, y_1) and (x_2, y_2) was calculated as a max of $(|x_1 - x_2|, |y_1 - y_2|)$, with wrap-around distances.

Additionally, another type of graph generation was explored using networkx's Barabasi-Albert model graph generator. This algorithm generates scale-free random graphs according to the Barabasi-Albert model, which works by preferential attachment. The resulting degree distribution follows the scale-free graph power law, meaning that the graph type generated is quite different from geometric and random graphs.

The graphs were originally meant to span from 25 to 400 vertices, but due to the limitations of Python and the efficiency of the algorithm, the largest size the program was able to solve mutually was $V = 50$. This is a very understandable issue, as the branch and bound algorithm used in this project can be very loosely represented as a binary tree. It is known that the maximum number of nodes a binary tree of depth k can have is equal to $2^{k+1} - 1$. This means a graph of size $V = 50$ has a maximum of $3.4 * 10^{184}$ possible nodes. This upper limit is actually far too high for an edge/collapse branch and bound algorithm: while it is true that following nodes that only draw an edge will result in a depth of $k = V$, each collapse of vertices decreases the number of potential edges by more than one, thus the maximum branch and bound tree is actually smaller than the maximum binary tree of the same depth k. Nonetheless, the large growth in runtime from adding a single extra vertex is the same, and large graph sizes were avoided.

3.2 Overview

3.2.1 Composition

The algorithm chosen to complete this task is a simple branch and bound algorithm consisting of the following:

- priority queue
- global upper bound, initialized to the size of the initial graph G
- upper bound, size of a greedy coloring of the given graph G'
- lower bound, size of a maximal clique of the given graph G'

3.2.2 Performance Measurements

The bounds and parameters were adjusted to try to improve the run time of the branch and bound algorithm as part of the experiments. These adjustments are discussed in detail in the next section. In order to limit the runtime in case of large node count, a *timeout* function was implemented, where the program ends when 100,000 nodes have been parsed and outputs the last global upper bound as the k value found. If a DIMACS graph resulted in timeouts for multiple algorithms, the graph was not included in the results. For graphs that did work, in order to measure the performance of the branch and bound algorithms, the following experimental measures were devised:

- k value found: the smallest k value returned, or the global upper bound if the algorithms timed out.
- Runtime of the algorithm.

- Nodes parsed before k value found.

3.3 Priority Queue Decisions

The way the priority queue is organized for a branch and bound algorithm could have huge implications for its runtime. If we place the nodes containing ideal solutions further down in our queue, then it will take significantly longer for us to pursue bad branches, create nodes from bad branches, discover that they are "bad", and then throw them away before moving on to a potentially ideal branch. However, the question that is raised as a result of this analysis is "how can we tell which branch is bad, without evaluating it?".

For example, consider this scenario: we want to keep the lower bound as small as possible in order to minimize the k value. Therefore, it would make sense to organize the priority queue based on the lowest lower bound found. That way, we would be exploring the smallest options first, giving us a chance to find the true minimum k value for a graph. However, this approach would raise issues if the ideal k value was significantly higher than the size of the maximal clique in G, as can happen in certain types of graphs. In this scenario, pursuing the lowest lower bound values would result in very slow growth, greatly increasing the runtimes and perhaps even resulting in a timeout. For this stage of experimentation, the priority queue organization was limited to two methods. The first was a placement of *last created first*, and the second was *lowest lower bound first*.

3.4 Upper Bound Determinations

We have already stated that one of the best methods to find an upper bound is to greedily color the graph. However, greedy coloring can be done in multiple ways, and determining the "best" strategy is another experimental problem in and of itself. To concentrate on obtaining results specific to branch and bound, the default networkx greedy coloring strategies were used in this stage of experimentation. There are eight of these default strategies, but we used the *largest first* and *independent set* greedy coloring strategies.

3.5 Edge Decisions

Another place for experimental algorithm improvement is in the choice of vertices to collapse or draw an edge between during the creation of graph G'. If the vertices with the largest degrees are used, it is likely that a solution can be reached faster than if the first two unconnected vertices processed are used. However, implementing a separate priority queue and parsing through all of edges can be very costly. In this stage, we explored a largest-degree priority implementation by

using the vertices with the largest amount of degrees between them. Uniqueness of edges was not considered, instead combinations

3.6 Graph Types

One of the key elements of graph coloring is, of course, the graph used. Network graphs come in all types of sizes, edge-vertex ratios, and distributions, which can change the behavior of algorithms. As a result, after the first initial algorithm performance analysis, one algorithm was chosen to explore the effects of random graph generation on algorithm performance. In particular, regular *random* graph, random *geometric* graph, and *scale-free* graph generation was explored. In order to accomplish this, several graphs of each type and of differing vertex size were generated and analyzed using the aforementioned algorithm.

It was necessary to keep the ratio of vertices to edges somewhat similar between sizes and types of graphs, as this would allow for a more stable analysis.

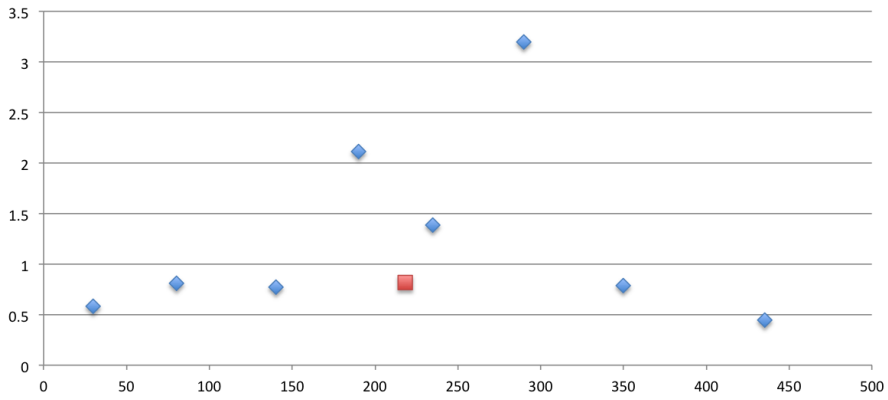

Figure 2: Calculating Runtimes of Various Edge Sizes for a $V = 30$ Graph

The blue dots symbolize random edge amounts for a $V = 30$ size graph, from $E = V$ to $E = \binom{V}{2}$, the maximum allowable amount of edges for a network graph, all plotted with their associated runtime for a minimization coloring algorithm. The result is a bell-shape distribution, but edge sizes in the 200s perform better than most surroundings. It turns out that a good ratio to use for the number of edges is $\binom{V}{2}/2$, or half of the maximum allowable number of edges for a graph. The red dot in Figure 5 demonstrates the coloring runtime result of a graph following this sizing convention. This method was used to generate several geometric, scale-free, and random graphs, of the following dimensions:

$G(V, E) = G(10, 23), G(20, 95), G(25, 150), G(30, 218), G(35, 298), G(40, 390), G(45, 495),$
$G(50, 612), G(55, 743), G(60, 885), G(65, 1040)$

Each graph type was generated using the above dimensions, and the solutions were analyzed by comparison.

3.7 Final Implementations

3.7.1 First Experiment

To summarize, using the parameter options 3.3-3.5, eight different branch and bound algorithms were created:

Branch and Bound Algorithms			
Algorithm	Priority Queue Implementation	Coloring Strategy	Edge Picking
BBC0	last created first	largest first	first encountered
BBC1	last created first	largest first	greedy
BBC2	last created first	independent set	first encountered
BBC3	last created first	independent set	greedy
BBC4	lowest lower bound first	largest first	first encountered
BBC5	lowest lower bound first	largest first	greedy
BBC6	lowest lower bound first	independent set	first encountered
BBC7	lowest lower bound first	independent set	greedy

3.7.2 Second Experiment

BBC5 was used as the algorithm of choice for running the random vs geometric vs scale-free graph comparison analysis outlined in 3.6, as it consistently performed well in the first half of the experimentation. One change was made with respect to the ordering of the priority queue - collapsing two vertices was placed at a higher priority than drawing an edge between them.

4 Results

4.1 Benchmark on Node Parsing

Each of the eight algorithms were run on a graph containing 450 vertices. This is one of the largest graph instances described in DIMACS, and is a complex graph. None of the algorithms were able to achieve a solution for this graph, so it was used to benchmark how fast each algorithm is able to parse nodes. A function to quit the program once 100 nodes were parsed was created, with the following results:

Figure 3: Benchmark Runtimes for 100 Nodes

4.2 Ideal k vs. Found k

All of the algorithms were able to match the ideal k value. In further experimentation, it would be interesting to run the algorithms on graphs that do not have an established k value.

4.3 Size of Graph vs Time to Solution

Although the size of the graph is relatively unimportant compared to the type of graph, in as much a clique of size 400 will generally result in a faster solution than a sparse and chaotic graph of size 300, it was interesting to see the performance relative to each branch and bound algorithm:

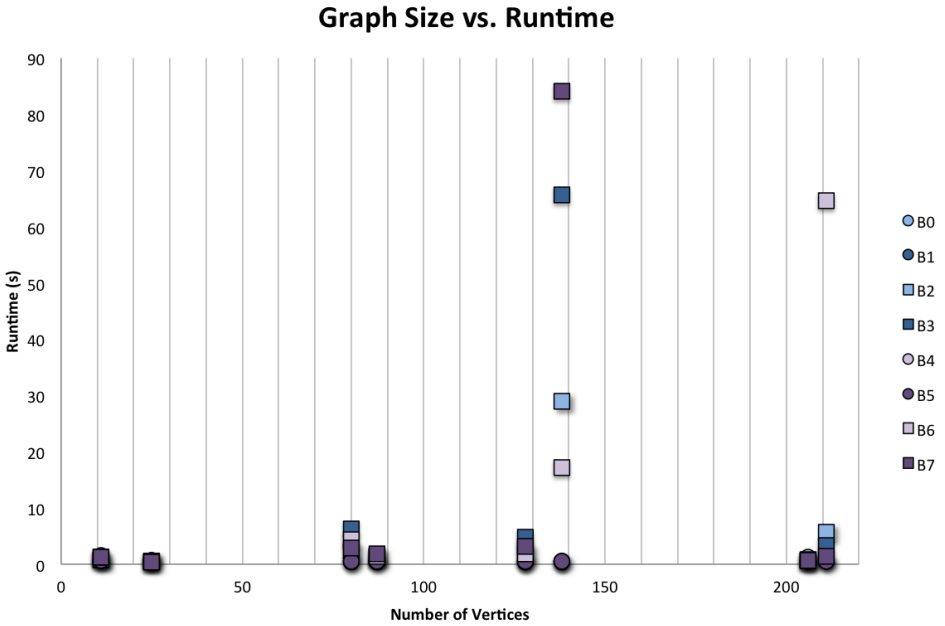

Figure 4:

In the above graph, there are a few interesting points to note. The first is that the square

data points (B2, B3, B6, B7) tend to have the highest runtimes. These are the algorithms that use the independent set strategy for the greedy coloring for the upper bound, which leads to the implication that greedy coloring using the independent set takes significantly more time. Therefore, if the primary concern for the algorithm used is runtime, this method of greedy coloring may not be the best option. B3 and B7 also had the highest benchmark runtimes as seen in Figure 2, further narrowing down the slowest algorithms as independent set greedy coloring and greedy edge picking. It is possible that the graphs are too small for improvements such as greedy edge picking to be efficient. This effect is best noticed in the results of the graph of size $V = 138$ and the graph of size $V = 211$ for the independent set greedy coloring algorithms. For $V = 138$, the runtimes in descending order are [B7, B3, B2, B6], whereas the runtimes for $V = 211$ are completely inverted with a descending order of [B6, B2, B3, B7]. This can be interpreted as the limit after which using greedy edge picking and independent set coloring strategy becomes worth the extra processing time it takes to set up the additional priority queues necessary. It is also possible that the opposite approach - picking edges with the smallest combined degree total - would result in better performance in smaller graphs, since the total number of edges in a graph grows as V^2. These observations can be influenced by graph types and degree distributions as well.

In addition, between the largest-first greedy coloring algorithms, the variance was quite small. For example, again note the results for the graph of size $V = 138$. The algorithms denoted with squares have a maximum difference of 66.881 seconds, whereas the maximum difference between the algorithms denoted with circles (largest first greedy coloring) is only 0.08 seconds. We can also observe ranking of the circle algorithms. Algorithm B0, the basic algorithm with no priority queue and first-occurrence edge picking, continuously performs the worst out of all of the largest-first greedy coloring algorithms, and the minor improvement B1 with greedy edge picking tends to have the lowest runtime. In fact, this is true for both greedy coloring strategies in the later graphs as well, which leads to the observation that the most significant improvement is with edge picking rather than with the organization of the priority queue or the method of coloring. Additionally, we can see that the circle algorithms tend to be fairly constant in runtime across graph sizes, unlike the square algorithms which tend to take more time as the amount of nodes increases. Interestingly, the independent set algorithms tend to either perform slightly better than at least the basic B0 algorithm, or perform significantly worse than any largest-first algorithm.

4.4 Nodes Considered Before Solution

In this graph, the amount of nodes that the algorithm parsed off of the priority queue before reaching the ideal solution is plotted:

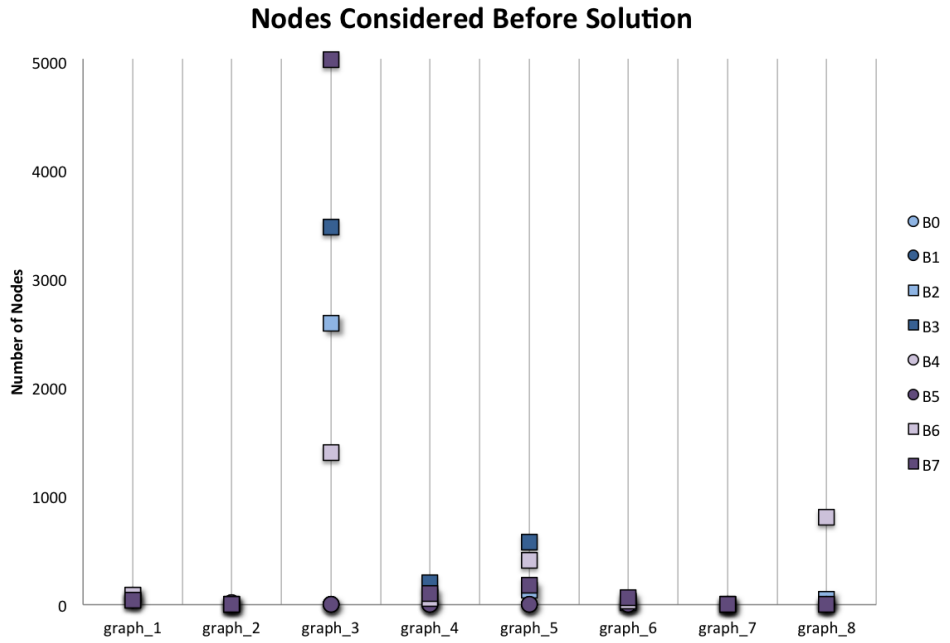

Figure 5: Graph Size vs Runtime for All Algorithm Types

The images are a bit misleading, as they do not include the size of the graph, which would have been a better metric than the name of the graph used. Regardless, graph_3 ($V = 138$), graph_4 ($V = 128$), and graph_8 ($V = 211$) are some of the largest graph sizes. Interestingly, graph_5 ($V = 80$) resulted in some large node parsing despite being on the smaller half of all the graph sizes, and graph_7 ($V = 206$) had a uniform count of 1 nodes parsed for all algorithms, despite being the largest size graph. It is possible that there is something about the graph_3 file that makes it difficult to find and utilize an independent set strategy for greedy coloring, resulting in the large amounts of nodes parsed and runtime. Additionally, the algorithms utilizing greedy edge picking tended to have higher amounts of nodes parsed compared to the first-occurrence edge picking algorithms.

4.5 Random, Geometric, and Scale-Free Graph Generation

In this section, the random, geometric, and scale-free generated graphs were colored using the BBC5 algorithm as outlined previously. In the interest of understandable data, some data points were left off of the result charts in the following sections:

- Data was obtained for $V = 45, 50$ for the random graph, but due to a large increase in runtime (41 minutes to run $V = 45$) these two data points were left off.
- Data was obtained for $V = 60$ for the geometric graph, but due to a large runtime to solution (72 minutes) this data point was left off.

These points are discussed in the analysis, but are not included in the charts due to skewing of the visual summary. For example, if the 72 minute runtime was included, the random graph runtimes would look linear in nature, as well as performing better than the geometric graphs. However, this is not the case. The full results for this section can be found on the associated GitHub page in the "BBC_random_geometric" Excel sheet.

4.5.1 Size of Graph vs Time to Solution

For these graphs, the geometric, random, and scale-free graph runtimes were compared across graph size with respect to vertices. The results are shown below:

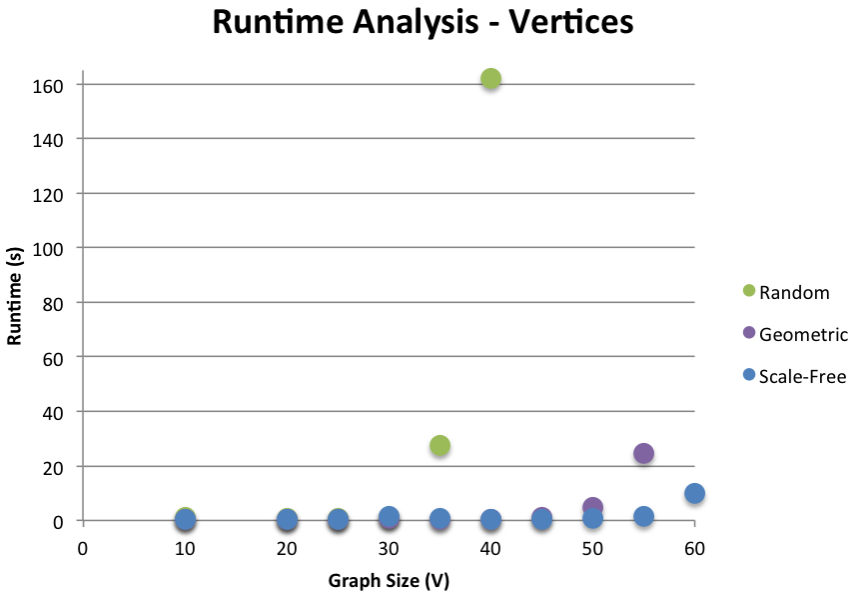

Figure 6: Runtime Analysis for Geometric and Random Graph Generation

Solving each type of graph resulted in quite similar performance shapes. In the beginning, the performance is almost indistinguishable between the three graph types, but as the graphs increase in size, each of the graph types starts to develop a threshold for drastically increased runtime. The scale-free graph type performed the best for the largest amount of vertices, and was fairly consistent throughout the graph types. The random graph fails earliest (on $G(35, 298)$).

Since a few data points have been ommitted in the interest of preserving the trendlines, there was one observation that is not fully evident: the geometric graph fails at a much quicker rate than both the random and scale-free graph types. An example can be found by comparing the rate of increase in runtime between geometric and random graphs: at $V = 35$, the random graph coloring takes about 27 seconds to find the ideal solution. The next graph, $V = 40$ takes only about 134 seconds more. However, when the geometric graph coloring takes about the same

amount of time, 24 seconds, at $V = 55$ to find the ideal solution, the next graph takes 72 minutes, and it isn't even able to reach an ideal solution in that time. This is quite different from the very gradual increase in runtime of the scale-free graphs.

4.5.2 Size of Graph vs Nodes Parsed Before Solution

The following results show how many nodes were parsed in each graph type before an ideal solution was reached:

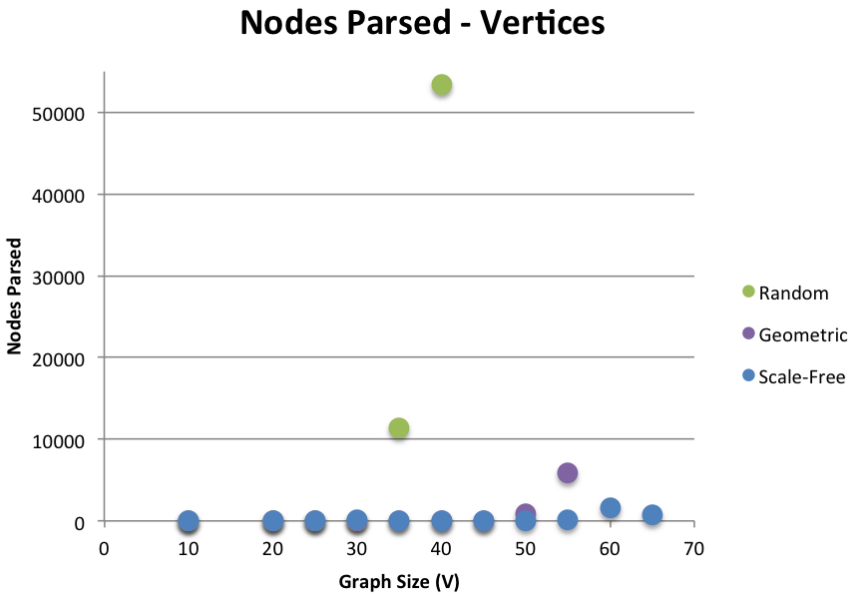

Figure 7: Nodes Parsed for Geometric and Random Graph Generation

These follow the same pattern as runtime analysis figure, and therefore support the same conclusions. However, it is not possible to make the same conclusions about the performance change of the geometric graphs at this point, as no data for total nodes parsed has been achieved without a segmentation fault. Thus, it is possible that the amount of nodes parsed could stay steady, unlike the runtime.

4.5.3 Nodes vs Runtime

Solving the scale-free graph takes about 0.0056 seconds per node, the geometric graph takes about 0.0040 seconds per node, and solving the random graph takes a little less at around 0.0023 seconds per node. This result is interesting, since solving the scale-free and geometric graphs showed better overall performance in the previous analysis, yet these have the worst runtime cost.

4.5.4 Graph Type Analysis

Many of the results obtained are natural reflections of the behavior of the graph type generation. For example, scale-free graphs rely on power law distribution to place edges. This results in high amounts of low-degree vertices, as well as clustering. In turn, such degree distributions limit the maximal clique values, but the greedy coloring is also lower than in highly connected graphs. Even as the graph size grows, the tendency of scale-free graphs is to add "hubs" (one large degree vertex connected to many one-degree vertices) rather than to increase degree size, resulting in slow-growing runtimes. Furthermore, collapsing or drawing an edge between hubs of a scale-free graph increases the amount of colors needed (a perfect hub only needs two colors), in turn increasing the upper bound and rendering the branch "dead".

On the other hand, random and geometric graphs have more bell-shaped distributions. Random graphs in particular have small amounts of extreme (high or low) degree vertices, whereas geometric graphs have a slightly skewed poisson-type degree distribution. As a result, one is much more likely to find larger maximal cliques in geometric or random graphs. One suggestion for future analysis would be to find the difference between the upper and lower bounds for each graph type, as well as the growth of these bounds. This would help to reveal the reasons behind some of the results discussed in this paper.

5 Future Work

- Determine which coloring algorithms work best on different graph types
- Evaluate upper-lower bound differences between graph types
- Explore different methods of choosing vertices other than greedy algorithms
- Check runtime performance against better-known coloring algorithms

6 References

[1] Stanford Slides on Branch and Bound: `http://stanford.edu/class/ee364b/lectures/bb_slides.pdf`

[2] Leighton Graphs and Difficult Coloring Instances: `https://archive.org/stream/jresv84n6p489/jresv84n6p489_A1b#page/n9/mode/2up`

[3] DIMACS Coloring Instances: `http://mat.gsia.cmu.edu/COLOR/instances.html`

[4] NetworkX Documentation for Greedy Coloring: https://networkx.github.io/documentation/development/reference/generated/networkx.algorithms.coloring.greedy_color.html

[5] Runtime Comparison of graph-tool and networkx: https://graph-tool.skewed.de/performance

[6] Early Estimates of the Size of Branch-and-Bound Trees: http://integer.tepper.cmu.edu/webpub/EarlyEstimates.pdf

[7] Percolation, Connectivity, Coverage and Colouring of Random Geometric Graphs https://www.stat.berkeley.edu/~aldous/206-SNET/Papers/BBS.pdf

[8] Graph Coloring Algorithms http://www.cs.cornell.edu/courses/cs3110/2011sp/recitations/rec21-graphs/graphs.htm

[9] All Referenced Code Can be Found on NCSU's GitHub: https://github.ncsu.edu/adubovi/Branch-and-Bound-Coloring