Analysis of Vertex Coloring Algorithms and Mappings to and From 3-SAT

Samuel Gaines
University of Alabama
Tuscaloosa, Alabama 35487
Email: smgaines2@crimson.ua.edu

***Abstract*—This project analyzed a brute-force and heuristic algorithm for the Vertex Coloring problem and observed if problems persisted after mappings to and from 3-SAT. A brute-force and heuristic algorithm for Vertex Coloring were written in C++ and measured for accuracy and efficiency. Additionally, mappings from Vertex Coloring to 3-SAT and from 3-SAT to Vertex Coloring were written in C++ and run on particularly interesting graphs and 3-SAT problems. Brute-force and heuristic algorithms were run on the results of these mappings and measured for accuracy and efficiency. Results show that while mapping one NP-Complete problem to another NP-Complete problem will likely increase the run time of the brute-force solution, it has the potential to increase the optimality of a heuristic solution.**

I. Introduction

Computing problems are often classified by complexity class. A decision problem is said to be in NP if given solutions can be verified in polynomial time. A decision problem is said to be in P (a subset of NP) if solutions can be found in polynomial time.

A decision problem is said to be NP-Complete if it is in NP and shown to be at least as hard as every other problem in NP. To prove that a problem is NP-Complete, it suffices to show that a reduction, or mapping, exists from another NP-Complete problem to the problem in question. Informally, a problem can be proven to be NP-Complete if it can be shown that it is at least as hard as another NP-Complete problem.

No polynomial time solutions to NP-Complete problems have been found; therefore, exact, or brute-force, solutions to NP-Complete problems can take an infeasible amount of time to run when input size surpasses a certain threshold. We say that these problems are ”intractable” for the algorithm. Because of this, in practice, polynomial time heuristic algorithms are used to find ”good”, but not necessarily perfect, solutions to the problem.

The Vertex Coloring problem is an example of an NP-Complete problem, as solutions can be verified in polynomial time, and mappings exist to Vertex Coloring from other NP-Complete problems, such as 3-SAT (a mapping also exists from 3-SAT to Vertex Coloring, discussed later). The DSatur algorithm is a common heuristic used to find solutions to the Vertex Coloring problem.

Despite the loads of research on Vertex Coloring and NP-Complete problems in general, little is known about the effects of mapping on the success of heuristic algorithms. In this project, the following questions are addressed:

RQ1: When does the brute-force algorithm for Vertex Coloring become intractable?

RQ2: When does the DSatur heuristic algorithm for Vertex Coloring fail to be optimal?

RQ3: Suppose an input to an NP-Complete problem causes a certain solution to fail. Is that input, after being reduced to an input for another NP-Complete problem, likely to fail on an algorithm for the second problem as well?

A. The Vertex Coloring Problem

The Vertex Coloring problem (k-coloring problem) poses the following question:

Given undirected graph $G = (V, E)$ with unweighted edges, does there exist an assignment of at most k colors to the vertices such that $color(u) \neq color(v)\ \forall\ (u, v) \in E$?

Informally, is it possible to color each vertex in a graph with some maximum number of colors such that no two vertices connected by an edge are the same color? If this is possible, we say there exists a k-coloring of the graph. Note that the Vertex Coloring problem is only NP-Complete when $k \geq 3$.

The Chromatic Number problem, a variation to the Vertex Coloring problem, asks the following question:

Given undirected graph $G = (V, E)$ with unweighted edges, what is the minimum number of colors able to be assigned to the vertices such that $color(u) \neq color(v)\ \forall\ (u, v) \in E$?

In other words, what is the smallest k such that the given graph has a k-coloring? We call this number the *chromatic number* of the graph. Note that the Chromatic Number problem is not NP-Complete, even though it closely related to the k-coloring problem. An example 3-coloring of a graph is shown in Figure 1.

B. The Boolean Satisfiability Problem

The Boolean Satisfiability problem (SAT) was the first problem proven to be NP-Complete by Stephen Cook and Leonid Levin. It poses the following question:

Given clausal formula ϕ, does there exist an assignment of values to the Boolean variables in ϕ to make ϕ true?

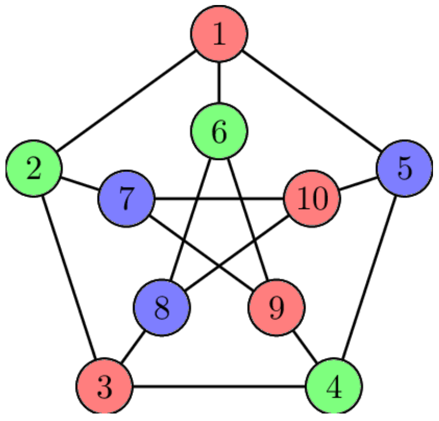

Fig. 1. A graph with chromatic number 3 [1]

If there does exist such an assignment, we say that ϕ is satisfiable. A special case of SAT is the 3-SAT problem, which restricts ϕ to have clauses with at most 3 variables. An example 3-SAT formula is shown below.

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4$$

The formula can be satisfied with $x_1 = true$, $x_2 = true$, $x_3 = false$, and $x_4 = true$.

II. Methods

C++ was used to implement four algorithms for this project: a brute-force algorithm to solve the Chromatic Number problem, the DSatur heuristic algorithm to solve the Chromatic Number problem, a mapping from Vertex Coloring to 3-SAT, and a mapping from 3-SAT to Vertex Coloring.

A. Backtracking Brute-force Algorithm for Vertex Coloring

The backtracking method is a well-known brute-force solution for the k-coloring problem.

Assume an ordering on the colors. Start by assigning the first color to the first vertex. If the first color is available for the next vertex, assign it and move to the next vertex. If not, then assign the next color. If it any point, no colors are available for a vertex, backtrack to the previous vertex and assign the next color to it. If all k colors are worked through for all vertices, and no solution is found, then there does not exist a k-coloring of the graph.

Run the algorithm for increasing values of k until a k-coloring is found at $k = k_c$. Then, you know that k_c is the chromatic number of the graph. The backtracking brute-force algorithm runs in $O(k_c^V)$ time.

B. DSatur Heuristic for Vertex Coloring

The Degree of Saturation (DSatur) algorithm is a common heuristic algorithm for the Chromatic Number problem.

Assume an ordering on the colors. Let the ”degree of saturation” of a vertex be the number of different colors being used by adjacent vertices. Choose the vertex with the highest degree of saturation. In the case of a tie, choose the one adjacent to the highest number of uncolored vertices. Assign the first available color to the chosen vertex. Repeat the process until all vertices are colored.

The number of colors used is a prediction of the chromatic number of the graph. The DSatur algorithm runs in $O(V^2)$ time. There is an alternate implementation of the DSatur algorithm that runs in $O((V+E)lgV)$ time. This is usually much faster, but can be slower in the densest of graphs (when E is close to V^2).

C. Mapping from Vertex Coloring to 3-SAT

Since the Vertex Coloring problem and 3-SAT are both NP-Complete, a Vertex Coloring problem can be reduced to a 3-SAT formula in polynomial time. First, reduce the Vertex Coloring problem to a SAT problem, then reduce the SAT problem to a 3-SAT problem.

1) Mapping from Vertex Coloring to SAT: Let $V = \{v_1, v_2, ..., v_V\}$ be the set of vertices and $C = \{c_1, c_2, ..., c_k\}$ be the set of k colors available. For $i \in \{1, ..., V\}$, $j \in \{1, ..., k\}$, let x_{ij} be a Boolean variable assigned to $true$ if $color(v_i) = c_j$ and assigned to $false$ otherwise.

In any of a graph, it is true that at most one color should be assigned to each vertex. We can say that for all vertices v_i, $x_{i1} \vee x_{i2} \vee ... \vee x_{ik} = true$. Thus, if there exists a k-coloring of the graph, then there exists an assignment of truth values to each x such that:

$$\bigwedge_{i=1}^{V} \bigvee_{j=1}^{k} x_{ij} = true$$

In any coloring of a graph, it is also true that at least one color must be assigned each vertex. We can say that for all vertices v_i, $color(v_i) = c \rightarrow color(v_i) \neq d \; \forall$ distinct $c, d \in C$. Equivalently, we can say that for all vertices v_i, $\neg x_{ic} \vee \neg x_{id}$ must be true $\forall$ distinct $c, d \in C$. Thus, if there exists a k-coloring of the graph, then there exists an assignment of truth values to each x such that:

$$\bigwedge_{i=1}^{V} \bigwedge_{c=1}^{k-1} \bigwedge_{d=c+1}^{k} (\neg x_{ic} \vee \neg x_{id}) = true$$

In a k-coloring of a graph, it is true that no vertices connected by an edge are the same color. We can say that for all colors c_j, $x_{aj} \rightarrow \neg x_{bj} \; \forall \; (v_a, v_b) \in E$. Equivalently, we can say that for all colors c_j, $\neg x_a j \vee \neg x_b j \; \forall \; (v_a, v_b) \in E$. Thus, if there exists a k-coloring of the graph, then there exists an assignment of truth values to each x such that:

$$\bigwedge_{(v_a,v_b) \in E} \bigwedge_{j=1}^{k} (\neg x_a j \vee \neg x_b j) = true$$

The above three formulas are sufficient to show that a graph has a k-coloring. Thus, a graph has a k-coloring if and only if the following formula is satisfiable:

$$(\bigwedge_{i=1}^{V} \bigvee_{j=1}^{k} x_{ij}) \wedge (\bigwedge_{i=1}^{V} \bigwedge_{c=1}^{k-1} \bigwedge_{d=c+1}^{k} (\neg x_{ic} \vee \neg x_{id})) \wedge (\bigwedge_{(v_a,v_b) \in E} \bigwedge_{j=1}^{k} (\neg x_a j \vee \neg x_b j))$$

The described mapping from Vertex Coloring to SAT takes $O(V^2 k)$ time to run. Note that $V_2 k < V^3$, as a graph is trivially k-colorable for $k \geq V$.

For example, consider the graph G in Figure 2. G is 3-colorable if and only if the following can be satisfied:

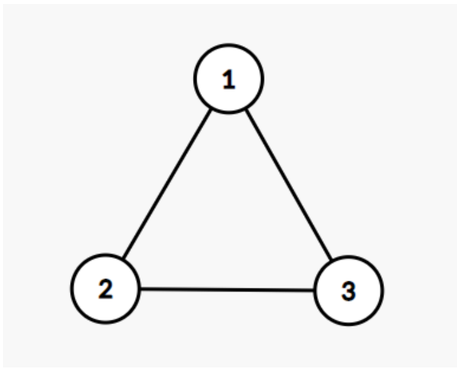

Fig. 2. Example graph for reduction to SAT

$$(x_{11} \vee x_{12} \vee x_{13}) \wedge (x_{21} \vee x_{22} \vee x_{33}) \wedge (x_{31} \vee x_{32} \vee x_{33}) \wedge (\neg x_{11} \vee \neg x_{12}) \wedge (\neg x_{11} \vee \neg x_{13}) \wedge (\neg x_{12} \vee \neg x_{13}) \wedge (\neg x_{21} \vee \neg x_{22}) \wedge (\neg x_{21} \vee \neg x_{23}) \wedge (\neg x_{22} \vee \neg x_{23}) \wedge (\neg x_{31} \vee \neg x_{32}) \wedge (\neg x_{31} \vee \neg x_{33}) \wedge (\neg x_{32} \vee \neg x_{33}) \wedge (\neg x_{11} \vee \neg x_{21}) \wedge (\neg x_{12} \vee \neg x_{22}) \wedge (\neg x_{13} \vee \neg x_{23}) \wedge (\neg x_{11} \vee \neg x_{31}) \wedge (\neg x_{12} \vee \neg x_{32}) \wedge (\neg x_{13} \vee \neg x_{33}) \wedge (\neg x_{21} \vee \neg x_{31}) \wedge (\neg x_{22} \vee \neg x_{32}) \wedge (\neg x_{23} \vee \neg x_{33})$$

Clearly, these inputs can expand quite fast.

2) Mapping from SAT to 3-SAT: Separate the clauses into cases and replace them accordingly.

Suppose a clause contains exactly three Boolean literals. Then it is already in 3-SAT form and does not need to undergo a transformation.

Suppose a clause contains exactly two Boolean literals x_1 and x_2, i.e. the clause is $x_1 \vee x_2$. Let z_1 be a new Boolean variable. Then it is equivalent to say:

$$(x_1 \vee x_2 \vee z_1) \wedge (x_1 \vee x_2 \vee \neg z_1)$$

Suppose a clause contains exactly one Boolean literal x_1. Let z_1 and z_2 be new Boolean variables. Then it is equivalent to say:

$$(x_1 \vee z_1 \vee z_2) \wedge (x_1 \vee z_1 \vee \neg z_2) \wedge (x_1 \vee \neg z_1 \vee z_2) \wedge (x_1 \vee \neg z_1 \vee \neg z_2)$$

Suppose a clause contains $n > 3$ Boolean literals $x_1, ..., x_n$, i.e. the clause is $\bigvee_{i=1}^{n} x_i$. Let $z_1, ..., z_{n-3}$ be new Boolean variables. Then the following is satisfiable if and only if the original clause is satisfiable:

$$(x_1 \vee x_2 \vee z_1) \wedge (x_3 \vee \neg z_1 \vee z_2) \wedge (x_4 \vee \neg z_2 \vee z_3) \wedge ... \wedge (x_{n-2} \vee \neg z_{n-4} \vee z_{n-3}) \wedge (x_{n-1} \vee x_n \vee \neg z_{n-3})$$

After all the replacements are made case-by-case, the resulting formula will be an equivalent 3-SAT problem. The described mapping takes $O(cn)$ time to run, where c is the number of clauses and n is the number of Boolean variables per clause.

For example consider the following SAT formula:

$$x_1 \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2 \vee x_3) \vee (x_1 \vee x_2 \vee \neg x_3 \vee \neg x_4 \vee x_5)$$

When reduced to 3-SAT, this formula becomes:

$$(x_1 \vee z_1 \vee z_2) \wedge (x_1 \vee \neg z_1 \vee z_2) \wedge (x_1 \vee z_1 \vee \neg z_2) \wedge (x_1 \vee \neg z_1 \vee \neg z_2) \wedge (\neg x_1 \vee \neg x_2 \vee z_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg z_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee z_4) \wedge (\neg x_3 \vee \neg z_4 \vee z_5) \wedge (\neg x_4 \vee x_5 \vee \neg z_5)$$

D. Mapping from 3-SAT to Vertex Coloring

Since the Vertex Coloring problem and 3-SAT are both NP-Complete, a 3-SAT problem can be reduced to a Vertex Coloring problem in polynomial time. First, reduce the 3-SAT problem to a 3-coloring problem, then reduce the 3-coloring problem to a k-coloring problem.

1) Mapping from 3-SAT to 3-coloring: For every clause in the 3-SAT formula, construct a triangle. Each vertex in the clause corresponds to a different Boolean literal from the clause.

For every Boolean variable in the 3-SAT formula, construct a triangle. Each of these triangles will share one common vertex. Of the other two vertices in each of these triangles, one will correspond to the variable itself, and the other will correspond to its negation. In a proper vertex coloring, these two vertices cannot be the same color; we can use one color to represent true and the other to represent false.

Finally, connect each vertex from the clause triangles to its corresponding vertex in the Boolean variable triangles. Here, a clause triangle is 3-colorable if and only if all three of its corresponding Boolean variable vertices are not the same truth value (color).

The above steps produce a graph that is 3-colorable if and only if the 3-SAT formula is satisfiable. The described mapping takes $O(c+n)$ time to run, where c is the number of clauses and n is the total number of Boolean variables. Assuming that all variables are used at least once in the 3-SAT formula, it is true that $n \leq 3c$, so the mapping takes $O(c)$ time to run.

For example, consider the following simple 3-SAT formula:

$$(\neg x_1 \vee x_2 \vee x_3)$$

This formula is satisfiable if and only if the graph in Figure 3 is 3-colorable.

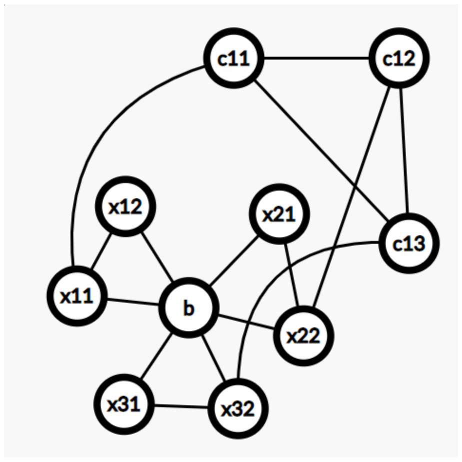

Fig. 3. Example graph for reduction from 3-SAT

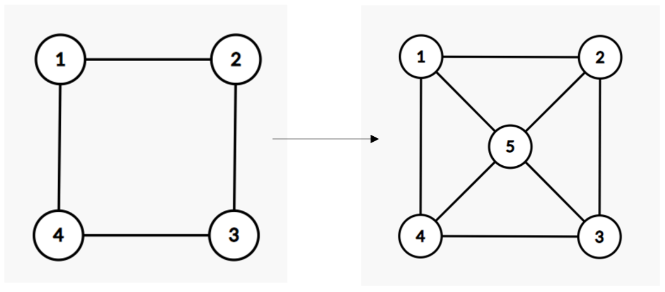

Fig. 4. Example reduction from a k-coloring problem (left) to an equivalent k+1-coloring problem (right)

2) Mapping from 3-coloring to k-coloring: Given a graph with chromatic number k, it is actually quite easy to transform the graph into one with chromatic number $k+1$. Construct one new vertex. Then, construct edges conecting this new vertex with every other vertex. This will force the new vertex to take a new color, and therefore increase the chromatic number of the graph from k to $k+1$.

Repeat this process $k-3$ times to map a 3-coloring problem to a k-coloring problem. The mapping takes $O(Vk)$ time to run.

An example mapping from a k-coloring problem to a k+1-coloring problem can be found in Figure 4

III. Findings

After the backtracking algorithm, the DSatur heuristic, and reductions to and from 3-SAT were implemented, graphs and 3-SAT formulas from across the input space of both problems were tested, and results were analyzed.

A. Notable Inputs

Some graphs exhibited especially interesting behavior and provide insight on *RQ1* and *RQ2*.

1) Intractable Problems for the Backtracking Algorithm: A C++ script was written to generate a ”random” graph with v vertices and no more than $3v$ edges. It was found that the backtracking algorithm becomes infeasible for random graphs of this nature that have $v \gtrapprox 70$. One of these random graphs in particular with $v = 75$ was chosen for analysis. The backtracking algorithm was unable to complete, and the DSatur heuristic estimated that 4 was the chromatic number of the graph.

Another C++ script was written to generate a ”fully connected” graph with v vertices. This graph would have $\frac{1}{2}v(v-1)$ edges and chromatic number equal to v. An examply fully connected graph can be seen in figure 5 The backtracking algorithm could handle a much higher v in a fully connected graph than in a random graph. This is because the algorithm eliminates the possibility of a k-coloring for $k < v$ early when the graph is fully connected. Precise definitions for ”intractable” can vary, but one could reasonably say that the fully connected graph is intractable for the backtracking algorithm when $v \gtrapprox 500$. The fully connected graph with $v = 1000$ was chosen for analysis. The backtracking algorithm was unable to complete, and the DSatur heuristic estimated (and was correct) that 1000 was the chromatic number of the graph.

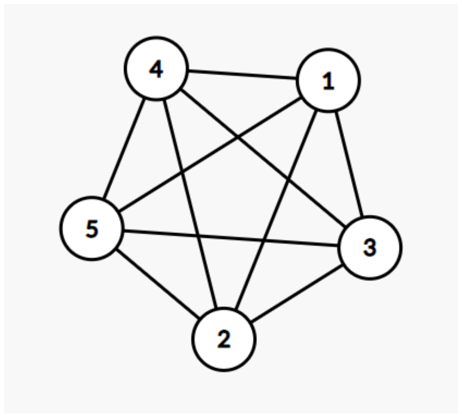

Fig. 5. Fully-connected graph with $v = 5$

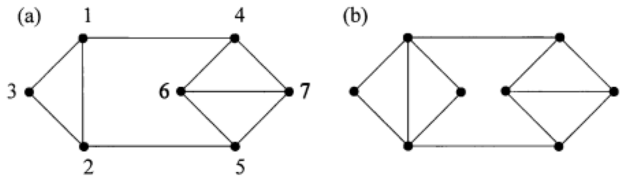

Fig. 6. (a) the smallest SHC problem for DSatur; (b) the smallest HC problem for Dsatur [2]

2) Slightly Hard-to-Color and Hard-to-Color Graphs for the DSatur Heuristic: Define $C(G)$ as the chromatic number of a graph G. We say a graph G is slightly hard-to-color (SHC) for a Chromatic Number heuristic A if $A(G) > C(G)$ for some ordering of vertices in G. Similarly, we say a graph G is hard-to-color (HC) for a Chromatic Number heuristic A if $A(G) > C(G)$ for all orderings of vertices in G [2].

Janczewski et al. find the smallest SHC and HC graph for the DSatur algorithm. These can be found in Figure 6.

The implemented DSatur algorithm was run on the two graphs. As expected, the DSatur algorithm found that the chromatic number of both graphs was 4, but the backtracking algorithm was able to find 3-colorings.

3) Notable 3-SAT Formulas: Classmate EJ Yohannan similarly explored the input space for the 3-SAT problem and found problems that were either intractable for his brute-force algorithm or hard for his heuristic.

B. Result of Running Algorithms on Reduced Notable Inputs

The previously specified notable inputs were reduced either from Vertex Coloring to 3-SAT or vice versa, and algorithms were then run on the reduced problems. 3-SAT algorithms were run by EJ Yohannan. The results provide insight on *RQ3*.

1) Intractable Vertex Coloring Graph: The intractable random graph was reduced to 3-SAT successfully. This reduction was done for both $k = 3$ and $k = 4$. The reduced problem was also intractable for the 3-SAT brute-force algorithm, and the 3-SAT heuristic was unable to satisfy both reductions. Note that the $k = 3$ reduction might or might not be satisfiable, but the $k = 4$ reduction is satisfiable, as there exists a 4-coloring of the original graph. This means that the DSatur heuristic for Vertex Coloring succeeded in a case where the 3-SAT heuristic could not.

The intractable fully connected graph was intractable for the reduction to 3-SAT, so no results were able to be gathered for that.

2) SHC and HC Vertex Coloring Graphs: The SHC graph was reduced to 3-SAT successfully. This reduction was done for $k = 3$. The reduced problem was satisfied by both the 3-SAT brute-force and heuristic algorithms. This means that the 3-SAT heuristic succeeded in a case where the DSatur heuristic for Vertex Coloring could not.

The HC graph was reduced to 3-SAT successfully. This reduction was done for $k = 3$. The reduced problem was satisfied by both the 3-SAT brute-force and heuristic algorithms. This means that, once again, the 3-SAT heuristic succeeded in a case where the DSatur heuristic for Vertex Coloring could not.

3) Intractable 3-SAT Formula: The provided 3-SAT formula that was intractable for the 3-SAT brute-force algorithm was reduced to a 3-coloring problem successfully. The reduced problem was intractable for the backtracking algorithm, and the DSatur algorithm estimated that the graph had a chromatic number of 4. This means that the DSatur algorithm was unable to find a solution, but since the 3-SAT heuristic was unable to satisfy the original 3-SAT formula, there is no way to know whether or not a 3-coloring of the graph exists.

4) Hard-to-Satisfy 3-SAT Formula: The provided satisfiable 3-SAT formula that was unable to be satisfied by the 3-SAT heuristic was reduced to a 3-coloring problem and a 4-coloring problem successfully. The backtracking algorithm correctly found the 3-coloring, but the 4-coloring problem was intractable for it. The DSatur algorithm estimated that the chromatic numbers for the 3-coloring problem and the 4-coloring problem were 4 and 5, respectively. This means that, interestingly, both the 3-SAT and Vertex Coloring heuristic were unable to correctly solve the problem.

IV. Conclusion

The results clearly show that for most intractable problems, their reduction will also be intractable for brute-force solutions. This makes sense, since reduction (at least the ones analyzed) increase the size of the input.

Findings are more interesting for the hard problems. There are some cases where both the 3-SAT and Vertex Coloring heuristic both failed, but for many cases, one succeeded in the case that the other failed. In a practical situation with an NP-Complete problem, if one has a heuristic unable to find a solution, it could be beneficial to reduce the problem to another NP-Complete problem and try a heuristic for that new problem. This could possibly catch some missing solutions that fell through the cracks with the first heuristic. Results would most likely be best if the two heuristics utilized two completely different strategies; further study could be beneficial in this space.

Acknowledgment

The author would like to thank EJ Yohannan for his willingness to share his exploration of input space for the 3-SAT problem and run his brute-force and heuristic 3-SAT algorithms on reduced notable Vertex Coloring graphs.

References

[1] F. J. A. Artacho and R. Campoy, ”Solving graph coloring problems with the Douglas–Rachford algorithm,” *Set-Valued and Variational Analysis*, 2018, p. 16, doi: 10.1007/s11228-017-0461-4.

[2] R. Janczewski et al., ”The smallest hard-to-color graph for algorithm DSATUR,” *Discrete Mathematics*, 2001, pp. 151-165, doi: 10.1016/S0012-365X(00)00439-8.