NP-Complete Project

CS 570

Andrew Hankins

Graph Coloring

For the NP project, the problem that I chose to examine was the NP-Complete Graph Coloring Problem, or more specifically the NP-Hard optimization version of it in the Minimum Graph Coloring Problem. This problem centers around finding the minimum numbers of that can color an undirected graph (G, E) such that no adjacent vertices share the same color. This solution is often called the chromatic number of the graph. Due to the time complexity of NP-Hard problems, finding this minimum number of colors is not possible in a deterministic amount of time. In fact, since the optimization version is not within the NP problem set, a given solution is also not verifiable in polynomial time. This leads to two possible options. One solution is to create a brute force algorithm that we know will not be able to handle graphs of any non-trivial size. The second is to create a heuristic algorithm that will attempt to guess a close to optimal solution. In the following sections I will discuss my implementation of both of these algorithms and provide a few examples of both of them running.

Brute Force Method

The first algorithm that I will discuss is the brute force version. Implementing the brute force method of the graph coloring algorithm was fairly straight forward. For this algorithm, I would start at one color and check all possible coloring combinations for the graph. In order to check if it was possible to color the graph with this number of colors, the program iterated through each node of the graph, and check to make sure that no adjacent nodes had the same color. If no solution was found i.e. no possible combinations remained, the program then incremented the number of colors allowed and tried again, repeating this process until a valid coloring was found. Using this method guarantees that the optimal solution will be found since all possible combinations from 1…k colors are checked. Therefore if a solution existed that was less it would have been found. As previously mentioned, this is not feasible once the number of vertices reaches any reasonable amount. In fact, during testing once the graph reached about 10 vertices it would take significant amount of time to solve.

Heuristic Method

The second algorithm that I will discuss is the heuristic graph coloring algorithm. Implementing the heuristic algorithm for the graph coloring problem was not nearly as straight forward as the brute force version. This was due to the vast number of options and strategies that could be used. After completing some research, the strategy that I decided to use is as follows. Just as in the brute force approach, the algorithm starts attempting the color the graph at k = 1. This time though, for each number of colors k, one attempt will be made to color the graph using a heuristic method. First the vertices in the graph are sorted in decreasing order of their degree, which is just the number of neighbors they are adjacent to. From there, the algorithm attempts to color the graph in the new sorted order, with each color being selected in the same order (always attempt to use color one first, then color two, etc.). As soon as a valid solution is found, the algorithm exits, guaranteeing a short runtime. The methodology behind this algorithm is that nodes with a higher degree will be harder to color due to them having more potential adjacent colors. Therefore, it would be best to color them at the beginning and get them out of the way. In practice, this worked surprisingly well, which will be shown in the following examples.

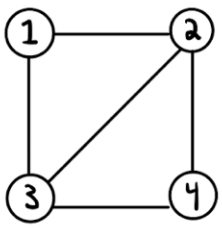
Examples

In this section, I will analyze a few different graphs that I ran through either the heuristic or brute force program. The evaluation of the two programs included testing them with three different types of graphs: a simple graph where both programs produced the optimal solution, an intractable graph where the brute force method failed to find a solution, and an interesting problem that challenged the heuristic approach and resulted in a suboptimal solution.

Simple

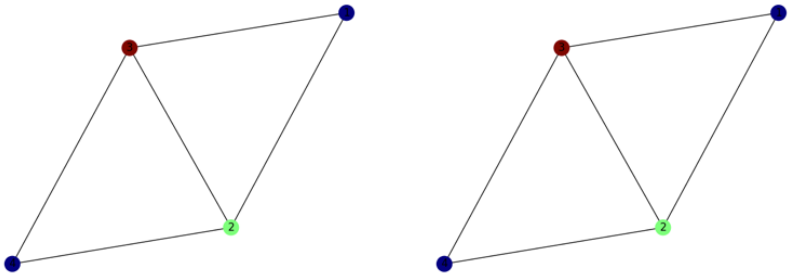
The first example I analyzed was a simple graph, i.e., a graph that only has a few nodes and vertices. As shown below, the graph contained just four nodes, with each node having two or three edges connected to it.

Original:



After running both the brute force and heuristic program on the graphs, the following results were produced.

Brute Force Result: **Heuristic Result:**

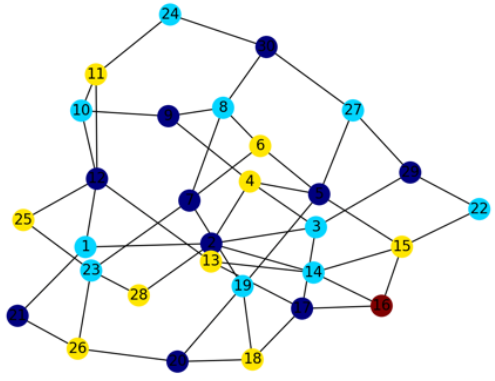


As you can see above, both the heuristic and brute force algorithm produced the same number of colors. In this case it was two, which was also the chromatic number of the graph. Another observation of note was that both approaches essentially found the solution instantaneously. This shows that for simple graphs, the brute force approach is often fast enough and will guarantee that the optimal solution is found.

Intractable

The second graph that I am going to analyze is an intractable one. This is an example of a graph that the brute force approach cannot solve due to the significant amount of time it would take. Therefore, the heuristic program had to be used to produce a close to optimal guess.

Heuristic Result:

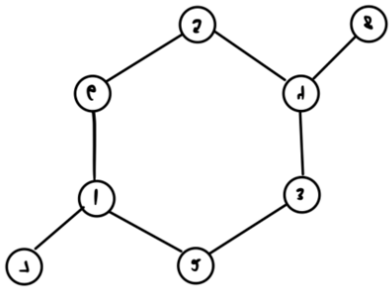


Using the heuristic method for this graph, a chromatic number of 4 was guessed. While we can assume that this is a close to optimal solution, there is no guarantee that it is the true chromatic number. Even at just 30 nodes it becomes very difficult, if not impossible to tell the chromatic number just by looking at the graph, showing why picking a good heuristic is so important.

Interesting

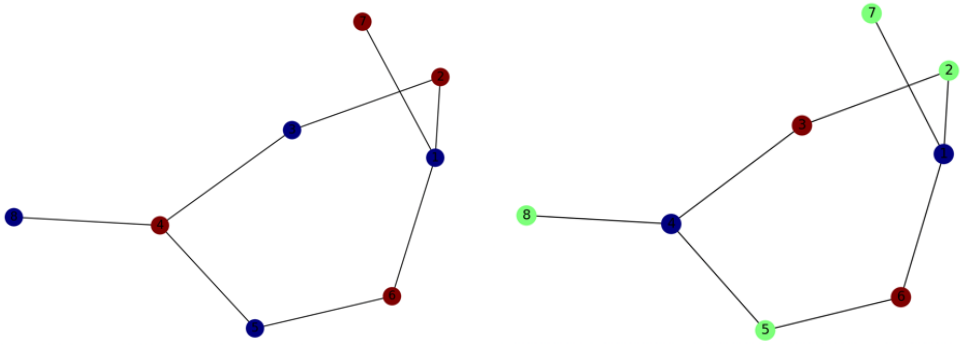
The third and most important example that will be discussed is a graph where the heuristic and brute force strategies produced different results. As shown below, the graph to be analyzed was a hexagon with a single node attached to the two opposing ends.

Original:



After running the brute force and heuristic algorithm, the following results were produced.

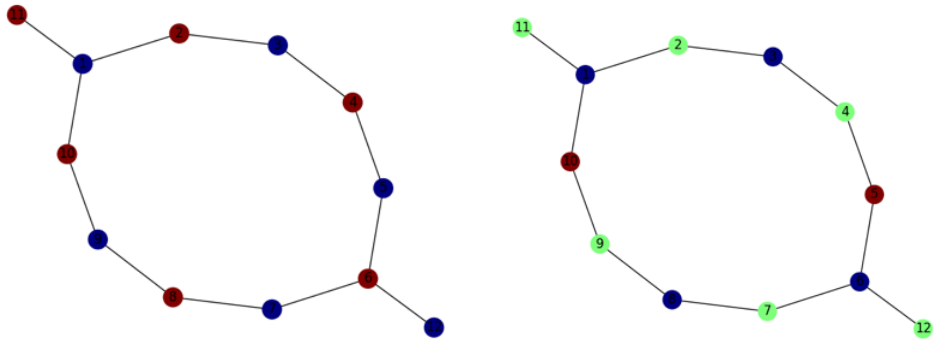
Brute Force Result: **Heuristic Result:**



The results show that while the brute force approach came up with a two coloring, the heuristic approach needed three colors in order to find a solution. Upon closer inspection, we can see why this occurs. Essentially, since the two opposing ends of the hexagon have an additional node attached to them, they both have a degree of three, with every other node having a degree of one or two. This means that they both will be colored first and therefore will be the same color. This then breaks up the graph into two halves with an even number of vertices composing the "bridge" between the two opposing ends. Analyzing this "bridge" allows us to see why the heuristic is generating a suboptimal solution. Starting from one end, if we attempt to color this line of vertices we can color the first node with color 1. Then once we attempt to color the second node, we realize the issue, because the opposing ends are the same color the next node cannot alternate back to color one, and instead has to increment up to color three. In comparison,

for the brute force approach, the algorithm can color the hexagon by just alternating between color one and two all the way around. Upon closer inspection, we can see that this same phenomenon occurs for any even sided polygon with an even number of vertices in the bridge. For example, if the graph was a decagon with an extra node attached to the two ends, the bridge would consist of four vertices, and once again require two new colors. This would then lead to the brute force method once again producing a two coloring, with the heuristic method guessing a suboptimal solution of a three coloring.

Brute Force: **Heuristic:**



Mappings

Due to the graph coloring problem being in the NP-Complete problem space, it is possible to reduce it to and from any other NP-Complete problem in polynomial time. This means that if we can solve a different NP-Complete problem, we can then use the same algorithm to solve the graph coloring problem after the reduction, and vice versa. In the following sections, I will discuss two such mapping, with one mapping to the graph coloring problem, and one mapping from the graph coloring problem.

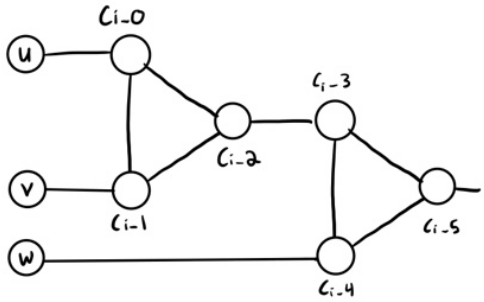
3-SAT to Graph Coloring

The first mapping that I chose to implement was the reduction of the 3-SAT boolean satisfiability problem to the graph coloring problem. Since both of these problems are in the NP-Complete problem space, there exists a polynomial reduction such that solving the graph coloring problem solves the 3-SAT problem. I will first discuss this reduction, and then provide an example of it working.

Steps:

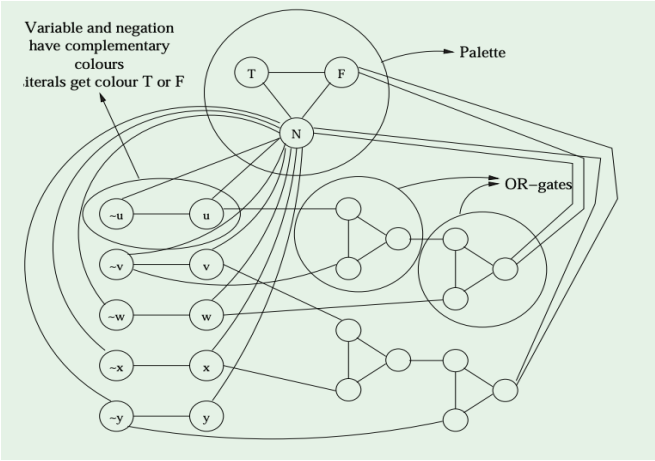
First let us assume that the 3-SAT problem has a 3-SAT formula consisting of m clauses with n variables denoted $x_1, x_2, \ldots, x_n$. The graph that will implement the reduction can be constructed using the following steps:

1. For every variable x_i construct a vertex v_i in the graph and a vertex v_i' denoting the negation of the variable x_i. An edge should then be added between these two vertices.
2. Add three vertices denoted 'T', 'F', and 'B'. These will denote the values True, False, and Base. Connect these vertices such that a triangle is formed.
3. Connect every v_i and v_i' with the vertex 'B'.
4. For each clause *(u or v or w)* using 6 vertices denoted $ci_0, ci_1, \ldots, ci_5$ create an OR gadget graph. The resulting subgraph will resemble the following:



5. Connect every ci_5 node to the 'B' and 'F' node.

Once this graph has been created it will resemble the following:



We then can run the graph coloring algorithm on the graph to determine if it is three colorable. If the graph is three colorable, then we know that the 3-SAT problem has a solution.

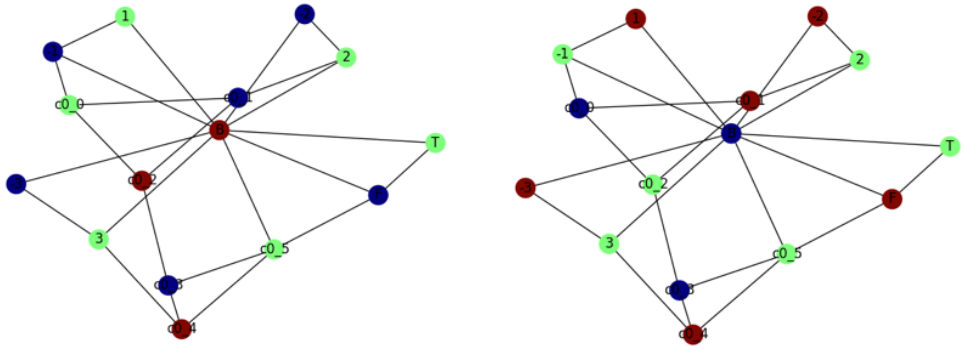
Example 1:

The first example that I will use to demonstrate this reduction is the simplest 3-SAT formula possible.

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

Using this 3-SAT formula, and the above steps, this problem can be reduced to the graph coloring problem. First, we will need to construct two vertices v_i and v_i' for x_1, x_2, and x_3. Each variable vertex should then be connected to its complement with an edge. Next, the vertices 'T', 'F', and 'B' should be created. Once this is done all six vertices that represent the variables, and their complement should be connected to 'B'. Finally, the OR gadget gates will need to be created for each clause of the formula, which in this case is just the one. Now that the reduction has been completed, we can run the graph coloring algorithms previously described on the graph. These results are shown in the following colorings.

Brute Force: **Heuristic:**



As you can see both of the approaches produced a three coloring, proving that the 3-SAT problem is solvable.

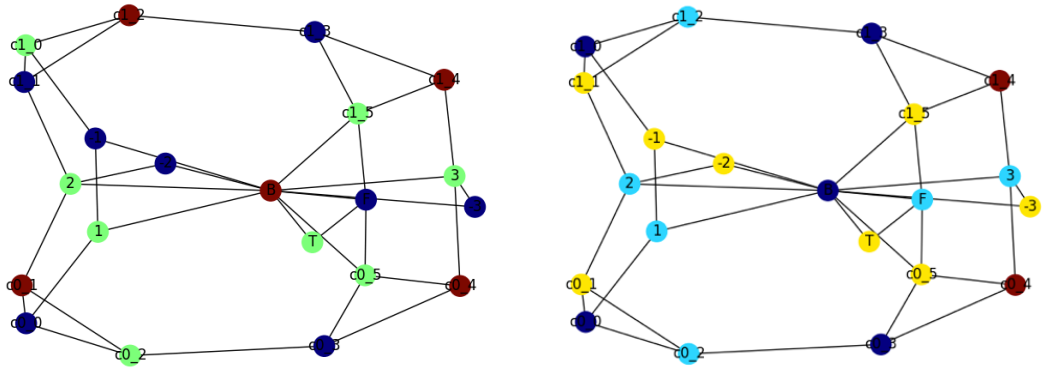
Example 2:

The second example I will analyze presents a potential problem with this technique. The formula we will use will just be slightly more complex than the previous one and include two clauses.

$$(x_1 \vee x_2 \vee x_3) \wedge (\sim x_1 \vee x_2 \vee x_3)$$

Just by looking at this problem, we can easily determine that this formula should be possible to solve, and therefore the resulting graph should be three colorable. In order to test this, I followed the same steps as previously discussed to produce the resulting graphs and then ran it through my graph coloring algorithms.

Brute Force: **Heuristic:**



Based on the above results, we can see that the while the brute force algorithm was able to find a three coloring of the graph, the heuristic approach needed to use four colors which presented a problem. At this point in time, using only two clauses, the brute force approach already took several minutes to complete. This was due to the reduction creating a graph with a significant number of nodes for even just a very simple 3-SAT formula. With the heuristic method not being able to provide a good enough guess, and the brute force algorithm taking an inadmissible amount of time, it became impossible to test any more complicated 3-SAT formulas. This also prevented me from implementing any meaningful chaining, since the most interesting problem I could work with contained only two or three clauses.

Graph Coloring to Clique

The second mapping that I chose to implement was the reduction of the graph coloring problem to the clique problem. Since both of these problems are in the NP-Complete problem space, there exists a polynomial reduction such that solving the clique problem will solve the graph coloring problem. I will discuss this reduction then provide an example of it working.

Steps:

Given an instance of the graph coloring problem with a graph G = (V, E) and k which is the number of colors we would like to test, we can construct a new graph G' as follows:

1. For each vertex v in V, we create k copies of v, denoted v_1, v_2, …, v_k. These copies represent the possible colors of vertex v.
2. Next we will connect the vertices in the new graph G' using the following criteria:
    a. If {u, v} is not an edge in G, connect {u,v} in G'.
    b. If {u, v} is an edge in G **and** color(u) != color(v), connect {u, v} in G'.

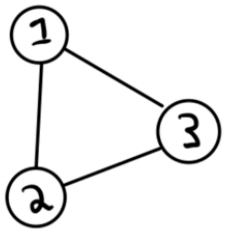
After the new graph is created, we next need to compute n = |V|, which is the number of nodes in the original graph. Once this is computed, the clique algorithm can be run on the new graph to determine if a clique of size n exists.

1. If a clique of size n exists, then the graph is k colorable.
2. If a clique of size n does not exist, then the graph is not k colorable.

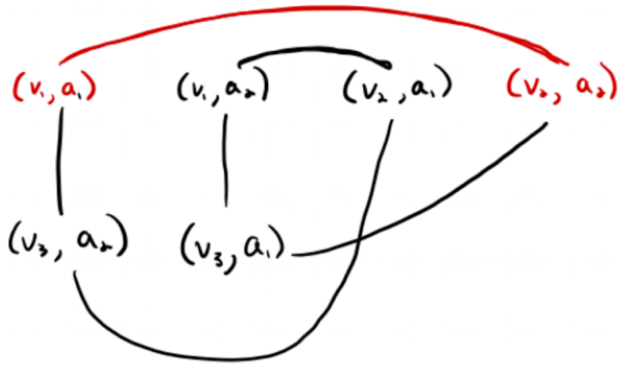
Example 1:

This mapping technique was used on the below graph to see determine how many colors were required to find a legal coloring. I first tested whether the graph was two colorable which should lead to false result, and then I tested whether it was three colorable which should result in true. Unfortunately, no one in class implemented the clique problem algorithm, so all examples were worked by hand.

Original:



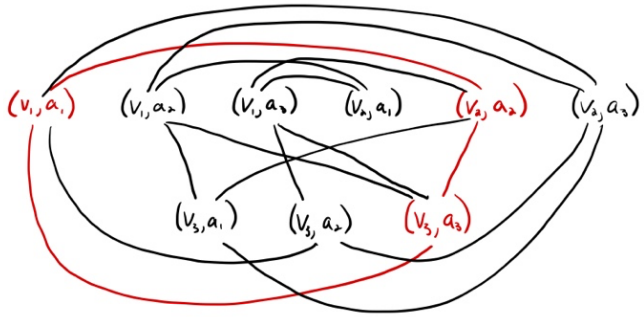
For the first example, I tested whether or not the graph was two colorable, which in this case should be false. Using the above steps, I first needed to create the vertices that would be used in G'. This means that for vertex v_1, v_2, and v_3, two nodes needed to be created. These represented each possible coloring of the node. Next, the nodes in the new graph needed to be connected with edges using the above conditions. This will result in the following graph.



When the clique problem algorithm is run over the graph it will determine that a clique of size 3 does not exist as shown above, with a max clique of two being found. This means that the original graph is not two colorable.

Example 2:

For the second example, I tested whether or not the graph was three colorable, which in this case should be true. I once again followed the above steps but instead of two nodes being created for each original vertex, three nodes were created. I then connected them by creating edges using the previously mentioned steps.



As shown above, a clique of size three does exist in the graph. Since the original graph had three vertices, this means that a graph coloring of three does exist.