NP-Complete Project

CS 570

Andrew Hankins

Graph Coloring

For the NP-Complete project, the problem that I chose to examine was the Graph Coloring Problem. This problem centers around coloring an undirected graph G with a set of edges E, such that no adjacent nodes share the same color. The objective is to minimize the number of colors used while still satisfying the constraint. The smallest possible number of colors that can accomplish this is also called the chromatic number of the graph. Due to the time complexity of problems in the NP space, solving them in polynomial time is not possible, meaning that most of the time a heuristic algorithm will have to be used. A heuristic algorithm provides a good work around, even if it may not always lead to the optimal solution. Therefore, in most situations, using a heuristic algorithm that provides a close to optimal solution is preferable compared to a brute force method that is impractical to implement.

Brute Force Method

Implementing the brute force method of the graph coloring algorithm was fairly straight forward. The first step in the algorithm was to read in the graph from the input file into a python dictionary. From there I would start at one color and check all possible coloring combinations. If no solution was possible, I then incremented the color and tried again, repeating this until a valid coloring was found . Using this method guarantees that the optimal solution will be found since all possible combinations for each k colors is checked, therefore if a solution exists it will be found. As previously mentioned, this is not feasible once the number of vertices reach any reasonable amount. In fact, during testing once a graph reaching about 10 vertices it would take noticeable amount of time to solve.

Heuristic Method

Implementing the heuristic algorithm for the graph coloring was not as straight forward, due to the vast number of options and strategies that could be used. After doing 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. For each k number of colors, one attempt will be made to color the graph using the following method. First the vertices in the graph are sorted in decreasing order of their degree, which is essentially 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 check color 1 first, then color 2, etc.). As soon as a valid solution is found, the algorithm exits, guarantying a relatively fast runtime. The methodology behind this algorithm is that nodes with more neighbors 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 section.

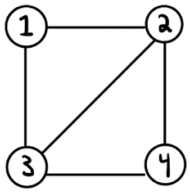
Examples

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 on 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

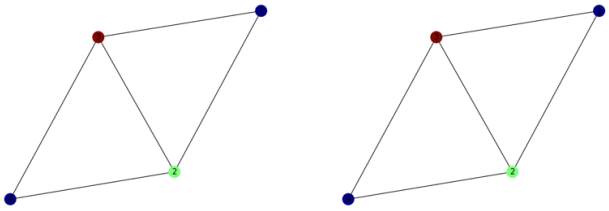
The first example I am going to analyze is a simple graph, i.e., a graph that only has a few nodes and vertices. As shown below, the graph contains just four nodes, with each node having two or three edges connected to it.

Original:



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

Brute Force Result: **Heuristic Result:**

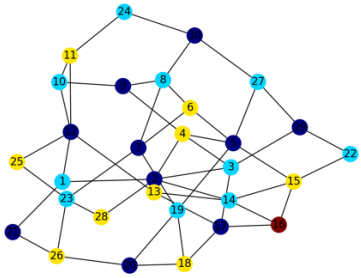


As you can see above, both the heuristic and brute force algorithm produced the same number of colors. In this case it was two, which was also the chromatic number of the graph. Another observation of note was that both approaches essentially found the solution instantaneously. This shows that for simple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

Intractable

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

Heuristic Result:

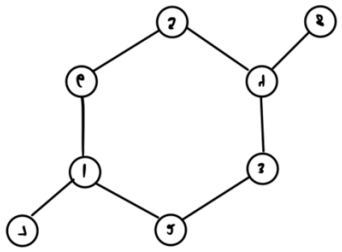


Using the heuristic method for this graph, a chromatic number of 4 is guessed. While we can assume that this is a close to optimal solution, there is no guarantee that it is the true chromatic number.

Interesting

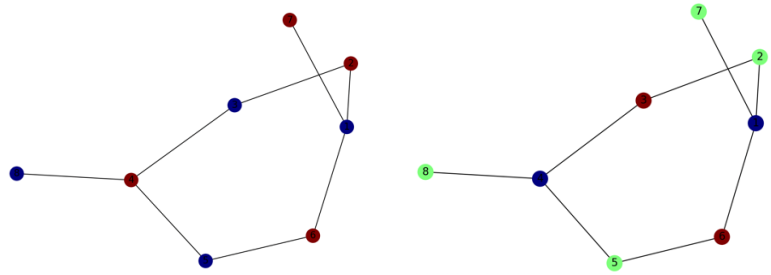
The third and most important example that will be discussed is a graph where the heuristic and brute force strategies produce different results. As shown below, the graph that will be analyzed is a hexagon with a singular node being attached to the opposing ends of it.

Original:

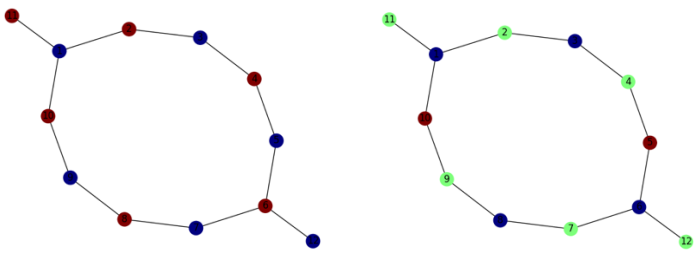


After running the brute force and heuristic algorithm, the following results were produced.

Brute Force Result: **Heuristic Result:**



The results show that while brute force came up with a two coloring, the heuristic method needed three colors in order to find a solution. Upon close 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 two or three. This means that they both will be colored first. This then leads to two nodes remaining bridging the gap on both sides. Due to this, two new colors are required to be used to prevent them from having the same color as the degree three node and the adjacent bridge node. Without coloring the degree three node first, the two opposing ends of the hexagon can end up as different colors allowing the bridge to just alternate the already used colors. Upon close inspection, we can see that this same phenomenon will occur 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 being attached to the two ends, the bridge between them once again require two new colors.



Mappings

Due to the graph coloring problem being in the NP-Hard problem space, it is possible to use the technique of NP-hard mapping to reduce it to and from any other NP-Hard problem. This means that if we can solve another NP-Hard problem using a given algorithm, we can also use the same algorithm to solve the graph coloring problem, 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.

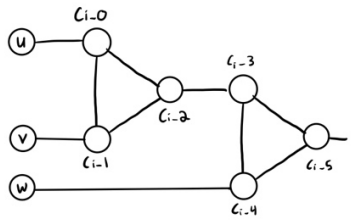
3-SAT to Graph Coloring

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 hard problem space, there exists a polynomial reduction such that solving the graph coloring problem will solve the 3-SAT problem. I will discuss this reduction then provide an example of it working.

Steps:

First let us assume that the 3-SAT problem has a 3-SAT formula of m clauses 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:

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 be the following:



Once this graph has been created, we then can run the graph coloring algorithm on it If the graph is 3 colorable, then we know that the 3-SAT problem has a solution.

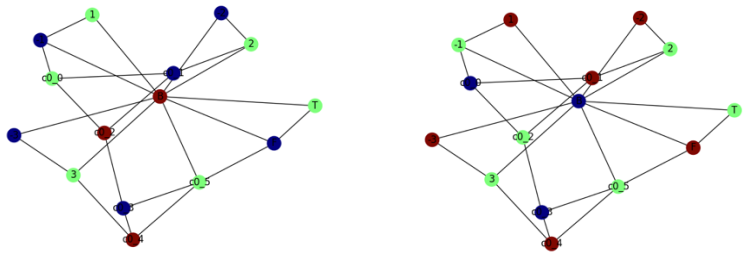
Example 1:

The first example that I will show is the simplest 3-SAT formula possible.

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

Using this formula, and the above steps, this problem can be reduced to a graph coloring. 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, OR gadget gates will need to be created for each clause, 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. This results 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.

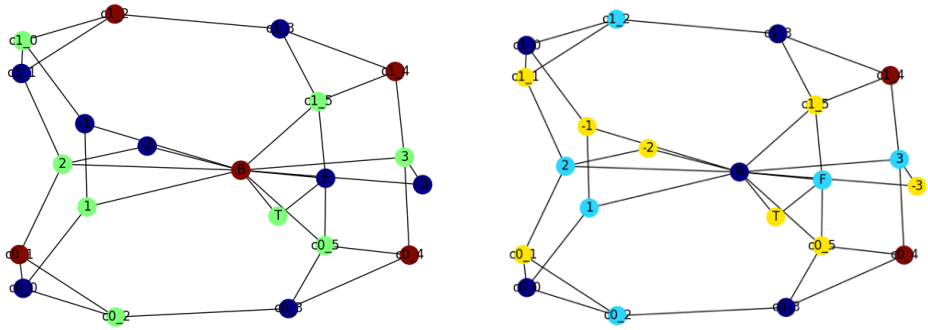
Example 2:

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 discussing to produce the resulting graphs and 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. 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 preventing me from implementing any meaningful chaining, since the most interesting problem I could work with contained only two or three clauses.

Graph Coloring to Max 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 hard 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 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. The k value needs to be the number of colors that you are testing for the graph.
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'.

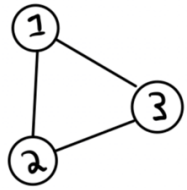
After the new graph is created, we next need to compute n = |V|, which is the number of nodes in the original graph. Once this is computed, the max clique algorithm can be run on the new graph. The final step is to determine if a clique of size n exists.

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.

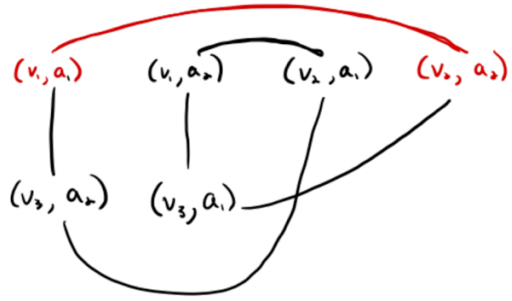
Example:

This mapping technique was used on the below graph to see determine how many colors are needed to find a legal coloring. I will first test whether this graph is two colorable which should be false, and then I will text whether it is three colorable which should be true. Unfortunately, no one in class implemented the clique problem algorithm, so all proofs will be worked by hand.

Original:

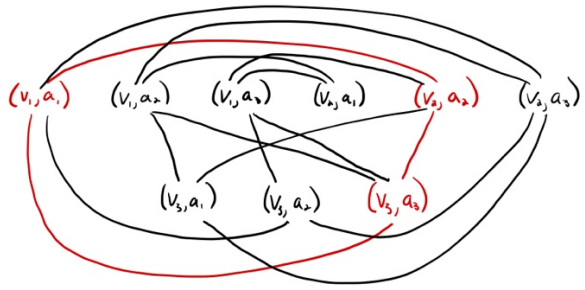


For the first example, I tested whether or not the graph was two colorable, which in this case should be false. Using the above steps, I first needed to create the vertices that will be used in G'. This means that for vertex v_1, v_2, and v_3, two nodes will need to be created. These will represent the possible colorings for the node. Next, the nodes in the new graph will need 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. This means that the original graph is not two colorable.

For the second example, I tested whether or not the graph is 3 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 3 does exist in the graph. Since the original graph had three vertices, this means that a graph coloring of three does exist.