Rosh Sharma
Hasan Asadi
Olivia Glass

First, we needed to implement a valid infrastructure. We created a State object and a Color object in order to start. The State object contains the name of the state and all the states that it is connected to, as extracted from the given file. This is an undirected graph; if State B is in the list of states that A is connected, then state A is also in B's list. Each state is initialized with some color, dummy for backtracking or random for hill climbing. The State object also has a list of BooleanColors that it can be given, these are called the allowedColors; this is a list of colors not already used by adjacent states. The list always has all the colors, but they are given values of True if no adjacent states currently have that color, i.e they are an allowed color, or False otherwise. The Color object holds the name of the color. BooleanColor, as mentioned before, extends that object and allows it to be given a label of True or False, so it can be used in the allowedColors list for each State object.

The first methodology we implemented was backtracking with Most Constrained Variable and Forward Tracking. We wanted to get the most constrained state first. This was defined as the state with the least possible color options that had not already been colored. In order to get this state first, we sorted the list of states by number of constraints, which was the amount of adjacent states that had already been colored. Then we got the first state in that list that had not already been colored. If all the states had been colored, we returned True. Otherwise, we assigned that state the first color in its list of possible, allowed colors. Then, we set that color's label to False in all the adjacent states' allowedColor lists to prevent future adjacent states from being colored the same as the current one. Next, we utilized Forward Tracking. After we assigned the color, we traversed through every state to see if any state had no coloring options left. If the path is still valid — no state has no color options, — then we proceeded to the next most constrained state to color that. If the path is not valid, we went through all the adjacent states and changed the color label back to True in their allowedColor lists and tried the next color in the current state's allowedColor list. We implemented this recursively, so as paths get disabled, it goes to the next possible color for any state. If no color combinations can be found, this method returns False; otherwise, we return the list of states and their assigned colors.

The second methodology we implemented was Hill Climbing. The first step was to assign every state with a random color. Then, we put the list of states with conflicts — adjacent states with the same color — into a queue. Next, we remove a state from the queue, traverse through the colors to see if assigning it a new color would create less or equal conflicts. If so, we assign the state that color and find the new list of states with conflicts. We repeat this step until the queue of states with conflicts is empty. If this runs for longer than a minute, we return False and print out the states with conflicts. If it has run for this long, it is because this list of states will always be in conflict with each other unless you start over — the program is just looping through this list repeatedly until the time limit stops it. If there is not a loop of conflicting states, then we return the list of states and their assigned colors.

Work Breakdown:

All three of us collaborated to brainstorm the initial design. Hasan and Rosh worked more in depth on creating the algorithms, with Rosh doing most of the implementation. Olivia worked with Rosh to understand the code, test it, and write the report.