**Notes**: I made the test file have all the input that you would need for test cases. The first number is how many matrices you want to test then everything under that is formatted to align with how you want the input to be. Also I only have 6 color names in so if you test with k>6 then it will just say (“Color: “ + i).

**Can example 3 have 3 colors:**

After thinking my algorithm was wrong and rewriting it a good amount of times I thought it’d be better to just try it by hand. Thinking that there was a solution since you asked about it and after an amount of time I do not wish to share I can almost confidently say that it is impossible to have example 3 only have three colors.

**Describe which problem-solving techniques you have used:**

I used a combination of 3 different techniques: Backtracking, Brute Force, and Greedy.

1. **Backtracking**, used in graphColoringUtil, which recursively assigned colors to each vertex and once it runs into a conflict it backtracks.
2. **Brute Force**, used in graphColoringUtil, checked all the possible combinations of colors until a solution was found.
3. **Greedy**, used in selectMostBalancedColoring, looked at all the valid solutions and then selected one that was the most balanced.

Using **Backtracking** and **Brute Force** in the same method: If only brute force was used when a conflict arose then Brute Force would continue till the end but by combining it with Backtracking once a conflict was hit then it backtracked. By combining the two it significantly reduced the number of combinations that were checked

**Describe the data structure you have used:**

I used a couple of different data structures to decrease storage size needed and make everything easier. Also, by using multiple data structures I was able to take advantage of their unique built-in functions which heavily streamlined the process.

1. **2D Array**: used to store the adjacency matrix.
2. **Array**, used to store the color assigned to each vertex.
3. **ArrayList**, used in allSolutions, was used to store information that satisfied using every single color. At first my solution returning a solution with the minimum number of colors needed on accident.
4. **HashMap**, used in printSolution, puts each color to corresponding vertex and has methods that is easy to work with for what was needed.

**List the major functions that you implemented:**

1. **graphColoringUtil**: is the main worker methods which tries to color the graph and then stores the solution in the variable ‘allSolutions’
2. **isSafe**: this method checks to see if a color can actually be assigned to a vertex.
3. **selectMostBalancedColoring**: checks all of the ‘allSolutions’ variable and returns the one that is the most balanced.
4. **calculateStandardDeviation:** works with the selectMostBalancedColoring method in finding which solution deviates the least, which in turn returns the most balanced solution.
5. **printSolution:** this method just makes the printed solution look pretty and also actually assigns the color variable a color name unless it is past 6 colors, then it just receives a generic color name.

**Complexity of your solution:**

**Time Complexity:** The time complexity of my solution is . This is because every vertex(V), all possible colors(m) are tried which are recursively called in the graphColoringUtil function.

Side note, if you include inserting the matrix into the program the time complexity is technically since I used a nested for loop, but I wasn’t too worried about making that time efficient since it was only for testing purposes.