```java
package edu.caltech.cs2.coloring;

import edu.caltech.cs2.datastructures.ChainingHashSet;
import edu.caltech.cs2.datastructures.MinFourHeap;
import edu.caltech.cs2.interfaces.IPriorityQueue;
import edu.caltech.cs2.interfaces.ISet;

public class DSatur {
    public static int color(NodeGraph g) {
        // Initialize the color mapping and priority queue
        int[] colorMapping = new int[g.numVertices()];
        IPriorityQueue<Integer> vertexQueue = new MinFourHeap<>();

        // Initialize the used colors set to keep track of already-colored neighbors
        ISet<Integer> usedColors = new ChainingHashSet<>();

        // Initialize the degree of saturation and uncolored neighbor count for each vertex
        int[] saturationDegree = new int[g.numVertices()];
        int[] uncoloredNeighbors = new int[g.numVertices()];

        // Initialize an array to keep track of which vertices are in the priority queue
        boolean[] inQueue = new boolean[g.numVertices()];

        // Add all vertices to the priority queue, ordered by decreasing degree of saturation
        for (int i = 0; i < g.numVertices(); i++) {
            int degree = g.neighbors(i).size();
            saturationDegree[i] = degree;
            uncoloredNeighbors[i] = degree;
            IPriorityQueue.PQElement<Integer> elem =
```

```
29 new IPriorityQueue.PQElement<>(i, -degree);
30             vertexQueue.enqueue(elem);
31             inQueue[i] = true;
32         }
33 
34         // Assign colors to each vertex in order of
   decreasing degree of saturation
35         while (vertexQueue.size() > 0) {
36             IPriorityQueue.PQElement<Integer> vertex
    = vertexQueue.dequeue();
37             if (vertex == null) {
38                 break;
39             }
40             int v = vertex.data;
41             if (colorMapping[v] != 0) {
42                 continue;
43             }
44             ISet<Integer> neighborColors =
   getNeighborColors(g, v, colorMapping);
45             int color = 1;
46             while (neighborColors.contains(color)) {
47                 color++;
48             }
49             colorMapping[v] = color;
50             usedColors.add(color);
51             for (int neighbor : g.neighbors(v)) {
52                 if (colorMapping[neighbor] == 0) {
53                     saturationDegree[neighbor] =
   getNeighborColors(g, neighbor, colorMapping).size();
54                     uncoloredNeighbors[neighbor]--;
55                     IPriorityQueue.PQElement<Integer
   > neighborElem = new IPriorityQueue.PQElement<>(
   neighbor, -saturationDegree[neighbor]);
56                     if (inQueue[neighbor]) {
57                         try {
58                             vertexQueue.decreaseKey(
   neighborElem);
59                         } catch (
   IllegalArgumentException e) {
60                             vertexQueue.enqueue(
   neighborElem);
```

```
61                          }
62                      } else if (saturationDegree[
   neighbor] > 0) {
63                          vertexQueue.enqueue(
   neighborElem);
64                          inQueue[neighbor] = true;
65                      } else {
66                          inQueue[neighbor] = false;
67                      }
68                  }
69              }
70              inQueue[v] = false; // remove vertex
   from the queue
71          }
72
73
74          return usedColors.size();
75      }
76
77      // Helper method to count the number of unique
   colors of a vertex's neighbors
78      private static ISet<Integer> getNeighborColors(
   NodeGraph g, int vertex, int[] colorMapping) {
79          ISet<Integer> neighborColors = new
   ChainingHashSet<>();
80          for (int neighbor : g.neighbors(vertex)) {
81              int color = colorMapping[neighbor];
82              if (color > 0) {
83                  neighborColors.add(color);
84              }
85          }
86          return neighborColors;
87      }
88 }
89
```