```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 boolean 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] = 0;
            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         int numColoredVertices = 0;
36         while (numColoredVertices < g.numVertices
   ()) {
37             IPriorityQueue.PQElement<Integer> vertex
    = vertexQueue.dequeue();
38             int v = vertex.data;
39             if (colorMapping[v] != 0) {
40                 continue;
41             }
42             System.out.println("Coloring vertex " + v
   );
43             ISet<Integer> neighborColors =
   getNeighborColors(g, v, colorMapping);
44             int color = 1;
45             boolean foundColor = false;
46             while (!foundColor) {
47                 foundColor = true;
48                 for (int c : neighborColors) {
49                     if (c == color) {
50                         color++;
51                         foundColor = false;
52                         break;
53                     }
54                 }
55                 if (color > neighborColors.size() + 1
   ) {
56                     // No available colors for this
   vertex
57                     return false;
58                 }
59             }
60             numColoredVertices++;
61             colorMapping[v] = color;
62             usedColors.add(color);
```

```
63              // Update saturation degree and
    uncolored neighbor count for each uncolored neighbor
     of v
64              for (int neighbor : g.neighbors(v)) {
65                  if (colorMapping[neighbor] == 0) {
66                      ISet<Integer> neighborColors1 =
    getNeighborColors(g, neighbor, colorMapping);
67                      int saturation = neighborColors1
    .size();
68                      saturationDegree[neighbor] =
    saturation;
69                      uncoloredNeighbors[neighbor]--;
70                      if (uncoloredNeighbors[neighbor
    ] == 0) {
71                          // Update saturation degree
    for neighbors whose uncolored neighbor count becomes
     zero
72                          for (int n : g.neighbors(
    neighbor)) {
73                              if (colorMapping[n] == 0
    ) {
74                                  saturationDegree[n
    ] = getNeighborColors(g, n, colorMapping).size();
75                              }
76                          }
77                          IPriorityQueue.PQElement<
    Integer> neighborElem = new IPriorityQueue.PQElement
    <>(neighbor, -saturation);
78                          if (inQueue[neighbor]) {
79                              try {
80                                  vertexQueue.
    decreaseKey(neighborElem);
81                              } catch (
    IllegalArgumentException e) {
82                              }
83                          } else {
84                              vertexQueue.enqueue(
    neighborElem);
85                              inQueue[neighbor] = true
    ;
86                          }
```

```
 87                     }
 88                 }
 89             }
 90             // Update the uncolored neighbor count
    for v itself
 91             uncoloredNeighbors[v] = 0;
 92         }
 93         // Check if all vertices are colored
 94         for (int i = 0; i < g.numVertices(); i++) {
 95             if (colorMapping[i] != 0) {
 96                 numColoredVertices++;
 97             }
 98         }
 99         return numColoredVertices == g.numVertices
    ();
100     }
101     // Helper method to count the number of unique
    colors of a vertex's neighbors
102     private static ISet<Integer> getNeighborColors(
    NodeGraph g, int vertex, int[] colorMapping) {
103         ISet<Integer> neighborColors = new
    ChainingHashSet<>();
104         for (int neighbor : g.neighbors(vertex)) {
105             int color = colorMapping[neighbor];
106             if (color > 0) {
107                 neighborColors.add(color);
108             }
109         }
110         return neighborColors;
111     }
112 }
113
```