```java
package vecDeffuant;


import java.util.*;

public class RegionCounter {

    /**
    This class defines functions to analyse  the global distribution of configurations.
*/

    private Vector<DeffuantAgent>  readyQueue = new Vector<DeffuantAgent>(20,10);
    private Vector myNeighborhood;

    int featureCount;
    int threshold;
    double dissociating;
    double impactIntensity;
    ArrayList agentList;
    Grid space;

    /**
        Create the region counter.

        @param featureCount the number of features in the cultural model.
        @param agentList the agents in the cultural model.
        @param space the grid of agents in the cultural model.
    */
    public RegionCounter(int featureCount, ArrayList agentList, Grid space, int threshold,
double dissociating) {
        this.featureCount = featureCount;
        this.agentList = agentList;
        this.space = space;
        this.threshold = threshold;
        this.dissociating = dissociating;
    }

    /**
        @param maxDistance if the number of features that are different exceed this,
            then sites are in different zones.
        @return number of regions that have more than maxDistance features that are
different.
    */
    private int analyzeZones(int maxDistance) {
        // count number of regions (maxDistance=0) or zones(maxDistance = threshold).
        // based on breadth first search. See Stubbs  and Webre, Data Structures, 359ff.
        int i;
        int regions = 0;
        readyQueue.clear();
        for (i = 0; i < agentList.size(); i++) {
            DeffuantAgent agent = (DeffuantAgent) agentList.get(i);
            agent.done = false;
        }

        for (i = 0; i < agentList.size(); i++) {
            DeffuantAgent agent = (DeffuantAgent) agentList.get(i);
            if(!agent.done) {// if not done then record it and visit it.
                ++regions;  // count regions
                visit(agent, maxDistance);  // 0 is min distance allowed for regions.
            }
        }

```

```
61          return regions;
62      }
63
64      private float analyzeAveZonesSize( int maxDistance ) {
65          // Gives the average size of regions (maxDistance=0) or zones(maxDistance =
    threshold).
66          // based on breadth first search. See Stubbs  and Webre, Data Structures, 359ff.
67          int i;
68          int regions = 0;
69          int sumOfRegionSizes =0;
70          readyQueue.clear();
71          for (i = 0; i < agentList.size(); i++) {
72              DeffuantAgent agent = (DeffuantAgent) agentList.get(i);
73              agent.done = false;
74          }
75
76          for (i = 0; i < agentList.size(); i++) {
77              DeffuantAgent agent = (DeffuantAgent) agentList.get(i);
78              if(!agent.done) {// if not done then record it and visit it.
79                  ++regions;  // count regions
80                   sumOfRegionSizes += visit(agent, maxDistance);
81                  visit( agent, maxDistance);  // 0 is min distance allowed for regions.
82              }
83          }
84
85          return (float) sumOfRegionSizes/regions;
86      }
87
88      private int analyzeDisagreements( int maxDistance ) {
89          // count number of regions, and types.
90          // based on breadth first search. See Stubbs  and Webre, Data Structures, 359ff.
91          int i;
92          int disagreements = 0;
93          for (i = 0; i < agentList.size(); i++) {
94              DeffuantAgent agent = (DeffuantAgent) agentList.get(i);
95              disagreements += visitD(agent, maxDistance);
96          }
97
98          return  disagreements/2;
99      }
100
101     private double analyzeSocialPopularity( int feature) {
102         // Gives the number of agents with a configuration
103         // where the specified feature (@param feature) is set to 1 minus
    (totalNumberSites/2).
104         int i;
105         int sites = 0;
106         int totalNumberSites = space.getSizeX()*space.getSizeY();
107         readyQueue.clear();
108         for (i = 0; i < agentList.size(); i++) {
109             DeffuantAgent agent = (DeffuantAgent) agentList.get(i);
110             agent.done = false;
111         }
112
113         for (i = 0; i < agentList.size(); i++) {
114             DeffuantAgent agent = (DeffuantAgent) agentList.get(i);
115             if(!agent.done) {// if not done then record it and visit it.
116                 sites += agent.getTrait(feature);
117             }
118         }
119         return (double) sites-(totalNumberSites/2);
120     }
```

```
121
122
123
124     private int spreadCount( int j) {
125         // Gives the number of agents with a configuration which has the Hamming   j
126         // from the configuration (0,...,0)
127         int i;
128         DeffuantAgent  zeroConfig;
129         int sites = 0;
130
131         int[] broadcastTraits  = new int[featureCount ];
132         for( i = 0; i < featureCount ; i++) broadcastTraits [i] = 0;
133         zeroConfig = new DeffuantAgent(-1, -1,  null, featureCount , 2, broadcastTraits ,
    null, threshold , 0.5, dissociating );
134
135         readyQueue .clear();
136         for (i = 0; i < agentList .size(); i++) {
137             DeffuantAgent  agent = (DeffuantAgent)  agentList .get(i);
138             if(agent.distance( zeroConfig )==j){
139             agent.done = false;}
140         }
141
142         for (i = 0; i < agentList .size(); i++) {
143             DeffuantAgent  agent = (DeffuantAgent)  agentList .get(i);
144             if(!agent.done) {// if not done then record it and visit it.
145                 ++ sites;    // count regions
146                  agent.done= true;
147             }
148         }
149
150         return sites;
151     }
152
153     /**
154         Count the number of regions.
155
156         @return the number of regions.
157     */
158     public int countRegions() { return(analyzeZones(0));}
159
160     /**
161         Count the number of cultural zones. A cultural zone is a set of contiguous sites,
162         each of which has a neighbour  with a "compatible" culture.
163
164         @return the number of zones.
165     */
166     public int countZones() { return(analyzeZones( threshold ));}
167
168     /**
169     Count the number of sites with j features.
170
171     @return the number of ones, twos, threes, ....
172 */
173     public int countSpread( int i) {return(spreadCount( i));}
174
175     public int countDisagreements() { return(analyzeDisagreements( threshold ));}
176
177     public double socialPopularity( int i) {return(analyzeSocialPopularity( i));}
178
179     public float aveZonesSize() { return(analyzeAveZonesSize( threshold ));}
180
181     public float aveRegionSize() { return(analyzeAveZonesSize(0));}
```

```
182
183     private int visit( DeffuantAgent  agent, int criticalDist ) {
184         //Gives the size of the region/zone surrounding an agent.
185         int regionSize = 0;
186         DeffuantAgent  active, neighbour;
187
188         readyQueue.add(agent);    // Add node to ready queue.
189         while(!readyQueue.isEmpty()) {   // while ready queue not empty...
190             // Get node from end of ready queue.
191             active = (DeffuantAgent)  readyQueue.lastElement();
192             readyQueue.removeElementAt( readyQueue.size()-1);
193
194             // Add to ready queue the neighbours  who are legal, not done, and 0 dist .
195             myNeighborhood  = space.getVonNeumannNeighbors( active.x, active.y, false);    //
    If space not torus, can return nulls?
196
197             for(int i = 0; i < myNeighborhood.size(); i++ ) {
198                 neighbour = (DeffuantAgent)  myNeighborhood.get(i);
199                 if( neighbour.done ) continue;
200                 if(active.distance(neighbour) <= criticalDist ) //crit distance = 0 for
    region, bitmax-1 for zone}
201                     {
202                     ++ regionSize;
203                      neighbour.done = true;
204                      readyQueue.add(neighbour);
205                     }
206             }
207         }
208         if(regionSize == 0) regionSize = 1; // Needed for 1 x 1  regions which have no valid
    neighbours.
209         return regionSize;
210     }
211
212     private int visitD( DeffuantAgent  agent, int criticalDist ) {
213         //Gives the number of neighbours  a specified agent disagrees with.
214         int disagreements  = 0;
215         DeffuantAgent  neighbour;
216
217             myNeighborhood  = space.getVonNeumannNeighbors( agent.x, agent.y, false); // If
    space not torus, can return nulls?
218
219             for(int i = 0; i < myNeighborhood.size(); i++ ) {
220                 neighbour = (DeffuantAgent)  myNeighborhood.get(i);
221                 if( neighbour.done ) continue;
222                 if(agent.distance(neighbour) > criticalDist )
223                     {
224                     ++ disagreements;
225                     }
226             }
227         return disagreements;
228     }
229
230 }
231
```