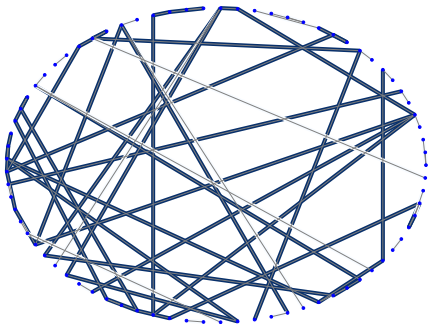
Cultural dissemination and self segregation in social communities



Peer To Peer Systems
And Blockchains

a.y. 2018-2019

Contents

1 Introduction

The problem of understanding whether groups of people tend to separate or aggregate according to their cultural background is crucial if we are interested in protecting the diversity of a multicultural society. Many people focused on this problems from the 90's of the Twentieth Century ([1], [3] [2]) to the present time and our aim is to confirm their theories, answering to the question: "why do differences persist although cultures tend to merge?".

2 Our model

Our analysis is based on Axelrod assumption of *homophilic social influence* , that states that an individual A acquires a certain "habit" from B if they agree on many others.

Let us introduce some formalism to express this model.

First of all, we assume to represent the spacial distribution of sites (places where individuals can live) as a graph $G(N, E)$.

Definition 2.1 (Culture). *A **culture** is represented as a set of* features *(or dimensions) $\mathscr{F}$, that has values in a set called* cultural traits *, which for simplicity is mapped into contguous integers, namely $\mathscr{T} = \{0, 1, \ldots, q-1\}$.*

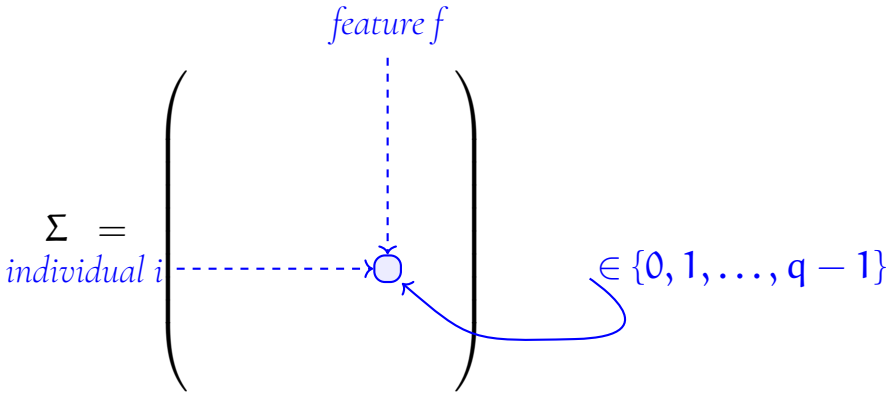
Example 2.1. *Let us assume that the culture of any individual belonging to a given society is represented as the colour of the bridal dress, the colour for mourning, the colour to express love and the color to express peace. The set of features $\mathscr{F}$ contains the following 4 elements: bride, mourn, love, peace; each of the ones has a value in the set of hex colours (mapped into integers for sake of representation), hence any cultural trait belongs to the set $\mathscr{T} = \{0, 1, \ldots, 16^6 - 1\}$. The individual Yoda, belonging to Jedi culture, can hence be represented as follows:*

Hence, its culture is expressed in our formalism as

1	2	3	4
255	153	15133951	10075135

Definition 2.2 (Cultural region). *A **cultural region** is a set of contiguous sites with identical culture.*

Definition 2.3 (Community). *We term* ***community*** *a set of individuals* $\mathscr{I}$*, spread on a network* $G(N, E)$*, together with their culture. The values of all such cultural traits are stored in a* $I \times F$ *matrix, called* Σ*, where* $I = |\mathscr{I}|$ *and* $F = |\mathscr{F}|$*.*



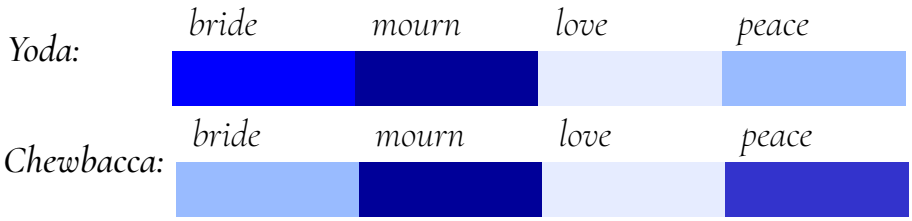
Definition 2.4 (Similarity). *The* ***similarity*** *between two individuals* A, B *is denoted as* $\omega_{A,B}$ *and expresses in a scale from* 0 *to* 1 *the percentage of indentical values for each feature.*

Formally,

$$\omega_{A,B} = \frac{1}{F} \times \sum_{k=1}^{F} \delta(\Sigma[i][k], \Sigma[j][k])$$

Where the function $\delta : \mathscr{T} \times \mathscr{T} \to \{0, 1\}$ *is the notorious Kronecker's delta.*

Example 2.2. *Resorting Yoda from the previous example*



and comparing him with Chewbacca, belonging to the Wookiee culture, we can observe that their similarity is computed as:

$$\omega_{Yoda,Chewie} = \frac{1}{4} \cdot (0 + 1 + 1 + 0) = \frac{1}{2}$$

We are now ready to describe the algorithm proposed by Gracia in [2] for the cultural influence.

Algorithm 1 Pseudocode for the behaviour of individuals

procedure INFLUENCE(G, $\mathscr{I}$, M, T, $\mathscr{F}$, Σ, q)
  convergence←FALSE;
  $m \leftarrow 0$;
  while ($m < M$ AND convergence =FALSE) **do**
    $m \leftarrow m + 1$;
    for $i \in \mathscr{I}$ **do**
      neighs = NEIGH(i);
      if (neighs IS EMPTY) **then**
        i moves to random empty site;
        CONTINUE;
      end if
      $j \leftarrow$ RANDOM(neighs);
      $\omega_{i,j} \leftarrow \frac{1}{F} \times \sum_{k=1}^{F} \delta(\Sigma[i][k], \Sigma[j][k])$;
      $k \leftarrow$ RANDOM($\{1, \ldots, F\} \setminus \{\text{shared features}\}$);
      $p_i \leftarrow$RANDOM($[0, 1]$);
      if ($p_i \leqslant \omega_{i,j}$) **then**
        $\Sigma[i][k] \leftarrow \Sigma[j][k]$;
      else
        $\omega_i \leftarrow 0$;
        for $j \in$ neighs **do**
          $\omega_i \leftarrow \omega_i + \frac{1}{F} \times \sum_{k=1}^{F} \delta(\Sigma[i][k], \Sigma[j][k])$;
        end for
        $\omega_i \leftarrow \frac{\omega_i}{\text{Size(neighs)}}$;
        if ($\omega_i < T$) **then**
          i moves to random empty site;
        end if
      end if
    end for
    convergence $\leftarrow$ CHECKCONVERGENCE();
  end while
end procedure

Where NEIGH(i) is the set of indexes of nodes in the neighbourhood of the i-th node and the function RANDOM takes a set as input and returns an element of such set (notice that $[0, 1]$ is the real interval from 0 to 1).

The constant T is called **intollerance threshold** and it expresses how likely are individuals towards cultural change. Moreover, CHECKCONVERGENCE is a function that returns a boolean according to the query: "has the procedure converged?".

It is important to notice that the indexes used to refer to individuals do not coincide with the ones that refer to nodes, so each reference to functions like NEIGH(i) take into account the fact that an individual may e phisically isolated or "virtually" isolated (the neighbouring sites are not inhabited).

3 Structure of experiments

Our analysis was run selecting the following values for the parameters:

N, the number of nodes in the network was picked in
$\{10, 20, 25, 64, 100, 225, 400, 1000\}$;

I, the number of individuals present in the network was chosen as a function of N, varying in $\{\frac{N}{10}, \frac{N}{5}, \frac{N}{2}\}$;

pn, the probability of two nodes being neighbours was chosen in the set
$\{0.016, 0.1, 0.2, 0.5, 0.7\}$;

T, the tollerance threshols was picked from $0.1, 0.2, 0.3, 0.5, 0.7, 0.9$;

M, the number of allowed iterations was chosen in $\{100, 1000, 10000\}$;

q, the number of cultural traits belongs to $\{5, 10, 15\}$;

F, the number of features (or dimension) was chosen among $5, 10, 15$.

Let us analyze the reasons behind this choice: as long as N is concerned, the state of the art on the subject ([1]) used such values to perform sperimentation, hence we decided not to proceed further on this parameter. On the other hand, the values of I were picked in a way to allow sparser and denser territories, in the reasonable ranges.

The choice of pn is indeed more sophisticated. The state of the art used only grid graphs, where the degree of central nodes is 4, becomes three on edges and 2 on vertexes. We performed a different choice, building a random graph and making some observations on the neighbouring probability of nations around the World (0.01586594391). For further details on this computation see Appendix A. Moreover, we decided to use some more values to increase the graph completeness. This choice is motivated by the need of augmenting the number of neighbours for

each node, which is equivalent to say that we allow our individual to interact with more people, which is a suitable hypothesis in a globalized world.

The tolerance threshold T was picked in a superset of the one used by Gracia in [2] in order to give more flexibility to the model.

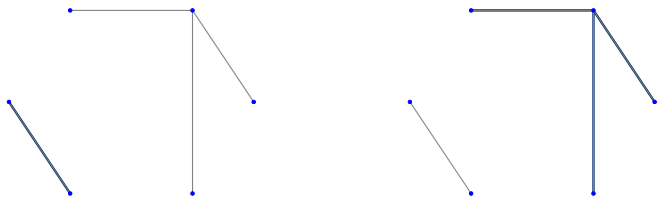
As long as M is concerned, we allowed an incremental amount of iterations for convergence.

The number of features F and the number of different cultural traits were picked according to Axelrod's analysis.

4 Experimental results

We are now ready to discuss what we found out in this analysis. In the plots, the grey edges are the phisical ones, while the thicker blue edges represent the individual similarity according to a blue scale.

We noticed that the convergence speed increased with the reciprocal of the sparity of the graph, as an example see Figure 1.



(a) *At the beginning* *(b)* *After 9999 steps*

Figure 1: $N = 20, I = 10, pn = 0.016, T = 0.1, M = 10000, q = 5, F = 5$

Such a conclusion was reached, analising multiple experiments repeted on the same parameters values and applying the clustering method K-means on such data. The clusters obtained are 5 and the silhouette of such clustering is 0.812, which is a very good clustering indees. In such clustering, two clusters correspond to those territories (networks) which have a very little number of nodes and of inhabitants, hence are not significant. For a pictorial representation of such graphs, see Figure 2

Moreover, we noticed that fewer and larger stable regions (contiguous individuals sharing the same culture) are found in presence of few cultural features (small F), but assuming many different values (large q), an example can be observed in Figure 3 and Figure 4.

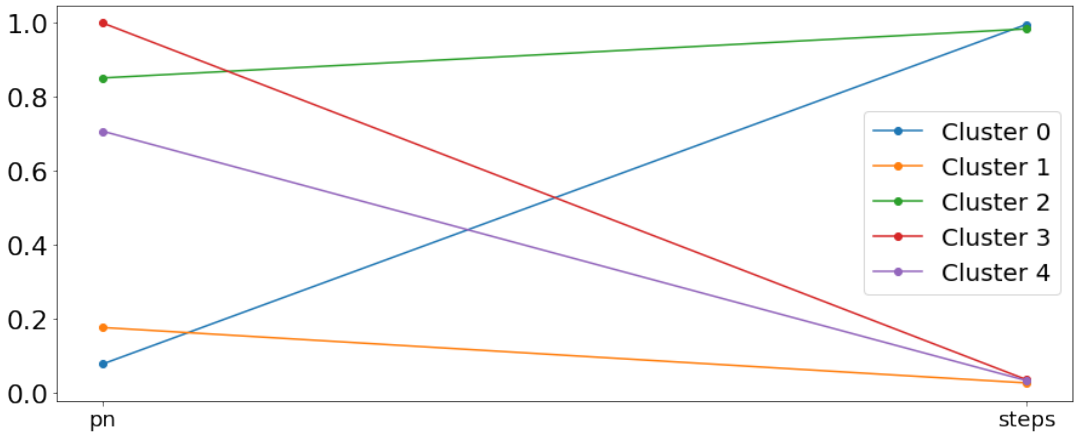
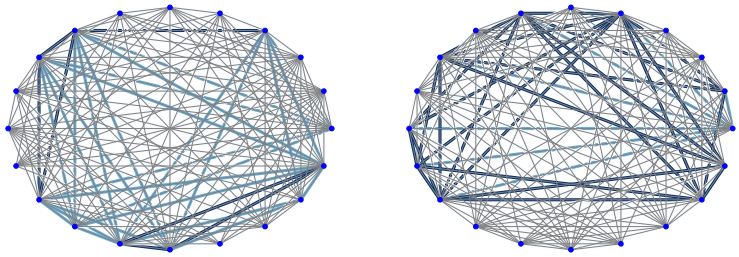
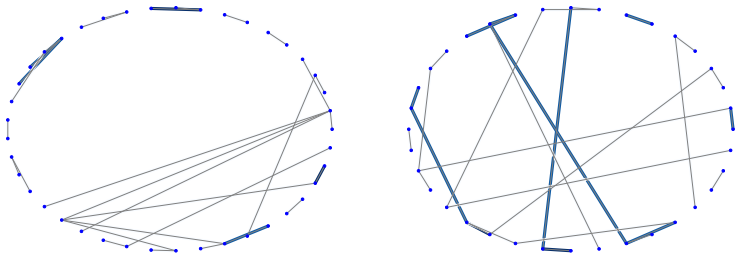


Figure 2: *The five clusters and their values on* $\mathtt{pn}$ *and on the number of steps needed for convergence. Both values of* $\mathtt{pn} \in \{0.016, 0.1, 0.2, 0.5, 0.7\}$ *and* $\mathtt{N}_{steps} \in \{0, \ldots, 10000\}$ *are normalized. The clusters number* 1 *and* 2 *are formed by low* $\mathtt{N}$ *and* $\mathtt{I}$ *values, hence do not break our hypothesis.*



(a) After 99 steps *(b) After 9999 steps*

Figure 3: $\mathtt{N} = 20, \mathtt{I} = 10, \mathtt{pn} = 0.7, \mathtt{T} = 0.5, \mathtt{q} = 15, \mathtt{F} = 5$



(a) At the beginning *(b) After 99 steps*

Figure 4: $\mathtt{N} = 64, \mathtt{I} = 32, \mathtt{pn} = 0.016, \mathtt{T} = 0.1, \mathtt{q} = 15, \mathtt{F} = 5$

5 Conclusions

Our analysis leads to the conclusion that the number of stable homogeneous regions increases with the number of different traits per feature, while descreases

when the number of cultural features or the size fo the territory increases.

Moreover, the convergence speed is inversely proportional to the sparsity of the graph.

These conclusions confirm the state of the art theories.

For future research it could be interesting to analyze connected components and their shape from an algeraic point of view and repeat all experiments on bigger territories.

A Neighbours computation

Country	Nº of neighbours
Abkhazia	2
Afghanistan	6
Albania	4
Algeria	7
Andorra	2
Angola	4
Antigua and Barbuda	0
Argentina	5
Armenia	4
Australia	0
Austria	8
Azerbaijan	5
Bahamas	0
Bahrain	0
Bangladesh	2
Barbados	0
Belarus	5
Belgium	4
Belize	2
Benin	4
Bhutan	2
Bolivia	5
Bosnia and Herzegovina	3
Botswana	4
Brazil	10
Brunei	1
Bulgaria	5
Burkina Faso	6
Burundi	3
Cambodia	3

Country	Nº of neighbours
Cameroon	6
Canada	1
Cape Verde	0
Central African Republic	6
Chad	6
Chile	3
Colombia	5
Comoros	0
Costa Rica	2
Côte d'Ivoire	5
Croatia	5
Cuba	0
Cyprus	1
Czech Republic	4
Democratic Republic of the Congo	9
Denmark	1
Djibouti	3
Dominica	0
Dominican Republic	1
East Timor	1
Ecuador	2
Egypt	4
El Salvador	2
Equatorial Guinea	2
Eritrea	3
Estonia	2
Eswatini (Swaziland)	2
Ethiopia	6
Federated States of Micronesia	0

Country	Nº of neighbours
Fiji	0
Finland	3
France	8
Gabon	3
Georgia	4
Germany	9
Ghana	3
Greece	4
Grenada	0
Guatemala	4
Guinea	6
Guinea-Bissau	2
Guyana	3
Haiti	1
Honduras	3
Hong Kong	1
Hungary	7
Iceland	0
India	6
Indonesia	3
Iran	7
Iraq	6
Ireland	1
Israel	4
Italy	6
Jamaica	0
Japan	0
Jordan	5
Kazakhstan	5
Kenya	5
Kiribati	0

Country	Nº of neighbours
Kosovo	4
Kuwait	2
Kyrgyzstan	4
Laos	5
Latvia	4
Lebanon	2
Lesotho	1
Liberia	3
Libya	6
Liechtenstein	2
Lithuania	4
Luxembourg	3
Madagascar	0
Madeira (Portugal)	0
Malawi	3
Malaysia	3
Maldives	0
Mali	7
Malta	0
Marshall Islands	0
Mauritania	4
Mauritius	0
Mexico	3
Moldova	2
Monaco	1
Mongolia	2
Montenegro	5
Morocco	3
Mozambique	6

Country	Nº of neighbours
Myanmar	5
Namibia	4
Nauru	0
Nepal	2
Netherlands	2
New Zealand	0
Nicaragua	2
Niger	7
Nigeria	4
North Korea	3
North Macedonia	5
Norway	3
Oman	3
Pakistan	4
Palau	0
Palestine	0
Panama	2
Papua New Guinea	1
Paraguay	3
People's Republic of China	14
Peru	5
Philippines	0
Poland	7
Portugal	1
Qatar	1
Republic of the Congo	5
Romania	5
Russia	14
Rwanda	4

Country	Nº of neighbours
Saint Kitts and Nevis	0
Saint Lucia	0
Saint Vincent and the Grenadines	0
Samoa	0
San Marino	1
São Tomé and Príncipe	0
Saudi Arabia	7
Senegal	5
Serbia	8
Seychelles	0
Sierra Leone	2
Singapore	0
Slovakia	5
Slovenia	4
Solomon Islands	0
Somalia	3
South Africa	6
South Korea	1
South Ossetia	2
South Sudan	6
Spain	5
Sri Lanka	0
Sudan	7
Suriname	3
Sweden	2
Switzerland	5
Syria	5
Tajikistan	4

Country	Nº of neighbours
Tanzania	8
Thailand	4
The Gambia	1
Togo	3
Tonga	0
Trinidad and Tobago	0
Tunisia	2
Turkey	8
Turkmenistan	4
Tuvalu	0
Uganda	5
Ukraine	7
United Arab Emirates	2
United Kingdom	1
United States	2
Uruguay	2
Uzbekistan	5
Vanuatu	0
Vatican City	1
Venezuela	3
Vietnam	3
Western Sahara	3
Yemen	2
Zambia	8
Zimbabwe	4

Thanks to these values, we obtain that the probability of two countries to be neighbours is $\frac{201}{779} = 0.01586594391$.

B Code

Here is the Java code, used for the experiments.

```java
import java.util.HashMap;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Random;

/**
 * @author gemma This class represents a population
 *         distributed in a certain location. It provides
 *         some methods that model its behaviour and some
 *         utilities for dealing with the Graph library,
 *         provided by Princeton University
 *
 */
public class Population {
  Graph G;
  // Obs: the function V(G) returns the number of nodes of
  // the graph G
  int N;// The total number of sites in the area
  int I;// The total number of individuals in the system
  double pn;// Probability of being neighbour sites
  double T;// Threshold for migration from one site to
        // another one in the
        // graph
  int M;// Maximum number of steps
  int q;// Maximum value of any cultural trait
  int F;// Amount of cultural traits
  String fileName;
  HashMap<Integer, Integer> indToNode;// This is ”full”,
                    // since N>I
  HashMap<Integer, Integer> nodeToInd;// If a node is
                    // empty -1 is the
                    // value of
                    // the key
  ArrayList<ArrayList<Integer>> sigma;// Works with
                    // individuals
  // sigma[individualx][featurek] -> the trait of the
  // featurek-th feature of
```

```
// individualx

/**
 * This method turns an Iterable<Integer> into an
 * ArrayList<Integer>
 *
 * @param itr the iterable
 *
 * @return ret the ArrayList
 */
private static ArrayList<Integer>
    toArray(Iterable<Integer> itr) {
  ArrayList<Integer> ret = new ArrayList<>();
  for (Integer t : itr) {
    ret.add(t);
  }
  return ret;
}

/**
 * This method returns the list of empty sites in the
 * current population
 *
 * @return res the list of integers
 */
private ArrayList<Integer> emptyNodes() {
  ArrayList<Integer> res = new ArrayList<Integer>();
  for (int i = 0; i < this.nodeToInd.size(); i++) {
    if (nodeToInd.get(i) == -1) {
      res.add(i);
    }
  }
  return res;
}

/**
 * This method computes the similarity between two
 * sites, aka the similarity between the inhabitants of
 * such sites. It goes without saying that if at least
```

```
 * one of such nodes is empty an ”error code”
 *
 * @param nodei
 * @param nodej
 * @return the value of similarity (between 0 and 1), it
 *         returns -1 in case of empty site(s).
 *
 *         Notice that we decided not to use an
 *         exception because it was easier to manage in
 *         the code
 */
private double computeSimilarity(int nodei, int nodej) {
  int individuali = (int) this.nodeToInd.get(nodei);
  int individualj = (int) this.nodeToInd.get(nodej);
  // If at least one of such nodes is not populated
  if (individuali == -1 || individualj == -1) {
    return -1;
  }
  double sym = 0;
  for (int k = 0; k < this.F; k++) {
    int sigmai = (int) this.sigma.get(individuali)
        .get(k);
    if (sigmai == (int) this.sigma.get(individualj)
        .get(k)) {
      sym += 1;
    }
  }
  sym = sym / this.F;
  return sym;
}

/**
 * This method allows an individual to move towards an
 * empty site in the network
 *
 * @param r           for random choices
 * @param individualx the moving individual
 */
private void moveToRandomEmptySite(Random r,
```

```
    int individualx) {
  int nodex = (int) this.indToNode.get(individualx);
  ArrayList<Integer> res = this.emptyNodes();
  int idx = r.nextInt(res.size());
  int newNode = res.get(idx);
  // Change individual-> node correspondence
  this.indToNode.put(individualx, newNode);
  // Set new node
  this.nodeToInd.put(newNode, individualx);
  // Unset old node
  this.nodeToInd.put(nodex, -1);
}

/**
 * This function writes the weighted graph into a file,
 * using the format <from, to, weight> The weight
 * represents the similarity between two adjacent nodes.
 * If at least one of such nodes is not populated the
 * weight of the edge is -1. If a node is isolated, it
 * is printed without any neighbour
 *
 * @param outfile the path of the output file
 * @param iter    the number of the iteration
 * @throws IOException
 */
private void writeWeightedGraph(String outfile,
    int iter) throws IOException {
  BufferedWriter writer =
      new BufferedWriter(new FileWriter(
          outfile + iter + ".csv", true));
  for (int nodei = 0; nodei < this.N; nodei++) {
    Iterable<Integer> neighs = this.G.adj(nodei);
    // If there is at least a neighbour write line
    int numNeighs = 0;
    for (Integer nodej : neighs) {
      numNeighs++;
      String line = String.valueOf(nodei);
      line = line + " " + String.valueOf(nodej);
      line = line + " " + String.valueOf(this
```

```
            .computeSimilarity(nodei, nodej));
        writer.write(line + "\n");
      }
      // Else write only node number
      if (numNeighs == 0) {
        String line = String.valueOf(nodei);
        writer.write(line + "\n");
      }
    }
    writer.close();
  }

  /**
   * This method implements the migrations as described in
   * the algorithm
   *
   * @param r -> for random generation
   */
  private void startMigrations(Random r) {
    boolean convergence = false;
    int m = 0;
    while (m < this.M && !convergence) {
      // Make copy of the current state for testing
      // convergence
      HashMap<Integer, Integer> currIndToNode =
          new HashMap<Integer, Integer>();
      for (int i = 0; i < this.indToNode
          .size(); i++) {
        currIndToNode.put(i, this.indToNode.get(i));
      }
      ArrayList<ArrayList<Integer>> currSigma =
          new ArrayList<ArrayList<Integer>>();
      for (int i = 0; i < this.N; i++) {
        ArrayList<Integer> row = new ArrayList<>();
        for (int j = 0; j < this.F; j++) {
          row.add(this.sigma.get(i).get(j));
        }
        currSigma.add(row);
      }
```

```
// Here the algorithm begins
m++;
for (int individuali =
    0; individuali < this.I; individuali++) {
  int nodei = (int) this.indToNode
      .get(individuali);
  Iterable<Integer> neighs =
      this.G.adj(nodei);
  ArrayList<Integer> arrNeighs =
      Population.toArray(neighs);
  if (arrNeighs.size() == 0) {
    // individual moves to random empty site
    moveToRandomEmptySite(r, individuali);
    continue;
  }
  ArrayList<Integer> populatedNeighs =
      new ArrayList<Integer>();
  for (int i = 0; i < arrNeighs.size(); i++) {
    if (this.nodeToInd.get(i) != -1) {
      populatedNeighs.add(i);
    }
  }
  int numPopulatedNeighs =
      populatedNeighs.size();
  if (numPopulatedNeighs == 0) {
    // individual moves to random empty site
    moveToRandomEmptySite(r, individuali);
    continue;
  }
  boolean nodeFound = false;
  int individualj = -1;
  int nodej = -1;
  double omegaij = -1;
  int iterNum = 0;
  while (!nodeFound
      && iterNum < numPopulatedNeighs
          * numPopulatedNeighs) {
    int idx = r.nextInt(numPopulatedNeighs);
    iterNum++;
```

```
  nodej = populatedNeighs.get(idx);
  individualj =
      (int) this.nodeToInd.get(nodej);
  if (individualj != -1) {
    omegaij = computeSimilarity(nodei,
        nodej);
    // If the two nodes do not share the
    // same culture
    if (omegaij != 1) {
      nodeFound = true;
    }
  }
}
// If the i-th individual has all
// same-culture neighs, visit
// next individual
if (!nodeFound) {
  continue;
}
int k = r.nextInt(this.F);
;
while (this.sigma.get(individuali).get(k)
    .equals(this.sigma.get(individualj)
        .get(k))) {
  // The cultural influence must occur
  // among the differing
  // ones
  k = r.nextInt(this.F);
}
double pi = Math.random();
if (pi <= omegaij) {
  ArrayList<Integer> rowi =
      this.sigma.get(individuali);
  ArrayList<Integer> rowj =
      this.sigma.get(individualj);
  rowi.set(k, rowj.get(k));
  this.sigma.set(individuali, rowi);
} else {
  double omegai = 0;
```

```
      for (individualj =
          0; individualj < arrNeighs
              .size(); individualj++) {
        omegai = computeSimilarity(nodei,
            nodej);
      }
      omegai = omegai / arrNeighs.size();
      if (omegai < this.T) {
        // Individual moves to random empty
        // site
        moveToRandomEmptySite(r,
            individuali);
      }
    }
  }
  // Check convergence
  if (this.indToNode.equals(currIndToNode)
      && this.sigma.equals(currSigma)) {
    convergence = true;
    System.out.println(
        "~~~~~~~~~~Convergence with " + m
            + " iterations~~~~~~~~~~");
    try {
      writeWeightedGraph(fileName, m);
    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }
  else {
    // Write graph to file each 100 steps
    if (m % 100 == 0) {
      try {
        writeWeightedGraph(fileName, m);
      } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
    }
```

```
      }
    }
    if (!convergence) {
      System.out.println(
          "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~NO Convergence~~~~~~~~~~~~~~~~~~");
    }

  }

  /**
   * Constructor
   *
   * @param r
   * @param N
   * @param I
   * @param pn
   * @param T
   * @param M
   * @param q
   * @param F
   * @param fileName
   */
  public Population(Random r, int N, int I, double pn,
      double T, int M, int q, int F,
      String fileName) {
    this.N = N;
    this.I = I;
    this.pn = pn;
    this.T = T;
    this.M = M;
    this.q = q;
    this.F = F;
    this.fileName = fileName;
    // Generation of the graph
    G = GraphGenerator.simple(N, pn);
    // Generation of the dictionaries <individual,
    // nodeID> and <nodeID,
    // individual>
    this.indToNode = new HashMap<Integer, Integer>();
```

```
    this.nodeToInd = new HashMap<Integer, Integer>();
    for (int i = 0; i < I; i++) {
      for (int j = 0; j < N; j++) {
        this.nodeToInd.put(j, -1);
      }
    }
    for (int i = 0; i < I; i++) {
      // For each individual a node in the network is
      // chosen at random
      int node = r.nextInt(this.N);
      while (this.nodeToInd.get(node) != -1) {
        node = r.nextInt(this.N);
      }
      // If the node is not inhabited
      this.indToNode.put(i, node);
      this.nodeToInd.put(node, i);
    }
    this.sigma = new ArrayList<ArrayList<Integer>>();
    for (int i = 0; i < this.N; i++) {
      ArrayList<Integer> row = new ArrayList<>();
      for (int j = 0; j < this.F; j++) {
        // Set the j-th feature with a random trait
        int trait = r.nextInt(q);
        row.add(trait);
      }
      this.sigma.add(row);
    }
  }

  /**
   * This is the engine of the experiments, which reads
   * the parameters from standard input, creates the
   * population and starts the migration process
   *
   * @param args
   */
  public static void main(String[] args) {
    // Set seed for random numbers
    Random random = new Random();
```

```
    // Read parameters
    int N = Integer.valueOf(args[0]);
    int I = Integer.valueOf(args[1]);
    double pn = Float.valueOf(args[2]);
    double T = Float.valueOf(args[3]);
    int M = Integer.valueOf(args[4]);
    int q = Integer.valueOf(args[5]);
    int F = Integer.valueOf(args[6]);
    String fileName = args[7];

    /* Test case 1 */
    /*
     * int N = 10;//10000; int I = 5; double pn = 0.016;
     * double T = 0.3; int M = 1000; int q = 5; int F =
     * 15; String fileName = "bau";
     * System.out.println(fileName);
     */

    /* Test case 2 */
    /*
     * int N = 100;//10000; int I = 50; double pn =
     * 0.1;//0.016; double T = 0.1; int M = 1000; int q
     * = 20; int F = 100; String fileName = "bau";
     * System.out.println(fileName);
     */
    Population myPopulation = new Population(random, N,
        I, pn, T, M, q, F, fileName);
    /* Debug prints */
    /*
     * for (int node=0; node<N; node++) {
     * System.out.println("Node: " + node + " Ind: " +
     * myPopulation.nodeToInd.get(node)); }
     * System.out.print(myPopulation.G.toString());
     */
    myPopulation.startMigrations(random);
  }
}
```

References

[1] Robert Axelrod. The dissemination of culture: A model with local convergence and global polarization. *Journal of Conflict Resolution*, 41(2):203–226, 1997.

[2] C. Gracia-Lázaro, L. F. Lafuerza, L. M. Floría, and Y. Moreno. Residential segregation and cultural dissemination: An axelrod-schelling model. *Phys. Rev. E*, 80:046123, Oct 2009.

[3] Thomas C. Schelling. Dynamic models of segregation. *The Journal of Mathematical Sociology*, 1(2):143–186, 1971.