Model of isolation of communities

Justyna Ilczuk

January 2015

Contents

1 Introduction

Project was meant to recreate model of isolated communities described papers in bibliography

- create a model base
- create model evolution
- detect isolated clusters
- experiment with different amounts of species
- experiment with external bias

2 Implementing the model

First, few words about the model on the example of chain and lattice.

2.0.1 Chain

From an initially empty chain of length N we choose randomly a node and insert a random specie into it. Basic model has only two types of species, but it will be further expanded.

If a group of nodes of the same type doesn't have any unoccupied node as neighbour or doesn't lay on boarder, it's called an isolated cluster. Another definition is: nodes of the cluster cannot communicate with the rest of their population.

Examples are illustrated on figures 1 and 2.

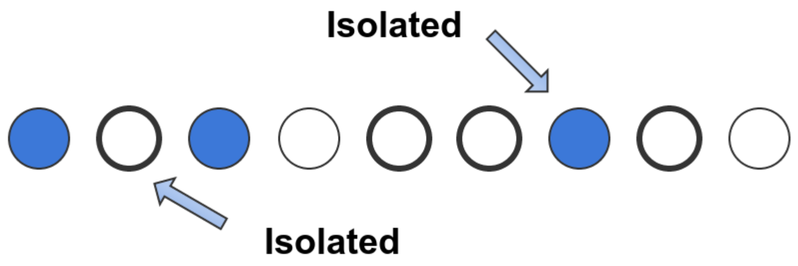

Figure 1: Chain Model - two species can isolate each other and create isolated nodes or clusters

Example of creating and filling up the chain:

```
makeChain[n_] := Table[0, {i, n}]
chain = makeChain[20]
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

n = 0;
emptyIndexes = makeEmptyIndexes[20];
chain = makeChain[20];
While[Length[emptyIndexes] > 0, x = RandomChoice[emptyIndexes];
  emptyIndexes = DeleteCases[emptyIndexes, x];
  chain[[x]] = RandomChoice[{-1, 1}]; n++;
  If[n == 5, Print[chain], n]];
Print[chain];

{0,0,1,-1,0,0,0,0,0,0,1,0,0,-1,0,1,0,0,0,0}
{-1,-1,1,-1,-1,-1,1,1,1,-1,1,1,-1,-1,-1,1,-1,-1,1,-1}
```

This example prints out two chains, one is filled only a little (it's a peek after only five iterations) and the latter is completely filled.

2.0.2 Lattice

Plain lattice was also easy to implement, as in the listing below:

```
    makeLattice[n_] := Table[Table[0, {i, n}], {j, n}]
makeEmptyIndexesLattice[maxLen_] :=
 Tuples[Table[i, {i, 1, maxLen}], 2]
```

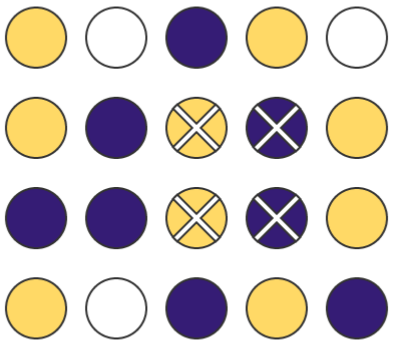

Figure 2: Lattice Model - isolated nodes are crossed. We use Von Neumann Neighbourhood

2.1 Finding clusters

How to count clusters in the general case? What kind of algorithm to use?

- able to count clusters of frozen chain/lattice
- able to count during model evolution
- fast and efficient
- extendable

Two algorithms for two different cases

- detection and counting during evolution
- detection and counting in frozen state

2.2 Algorithms design

Main Ideas behind the algorithm:

- breadth first search `http://en.wikipedia.org/wiki/Breadth-first_search`
- inserting a specie in empty space can turn clusters of nodes into isolated clusters if they contain a neighbour of new node

Start simple:

- chain model
- check it and analyze results of simulations
- make similar model for lattice

Papers about community isolation illustrate analytical predictions for model with simulations result.

- recreate figures
- compare curves and trends
- check if simulation agrees with prediction

First version of algorithms, which running on the chain, finds all the isolated clusters:

```
findIsolatedClusters[chain_] := {isolatedClusterCount = 0;
  isolatedCount = 0;
  howLong = Length[chain];
  startNodes = getChainIndexes[chain];
  visited = {};
   toVisit = {}; centers = {};
  While[Length[startNodes] > 0 , firstNode = Extract[startNodes, 1];
   startNodes = startNodes[[2 ;;]];
   toVisit = Append[toVisit, firstNode];
   currentType = chain[[firstNode]];
   currentCount = 0;
   (* used to count how many members are in cluster if it's a cluster *)

   fine = False;
   While[Length[toVisit] > 0, node = Extract[toVisit, 1];
    toVisit = toVisit[[2 ;;]];
    nodeType = chain[[node]];
    If[nodeType == 0, fine = True;, If[nodeType == currentType,
       currentCount ++; visited = Append[visited, node];
       startNodes = DeleteCases[startNodes, node];
```

```
          neighbourNodes = createNeighbours[node];
          neighbour1 = neighbourNodes[[1]];
          neighbour2 = neighbourNodes[[2]];
          If[inBoundaries[neighbour1, howLong],
           If[! MemberQ[visited, neighbour1],
            toVisit = Append[toVisit, neighbour1];], fine = True];
          If[inBoundaries[neighbour2, howLong],
           If[! MemberQ[visited, neighbour2],
            toVisit = Append[toVisit, neighbour2];], fine = True];];];
       ]; If[! fine,
           (*we found an isolated cluster *)
           isolatedClusterCount++; centers = Append[centers, firstNode];
           isolatedCount = isolatedCount + currentCount;]];
    {isolatedClusterCount, centers, isolatedCount}}
```

This function is very useful, but when we apply it over and over it does some unnecessary work. It furhter evolved into a function that checks only for changes introduced by the new node.

```
findIsolatedClusters[chain_, startNode_, chainSize_] := {
(* initialization *)
isolatedClusterCount = 0; isolatedCount = 0;
  howLong = Length[chain];
  startNodes =
   Join[{startNode},
    Select[createNeighbours[
      startNode], # >= 1 && # <= chainSize &]];
  visited = {};
   toVisit = {}; centers = {};
  While[Length[startNodes] > 0 , firstNode = Extract[startNodes, 1];
   startNodes = startNodes[[2 ;;]];
   toVisit = Append[toVisit, firstNode];
   currentType = chain[[firstNode]];
   currentCount = 0;
   (* used to count how many members are in cluster if it's a cluster *)

      fine = False;
   While[Length[toVisit] > 0, node = Extract[toVisit, 1];
    toVisit = toVisit[[2 ;;]];
    nodeType = chain[[node]];
    If[nodeType == 0, fine = True;, If[nodeType == currentType,
       currentCount ++; visited = Append[visited, node];
       startNodes = DeleteCases[startNodes, node];
       neighbours = createNeighbours[node];
       toVisit =
        Join[toVisit,
         Select[neighbours,
             inBoundaries[#, howLong] && ! MemberQ[visited, #] &]];
           If[
             Length[Select[neighbours, ! inBoundaries[#, howLong] &]] > 0,
             fine = True];];];
       ]; If[! fine, (* we found an isolated cluster *)
             isolatedClusterCount++; centers = Append[centers, firstNode];
             isolatedCount = isolatedCount + currentCount;]];
    {isolatedClusterCount, centers, isolatedCount}}
```

Some other functions used above

```
makeChain[n_] := Table[0, {i, n}]
makeEmptyIndexes[n_] :=  Table[i, {i, 1, n}]
createNeighbours[node_] := {node - 1, node + 1}
inBoundaries[node_, N_] := node >= 1 && node <= N
```

2.3 Simulating chain

Function used to simulating chain fills the chain in each iterations and check for new clusters.

```
getFromRandomChoice[arg_] := RandomChoice[Table[i, {i, 1, arg}]];

simulateChain[maxLength_, typeGenerator_: getFromRandomChoice,
   arg_: 2] :=
  Module[{maxLen = maxLength, isolatedCountArray,
     notIsolatedCountArray, emptyIndexes, chain, iter}, iter = 0;
   allIsolated = 0;
   isolatedCountArray = {};
   notIsolatedCountArray = {};
   emptyIndexes = makeEmptyIndexes[maxLen];
   chain = makeChain[maxLen];
   While[Length[emptyIndexes] > 0,
    iter++;
    (* fill the random empty place with random type *)

    x = RandomChoice[emptyIndexes];
    emptyIndexes = DeleteCases[emptyIndexes, x];
    chain[[x]] = typeGenerator[arg];

    clusters = findIsolatedClusters[chain, x, maxLength];
    isolated = clusters[[1, 3]];

    allIsolated += isolated;
    isolatedCountArray = Append[isolatedCountArray, allIsolated];
    notIsolatedCountArray =
     Append[notIsolatedCountArray, iter - allIsolated];];
  {notIsolatedCountArray, isolatedCountArray}]
```

2.4 Detecting clusters in lattice

Detecting clusters in lattice is very similar, but neighbourhood and boundaries are a bit different. However functions could be generalized.

```
createNeighboursLattice[
  node_] := {{node[[1]] - 1, node[[2]]}, {node[[1]] + 1,
   node[[2]]}, {node[[1]], node[[2]] - 1}, {node[[1]],
   node[[2]] + 1}}

inBoundariesLattice[node_, howLong_] :=
 node[[1]] >= 1 && node[[2]] >= 1 && node[[1]] <= howLong &&
  node[[2]] <= howLong

findIsolatedClustersOnLattice[chain_, startNode_,
  chainSize_] := {isolatedClusterCount = 0; isolatedCount = 0;
  howLong = Length[chain];
  startNodes =
   Join[{startNode},
    Select[createNeighboursLattice[startNode],
     inBoundariesLattice[#, chainSize] &]];
  visited = {};
   toVisit = {}; centers = {};
  While[Length[startNodes] > 0 , firstNode = Extract[startNodes, 1];
   startNodes = startNodes[[2 ;;]];
   toVisit = Append[toVisit, firstNode];
   currentType = Extract[chain, firstNode];
   currentCount = 0; (*
   used to count how many members are in cluster if it's a cluster *)

      fine = False;
   While[Length[toVisit] > 0, node = Extract[toVisit, 1];
    toVisit = toVisit[[2 ;;]];
    nodeType = Extract[chain,  node];
    If[nodeType == 0, fine = True;, If[nodeType == currentType,
       currentCount ++; visited = Append[visited, node];
       startNodes = DeleteCases[startNodes, node];
       neighbours = createNeighboursLattice[node];
       toVisit =
        Join[toVisit,
         Select[neighbours,
          inBoundariesLattice[#, howLong] && ! MemberQ[visited, #] &]];
       If[
        Length[Select[
            neighbours, ! inBoundariesLattice[#, howLong] &]] > 0,
         fine = True];
        ];];
    ]; If[! fine, (* we found an isolated cluster *)
    isolatedClusterCount++; centers = Append[centers, firstNode];
    isolatedCount = isolatedCount + currentCount;]];
  {isolatedClusterCount, centers, isolatedCount}}
```

2.5 Simulating the lattice

Below I present the listing for simulating the lattice behaviour:

```
makeLattice[n_] := Table[Table[0, {i, n}], {j, n}]
makeEmptyIndexesLattice[maxLen_] := Tuples[Table[i, {i, 1, maxLen}], 2]
simulateLattice[maxLength_, typeGenerator_: getFromRandomChoice,
  arg_: 2] :=
 Module[{maxLen = maxLength, isolatedCountArray,
   notIsolatedCountArray, emptyIndexes, chain, iter}, iter = 0;
  allIsolated = 0;
  isolatedCountArray = {};
  notIsolatedCountArray = {};
  emptyIndexes = makeEmptyIndexesLattice[maxLen];
  chain = makeLattice[maxLen];
  While[Length[emptyIndexes] > 0,
   iter++;
   (* fill the random empty place with random type *)

   x = RandomChoice[emptyIndexes];
   emptyIndexes = DeleteCases[emptyIndexes, x];
   newVal = typeGenerator[arg];
   chain[[x[[1]], x[[2]]]] = newVal;
   clusters = findIsolatedClustersOnLattice[chain, x, maxLength];
   isolated = clusters[[1, 3]];
   allIsolated += isolated;
   isolatedCountArray = Append[isolatedCountArray, allIsolated];
   notIsolatedCountArray =
    Append[notIsolatedCountArray, iter - allIsolated];];
  {notIsolatedCountArray, isolatedCountArray}]
```

3 Results

3.1 Key interests

My key Interests were

- trends of increasing number of isolated nodes, how curves look like

- t_c critical time when appears first isolated cluster
- observing the impact of bias, bias is the difference between probabilities of creating new node of the given specie (more about bias in bibliography)

More about models, what are exact solutions to the t_c changing with time and bias can be found in [2], [1].

3.2 Visualization

Next I want to present the visualizations of the results I got from simulations focused on different aspects of the model.

3.2.1 Plots of numbers of free vs isolated nodes

Code below generates figure 3

```
Timing[maxLen = 1000;
 countsArrayAll = {};
 countsArray = simulateChain[maxLen];
 countsArrayAll = Append[countsArrayAll, countsArray];
 countsArray = Mean[countsArrayAll];]

 {0.176198, Null}

ListLogLogPlot[countsArray, AxesLabel -> {"t", "Z, (t - Z)"}]
```

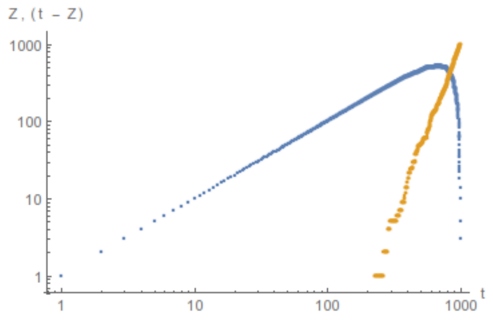

Figure 3: Number of isolated nodes (dashed lines) and not isolated nodes of each specie (solid lines) versus time for chain size N = 1000

Next is lattice at fig 4.

```
Timing[maxLen = 20;
 countsArrayAll = {};
 countsArray = simulateLattice[maxLen];
 countsArrayAll = Append[countsArrayAll, countsArray];
 countsArray = Mean[countsArrayAll];]
```

```
ListLogLogPlot[countsArray, AxesLabel -> {"t", "Z, (t - Z)"}]
```

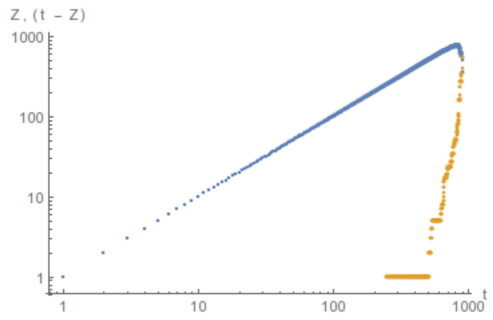

Figure 4: Number of isolated nodes (dashed lines) and not isolated nodes of each specie (solid lines) versus time for lattice size NxN, N = 30

To see the relation between the size of the chain or lattice and trends of appearing of the isolated clusters we need to run simulations for different sizes and illustrate them on the same figure.

Listings generating this relation for chain, result is on fig **??**.

```
getZAndRest[maxLen_, types_: 2] := Module[{allZ, allRest, i},
  allZ = {};
  allRest = {};
  For[i = 0, i < 10, i++, countsArray = simulateChain[maxLen, types];
   Z = countsArray[[2]];
   allZ = Append[allZ, Z];
   allRest = Append[allRest, countsArray[[1]]];
   ];
  meanZ = Mean[allZ];
  meanRest = Mean[allRest];
  {meanZ, meanRest} ]

{meanZ100, meanRest100} = getZAndRest[100];
{meanZ200, meanRest200} = getZAndRest[200];
{meanZ400, meanRest400} = getZAndRest[400];
{meanZ800, meanRest800} = getZAndRest[800];

Show[ListLogLogPlot[{meanZ100, meanZ200, meanZ400, meanZ800},
  PlotStyle -> {Dashed, Gray}, Joined -> True],
 ListLogLogPlot[{meanRest100, meanRest200, meanRest400, meanRest800},
  PlotStyle -> Thick, Joined -> True] ,
 AxesLabel -> {HoldForm[t], HoldForm["Z, t - Z"]},
 PlotLabel -> HoldForm[Free vs Isolated]]
```

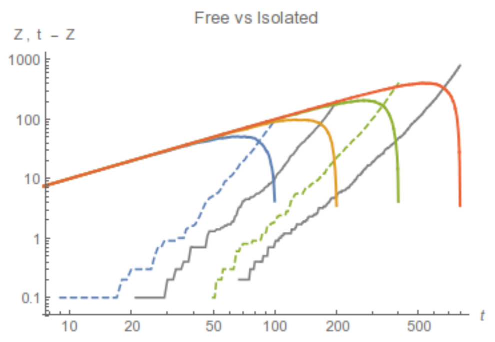

Figure 5: Number of isolated nodes (dashed lines) and not isolated nodes of each specie (solid lines) versus time for different chain sizes N, sizes are from left to right: 100, 200, 400, 800

3.2.2 Critical times t_c

Generate the critical value of time t_c versus size of the chain size N. Fit the $x^{\frac{2/3}{}}$. Result presented on fig 6.

```
allTCs = {};
allMeanTCs = {};
maxLens = {};
For[g = 100, g < 2000, g += 500,
  For[i = 0, i < 40, i++, countsArray = simulateChain[g, 2];
   Z = countsArray[[2]];
   Clear[x];
   tC = Length@First@Split[Z, #1 == 0 &];
   allTCs = Append[allTCs, tC];
   ];
  maxLens = Append[maxLens, g];
  allMeanTCs = Append[allMeanTCs, Mean[allTCs]];
  allTCs = {};
  ];
data = Transpose[{maxLens, allMeanTCs}];
fit = Fit[data, { x ^ (2 / 3)}, x];
Print[fit];
Show[ListLogLogPlot[data], LogLogPlot[fit, {x, 1, 4000}],
 AxesLabel -> {"N", "t_c"}]
```

Previous result can be easily extended to use more types of species than two. Below I plot the values of t_c for species numbers: 2, 4, 8.

Results are shown on graph 7.

```
allTCs = {};
allMeanTCs = {};
allMeanTCsPerM = {};
```

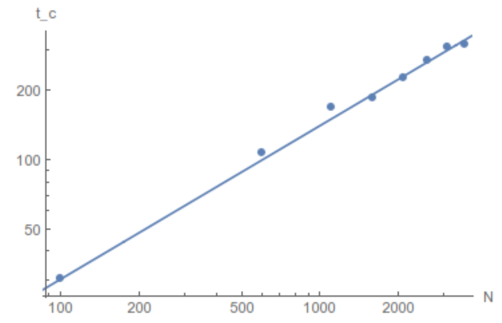

Figure 6: The critical value of time t_c versus the chain size N, m=2. Values are from numerical simulations (Mean of 40 for each size) and solid line is fitted curve (2/3)

```
maxLens = {};
For[m = 2, m < 16, m *= 2,
  For[g = 100, g < 500, g += 100,
   For[i = 0, i < 40, i++,
    countsArray = simulateChain[g, getFromRandomChoice, m];
    Z = countsArray[[2]];
            Clear[x];
    tC = Length@First@Split[Z, #1 == 0 &];
    allTCs = Append[allTCs, tC];
    ];
   maxLens = Append[maxLens, g];
   allMeanTCs = Append[allMeanTCs, Mean[allTCs]];
   allTCs = {};
   ];
  data = Transpose[{maxLens, allMeanTCs}];
  allMeanTCsPerM = Append[allMeanTCsPerM, allMeanTCs]; allMeanTCs = {};
  maxLens = {};
  ];

Print[allMeanTCsPerM ]; ListLogLogPlot[allMeanTCsPerM,
 AxesLabel -> {"N", "t_c"}, PlotMarkers -> {Automatic, 24} ]
```

3.2.3 Simple simulations with bias

Bias is simulated by disturbing equal possiblities to choose a specie, introducing ϵ. I introduce new type generator:

```
typeWithBias[bias_] :=
 RandomVariate[
  EmpiricalDistribution[{1 /2 + bias, 1/ 2 - bias} -> {1, 2}]]
```

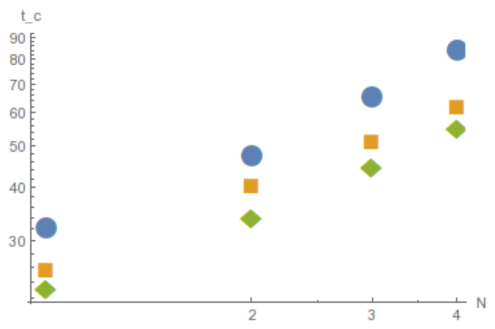

Figure 7: The critical value of time t_c versus different numbers of species m: 2, 4, 8 from top to bottom. Values are from numerical simulations (Mean of 40 simulations for each size)

So chain simulation function with bias can be now written as

```
simulateChainWithBias[maxLength_, bias_] :=
    simulateChain[maxLen, typeWithBias, bias]

maxLen = 2000;
biases = Table[simulateChainWithBias[maxLen, i / 20.0][[2]], {i, 9}];
ListLogLogPlot[biases, AxesLabel -> {"t", "Z, (t - Z)"}]
```

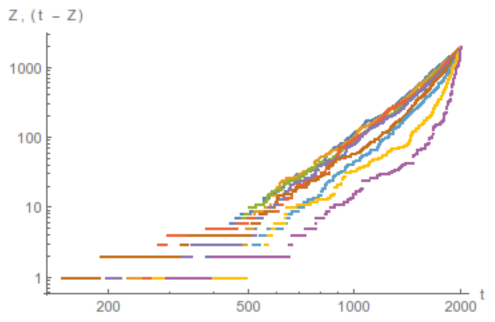

Figure 8: Number of blocked nodes for bias values, ϵ; 1/20, 2/20 ... 9 / 20

Similar can be done with the 2D-lattice:

```
simulateLatticeWithBias[maxLength_, bias_] :=
    simulateLattice[maxLength, typeWithBias, bias]

maxLen = 20;
biases = Table[
   simulateLatticeWithBias[maxLen, i / 20.0][[2]], {i, 0, 5}];
ListLogLogPlot[biases, AxesLabel -> {"t", "Z, (t - Z)"}]
```

4 Summary

My work doesn't exhaust what could be done with those models.

Definitely work on bias could be extended, but it wasn't meant to be exhaustive.

Having more time I would make better comparisons to theoretical results described in [2] and [1]. Still, implementing those algorithms gave me a lot of insight, how to simulate this kind of models and how do they behave.

References

[1] Grzegorz Siudem Julian Sienkiewicz and Janusz A. Holyst. External bias in the model of isolation of communities. 2010.

[2] Julian Sienkiewicz and Janusz A. Holyst. Nonequilibrium phase transition due to communities isolation. 2009.