**基础模型**

假设居住地为​的方格棋盘，内共有​个个居民，均匀地分为不同的两类。

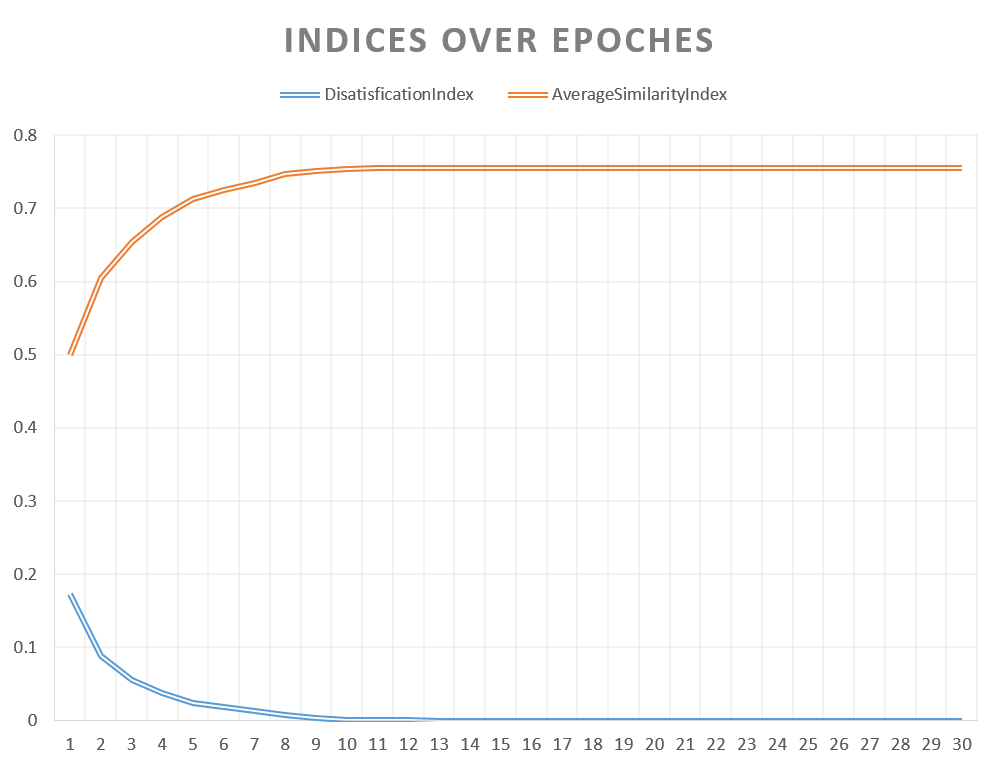
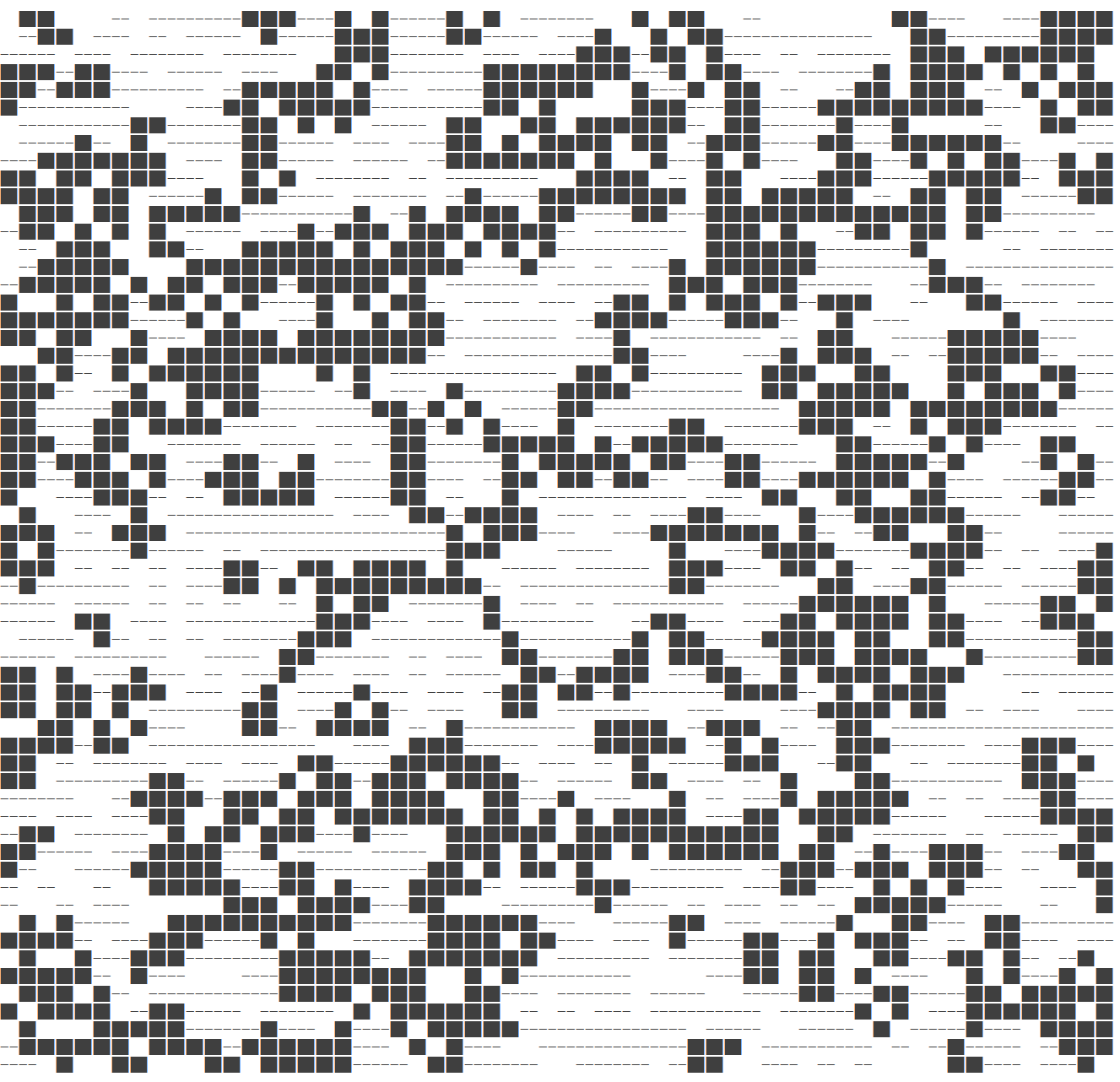
模型的一次迭代中，每个居民根据其相邻的八个格子中居民的分布，能够给出一个相似度（同类型邻居数量与邻居总数之比），若相似度低于一定的阈值，称居民不满意当前环境，从而在本次迭代中随机迁移至其他住处。当每个节点相似度均低于给定阈值时，模型趋于收敛，停止迭代。

谢林模型告诉我们：

* 微观动机不等同于宏观上表现：若移动倾向阈值过高，虽然居民均有很强的离开与自身不同单位的倾向，但由于大部分的居民都难以稳定在一个位置，从而导致宏观上的混杂，而非期望中的隔离。
* 在宏观层次看到的实际可能不是微观层面正在发生的事：在阈值合适的情况下，宏观上的居民分布会在几次迭代后快速趋于稳定，产生隔离，而微观上，居民搬家的位置是随机洗牌的，并没有刻意向理想位置迁移。

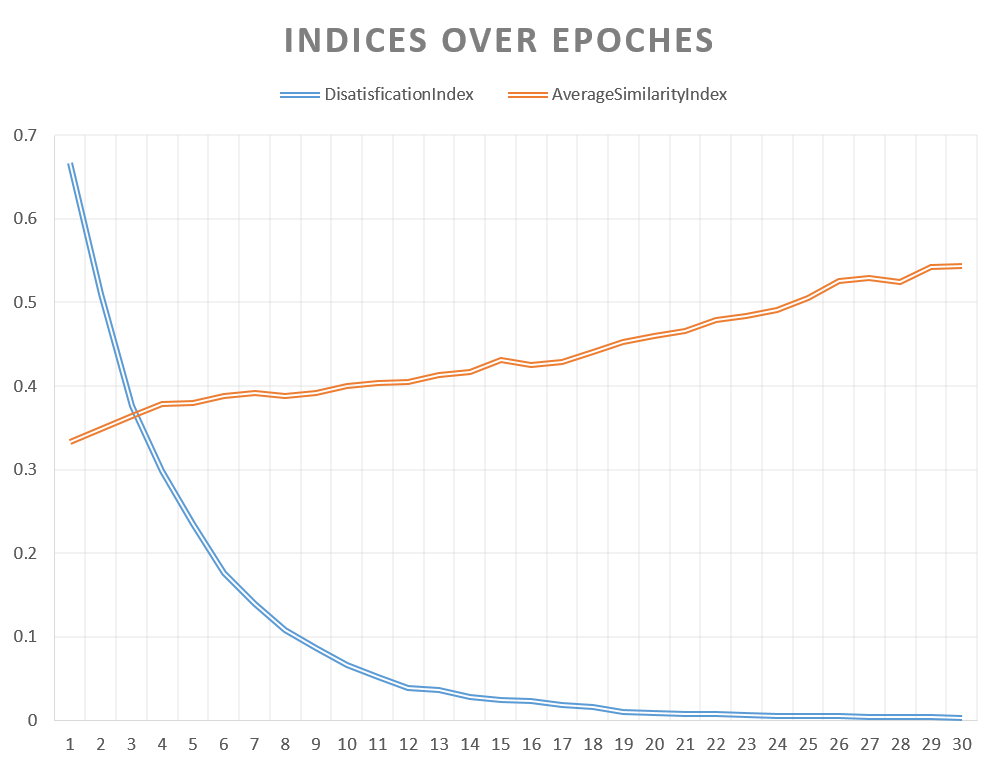
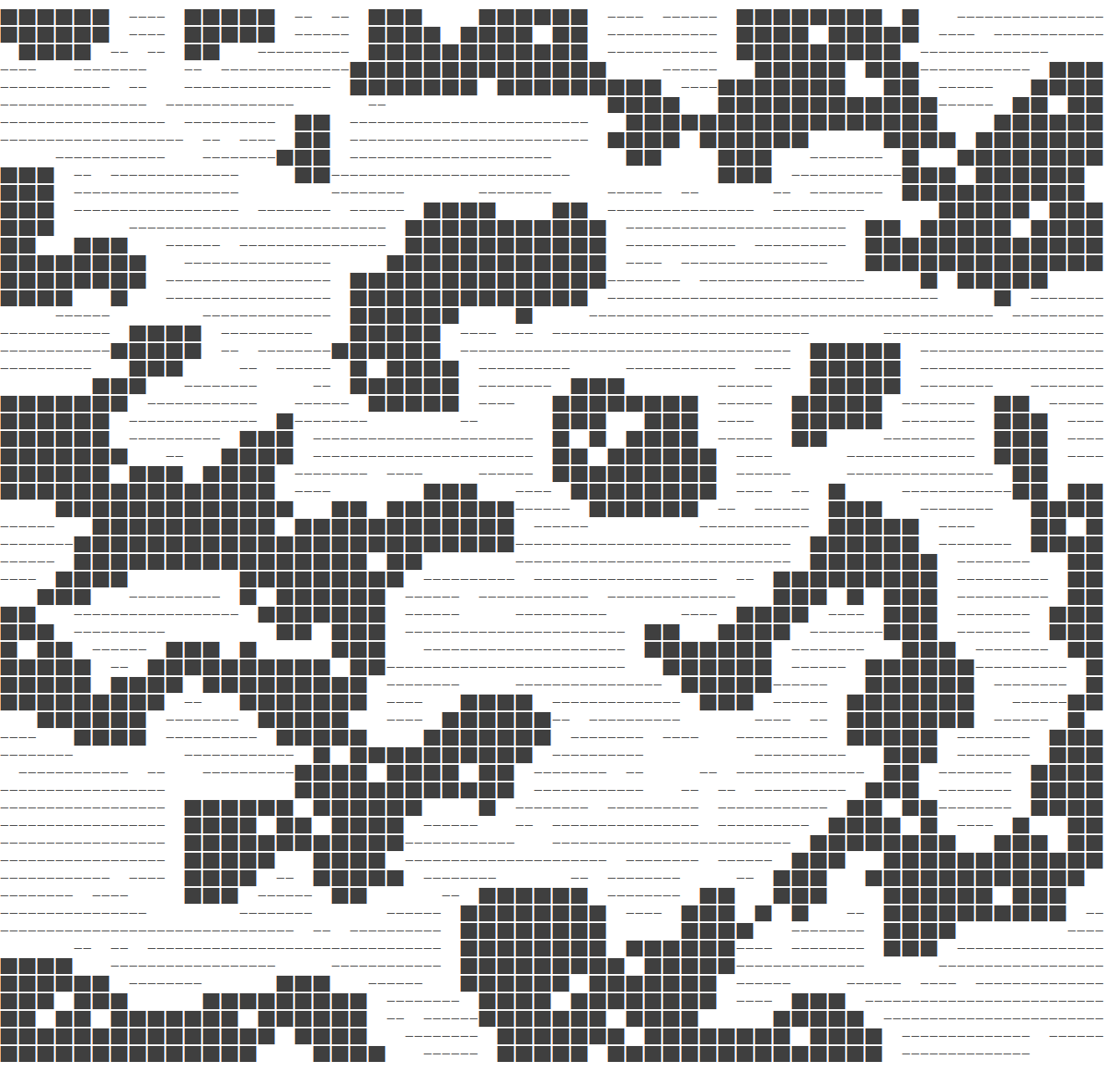
当相似度阈值分别为的情况下，本模型的蒙特卡罗模拟结果如下：

### 相似度阈值0.3



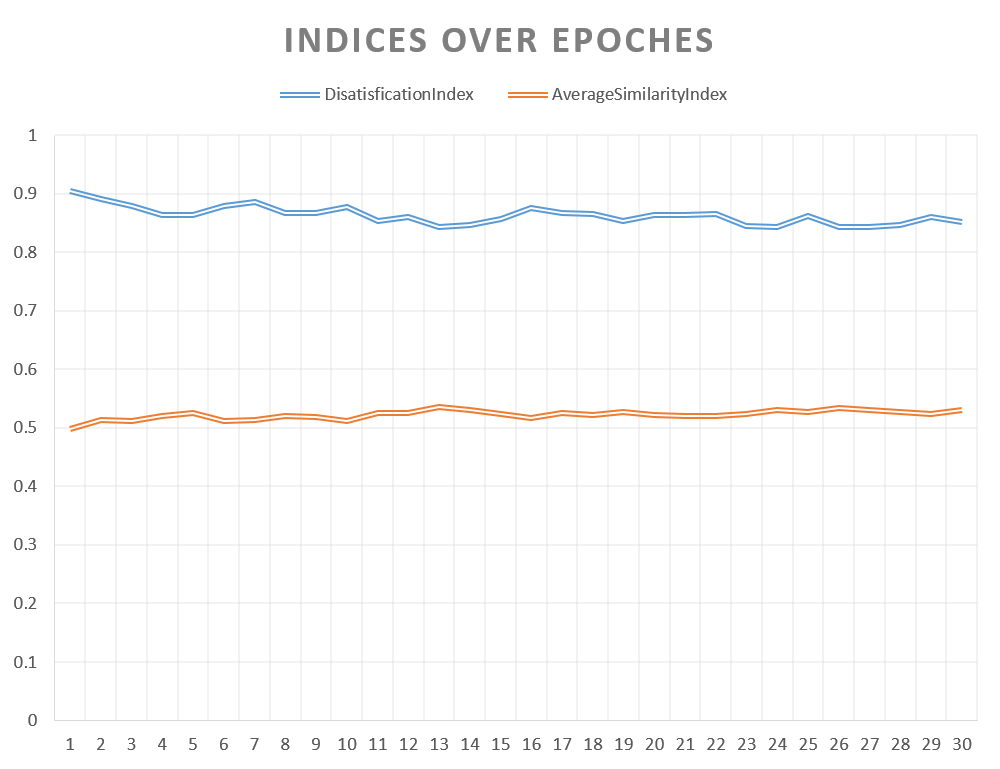
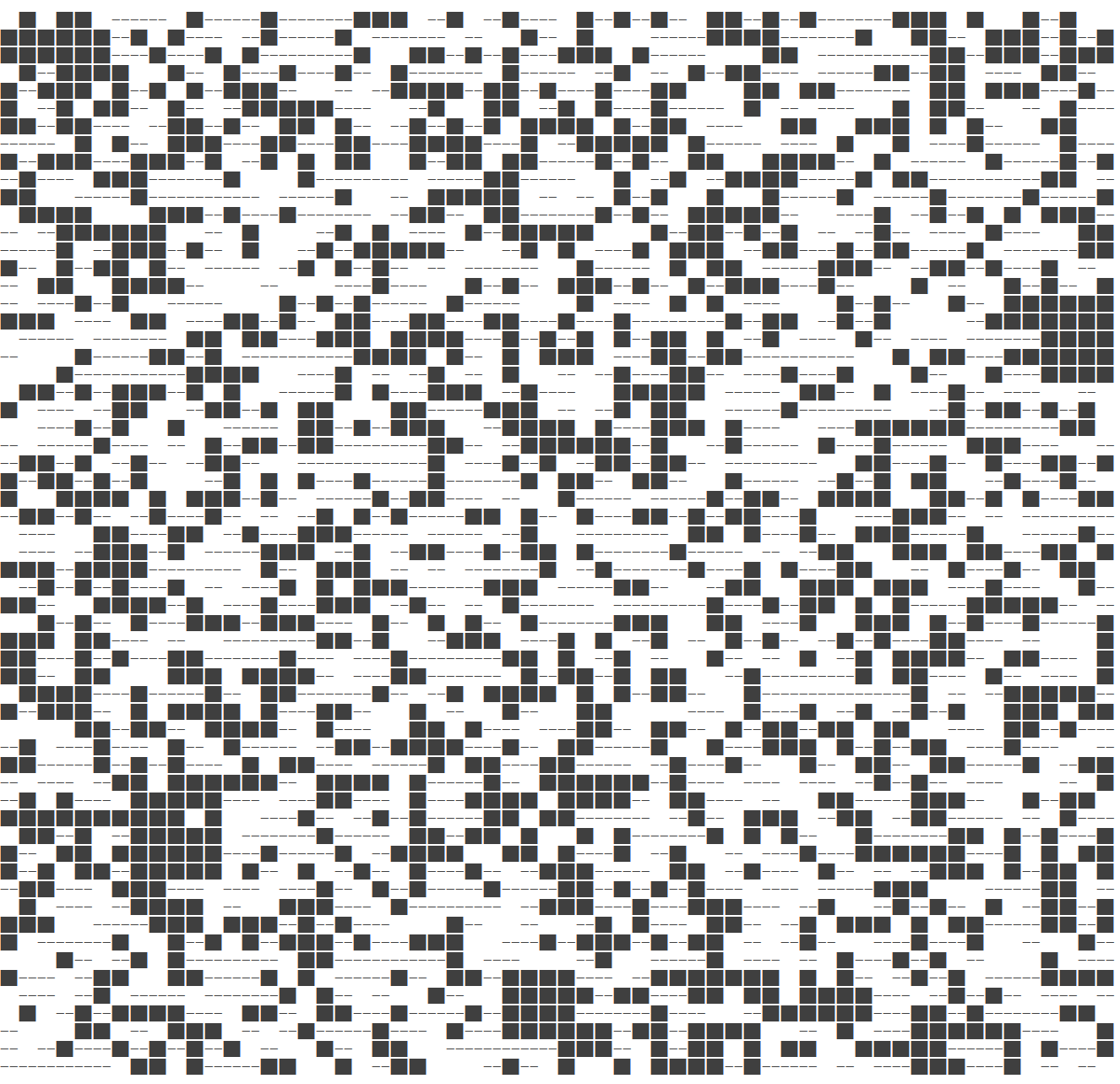
当相似度阈值较低时，由于居民需求较低，在形成明显大块的隔离前整个模型便提早收敛，趋于稳定。

### 相似度阈值0.6



当相似度阈值为时，随着迭代，模型快速收敛并良好地表现了两类不同居民的隔离现象。

### 相似度阈值0.8

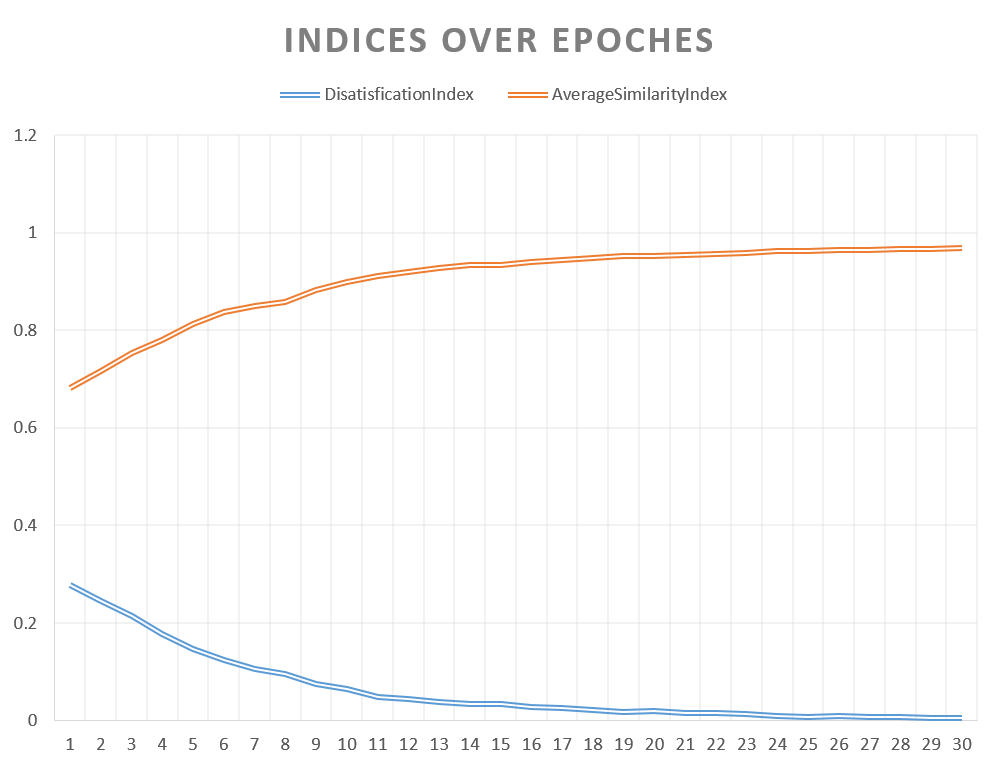
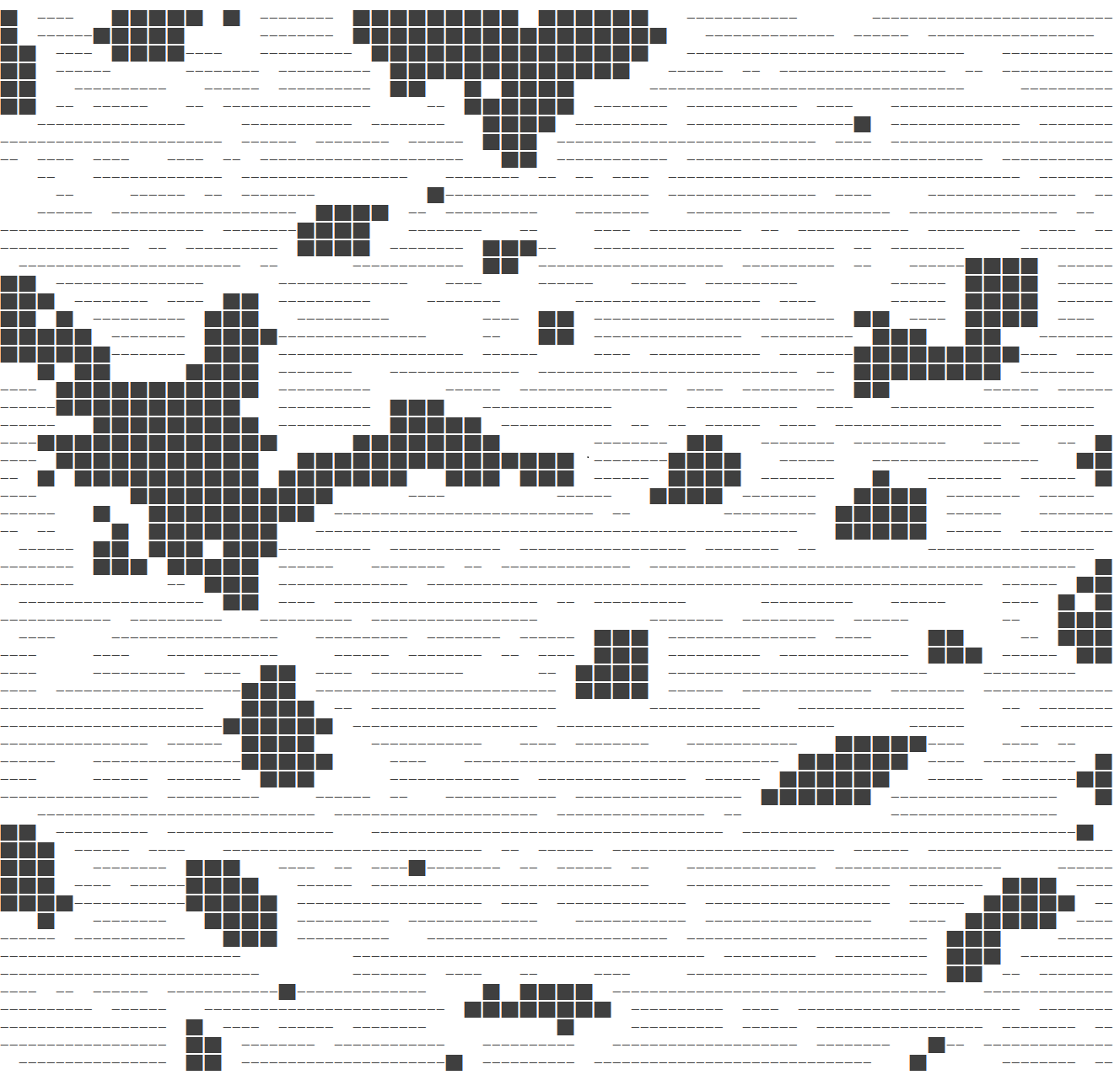


当相似度阈值为时，由于居民期望过高，导致模型持续迭代而难以进入稳定状态，居民分布趋于随机，与其随机迁移的行为一致。

**增强模型**

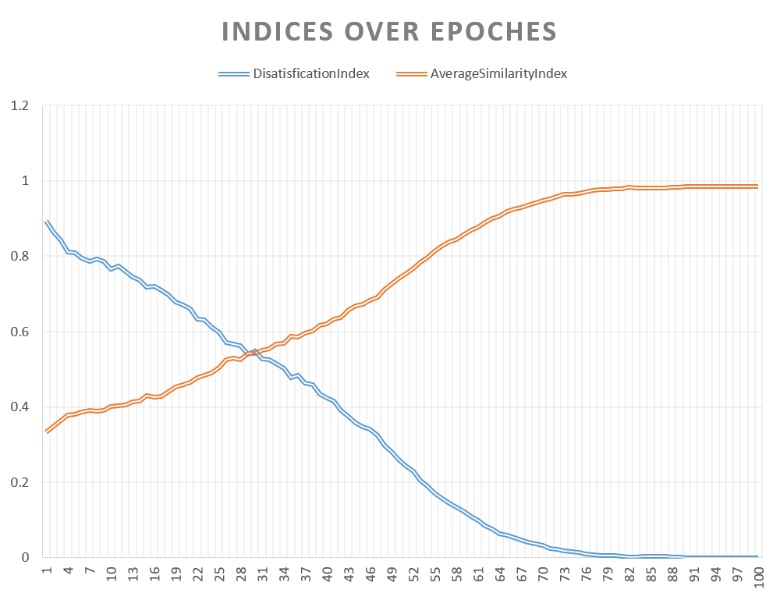
由于上述模型基于居民严格分为两类且数量大致相同的假设，而现实状况则往往更为严格，下面是一些随意的增强参数，来检验更严格状况下本模型的结果。

### 相似度阈值0.6；居民比2：8



在一类居民数量较少的情况下，模型趋于部分居民集中在少数大聚集区，而部分居民则分散聚集在各个小聚集区的情况。

### 相似度阈值0.6；居民比1：1：1

在三类居民的情况下，模型的收敛速度明显放缓，但最终还是趋于形成三类居民的相互隔离的情况。

## 模型实现

如下代码能够在Visual Studio 2017中打开language standard c++17后通过编译。

#include <vector>

#include <cassert>

#include <optional>

#include <random>

#include <algorithm>

#include <fstream>

#include <memory>

struct Coord

{

int x;

int y;

};

class ResidentType

{

public:

ResidentType(float weight, const std::string& symbol, const std::string& name)

: weight\_(weight), symbol\_(symbol), name\_(name) { }

float Weight() const { return weight\_; }

const auto& Symbol() const { return symbol\_; }

const auto& Name() const { return name\_; }

private:

float weight\_;

std::string symbol\_;

std::string name\_;

};

class Board

{

public:

int Width() { return width\_; }

int Height() { return height\_; }

int Population() { return population\_; }

float Threshold() { return threshold\_; }

float DisatisficationIndex() { return disatisfication\_index\_; }

float AverageSimilarityIndex() { return averge\_similarity\_index\_; }

void Initialize(int width, int height, int population, float threshold,

const std::vector<ResidentType>& residents)

{

assert(!residents.empty());

assert(width > 0 && height > 0 && population > 0);

assert(population < width \* height);

assert(threshold > 0 && threshold < 1);

// initialize parameters

//

residents\_ = residents;

width\_ = width;

height\_ = height;

population\_ = population;

threshold\_ = threshold;

// initialize board

//

RefreshBoard();

// calculate indices and next board

//

Iterate();

}

// discards current board and calculate the next

void Iterate()

{

// duplicate the board

//

cur\_board\_ = next\_board\_;

// find empty blocks and candidates to move

//

std::vector<Coord> unstable\_blocks;

auto moving\_counter = 0.f;

auto similarity\_ind\_sum = 0.f;

for (int j = 0; j < Height(); ++j)

{

for (int i = 0; i < Width(); ++i)

{

auto pos = Coord{ i, j };

auto similarity = CalcSimilarityIndex(pos);

if (similarity)

{

// residented

similarity\_ind\_sum += \*similarity;

if (\*similarity < threshold\_)

{

// willing to move

moving\_counter += 1;

unstable\_blocks.push\_back(pos);

}

}

else

{

// not residented

unstable\_blocks.push\_back(pos);

}

}

}

disatisfication\_index\_ = moving\_counter / population\_;

averge\_similarity\_index\_ = similarity\_ind\_sum / population\_;

// perform moving

//

// generate some randome permutation

std::vector<int> perm;

std::generate\_n(

std::back\_inserter(perm), unstable\_blocks.size(),

[i = 0]() mutable { return i++; }

);

std::shuffle(perm.begin(), perm.end(), rng\_);

// ordinals in perm denotes index of moving blocks

for (int i = 0; i < perm.size(); ++i)

{

auto dest\_pos = unstable\_blocks[i];

auto src\_pos = unstable\_blocks[perm[i]];

next\_board\_[CoordToOffset(dest\_pos)]

= cur\_board\_[CoordToOffset(src\_pos)];

}

}

void PrintBoard(const char\* placeholder = " ")

{

for (int j = 0; j < Height(); ++j)

{

for (int i = 0; i < Width(); ++i)

{

auto res = cur\_board\_[CoordToOffset({ i, j })];

printf(res ? res->Symbol().c\_str() : placeholder);

}

putchar('\n');

}

}

private:

void RefreshBoard()

{

// resize board

//

next\_board\_.resize(ActualWidth() \* ActualHeight());

std::fill(next\_board\_.begin(), next\_board\_.end(), nullptr);

cur\_board\_.resize(ActualWidth() \* ActualHeight());

std::fill(cur\_board\_.begin(), cur\_board\_.end(), nullptr);

// initialize next board in random

//

std::vector<bool> occupied(width\_\*height\_, false);

std::fill\_n(occupied.begin(), population\_, true);

std::shuffle(occupied.begin(), occupied.end(), rng\_);

auto res\_weight = std::vector<float>{};

for (const auto& res : residents\_)

res\_weight.push\_back(res.Weight());

std::discrete\_distribution<int> dis{ res\_weight.begin(), res\_weight.end() };

for (int j = 0; j < Height(); ++j)

{

for (int i = 0; i < Width(); ++i)

{

if (occupied[j\*width\_ + i])

next\_board\_[CoordToOffset({ i, j })] = &residents\_[dis(rng\_)];

}

}

}

std::optional<float> CalcSimilarityIndex(Coord pos)

{

static constexpr Coord kNeighborCoordDelta[] = {

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

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

};

auto res = next\_board\_[CoordToOffset(pos)];

// no resident living at the position

if (!res) return std::nullopt;

int same = 0, diff = 0;

for (auto delta : kNeighborCoordDelta)

{

auto neighbor\_pos = Coord{ pos.x + delta.x, pos.y + delta.y };

auto neighbor\_res = next\_board\_[CoordToOffset(neighbor\_pos)];

if (neighbor\_res)

{

if (neighbor\_res == res)

same += 1;

else

diff += 1;

}

}

// no neighbor

if (same + diff == 0)

return 1.f;

else

return static\_cast<float>(same) / (same + diff);

}

int ActualWidth() { return width\_ + 2; }

int ActualHeight() { return height\_ + 2; }

int CoordToOffset(Coord pos)

{

auto actual\_pos = Coord{ pos.x + 1, pos.y + 1 };

return actual\_pos.y \* ActualWidth() + actual\_pos.x;

}

// TODO: fix insufficient entropy

std::mt19937 rng\_{ std::random\_device{}() };

std::vector<ResidentType> residents\_;

int width\_ = 0;

int height\_ = 0;

int population\_ = 0;

float threshold\_ = 0.f;

float disatisfication\_index\_;

float averge\_similarity\_index\_;

std::vector<ResidentType\*> next\_board\_;

std::vector<ResidentType\*> cur\_board\_;

};

int main()

{

using namespace std;

std::vector<ResidentType> res\_vec = {

{ .1f, "█", "rich" },

{ .1f, "||", "midclass" },

{ .1f, "--", "poor" },

};

Board board;

board.Initialize(60, 60, 2800, 0.6, res\_vec);

for (const auto& res : res\_vec)

printf("%s DENOTES %s with weight of %f\n",

res.Symbol().c\_str(), res.Name().c\_str(), res.Weight());

for (int i = 0; i < 30; ++i)

board.Iterate();

board.PrintBoard();

system("pause");

}