基础模型

假设居住地为60×60的方格棋盘，内共有个2800个居民，均匀地分为不同的两类。

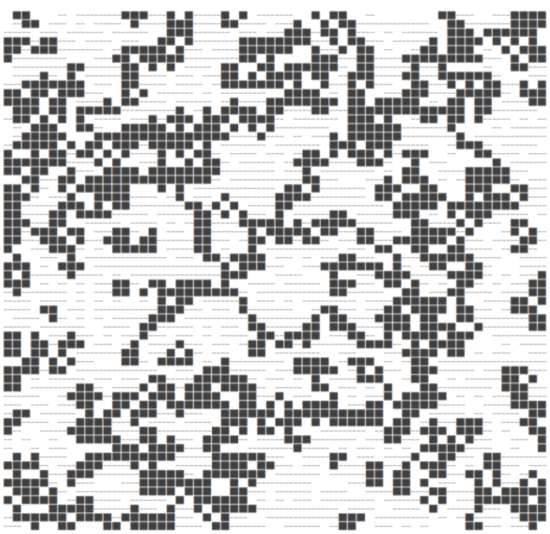
模型的一次迭代中，每个居民根据其相邻的八个格子中居民的分布，能够给出一个相似度（同类型邻居数量与邻居总数之比），若相似度低于一定的阈值，称居民不满意当前环境，从而在本次迭代中随机迁移至其他住处。当每个节点相似度均低于给定阈值时，模型趋于收敛，停止迭代。

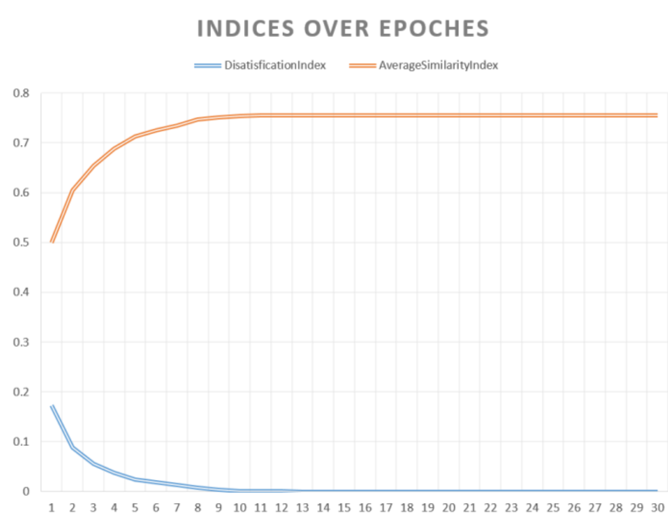
谢林模型告诉我们：

- 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$0.3, 0.6, 0.8$的情况下，本模型的蒙特卡罗模拟结果如下：

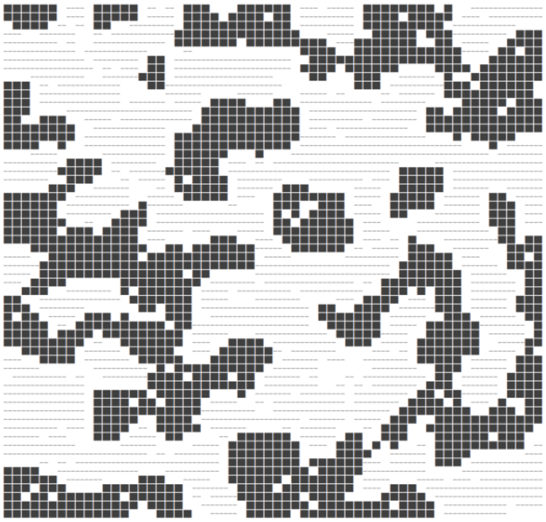
相似度阈值 0.3

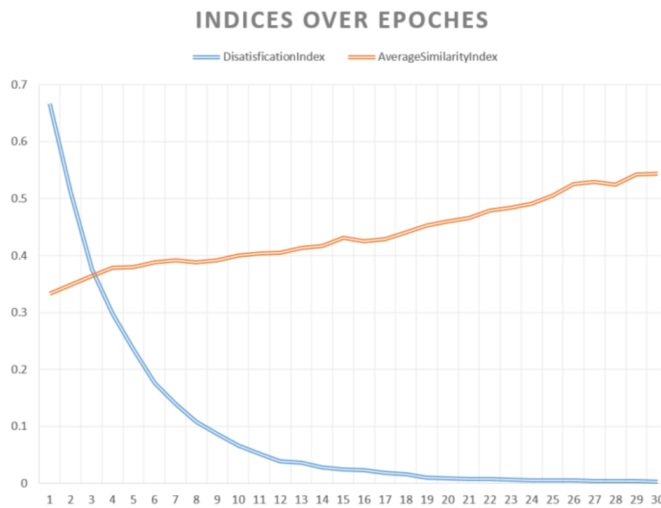




当相似度阈值较低时，由于居民需求较低，在形成明显大块的隔离前整个模型便提早收敛，趋于稳定。

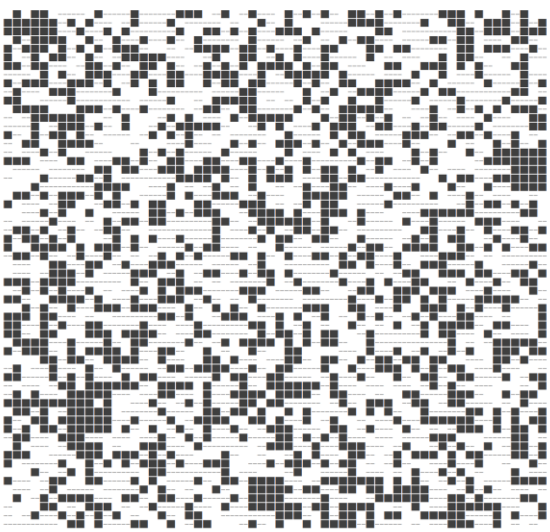
相似度阈值 0.6

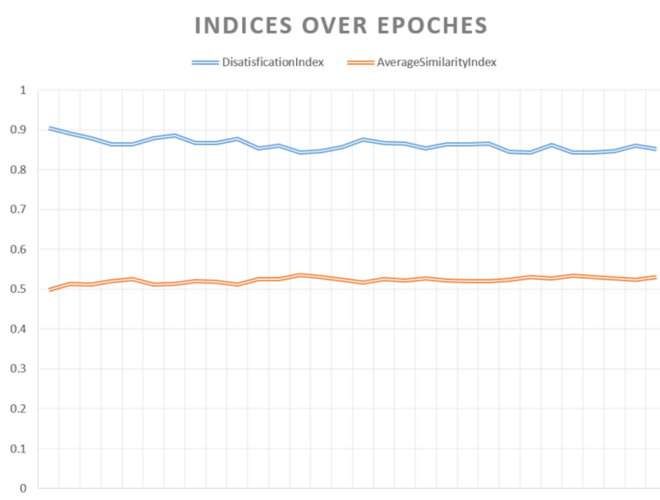




当相似度阈值为0.6时，随着迭代，模型快速收敛并良好地表现了两类不同居民的隔离现象。

相似度阈值 0.8



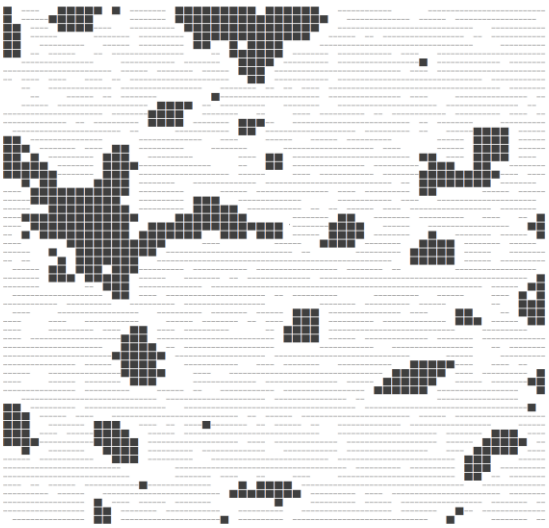


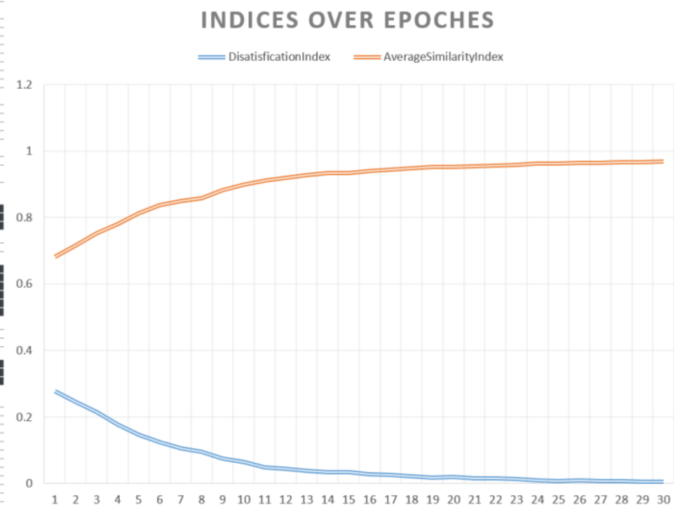
当相似度阈值为0.8时，由于居民期望过高，导致模型持续迭代而难以进入稳定状态，居民分布趋于随机，与其随机迁移的行为一致。

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。

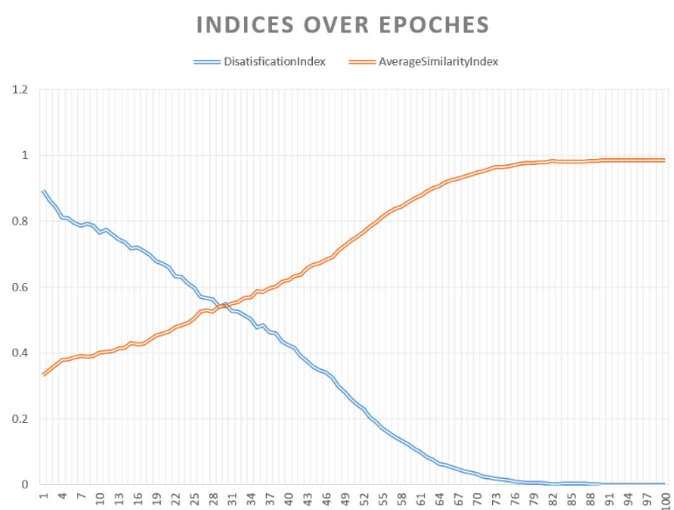
相似度阈值 0.6；居民比 2：8





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");
}
```