ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Московский институт электроники и математики им. А.Н. Тихонова

Малютин Александр Альбертович

МОДЕЛИРОВАНИЕ ПРОСТРАНСТВЕННЫХ ЭВОЛЮЦИОННЫХ ИГР

Междисциплинарная курсовая работа
студента образовательной программы
«Прикладная математика»
группы БПМ-152

Руководитель
ученая степень, звание:

_____
Подпись студента
А.А. Малютин

_____
Е.А. Буровский,
доцент

Москва 2018 г.

Аннотация

Я изучаю динамику модели пространственных эволюционных игр, основанную на дилемме заключённого. Особенности игры заключаются в нескольких резких переходов между режимами, которые контролируются параметром взаимодействия между игроками, и нетривиальном изменении состояния игрового поля. Я исследую критические свойства полученных переходов и геометрические свойства возникающих структур игроков.

Abstract

I study the dynamics of a spatial evolutionary game based on cellular automata and use the prisoner’s dilemma as rule for interaction. The game features a series of sharp transitions between several regimes, controlled by the interaction parameter between players, and non-trivial geometric rearrangements of the game field.

Оглавление

1. Введение

Моё исследование основывается на результатах полученных и описанных М.А.Новаком и Р.М.Мэйем в статье «Evolutionary games and spatial chaos» [1]. В работе описана сама модель пространственных эволюционных игр и описаны проведённые статистические исследования.

Модель описывает поведение двух видов игроков, расположенных в дискретном пространстве, на квадратной решётке. Первый тип – кооператоры(cooperator), второй – дефекторы(defector). Каждый игрок может взаимодействовать с восьмью ближайшими соседями. Взаимодействие описывается дилеммой заключённого: каждый игрок играет со своими соседями и подсчитывает свой выигрыш. После того, как все игроки проведут игру, исходя из своего полученного выигрыша, игроки выбирают новую стратегию следующим образом: игрок сравнивает выигрыши своих соседей и принимает стратегию того, чей выигрыш максимальный. Таким образом мы получаем дискретную динамическую модель, эволюционирующую со временем.

2. Теоретическая часть

Для того, чтобы разобраться как происходит игра, забудем про поле игроков, и посмотри на случай, когда у нас всего два игрока. Их взаимодействие описывается дилеммой заключённого и может быть представлено в виде матрицы:

	C	D
C	1	0
D	b	0

где C – кооператор, D – дефектор, а на пересечениях строк со столбцами стоят значения выигрыша, b – специальный параметр, влияющий на поведение системы, $1<b<3$.

Во избежание усложнения модели, мы можем допустить, что второй столбец равен нулю. Параметр b описывает выигрыш дефектора, при игре с кооператором.

Теперь играя друг с другом, игроки считают свой выигрыш, исходя из своей стратегии и стратегии своего оппонента. После этого они выбирают новую стратегию, самую выгодную на этом шаге: игрок смотрит на выигрыши своих оппонентов и берёт стратегию того, чей выигрыш максимальный.

Смоделируем ситуацию: играют два кооператора, после взаимодействия они получают выигрыш равный 1 каждый. Далее, выбор стратегии. Здесь всё тривиально, ничего не изменится, так как стратегия всего одна. Теперь допустим, что у нас кооператор и дефектор, теперь выигрыши составят 0 и b соответственно. И на этом шаге кооператор поменяет стратегию на дефектора, так как она более выгодная.

Вернёмся к полю игроков: теперь каждый игрок может взаимодействовать с восьмью ближайшими соседями и сам с собой и выигрыш считается суммой всех выигрышей попарных игр. Система стартует из некоторого начального состояния, поле засеивается случайным образом кооператорами, с вероятностью 21%, и дефекторами, и при этом мы получаем непредсказуемое поведение системы.

Динамика системы зависит от значения параметра b. Из дискретности

природы возможных выигрышей следует, что существуют только дискретные точки перехода параметра b, которые влияют на динамику системы. Для $1<b<3$ эти переходы происходят в точках 9/8, 8/7, 7/6, 6/5, 5/4, 9/7, 8/6, 7/5, 6/4, 8/5, 5/3, 7/4, 9/5, 2, 9/4, 7/3, 5/2, 8/3. Эти значения прямо вытекают из структуры связей между игроками (рис. 1).

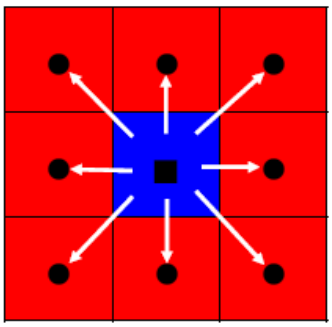

Рис. 1: Взаимодействие игроков на прямоугольной решётке.

Самой интересной точкой в системе является значение $b = 9/5$. В промежутке от 7/4 до 9/5, в системе существует несколько больших кластеров кооператоров, и «нити» из дефекторов, которые осциллируют в некоторых точках, но не меняя общего состояния, то есть система приходит в стационарное состояние (рис. 2).

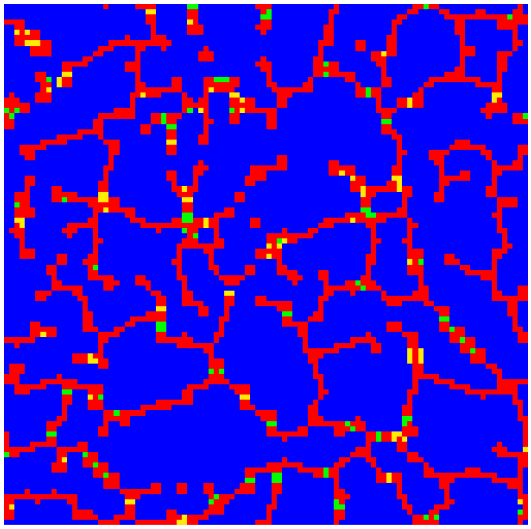

Рис. 2: Скриншот состояния системы при $b = 1.79$. Цвета соответствуют ссылкам [1, 2]: синий – *C*, красный – *D*, жёлтый – *D*, который был *C* на прошлом шаге, и зелёный – *C*, который был *D* на прошлом шаге.

Но при b в промежутке от 9/5 до 2, система ведёт себя хаотично, и не приходит в стационарное состояние (рис. 3а), а среднее значение плотности кооператоров в системе на бесконечности принимает определённое значение, при любых начальных условиях (рис. 3б).

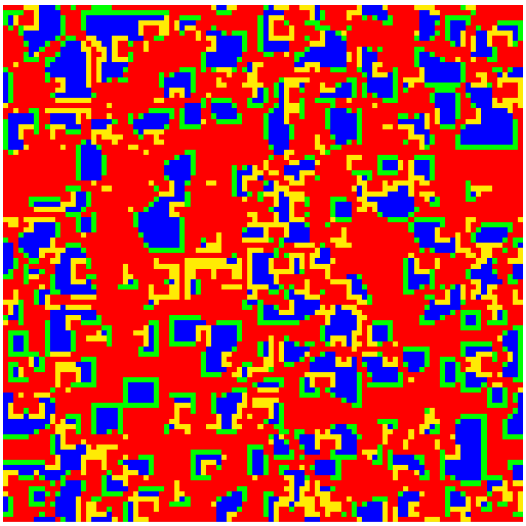

Рис. 3а: Скриншот состояния системы при b = 1.81. Аналогично рис. 2.

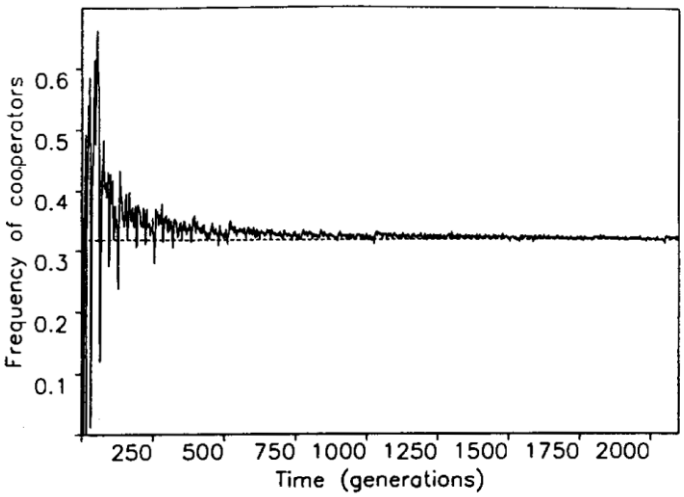

Рис. 3б: График зависимости плотности кооператоров от времени.

На рисунке 3б показано изменение среднего значения плотности кооператоров от времени, на котором можно видеть, что при больших временах это значение стремится к конечному числу, и это число $x = 12 \text{ Log } 2 - 8$ [1].

Так же было произведено измерение плотности кооператоров и в других точках [3]. Это измерение также выявило особенность в точке 9/5.

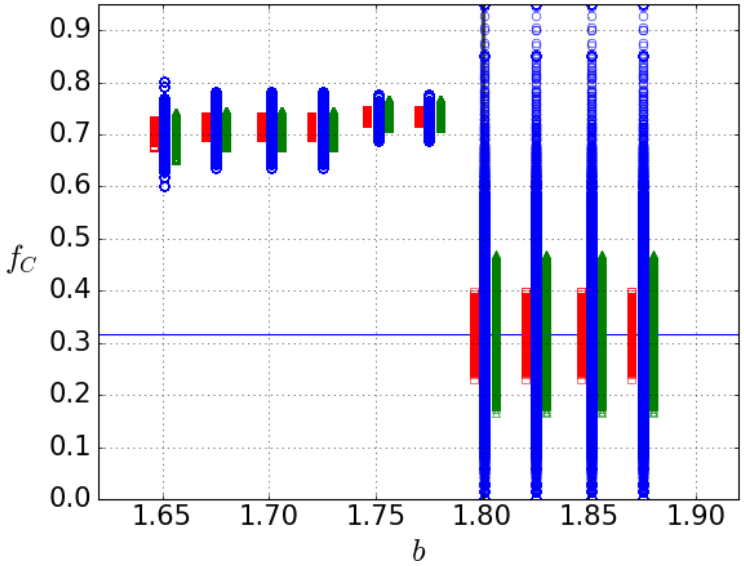

Рис. 4: Концентрация C для решёток размеров 20×20 (синий), 50×50 (зелёный), 100×100 (красный).

3. Практическая часть

3.1 Модель

Основным опорным документом в реализации модели является статья [1]. В ней описан сам, довольно простой, алгоритм игры, и его было не трудно реализовать на одном из известных мне языков программирования.

Первая реализация была написана на языке «C#» в среде «Unity3d». Язык оказался крайне удобен для первого приближения модели, а среда Unity3d была использована для визуализации (рис. 2 и 3а были созданы именно там). Моделирование производилось на лэптопе.

Следующая реализация была написана на языке «C++», для суперкомпьютера «Мантикор», расположенного в «ВЦЧ РАН». Эта реализация потребовалась для симуляции модели с игровым полем размером L×L больше 1000×1000, что занимало довольно много времени на лэптопе.

Для оптимизации была написана программа на чистом «C», для симуляции игрового поля экстремального размера L×L = 50.000×50.000, для подтверждения тенденции измеряемых показателей.

3.2 Исследование модели

Главным исследуемым параметром была выбрана фрактальная размерность Минковского границ кластеров. Кластеры – это группы игроков, имеющих одинаковую стратегию и расположенных рядом с друг другом.

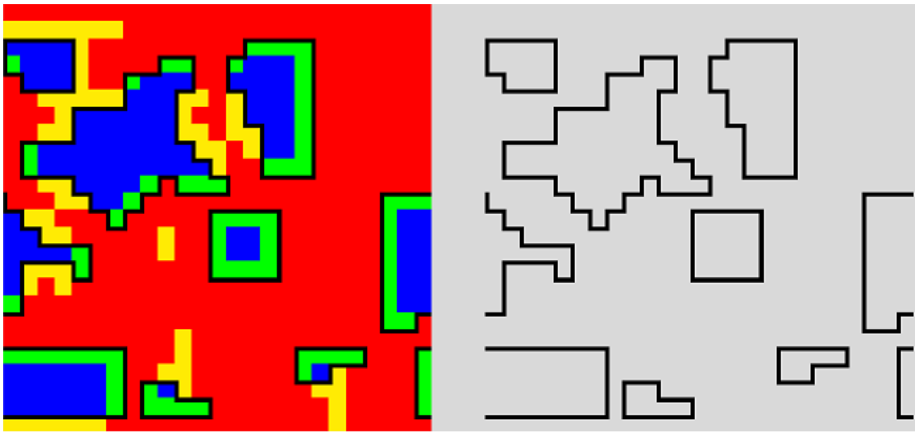

Рис. 5: Чёрные линии являются границей кластеров.

Задачей было установить зависимость этой величины от параметра b. Для этого нужно было запрограммировать метод получения карты границ и сам метод измерения фрактальной размерности.

3.3 Фрактальная размерность Минковского

Проведя исследование, оказалось, что фрактальных размерностей в природе много, и было трудно выбрать подходящую. В итоге выбор пал на размерность Минковского или Box-Counting Dimension, которая определяется следующим образом:

$$\dim_{box}(S) = \lim_{\epsilon \to 0} \frac{\log N(\epsilon)}{\log(1/\epsilon)}$$

где $N(\varepsilon)$ минимальное число квадратов размера ε, нужное для покрытия границы кластера, как показано на рисунке 6.

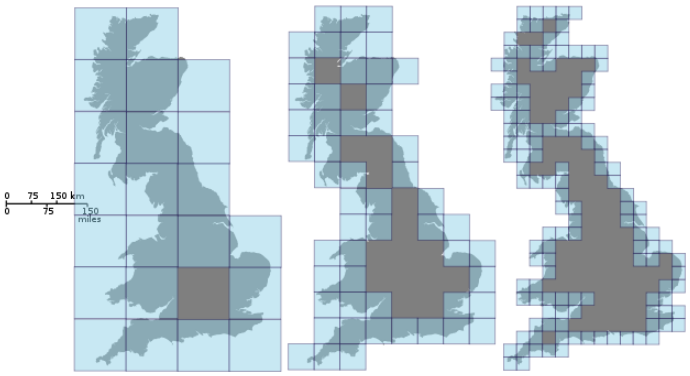

Рис. 6: Оценка размерности Минковского побережья Великобритании.

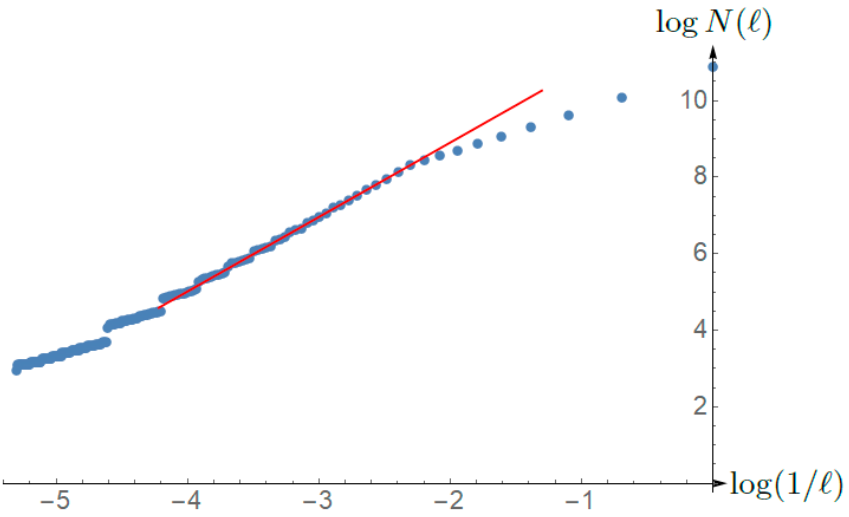

Рис. 7: Логарифмический график зависимости количества квадратов и их размера для системы размера 1000×1000.

На рисунке 7 изображены точки, показывающие зависимость минимальное количество квадратов нужное для покрытия границ кластеров и размера квадратов. Теперь нужно построить аппроксимацию этого графика. Как можно заметить, не все точки пригодны для аппроксимации по следующим причинам. Правая часть графика – покрытие квадратами маленького размера, размерами сопоставимыми с толщиной границы и, если аппроксимировать эту часть графика, мы получаем наклон прямой 1. В левой части графика квадраты, наоборот, очень большие, и здесь даёт шумы дискретность самой модели. Поэму мы не берём в расчёт левую и правую часть графика при аппроксимации. Наклон полученной прямой и является размерностью Минковского.

Я использовал этот метод для систем различных размеров. Результат оказался неожиданным: для обоих режимов $\dim_{box} \to 2$ при $L \to \infty$.

Size	$b = 1.79$	$b = 1.81$
100×100	1.77639(48)	1.38031(70)
200×200	1.76214(30)	1.76215(45)
500×500	1.93572(07)	1.93568(07)
1000×1000	1.95718(03)	1.95721(02)

Таблица 1: Dim_{box} для систем размеров 100×100, 200×200, 500×500, 1000×1000.

Из этих данных следует вывод, что граница кластеров является кривой, заполняющей всё пространство. Для режима при $b > 9/5$ этот исход вполне очевидный, но не для режима при $b < 9/5$, так как структура границы совсем не похожа на такую кривую. Но в самом деле, если взглянуть на границу системы очень большого размера, можно видеть такую кривую.

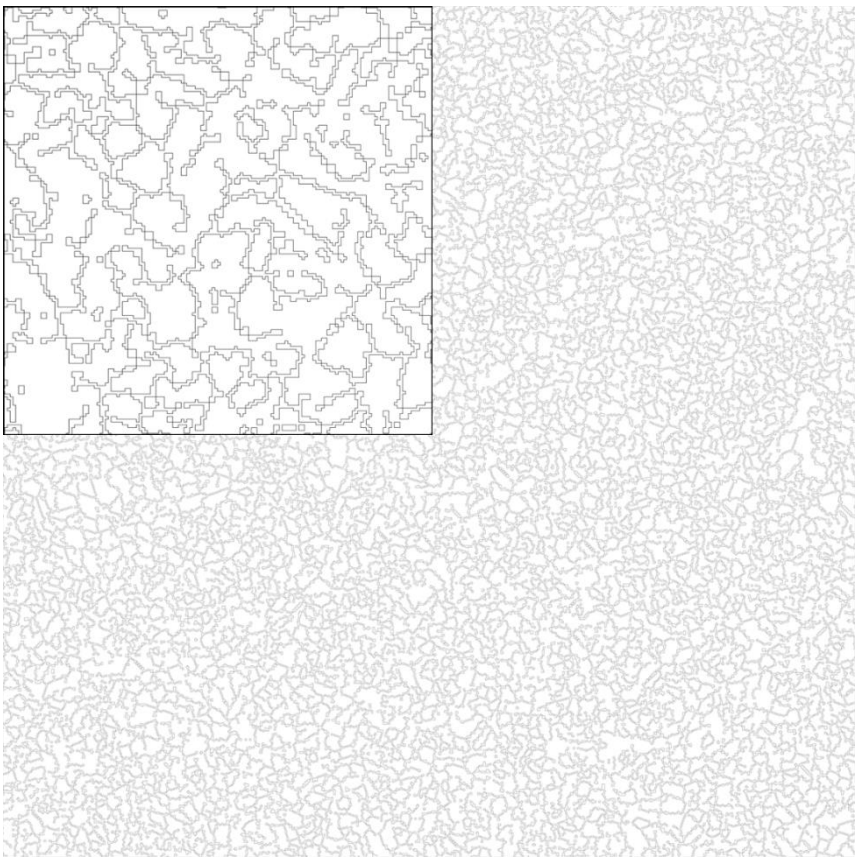

Рис. 8: Скриншот границ кластеров системы размера L×L = 1000×1000 при b = 1.79.

3.4 Треугольная решётка

Измерение фрактальной размерности границ кластеров дало не тривиальный результат. Требовалось понять, почему граница ведёт себя как линия, заполняющая всё пространство. Было решено поменять структуру локальных связей: вместо квадратной решётки использовать треугольную.

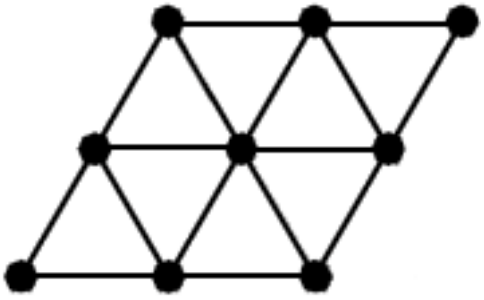

Рис. 9: Треугольная решётка, игроки находятся в узлах решётки.

Используя такую модель, у игроков меняется количество связей с соседними игроками: в квадратной решётке их восемь, а в треугольной – шесть.

Теперь нужно провести аналогию с задачей на квадратной решётке, и найти аналогичные режимы и точку фазового перехода. Построим график плотности кооператоров, как и в исходной задаче: система так же стартует из случайного состояния, с вероятностью 21% появления кооператоров.

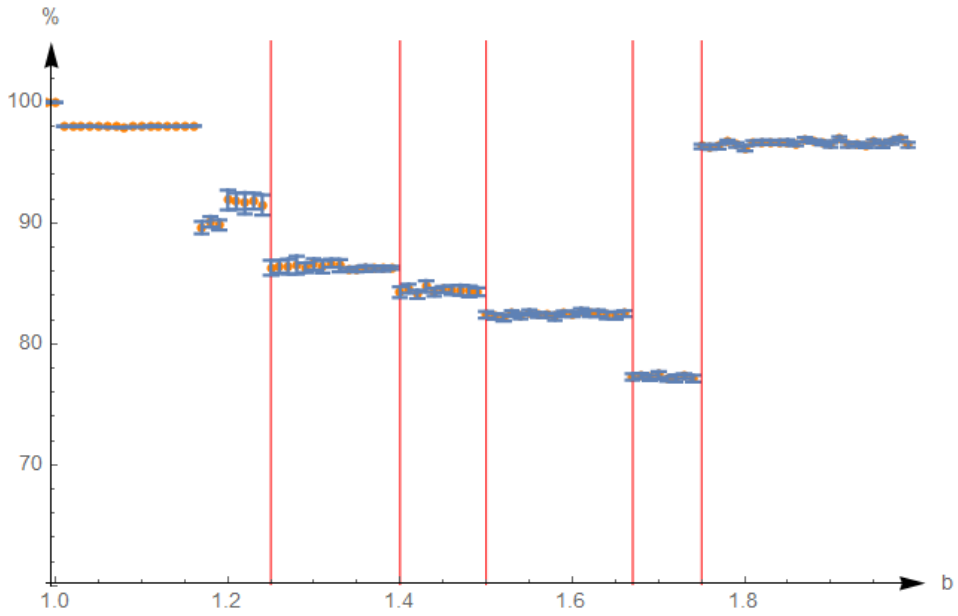

Рис. 10: Концентрация C для треугольной решётки размера 250×250.

Как можно видеть по рисунку 10, в системе с треугольной решёткой обнаружены скачки плотности. Так же существенное отличие от квадратной решетки: во всех режимах наблюдается стационарная картина. Из этого следует, отсутствие режима динамического равновесия, соответствующий режиму $9/5 < b < 2$ квадратной решетки.

Был обнаружен режим, при котором дефектор, стоящий в центре в поле из кооператоров, бесконечно растёт, и наблюдаются изолированные кластеры кооператоров определенной формы (рис. 11). Кластеры такой формы выживают в том случае, если рядом с ними, на расстоянии двух узлов, находится другой такой же.

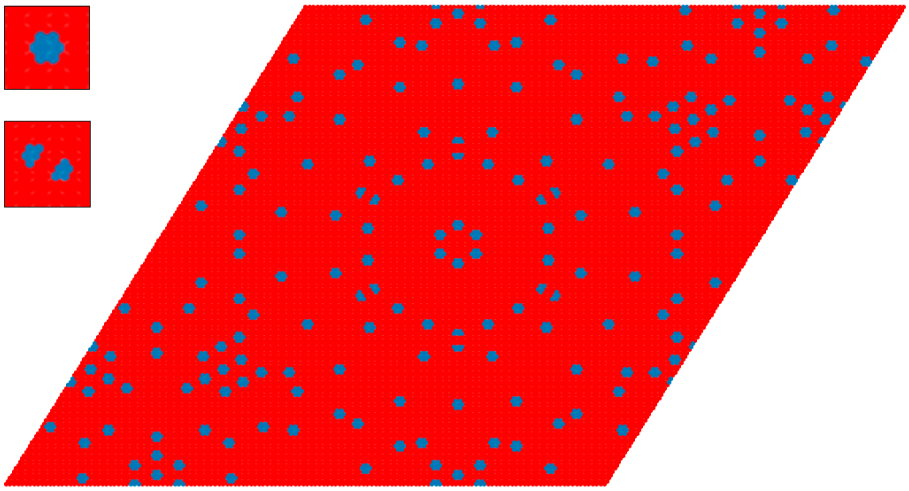

Рис. 11: Скриншот состояния системы размера 150×150 при b=2.34. Синим цветом обозначены кооператоры, красным – дефекторы. Справа формы устойчивых кластеров при данном режиме.

4. Заключение

Таким образом отличие размерностей Минковского границ кластеров режимов при $b > 9/5$ и при $b < 9/5$ существует только при малых размерах, но при $L \to \infty$ $\dim_{box} \to 2$ для обоих режимов. Треугольная решётка дала новые данные для исследования, хотя и не похожие на результаты квадратной решётки, но система сама по себе имеет интересную динамику и требует дополнительного исследования.

5. Список используемой литературы

1) M.A. Nowak and R.M.May. *Evolutionary games and spatial chaos*. Nature **359**, page 826, (1992).
2) M.A. Nowak and R.M. May. *The spatial dilemmas of evolution*. Int. J. Bifurcation and Chaos 3(1), pages 35–78, (1993).
3) Sergei Kolotev, Aleksandr Malyutin, Evgeni Burovski, Sergei Krashakov, Lev Shchur, *Dynamic fractals in spatial evolutionary games*, Physica A: Statistical Mechanics and its Applications **499**, pages 142-147, (2018).

6. Приложение

Программа на C++

Ссылка на репозиторий: https://github.com/AlezM/spatial-games-c

```
#include <iostream>
#include <math.h>
#include <vector>
#include <ctime>
#include <fstream>

using namespace std;

//Type: fasle - Cooperator, true - Deffector
class Tile {
public:
	bool type;
	double score;
	Tile() : type(false), score(0) {};
	Tile(bool _type) : type(_type), score(0) {};

	void InterractWith(Tile member, double b) {
		if (member.type == 0) {
			if (type == 0)
				score += 1;
			else
				score += b;
		}
	}

	void SetType(bool _type) {
		type = _type;
	}
};

int main() {
	int mapSize = 25000;
	double b = 1.79;
	vector<Tile> map;
	vector<Tile> newMap;

	double defectorsSpawnPercentage = 21;
	string seed;

	int borderMapScale = 10;
	vector<bool> borderMap;

	int startSize = 1;
	int finishSize = 500;
	int step = 1;

	map.resize(mapSize*mapSize, Tile(0));
	borderMap.resize(mapSize*mapSize, false);

// SetUp
	srand(time(0));
	for (int i = 0; i < mapSize*mapSize; i++) {
		map[i] = (1 + rand() % 100 < 21) ? Tile(true) : Tile(false);
	}
	cout << "> Map set up." << endl;

////////////////////////////////

// Generations
	for (int i = 0; i < 200; i++) {
```

```
		//Scores
		for (int i = 0; i < mapSize*mapSize; i++) {
			for (int x = -1; x < 2; x++) {
				for (int y = -1; y < 2; y++) {
					int memberX = ((i + x) + mapSize) % mapSize;
					int memberY = (((i / mapSize) + y) + mapSize) % mapSize;

					map[i].InterractWith(map[memberX + memberY * mapSize], b);
				}
			}
		}

		newMap.insert(newMap.begin(), map.begin(), map.end());

		for (int i = 0; i < mapSize; i++)
		{
			for (int j = 0; j < mapSize; j++)
			{
				//Search for the reachest member
				int newType = map[i + j * mapSize].type;
				double maxMemberScore = 0;
				bool uncertainty = false;
				for (int x = -1; x <= 1; x++)
				{
					for (int y = -1; y <= 1; y++)
					{
						int memberIndex = (i + x + mapSize) % mapSize + (j + y +
mapSize) % mapSize * mapSize;

						double memberScore = map[memberIndex].score;
						int memberType = map[memberIndex].type;

						if (memberScore > maxMemberScore)
						{
							maxMemberScore = memberScore;
							newType = memberType;
							uncertainty = false;
						}
						else if (memberScore == maxMemberScore && newType != member-
Type)
						{
							uncertainty = true;
						}
					}
				}
				newMap[i + j * mapSize] = Tile(newType);
			}
		}

		for (int i = 0; i < mapSize*mapSize; i++) {
			newMap.at(i).score = 0;
		}

		map.clear();
		map = newMap;
		newMap.clear();
	}

	cout << "> Map ready." << endl;

////////////////////////////////

// BorderMap

	int borderMapSize = borderMapScale * mapSize;

	borderMap.resize(borderMapSize * borderMapSize);

	for (int i = 0; i < mapSize; i++)
```

```
	{
		for (int j = 0; j < mapSize; j++)
		{
			if (map[i + j * mapSize].type != map[(i - 1 + mapSize) % mapSize + j * map-
Size].type)
			{
				for (int k = borderMapScale * j; k < borderMapScale * (j + 1) + 1; k++)
				{
					borderMap[i * borderMapScale + k % borderMapSize * mapSize] = true;
				}
			}
		}
	}

	for (int i = 0; i < mapSize; i++) {
		for (int j = 0; j < mapSize; j++) {
			if (map[j + i * mapSize].type != map[j + (i - 1 + mapSize) % mapSize * map-
Size].type) {
				for (int k = borderMapScale * j; k < borderMapScale * (j + 1) + 1; k++) {
					borderMap[k % borderMapSize + i * borderMapScale * borderMapScale]
= true;
				}
			}
		}
	}

	cout << "> Border map ready." << endl;


	map.clear();

////////////////////////////////


	vector<bool> filledBoxes;
	ofstream fout("out_179_20k.txt", ios_base::trunc);

	filledBoxes.resize(borderMapSize * borderMapSize);

	for (int b = startSize; b <= finishSize; b += step)
	{
		// Filling Boxes
		size_t hCount = borderMapSize / b; //Hight
		size_t wCount = borderMapSize / b; //Width


		for (size_t x = 0; x < borderMapSize; x++) {
			for (size_t y = 0; y < borderMapSize; y++) {
				if (borderMap[x + y * borderMapSize]) {
					size_t xBox = x / b;
					size_t yBox = y / b;
					filledBoxes[xBox + yBox * borderMapSize] = true;
				}
			}
		}

		// Counting Boxes
		size_t a = 0;
		size_t fbSize = sqrt(filledBoxes.size());
		for (size_t i = 0; i < fbSize; i++) {
			for (size_t j = 0; j < fbSize; j++) {
				if (filledBoxes[i + j * fbSize]) {
					a++;
				}
			}
		}

		for (size_t x = 0; x < borderMapSize; x++) {
```

```
                        for (size_t y = 0; y < borderMapSize; y++) {
                                filledBoxes[x + y * borderMapSize] = false;
                        }
                }

                cout << "|";

                fout << "{" << b << ", " << a << "},"  << endl;
        }
        cout << "\nComplete.\n";
        fout.close();


        return 0;
}
```