Frontier Research Topics in Network Science Experimental Report

Evolutionary Games and Spatial Chaos

Zhongyuan Chen

Evolutionary Games implementation using Python

Hangzhou Dianzi University
School of Cybersecurity
2019-6-15

Contents

1 Overview

This paper introduces the concept of Prisoners' Dilemma based evolutionary games and offers code implementation.

1.1 Abstract

Prisoners' Dilemma is a well-known problem. The related evolutionary games are studied by Martin et.al. [1]. In this paper, we propose a simple game, namely the name of cooperators and defectors, and let the game run for 200 generations. Finally, we see some strange patterns.

1.2 Background

In our life, Prisoners' Dilemma exists every where. For example, who is to clean the dormitory room? Of course that should be you or one of your roommates. If all of you choose to cooperate (namely to clean the room), then all of you will benefit a lot. But if only one of you choose to be the cooperator, then the other three defectors will win a lot, but the only cooperator will be the loser.

2 Important Concepts

Several concepts appear frequently in this paper are introduced here.

2.1 Prisoners' Dilemma

This is copied from wikipedia.

The **prisoner's dilemma** is a standard example of a game analyzed in game theory that shows why two completely rational individuals might not cooperate, even if it appears that it is in their best interests to do so. It was originally framed by Merrill Flood and Melvin Dresher while working at RAND in 1950. Albert W. Tucker formalized the game with prison sentence rewards and named it "prisoner's dilemma", presenting it as follows:

Two members of a criminal gang are arrested and imprisoned. Each prisoner is in solitary confinement with no means of communicating with the other. The prosecutors lack sufficient evidence to convict the pair on the principal charge, but they have enough to convict both on a lesser charge. Simultaneously, the prosecutors offer each prisoner a bargain. Each prisoner is given the opportunity either to betray the other by testifying that the other committed the crime, or to cooperate with the other by remaining silent. The offer is:

1. If A and B each betray the other, each of them serves two years in prison.
2. If A betrays B but B remains silent, A will be set free and B will serve three years in prison (and vice versa)
3. If A and B both remain silent, both of them will serve only one year in prison (on the lesser charge).

3 Model Description

Our model is basically a board filled with cooperators and defectors. Here each player will play with its neighbors and win some scores. The one with the highest scores will have one more lattices to play.

4 Code Implementation

The idea is simple, so the code is simply simpler.

4.1 Data Structures

No fancy data structures actually.

4.1.1 Numpy ndarray

According to wikipedia, an adjacency list is a collection of unordered lists used to represent a finite graph and each list describes the set of neighbors of a vertex in the graph. We use STL array and vector to implement this. We make an array of vectors a private variable of class Graph. The index of a vector serves as an

identifier of a certain vertex. Neighbors of that certain vertex are stored in the vector.

4.2 Simulation Steps

We let the players play for 200 generations. Then we plot the whole board.

5 Results and Analysis

To be honest, I did not figure out some patterns mentioned in the original essay, here are what I got.

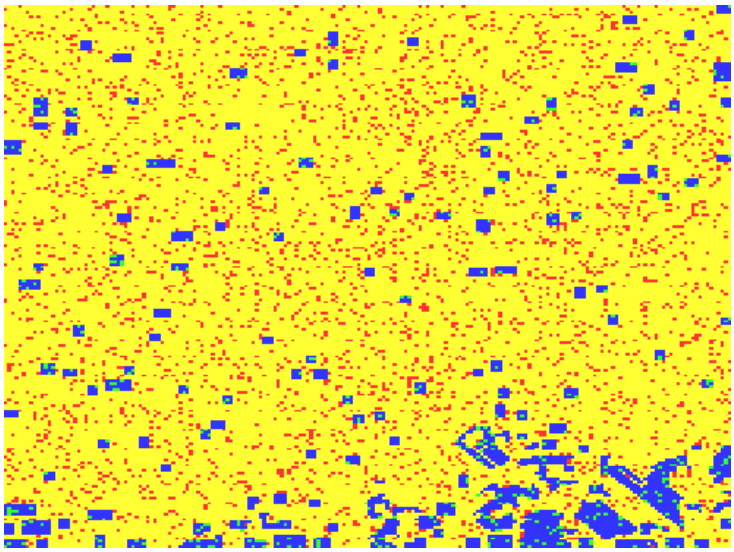

Figure 1: Game pattern with $b = 1.70$

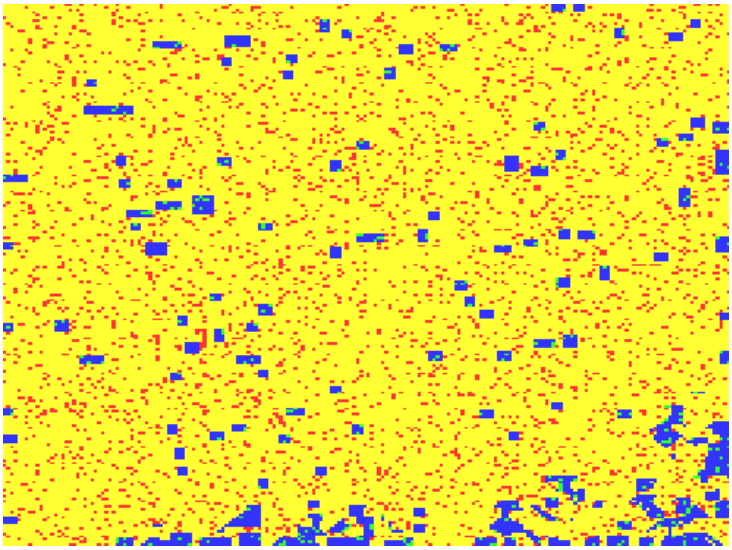

Figure 2: Game pattern with $b = 1.74$

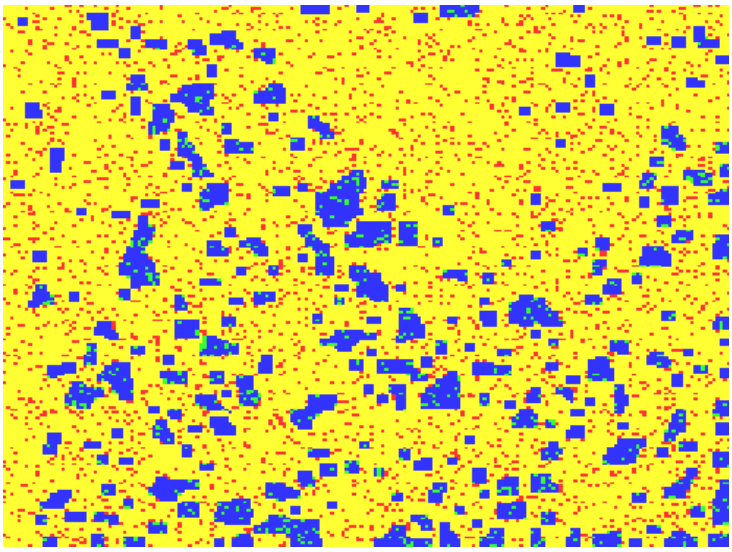

Figure 3: Game pattern with $b = 1.78$

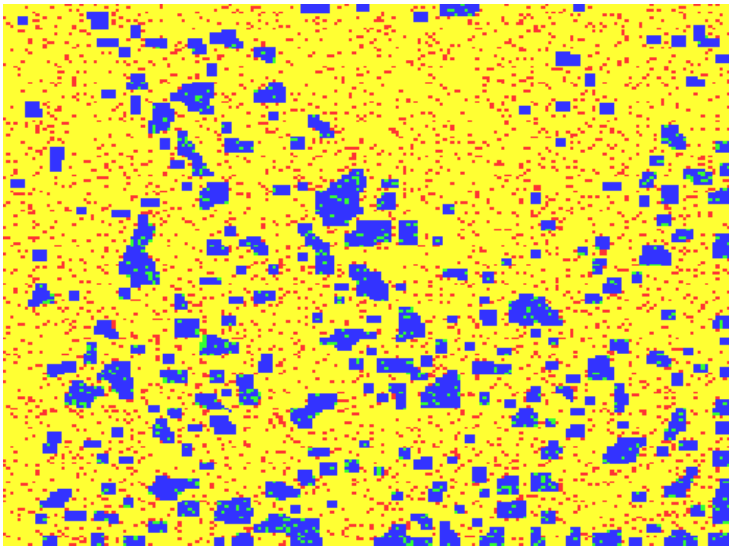

Figure 4: Game pattern with $b = 1.82$

6 Conclusion

I am sorry that I did not pay enough attention to this homework. Though this is essential the last homework in my college life.

References

[1] M. A. Nowak and R. M. May, "Evolutionary games and spatial chaos," *Nature*, vol. 359, no. 6398, p. 826, 1992.

Appendices

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

def main(factor_T):
    del_line = '\r\x1b[K'

    SIDE = 200
    GENERATIONS = 200
    board = np.random.rand(SIDE, SIDE)
    scores = np.zeros((SIDE, SIDE))

    # factor_T = 1.62 # defector vs cooperator
    factor_R = 1 # cooperator vs cooperator
    factor_P = 0 # defector vs defector
    factor_S = 0 # cooperator vs defector

    # 90% cooperators (True) and 10% defectors (False)
    board = board > 0.1
    board_prev = board.copy()

    np.random.seed(2333)

    def index_valid(index, board):
        return (0 <= index[0] < board.shape[0]
            and 0 <= index[1] < board.shape[1])


    def play(times=1):
        for t in range(times):
            # print(board)
            print(f'{del_line}{t + 1} / {times}', end='')
            board_prev = board.copy()
            scores = np.zeros((SIDE, SIDE))

            it = np.nditer(board, flags=['multi_index'])
            while not it.finished:
                owner = it.multi_index
                owner_cooperate = it[0]

                # print(f'\nowner: {repr(owner)} {owner_cooperate}')
```

```
        for offset in [(0, 1), (1, -1), (1, 0), (1, 1)]:
            guest = (owner[0] + offset[0],
                owner[1] + offset[1])
            # print(f'guest: {repr(guest)}', end=' ')
            if not index_valid(guest, board):
                # print('guest invalid')
                continue
            else:
                # print(f'{board[guest]}')
                pass


            if board[owner] and board[guest]:
                # print('both +1')
                scores[owner] += factor_R
                scores[guest] += factor_R
            elif board[owner] and not board[guest]:
                # print('owner +0, guest +b')
                scores[owner] += factor_S
                scores[guest] += factor_T
            elif not board[owner] and board[guest]:
                # print('owner +b, guest +0')
                scores[owner] += factor_T
                scores[guest] += factor_S
            else:
                # print('owner +0, guest +0')
                scores[owner] += factor_P
                scores[guest] += factor_P

        it.iternext()

    # print(scores)

    it = np.nditer(scores, flags=['multi_index'])
    while not it.finished:
        lattice = it.multi_index

        candidates = [(lattice[0] + i, lattice[1] + j)
            for i in range(-1, 2) for j in range(-1, 2)]

        candidates_score = [(i, j, scores[i, j])
            for i, j in candidates
            if index_valid((i, j), scores)]

        winner = max(candidates_score, key=lambda x: x[2])
```

```
                board[lattice] = board[winner[:2]]
                it.iternext()




    def show():
        blue = [51, 51, 255] # C -> C
        red = [255, 51, 51] # D -> D
        yellow = [255, 255, 51] # C -> D
        green = [51, 255, 51] # D -> C

        pic = np.zeros((SIDE, SIDE, 3), dtype=np.uint8)
        for i in range(SIDE):
            for j in range(SIDE):
                # raise Exception
                if board_prev[i, j] and board[i, j]:
                    pic[i, j] = blue
                elif not board_prev[i, j] and not board[i, j]:
                    pic[i, j] = red
                elif board_prev[i, j] and not board[i, j]:
                    pic[i, j] = yellow
                else:
                    pic[i, j] = green

        # raise Exception

        plt.imshow(pic, interpolation='nearest', aspect='auto')
        plt.axis('off')
        plt.savefig(f'evo_{factor_T}.png')
        plt.savefig(f'evo_{factor_T}.eps', format='eps')

    play(200)
    show()

if __name__ == '__main__':
    for i in range(1, 5):
        main(1.58 + i * 0.08)
```