Assignment 2: Design strategies for a tournament in Iterated Prisoner's Dilemma

ENM140, Game theory and rationality 2017

Contents

1 Overview

With this assignment, we organize a competition where the course participants contribute strategies for playing the iterated prisoners dilemma. In the competition each strategy will play against all other strategies (including against itself). Your job is to implement the strategies as functions in Javascript. We have written all the code for running the game, so you only have to write a set of functions that return a value corresponding to "defect" or "cooperate".

We will play three versions of the game:

1. Played for 10 rounds. Submit 3 strategies for this version.
2. Played for 200 rounds. Submit 3 strategies for this version.
3. Played for 200 rounds with a mistake rate of 2%. This means that your chosen action will be used with 98% probability and otherwise changed to the other action. Submit 1 strategy for this version.

We will run the game with payoffs $u_{dd} = 1, u_{dc} = 5, u_{cd} = 0, u_{cc} = 3$, where u_{dd} is the payoff for player 1 if both defect, u_{dc} is the payoff if player 1 defects and player 2 cooperates, etc. The winning strategy in each version of the game will the one who gets the highest average payoff across all the matches against other strategies, including

themselves. (All the 10-round strategies will play in one tournament, all the 200-round strategies in another, and all the 200-round strategies with mistakes in a third.)

You are encouraged to work together in groups and, if you wish, to try your strategies against each other (see Section 2.5). However, everyone must finally submit their own set of strategies as described in Section 3.

Please read these instructions carefully before starting your work.

2 How to get started

First of all, download a zip file with the code from the course homepage. Unpack the zip file and you will have a folder called `ipd-tournament` with a few different things in it:

- The file `develop-strategy.html` is the best place to start developing your own strategies. More about this in a moment.
- The file `test-all.html` and the file `js/strategies.js` are for testing and submitting your work. More about this in Section 3.
- The files `js/game.js`, `js/opponents.js` and all the other files contain the code that actually runs the game. You don't have to read these unless you want to.

2.1 Develop your strategies in `develop-strategy.html`

Open the html file `develop-strategy.html` in a web browser. We have successfully tested the code on Firefox and Chrome. It might work in other browsers, too, but Firefox and Chrome are more likely to work.

The page should show some Javascript code which implements the tit-for-two-tats strategy. You can change the strategy code, the number of rounds played, and the mistake rate. Run the code by clicking the Run button or pressing Ctrl+Enter on your keyboard. You should then see tables of results below the strategy code editor.

You can use this application to test your strategies against some opponent strategies. Just change the code in the code editor and run again to see results.

2.2 More examples in `js/strategies.js`

To see more examples of how strategies can be implemented, open the file `js/strategies.js` in a text editor. This file defines strategies named like `example`$\texttt{id}N\mathit{variant}$, where $N \in \{10, 200\}$ and $\mathit{variant} \in \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{mistakes}\}$. This is where you should insert your own strategies. The numbers indicate how many rounds will be played with the strategy and the letters `a`, `b`, `c` identify your three different strategies in the first two variants of the game. The strategy ending in `200mistakes` is the one that

will be used in the game with 2% mistake rate.

2.3 Using the action history in your strategies

Look closely at the `chooseAction` function in the example strategies in `js/strategies.js` (e.g. "always cooperate", "always defect", "tit-for-tat", etc). Note that they are all functions that take three arguments `(me, opponent, t)` and always return either `0` (= defect) or `1` (= cooperate). The argument `t` will be an integer $0 \leq t \leq (N-1)$ representing the current "time" or round number. The arguments `me` and `opponent` will be arrays of the form

$$\texttt{me} = [m_0, m_1, \ldots, m_{t-1}, 0, 0, \ldots],$$

$$\texttt{opponent} = [o_0, o_1, \ldots, o_{t-1}, 0, 0, \ldots],$$

where m_t is "my" action at time t and o_t is the opponent's action. The arrays `me` and `opponent` will always have N elements in an N-round game, but the elements at position t and later will always be zero. Note that Javascript arrays are zero-indexed, i.e., you should think of time as starting at $t = 0$ and running until $t = N - 1$ in an N-round game.

For example, a tit-for-tat strategy can be mathematically defined as

$$\begin{cases} m_0 = 1, \\ m_t = o_{t-1}, \quad t = 1, 2, 3, \ldots. \end{cases}$$

At time $t = 0$, no history has passed, and the tit-for-tat strategy generously starts with cooperation, $m_0 = 1$. In all following time steps, the tit-for-tat strategy always copies what the opponent did last.

Your job in the `chooseAction(me, opponent, t)` function is to return a value which will then become the value of `me[t]`. In Javascript, the choice of action in tit-for-tat strategy can be formulated as follows:

```
// Tit for tat
function chooseAction(me, opponent, t) {
  if (t == 0) {
    return 1; // cooperate in first round
  } else {
    return opponent[t-1]; // otherwise copy opponent
  }
}
```

Note that if you want to use history, it only makes sense to use it at time $t = 1$ and later, since all elements in `me` and `opponent` will be 0 when $t = 0$. Use something like the `if`-clause above to check that `t > 0`.

A few specific hints for how to use the history:

```
// the opponent's action in the previous round
opponent[t-1]

// your action in the previous round
me[t-1]
```

```
6
7  // all your previous actions in an array of length $t$,
8  // i.e., [m_0, m_1, ..., m_{t-1}]
9  me.slice(0, t)
10
11 // all the opponent's previous actions in an array of length $t$
12 opponent.slice(0, t)
```

2.4 Using state variables

For some strategies you might want to keep track of some state in your strategy. This can be inconvenient or impossible to do without some additional variable, so we allow you to define variables in the outer function that wraps each of your `chooseAction` functions. All the state variables that are defined just before the `chooseAction` function are (re)initialized before each game, so there is no memory between games. You must define your state variables using the keyword `var`, for example as follows:

```
1  strategies[cid + '10a'] = function () {
2    var oneStateVariable = 0;
3    var anotherOne = true;
4
5    function chooseAction(me, opponent, t) {
6      // choose actions in here, possibly
7      // using and/or changing the state variables
8    }
9
10   return chooseAction;
11 }
```

This is the reason why each strategy is a function inside another function: the outer function initializes your state variables (which only the inner function `chooseAction` can access), and then the outer function returns the inner function which actually makes the choices in each time step.

There is an example in the `js/strategies.js` file that may help to explain this:

```
1  strategies[cid + '200b'] = function () {
2    // evil is a state variable for this strategy.
3    // The initialization code ('var evil = false;'') will be run
4    // before each game, so the variable evil will always equal
5    // false when the game starts.
6
7    var evil = false; // start out as not evil
8
9    function chooseAction(me, opponent, t) {
10     // This strategy uses the state variable 'evil'.
11     // In every round, turn evil with 5% probability.
12     // (And remain evil until the 200 rounds are over.)
13     if (Math.random() < 0.05) {
14       evil = true;
15     }
16
17     // If evil, defect
18     if (evil) return 0;
```

```
19
20      return 1; // and cooperate otherwise
21    }
22    return chooseAction;
23  }
```

What this means is the following: The strategy is to cooperate unconditionally, unless the agent suddenly turns madly evil. The state variable `evil` is initialized as `false`. In each round the state variable `evil` is set to `true` if it is already `true` or if a random number $r \sim \text{Uniform}(0, 1) < 0.1$, otherwise it is kept `false`. In other words, once the strategy goes evil it will never go back. The strategy returns `0` (defect) if evilness has struck and `1` (cooperate) otherwise.

2.5 Adding more opponents in `js/opponents.js`

The opponents your strategy plays against in the `develop-strategy.html` environment are all defined in `js/opponents.js`. If you want to test your strategy against more strategies, feel free to make more. Just copy one of the example strategies, rename it to something else and implement your own opponent score. Note that opponents are implemented just like your own strategies, so you can develop your new opponents in `develop-strategy.html` and then copy them into the `js/opponents.js` file.

When you make changes in `js/opponents.js`, you must reload the page before the changes take effect.

2.6 Learning more Javascript

If you have not programmed before in Javascript, you can probably learn a few things by studying the examples in the `js/strategies.js` file and the `js/opponents.js` file.

An excellent resource for learning Javascript is the Mozilla Developer Network (MDN) website: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide.

3 How to submit

Work on your strategies as much or little as you want. Collect your finished strategies in the file `js/strategies.js`. You may use the same strategy in several cases, but all the seven functions (10a, 10b, 10c, 200a, 200b, 200c, 200mistakes) must be defined. (Make sure that you change the names: in the testing environment in `develop-strategy.html`, your strategy must be named `'testStrategy'`, but in the file with finished strategies they must be named things like `cid + '200b'`.

When you are done, follow these instructions to test your strategies and submit them:

1. Open the `js/strategies.js` file and replace the example strategies by your own

strategy functions. It should work perfectly well to everything but the name `'testStrategy'` from the `develop-strategy.html` page where you developed the strategies.

2. Replace the string `'examplecid'` for your own cid near the top of `js/strategies.js`.

3. Save the `js/strategies.js` file.

4. Open the file `test-all.html` in a web browser (preferably Firefox or Chrome).

5. As soon as you open this file, your strategies will be played against all the test strategies in `js/opponents.js`. If everything goes well you should have a table full of success reports. If there are any problems, you will see some more or less helpful error messages. If there are error messages problem, try to fix it and then reload the file by clicking a refresh button, or pressing Ctrl+R, Cmd+R, F5, or whatever keyboard shortcut is used on your system.

6. When the testing page says success on everything, you are done. Please go to https://www.dropbox.com/request/D2H99U6PBQItrLCGsGzv and submit your `strategies.js` file. Make sure you leave your name and student email address in the submission form.