CT421: Project 2: Iterated Prisoner's Dilemma

1 Introduction

The **Prisoner's Dilemma** can be defined as follows:

> Suppose a man with a chest full of gold coins invites you and another player to participate in a game. Each of you has two choices: you can either cooperate or defect. If you both cooperate, each of you receives 3 coins. If one of you cooperates while the other defects, the defector receives 5 coins and the cooperator gets nothing. If you both defect, each of you receives 1 coin. Suppose your opponent cooperates; you could choose to cooperate and get 3 coins, or you could defect and get 5 coins instead—so you're better off defecting. But what if your opponent defects instead? In that case, if you cooperate, you receive no coins, but if you defect, you get at least 1 coin. Thus, no matter what your opponent chooses, you are always better off defecting. However, if your opponent is also rational, they will reach the same conclusion and also defect. As a result, when both players act in their best interest, they end up in a suboptimal situation, each receiving only 1 coin, whereas mutual cooperation would have yielded 3 coins each [1].

This is one of the most famous problems in game theory and it resembles many real-world scenarios, from the Cold War to the competitive brands/Companies and their advertising strategies. However, in these real-life situations, we are not just playing the Prisoner's Dilemma once; we play it repeatedly. If I defect now, my opponent will know and can use that against me. So, when this game is repeated, what is the best strategy?

2 Implementation Details & Design

2.1 Genome Structure

I defined the genome structure as a strategy class with the following fields:

- `First move`: The initial action taken by the strategy.
- `Responses`: A dictionary that defines how to respond to the opponent's last two moves.
- `Hold grudges`: A maximal hold-grudge trait, similar to the Friedman strategy, where once the opponent defects, the strategy continues defecting for the rest of the game.
- `Random defection`: This allows the strategy to incorporate an element of sneakiness by randomly defecting with a certain probability, provoking the opponent occasionally.

The genome has a memory length of 2, meaning the strategy considers the opponent's last two moves when making decisions. This results in 1 (first move) + 4 (responses) = 5 binary decisions, leading to a strategy space of 2^5.

- A decision for the first move.
- A response for each possible history of two moves: (CC, CD, DC, DD).

Initially, I found that even with this relatively large sample space, an optimal solution could often be found within a single generation. So to increase the complexity of the strategy space, I introduced the `hold`

`grudges` trait, ensuring that once the opponent defects, the strategy permanently defects in response. This trait is binary, effectively doubling the strategy space:

$$2^5 \times 2 = 2^6.$$

And I incorporated `random defection`, allowing the strategy to defect with a small probability. This probability has 10 possible values (0, 0.1, 0.2, ..., 0.9), further expanding the strategy space:

$$2^6 \times 10 = 640.$$

Not only does this structure significantly expand the strategy space, but it also enabled me to incorporate some of the well-known fixed strategies from Axelrod's first and second tournament, such as `Friedman`, `Joss`, and `tit for two tats`

2.2 Population

To begin, I have chosen to use a randomly generated population to ensure diversity and maximize coverage of the sample space. The code, with equal probability, chooses a random first move, randomly decides to hold grudges, a random response to each last move or last two moves, and a random defection percentage

2.3 Fitness

Once we have generated the solutions—either through random initialization in the first iteration or through reproduction in later iterations—we need an evaluation function to distinguish between better and worse solutions [2].
In this Prisoner's Dilemma setup, the fitness of a genome is determined by how well it performs against all fixed strategies. The more points it accumulates in total across all strategies, the higher its fitness.

2.4 Selection

After evaluating the fitness of our solutions, we need a selection function to determine which solutions should progress to the next generation for reproduction. Typically, we want the best solutions to advance to create better offspring.
I implemented tournament selection, as required by the assignment, where a subset of the population is randomly chosen, and within that subset, the solution with the highest fitness is selected as a parent.

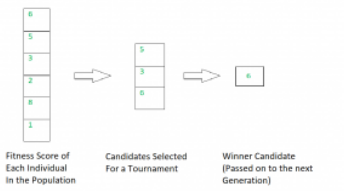

Figure 1: Illustrating Tournament selection. [**?**]

2.5 Reproduction

After selecting the parents, we need to put the parents through the process of reproduction, which consists of crossover and mutation in order to generate new offspring. **Crossover** is used to combine the genetic information of two parents to generate new offspring. The idea is that combining parts of two existing *fit* solutions has the potential to produce better solutions over successive generations. **Mutation** then occurs at a much lower rate than crossover. It is still important as it introduces random changes to individual solutions in the population. This helps maintain diversity in the population and explore areas of the solution space that may not be reachable through crossover alone [2].

Crossover

Crossover involves combining the first move, hold grudges, and random defection traits of the parents. Additionally, the response maps of the parents are merged by randomly selecting values from either parent for each key.

Mutation

Mutation occurs with a low probability. If mutation happens, each trait has a 50% chance of changing. Even if a trait changes, it might still be reassigned the same value. Specifically:

- The first move has a chance of being randomly changed.
- Each value in the response map has a chance of being randomly changed.
- The hold grudges trait may flip its value.
- The random defection rate may be replaced by a new random value from a predefined set.

2.6 Genetic Algorithm

I designed the genetic algorithm to accept the mutation rate parameters, the crossover rate, the population size, and the number of generations as input. This allows me to easily iterate over the algorithm with different parameter values. The process begins by initializing the population, and for each generation, every genome in the population competes against the fixed strategies to determine its fitness. Then, selection is applied to choose the best solutions, followed by crossover and mutation.

3 Experimental Results and Analysis

3.1 Testing The GA (Against fixed strategies for 100 rounds)

To test the accuracy of my genetic algorithm, I first ran it against only Always Cooperate and Always Defect strategies. In this scenario, the optimal solution should be Always Defect, as it maximizes the payoff.

```
0-{(1,): 0, (0,): 0, (1, 1): 0, (1, 0): 1, (0, 1): 0, (0, 0): 0}-False-0.5-100
```

As expected, the genetic algorithm successfully evolved an Always Defect strategy, which is the best possible outcome in this environment. (Although the solution says after a cooperation an defection by the opponent the move will cooperate, but against only always defect that scenario will never occur so this is effectively and always defect strategy in this environment)

Next, I tested the ga against a Tit-for-Tat strategy. Tit-for-Tat is a form of negative reinforcement, meaning that if the genetic algorithm behaves optimally, it should discover that the best strategy is to always cooperate. Against Tit-for-Tat, an Always Defect strategy would score 104 points, while an Always Cooperate strategy would score 300 points.

```
1-{(1,): 1, (0,): 0, (1, 1): 1, (1, 0): 1, (0, 1): 0, (0, 0): 0}-False-0.0-300
```

Again, the algorithm finds an Always Cooperate solution. (Technically, this is not a pure Always Cooperate strategy, as it responds with defection after being defected on and after a DC sequence, but it does not hold maximal grudges or implement a sneaky strategy with random defections. Instead, it cooperates consistently with cooperative opponents, meaning that against Tit-for-Tat, it will always cooperate, making it the optimal solution in this environment.)

Lastly, I tested the genetic algorithm in an environment containing Always Defect and Tit-for-Tat strategies. In this scenario, the optimal solution should be Tit-for-Tat.

```
1-{(1,): 1, (0,): 0, (1, 1): 1, (1, 0): 1, (0, 1): 0, (0, 0): 0}-False-0.0-399
```

Once again, in this environment, the evolved strategy is effectively Tit-for-Tat, as the situations where the opponent's moves are (1,0) or (0,1) cannot arise in this setting.

These results provided strong evidence that my genetic algorithm was functioning correctly and was logically sound.

Link to GitHub Repository

References

[1] Wikipedia Contributors (2019) *Prisoner's dilemma.*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Prisoner%27s_dilemma.(accessed: 1 March 2025)

[2] Colm O'Riordan. (2024). *Learning and IR_p1*. [online] Available at: https://universityofgalway.instructure.com/courses/31932/files/1988333?module_item_id=631165. (accessed: 1 March 2025)

[3] GeeksforGeeks. (2018). *Tournament Selection (GA)*. [online] https://www.geeksforgeeks.org/tournament-selection-ga/ (accessed: 1 March 2025)