Basics

Project Name: Becoming the Optimal Prisoner

Lucy Cheng (lcheng@college.harvard.edu)
Aaron Landesman (aaronlandesman@college.harvard.edu)
Ray Qian (rqian@college.harvard.edu)
Albert Wu (awu@college.harvard.edu)

Brief Overview

We are trying to determine the optimal strategy for the Prisoner's Dilemma through a completely computerized approach without a priori information. The optimal strategy will be most successful when played against a range of other strategies. We must test each strategy against a range of different strategies because for any deterministic strategy, we can always formulate a strategy that beats it.

We will use a genetic algorithm approach to find this optimal strategy. Specifically, we will run multiple generations of the iterative Prisoner's Dilemma, repeatedly evolving the most successful strategies of the previous generation.

Our goal is to learn about the application of machine learning techniques to game theory and to test if well-known results actually hold. For example, many claim that tit-for-tat is the best strategy, and we will empirically test it.

Our overview remained the same.

Features List

Core features

In our assignment, we will create agents which either cooperate or defect according to some fixed set of rules. We will implement these rules to be probabilistic, so that the agents will either cooperate or defect with certain probabilities given the current situation. We will then need to be able to run simulations that compare the various agents' performances when played against each other. Then, we will evolve the best strategies, run a simulation of the game again, and againt evolve the newest best strategies, etc. At the end, we want to output the optimal strategy for the game.

Naturally, the most algorithmically interesting part of this project is the manner in which we mutate from one generation to the next. We will start with several variables that our Players are keeping track of. For instance they might keep track of what was played on the last move by their opponent, and their opponent's average cooperation rate. Each Player gives a fixed weight to each of these factors, and these weights combined with data from their opponent help determine the Player's next move.

In this spec, we did not describe the precise implementation of how the probability of cooperating is determined given what a player is keeping track of. Please see the "Implementation and Design" section in the paper for more details.

We will now describe how these Players will be mutated in our first version, although if we have time, we may try to mutate in a more interesting way in our later extensions. After one generation plays, the top 10% are chosen, and for each such player, one of their characteristics is changed uniformly at random to a value within 8% (or some similar constant) of its initial value. Five such mutated players are created and placed into the next round. This gives us 50% of the players in the next round, and they will be pit against a new set of other random players. This mutation process continues between every pair of rounds until we decide to stop the simulation.

We ended up playing around with the constants to mutate by. For instance, we often chose less than the top 10 percent, and we also output more than 50 percent of the next round's players so that the results would converge more quickly.

Cool extensions

There are many cool extensions we can look at from this project. One extension is to try various different types of genetic evolution. Instead of simple asexual mutations, we could consider sexual offspring, where we determine the next generation by combining favorable characteristics from two parents. We might take the average parameters of parents to produce offspring, or we might take some sort of weighted average, or even try to weight some aspects of certain parents more than others. An obvious further generalization of this is that we may produce offspring from more than just two parents, and in general try to pick the best subset of players to combine. This might give rise to an interesting data structure, as we would have to somehow rank all 2^n subsets of players, to give rise to the next generation.

In addition to the simple mutation described in the core features list, we ended up adding an evolution that takes in two parents and outputs a child whose characteristics are the average of the parents' characteristics. We also made an evolution module that outputs children whose characteristics are the weight average (weighted by score in the tournament) of their parents characteristics. We didn't end up using any more interesting data structures as described in the previous paragraph because we were running it on tournaments of size between 100 and 1000, on which the up front cost of introducing a more complex data structure would been bigger than any benefits due to its asymptotic running time.

A second cool extension to consider would be to examine how quickly a given parameter space converges to a solution, or if it converges at all. Our hope for the project is that it will converge, or else we won't really get a good playing solution. However, once we do have existence of a pretty good strategy, we can ask how close it gets to this stable strategy after n turns. We can also ask how many such stable strategies exist.

We found that both players who kept track of many moves, and those who just kept track of *one move converged. But those who kept track of only 1 move converged much more quickly.*

A third extension is an attempt to theoretically check the experimental results we obtain. For example, we might be able to use calculus of variations to see if the determined solutions lie at stationary points in our parameter space.

We didn't do this, since it seemed a bit off the track of computer science.

A fourth extension would be to ask which parameters would be useful to add. For example, it seems fairly likely that the opponent's last move should be a very important consideration in our current move. However, it's reasonable to ask if the opponents move 5 turns ago is any more important than their move 6 turns ago, and whether they matter at all. It would also be interesting to see if the stable states achieved from looking at the last 5 moves are the same (or at least very close) to those states given by incorporating only the last 4 moves.

We found that the most useful parameter was only what the opponent played on the last move, and keeping track of other parameters didn't seem to help much.

Technical Specification

We will modularize our project into three main portions, specifically, a Player module, a Tournament module, and an Evolution module.

Player Module

A Player is an agent in the Prisoner's dilemma that executes some strategy. Player p will store a number of attributes, including, but not limited to, the past k moves of that player $\{m_i^p\}$ with $1 \leq i \leq k$, the overall cooperation rate c^p, and the weights of the strategy $\{w_i^p\}$ and w_c^p. We will define these in the next few sentences. The strategy is implemented by a method that takes in another Player q as an argument (along with its attributes) and returns a move (cooperate or defect). Currently, we plan to implement this method as follows: we will take a linear combination of the opposing Player last k moves and overall cooperation rate, and output some probability α of cooperation. Then, we use a random number generator to output a "Cooperate" move with probabiliy, α, and a "Defect" move with probability, $1-\alpha$. For example, if we want to implement the Tit for Tat strategy, the method would use a linear combination with weight 1 for the opposing Player's last move and 0 for all other attributes. This way, we will just copy the opposing Player's last move.

We ended up implementing the cooperation probability with an exponential function, and so we needed weights to vary across all real numbers, not just $[0,1]$*. Because of this, the weights described above for the Tit-for-tat strategy above are not correct based on the way we actually implemented. A tit for tat strategy would having a very positive constant reflecting* *the opponents last move, which is much bigger than all the other constants.*

To store the last few moves, we will test two different data structures. We will either use an array, which will require shifting over all the indices when adding a move, or a linked list, which remedies this problem. It is difficult to say which is better *a priori* since we are not storing that many elements.

We ended up only using arrays to implement this. It is harder to find elements in linked lists, and easier to delete elements. Since we're trying to find elements a lot, and not delete them, we realized linked lists wouldn't be as efficient.

Tournament Module

A Tournament will run a single generation of iterative Prisoner's Dilemma. It will take in a set of Players and return the same set of Players with their corresponding scores. In each generation, we will run a series of rounds. In each round, we create a perfect pairing of our set of Players. Each pair will play against each other in n separate instances of Prisoner's Dilemma, where a single instance is defined as each Player choosing whether to "Cooperate" or "Defect" and recieiving the corresponding payout. The number of rounds will depend on tournament style and running time. Ideally, we will use a Round Robin tournament, but this may not be possible due to running time constraints.

We ended up not playing a round robin because of time constraints. Instead we just made about 100 random pairings per round.

Evolution Module

This module will take in the set of Players with corresponding scores returned by the Tournament module and return a new set of Players for the next generation of the tournament. More specifically, we want to *evolve* the players from one generation to another. There are a few analogs to features of biological reproduction that we want to incorporate: sexual reproduction and mutations. One idea we had is to take the top 10% of strategies and perform small mutations to them and perform some sort of transformation involving these three features. For example, we could come up with a way to combine pairs of Players into one Player with a "combined" strategy. An extension of this is to combine k Players into a single Player instead of just having 2-Player reproduction. We could also mutate some of the strategies by perturbing the weights by a small, random amount and normalizing. This process was described in more detail above in the "Core Features" and "Cool Extensions" sections.

Here are the preliminary signatures:

```
module Player:
    k weights
    last k moves
    cumulative cooperation rate
    cumulative score

    method returnMove: Player j -> Move
    method getMovesAndCooperationRate
    method setMoves
    method setCooperationRate
    helper method getMoveWithProbability

module Evolution(Set of Players):
    method evolve
    helper method mutate?: Player -> Player
    helper method reproduce?: Player 1, Player 2 -> Player

module Tournament(Set of Players):
    method runRounds: no parameters -> Set of Players
    helper method runRound
    helper method runGames
    helper method runGame: Player1, Player 2, # of games so far -> no return
    helper method pairPlayers
```

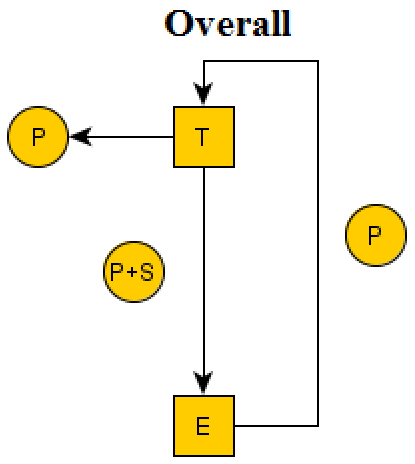

Figure 1: In this figure, T represents a tournament module while E represents the evolution module. The output of T, the players with scores (P+S), is the input to E. E then outputs the top players, P, which is then inputted into T again. At some point, we can stop the program and return the top Player, the optimal strategy.

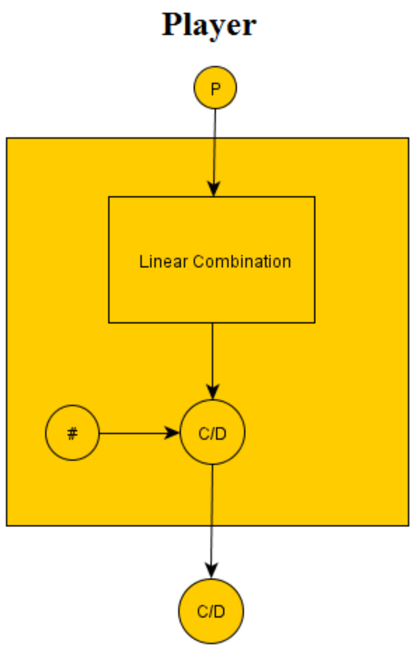

Figure 2: In this figure, the input P is the opponent player. We then take a linear combination using the attributes of the opponent and itself. We generate a random number # to determine the move, C/D.

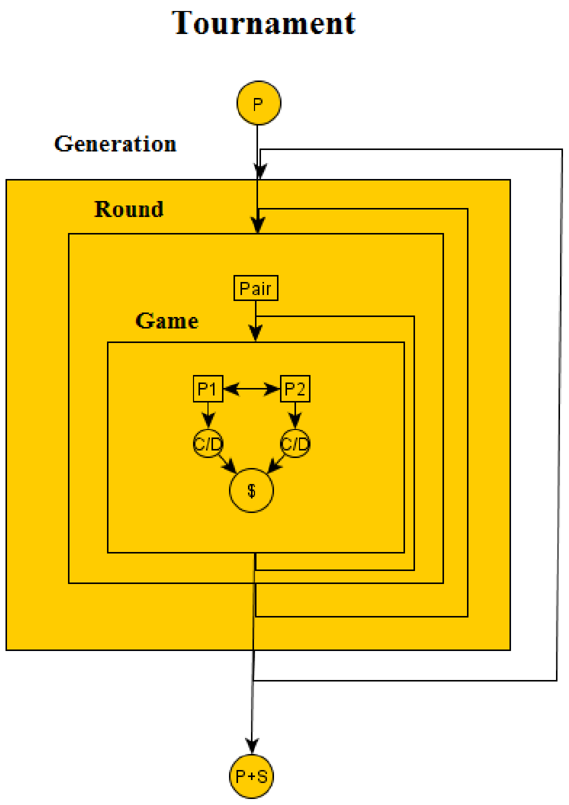

Figure 3: In this figure, the input P is a set of Players. The outer box is a generation, which consists of many rounds. In each round, we first split the set of Players into pairs. Then, for each pair, we run many games. In each game, we determine the moves for both Players, P1 and P2. The resulting moves are then used to generate the outcome and assign scores to the Players.

We also used this modular tournament system to easily implement another game called Blotto, as described in the final report.

Progress Report

We have decided to use Python for our project. At this moment, we have set up our IDE's and code repository on code.seas. We also have skeletons for the modules we are writing. We haven't had the time to completely flesh out their implementations, but each of us has a clear idea of what we need to write based on the preliminary signatures we provided above. In the next week, we'll be coding basic implementations of these modules (more detailed timeline below).

Our naturally modular approach makes our project very easy to divide into parts. The division will most likely be as follows: Albert will write the interfaces, Aaron will implement the Players, Ray will implement different types of Evolution, and Lucy will implement the Tournament.

Timeline

April 15-21

- Write basic code for each of our modules
- Test to make sure each module works on its own
- Run some preliminary tests using small numbers of Players
- Attempt to demonstrate some direction of convergence

April 22-28

- Run tests with larger numbers of Players
- See if we need to optimize some of our code
- Increase the number of parameters each Player takes into account
- Determine the number of parameters we should use for our final implementation
- Determine the convergence, if any
- Have all our core features completely implemented

April 29-May 5

- Experiment with different types of evolution
- Optimize running times, as this will likely be an issue
- Optimize for convergence time
- Consider using more advanced data structures if warranted

We ended up sticking to these schedules pretty well.