Becoming the Optimal Prisoner

Lucy Cheng, Aaron Landesman, Ray Qian, and Albert Wu

May 4, 2013

We originally made our video when the spec still said it needed to be 3-5 min. As per this Piazza post, we have shortened our video so that it is closer to 3 min than it is to 5. The new video for our project is here. The longer, original version of the video is here.

Overview

We implemented various genetic algorithms in order to find optimal randomized solutions to the iterated prisoners dilemma. We use *prisoners dilemma* to mean any two player game where either player can choose to cooperate or defect, and recieves some symmetric payouts based on their actions. That is, if both players defect, they receive the same payout, if they both cooperate they receive the same payout, and if one cooperates and the other defects, the cooperator will recieve a fixed payout and the defector will recieve a fixed payout. We use *iterated prisoners dilemma* to mean that two players will play multiple games against each other, and not just have a single encounter. In our project, we run multiple tournaments in which we have a group of players play iterated prisoner's dilemma with each other. We then take the players with the highest scores and evolve them to make better players which will be members of the next tournament. This idea of evolving players between tournaments is what we mean by using a *genetic algorithm*.

We implemented our genetic algorithm by making three modules. First, there is a module representing a player, that keeps track of the player's weights, the player's recent moves, and the player's score. Second, there is a module representing the tournament, that runs many games amongst players. Finally, there is an evolution module that tells the players how to evolve between generations.

Planning

We planned to apply genetic algorithms to Prisoner's Dilemma problem, as described in the overview. Here is our annotated technical spec and functional spec where annotations to the technical spec are made in italics. We did not annotate the functional spec separately because it is a strict subset of our technical spec.

For the most part, we implemented all the major aspects of both specs. Our project's original planning stayed largely the same; it worked out well, and we essentially hit our milestones perfectly. Our general interfaces for Player, Tournament, and Evolution stayed the same. However, some more specific things changed as we ran into issues, as described in the design and implementation section.

Design and Implementation

We found that our original modularization of this project into three files (player, tournament, and evolution) proved quite effective. In each file, we had several different implementations of the object, so that the main method would be able to play different types of players against eachother, and evolve the players in different ways. In order to test our code, we had the main method print out some of the leading players after every round, and we checked whether they were evolving the way we specified, as well as if the results made sense with the given payoff matrix.

The tournament module stayed essentially the same as we described in our specs. We had one small change in our evolution module, and a significant change in the way players determine their moves.

Evolution Module

In addition to simply mutating a single one of the player's weights, we experimented with different types of sexual selection after doing simple mutations. We took subsets of the best players and made them reproduce by averaging their attributes. An example of this is here. After that, we tweaked the algorithm to make a sexual reproduction that would be weighted more towards players that performed better. An example is here.

Player Module

In the specs, we didn't really explain how a player would determine their next move given the information they were storing and their weights. First, we changed the way we calculated our strategy from parameters. In the beginning, we intended to do the following.

$$p = \sum x_i m_i \tag{1}$$

where x_i are the parameters of the strategy, m_i are the opponent's previous moves, and p is the probability we would then cooperate. However, we discovered a few issues with this: (1) if all the opponents previous moves were defections, we would automatically have $p = 0$ and thus defect, which doesn't seem right since this probability should be able to vary, and (2) it is difficult to make sure our linear combination of m_i yields a probability between 0 and 1.

So, we made the following change. To fix problem (1) above, we added a fixed weight with input always 1. This basically adds a constant to our sum $\sum x_i m_i$ and allows it to take on any value for any input. To fix problem (2) above, we use ideas from logit regression, to guarantee that we have a probability between 0 and 1. We made the probability of cooperation be

$$\log \frac{p}{1-p} = x_0 + \sum x_i m_i$$

$$p = \frac{e^{x_0 + \sum x_i m_i}}{1 + e^{x_0 + \sum x_i m_i}}$$

We also defined specifically the order of the attributes stored. A player stores its opponent's overall cooperation percentage as the first attribute, a constant (1) as the second attribute, and the opponents' most recent moves are the third attribute and on, depending on the number of opponents' moves a player keeps in its memory.

Interface

We built a command-line interface specificed in the code/README file, which allows us to easily run different experiments for Players with different move memories, with different types of Evolution, with different payoff matrices, and with either the Prisoner's Dilemma or Blotto tournaments. More changes can be made to both the Evolution and Player move memories by changing the constants at the top of main.py.

Our output shows the progress of the genetic algorithm by outputting the optimal strategy after each generation. The format of the printed strategy differs for different players. For the single-move memory player, we explicitly calculate and print out the probability we will cooperate given that either the opponent cooperated or defected on the last move. For multiple-move memories, there are too many cases to print out all the cooperation probabilities, so we directly print out the list of weights for our strategy. Roughly, the magnitude of a weight indicates the significance of the corresponding attribute to the cooperation percentage.

Reflection

What we learned

We learned a lot more about the dynamics of the Prisoners' Dilemma. We were a bit skeptical about whether tit-for-tat would perform well against non-determinstic strategies, which motivated our thinking about genetic algorithms.

We found that, for our 1-move memory players, the convergent strategy was qualitatively similar to tit-for-tat, in that the best player was very likely to cooperate if the other player cooperated in its last turn. However, this player also seemed to cooperate around 70% of the time when the opposing player just defected as compared to 90% of the time when the opposing player just cooperated. The raw results are here, where the final strategy is at the bottom of the raw file and visualized in this image. This seems to suggest that cooperating sometimes even when the other person defects could be a good strategy.

For multiple-move memory players, the the convergent strategy indicated that of all the opponent's most recent moves, the most recent move was more important, i.e., the magnitude of the weight is biggest, as shown in the raw data. The strategy was pretty consistent qualitatively in that the player is more likely to cooperate if the opponent cooperated last rather than defected. This image visualizes the weights for the past opponent's most recent four moves.

We also ran mixed experiments with players of multiple-move memories and single-move memories. We ran this experiment once here and verified it here. We found that the strategies that performed best in our genetic algorithms were the players with single-move memories, suggesting that the later moves are less significant in terms of making cooperation decisions. This result is also supported by the literature, which indicates that players with multiple-move memories are no better off than players with single-move memories.

With regards to different types of evolution, we observed that the pure mutation approach (SimpleEvolution) seemed to lead to strategies that converged more readily than both of the sexual reproduction approaches (SimpleSex and ComplexSex). For the standard payoff matrix (4, 5/0, 1), only the SimpleEvolution led to convergence for the NMovePlayer and SimplePlayer. After 100 iterations, there was no evidence of any convergence in the "sexual" strategies. For comparison, we present the raw SimplePlayer results for SimpleSex and ComplexSex. Here are the raw NMovePlayer results for SimpleSex and ComplexSex.

Modularity, Systems, and Testing

We spent a lot of time on our design and making sure our interfaces were completely modular. Nevertheless, though our basic idea worked out well, we should have more explicitly written all of our interfaces and even pseudocode, which would allow us to think through our algorithm completely before beginning the implementation.

We realized that the modularity of our files lent themselves to implementing many other randomized games. We used this modularity to easily implement a seemingly different game, called Blotto. In this game, each player has 100 soliders that can be distributed among 10 castles. They each win the castles where they have more soldiers than their opponent, and the person who wins the most castles wins the round. In order to implement this game, we only had to add two small classes, each about 10 lines long in python. Our ease in implementing this game in our system showed how extensible our abstractions were for any tournament-style game, a proof-of-concept for our interfaces.

We also ran our Blotto experiment not once, not twice, but three times! All the strategies have half or a little bit over half of their castles with under ten soliders. This means distributing 10 soldiers into each castle would beat the computer's best strategy. It seems that Blotto is not a good application of the genetic algorithm because it depends too much on human psychology, which is not captured by our random starting conditions and evolution.

In terms of system improvements, we tried out parallelism for each round within each generation, except it did not improve our running time. The reason for this is that we did not think about Amdahl's Law: each generation was quite fast (0.1 seconds each), so the time to setup new threads was longer than the time to run sequential code. Furthermore, we could not parallelize the generations (since players evolve from generation to generation). Therefore, even when we parallelized within a generation, we could not improve overall running time.

It was a bit tough to test our project since we do not know the right answers, but we ended up testing based on how plausible our results were for specific payoff matrices. When we had the entire payoff matrix filled with 0 except

for 1 if they both defect, the best players ended up pretty much always defecting (cooperate about 15% of the time overall) when we ran genetic evolution for SimplePlayer.

Group Contributions

Our contributions pretty much went as planned. Albert worked on the three interfaces, worked on the main procedure, implemented BlottoEvolution, and provided a lot of high-level thinking about abstractions and representations of the players' strategies. Ray researched and came up with the idea of using genetic evolution, implemented most of the Evolution module, and found literature that supported our results. Lucy implemented the Tournament module and helped think through the flow of the entire program. Aaron implemented the Player module and provided a lot of high-level thinking about the players' strategies.

Future Work

There are two extensions we might implement if we had more time. First, we would experiment with different types of genetic recombination. For example, we might try crossing over. This would require us to rethink our interface a bit. We would need Player to store its two parents' attributes and then pass a recombination of the two parents' attributes to its "child." Second, another item to improve would be our initial distribution. Currently, the weights can be arbitrary real numbers, but are randomly generated in the interval $[-1, 1]$. While this accounts for most of the possible situations, it makes it hard to evolve to a point where a player will cooperate with probabilities very close to 1 or 0. To rectify this, if we were to redo the project, we might alter the evolution function to allow for this, and force the weights to lie in the range $[0, 1]$.

Advice for future students

For future students implementing a similar project, we cannot stress enough the importance of fully writing down the complete interfaces and pseudocode. We learned through this project that explicitly writing down interfaces and pseudocode would have helped sort out logical problems before starting our implementation. Also, we would read more papers before beginning our project, so that we would have a better idea of all the extensions we would want to do before starting.