Repeated games with mistakes

Nikolas M. Skoufis
Supervisor: Julian Garcia

September 11th, 2014

Objectives

The objectives of this project are to:

- study the expected payoff structure of repeated games with mistakes using the prisoner's dilemma as a model game.
- design an algorithm for efficiently computing the expected payoff of each player in the iterated prisoner's dilemma with mistakes.
- learn about tools to speed up numerical calculations in Python such as profiling and multiprocessing.
- study strategies that 'count' particular outcomes and how they perform in the iterated prisoner's dilemma with mistakes.

Progress

My work in this project so far has been focussed on laying the foundations for studying the repeated prisoner's dilemma with mistakes. To this end I have implemented a `Strategy` class which implements basic functionality of a strategy such as tracking the player's history and computing the next move for the player. This `Strategy` class is never instantiated directly, but is instead subclassed by concrete implementations of strategies ie. `AllC`, `AllD`, `TitForTat` etc. This implementation should allow for any strategy to be implemented (including non-deterministic strategies) since the method that computes the next move for the strategy knows about both the player's own history as well as being passed the opponent's history.

Once the strategy implementation and signature was finalised, I was able to implement methods that would compute the expected payoff when different strategies play each other. Initially I considered the scenario without mistakes. This resulted in two different methods for calculating this expected payoff: a method based on an infinite sum (as described in the appendix of [1]) and a method based on Monte Carlo simulation. Although this project is primarily concerned with the infinite sum method, it is useful to have a working Monte Carlo simulation for approximating these values. This Monte Carlo simulation becomes useful when we stray outside the simple combinations of strategies that have been considered so far and into more complex strategies or combinations which lack closed form solutions against which we can check our caclulations.

In order to verify that the implementations of these methods were correct, eight simple strategies were implemented as subclasses of the `Strategy` class. These strategies represent the eight deterministic strategies that only track the opponent's last move. The results from the infinite sum (herin the calculation method), the Monte Carlo simulation and the exact closed-form results (as described in [1] were then compared over the 64 possible strategy combinations as a method of verification. The Monte Carlo method offers a fixed trials mode where the number of trials is predetermined. Alternatively if the user wishes to guarantee the distribution of their compute Monte Carlo result (as described in [3]), the trials are repeated until the desired standard deviation is achieved. As an added level of verification, I utilised the *Hypothesis* [2] library in order to systematically check possible edge cases in both the calculation methods as well as the strategy framework.

Once these methods had been implemented, the next step was to add mistakes. Implementing mistakes in the Monte Carlo method is relatively straight-forward, and I have also succeeded in parallelising this code. The parallel implementation of the Monte Carlo method distributes the desired number of trials across the

available cores, speeding up execution greatly. Adding mistakes to the calculation method has proved more difficult, since the number of terms in the sum grows exponentially. In order to enumerate and add all of the terms in this infinite sum, I have designed the following algorithm:

Initialize Create a FIFO queue, and add the entry (1, ", "). This represents a coefficient of 1, and two empty histories.

1 Remove the first item from the queue. Take the histories and initialise the players to have those histories.

2 Compute the next move that would occur for each player, given the past histories.

3 For each of the possible sets of mistakes (no mistake, one mistake two different ways, two mistakes), compute the probability of this occurring . We can compute this by multiplying the probability of the event occurring in this round (ie.the continuation probability δ multiplied by the appropriate mistake probability) by the coefficient in the data we removed from the queue.

4 For each possible set of mistakes, compute the payoff. We will need to multiply the coefficient compute in step 3 by the value in the payoff matrix for this particular move combination.

5 For each of the terms computed in step 4, if the term is larger than some threshold (a chosen parameter which determines when we begin to truncate the sum), we add a new entry to the queue which consists of the coefficient from step 3, and the previous history of each player along with the next move computed in step 1, modifying the move based on mistakes.

6 Go to step 1 and repeat

Eventually all the terms will be below the threshold, the queue will empty and the sum will terminate. The advantages of this method over a simple sum are:

- Since we throw away terms when they become too small on a term-by-term basis, we greatly reduce the number of terms computed compared with an algorithm which simply computes terms to a given number of moves. In effect we are pruning unlikely paths on the outcome tree as soon as their likelihoods become small.
- This algorithm should be easy to parallelize since it makes use of the common 'multi-producer, multi-consumer' pattern via the queue.

References

[1] Julian Garcia, Arne Traulsen, *The Structure of Mutations and the Evolution of Cooperation*, PLoS ONE, Vol. 7, No. 4, 2012.

[2] David R. MacIver, *Hypothesis*, https://github.com/DRMacIver/hypothesis, 2015.

[3] Sheldon M. Ross, *Simulation*, Fourth edition, Elsevier Academic Press, 2006.