Final Year Project

Toby Devlin

November 19, 2017

Contents

Chapter 1

Introduction

General discussion on what game theory is and what the PD is. What this topic is and how it fits into the big picture.

The Prisoners Dilemma is a classic game theory topic... Its important to game theory because... creating general sequences to strategies is important because... models in the real world that follow certain strategies... how we can leverage these to peruse goals.

1.1 Background

fill this section with background on game theory in when I do the course?

1.1.1 Iterated Prisoners Dilemma

The Prisoners Dilemma is a well known game theory problem based on the example of a pair of prisoners and their subsequent interrogation. The game is as follows:

Something about the PD

The single game itself is very basic and is modelled in the following way:

$$give - a - model - here$$

The Iterated Prisoners Dilemma is the iterated version of the Prisoners Dilemma[1]. The iteration of the game is what makes the game an interesting concept, as now **learn the technical stuff and put it here!!!!!** we are able to create strategies[2] that look to gain an upper hand based on **Something here**

[1] reference this stuff dude, come on..

[2] When referring to ourselves, we will describe our moves as a strategy. When referring to

1.1.2 Machine Learning Concepts

This section will briefly provide a background to machine learning algorithms implemented in the axelrod-dojo[3]. This is by no means a comprehensive look into these subjects but will provide sufficient background on technical discussion later on.

Genetic Algorithms

Genetic Algorithms are a description of techniques for generating solutions complex problems such as searching and, in our case, optimization[4]. The basis of a genetic algorithm is focused on a cycle of evolution. Like nature, we create a survival of the fittest concept[5] to evaluate a population, kill off the weakest members and create offspring from the most successful population.

Put a figure of the cycle here.

Initially we create a heuristic function, say our fitness function, which is a measure of how successful a candidate in our population is. Then we run our whole population through this function, ranking each one by how successful their score is. At this point we can create a cut off[6] to decide which of the population not to put through to the next round.

Bayesian Optimization

1.2 Brief Overview

In this document I will be be looking at the creation of sequences to beat given players in The Iterated Prisoners Dilemma[7]. My research looked into just the single opponent use case, but the idea of designing a sequence for a given number of opponents is looked at in the further study of the report. This task is the Problem:

> Given a certain opponent, O, (with a provided strategy, S) what is the best possible sequence of moves, in a game of n turns, made by my strategy to maximise my players score?

---

an opponent we can use the term opponent and strategy interchangeably.

[3] referencing opportunity here

[4] Mitchell, Melanie (1996). An Introduction to Genetic Algorithms. Cambridge, MA: MIT Press. ISBN 9780585030944. learn to reference soon

[5] need to reference Darwin?

[6] Can often be referred to as the bottleneck

[7] Reference this for some background

Chapter 2

Literature Review

2.1 Background

- book1
- book2

2.2 Strategies Of Interest

2.2.1 Tit for Tat

this is a sentence which has been changed[**?**]

2.2.2 Cycler

this is a type of opponent

2.2.3 Other

add at least two here

Chapter 3

Approach To Problems

3.1 In Depth Definition Of Task

3.2 Solution Form

The sequence archetype will use the $Cycler()$ player for our strategy each time, only editing the input to improve our score. To this model we can apply an optimised input of length 200 to the player, this sequence, as per the design of the strategy will then be repeated until the games end (if n = length of game, we are just calculating the sequence for the whole game).

The input sequence itself will be created using a genetic optimisation. Starting with a set of randomly generated sequences, we will have each one play the opponent and return with a score. These sequences will be ranked and the lowest x% will be discarded, resulting in a fitter, but smaller, population than before. This smaller population will then create offspring using a —X TBD method X— pairing algorithm before mutating with —X TBD method X—. This new set of offspring will be included in the next scoring round and the process repeats for k number of rounds

This sequence of Play-Rank-Create-LOOP will be the basis of creating the optimal strategy for each other opponent.

3.3 initial research

This will look into how changes in the initial population size for the genetic algorithm effect the number of generations until convergence. We will describe a series of sequences as 'converged' if the best score over the set number of generations has reached a stable point. This stable point may not be the optimal point, we may have found a local maxima for the solution sequence rather than

the global maximum.

The opponents we select are interesting, they are all 'simple' and can be explained in a very brief sentence or two. We will look into these as we can confirm that the genetic algorithm will select the optimal sequence solution for the selected opponent.

This bit should be on how we created the data. The data we have produced can be read from the csv, and as long as we provide some names the data is easily accessible.

What we want to look at is how the best score rises as we change certain features of the algorithm. Once the best score hits a maximum such that it wont change no mater how many more generations are run (designated as an optimal solution sequence) or it hits a score which doesn't change over a significant period of generations (solution sequence) we can say that the sequence has converged. The generation number where this plateau occurs is called the solution sequence distance, or solution distance; minimising this is that aim of this investigation. We may find solutions that are not optimal, meaning that the algorithm will narrow down a sequence that will do well against an opponent but wont actually find the best sequence that we can possibly get. These are called local maximums and we will look into how to overcome the possibility of our algorithm finding a local, rather than, global maximum later. If we make each sequence more likely to mutate generation to generation what will happen? If we have a larger initial population sample to start with, will we reach our maximum best score earlier? What about increasing the generations, what is the optimal number of generations to run the algorithm for such that we always find a solution sequence. the selection of opponents for this section is such that we are able to know their optimal solution sequence as to identify local maximum solution sequences.

3.3.1 Initial Population Size

The initial population size defines the range of scores that we can achieve against an opponent in any given generation. Because of this it is assumed the larger a population the larger chance of finding a solution sequence. This, however, will have an impact on computation time; each generation must process the full population, this is a linier relationship, $O(n)$, on computation so we would need to find a higher order of decrease in out solution distance for this to be a viable option.
The concept of population increase is simple, all we need to look for is the solution distance as we increase our population: $[25, 50, 100, 150, 200, 250, 500]$

$def populationChecker(opponent) : \#makeanicefilenamefile_name = "data/" + str(opponent).replace("",$

Its clear that from the graph that the initial population size has a huge effect on finding the Tit For Tat solution sequence. It can also be seen from the table that as we increase the population our best score statistics go up across the board. Looking into alternator we will see the same.

Analysis of initial population sizes

As the figures above shows there is an obvious trend that the higher the overall average score. What is interesting is that the result is non linier, the best score difference for a population of 50 to a population of 100 is huge in comparison to the same relative increase from 200 to 250.

Another interesting point is that none of these results have found a solution sequence (or at least we cant tell from the graph). There are no large plateaus for the graph, and so perhaps using a medium size population, say 150, and increasing the generations will have better results for a solution sequence. 150 is selected because there it find the same scoring solution as the others, but takes less time

This is interesting, the random results tend to fare better with larger population, this may just be based on the probability that there is a sequence in the population that does better against the specific random sequence. This may also be the genetic algorithm optimising itself to the pseudorandom generator, building the $Random()$ opponent solution for next generation, this could be a reason as all of the final generation best performers are scoring higher than the original starting population.

3.3.2 Generation Length Analysis

Here we will look into how close to a solution sequence we get when we increase the generations the algorithm runs for. We will be using a population of 150, as this was the best average for score vs computation time

$$generation_list = [50, 150, 250, 350, 450, 500]$$

$def generationSizeChecker(opponent) : file_name = "data/"+str(opponent).replace("","_").replace(" : ",$

Generation size analysis is a bit different as we want to see what length of generation will provide us with the largest change in score from first iteration to last, but we don't want a huge calculation time. If we look at the scatter of mean best score difference against the number of generations we can observe how, on average, the number of generations effects the overall change in our best score.

This is is showing that for both Alternator and Tit For Tat opponents, the mean increase per generation is declining. On this result we can conclude as we increase generations there is less and less benefit per generation. From these plots it can be concluded that running 150 generations has the best time vs mean best score increase per generation. However in this investigation we will want to find the absolute optimal, so an indefinite generation length would be preferable

Now we can look at proximity the solution sequence once the analysis has concluded, this will provide details on what the solution looks like as we extend our generation length. Obviously we want to find the absolute solution sequence, if it exists, every time; the reason for looing at length is so that we can observe what the average number of generations is required to find the solution sequence.

After 250 generations we seem to have reached the solution state for our opponents Tit for tat and alternator but not for grudger. From the combination of the two plots, having more generations means that there is, on average, less of an improvement per generation but the longer analysis is required to find the solution sequence for an opponent. From now on, 250 is the number of generations we will use to find out solution sequence.

A quick note on Grudger It appears that the grudger opponent is being optimised into becoming 2 sections of opposing moves. if we look at the start and end of a generation set we can see that the genetic algorithm is trying to remove Cs after the defect point and add Cs before the defect point. This solution is due to the fact a good solution will have found that ending in lots off defections is good, and as it progressed there isn't a chance to observe an ending of cooperation moves. This is a clear sign of the algorithm locating a local maximum, looking into the mutation rate might provide a reason why this is happening.

Grudger best start:$[C, C, C, C, C, C, D, D, D, D, D, D, D, C, C, D, C, C, D, C, C, C, C, C, C, D, C, C, D, D,$

Grudger best end:$[C, C, C, C, C, C, C, C, C, C, C, C, C, C, C, C, C, C, C, C, C, C, D, D, D, D, D, D, D, D, D$

For the some of the most 'simple' opponents 250 generations seems reasonable to reach a solution sequence as shown in the Alternator and Tit For Tat. The complexities lie with mitigating local maximums during the generations.

3.3.3 Changeing Our Mutation Rate

By changing the rate at which we mutate our genes within a sequence, we might be able to more effectively narrow in on a solution sequence, i.e. One that has converged. The default is set to 0.1, which means, for every 10 members of our population that continue into the next generation one of these has a single gene, or action, that is changed in its sequence.
Here we will look at 2 different concepts:

- Is changing one or more actions of a members' sequence the best way of mutating a candidate? (More potent mutation)
- Is it beneficial for more/less than 1 in 10 members to be mutated generation to generation? (More frequent mutation)

These are two separate questions, so first we will look at increasing the potency of our mutation. Then, once we have found some information out on this, we can look into the frequency of our mutations with the new potency as a permanent setting. This approach allows for an $O(1)$ factor of scaling making this a great candidate for an approach to reduce our solution sequence distance compared with other approaches, for example increasing the population size.

Changing Mutation Potency

$$mutation_potency_list = [1, 2, 3, 5, 10, 15, 20]$$

$def mutationPotencyChecker(opponent) : file_name = "data/"+str(opponent).replace("","_").replace(" :$

This single graph shows no clear benefit from increasing the mutation potency. We can see that having changed 15 genes in our sequence each time we are still not improving our score as much as using 1. This may be down to chance (and if the file is regenerated this may disappear) looking at more opponents than just grudger we may find a clear improvement. Now we can look at how our best score is effected against other opponents.

From these graphs there is no clear benefit to increasing the mutation potency to affect the overall best score value against an opponent. If we instead look at what our average increase of score per mutation is we may observe a useful result.
From further analysis there is not much of an improvement by increasing your mutation potency with regards to score or for average increase of score. The increase in mean best score difference is not substantial and could be down to chance

Changing Mutation Frequency

[

$$mutation_frequency_list = [0.1, 0.15, 0.2, 0.25]$$

$def mutationFrequencyChecker(opponent) : file_name = "data/"+str(opponent).replace("","_").replace("$

The results on changing the mutation rate don't obviously effect that generations until convergence from this overview. There is an interesting result that can be seen on the grudger plot; the algorithm has found 2 different maximums.

From this there we can see that the mutation frequency of 0.1 and 0.2 produced higher scoring solutions than the other mutation frequencies. This shows that we have found 3 different solution sequences (in freqs .15, .2, .25) with the solution for .1 continuing to improve as the generations ended.

As an additional point, we can also observe that increasing the mutation frequency means that there is less variation in the best scoring sequences. (TODO: Does this have an impact on escaping local maximums?)

3.3.4 Using crossover and mutation to remove local maximum solutions

The occurrence of local maximums is something that has only occurred for the Grudger opponent so far. The difference between the Grudger and the other opponents were looking at is that the Grudger has a singularity where its behaviour changes. The change in behaviour is not uncommon, Tit For Tat works in the same way, however this algorithm has managed to identify its behaviour and adapt to overcome its negative effects.

Grudger and Tit For Tat differ in their responsiveness in two ways:

- Grudger never changes its mind. There is one change in behaviour for the entire game. Unlike Tit For Tat, this means that the algorithm only has a single opportunity to observe this once every per population per generation meaning the behaviour is much less frequently observed.
- The Grudger also only does this change once no matter the games length. This means that the genetic algorithm picks up the effect of this choice as early in the match as the first defection in its random sequence. A random start of C and Ds puts the likeyhood of at least 1 defection occurring in the first 10 moves at 99.99; this means our algorithm will, most likely, always encounter this grudging effect within the first 10 moves and will never score the full 600 points. (see below)

Below are two totality games, one of all Cs and one of all Ds. These are edge cases and would be incredibly rarely encountered as a starting point in the initial population. Because of this the algorithm has to shuffle towards the potential benefit of using these totalities rather than start with analysing them, and in our case the algorithm will probably first encounter the Grudging effect of out opponent before trying out [CCC..C] or [CCC..D] and so will probably never find the highest scoring solution.

$$players = (axl.Grudger(), axl.Cycler("C"))match = axl.Match(players, 200)match.play()print(match.f$$

$$players = (axl.Grudger(), axl.Cycler("D"))match = axl.Match(players, 200)match.play()print(match.f$$

Strangely, our solution sequence is set to find where we have the objective of "score" (see objective statement) which actually tries to improve the score per turn ($axelrod_dojo\ utils.py$: 67); it should be converging on a totality of Cs rather than what its doing by finding the totality of Ds. This is probably because the algorithm initially limits its best score per turn once the first generation is complete and a cut-off has been established for each of the initial population. The crossover method between generations then doesn't provide enough of a mix up to allow the algorithm to escape the local minimum by switching a subsection with a sufficiently different potentially better subsection. Then when it comes to mutating, there is little any number of mutations can do to drastically change large sections of the sequence without having a huge effect on the score.

This then sheds light on the path the algorithm takes to find a solution. If we are to find the optimal solution, we must take a crossover and mutation path which doesn't cut off better paths as we work our way towards a solution; this is much easier said than put into practice due to the way the algorithm "cuts off paths". If we reverse this thinking and try to alter our crossover design and mutation rate such that instead of "cutting off" a path we are able to "build" new ones. We can re-design the crossover to switch up large subsections of the sequence then allow the mutations to optimise these sub-sequences.

currently we have the following design:

$$def crossover_old(self, other_cycler) : \#boring single point crossover : crossover_point = int(self.get_sequence$$

We want to allow the crossover to have more of an impact than just halving the sequence and optimizing each section. i.e. go from:
$|----------|and|++++++++++|$
$++++++|= |----------++++++++++|$ to, say:
$|----------|and|++++++++++|$
$++++++|= |--++--+--++--++--+--++|$
This will allow the mutation rate to edit the subsections in a more interlaced manner, hopefully overcoming the pitfalls of sparse mutations to escape local maximums. our new crossover method is as follows:

$$def crossover(self, other_cycler) : \#10 crossover points : step_size = int(len(self.get_sequence())/10)\#empt$$

Below is an exapmple of the new crossover sequence:

$$seq1 = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]seq2 = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]step$$

now we can look at how this new crossover algorithm works with the default mutation (freq=.1 and pot=1) to improve our local maximums with the grudger opponent:

3.3.5 Altering Initial Population

This section is to display the results of working on an initial population that contain common solution sequences. We will discuss entropy of a solution and that by creating "neat" starting points we can decide where to start on the plane, mitigating potential sub-optimal solutions.
From the conclusions in previous sections the problem with creating

3.4 Conclusion of approach

Chapter 4

Implementation Of Sequence Discovery

Chapter 5

Results and Discussion

Chapter 6

Practical Applications for Solution Sequences

Chapter 7

Summary and Future Research

Bibliography