Final Year Project

Toby Devlin

November 30, 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.

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 pairings% 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

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

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

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?

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[A$^+$87]

2.2.2 Cycler

this is a type of opponent

2.2.3 Other

add at least two here

Chapter 3

Approach To The Problem

3.1 In Depth Definition Of Task

What is the task? *When playing an Iterated Prisoners Dilemma strategy what is the best ordered series of moves to play such for us to obtain the highest possible average score per move across the game.//*
Section questions:
What is the motivation for the task?
What are we expecting to come from the task?
Terminology?

In this chapter, consideration will be given to finding the optimal sequence of moves against another player. The various approaches used and a detailed analysis of the optimisation procedures and parameters will be described.

3.1.1 Notation

Let S be a sequence composed of move elements: $S = X_1X_2 \dots X_L$.

- $X_i = C$ is a cooperation type move element
- $X_i = D$ is a defection type move element.
- X_i denotes the ith move element in a sequence, $X_i \in \{C, D\}$.
- L denotes the length of a sequence; $L = 200$ is used throughout this report.

We can split up sequences into blocks of consecutive move elements of the same type. We will use B_i to denote the block after i changes of move type from the explicitly stated starting move type.

- Every move in a block is of the same type; the type is implicit based on whether i is even and what the starting move type was.

- We can use the notation $|B_i|$ to denote the length of the ith block in the sequence. $|B_i| \in \mathbb{Z}$

This means we can write a sequence as a series of blocks:

$$S = B_1 B_2, \ldots, B_n$$

A Sequence can also be defined shorthand by specifying the starting move and the length of subsequent blocks:

$$S = C : \left[\ |B_1|, |B_2|, \ldots, |B_n|\ \right] \quad \Rightarrow \quad S = \overbrace{C \ldots C}^{|B_1|}\overbrace{D \ldots D}^{|B_2|} \ldots \overbrace{(C|D) \ldots (C|D)}^{|B_n|}$$

We can also construct sequences from repetitions of a sequence of blocks when it makes sense:

$$C : \left[\ \left(|B_1|, |B_2|, \ldots, |B_m|\right)^k\ \right] \quad \Rightarrow \quad \overbrace{C \ldots C}^{|B_1|}\overbrace{D \ldots D}^{|B_2|} \ldots \overbrace{D \ldots D}^{|B_m|}{}^{k-times}$$

The two notations can be combined to add starting and ending blocks to a repeating sequence (shown in the examples).

It is also possible to define sets of sequences by adding variables to parameters of the sequence. Appropriate selection of parameters mean the length of sequences shouldn't grow.

$$\{C : [i, l-i]\} \quad i \in [a, b] \Rightarrow \{\underbrace{C \ldots C}_{a}\overbrace{D \ldots D}^{l-a}, \underbrace{C \ldots C}_{a+1}\overbrace{D \ldots D}^{l-(a+1)}, \ldots, \underbrace{C \ldots C}_{b}\overbrace{D \ldots D}^{l-b}\}$$

For long sequences where there is no recognisable pattern the wildcard $*^j$ may be used. This is to signify that j consecutive move elements have no pattern and are randomly distributed. We will mainly use this when referring to sections of a sequence that would otherwise be too lengthy to write out.
Examples:

$$C : [1, 4, 3, 2] = CDDDDCCCDD$$

$$D : [\ (1, 1)^5] = DCDCDCDCDC$$

$$C : [1, (2, 1)^2, 2, 1] = CDDCDDCDDC$$

$$\{D : [i, 5 - i]\} \quad i \in [2, 4] = \{DDCCC,\ DDDCC,\ DDDCC\}$$

3.2 Solution Form

Our algorithm will, after its generations are concluded, an arbitrary sequence of the form:

$$X_1, X_2, X_3, \ldots, X_n \text{ where } X_i \in C, D$$

This sequence will be represent what moves we should play against the opponent to get our largest potential score per turn. In this notation a C represents a cooperation move and a D represents a defection move.

We are looking for sequences that will allow us to maximise our score overall, rather than just beating any given opponent. A nice analogy of this concept is a team playing a football tournament, but instead of a knockout competition our team is placed in the standings based off the total goals they have scored across the tournament. More applications of these results are discussed in Chapter 6.

3.3 Finding Solutions

The sequence archetype will use the $Cycler()$ player for our strategy each time, only editing the input parameter to improve our score against an opponent. 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, see Section 1.1.2 for in depth explanations.

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

3.4 Initial Research

Before conducting the bulk calculations for the set of opponents listed in the appendix we will test the algorithms' parameters to see what the best settings are for finding solution sequences. We will describe a series of sequences as 'converged' if the best score over the set number of generations has reached a stable point; described as when none of the moves in the sequences has changed over a number of generations. This stable point, however, may not be the optimal solution. we may have found a local maxima for the solution sequence rather than the global maximum.

The opponents we select are in some way interesting. They are all 'simple' and can be explained in a very brief sentence or two, but each one has a fundamentally different structure to how they work. We will look into these as we can confirm that the genetic algorithm will select the optimal sequence solution for the selected opponent.

Player	Optimal Sequence
axl.TitForTat()	$CCC...CD$
axl.Alternator()	$DDD...DD$
axl.Grudger()	$CCC...CD$
axl.Random()	$DDD...DD$
axl.EvolvedFSM16()	need to find
axl.CollectiveStrategy()	need to find

What we want to look at is how the best score rises over generations as we change certain features of the algorithm. Once the best score per turn hits a maximum such that it wont change no mater how many more generations are run; as described in Section 3.3 this the optimal solution sequence and it is unique (probably, I will have to prove this first[1]).

Once a solution has been found the generation number where this plateau occurs is called the solution sequence distance, or solution distance; one of the goals of this initial investigation is to see how parameters affect the distance. During the investigation we may find solutions that are not optimal, meaning that the algorithm will have found a sequence that will do well against an opponent but wont find the best sequence that will return can possibly get. These sub optimal solutions are due to the occurrence of local maxima in the set of scores of neighbouring sequences. Genetic algorithms are designed in way to avoid local maxima's; the property of mutating (i.e jumps of their features) allow members of the population to potentially remove themselves from these local maximas. We will look in depth into how to overcome the possibility of our algorithm finding a local, rather than, global maximum in Section 3.4.4. Some of the questions we will hope to be answering include:

- If we have a larger initial population sample to start with, will we reach our maximum best score earlier?
- What about increasing the generations, is there an optimal number of generations to run the algorithm for such that we always find a solution sequence.
- If we make each sequence more likely to mutate generation to generation what will happen? What about increasing how potent our mutations are?

3.4.1 Changing Initial Population Size

The initial population size is the number of starting sequences we use in our algorithms first generation. Once this generation concludes the population will go through the series of phases outlined in figure **??**; altering the population to keep the best performers against our opponent to continue on to subsequent generations. During any given generation the population defines the maximum

[1]Proof of a unique solution sequence for an opponent is out of scope

potential range of scores that we can achieve against our opponent. For example, having 2 members with distinct sequences in our population would provide us with 2 distinct Because of this we can reasonably assume the larger our population the larger the number of distinct scores leading to a larger chance of finding the solution sequence with the optimal score; hence we should converge to the solution sequence in less generations.

The implementation of analysing a range of populations requires us to understand how the solution distance is affected as we run our algorithm through a set of population sizes, say $p \in [25, 50, 100, 150, 200, 250, 500]$.

EFFICIENCY NOTE: Increasing the size of our population will have an impact on computation time; each generation must process the full population in a linear fashion causing a computation overhead of $O(n)$. For an increase to be useful a in any time restricted scenario our algorithm would need to show a higher order benefit in our distance to convergence, or in our average score per turn. However We are not working in a time restricted scenario, and so we should just select the best overall initial population size independent of computation overhead. In a perfect world where everything was time independent we would brute force every possible sequence combination

The code in Snippet 3.1 is an implementation how we go about analysing and storing the tests on generation sizes listed. It leverages the use of the function 'runGeneticAlgo' show in appendix Snippet **??**

This output will provide us with a table with the following form:

best score	gen.	mean score	population	sequence	std dev.	time taken
2.425	1	2.264	**25.0**	DD...	0.067	6.646
2.425	2	2.343	**25.0**	DD...	0.046	6.646
2.425	3	2.393	**25.0**	DD...	0.038	6.646
...	...	...	...	...	...	...
2.830	102	2.782	**100.0**	CC...	0.112	28.425
...	...	...	...	...	...	...
2.980	150	2.911	**500.0**	CC...	0.158	152.684
...	...	...	...	...	...	...

By grouping this data by the population we observe how initial populations affect different opponents. Its clear that from figure 3.2 that the initial population size has a significant effect on finding better sequences. We can see if there is a larger initial population there is typically a higher best score shown once concluding all of the generations. This can also be shown in figure 3.3 that... It doesn't, however, ensure that we find the solution sequence; as is shown in the lack of long plateaus in the lines.

```
def populationChecker(opponent):
# make a nice file name
file_name = "data/" + str(opponent).replace(" ", "_")
                                   .replace(":", "_")
                                   .lower()
                    + "_pop.csv"
# if the file exists don't run, it takes forever, make sure it exists
if not os.path.isfile(file_name):
    df_main = pd.DataFrame(data=None, columns=col_names)
    for pop_size in populations:
        start_time = time.clock()
        pop_run = runGeneticAlgo(opponent,
                                 population_size=pop_size,
                                 number_of_game_turns=200,
                                 cycle_length=200,
                                 generations=150,
                                 mutation_probability=0.1,
                                 reset_file=True)
        end_time = time.clock()
        tmp_df = pd.read_csv(pop_run[0], names=col_names)
        tmp_df["population"] = pop_size
        tmp_df["time_taken"] = end_time - start_time
        df_main = df_main.append(tmp_df, ignore_index=True)
    df_main.to_csv(file_name)
    print("List Complete:", file_name)
    return df_main
else:
    print("file already exists, no calcs to do.")
    file_df = pd.read_csv(file_name)
    # remove first column
    file_df = file_df[list(file_df)[1:]]
    return file_df
```

Figure 3.1: code to check multiple populations

Figure 3.2: Best score per turn vs generation for different initial population sizes

Figure 3.3: Scatter of max best score vs different initial populations

The improvement's from this effect are non-linier from observation. The change in overall final best score for a population of 50 compared with a population of 100 is huge in comparison to the same relative increase from 200 to 250. This may suggest there are more effective approaches to improving our score after a certain size of initial population than to continuing to increase it further.

None of these results have found a solution sequence (or at least we cant tell from the graph). It is clear that larger initial populations do, on a relative scale, much better than small ones. There are no large plateaus for the graph, so as we continue our research the initial population size will be increased to 150 to keep computation times manageable.

3.4.2 Generation Length Analysis

Another major component parameter of a genetic algorithm is the number of generations it will run before outputting a final sequence. The number of generations has an influence on a number of different things within the algorithm:

- The total combinations of features[2] (sequence elements) that the algorithm can test.
- The number of low performers we can remove in our population.

For our goal of finding the optimal solution sequence for each opponent it would be useful to extend the generations as far as possible; this would provide the most combinations of features possible. Here we will look into how close to a solution sequence we get when we increase the generations the algorithm runs for. Like in previous experiments with other variables we will use a range of sizes for our parameter to run our analysis over[3]; say, generation lengths, $g \in 50, 150, 250, 350, 450, 500]$. The code in Snippet 3.4 shows how we will approach the analysis.

Generation size differs from other parameters in the fact this is purely performance based. A genetic algorithm with 1 generation is just a series of tests; with the results split into 2 sets — winners and losers. As we extend the generations we would be more focused on what happens to certain averages of results across the whole run, rather than absolute improvement. If we look at figure 3.5, mean best score difference against the number of generations, we can observe how, on average, the number of generations has a declining effect the overall change in our best score per generation.

This mean increase of score per generation trend is to be expected; when we are close to a maximum it is more difficulty to randomly select which element

[2]Section 1.1.2 explains what we mean by feature selection

[3]We will be using a population of 150 as this was the best average for score vs computation time for analysis.

```
def generationSizeChecker(opponent):
    file_name = "data/" + str(opponent).replace(" ", "_")
                                       .replace(":","_")
                                       .lower()
                        + "_generation.csv"
    if not os.path.isfile(file_name):
        df_main = pd.DataFrame(data=None, columns=col_names)
        for gens in generation_list:
            start_time = time.clock()
            pop_run = runGeneticAlgo(opponent,
                                     population_size=150,
                                     number_of_game_turns=200,
                                     cycle_length=200,
                                     generations=gens,
                                     mutation_probability=0.1,
                                     reset_file=True)
            end_time = time.clock()
            tmp_df = pd.read_csv(pop_run[0], names=col_names)
            tmp_df["generations"] = gens
            tmp_df["time_taken"] = end_time-start_time
            tmp_df["opponent"] = str(opponent)
            tmp_df["best_score_diff"] = np.append([0],np.diff(tmp_df["best_score"]))
            df_main = df_main.append(tmp_df, ignore_index=True)
        df_main.to_csv(file_name)
        print("List Complete:",file_name)
        return df_main
    else:
        print("file ",file_name," already exists, no calcs to do.")
        file_df = pd.read_csv(file_name)
        \# remove first column
        file_df = file_df[list(file_df)[1:]]
        return file_df \]
```

Figure 3.4: code to check multiple generation lengths.

Figure 3.5: Mean Best Score diff vs Total Generation Lengths

Figure 3.6: Max best score vs total number of generations

in the sequence needs changing to improve our score. On this result we can conclude as we increase generations there is less and less benefit per generation. There is, however, still a benefit to extending the generations but we may have better performance by altering another parameter of the algorithm. There may be a benefit from increasing the mutation rates when we get close to one of these maximums; the more noisy our algorithm is for sequences could improve our chance of finding the correct solution. The probably of finding a solution as we narrow in on a maximum decreases due to the number of elements that, when changed, will provide a better score. Increasing the mutation frequency at this point means that there will be more members of the population that could potentially mutate the elements needed to improve the sequence.

Figure 3.6 shows the proximity the optimal solution sequence once the analysis has concluded. A good score is a score of 3 or more; this can change from player to player, and is never explicitly obvious. We can see that after a number of generations that solutions sometimes get 'stuck' in a local maximum score.
After 250 generations we *seem* to have reached a solution state for our opponents Tit for tat and alternator but not for grudger. Against Grudger we see an example, we have only reached an average score per turn of ___, which is obviously far from its optimal sequence. From the combination of the plots, having more generations means that there is, on average, less of an improvement per generation. It is clear that a higher number of generations is required to find a better solution sequence for an opponent. From now on, 250 is the number of generations we will use to find our solution sequence during the analysis.

For most of the opponents 250 generations seems reasonable to reach a solution sequence as shown in the Alternator and Tit For Tat. However, there are clear signs of local maximums occurring in the Grudger example. Figure 3.6 has reached a better sequence in 450 generations than 500[4]; meaning that increasing the generation length doesn't necessarily mean a local maximum. The complexities with local maximums during the generations lie with mutation rates and crossovers. We will cover this in Section 3.4.4

In this investigation we will want to find the optimal solution and so, from these results, we will want to extend the generation length as far as possible. An infinite number of generations would be preferable, but we don't have an eternity so a selection of a relatively large generation size will be adequate when coming to the final series of tests.

[4]These are indipendent trials and have different sequences.

```
def mutate(self):
"""
Basic mutation which may change any random gene(s) in the sequence.
"""
# if the mutation occurs
if random.rand() <= self.mutation_probability:
    mutated_sequence = self.get_sequence()
    for _ in range(self.mutation_potency):
        index_to_change = random.randint(0, len(mutated_sequence))
        # Mutation - change a single gene
        if mutated_sequence[index_to_change] == C:
            mutated_sequence[index_to_change] = D
        else:
            mutated_sequence[index_to_change] = C
    self.sequence = mutated_sequence
```

Figure 3.7: The mutation code as given in the axelrod-dojo

3.4.3 Changeing Our Mutation Rate

By changing the way in which we mutate our elements within a sequence, we might be able to more effectively narrow in on an optimal solution sequence. The default settings are a frequency of 0.1; meaning for every 10 members of our population that continue into the next generation one of these has some elements in its sequence changed, and a potency of 1; meaning that every sequence that was altered only has 1 element altered. Here we will look into these 2 different concepts and see how they might improve our distance to an optimal sequence and whether we can escape local maximums.

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

These are two separate questions, so first we will look at increasing the potency of our mutation. Once we have found some information on how this effects our solution, we can look into the frequency of our mutations with the new potency as a permanent setting. As shown further on, there is not much of an improvement on our algorithm to changing either of these.The mutation algorithm is shown in Snippet 3.7

EFFICIENCY NOTE: This approach allows for an $O(1)$ factor of scaling. This makes changes in mutation a great candidate for an approach to reduce our solu-

tion sequence distance compared with other approaches, for example increasing the population size.

Changing Mutation Potency

Changing the potency of the algorithm will mainly generate the noise in our sequence generation to generation, increasing the distance[5] between the mutated sequence from the original.

This potentially could create an algorithm that is too 'jumpy' for narrowing in on a solution. We can imagine a sequence as a vector in 200 dimension space then a mutation for element X_i is the same as changing the vector in its i^th dimension. Shortening this example to a vector in 3 dimensions (or a sequence of length 3) then a mutation is much more easily visualised. It is clear that a mutation potency should be kept low as to keep consecutively mutated sequences more similar; we will only be looking at mutating our sequences at up to 10 percent of their elements. We will look into having mutation potencies $m_p \in [1, 2, 3, 5, 10, 15, 20]$

Using the data generated from Snippet 3.8 of code we are able to look at how our best score and our best score diff is affected as we increase the number of positions.

Figure 3.9 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 changing only 1. This may be down to chance (and if the test is rerun this may disappear), however looking at more opponents than just grudger we find there is no clear benefit to increasing the mutation potency with respect to 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 looking at how our average best score difference changes as we increase the mutation potency there is no sign that there is a significant improvement to our sequence. The increase in mean best score difference is not substantial and could be down to chance

Changing Mutation Frequency

In contrast to changing the mutation potency, increasing the frequency should allow us to generate more unique sequences generation to generation. We will look at what happens when we run the genetic algorithm on a set of mutation

[5]Distence concept from coding theory; $d(s_1, s_2) =$ the number of differing positions between 2 sequences s_1 and s_2. $d(111, 110) = d(CCC, CCD) = 1$

```
    def mutationPotencyChecker(opponent):
file_name = "data/" + str(opponent).replace(" ", "_").replace(":","_").lower() + "_mutat
if not os.path.isfile(file_name):
    df_main = pd.DataFrame(data=None, columns=col_names)
    for potency in mutatuon_potency_list:
        start_time = time.clock()
        pot_run = runGeneticAlgo(opponent,
                                 population_size=150,
                                 number_of_game_turns=200,
                                 cycle_length=200,
                                 generations=250,
                                 mutation_probability=0.1,
                                 mutation_potency=potency,
                                 reset_file=True)
        end_time = time.clock()
        tmp_df = pd.read_csv(pot_run[0], names=col_names)
        tmp_df["mutation_potency"] = potency
        tmp_df["time_taken"] = end_time-start_time
        tmp_df["opponent"] = str(opponent)
        tmp_df["best_score_diff"] = np.append([0],np.diff(tmp_df["best_score"]))
        df_main = df_main.append(tmp_df, ignore_index=True)
    df_main.to_csv(file_name)
    print("List Complete:",file_name)
    return df_main
else:
    print("file ",file_name," already exists, no calcs to do.")
    file_df = pd.read_csv(file_name)
    \# remove first column
    file_df = file_df[list(file_df)[1:]]
        return file_df
```

Figure 3.8: Mutation potency code

Figure 3.9: Best score vs generation for different mutation potencies

Figure 3.10: Average best score diff vs mutation potencies

frequencies; $m_f = [0.1, 0.2, 0.3, 0.4, 0.5]$ The code in Snippet 3.11 shows the algorithm that will complete this analysis.
————————————-EDIT LINE————————————

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 Figure **??** 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?)

3.4.4 Mitigating 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 is an opponent which it is possible to attain a local maximum score and be 'trapped' in this solution sequence, as shown in the multiple plateaus in the . From the output sequences 3.4.4 and 3.4.4 it appears that the sequence of moves 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, we will look into ways to mitigate these effect in Section 3.4.4.

Grudger best start: $C : [6, 8, 2, 1, 2, 1, 6, 1, 2, 2, 4, 2, 1, 1, 2, 10, 1, 1, 1, 3, 1, 3, 1, 1, 1, 1, 2, 2, 1, 1, 1, 2, 3, 5, 2, 2, 2, 1, 1,$

Grudger best end: $C : [22, 178]$

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.

```
def mutationFrequencyChecker(opponent):
file_name = "data/" + str(opponent).replace(" ", "_")
                                    .replace(":","_")
                                    .lower()
                    + "_mutation_frequency.csv"
if not os.path.isfile(file_name):
    df_main = pd.DataFrame(data=None, columns=col_names)
    for freq in mutation_frequency_list:
        start_time = time.clock()
        pot_run = runGeneticAlgo(opponent,
                            population_size=150,
                            number_of_game_turns=200,
                            cycle_length=200,
                            generations=250,
                            mutation_probability=freq,
                            mutation_potency=1,
                            reset_file=True)
        end_time = time.clock()
        tmp_df = pd.read_csv(pot_run[0], names=col_names)
        tmp_df["mutation_frequency"] = freq
        tmp_df["time_taken"] = end_time-start_time
        tmp_df["opponent"] = str(opponent)
        tmp_df["best_score_diff"] = np.append([0],np.diff(tmp_df["best_score"]))
        df_main = df_main.append(tmp_df, ignore_index=True)
    df_main.to_csv(file_name)
    print("List Complete:",file_name)
    return df_main
else:
    print("file ",file_name," already exists, no calcs to do.")
    file_df = pd.read_csv(file_name)
    \# remove first column
    file_df = file_df[list(file_df)[1:]]
    return file_df
```

Figure 3.11: Mutation potency code

Figure 3.12: Best score vs generation for different mutation frequencies

Figure 3.13: Average best score diff vs mutation frequencies

- 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.

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

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

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:

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

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:

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

Below is an exapmple of the new crossover sequence:

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

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:

3.4.5 Altering Initial Population

We want to select starting sequences that fit patterns we know will have good results. For example totalities, with heads/tails, are usually very effective against simple opponents; for example Tit For Tat, 'dumb' opponents or Grudger. Alternating and certain known solution optimal sequences are good starting sequences for an initial population, allowing a more intelligently distributed set of starting points for the random mutation process.[6]
This section discusses the results of working on an initial population that contain common solution sequences. We will start by creating a population of 'neat' starting members then allow the entropy of the genetic algorithm to alter these sequences. Deciding where to start our algorithm may mitigate potential sub-optimal solutions by reducing the distance between the starting sequences and optimal solutions. the list of 'neat' starting points are below:

Totalities

[6] This can be visualised as placing balls on a 'lumpy' 2d plane to try and find the minimum, starting with an educated guess means we wont get all the balls stuck in one vally which isnt the deepest.

- $C : [200]$

Trailing Defectors

- $\{C : [i, 200 - i]\} \quad i \in [1, 5]$
- $\{C : [200 - i, i]\} \quad i \in [2, 5]$

Odd Repetitions

- $C : [1, 198, 1] = CD...DC$
- $C : [2, 197, 1] = CCD...DC$
- $C : [1, 197, 2] = CD...DCC$

Even Alternating & extensions

- $C : [\overbrace{(1, 1)}^{100}] = CDCD....CD$
- $C : [\overbrace{1, (2, 2)}^{50}] = CCDD...CCDD$
- $C : [\overbrace{(2, 2)}^{50}] = CCDD...CCDD$

For each of these sequence sets the inverse?? will also be applied

3.5 Conclusion of approach

Chapter 4

Implementation Of Sequence Discovery

Chapter 5

Results and Discussion

Chapter 6

Practical Applications for Solution Sequences

maybe how this works with respect to pathfinding, who we should select in a set of opponents if we only have to play a single opponent in the set.

Chapter 7

Summary and Future Research

Bibliography

[A+87] Robert Axelrod et al. The evolution of strategies in the iterated prisoners dilemma. *The dynamics of norms*, pages 1–16, 1987.