Final Year Project

Toby Devlin

January 31, 2018

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.

1.1 Iterated Prisoners Dilemma

To be written.

1.2 Machine Learning & Computer Intelligence

This section will briefly provide a background to machine learning algorithms. This is by no means a comprehensive look into these subjects but will provide sufficient background to enable a technical discussion in further sections.

Machine Learning is a field of computer science that has existed since the first computers have been around. Most famously the questions posed by Alan Turing in 1950 [Tur50] asks 'can machines think', a question that has been refined and analysed to this day. The field of computer intelligence is rich in its complexities and has recently been making breakthroughs [Kni17] on topics which would traditionally be considered 'thinking'. Along with this, recently there has also been record levels of funding [Chu17] put in to companies which operate in this field, producing results in areas that would usually seem 'solved'.

1.2.1 Genetic Algorithms

This report will cover one of the forms of machine learning called genetic algorithms. Techniques of machine learning can be combined and used together in many situations, the field of mathematics research is one; for example machine learning is used in [CB97].

Genetic Algorithms fall under a branch of machine learning called feature selection. More generically, genetic algorithms are put into a class of unsupervised reinforcement learning algorthms Techniques of using genetic algorithms for generating solutions to problems typically revolve around heuristically improving members of a population who represent these solutions. The concepts of a genetic algorithm come from nature; like nature we create a survival of the fittest selection [DB09] competition to evaluate a population then kill off the weakest members. After this cull we create offspring from the most successful population or introduce new members from a predefined source. This process is then repeated until we stop it, or forever in the case of nature.

We say a genetic algorithm is structured in the following way. Given a population, P, each with unique genes (genes and member properties are interchangeable), and a number of generations, $G \in \mathbb{N}$, the algorithm will create G loops of scoring and potentially removing each of the members of the population. It does this by using a mapping from a member of the population to an ordered set, for example $f(p_i) \mapsto \mathbb{R},\ p_i \in P$. This function, f, is defined beforehand in a way which describes the goal of our investigation. Defining a cutoff or bottleneck $b < |P|$, such that on conclusion of scoring the population, the top b ranking members by score can be kept and the rest discarded. Once we have removed a certain

percentage of our population we can rebuild it using a series of crossovers and mutations (and possibly introducing new members into the population).

- Crossovers take in 2 members of the population and return a new member based on some parameters of the 2 'parents'. For example, our crossover takes the first half of a sequence from one and the second half from the other, merging them to form the third.
- Mutations allow a (possibly targeted[1]) change in a single member of the population. A mutation has 2 parameters, a potency $M_p \in \mathbb{R} > 0$ and a frequency $M_f \in [0, 1]$. M_p describes how strong the mutation is, the higher it is the larger change to the member occurs. M_f explains the percentage of how many members of the population are mutated.

Figure 1.2.1 shows a flow diagram of this general cycle.

Put a figure of generic cycle here.

Our implementation of a genetic algorithm is more custom and has the following steps:

1. Start with a predefined population, supplemented with randomly generated member until to size.
2. Each member plays the given opponent with their sequence and returns with the average score per turn.
3. Members of the population are ranked by this average score per turn and the highest scoring 25% will be kept for the next round. The remaining 75% are killed off.
4. The remaining population will then be copied and these copies mutated to create unique sequences before being merged back in to the main population.
5. The remaining 50% difference is then made up of mutated results of crossovers from members of the current population or random new members, depending on a random selection algorithm.[2]
6. A generation has now concluded. Repeat from step 2 until the desired number of generations are finished and a final best sequence is returned.

Figure 1.2.1 shows a flow diagram of our cycle. This is the algorithm we will use in Chapter 4 for analysing parameters

Put a figure of our cycle here.

1.2.2 Bayesian Optimization

1.3 Brief Overview

This document will be be looking at the creation of sequences to beat given players in The Iterated Prisoners Dilemma. Analysis will be focused on looking into just the single opponent use case, but the idea of designing a sequence for a tournament for a given number of opponents is a potential follow on to this work. This task is as follows:

Problem:
When playing a given Iterated Prisoners Dilemma strategy, O, as an opponent, what is the best ordered solution sequence of moves, S, to play in order for us to obtain the highest possible average score per move across the game.

For example an opponent known as Tit For Tat, which cooperates on its first move and the copies your last move on subsequent moves, will have a solution sequence of moves that are all cooperation apart from the last. This is an obvious example and a simple strategy to calculate for, the ultimate goal of this investigation is to look into all the strategy as defined in the Axelrod Library. These are listed in appendix Section **??**

[1] For example using intuition and targeting specific genes, or allowing another algorithm to improves the targeting of this meta function.
[2] This algorithm had a bug which would change the size of the population in the first generation. This was fixed after Section 4.7 was written.

Chapter 2

Literature Review

2.1 Background

see trello board for writings. Question to be asked: "what pompous should this section serve?"

- book1
- book2

2.2 Strategies Of Interest

2.2.1 TitForTat

2.2.2 Alternator

2.2.3 Grudger

2.2.4 Random

2.2.5 EvolvedFSM16

2.2.6 CollectiveStrategy

2.2.7 ZDExtortion

2.2.8 Cycler

Chapter 3

Task Background

This chapter will cover sections on how each part of the project will been carried out. Useful notation is given in Section 3.1 however in some areas of discussion sequences are more easily denoted by explicitly describing this sequence. Section 3.3 on code techniques will not cover setting up an environment to conduct similar analysis; this walk through will eventually be given on my github page, along with example code.

3.1 Notation

Let $S \in \{C, D\}^L$ where C, D represents a cooperation, defection respectively. $L = 200$ 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 it is typically easier to describe the solution. Otherwise we use the notation to describe a sequence.

Examples:

$$C : [1, 4, 3, 2] = CDDDDCCCDD \tag{3.1}$$
$$D : [\ (1, 1)^5] = DCDCDCDCDC \tag{3.2}$$
$$C : [1, (2, 1)^2, 2, 1] = CDDCDDCDDC \tag{3.3}$$
$$\{D : [i, 5 - i]\} \quad i \in [2, 4] = \{DDCCC,\ DDDCC,\ DDDCC\} \tag{3.4}$$
$$\tag{3.5}$$

3.2 Solution Form

In this research our goal will be to use the algorithm described in Section 1.2.1 produce an arbitrary sequence for an opponent, S_o. This sequence will represent what moves we should play against the opponent to get our largest potential score per turn.

This investigation focuses on sequences that will allow us to maximise our score overall, rather than just beating any given opponent. An 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 real world applications of these results are discussed in Chapter 6.

Each solution sequence is uniquely generated for each opponent. If there are two similar opponents, say Grudger and Collective Strategy, these will be independently analysed and solutions generated. We will discuss how similar solution sequences relate to how similar strategies are in Chapter 5. When working with stochastic players, we will be seeding them in order to determine the best sequence. Each stochastic player will be considered under a variety of seeds. The motive to allow this to happen is explained described in Section 4.6.
Sequences returned by the algorithm will be classified in the following way:

-

3.3 Code Techniques

This section will discuss how we will leverage code to complete aspects of the investigation. We will be using Python because of its flexibility and it is the language the required libraries are written in.

3.3.1 Building The Algorithm

In order to generate a solution sequence we have to train against each Axlrod opponent against another Axelrod opponent. We will use the Cycler strategy[1] in the algorithm by editing the input parameter to improve our score against an opponent.

This model means we can create a population of Cycler players and input a sequence of length 200 as a parameter to set off our genetic algorithm. The subsequent inputs for the populations Cycler players will be created using the genetic mutation and crossover techniques, see Section 1.2.1 for details.

The flow of Play-Rank-Create looping (see Figure 1.2.1) will be the basis of creating the optimal strategy for each other opponent. Each step is defined in the Axelrod-Dojo librarys' Population and CyclerParams classes. Rather than store all the functionality in one place we are able split up aspects of the flow to allow for flexibility in what type of population can be used.

3.3.2 Conducting analysis

For the analysis a mix of Jupyter Notebooks and IDE[2] development were used.

[1] See Section 2.2.8 for description
[2] Pycharm

Libraries And Codebases

Main Research Libraries

- **Axelrod** — Used for the core of the prisoners dilemma and iterated prisoners dilemma functionality code.
- **Axelrod-Dojo** — Applied machine learning techniques that revolve around generating solutions to questions relating to the Axelrod library.

Functional libraries Table 3.1 shows the functional libraries used. These are codebases which are not involved in the core functionality of the IPD.

Library	Reason
os	For operating system functionality.
time	For time calculations.
matplotlib pyplot	For plotting graphs and images with data
pandas	For data manipulation.
numpy	For reducing complexity of numerical calculations.
itertools	For easier iterations over data structures.

Table 3.1: Functional Python libraries for analysis

3.3.3 Setting Up A Research Environment

To be written.

Chapter 4

Implementation Of Sequence Discovery

In this chapter, consideration will be given to the process of 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.

Before conducting the bulk calculations for the set of opponents listed in the appendix we will test values for 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 mostly 'simple' and can be explained in a 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 type of opponent.

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 3.2 this the optimal solution sequence and it is unique. NOTE: unique solution sequences only exist for non stochastic opponents.

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 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.

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

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
...	...	...	...	...	...	...

Table 4.1: Output data table

- If we make each sequence more likely to mutate generation to generation what will happen? What about increasing how potent our mutations are?

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 1.2.1; 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 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 new sequences after mutation. 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 4.1 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 code will output data in the form of Table 4.1.

By grouping this data by the population we observe how initial populations affect different opponents. Its clear that from figure 4.2 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 4.3 that... It doesn't, however, ensure that we find the solution sequence; as is shown in the lack of long plateaus in the lines.

The improvement's from this effect are non-linear 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.

```
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_one, 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 4.1: code to check multiple populations

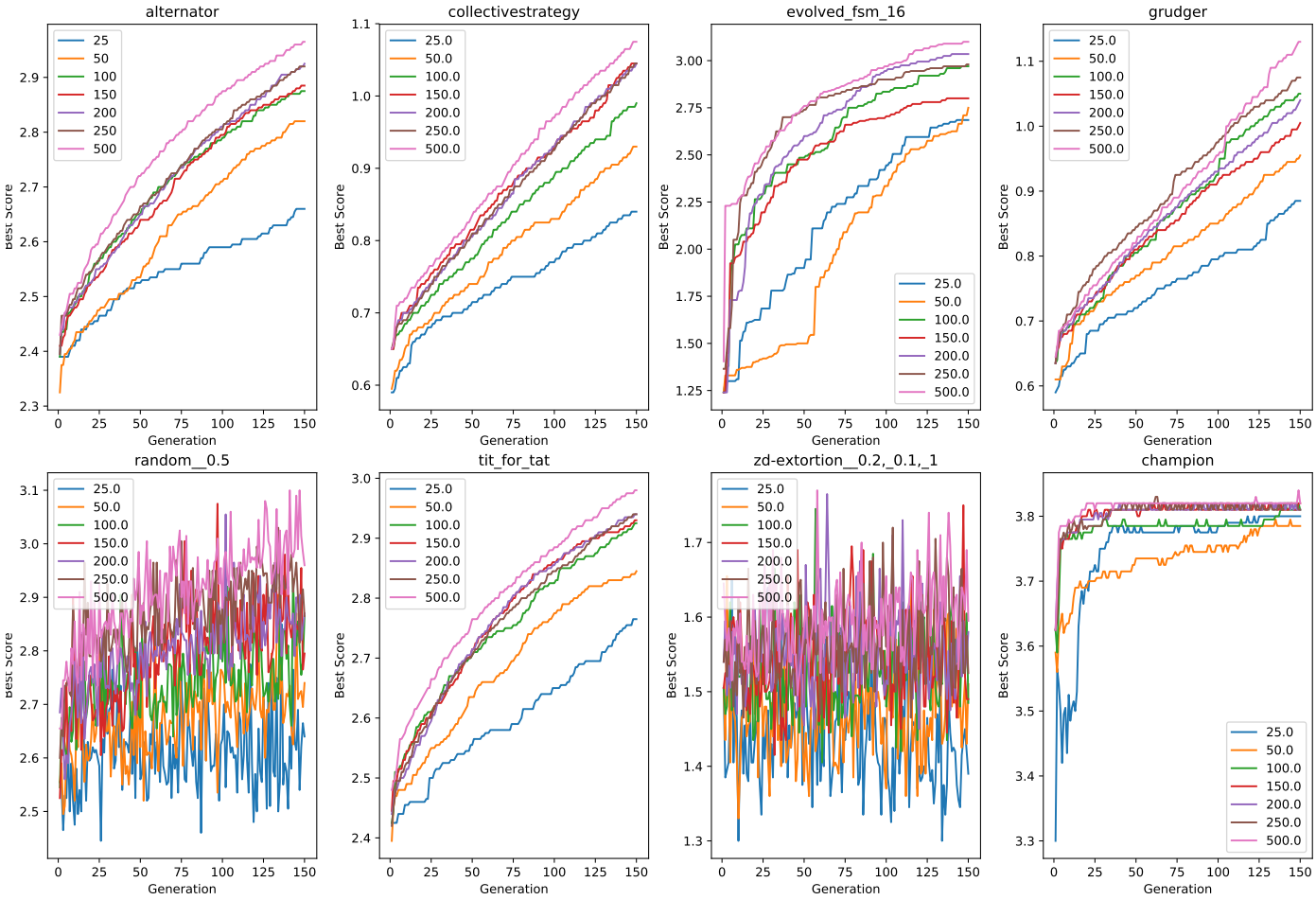

Figure 4.2: Best score per turn vs generation for different initial population sizes

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, (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 set of sizes for our parameter to run our analysis for each size, here we will take generation lengths $G \in [50, 150, 250, 350, 450, 500]$.[1] The code in Snippet 4.4 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 4.5, 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 in the sequence needs changing to improve our score. On this result we can conclude

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

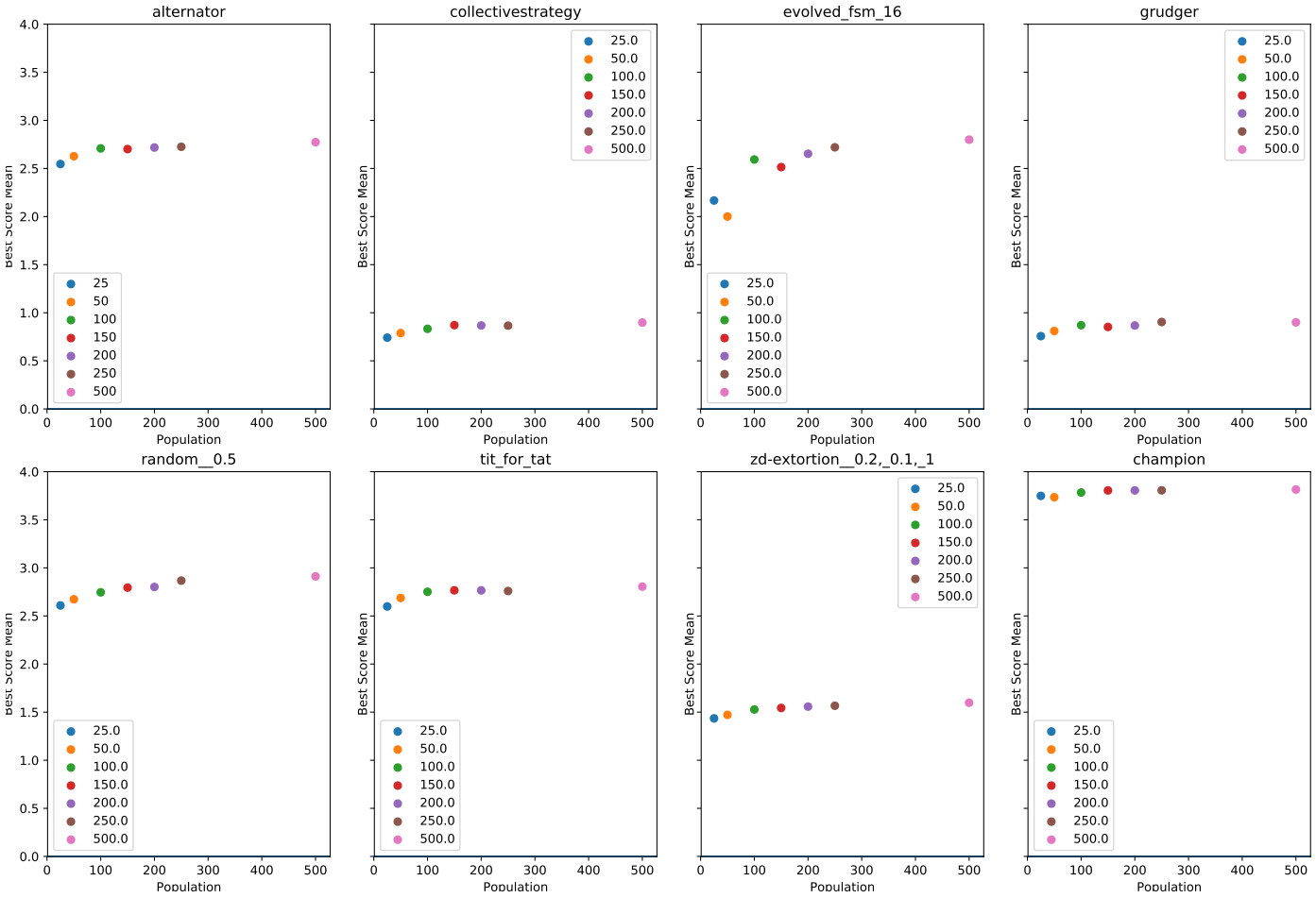

Figure 4.3: Scatter of max best score vs different initial populations

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 **??** 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 **??** has reached a better sequence in 450 generations than 500[2]; 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.

[2]These are independent trials and have different sequences.

```
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 4.4: code to check multiple generation lengths.

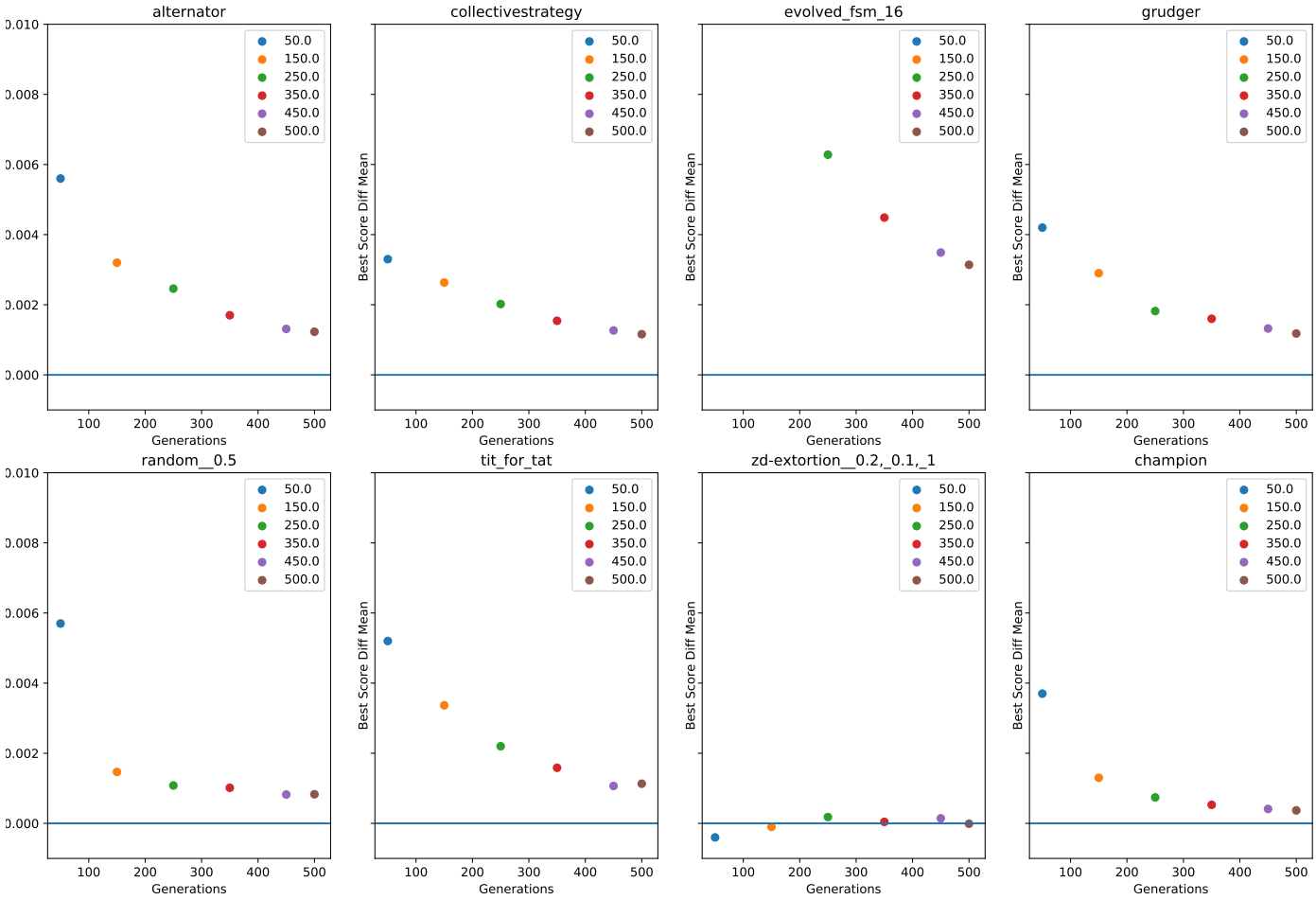

Figure 4.5: Mean Best Score diff vs total generation lengths

We will cover this in Section 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.

4.3 Changing 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 4.6

```python
def mutate(self):
    # 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 4.6: The mutation code as given in the axelrod-dojo

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

4.3.1 Changing Mutation Potency

Changing the potency of the algorithm will mainly generate the noise in our sequence generation to generation, increasing the distance[3] 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 S_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 4.7 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 4.8 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.

4.3.2 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 frequencies $M_f \in [0.1, 0.2, 0.3, 0.4, 0.5]$. The code in Snippet 4.10 shows the algorithm that will complete this analysis.

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.

[3]Distance concept has been taken from coding theory; $d(s_1, s_2)$ = the number of differing positions between 2 sequences s_1 and s_2. For example: $d(111, 110) = d(CCC, CCD) = 1$

```
def mutationPotencyChecker(opponent):
    file_name = "data/" + str(opponent) \
        .replace(" ", "_") \
        .replace(":", "_") \
        .lower() + "_mutation_potency.csv"
    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 4.7: Mutation potency code

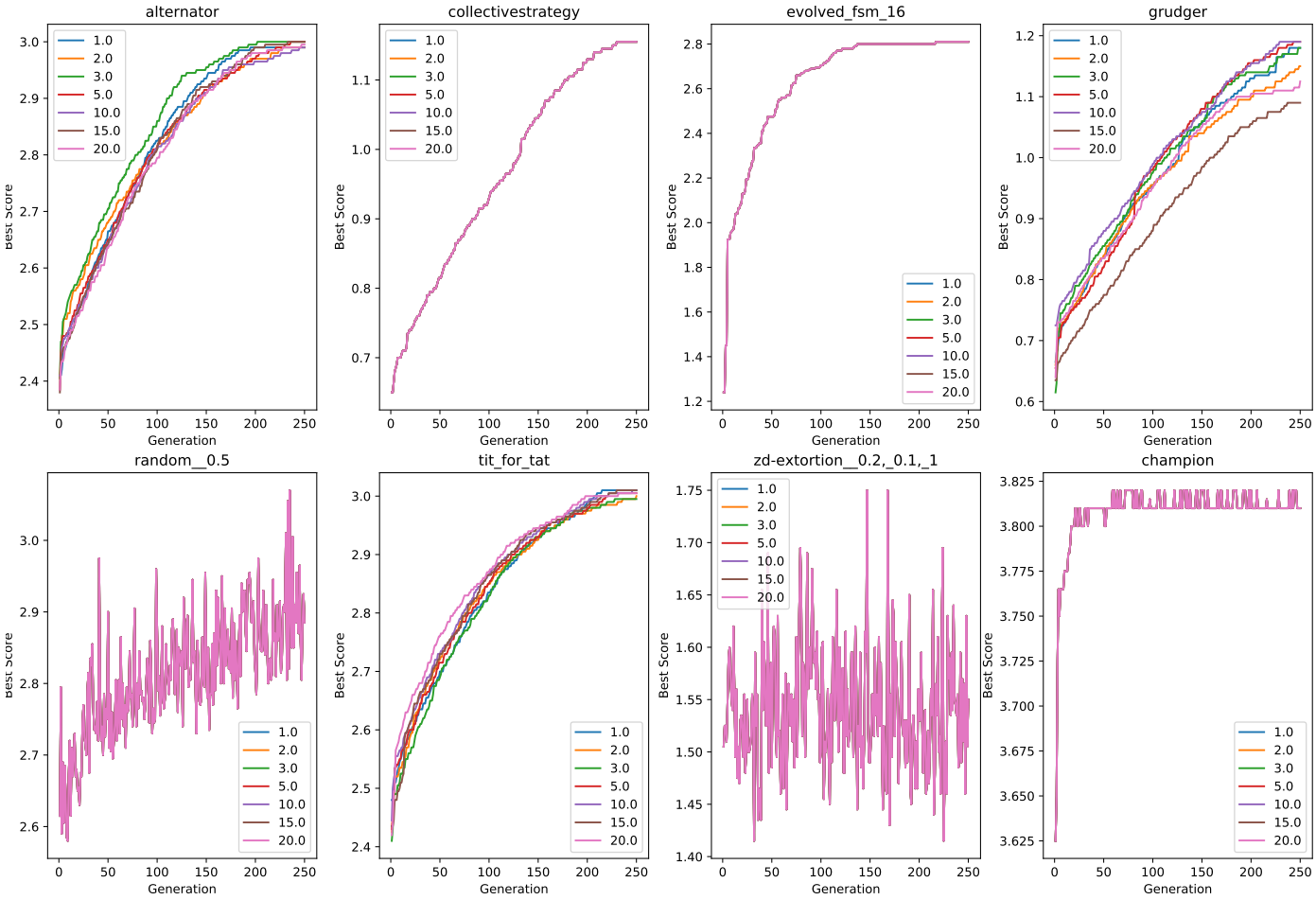

Figure 4.8: Best score vs generation for different mutation potencies

From Figure 4.11 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.

4.4 Mitigating local maximum solutions

The occurrence of local maximums is something that has only occurred in sequences against for the Grudger opponent so far. Figure 4.11 shows that there are clearly 2 distinct plateaus that are reached in terms of score, and the overall scores are much lower than the optimal, around 1.4.

The difference between the Grudger and the other opponents have been looking at is that the Grudger has a singularity point where its behaviour changes. The change in behaviour is not uncommon, Tit For Tat works in a similar way and in both cases our algorithm has managed to identify this 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. From the output sequences below it appears that the sequence is being changed into becoming 2 sections of opposing moves. If we look at a random start sequence and then the end sequence after 250 generations we can see that the genetic algorithm is learning defect after some point and cooperate before. This is due to the fact a good solution will end in lots off defections after it has already defected, but cooperate beforehand to avoid this harsh behaviour.

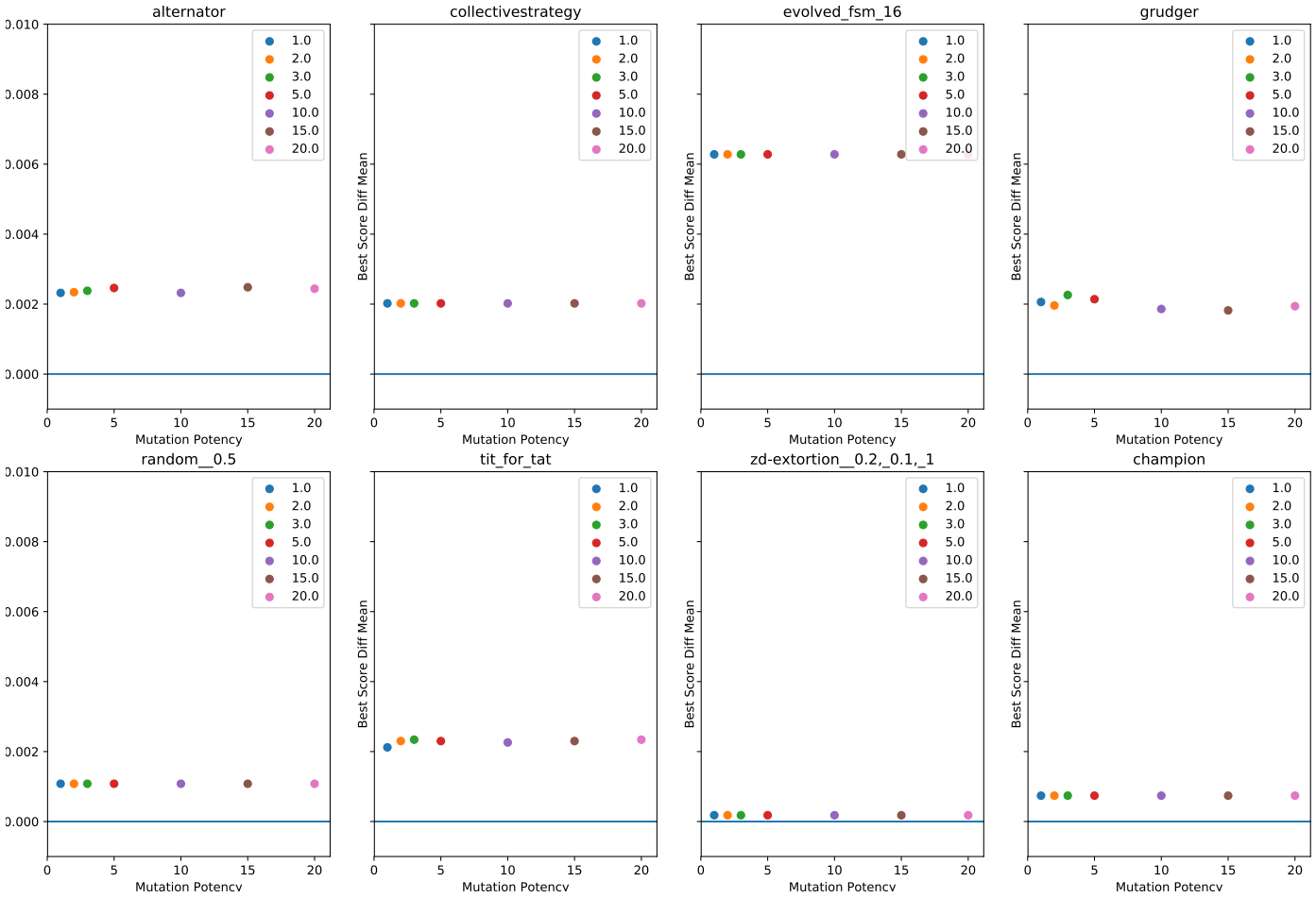

Figure 4.9: Average best score diff vs mutation potencies

Grudger best start: CCCCCCDDDD... (No pattern is obvious.)
Grudger best end: $C : [22, 178]$

We may be able to identify a way to mitigate this singularity effect by looking at the difference between 2 similar solutions. Grudger and Tit For Tat differ in their strategies in two ways:

- Grudger never changes its mind. There is one change in behaviour for the entire game, unlike Tit For Tat. The algorithm then only has a single opportunity to observe this per population per generation meaning the behaviour is much less frequently encountered.
- Grudger will not become 'smart' again. This means that the move genetic algorithm picks up the effect of a single defection in its sequence it starts playing a 'dumb' opponent. A random start of Cs and Ds puts the likeyhood of at least 1 defection occurring in the first 10 moves at above 99.99%. This swap from 'smart' to 'dumb' will, most likely, always occur in the first 10 moves; the only way of extending the 'smart' player is to add a cooperation to the end of the starting Cs.

Snippet 4.13 shows two totality games against grudger, 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 random initial population. Because of this, the algorithm has to shuffle towards the potential benefit of using these totalities rather than start with analysing them. Our case against grudger requires the algorithm to attempt this shuffle towards a totality after encountering the grudging effect. This would then require the algorithm to select (using random selection out of 200) the first defection move and change it (only 1 in 10 members are mutated) to a cooperation move. This likelihood is incredibly small, and starting at common or uniform sequences would be more beneficial, as posed in Section 4.5.

```
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 4.10: Mutation potency code

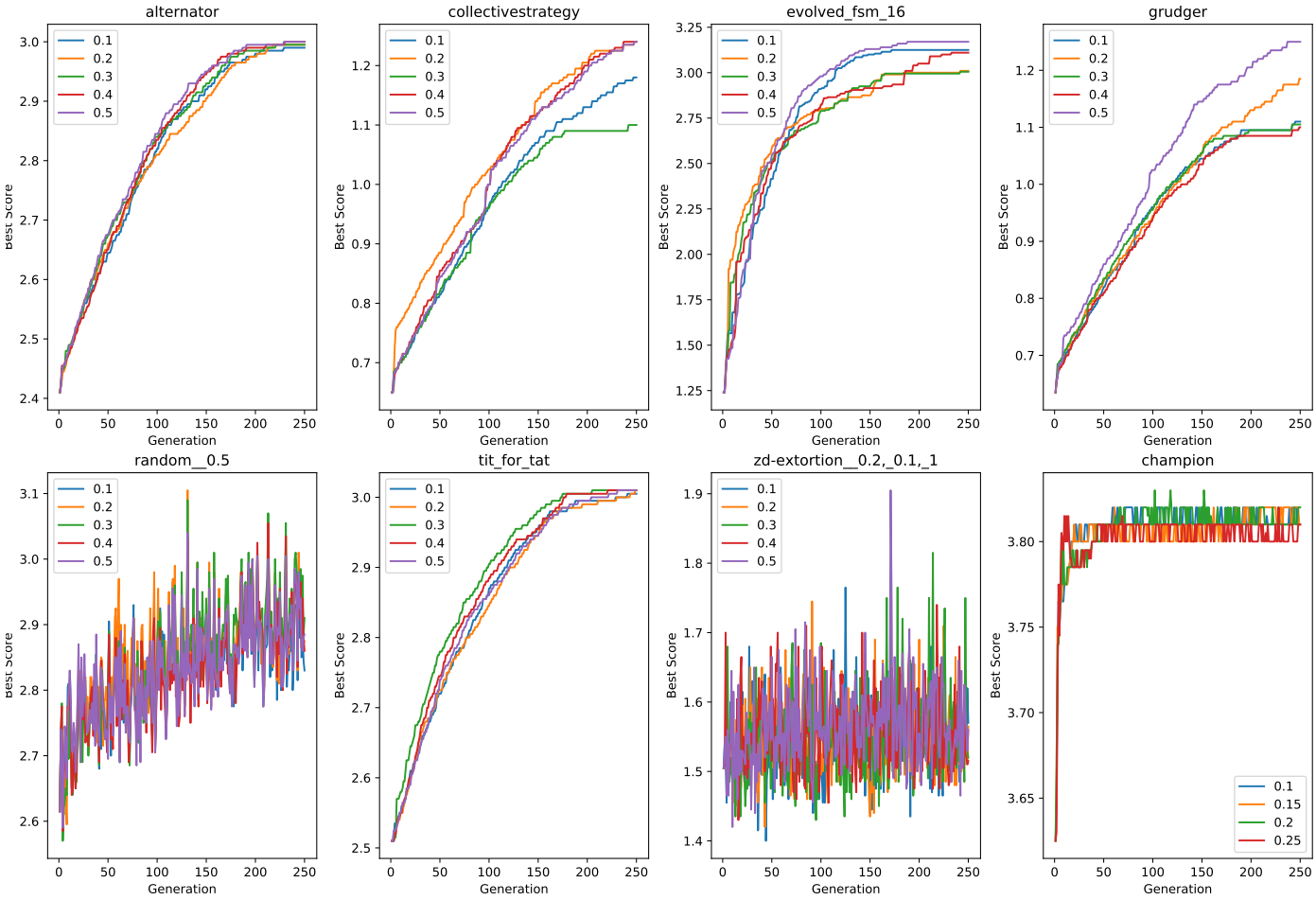

Figure 4.11: Best score vs generation for different mutation frequencies

As Snippet 4.13 shows, we should be converging on a totality of Cs rather than what its doing; 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.

4.4.1 Ineffective Approach of Altering Crossover And Mutation

The process of converging to Ds when building a solution against grudger then sheds light on the process the algorithm takes to find a solution. If we are to find the optimal solution sequence, we must take a crossover and mutation path which doesn't cut off better paths as we work our way towards good solutions; 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:
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:

$$|----------|and|++++++++++|$$
$$=|----------++++++++++|$$

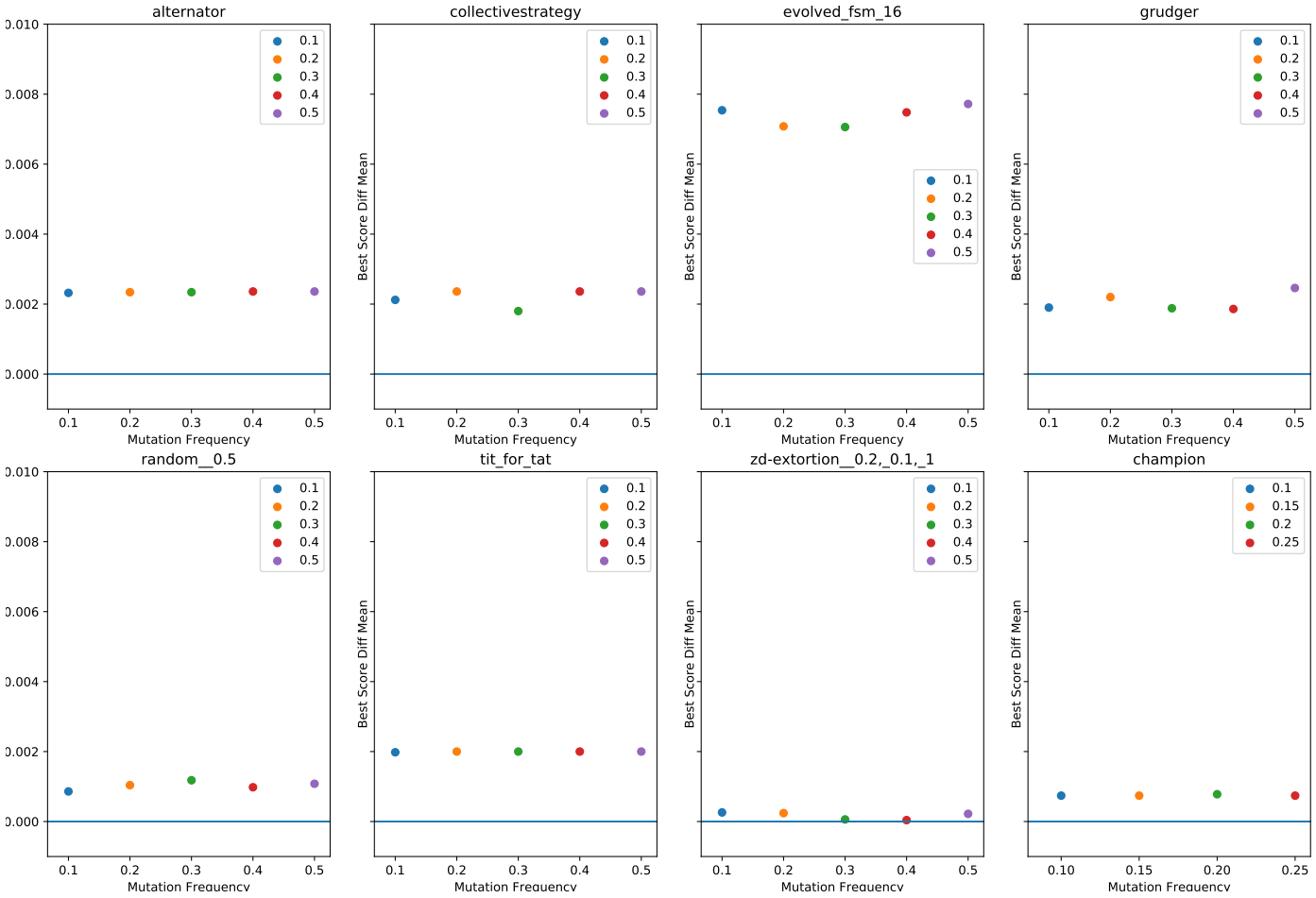

Figure 4.12: Average best score diff vs mutation frequencies

to the mixture:

$$|----------|and|++++++++++|$$
$$=|--++--++--++--++|$$

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 shown in Figure 4.15. As shown, the algorithm splits the two sequences into 10 section and the new sequence is formed from Alternating sections. Figure 4.16 shows an example of the new crossover method.

We will look at how this new crossover algorithm works with the default mutation (freq=.1 and pot=1) to improve our local maximums with the selection off opponents.

There was no noticeable improvement from introducing this change of function. Each players average score per turn was in the expected score if the algorithms crossover method was unchanged. The problem of mitigating sub optimal solutions may be more efficiently solved using a predefined population. Section 4.5 discusses this in more detail.

4.5 Altering Initial Population

Altering the initial population allows us 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,

```
players = (axl.Grudger(), axl.Cycler("C"))
match = axl.Match(players, 200)
match.play()
print("final scores:", match.final_score())
print("per turn:", match.final_score_per_turn())

# >final scores: (600, 600)
# >per turn: (3.0, 3.0)

players = (axl.Grudger(), axl.Cycler("D"))
match = axl.Match(players, 200)
match.play()
print("final scores:", match.final_score())
print("per turn:", match.final_score_per_turn())

# >final scores: (199, 204)
# >per turn: (0.995, 1.02)
```

Figure 4.13: Grudger matches against totalities

```
def crossover_old(self, other_cycler):
    # single point crossover:
    crossover_point = int(self.get_sequence_length() // 2)
    # get half 1 from self
    seq_p1 = self.get_sequence()[0: crossover_point]
    # get half 2 from the other_cycler
    seq_p2 = other_cycler.get_sequence()[crossover_point: other_cycler.get_sequence_length()]
    crossed_sequence = seq_p1 + seq_p2
    return CyclerParams(sequence=crossed_sequence)
```

Figure 4.14: Old Crossover algorithm

```
def crossover(self, other_cycler):
    # 10 crossover points:
    step_size = int(len(self.get_sequence()) / 10)
    # empty starting seq
    new_seq = []
    seq1 = self.get_sequence()
    seq2 = other_cycler.get_sequence()
    i = 0
    j = i + step_size
    while j <= len(seq1) - step_size:
        new_seq = new_seq + seq1[i:j]
        new_seq = new_seq + seq2[i + step_size:j + step_size]
        i += 2 * +step_size
        j += 2 * +step_size
    return CyclerParams(sequence=new_seq)
```

Figure 4.15: New Crossover algorithm

```
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 = int(len(seq1) / 10)
i = 0
j = i + step_size
new_seq = []
while j <= len(seq1) - step_size:
    new_seq = new_seq + seq1[i:j]
    new_seq = new_seq + seq2[i + step_size:j + step_size]
    i += 2 * +step_size
    j += 2 * +step_size
print(seq1)
print(seq2)
print(new_seq)

# >[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
# >[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
# >[1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0]
```

Figure 4.16: Example of new crossover algorithm

'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.[4] 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 stated below:

Totalities

- $C : [200]$ — 1 sequence

Single Change Sequences

- $\{C : [i, 200 - i]\} \quad i \in [1, 10],\ i \in \mathbb{Z}$ — 10 sequences
- $\{C : [200 - i, i]\} \quad i \in [1, 10],\ i \in \mathbb{Z}$ — 10 sequences

Matching Tail Sequences

- $\{C : [i, 200 - (i + j), j]\} \quad i, j \in [1, 5],\ i, j \in \mathbb{Z}$ — 25 sequences

Alternating

- $C : [\ (i, i)^{100/i}] \quad i \in \{1, 2, 4, 5\}, i \in \mathbb{Z}$ — 4 sequences

Handshakes

- $C : [\ i, j, k, 200 - (i + j + k)] \quad i, j, k \in \{0, 1, 2, 3\}, i \in \mathbb{Z}$ — 4 sequences

For each of these sequence sets the inverse, with a defection move start, will also be added to the set of starting sequences. This in total gives 164 (after code 4.18) of sequences, we will then make up the difference to the population limit using a set of random sequences.

We can now look into how this has effected the previous experiments.

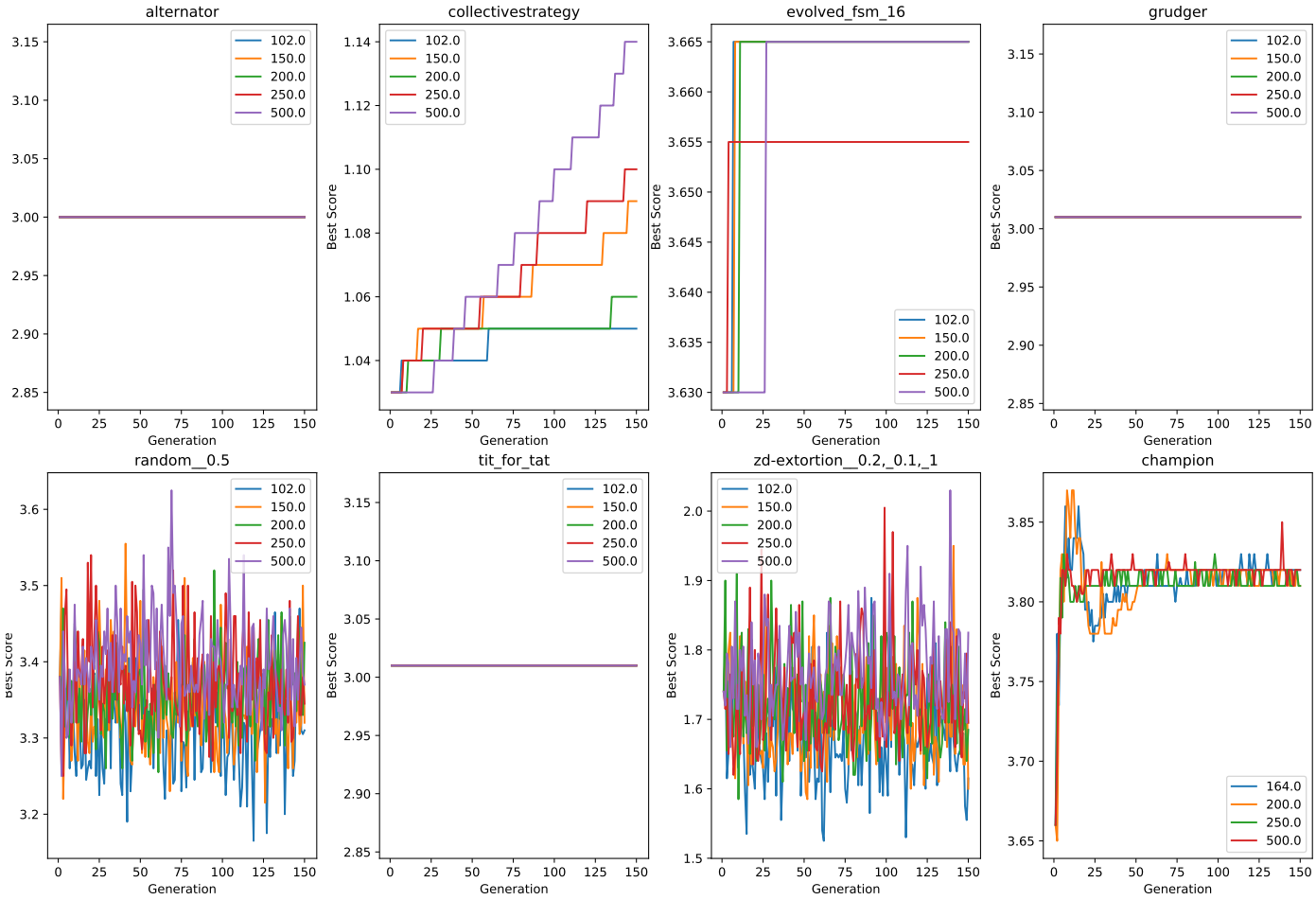

Figure 4.17: Best Score vs generations for preset initial populations on top of random sequences

4.5.1 Population Size

Figure 4.17 shows the results of using a given initial population of sequences of size 102 along with supplementing this with random members. We will have $|P| \in [102, 200, 250, 500]$. The Collective Strategy run in figure looks as if there are much better results with an initial set population for the algorithm. Of the two that are not showing straight lines we can observe that the algorithm has found the optimal sequence for all but Random & Collective Strategy. This is probably due to 2 different reasons:

Random: The 'dumb' strategy should being beaten with a totality of D, the algorithm has in fact converged to almost this totality, but still has intermittent Cs. The reason for this is the scoring grade 'score per turn' will reflect on the number of intermittent Cs in the Random players sequence. More Cs in the Random sequence will allow the algorithm to score more. This leads to a solution sequence containing some random Cs not because they score better in some turns, but because the totality of Ds played against a Random sequence would 'loose' to a sequence containing some Cs in because its not a fair trial; the two sequences play against different Random opponent sequences. **Collective Strategy**: as described in chapter 2.2 this strategy is, basically, a handshake + Grudger. If we look into the sequence it selected we can see its found the handshake but then arrives on Grudger. After this encounter the same problem as we had before population selection occurs and the algorithms limits the damage by splitting into Cs then Ds. Solving the collective strategy (and handshakes in general) may be simple; we just put in all the possible n move handshakes followed by totalities and then set to work on the 2nd part of the sequence.

It is clear that having a larger population is good from the old analysis, but the initial population still has to be tweaked to improve its scores against certain handshake opponents. The Random opponent is a special case that requires more analysis to find the absolute optimal. The new initial population, now of size 164, will include all combinations of C & D of length 5, followed by finishing on all Cs or Ds as shown in the Totalities & handshakes section of Snippet 4.18. This

[4]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.

run will also supplement with random members for running test for $|P| \in [164, 200, 250, 500]$
After adding the extra members of the initial population we can see, from Figure 4.19, that now the handshake strategy (such as Collective Strategy) are solved much sooner. Also shown in Figure 4.19 is the Random and ZD extort players, these are examples of Stochastic players which are not finding optimal strategy's. These will be discussed further in Sections 4.5.4& **??**.

4.5.2 Generation Length

Figure 4.20 shows the results from changing the number of generations the algorithm will run to $G \in [50, 150, 250, 350, 450, 500]$ whilst also running with a population size of $|P| = 200$.
The Figure 4.20 shows no real improvement from increasing the generations, we will use the results from analysis above sections for constructing the full analysis.

4.5.3 Mutation Potency

Figure 4.21 shows the results from changing mutation potency the algorithm will use to be $M_p \in [1, 2, 3, 5, 10, 15, 20]$ whilst also running with a population size of $|P| = 200$.
The Figure 4.21 shows no real improvement from increasing this variable of the algorithm, we will use the results from analysis in previous sections for constructing the full analysis.

4.5.4 Mutation Frequency

Figure 4.22 shows the results from changing mutation potency the algorithm will use to be $M_f \in [0.1, 0.15, 0.2, 0.25]$ whilst also running with a population size of $|P| = 200$.
The Figure 4.22 shows no real improvement from increasing this variable of the algorithm, we will use the results from analysis in previous sections for constructing the full analysis.

Discussion

The approach of adding a predefined set of members to the population before running the algorithm was incredibly successful. We have seen that for most opponents the solution sequence were looking for is given almost immediately. This result will mean that when calculating the solution sequence for the main population the predefined set, generated by Snippet 4.18, will be used to shorten analysis times.
Opponents such as Random, ZDExtort and Champion are showing that they're not finding the optimal solution. This is due to the fact these player belong to the class of stochastic opponents. Section 4.6 will look into these in more detail, putting forward possible approaches to simplify 'solving' these opponents.

4.6 Stochastic Opponents

Stochastic opponents form a problem with respect to continuity. The algorithm we're using continues to generate a new opponent sequence for every generation we run. This, essentially, leads to the algorithm playing a new version of the opponent every time, leaving very little opportunity to identify features which can be exploited.
The concept of a stochastic opponent can be somewhat 'overridden' by seeding the psudo random number generator that creates the parameters that define what moves our opponent will take. Currently the only way of doing this is by seeing the whole Axelrod library upon initialising the opponent instance, Snippet 4.23 shows the code for wrapping any given opponent with the global seed command.
When calculating the sequence solution for all of the opponents we will do so with seeded versions of the stochastic opponents. The parameters for stochastic opponents will be the default set by the creators and new instances will be made for any variations with real world applications. For example, default Random players will have a 0.5 chance of playing a C move; this parameter can optionally be changed, and will for certain opponents.

4.7 Conclusion of approach

```python
def getCyclerParamsPrePop2(pop_size=200, mutation_prop=0.1, muation_pot=1):
    pop = []
    if pop_size < 164:
        print("population must be 164+,")
        return

    # Totalities & Handshakes
    handshake_leng = 5
    for start in itertools.product("CD", repeat=handshake_leng):
        pop.append(axl_dojo.CyclerParams(list(start) + [C] * (200 - handshake_leng)))
        pop.append(axl_dojo.CyclerParams(list(start) + [D] * (200 - handshake_leng)))

    # 50-50
    pop.append(axl_dojo.CyclerParams([C] * 100 + [D] * 100))
    pop.append(axl_dojo.CyclerParams([D] * 100 + [C] * 100))

    # Single Change
    for i in range(1, 11):
        pop.append(axl_dojo.CyclerParams([C] * i + [D] * (200 - i)))
        pop.append(axl_dojo.CyclerParams([D] * i + [C] * (200 - i)))

    for i in range(1, 11):
        pop.append(axl_dojo.CyclerParams([C] * (200 - i) + [D] * i))
        pop.append(axl_dojo.CyclerParams([D] * (200 - i) + [C] * i))

    # Matching Tails
    for i in range(1, 6):
        for j in range(1, 6):
            pop.append(axl_dojo.CyclerParams([C] * i + [D] * (200 - (i + j)) + [C] * j))
            pop.append(axl_dojo.CyclerParams([D] * i + [C] * (200 - (i + j)) + [D] * j))

    # Alternating
    pop.append(axl_dojo.CyclerParams([C, D] * 100))
    pop.append(axl_dojo.CyclerParams([D, C] * 100))
    pop.append(axl_dojo.CyclerParams([C, C, D, D] * 50))
    pop.append(axl_dojo.CyclerParams([D, D, C, C] * 50))
    pop.append(axl_dojo.CyclerParams([C, C, C, C, D, D, D, D] * 25))
    pop.append(axl_dojo.CyclerParams([D, D, D, D, C, C, C, C] * 25))
    pop.append(axl_dojo.CyclerParams([C, C, C, C, C, D, D, D, D, D] * 20))
    pop.append(axl_dojo.CyclerParams([D, D, D, D, D, C, C, C, C, C] * 20))

    seq_len = 200
    while len(pop) < pop_size:
        random_moves = list(map(axl.Action, np.random.randint(0, 1 + 1, (seq_len, 1))))
        pop.append(axl_dojo.CyclerParams(random_moves))

    return pop
```

Figure 4.18: Initial Population Code

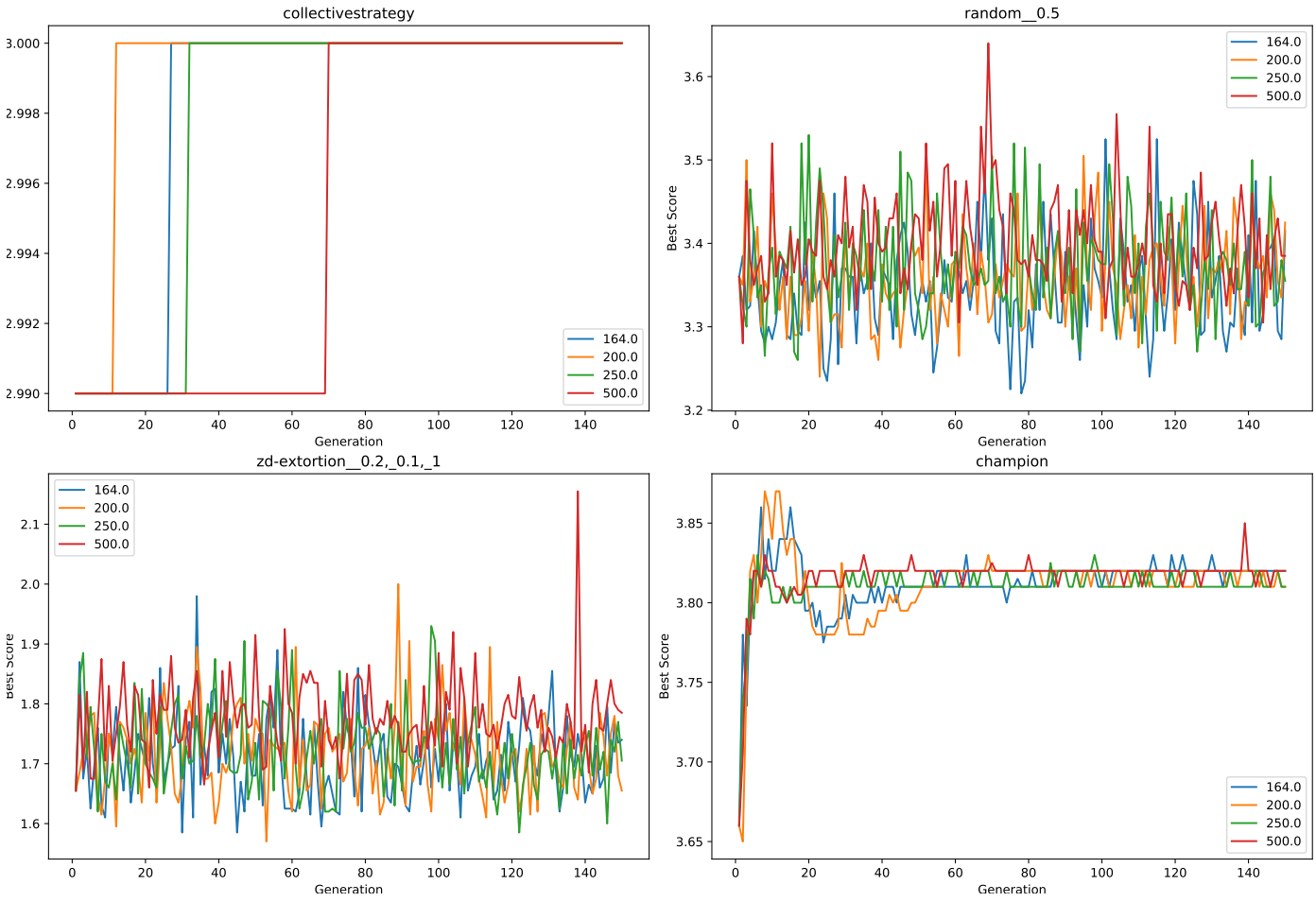

Figure 4.19: Non optimal sequence players after changing initial population

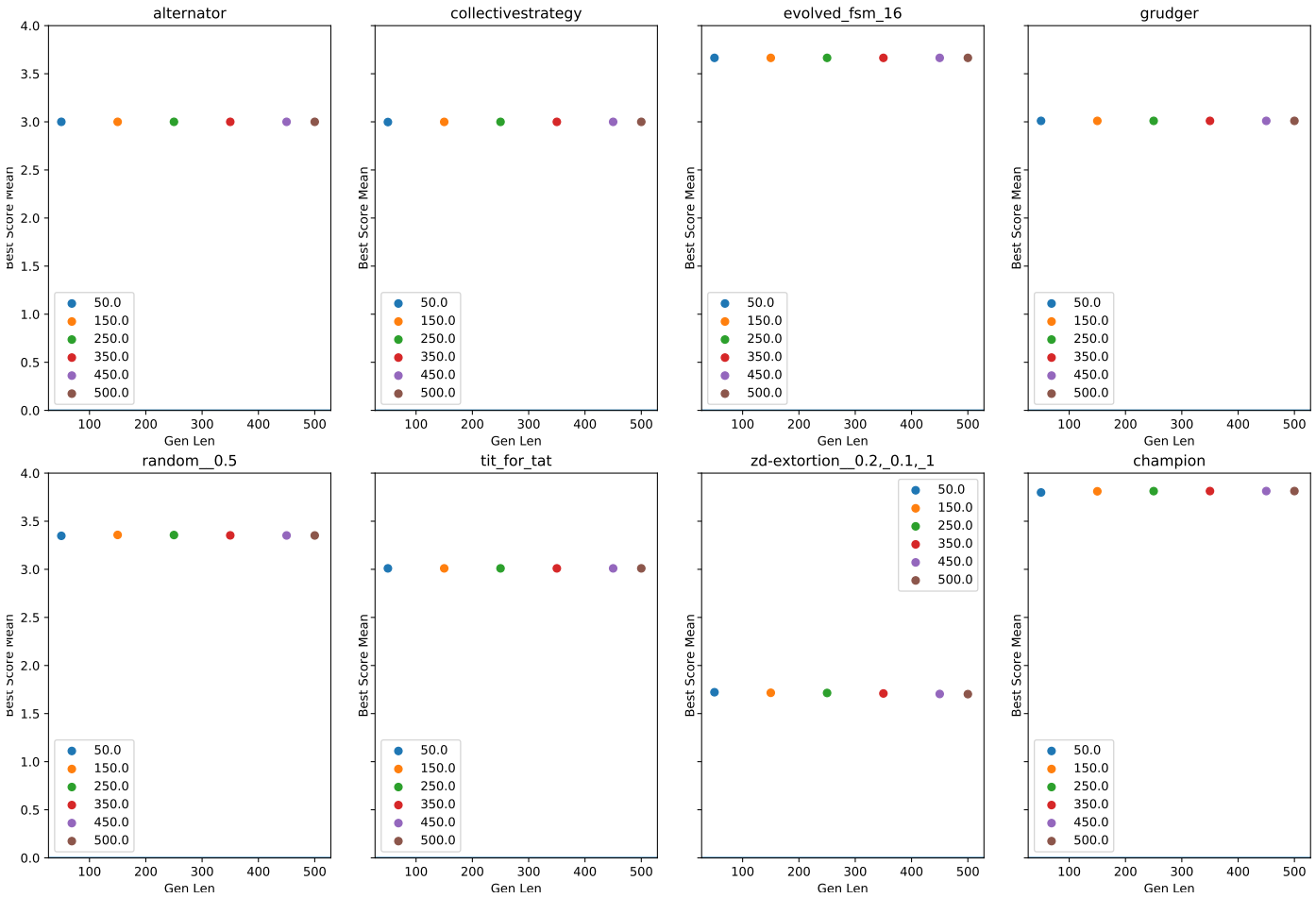
alternator
collectivestrategy
evolved_fsm_16
grudger
random__0.5
tit_for_tat
zd-extortion__0.2,_0.1,_1
champion
Best Score Mean
Gen Len
50.0
150.0
250.0
350.0
450.0
500.0

Figure 4.20:

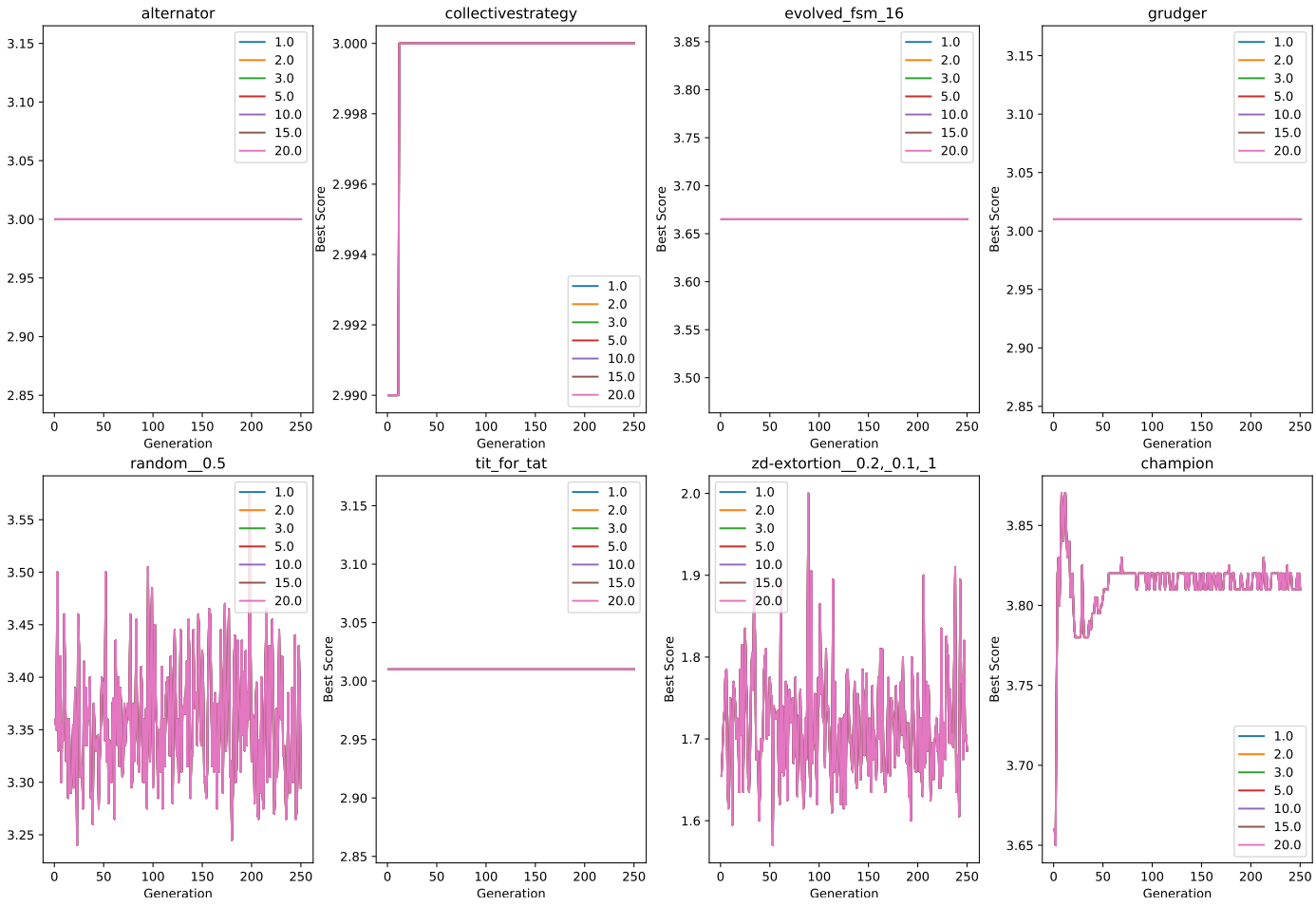
alternator
collectivestrategy
evolved_fsm_16
grudger
random__0.5
tit_for_tat
zd-extortion__0.2,_0.1,_1
champion
Best Score
Generation
1.0
2.0
3.0
5.0
10.0
15.0
20.0

Figure 4.21:

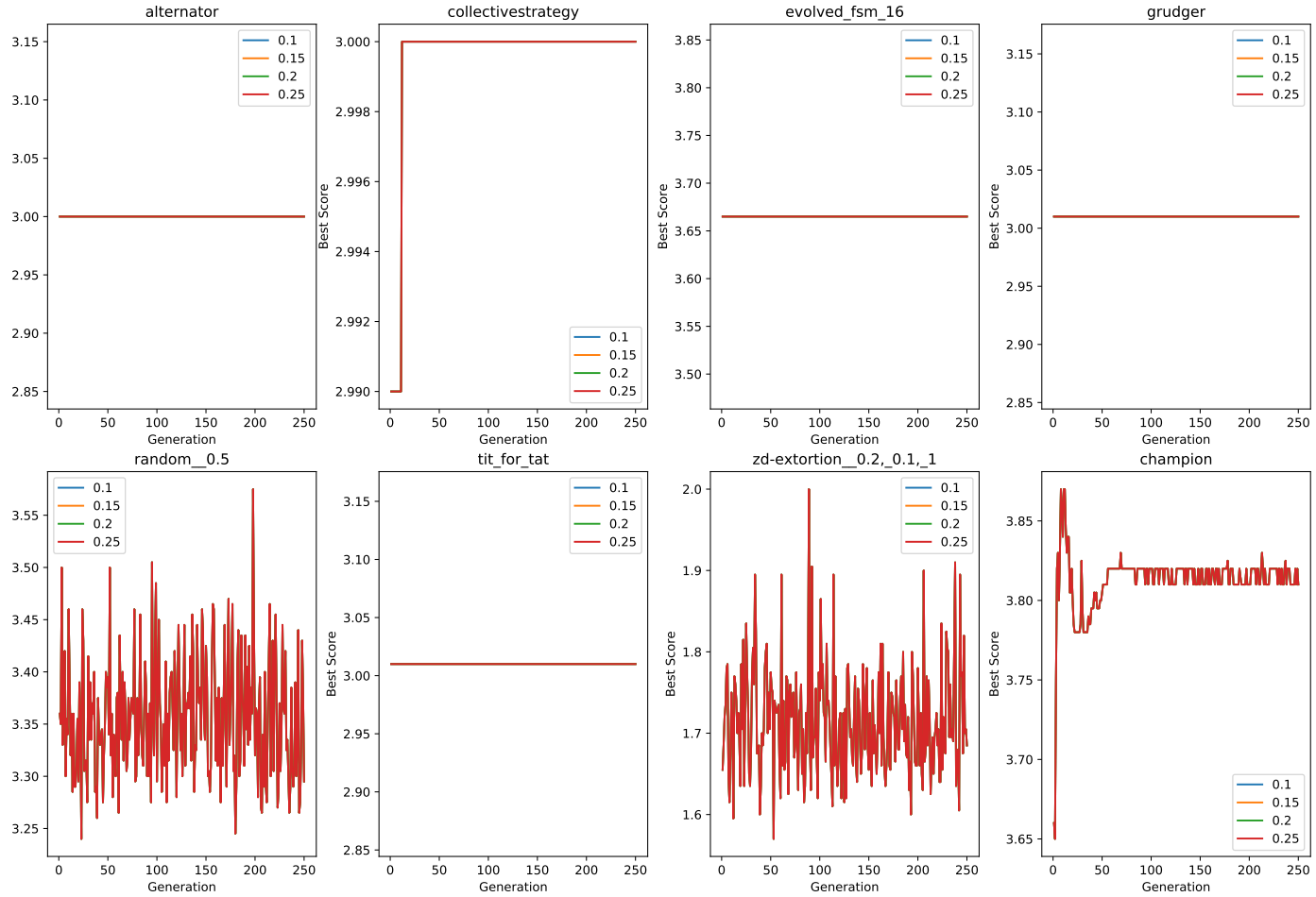

Figure 4.22:

```python
def getSeededPlayer(player_class):
    class NewClass(player_class):
        def __init__(self, seed=0):
            axl.seed(seed)
            super().__init__()

    return NewClass
```

Figure 4.23: A function for wrapping a player with a global seed function call

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. If we can score an average of 3.8 against op1 or 3 against op2, we play op2 etc.
game shows to win money? We want to win as much money as possible but we don't mind how much our opponent wins, as long as we get lots.]
What were actually doing is finding a sequence that will manipulate our opponent into providing us the most number of cooperation moves we can subsequently defect against. Failing that we are manipulating the opponent for cooperation moves we can cooperate against, then defection moves we can defect against, then defection moves we must cooperate for.

Chapter 7

Summary and Future Research

List of Figures

List of Tables

Bibliography

[CB97] Paul C Chu and John E Beasley. A genetic algorithm for the generalised assignment problem. *Computers & Operations Research*, 24(1):17–23, 1997.

[Chu17] Michael Chui. Artificial intelligence the next digital frontier? *McKinsey and Company Global Institute*, page 47, 2017.

[DB09] Charles Darwin and William F Bynum. *The origin of species by means of natural selection: or, the preservation of favored races in the struggle for life*. AL Burt, 2009.

[Kni17] Will Knight. Alpha zeros alien chess shows the power, and the peculiarity, of ai. *MIT Technology Review*, 2017.

[Tur50] Alan M Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.