The Iterated Prisoner's Dilemma

Generating Solution Sequences for Strategies Using Genetic Improvement Algorithms

Toby Devlin

B.Sc. Final Year Dissertation

Cardiff School of Mathematics

Contents

1 Introduction **4**

1.1 The Prisoners Dilemma and Its Iterated Form 4

1.2 Machine Learning & Computer Intelligence 5

1.2.1 Genetic Algorithms 5

1.3 Brief Task Overview 6

1.4 Conclusion & Structure of This Report 7

2 Task Methodology **8**

2.1 Notation 8

2.2 Customizing an Algorithm 9

2.3 Solution Form 10

2.4 Conclusion 10

3 Literature Review **12**

3.1 Background 12

3.2 Strategy Structures 13

3.2.1 Equivalent Strategies 13

3.3 Strategies Of Interest 14

3.4 Genetic Algorithms 14

3.5 Conclusion 15

4 Developing The Codebase **16**

4.1 Codebase Contributions 16

4.1.1 Version Control 16

4.1.2 Testing 17

4.2 Libraries And External Modules . . . 18
4.3 Reproducing Analysis . . . 18
4.4 Conclusion . . . 20

5 Implementation Of Sequence Discovery **21**

5.1 Background . . . 21
5.2 Changing Initial Population Size . . . 22
5.3 Generation Length Analysis . . . 23
5.4 Changing Mutation Rate . . . 25
5.4.1 Changing Mutation Potency . . . 26
5.4.2 Changing Mutation Frequency . . . 27
5.4.3 Conclusions of altering Mutation parameters . . . 28
5.5 Mitigating Local Maximum Solutions . . . 28
5.5.1 Ineffective Approaches of Altering Crossover And Mutation . . . 30
5.6 Altering Initial Population . . . 31
5.6.1 Constructing a population . . . 32
5.6.2 Population Size . . . 33
5.6.3 Generation Length, Mutation Potency & Mutation Frequency Analysis After Change of Initial Population . . . 34
5.7 Stochastic Opponents . . . 34
5.8 Conclusion of approach . . . 35

6 Results and Discussion **41**

6.1 Resulting Data . . . 41
6.2 Solution Distance Matrices . . . 43
6.3 Solution Groups . . . 44
6.4 Clustering Analysis . . . 45
6.4.1 K Means clustering [46] . . . 46
6.4.2 Regression Trees [48] . . . 46
6.5 Conclusion . . . 47

7 Conclusions **51**

7.1 Reflection of Approach . . . 51

7.2 Summary of Analysis Execution 51
7.3 Applications of results 52
7.4 Potential follow up work 52

A Online Resources **54**

B Code Appendix **55**

C List of Axelrod Opponents **62**

D List of Best Solution Sequences **64**

E Further Figures **97**

Chapter 1

Introduction

In this report we will be looking for best responses of strategies in the Iterated Prisoners Dilemma (IPD); we will try to identify the best overall score, and its corresponding set of moves, for a player on the other side of the game to the strategy. We will look into explicit sequences which should be played for 1 vs 1 games of length 200 to gain the highest score per turn upon the games conclusion.

This task will be completed using reinforcement techniques, specifically genetic algorithms, to continuously improve our sequences during training. Once we have best responses, these sequences will then be compared and contrasted to understand how opponents could be grouped together even though they have no structure in common.

This work assumes a basic knowledge of game theory and how to model a normal form game. Knowledge of the Prisoners Dilemma is helpful but not required, the IPD game itself is described in Section 1.1 and TODO are discussed in Chapter 3. There is also the assumption the reader is familiar with basic algorithms and programming.

1.1 The Prisoners Dilemma and Its Iterated Form

The Prisoners Dilemma (PD) is a normal form game in the space $\mathbb{R}^{2\times 2^2}$ with utility matrices and mixed strategies for the row and column player as follows:

$$A = \begin{pmatrix} R & S \\ T & P \end{pmatrix} \quad B = \begin{pmatrix} R & T \\ S & P \end{pmatrix}$$

$$\sigma_r = \begin{pmatrix} C & D \end{pmatrix} \quad \sigma_c = \begin{pmatrix} C & D \end{pmatrix}$$

with the following utility constraints:

$$T > R > P > S, \qquad 2R > T + S$$

These constraints mean that the defection action, D, dominates the cooperation action, C, for both player and that the largest payoff comes when both players cooperate. This payoff model has the following interpretation:

- T — Temptation, the utility for successfully tricking your opponent to cooperate while you defect.
- R — Reward, by both coopering with your respective opponents you both receive the reward.
- P — Punishment, if both you and your opponent defect you will both receive the punishment
- S — Sucker, the utility for being tricked by your opponent to cooperate while they defect; they get temptation and you get sucker.

Because of the way the game is put together we arrive at the dilemma. Do you cooperate and risk being taken advantage of by your opponent, first row or column. Or do you defect and hope your opponent cooperates to give you a bonus,

second row or column. In this work we will be using the T, R, S & P utilities originally introduced in axelrods 1980s work [1].

$$A = \begin{pmatrix} 3 & 0 \\ 5 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 3 & 5 \\ 0 & 1 \end{pmatrix}$$

From a game theoretic point of view it can be seen that there exists a strongly dominant strategy for both the row and column player: defection. This however does not reflect on the iterated version of the game where repetition can allow for higher overall scores and players can take advantage of each other.

We will be looking into the iterated form of the game which, simply put, is a series of single games played back to back. Players do not know what their opponent will play on any given turn, they only know what has been played on previous turns. Players can algorithmically, stochastically or using a combination of the two decide on their next move. The number of turns is often provided beforehand, but does not need to be [2], and the overall score of the game is normalised to by the number of turns to allow for comparisons between games of different length.

1.2 Machine Learning & Computer Intelligence

Machine learning (ML) was most famously introduced by questions posed by Alan Turing in 1950 [3] asking '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 [4] on topics which would traditionally be considered 'thinking'. Along with this, recently there has also been record levels of funding [5] put in to companies which operate in this field, producing results in areas that would usually seem 'solved'. For example TODO.

Computer intelligence breaks down into 2 general topics, supervised and unsupervised learning. Supervised learning is done with help from an external source; inferring a function from labelled examples that will be used to map unseen data to their potential labels. Unsupervised learning is the process of creating new data about a dataset which will be used to expand the understanding of existing data; for example finding a hidden structure or searching a solution space. Reinforcement learning is a technique that can be applied to both supervised and unsupervised learning. What it means is to repeat an algorithm using results from a previous run to improve parameters of its next run; genetic algorithms are a prime example of this.

1.2.1 Genetic Algorithms

This report will cover a class of ML called genetic algorithms (GAs). GAs fall under a branch of ML called evolutionary algorithms. More generally, genetic algorithms are put into a class of unsupervised reinforcement learning algorithms. These represent techniques of using genetic algorithms for generating solutions to problems that typically revolve around heuristically improving members of a population who represent these solutions,[6, 7]. The concept of a genetic algorithm, and more generally an evolutionary algorithm, comes from nature; like nature we create a survival of the fittest selection [8] 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 (the terms 'genes' and 'member properties' are interchangeable), and a number of generations, $G \in \mathbb{N}$, the algorithm will create G cycles of scoring and potentially removing each of the members of the population, $p_i \in P$. It does this by using a mapping from a member of the population to an ordered set, for example $f(p_i) \mapsto \mathbb{R}$. This function, f, is defined beforehand in a way which describes the goal of our investigation. Defining a cut-off or bottleneck $b < |P|$, such that on conclusion of completing any cycle, the top b ranking members (or proportion of members) by score can be kept and the rest discarded. By doing this we are saving the more successful candidates allowing us to rebuild the population using a series of crossovers and mutations (and possibly introducing new members into the population) with the genes which were successful.

- 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 1.1 shows a flow diagram of a generic genetic algorithm cycle. The specific algorithm we will be using is described in Subsection 2.2.

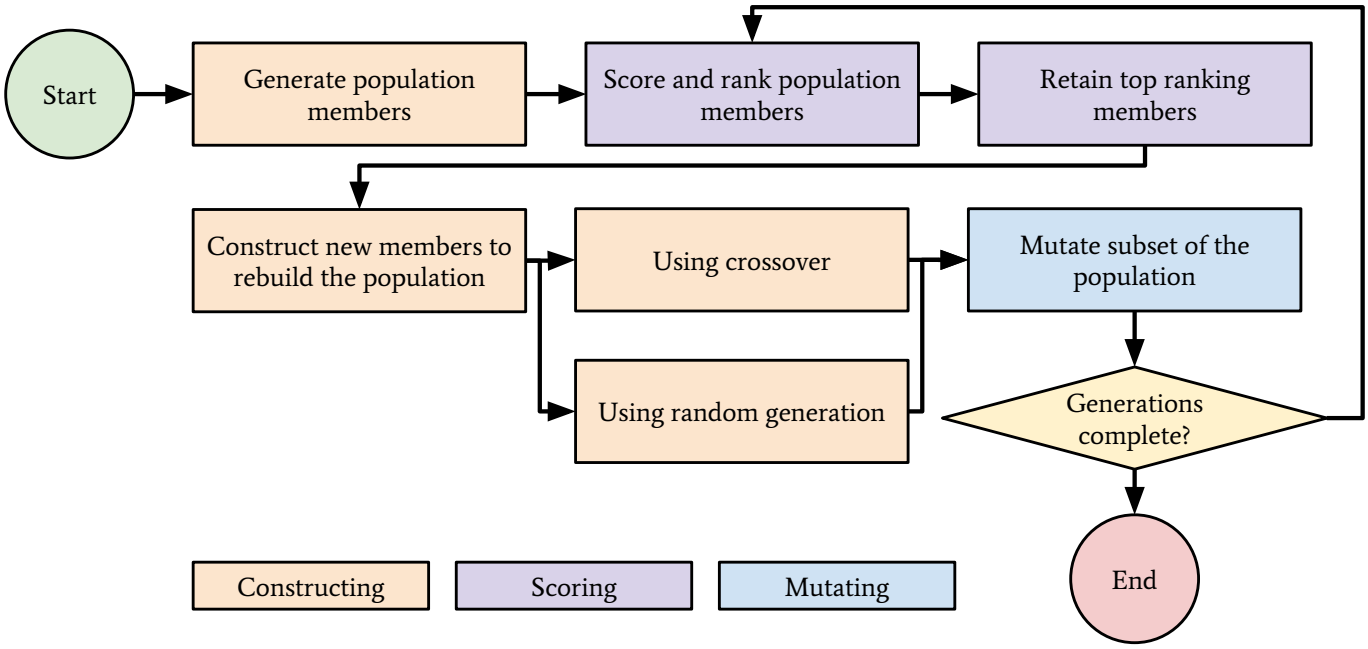

Figure 1.1: Generic genetic algorithm cycle diagram

Genetic algorithms are designed in way to avoid local maxima/minima when searching a solution space. The property of mutating a member of its population allows alterations to the members features; this in turn will lead to the change in that members fitness. If a member is 'stuck' in a local maxima/minima with its fitness score, a mutation of a certain feature could potentially move their position in the solution space to better local maxima/minima. Along with mutation, the crossover property explicitly combines 2 members of the population to create a new member. Crossover adds more variance in the population by taking properties of both parent members and introducing the evaluation of their combination to the solution space. By doing this the algorithm can search multiple local maxima/minima at once with its population and start to identify the global maxima/minima as members become more optimized.

1.3 Brief Task Overview

This work will look at discovering which sequences provide the highest score in a game for each given opponent in the Iterated Prisoners Dilemma. For every game of length n there are 2^n possible combinations of moves we can make against an opponent, we are trying to find the sequence which provides us with the highest score overall.

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. Our task is as follows:

Problem:

When playing a given Iterated Prisoners Dilemma strategy as an opponent, what is the best ordered sequence of moves

[1]For example using intuition and targeting specific genes, or allowing another algorithm to improve the targeting of this meta function.

to play in order for us to obtain the highest possible average score per move across the game? Are there any patterns in these solution sequences?

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 easy example and a simple strategy to calculate a solution for due to its explicit rules. The ultimate goal of this investigation is to look into every strategy as defined in the open source Axelrod Library[9], where over 200 strategies are implemented. These opponents are listed in Appendix Section C and definitions can be found in the Axelrod Library codebase.

This problem can be reduced to searching the solution space of every single possible combination of C and D moves. With a game length of 200 however, this is an infeasibly large number of checks and so we will use the genetic algorithm to selectively improve potential sequences. Some examples of 5 move games against Tit For Tat are shown below. The optimal solution for Tit For Tat in this scenario is game 5 whereas all the others are sub optimal. This is the process we will be doing for every opponent in the context of a genetic algorithm.

Game 1	1	2	3	4	5	final score	per turn
Tit For Tat:	C	D	D	D	D	4	0.8
Solution:	D	D	D	D	D	9	1.8

Game 2	1	2	3	4	5	final score	per turn
Tit For Tat:	C	D	C	D	D	7	1.4
Solution:	D	C	D	D	D	12	2.4

Game 3	1	2	3	4	5	final score	per turn
Tit For Tat:	C	D	C	D	C	10	2.0
Solution:	D	C	D	C	D	15	3.0

Game 4	1	2	3	4	5	final score	per turn
Tit For Tat:	C	D	C	C	C	11	2.2
Solution:	D	C	C	C	D	16	3.2

Game 5	1	2	3	4	5	final score	per turn
Tit For Tat:	C	C	C	C	C	12	2.4
Solution:	C	C	C	C	D	17	3.4

Game 6	1	2	3	4	5	final score	per turn
Tit For Tat:	C	C	C	C	C	15	3.0
Solution:	C	C	C	C	C	15	3.0

1.4 Conclusion & Structure of This Report

This chapter contained introduction to content that will be needed and expanded on moving forward. All the topics mentioned in this chapter are core themes to what the rest of this report focuses on.

This report will contain the following chapters:

- Chapter 2 describes methodology and concepts that will be needed to understand results in later chapters.
- Chapter 3 looks into previous work and literature behind the concepts we will be working with.
- Chapter 4 looks into the programming and development that was undertaken to complete the work.
- Chapter 5 looks at how the application of our genetic algorithm was improved before running the final analysis.
- Chapter 6 is the chapter communicating the results of the final analysis.
- Chapter 7 describes the takeaways and possible applications of the work.

Chapter 2

Task Methodology

This chapter will cover sections on how each part of the project will been carried out. Useful notation is given in Section 2.1 which will allow us to describe a given sequence or set of sequences in a compact form. We will also look at our specific instance of genetic algorithm that exists in the Axelrod Dojo [10] codebase. Lastly Section 2.3 looks at the what results we will expect from the analysis and the potential problems we want to mitigate.

2.1 Notation

As described in Chapter 1 this work focuses on identifying best responses against the set of opponents in the Python Axelrod Library. The best responses are in a sequence format with a length of 200. In this Section we introduce a notation which will allow us to represent such sequences in a concise format.

Let S be a sequence where $S \in \{C, D\}^L$ and C, D represents a cooperation, defection respectively. $L = 200$ is used throughout this report. Using the binary nature of the sequence elements, 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, C or D; the type is implicit based on whether i is even or not and what the starting move type was.
- We can use the notation $|B_i|$ to denote the length of the ith block (number of moves within the 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 represented shorthand by specifying the starting move and the length of subsequent blocks:

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

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

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

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 should not grow.

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

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. When we look at solutions there too long to write the sequence directly, in which case we may abbreviate in the following style: $C1,5,2,3,5,6,... = CDDDDDCCDDDCCCCCDDDDDD....$

Examples:

$$C1,4,3,2 = CDDDDCCCDD \tag{2.1}$$
$$D(1,1)^5 = DCDCDCDCDC \tag{2.2}$$
$$C1,(2,1)^2,2,1 = CDDCDDCDDC \tag{2.3}$$
$$\{D[i,5-i]\} \quad i \in [2,4] = \{DDCCC,\ DDDCC,\ DDDCC\} \tag{2.4}$$
$$\tag{2.5}$$

To avoid confusion between a sequence for a given opponent, S_{O_i}, and the score for a given sequence we define the score function $f(S_{O_i})$ as follows.

$$f(S_{O_i}) : S_{O_i} \to [0,5]$$

This score function represents playing the sequence against the given opponent and calculating the score per turn over 200 turns. In this report the score will not be given its own notation other than the result of the function f.

2.2 Customizing an Algorithm

In order to generate a solution sequence we have to train against each Axelrod opponent. We will expand on the GA described in Section 1.2.1 to produce our own. Our implementation of a GA 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 are 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.[1]
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 2.1 shows a flow diagram of our cycle. This is the algorithm we will use in Chapter 5 when analysing the algorithms parameters. The model means we can create a population of Cycler players, described in Section 3.3, 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 looping will be the basis of creating the optimal strategy for each other opponent. Each step is defined in the Axelrod Dojo `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.

[1] This algorithm had a bug which would change the size of the population in the first generation. This was fixed after Section 5.8 was written. See: https://github.com/Axelrod-Python/axelrod-dojo/issues/43

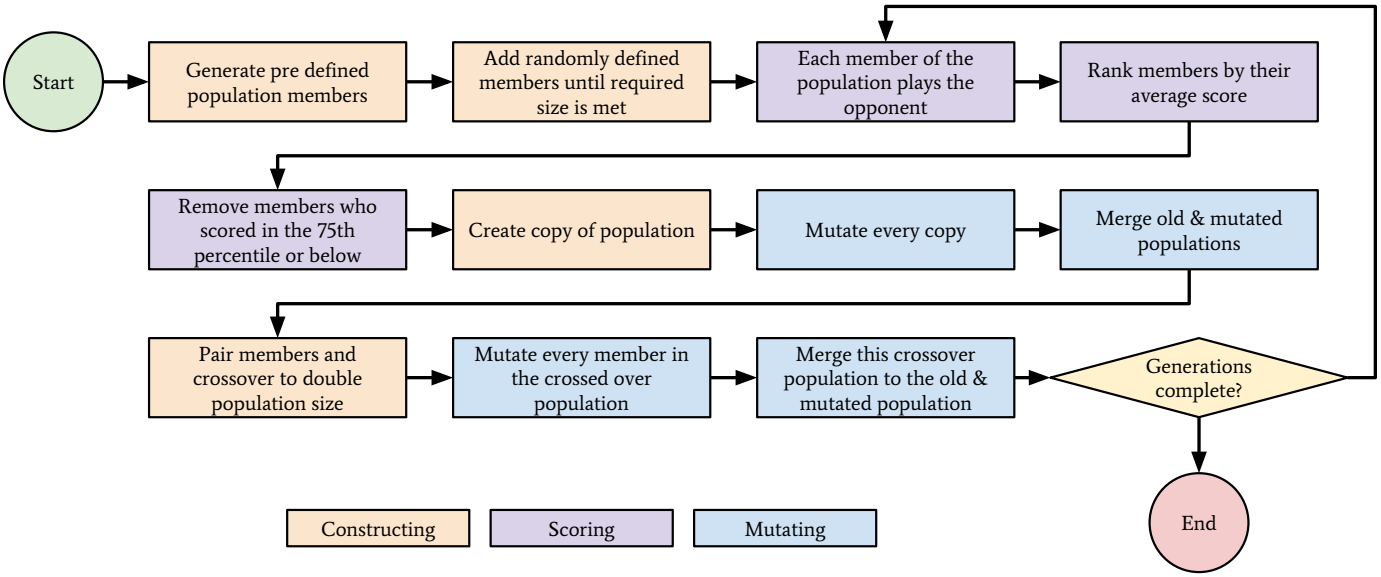

Figure 2.1: High Level Genetic Algorithm Cycle. Extension of Figure 1.1

2.3 Solution Form

In this research our goal will be to use the algorithm, described in Subsection 2.2, to produce an arbitrary sequence for an opponent, S_o. This sequence will represent the moves we should play against the opponent to get our largest potential score per turn for a single game of 200 turns.

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 7.

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 individual solutions will be generated.

If needed, we may re-run analysis on some opponents if there are bugs that arise in the code. In this case older results will be overwritten and previous analysis discarded.

When working with stochastic players, we will be seeding the random numbers they use 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 described in Section 5.7. Non-stochastic opponents will be seeded in the code, however there is no change in their behaviour because of this.

2.4 Conclusion

Here we have looked at the background required to answer a specific question, build a relevant model and understand what were looking for. The concepts themselves are extensions to the basic models brought up in Chapter 1 and should allow us to provide relevant results in a uniform manner.

We have also looked at what problems may arise from certain areas of solving the task, such as understanding what a best response for a stochastic opponents could be or how to represent a complex solution in a result. Because of this premeditation of the problem and its components Chapters 5 and 6 will be more concise.

After we conclude the runtime of the GA and obtain a solution we will have an output sequence for each opponent. These solution sequences can then be compared on how similar they are to each other; what type of opponents and their corresponding 'best scores' are and whether there are patterns to how to group strategies, Chapter 6 contains further details.

Chapter 3

Literature Review

In this chapter we will look at previous works in areas relevant to this report. We will look at how these pieces of work developed standards in the areas we are interested in and the history of studying them with computer models.

Section 3.3 describes interesting opponents we will look at in more detail during Chapter 5 to provide our algorithm a variety of opponents to be developed against.

3.1 Background

The PD and its a large area of repeated games in GT and has applications in the real world. This is due to the game being a good example of strategies that give a cooperative benefit to repeated games. The PD was first formally presented by Albert W. Tucker [11, 12] and the iterated version was made famous by Robert Axelrods work in the 1980s [1]. It has been used to describe actions of people and governments in situations stretching from warfare [13, 14] and finance [15] to politics [16] and relationships [17].

The Prisoners dilemma became a large research area in the combination of mathematics and computer science after Robert Axelrod published his work named 'Effective Choice In The Prisoners Dilemma' [1]. In it he makes an introduction to how tournaments are run and the properties of successful results. His method of experimenting became the standard for handling the IDP problem. This was also the first example of computer tournaments, for which he asked for strategies as computer programs which had inputs of the history for both players and resulted in an output move for that next turn. After the tournament is complete he describes what successful strategies had in common; It turns out the majority of successful strategies, including the winner, Tit For Tat, have properties of niceness and forgiveness. Niceness is the property of not defecting to start and forgiveness is the property of forgetting previous defections in a timely manner. This allowed them to thrive in the tournament and have overall scores that rose above strategies without niceness or forgiveness.

Since Axelrods' original tournament there have been many research papers on what makes a successful strategy for a specific objective. For example William H Press and Freeman J Dyson, [18], looked at how to it is possible to deterministically set an opponents score and Shashi Mittal and Kalyanmoy Deb have looked at satisfying a range of different objectives at once [19]. These specific objectives are the core part of applying GT and the PD in the real word [20, 21, 22]. Objectives are the wrapper for which we can work with real world scenarios, for example in a the cold war it was not the goal to get as many missiles as possible to pass an oppositions defence but to not allow any missiles through your own defence; i.e.minimising your opponents score. In a football tournament where winning is the number or goals your team scores it does not matter how many goals you get in so long as you score the most overall. This leads to observing that the IPD describes that a players world is better if their opponent is cooperative. There are also some very useful applications levering these ideas, such as modelling rent splitting or work assignments [23]; all of which were programmed and deployed to a website for use online at [24].

3.2 Strategy Structures

A strategy is a way of defining how to play the IPD. It is a way of identifying which move to play next based on some parameters; for example the strategy of all C moves is valid, as is a wildly complicated description of what to do based on the last 20 moves of an opponent. Some examples, further looked into in Section 3.3, include creating strategies that extort others to create overall scores it desires or strategies with handshakes for self identification [25].

Individual strategies can be defined in multiple ways [26]. Each method of representing a strategies has its benefits and drawbacks, typically due to fundamental properties of the strategy (like being stochastic) or that it is more concise to write in a certain way. Ways of structuring a strategy are shown below:

- LookerUpper, for example figure 3.2.
- Gambler, for example the Stochastic strategies; examples can be found in [18].
- Neural Networks.
- Finite State Machines, for example figure 3.1.
- Hidden Markov Models.
- Explicit Move List, for example the solutions given in Appendix D.
- Mixtures.

3.2.1 Equivalent Strategies

Looking at certain opponents there are occasions some strategies that look indistinguishable from others; for example two stochastic opponents can can be incredibly similar, but identifying which one is which from their play history could be impossible. One of the ways we are able to identify strategies is the process of fingerprinting [27, 28, 11]. However this can be inconclusive and less accurate than desired. An effective method of identifying equivalent strategies is to write them down in the form of the other. For example if we can write any strategy in one of the forms given in figures 3.1, 3.2 we know its an equivalent to Tit For Tat.

The work being completed in this paper is another method of identifying strategies; identifying the best response to them may lead to observations about how and why different strategies act in similar manners. This wont lead to show strategy equivalence, but it will show how solution equivalence.

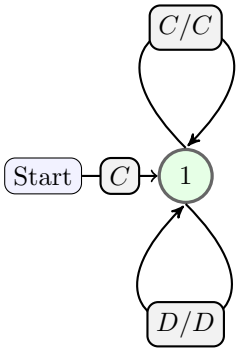

Figure 3.1: Finite State diagram of strategy Tit For Tat

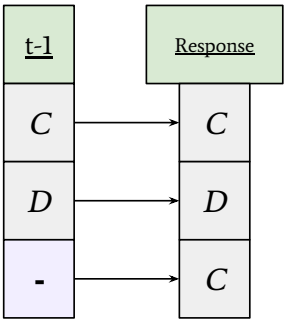

Figure 3.2: Look Up diagram of strategy Tit For Tat

3.3 Strategies Of Interest

Tit For Tat was the winner of the original axelrod tournament in 1980 [1]. It s a very basic opponent who has strong forgiveness (it will forgive a defection after 1 move) and generosity (it will start with a cooperation) which is thought to give its well overall score in tournaments.

Alternator is a 'dumb' opponent, i.e. no strategic method at all. All it will do is alternate between cooperation and defection until the game ends. Playing against an Alternator effectively is to defect the whole game; identifying an alternator to play this sequence of defections without backlash can be difficult. If we are playing a similar starting strategy such as Collective Strategy defecting in the first two moves will cause us to sore badly overall.

Grudger can be considered as the most unforgiving strategy that exists. Starting by cooperating, if you defect even once then the Grudger will defect until the end of the game. In line with Tit For Tat, our best score will come from not 'upsetting' the opponent until the last move where it cant retaliate.

Random is the most basic stochastic opponent, and like Alternator and Cycler it is 'dumb'. With a probability p of cooperating and $1 - p$ of defecting, we can just defect the whole time to beat this opponent; picking up bonus points on its cooperation moves.

EvolvedFSM16 is a 16 state Finite State Machine (FSM) that has been trained with an evolutionary algorithm. It has been optimised for tournament matches, the definition can be found in the axelrod documentation [9].

CollectiveStrategy was defined in [29]. It always cooperates in the first move and defects in the second move. If the opponent also cooperates in the first move and defects in the second move, CS will cooperate until the opponent defects. Otherwise, CS will always defect..

ZDExtortion is first mentioned in the paper *Iterated Prisoners Dilemma contains strategies that dominate any evolutionary opponent* [18]. In it the authors discuss methods of extorting score from another player. We use this player an example of a difficult to beat player.

Cycler is a strategy which cycles and repeats a sequence until the end of the match. For example `cycler("CCD")` will play $C2, 1, 2, 1, 2, \ldots$ for as long as necessary. The Cycler strategy will be directly used in the GA by using the current solution, of length 200, as the parameter to play a specific absolute sequence during a generations scoring round. This cycler class will contain our solution sequence and the sequence can be extracted for mutation & crossover as needed.

3.4 Genetic Algorithms

This work will focus on GAs, which are a specific form of ML. Advanced techniques of ML can be combined and used together[1]. The field of mathematics research is one which has plenty of examples of GAs in action; for example the same techniques as we will using is used in [31] to solve the Generalised Assignment Problem. In it the authors discuss how to solve an NP-complete problem by exploring solutions in the problem space using a GA. This is similar to what the result of this report will contain; we will be searching the problem space (Every possible sequence permutation of C and D with length 200) to create solutions to the problem described in Chapter 2.

GAs are not limited to mathematics, in [32] there are discussions around the applications of GAs in art and music. The goals of a GA have to be defined beforehand and can be as abstract as we like, both art and music are abstract and would require careful consideration for what the algorithms fitness function would do. Because of their flexibility GAs are used in a wide range of application in many areas including antenna design [33] and creating cost effective networks [34].

[1] An example includes techniques for teaching and developing static algorithms. Building a video game AI where the core goal of the AI is fine tuned using a GA in an development environment, then only the trained algorithm and not the training is implemented into the game [30].

3.5 Conclusion

In this chapter we have looked at previous work in areas of the IPD and GAs. Importantly we have looked at structures of strategies and some various examples that would be good to peruse when looking into constructing an effective GA. We have also outlined how strategies can be linked together through certain properties, such as their structure. As we continue looking at solutions for certain strategies we will be constructing another grouping of strategies in their solution sequences. This may lead to finding similarities to opponents who seem to have vastly different in structure.

Chapter 4

Developing The Codebase

This chapter will discuss the development practices encountered during the project and a walkthrough of setting up a research environment. Details on the set up and reasoning behind using specific development environments will be given along with a look at the open source contributions made. A brief look at version control will also show how modern code is developed to be sustainable and reproducible.

4.1 Codebase Contributions

Throughout the project I had split time between writing my own code and expanding the Axelrod Dojo codebase. In modern software development there is a commitment in the developer community to track and reuse as much code as possible, typically using a version control system such as Git or Subversion. The majority of open source community development is conducted on GitHub[35] using Git. The Axelrod libraries, along with most other libraries I used, are primarily hosted here.

When writing code in a professional environment there is a predetermined scope that all parties agree upon before the work commences. However research development is more organic; final products that are created when conducting research for papers are highly personalised and typically cannot be used in other areas. This leads to more flexible products that operate more as platforms or tools for further research. The Axelrod, Axelrod Dojo and other libraries mentioned in table 4.1 exist as platforms because of this and the open source drive for reusable code. My final code will only be used for researching the IPD in my projects specific direction, as I worked on the project I had to extend the platforms in order to handle what my project needed them to do, subsection 4.1.1 looks into how this was completed in my project.

4.1.1 Version Control

During the project I created a 'fork' of the Axelrod Dojo in order to add content to the open source community. This resulted in using Git to create a new 'branch' on which to write my code before asking the owners of the repository to pull my work back into the core product using a 'pull request'(PR). The PR opened for my code[1] resulted in changes to 457 lines of code and 14 files, adding classes and fixing bug that had been previously flagged. Figure 4.1 shows the initial scope of the PR and Figure 4.2 shows a section of the commits made before merging the branch.

After a request is opened there is a period of reviews by the owners, this is to ensure code quality and scope coverage. As my work progressed I continued to add to the PR which lead to more and more requests being added for features that fell outside the scope of the PR. Eventually we decided to create another fork of the PR that contained specialist code for my project that would not benefit the codebase as a whole. Figures 4.3 & 4.4 show examples of requests and discussions around features and code quality. Because of this review process the overall quality of the codebase can be kept high and

[1] https://github.com/Axelrod-Python/axelrod-dojo/pull/45

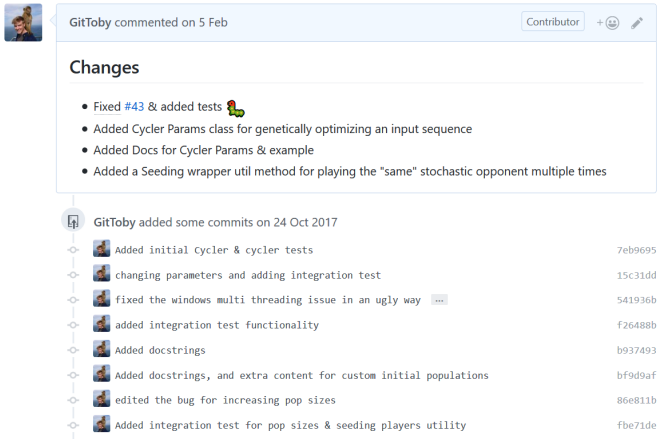

Figure 4.1: Description and commits for PR on Github

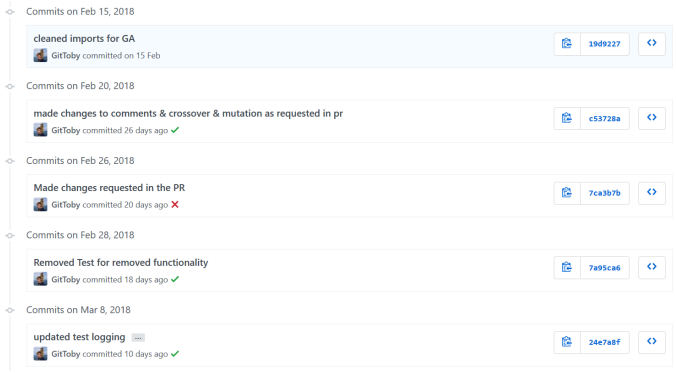

Figure 4.2: Tail of code commit log as shown on Github

the owners can decide on how their platforms are developed.

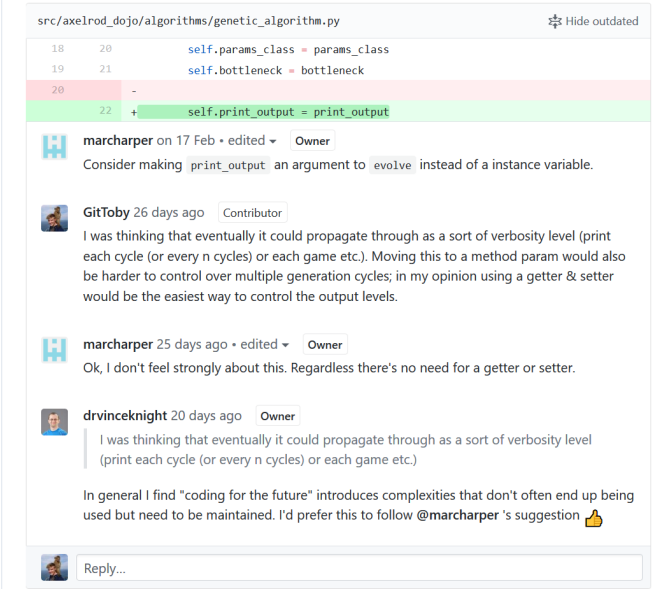

Figure 4.3: Feature discussions in PR on GitHub

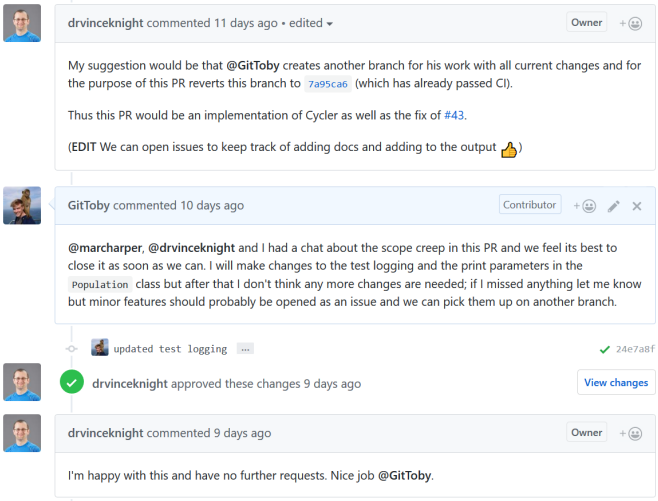

Figure 4.4: Scope Discussion in PR on GitHub

4.1.2 Testing

Code testing and version control go hand in hand. When new releases of code are made it is important to ensure that the new changes dont break previous functionality. This is kept in check by the presence of continues integrations (CI) and test environments. As of this project the CI for the Axelrod Dojo keeps the libraries tests running on a linux environment and the output fed to the Github page. During my development I had to implement tests for any functionality I wrote to ensure the library still worked as intended. Following the mixed practice of test driven design (TDD) and behaviour driven design (BDD) the code written to extend the Axelrod Dojo had tests to cover examples of what happens in a production environment [36, 37, 38]. Snippet 4.5 shows an example of a test.

```
def test_creation_seqLen(self):
    axl.seed(0)
    test_length = 10
    self.instance = CyclerParams(sequence_length=test_length)
    self.assertEqual(self.instance.sequence, [D, C, C, D, C, C, C, C, C, C])
    self.assertEqual(self.instance.sequence_length, test_length)
    self.assertEqual(len(self.instance.sequence), test_length)
```

Figure 4.5: An Example of a test in the Axelrod Dojo

4.2 Libraries And External Modules

Main Research Libraries

- **Axelrod** — Used for the core of the prisoners dilemma and iterated prisoners dilemma functionality code [9].
- **Axelrod Dojo** — Applied machine learning techniques that revolve around generating solutions to questions relating to the Axelrod library [10].

Functional libraries Table 4.1 shows the external functional libraries used, while Table 4.2 shows the internal python built in modules that where leveraged during development. These are libraries which are not involved in the core functionality of the IPD.

Library	Reason	Reference
matplotlib pyplot	For plotting graphs and images with data	[39]
pandas	For data manipulation.	[40, 41]
numpy	For reducing complexity of numerical calculations.	[42]
SciKit Learn	For existing ML tools.	[43]

Table 4.1: Functional Python libraries for analysis

Module	Reason
os	For operating system functionality.
time	For time calculations.
itertools	For easier iterations over data structures.

Table 4.2: Internal built in python modules used

4.3 Reproducing Analysis

A mix of Jupyter Notebooks and integrated development environments[2] were used to write and execute code. The main analysis was run using a factory class pattern, called `AnalysisRun` in the `full_analysis` module, show in Appendix Figure B.2. This class was used to wrap a query to the Axelrod Dojo functionality and subsequently to the Axelrod Library in such a way that was easy to control batch executions. The analysis itself was done using native multi-threading on a Linux OS to improve individual opponent analysis run times and the overall scalability of the project.

[2]Pycharm Professional Edition and Microsoft VS Code.

The remainder of this tutorial will also assume you're working with a local development station; analysis on remote cloud instances of Jupyter Notebooks is possible but the set up is different. The steps for working on a cloud instance are described after the local set up.

Installing Basic Libraries Your first step should be to download and install the Anaconda distribution for your OS here: https://www.anaconda.com/download/. This will allow you to use the integrated c++ libraries python has to offer without needing to mess about too much. Anaconda also has them majority of Functional Libraries above and Jupyter Notebooks pre installed to make setting up much easier. From here, follow the instructions the install wizard has to add any environment variables to allow CMD/Bash access to binaries.

Installing the Axelrod and Axelrod Dojo libraries uses the pip tool that already comes with Anaconda and should be ready to execute after the last step. Running 'pip install axelrod' then 'pip install axelrod-dojo' will install these.

Once this is installed the `full\_analysis.py` file has to be downloaded from GitHub[3], it can be found in the code directory. This can just be copied and pasted if needed all were interested in is the class to generate a sequence for an opponent.

Running a Test Figure 4.6 is some sample code that will run an analysis with the following settings:

- Override the default mutation frequency of 0.1 to 0.33.
- Set the prefix for all the files to be 'example-'.
- Analysing 3 opponents. (Random will automatically have multiple instances for different seeds.)

```
from full_analysis import NewAnalysisRun
import axelrod as axl

run = NewAnalysisRun(mutation_frequency=0.33)
run.save_file_prefix = "example-"

run.add_opponent(axl.TitForTat())
run.add_opponent(axl.Random())
run.add_opponent(axl.Grudger())

run.start()
```

Figure 4.6: Code to create a sequence result to optimise best score for 3 opponents

After it has run the generated data output should be stored in the './output' directory. If the code fails to run there may be issues with this directory being created. There should be multiple output files, each with one opponents evolution stages through the generations.

Cloud Notebook Setup If you want to use a Cloud service, such as Azure Notebooks or AWS Sagemaker, the set up is similar to the above just executed differently. Installing Anaconda is not needed, the environment has the required installs. Using pip to install the Axelrod libraries and download the full analysis script can be done in an integrated web terminal or directly in a notebook. Figure 4.7 shows and example in azure, copy these in your top few cells of your jupyter notebook and it will work as required.

[3] https://github.com/GitToby/FinalYearProject

```
# -------- CELL 1 --------
# The ! means 'run this as a bash script'
! pip install axelrod
! pip install axelrod-dojo
! wget https://raw.githubusercontent.com/GitToby/FinalYearProject/master/code/full_analysis.py

# -------- CELL 2 --------
import axelrod as axl
import full_analysis as fa

run = fa.NewAnalysisRun(population_size=40)

run.add_opponent(axl.TitForTat())
run.add_opponent(axl.Random())
run.add_opponent(axl.Grudger())

run.start()
```

Figure 4.7: Cells for creating the jupyter instance of a research environment

4.4 Conclusion

In this chapter we covered the contributions needed to the Axelrod Dojo to carry on the work described in this report. Moreover, we covered the procedure of implementing and contributing to an open source package. Of the learning that was complexity during the project, learning a new version control system was the hardest along with handling the organic growth of the project. While writing code there where sections that needed changing or removing, however there was no scope or predetermined set of outcomes causing my own analysis notebooks to swell and become unwieldy at times. These problems were solved by selectively isolating work and leveraging 3rd party libraries more.

The final codebase contributions were a success and version 0.0.8 of the Axelrod Dojo has been released to the python installer package, 'pip' [10]. Following the steps in Section 4.3 will allow a working environment to be set up on any platform, letting this experiment be repeated. Providing a reproducible experiment allows the open source and research communities independently verify any findings in this work to ensure a robust foundation for future work. All finalised notebooks and final code are published online on GitHub, online resources can be found in Appendix A.

Chapter 5

Implementation Of Sequence Discovery

Genetic algorithms can have many variations depending on parameters and implementations of any attached methods. In this chapter, consideration will be given to the process of finding the optimal sequence of moves against another player. There will be various approaches used and a detailed analysis of the optimisation procedures and parameters will be described for each parameter.

In this chapter the terms strategy and opponent are used interchangeably. An opponent has a set of rules for calculating its next move, this is known as its strategy and is unique. Also in this chapter the terms solution and sequence are interchange, often called the solution sequence. A solution is a sequence of C and D moves, as described in Section 2.3.

5.1 Background

Before conducting the bulk calculations for the set of opponents listed in Appendix Section C we will test values for the algorithms' parameters to see what values of parameters are best for finding solution sequences. We will run a series of tests against pre selected strategies and review the results of their best score per turn before running the full analysis. These test opponents have been selected because they are interpretable and have either calculable solution sequences or are interesting stochastic opponents. Table 6.3 shows the opponents and their solution. Each one has a fundamentally different structure to how they work, because of this we can confirm that the genetic algorithm can select the optimal sequence solution for each structure.

In this chapter sequences will be loosely described as 'converged' if the best score of the population has reached a stable point considering the generations it has run. In order to find these settings we look at is how the best score in a population rises over the number generations after changing the parameter were observing. Once the best score per turn hits a maximum such that it doesn't appear to change no mater how many more generations are run this will be described as an optimal solution sequence. Note that for a solution to be unique we must be playing a non stochastic opponent, in

Player	Optimal Sequence	Representation of game length n
axl.TitForTat()	$CCC \dots CD$	$Cn-1,1$
axl.Alternator()	$DDD \dots DD$	Dn
axl.Grudger()	$CCC \dots CD$	$Cn-1,1$
axl.Random()	$DDD \dots DD$	Dn
axl.EvolvedFSM16()	$CDC \dots DD$	$C1,1,n-2$
axl.CollectiveStrategy()	$CDC \dots CD$	$C1,1,n-3,1$
axl.Champion()	Various[1]	NA
axl.ZDExtort()	Various[2]	NA

Table 5.1: Table of test opponents

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 5.2: Output data table

the event of playing a stochastic opponent we will seed the random element of their behaviour for reproducibility.

During the investigation we may find solutions that are not optimal, meaning that the algorithm will have found a solution that will do well against an opponent but hasn't found the best from the solution space. These sub optimal solutions are due to the occurrence of local maxima (due to an increasing fitness function) in the space of neighbouring solution sequences to members. Section 1.2.1 discusses how GAs are designed to mitigate local maxima.

Some of the questions we will hope to answer in this chapter include:

- If we have a larger initial population sample to start with, will we reach our maximum best score earlier?
- Will increasing the number of generations impact our ability to find an optimal solution? 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?
- How can we overcome the possibility of our algorithm finding a local maximum rather than the global maximum?

5.2 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 2.1. This will alter the population to keep the fittest members to continue on to subsequent generations. We will look into population size because, during any generation the size of the population influences the range of scores that we can achieve against our opponent. The more unique members we have the more unique evaluations of the fitness function exist. Because of this we can reasonably assume the larger our population gets the chance of finding the solution sequence with the optimal score will increase. Hence with a larger population we should converge to the solution sequence in less generations. For this section we analysed a range of populations to understand how solutions are affected when we run our algorithm through the set of population sizes $|P| \in [25, 50, 100, 150, 200, 250, 500]$.

The code used for this these tests will be shown in Appendix Snippet B.5 as an implementation how this analysis was conducted. It leverages the use of the function `runGeneticAlgo` show in Appendix Snippet B.1. This code will output data in the form of Table 5.2.

As a note on efficiency; 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 generations to convergence, or in an increase of 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.

After running the tests for the populations stated above we can group the data by population size and observations on how changing this parameter affects different opponents can be made. Figure 5.1 shows the best score in the population

for each opponent as the generations increase for every population size. We see that the initial population size has a significant effect on finding better sequences. This figure shows if there is a larger initial population there is typically a higher best score shown once concluding all of the generations. It doesn't, however, ensure that we find the solution sequence, as is shown in the lack of long plateaus of best score across opponents.

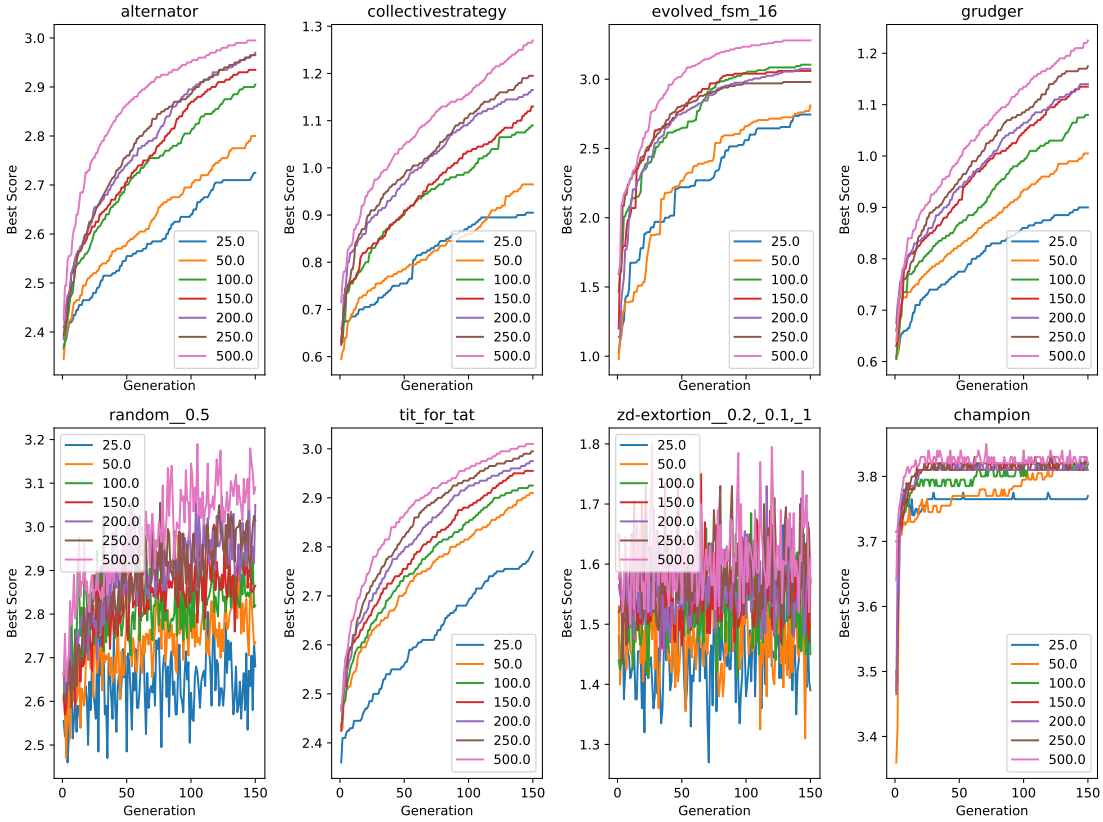

Figure 5.1: **Initial Population Size Analysis:** Best score per turn vs generation for different initial population sizes

The improvements from the effect of population increase are non-linear from observation of Figure 5.2. This figure shows the change in mean best score across population in the final generation against the population size. The change in final best score for a population of 50 compared with a population of 250 is large in comparison to the same relative increase from 250 to 500. This may suggest there are more efficient approaches to improving our final best score after a certain initial population is reached.

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 test computation times manageable. The actual parameter we will use in the full analysis will be given consideration in Section 5.8.

5.3 Generation Length Analysis

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

- The total combinations of features that the algorithm can evaluate. The more generations it runs for the more combinations of members we can evaluate a fitness for.

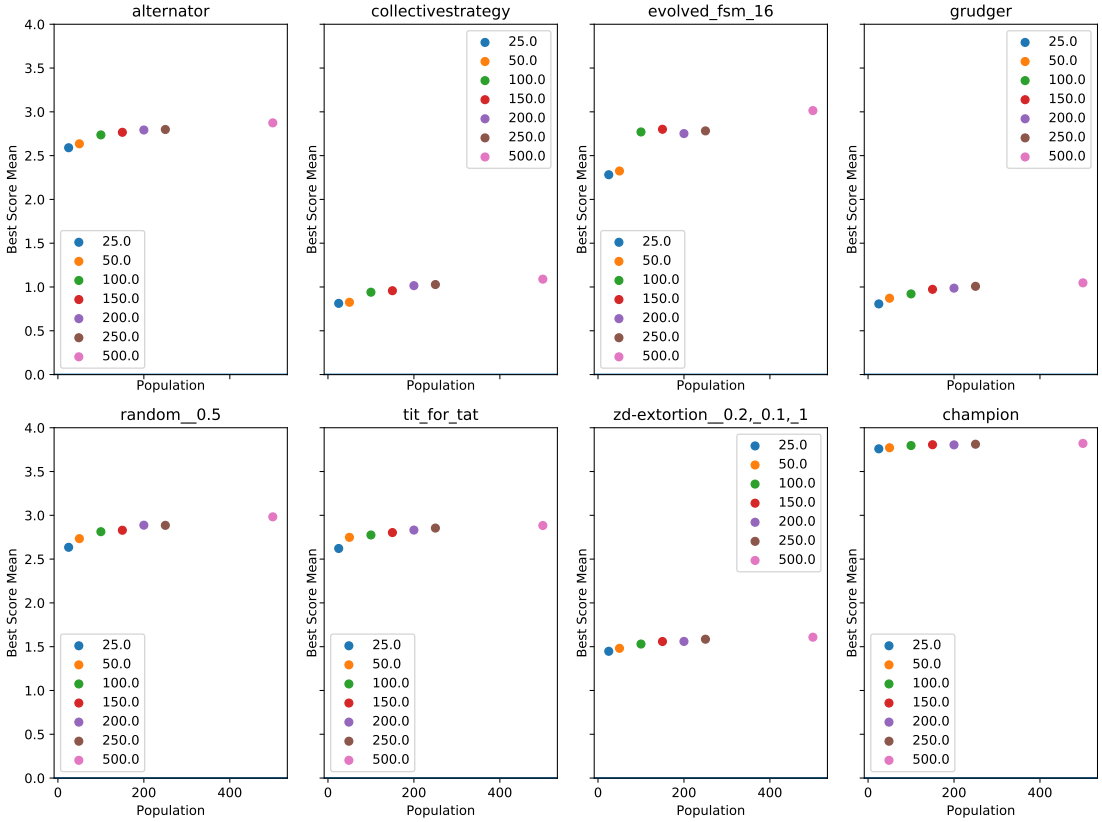

Figure 5.2: **Initial Population Size Analysis:** Scatter of mean best score vs different initial populations

- The total number of low performers it removes in the population over the course of its run. Each generation removes a proportion of its lowest fitness members. By definition of the algorithm we have a non-decreasing sequence of fitness scores; removing more lower fitness members will lead to a population of better or equal scoring members.

Generation size differs from other parameters in the fact this is purely performance based, there is no changes to the inner mechanics of the algorithm only the amount of time it will run for. A genetic algorithm with 1 generation is just a series of tests split into 2 results sets — good performers and bad performers. As we extend the number of generations we want be more focused on what happens to measured quantities normalised by generations, rather than any absolute improvement.

As a note on efficiency, the goal of finding the optimal solution sequence for each opponent would be solved by extending the generations to infinity, i.e exhaustively search the whole solution space. This, however, is not a feasible solution, and so this section looks in to the effect of increasing generations has on improving a solution sequence with respect to the generations its run. Here we will take generation lengths $G \in [50, 150, 250, 350, 450, 500]$ during these tests and a population size of 150. The code in Appendix Snippet B.6 shows how the approach the generation analysis was undertaken.

Figure 5.3 shows how the mean best score across the population changes generation to generation normalized by generation. It shows that over 150 generations the best scores against Alternator across its population increased, on average, by 0.0035 per generation. From this we can observe how the number of generations has a declining effect the overall change in our mean best score per generation. This trend is to be expected, when we are close to a maximum it is more difficult to randomly select which element in the sequence needs changing to improve a members score. On this result we can conclude as we increase generations there is less and less benefit per generation.

Figure 5.4 shows the maximum best score in a population once the analysis has concluded. 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 EvolvedFSM16 example. In Figure 5.4 EvolvedFSM16 has reached

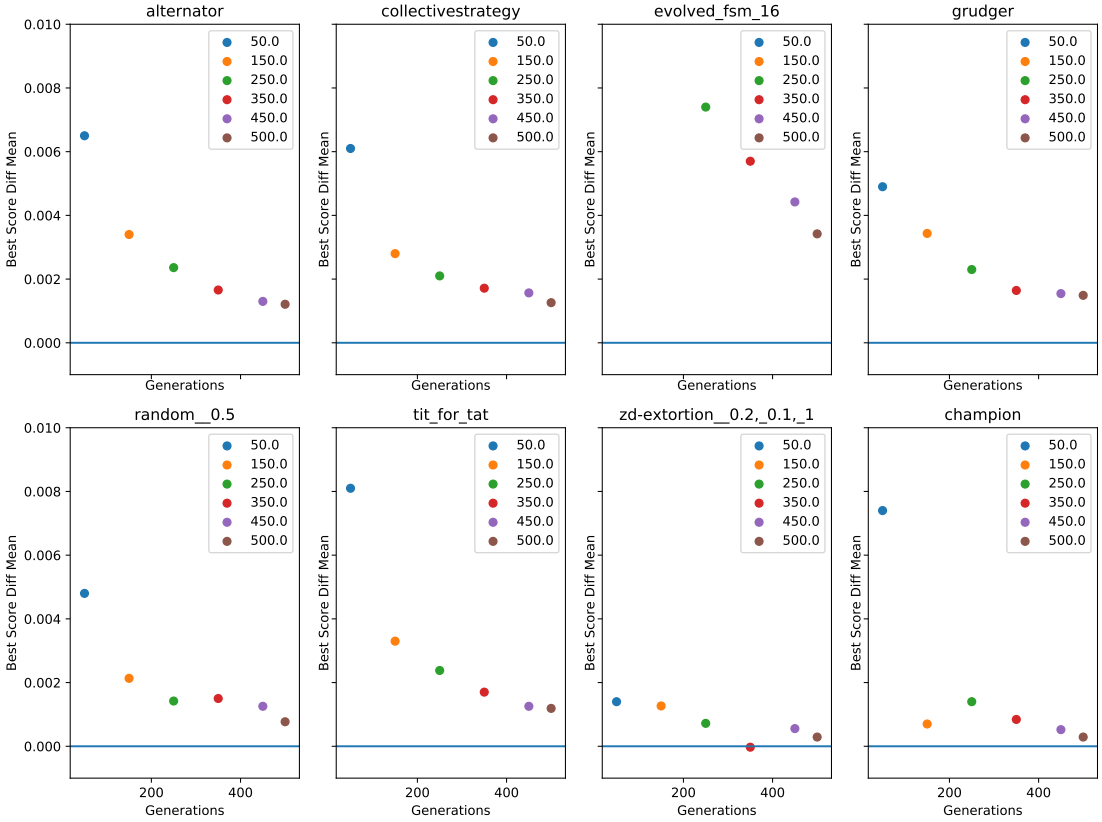

Figure 5.3: **Generation Analysis:** Mean Best Score diff vs total generation lengths

a better sequence in 150 generations than 500[3], meaning that increasing the generation length doesn't necessarily mean finding a global maximum. The complexities with local maximums during the generations lie with mutation rates and crossovers. We will cover this in Section 5.5

It is clear that a higher number of generations is preferred to find a better solution sequence. As the number of generations increase each generation provides less of an improvement to the best scores. This is due to the probability of finding a better solution sequence decreasing as we continue to improve a population. The benefits of extending the generations are incredibly useful and due to the amount of computation time we can probably incorporate a high number of generations. However we may have better performance by altering another parameter of the algorithm. The actual parameter we will use in the full analysis will be given consideration in Section 5.8.

5.4 Changing Mutation Rate

This section looks at changing the amount of mutations that occur in our population and the number elements within a sequence each mutation effects. The default settings are a mutation frequency, M_f, 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 mutation potency, M_p, of 1, meaning that every sequence that is mutated only has 1 element altered.

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

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

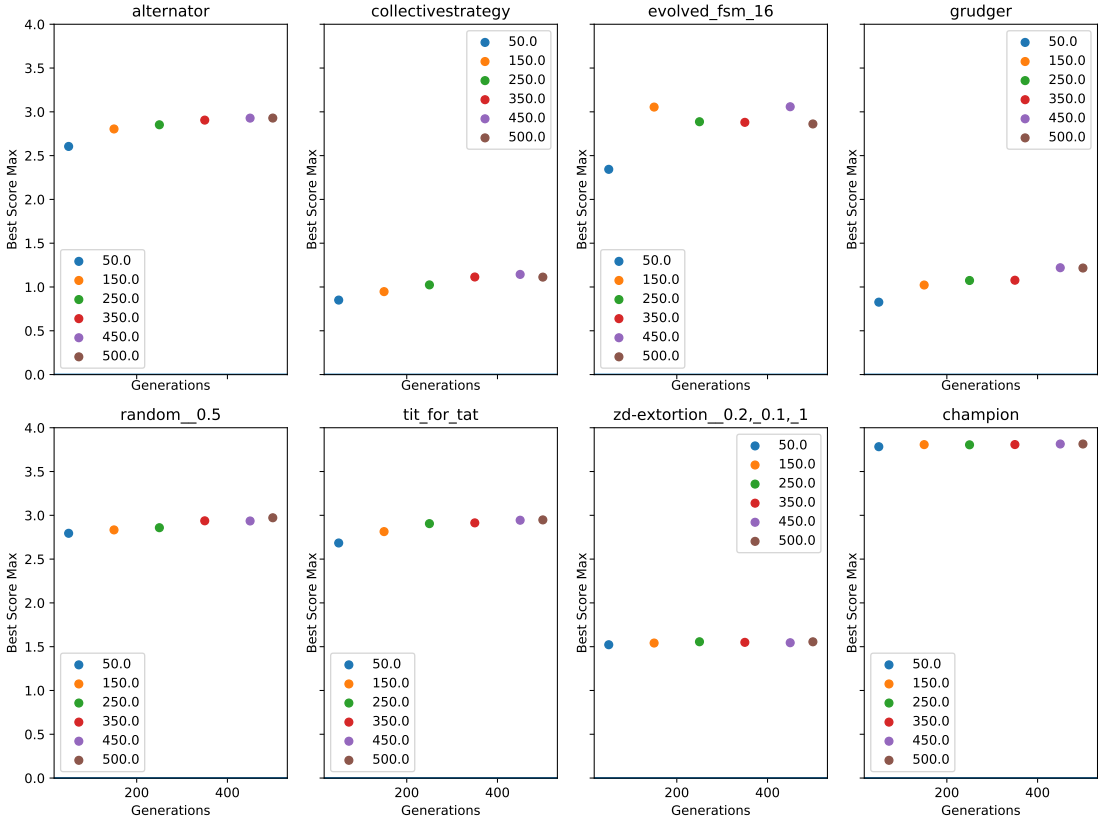

Figure 5.4: **Generation Analysis:** Max best score vs total number of generations

These are two separate questions; 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 a not on efficiency, this approach allows for an $O(1)$ factor of computation scaling. Changes in mutation are great candidates for an approach to reduce the number of generations to a solution sequence compared with other approaches.

5.4.1 Changing Mutation Potency

Changing the potency of the algorithm will provide small changes in a members features generation to generation. Increasing the potency also has the effect of increasing the Hamming distance[4] between original and the mutated sequence.

Increasing potency too much has the potential for an algorithm that is too unstable for convergence. We can imagine a sequence as a vector in 200 dimensional space then a mutation for the i^{th} sequence element 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. Mutation potency should be kept low to keep consecutively mutated sequences more similar, keeping results of the mutation within a small neighbourhood of the original. We will look into having mutation potencies $M_p \in [1, 2, 3, 5, 10, 15, 20]$.

Figure 5.5 shows the best score as the algorithm progresses through generations. There no clear benefit from increasing the mutation potency. For example, in the Collective Strategy plot, having 15 genes changed per mutation still does not improve our score as much as changing only 2 or 3. This may be down to chance of the random parameters used, however looking at more opponents than just Collective Strategy (EvolvedFSM16 for example) we find there is no clear benefit to

[4]Hamming distance: $d(s_1, s_2)$ = the number of differing positions between 2 sequences s_1 and s_2. For example: $d(111, 110) = d(CCC, CCD) = 1$. This is covered in details in section 6.2

increasing the mutation potency with respect to the overall best score against an opponent.

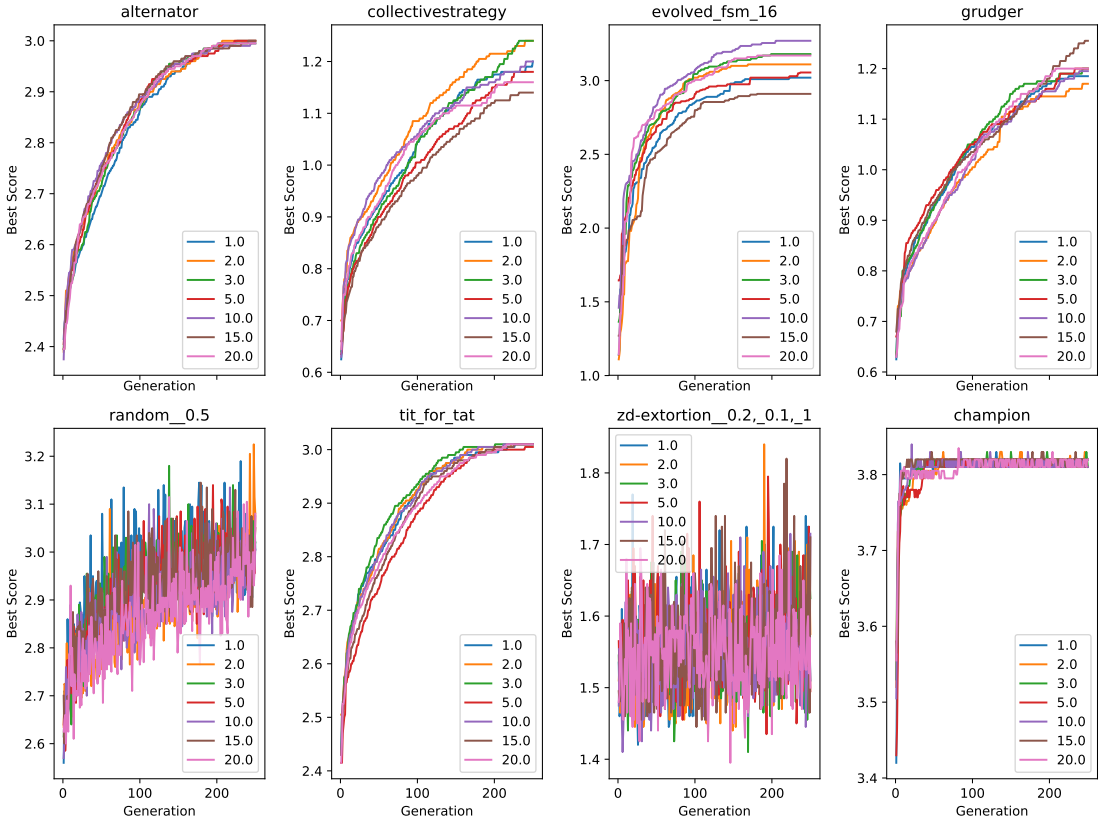

Figure 5.5: **Mutation Potency Analysis:** Best score vs generation for different mutation potencies

Figure 5.6 shows the trend of average increase of final best score across the population per generation against the mutation potency. The increase in mean best score difference is not substantial and is most likely be down to chance in the areas it does change. We see there is no sign these variables are correlated and hence can assume this parameter has negligible effect on the outcome of our sequence. The actual parameter we will use in the full analysis will be given consideration in Section 5.8.

5.4.2 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 Appendix Snippet B.9 shows the code that completed this analysis.

Figure 5.7 shows how the best score improved generation to generation for each opponent across different mutation frequencies. The results show there is little effect on the best score as we increase the mutation frequency. However there is an interesting result that can be seen on the Grudger and EvolvedFSM16 plots; the algorithm has found 3-4 clearly different solution sequences each with different scores. In the Grudger plot, we can see that the mutation frequencies of 0.15 and 0.2 produced higher scoring solutions than the other mutation frequencies. In the EvolvedFSM16 Plot we have the same result. There is no positive (or negative) correlation displayed in any of the opponents however.

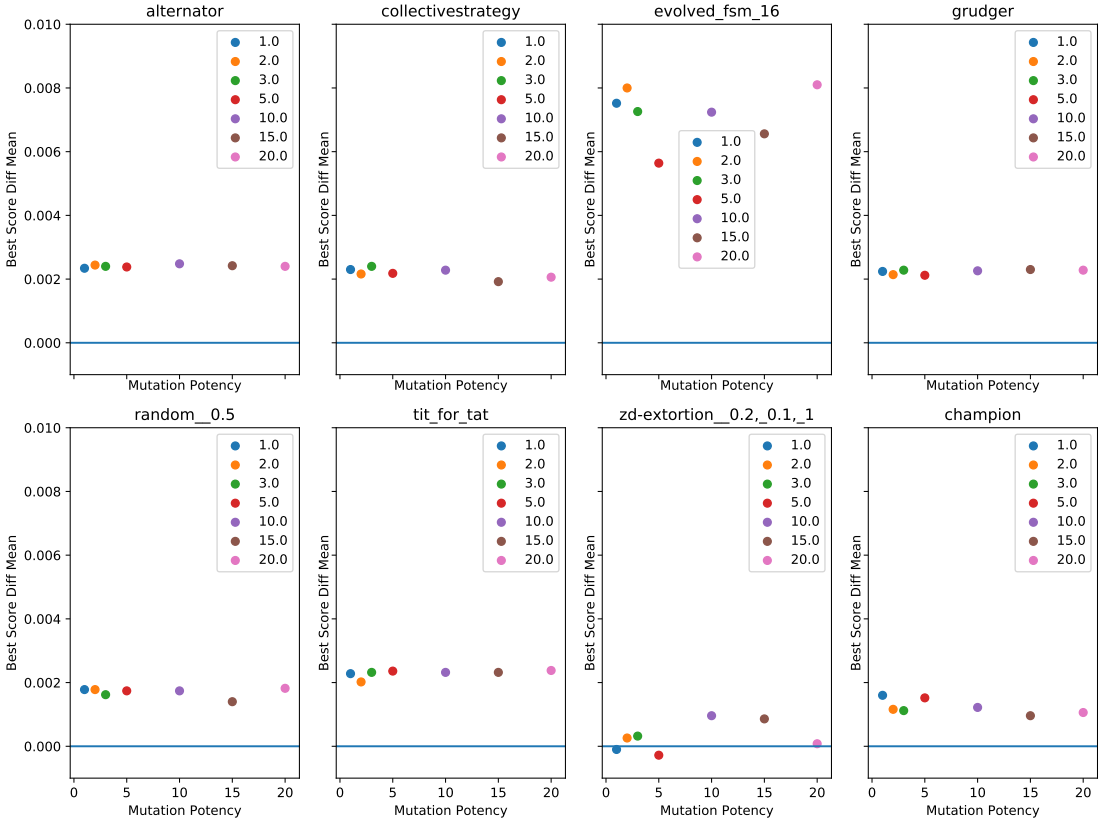

Figure 5.6: **Mutation Potency Analysis:** Average best score diff vs mutation potencies

5.4.3 Conclusions of altering Mutation parameters

In the previous sections no improvements were found from altering any of the two parameters. The default of $M_f = 0.1$ and $M_p = 1$ will be used in further analysis, and the actual parameters we will use in the full analysis will be given consideration in Section 5.8.

These sections did show, in a clear manner, that there were local maxima found. Section 5.4.2 showed clearly that sub optimal solutions for EvolvedFSM16 and Grudger were common. Section 5.5 will discuss approaches we can take to reduce the chances of this happening.

5.5 Mitigating Local Maximum Solutions

Local maxima have clearly occurred when the algorithm calculates sequences for opponents Grudger and EvolvedFSM16. There may be other occurrences we have not observed. Each stochastic opponent is also incredibly difficult to understand whether a solution is actually found, this is discussed further ing Section 5.7

Figure 5.7 shows that there are clearly multiple distinct score plateaus reached for the Grudger opponent. The difference between Grudger and the other opponents considered is that the Grudger has a singularity where its behaviour changes. The change in behaviour is not uncommon, Tit For Tat, Collective Strategy and others work in a similar way and in all cases the 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. 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 cooperate to the

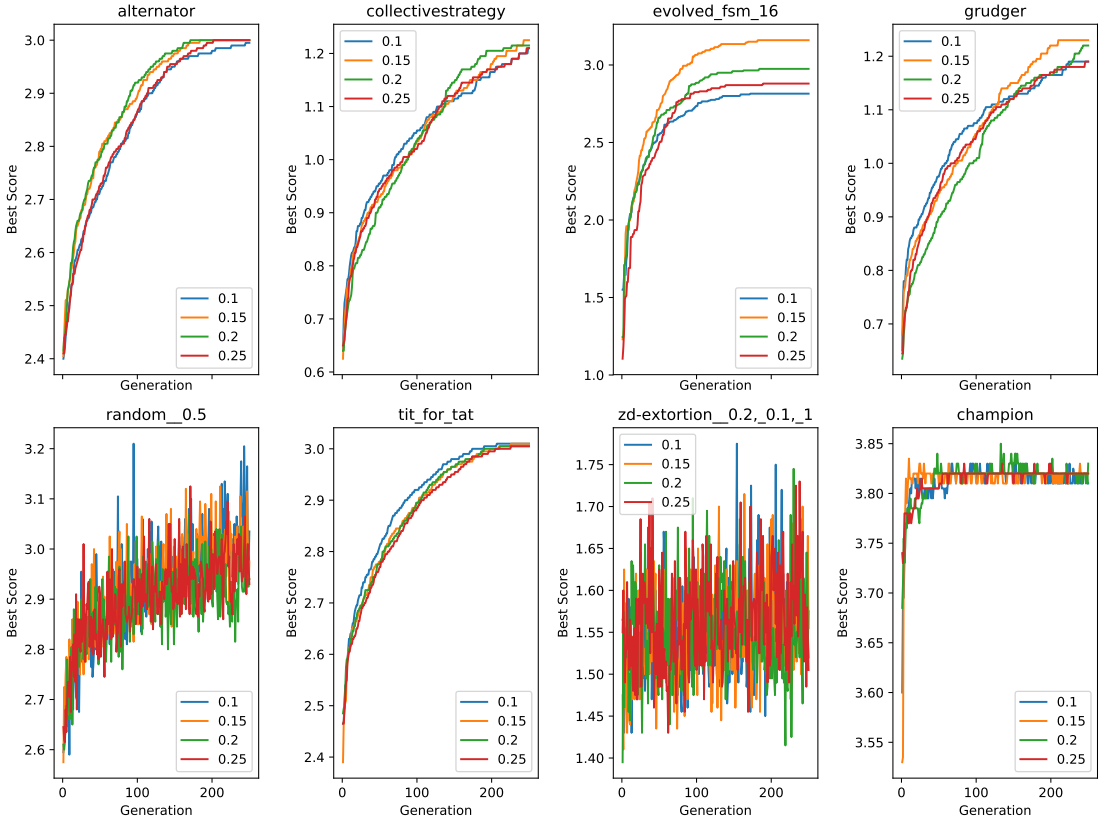

Figure 5.7: **Mutation Frequency Analysis:** Best score vs generation for different mutation frequencies

point of a defection and end in lots of defections after it has already defected counter Grudgers' harsh behaviour (see Section 3.3 for the Grudger strategy).

Grudger best start: $C6, 8, 2, 1, 2, 1, 6, 1, 2, 2, 4, 2, 1, 1, \ldots$ (No pattern is obvious.)

Grudger best end: $C22, 178$

We may be able to identify a way to mitigate this singularity effect by looking at the difference between 2 opponents whose strategy differs by their forgiveness parameter. 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 forgive and becomes a 'dumb' only defect opponent. This means that the move the 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 likelihood 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.

To investigate this we will play two totality games against grudger, one of all Cs and one of all Ds, shown in Snippet 5.8. These games show we should be converging on a totality of Cs rather than what its doing, finding the totality of Ds after a number of cooperations. This is probably because the algorithm initially limits its best score per turn once the first generation is complete and a cut-off defection 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 solution sequence without knowing exactly where to target.

```
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 5.8: Grudger matches against totalities

The example totalities, shown in Snippet 5.8, are edge cases and would be 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 the first defection move and change it to a cooperation move all by chance. The likelihood of this occurring is incredibly small, and starting at common or uniform sequences would be more beneficial to searching to solution space, as posed in Section 5.6.

5.5.1 Ineffective Approaches 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 the algorithm is to find the optimal solution sequence, it must take a crossover and mutation path which doesn't cut off better paths as the algorithm work our way towards good solutions. This is much easier said than put into practice due to the way the algorithm 'cuts off paths'. The current technique is taking halves from 2 members and merging them, potentially removing successful relationships between elements in both the first and second halves. If we reverse this thinking and try to alter our crossover design and mutation rate such that, instead of 'cutting off' a path by taking large sections of each member, it may be possible to 'build' new ones using more, smaller, sections of the 2 parents.

We can re-design the crossover to switch up smaller subsections of the solution sequence then allow the mutations to optimise these sub-sequences. The current design is shown in Figure 5.9. We want to allow the crossover to have more of an impact on optimizing each section. i.e.go from:

$$|oooooooooooooooooo| \text{ and } |xxxxxxxxxxxxxxxxxxx| = |ooooooooooxxxxxxxxxx|$$

to the mixture:

$$|oooooooooooooooooo| \text{ and } |xxxxxxxxxxxxxxxxxxx| = |ooxxooxxooxxooxxooxx|$$

This will allow the mutation to alter 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 5.10. As shown, the algorithm splits the two sequences into 10 section and the new sequence is formed from alternating sections. Figure 5.11 shows an example of the new crossover method.

```
def crossover_old(self, other_cycler,in_seed=0):
    seq1 = self.sequence
    seq2 = other_cycler.sequence

    if not in_seed == 0:
        # only seed for when we explicitly give it a seed
        random.seed(in_seed)

    midpoint = int(random.randint(0, len(seq1)) / 2)
    new_seq = seq1[:midpoint] + seq2[midpoint:]
    return CyclerParams(sequence=new_seq)
```

Figure 5.9: 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 5.10: New Crossover algorithm

After testing at this new crossover algorithm with the default mutation (freq=.1 and pot=1), there was no noticeable improvement of mitigating local maxima from introducing this change of function. Each players average score per turn was within the original score per turn if the algorithms crossover method was unchanged. The problem of mitigating sub optimal solutions may be more efficiently solved using a predefined population. Section 5.6 discusses this in more detail.

5.6 Altering Initial Population

When performing analysis of an opponent using a GA it can sometimes be useful to allocate the starting positions of our feature selection to create uncommon[5] starting points. Altering the initial population allows selection of starting sequences that fit patterns known to have good results against certain strategies. For example totalities, with some head/tail moves, are usually very effective against some opponents; Tit For Tat, Random or Grudger for example. Alternating sequences are also good starting sequences for an initial population, allowing a more intelligently distributed set of starting points for the random mutation process.

Adding a pre defined starting population can be visualised as placing balls on a lumpy 3d plane to try and find the deepest valley. Starting with an educated guess and distributing the starting positions evenly means we are less likely get all the

[5]For example of a randomly constructed sequence that is a totality of Cs is incredibly rare, with around a 6.2^{-61} chance of occurring naturally with a random selection.

```
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 5.11: Example of new crossover algorithm

balls stuck in one, sub optimal, valley. This section looks into benefits of using a pre defined population in the GA.

5.6.1 Constructing a population

We will start by creating a population of successful known starting members. When starting with these members we allow the entropy of the genetic algorithm to alter these sequences towards optimal solutions (assuming these are not already optimal). Deciding where to start our algorithm may mitigate potential sub-optimal solutions by reducing the distance between the starting sequences and optimal solutions. This list of starting members are stated below:

Totalities

- $C : [200]$ — 1×2 sequence

Single Change Sequences

- $\{C : [i, 200 - i]\} \quad i \in [1, 10]$ — 10×2 sequences
- $\{C : [200 - i, i]\} \quad i \in [1, 10]$ — 10×2 sequences

Matching Tail Sequences

- $\{C : [i, 200 - (i + j), j]\} \quad i, j \in [1, 5]\}$ — 25×2 sequences

Alternating

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

3 Block Handshakes*[6]

[6]This was added after the first test, with different population sizes, with the new population

- $C : [\ i, j, k, 200 - (i + j + k)] \quad i, j, k \in \{0, 1, 2, 3\}$ — 32×2 sequences

For each of these starting members, a sequence with a defection move to start but with the same pattern will also be added to the set of starting sequences. This in total gives 164 (after code 5.13) constructions of sequences, we will then make up the difference to the population limit using a set of random sequences.

Now that the Initial population has been altered we can conduct the analysis in Sections 5.2, 5.3, 5.4.1 and **??** again.

5.6.2 Population Size

Figure 5.12 shows the best score generation to generation for each population size. We will have $|P| \in [102, 200, 250, 500]$ due to starting with the 102 initial members (*this was the initial size of pre set population, this increased to 164 after this test).

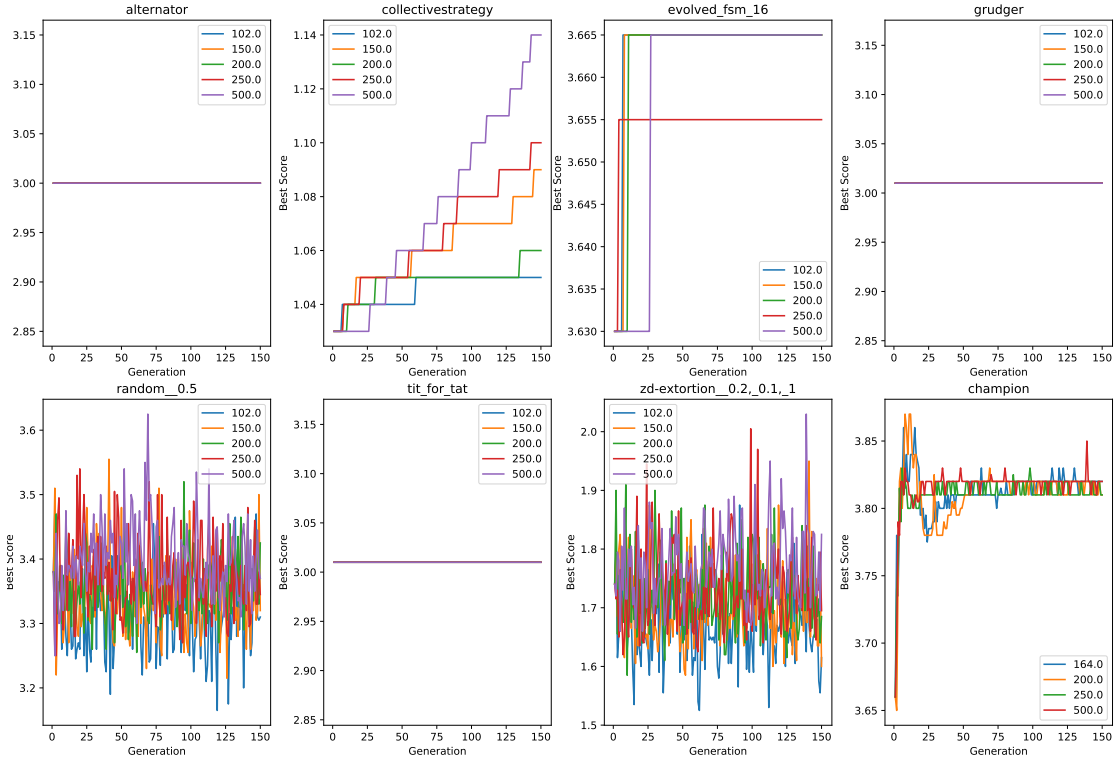

Figure 5.12: Best Score vs generations for pre-set initial populations on top of random sequences

Figure 5.12 appears to have a better results for Collective Strategy with an initial set population for the algorithm. Of the results that are not showing optimal solutions we can observe that the algorithm has score plateaus for all but Random, Collective Strategy, ZD Extort and Champion. This is probably due to 2 different reasons:

Random & Stochastic Opponents: This 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 Ds in the Random sequence will allow the solutions defections to score less. This leads to a solution sequence containing some random Cs not because they score better in some turns, but because these tests not a fair trial. The two sequences play against different Random opponents generation to generation without then being seeded.

Collective Strategy: This strategy is a combination of a handshake and the Grudger strategy. If we look into the solution it would be seen to have found the handshake but then arrives on the Grudger effect later in the game. After this encounter the same problem as we had before, with grudger, the algorithms limits the damage by splitting solutions 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. (*This means we increase our initial population from 102 members to 164)

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 and other stochastic opponents are a special cases that require more analysis to find the absolute optimal. The new initial population, now of size 164, will include all combinations of C & D of length 4, followed by finishing on all Cs or Ds as shown in the Totalities & handshakes section of Snippet 5.13. Further tests will also supplement this population with random members for $|P| \in [164, 200, 250, 500]$

Figure 5.14 shows the best score generation to generation for each population size, this time for the previous strategies where an optimal solution was not found. After adding the extra members of the initial population we can see that now the handshake strategy (such as Collective Strategy) are solved much sooner. Also shown is the Random and ZD extort players, these are examples of Stochastic players which are still not finding optimal solutions due to the observation of seeding stochastic opponents. These classes of opponent are examined further in Section 5.7.

It is clear the change of initial population is beneficial. The actual parameter we will use in the full analysis will be given consideration in Section 5.8, but the initial population will stay as an effective starting point for the GA.

5.6.3 Generation Length, Mutation Potency & Mutation Frequency Analysis After Change of Initial Population

Figure 5.15 shows how the best score is affected while changing the number of generations the algorithm will run to $G \in [50, 150, 250, 350, 450, 500]$. This figure is also displaying with a population size of $|P| = 200$, 164 of which are pre defined. It shows no real improvement from increasing the generations, the actual parameter we will use in the full analysis will be given consideration based off conclusions made in Section 5.3.

Figure 5.16 shows the best score generation to generation while changing the mutation potency. $M_p \in [1, 2, 3, 5, 10, 15, 20]$ was used whilst also running with a population size of $|P| = 200$. The figure shows no real improvement from increasing this variable of the algorithm. The actual parameter we will use in the full analysis will be given consideration based off conclusions made in Section 5.4.1.

Figure 5.17 shows the best score generation to generation while changing the mutation frequency. $M_f \in [0.1, 0.15, 0.2, 0.25]$ was used whilst also running with a population size of $|P| = 200$. The figure shows no real improvement from increasing this variable of the algorithm. The actual parameter we will use in the full analysis will be given consideration based off conclusions made in Section 5.4.2.

Discussion

The approach of adding a predefined set of members to the population before running the algorithm was incredibly successful. Most opponents have solution sequence equal to or very close to members in the starting population. This result will mean that when calculating the solution sequence for the main population the predefined set, generated by Snippet 5.13, will be used to shorten analysis times.

Random, ZDExtort and Champion are still being shown that the algorithm is not finding the optimal solution for these opponents. This is due to the fact these player belong to the class of stochastic opponents. Section 5.7 will look into these in more detail, putting forward a possible approach to simplify finding of a solution sequence for these opponents.

5.7 Stochastic Opponents

Stochastic opponents create a problem with respect to continuity of testing. The programming of the algorithm we are using generates a new opponent for every member 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 of stochastic opponents which can be exploited.

The concept of a stochastic opponent can be somewhat 'overridden' by seeding the pseudo random number generator that creates the parameters that define what moves the strategy will take. Currently the only way of doing this is by seeing the whole Axelrod library upon initialising the opponent instance. Snippet 5.18 shows the code for wrapping any given opponent with the global seed command.

When calculating the solution sequence 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 with the seed that is set for algorithm as a whole. The means we will be playing the same 'version' of a stochastic opponent over and over, removing the layer of abstraction in its strategy.

5.8 Conclusion of approach

Each section of testing allowed the algorithm to become more specialised in solving the given task. The list at the end of this section shows which parameters we have chosen for the final analysis.

After constructing the final analysis factory and running some environment checks we made the following changes to the code for ease of use:

- Number of generations changed: $300 \rightarrow 600$.
- Crossover algorithm, shown in Figure 5.10, was reverted to code given in Figure 5.9

These changes resulted in a streamlined piece of code we were able to run on the Cardiff University Mathematics department big compute machine. Reflecting on the execution and deployment of the code shows that we could have improved the analysis of short run time strategies against longer run time strategies and the reproducibility of the batch processing within the Axelrod Dojo Module. As far as each individuals analysis goes, I feel like the approach of adding predetermined sequences to an initial population was a huge improvement in compute time and overall predictions of best sequences.

The algorithm parameters described in section 1.2.1 have the following parameters before running the final analysis:

- Initial Population, P, of 164 pre generated members + 86 random members for a total of 250.
- Generation length, G, of 600.
- Mutation Potency, M_p, of 1.
- Mutation Frequency, M_f, of 0.1.
- Bottleneck, $b = P/4$.
- Crossover method as that shown in Snippet 5.9.
- Mutation method as that shown in Snippet B.7
- Each opponent is wrapped in the seed wrapper shown in Snippet 5.18, stochastic opponents will be run 10 times each with different seeds.

```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 5.13: Initial Population Code

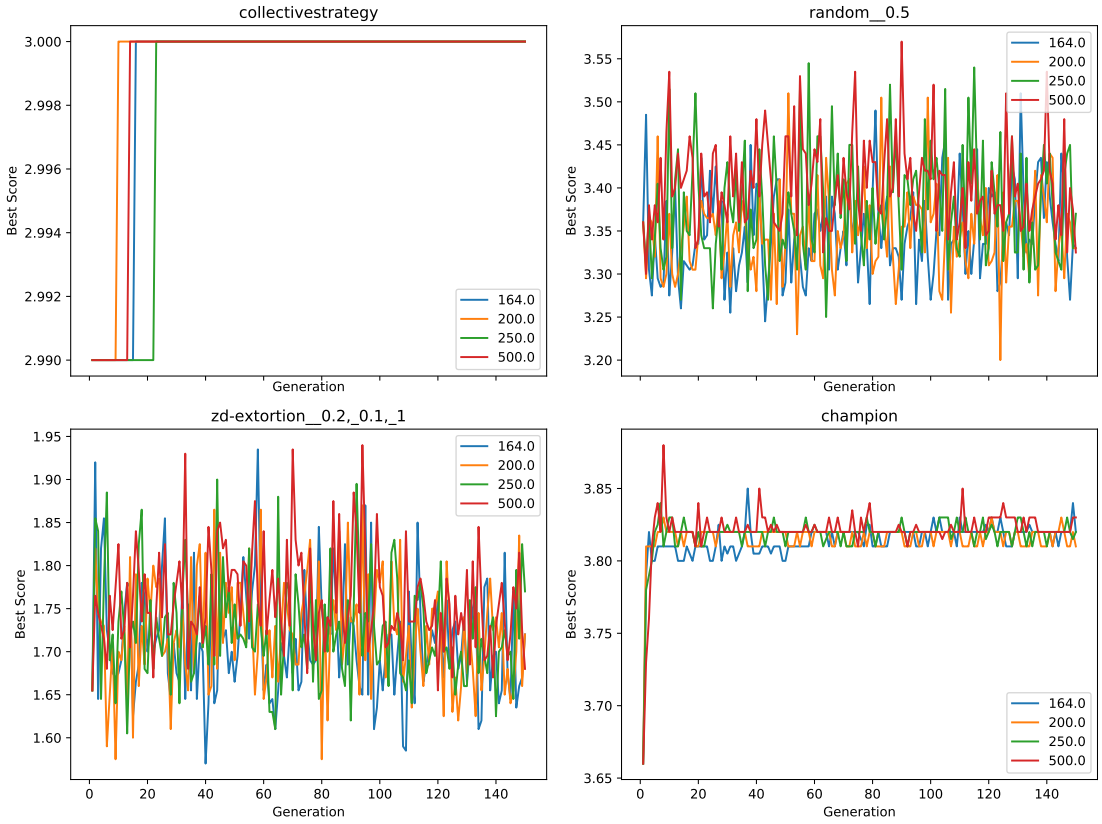

Figure 5.14: **New Population:** Non optimal sequence players after changing initial population

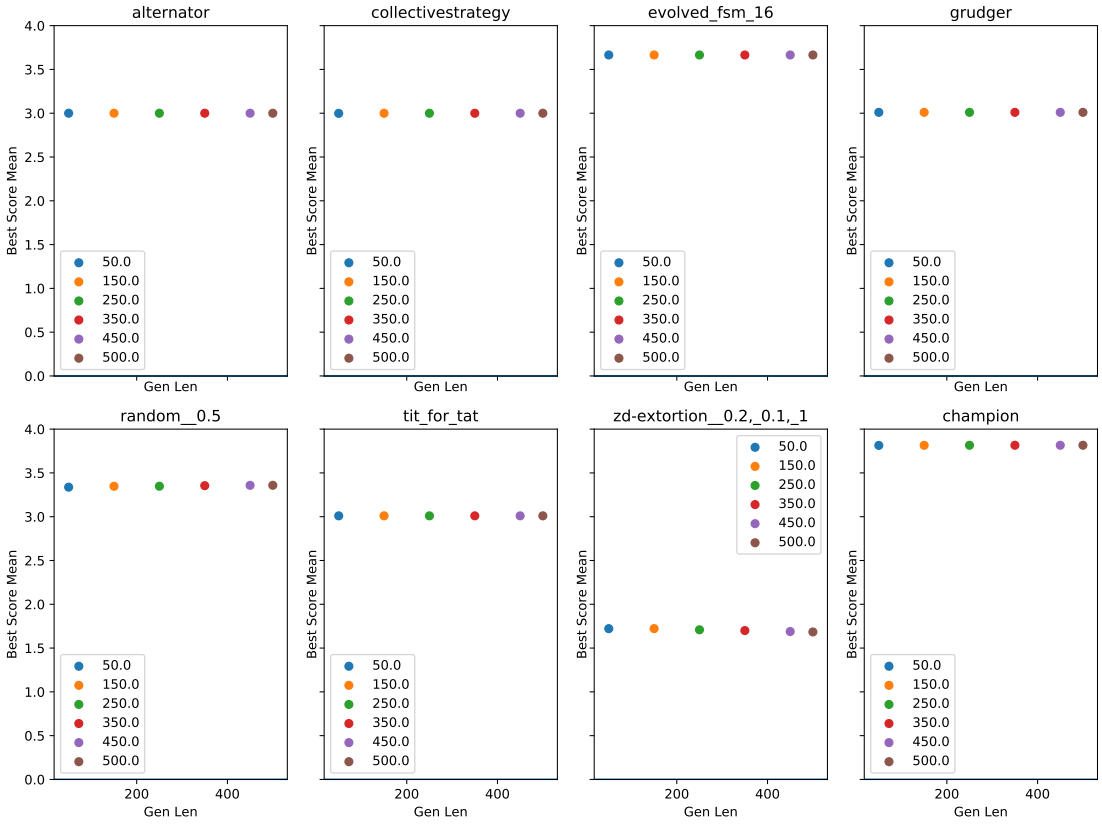

Figure 5.15: **New Population:** Generation Length analysis for best score mean against generation length.

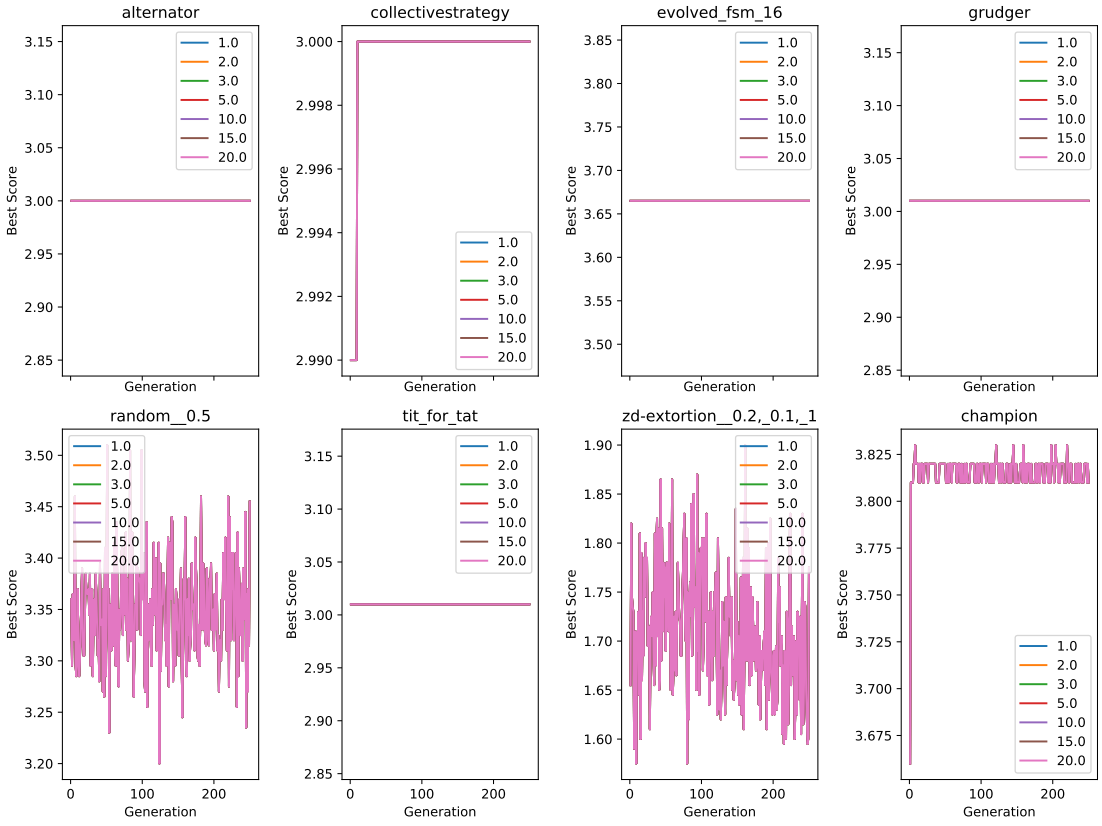

Figure 5.16: **New Population:** Best score against generation for different mutation potency levels

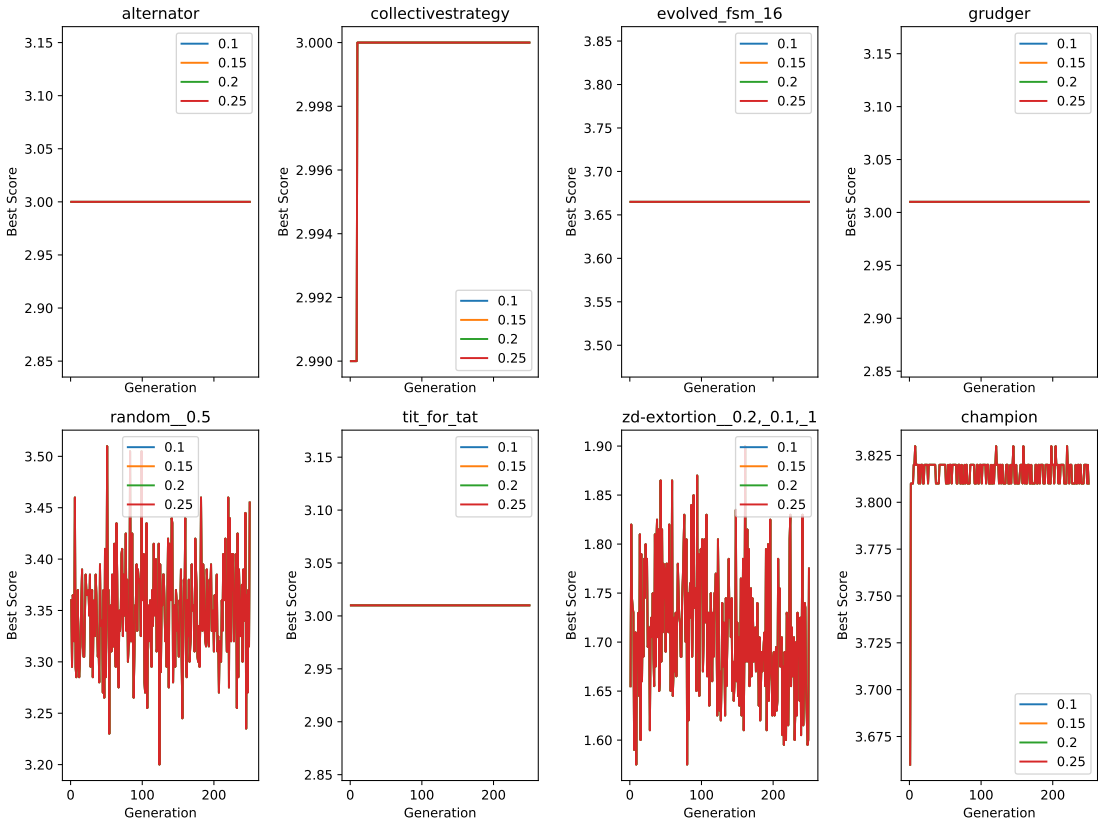

Figure 5.17: **New Population:** Best score against generation for different mutation frequencies

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

    return NewClass
```

Figure 5.18: A function for wrapping a player with a global seed function call

Chapter 6

Results and Discussion

This chapter will look into the results produced after running the analysis across all opponents. Details of the distribution of the results and basic output from the computation are given in section 6.1.

The analysis performed in this chapter contains data for opponents listed in Appendix C. Opponents without results due to being incomplete are all the Long Run Time (LRT) strategies. As these become available they will be added to the analysis.

Each opponent was submitted to the compute engine using the standard analysis factory class `AnalysisRun`, shown in Figure B.2, using native OS multi-threading to improve individual opponent analysis run times and the overall scalability of the project. We ran the code over a period of days resulting in 200+ output files of data which could then be analysed.

6.1 Resulting Data

The data we generated from the final analysis contained 760 of the 959 opponents we set out to analyse. Figure 6.1 shows a raw data head from the output of the ϕ opponent.

gen	score mean	score median	score var	score range	best score	best sequence	name	seed
1	2.90966	3.0575	1.665496	4.985	5.0	DDD...	ϕ	0
2	4.79004	4.9250	0.420476	2.275	5.0	DDD...	ϕ	0
3	4.79352	4.97	0.489173	2.185	5.0	DDD...	ϕ	0
4	4.81386	5	0.460776	2.2	5.0	DDD...	ϕ	0
5	4.81826	5	0.479307	2.18	5.0	DDD...	ϕ	0
6	4.83508	5	0.430063	2.03	5.0	DDD...	ϕ	0
...	...	...	...	...	...	...	...	...

Table 6.1: Raw data from `AnalysisRun.py` output file

The data collected was, for each of the strategies, merged and collated with metadata to form an analysis table which was then processed. This metadata included player classifiers as given in the Axelrod library, generated values from fields in Table 6.1 and information generalizing stochastic players to their original base player before seeding. Table 6.2 shows the first few rows of this information.

When analysing the overall best sequences for each opponent we can pick out some interesting and descriptive data. The distribution on best scores are showing in Figure 6.1, its kernel density estimate (KDE) [44] exaggerates that fact we have a skew towards the higher scores with a fat tail on 4.5 to 5. This shows we can average a score higher than 3.0 against the majority of opponents per turn. From this we can infer that there is a way of outplaying many of the opponents in the Axelrod Library (i.e. we can perform better than a game where both players would have an average score of 3.0 from

base	stochastic	memory depth	makes use of	score bin	start move	blocks	mean block length
ϕ	False	inf	`set()`	very high	D	1	200
π	False	inf	`set()`	very high	D	1	200
e	False	inf	`set()`	very high	D	1	200
ALLCorALLD	True	1	`set()`	very low/high[1]	D	1	200
Adaptive	False	inf	`'game'`	very high	C	2	100
...	...	...	...	...	...	...	...

Table 6.2: Metadata which was merged to Table 6.1 during analysis; joined base on name.

mutual cooperations). If there is a way of identifying an opponent then we would be able to tip the scales in a tournament by scoring 3 or more every turn by playing the sequences shown.

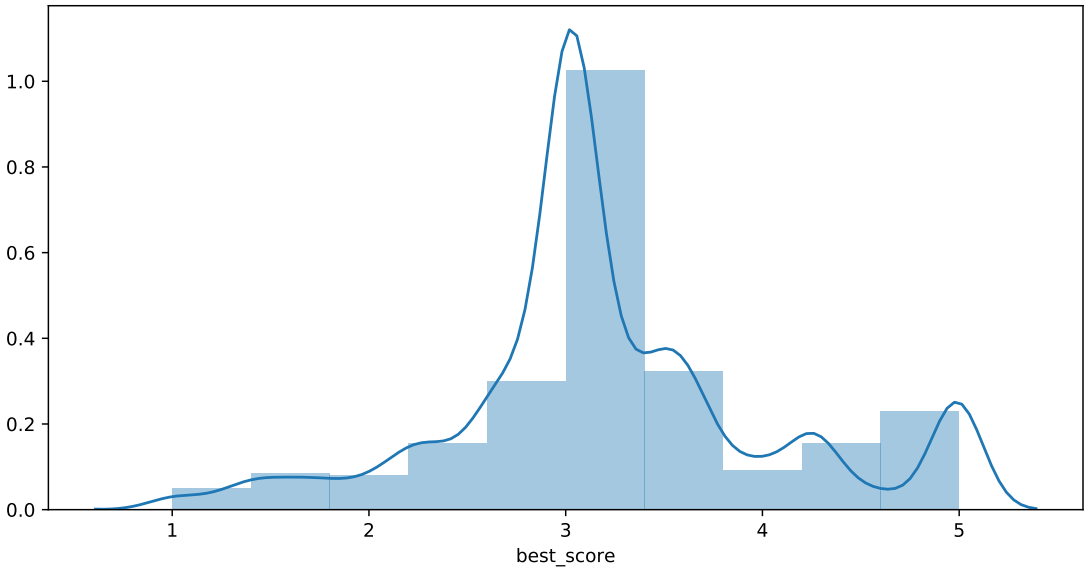

Figure 6.1: A histogram showing the distribution of best scores with overlaid KDE

Figure 6.2 shows patterns related to how the number of blocks within a sequence changes depending on score, opponent type and start move. In the left plot, solution sequences starting with a defection lead to 3 groupings; low block number, between 50 & 125 blocks and near 200. This pattern has little information that describes the relationship between the solutions, but there could be an underlying reason in the make-up of an opponent which describes the groupings. All 3 contain both stochastic and non stochastic and have a wide range of scores. Starting with a cooperation shows a trend for non stochastic opponents; ignoring the near totalities on the far right shows a linear trend from 0 to 200 blocks as the score increases. This trend describes that, for non stochastic opponents, there are complicated solutions that score us more than simple solutions for particular opponents. It shows evidence that if a non stochastic opponent is more complicated in its solution, we can do better than if it is simple; for example, the solution for Tit For Tat is $C199, 1$ which scores 3.01 whereas Adaptive Pavlov 2011 has the alternator solution $C1, 1, 1, 1, \ldots$ and scores 3.97. This does not hold for stochastic opponents as, for example, ZD Extort with seed 0 has the solution sequence $D4, 5, 1, 9, 1, 41, 1, 42, 1, 90$, and only scored a 1.355.

By looking into the distributions of best scores and the make up of solutions we are showing 2 things. That most opponents can be outplayed; we are able to score higher than an average of 3 per turn over the course of the game and hence, by the definition of the prisoners dilemma, defiantly come out with a higher score than our opponent. And that solution sequences are varied and there is little correlation between the complexity of a solution and the best score we can achieve.

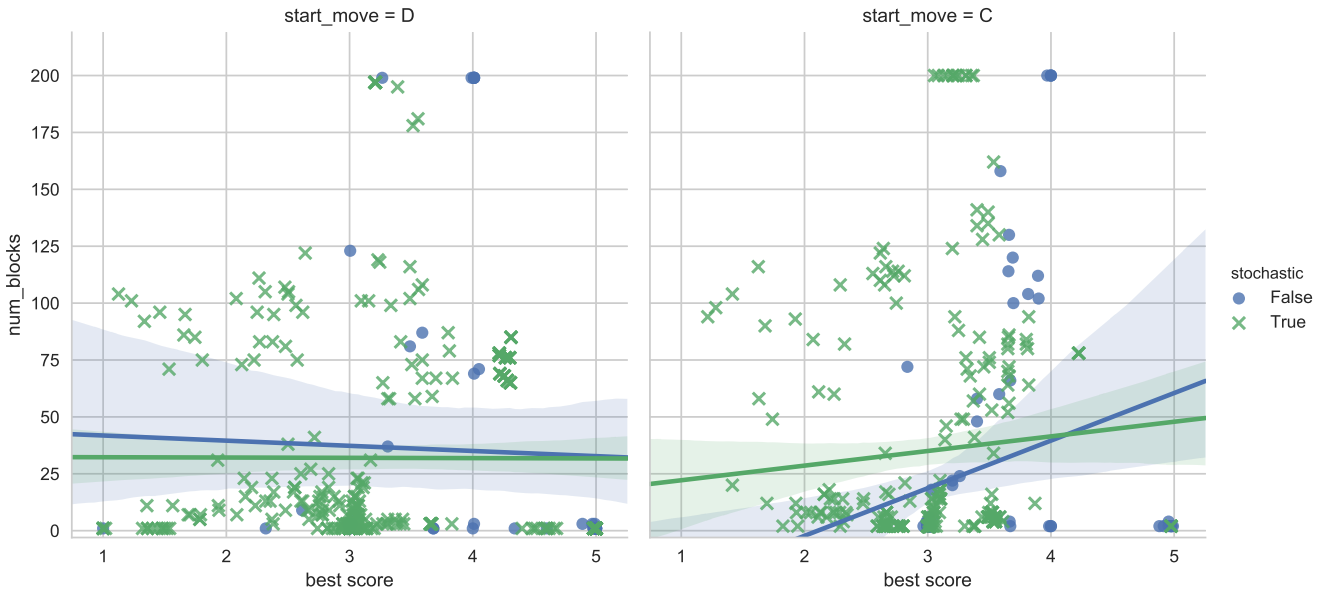

Figure 6.2: A joint plot of best score vs number of blocks coloured by stochastic boolean

6.2 Solution Distance Matrices

The first avenue of analysis, after constructing descriptive data, was to look at the relationship best response sequences have with each other. A distance matrix shows how much sequence for one opponent differs from every other, if 2 sequences are similar with respect to the distance function then they will score lower than 2 sequences that are more distinct.

In each matrix we order the opponents, $S_(O_i)$, by the score of their sequences[2] so that the order is now $f(S_{O_0}) \geq f(S_{O_1}) \geq \ldots \geq f(S_{O_n})$. but we will shorten this notation S_{O_i} to S_i for simplicity. The best response sequence for the ith opponent, S_i vs the best response sequence for the jth, S_j is scored using our distance function, for example $d(S_0, S_n)$ is the distance between the best and worst scoring sequences respectively. The Matrix itself will be symmetric down the diagonal, so looking across rows vs columns tends to make more logical sense; the top rows are the highest scoring best response sequences, and the lower rows the worse scoring sequences.

There were two distance measures considered: Hamming Distance and Cosine Distance.

Hamming Distance [45]

$$d(S_i, S_j) = S_i \cdot S_j^T$$

This corresponds to

$$1 - \frac{\sum_{i,j=0}^{n} \delta_{ij}}{n} \text{ where } \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

The Hamming Distance represents the count of the elements that differ in any two sequences. A Hamming Distance will thus correspond to the number of places we have to play different moves against opponents to get out best score. Figure 6.3 shows the matrix generated by the code in Figure B.3.

By observing the graph row by row, we can build an idea of how similar each sequence is to the range of others. The top section of the plot shows the best scoring sequences vs other high scoring sequences on the left, and vs the worse scoring sequences on the right. At the very top left there crossing dark and light columns, suggesting that there are lots of very similar or dissimilar solutions at the high vs high end of the score level. As we move to the high vs low scores there are larger blocks of dark red representing high dissimilarity. This is shown again as we move to the bottom of the diagram.

[2] See section 2.1 for explanations of notation.

The large blocks or red covering the bottom 100 or so rows show that the lowest scoring opponents have very different sequences to the higher scorers, but as a group of low scoring vs low scoring they are quite similar.

We can look into what distances we expect by trying to analyse how scored are formed. The recording of an average score means that we can estimate the ratio of move combinations that formed a score because we know, if we are the first player, the pay offs for the combinations: $(C, C) = 3$, $(C, D) = 0$, $(D, C) = 5$, $(D, D) = 1$. Thus we can assume scoring in the midrange (3 ish) means mostly cooperating, or a combination of (C, D) and (D, C), whereas at the high and low ends its more likely we are to be defecting. This would mean when looking at the high vs low section and low vs high section of the graph we would expect a high level of similarity; $d(S_i, S_j) \approx 0$. However this hypothesis is not supported; its clear there is a mix of distances because all 4 corners of the graph are not showing $d(S_i, S_j) \approx 0$.

To understand why this disparity between expectation and result exists we can look at a visual representation of each of the strategies. Figure 6.6 shows what the solution sequence looks like move by move, sorted by best scoring at the top of the left image moving down to worst scoring on the bottom of the right. From this diagram it is clear that some of the predictions above were true; the very high and low scoring solutions are totalities of defections. There is however much more noise for sequences at the top of the scoreboard, resulting in the large distances between the scores.

Figure 6.6 also shows some interesting results about picking up points in patterns. Some opponents (approximately half way down on right image) work using ratios, we can trick them for half the game and take advantage for the remainder. Others are much shorter term solution, alternating between C and D every other turn (1/4 down the left image). When scoring in the mid range (bottom of right, top of left) there seems to be patterns too; totalities of C are to be expected but there are some results that seem to be tricking an opponent for the first number of moved before defecting for the rest of the game. At the higher score ranges there seems to be lots of noise, many of these opponents are rather complicated in their Strategy, the top 12 non totalities are listed below:

One final point regarding what Figure 6.6 shows is the effectiveness of the genetic algorithm. From the theory of repeated games, an equilibria for the repeated game must end in an equilibria for the static game. Hence for one of our solution sequences to be optimal the result must end in a D. This is true for all but 57 of the solutions, meaning there is room for possible improvement in more than one result. These sub optimal results were all stochastic.

Cosine Distance [46] The cosine of two vectors constructed by using the dot product formula as shown. In our interpretation each dimension represents a sequence element so we are working in $\mathbb{R}^{200}$ with every value taking a $C : 1$ a $D : 0$. Figure 6.4 shows the distance matrix generated from the data files using code in Figure B.3

$$d(S_i, S_j) = \cos(\theta) = \frac{S_i \cdot S_j}{||S_i||\ ||S_j||}$$

The Matrix for Cosine and Hamming are almost exactly similar, the only difference being the value of the distance between sequences. This is due to the two measures being similar in their relationship of space [47]. We can conclude that there is no extra information shown in this diagram.

6.3 Solution Groups

If we want to group opponents together, the most obvious way is to look at which opponents have have the same best score sequence. Appendix D has full details, and figure 6.5 shows a plot of the trends. This figure shows an almost logarithmic/polynomial trend in score as the number of blocks in a sequence increases. Below this trend there are also a group of stochastic opponents who dont seem to follow the pattern and form a group below this. This could indicate that the trend only applies to specific strategies and that this grouping of stochastic opponents use more unforgiving approaches, such as the ZD Extort strategies.

Figure 6.6 provides a visual representation of sequences so that identifying groups of equivalent or almost equivalent solution sequences is easy. Families of similar solutions can be picked out, for example the Qlearners at the bottom right of the figure all have chunky sections to their solution sequences and are all grouped near one another, even though the

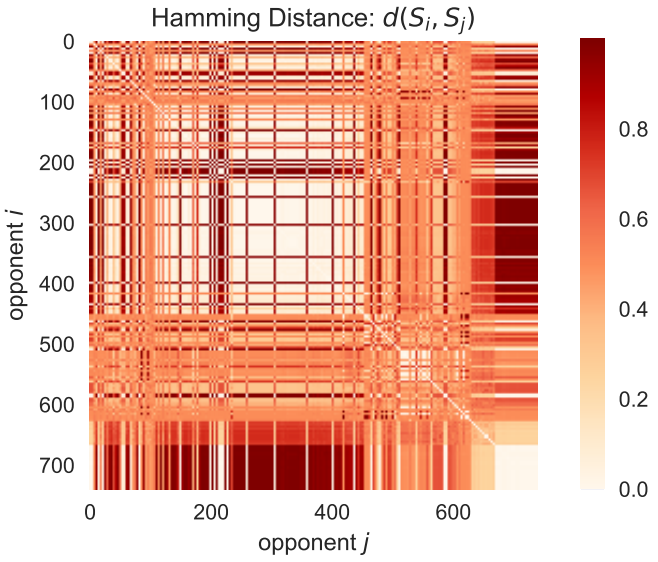

Figure 6.3: Distance Matrix for Hamming Distance

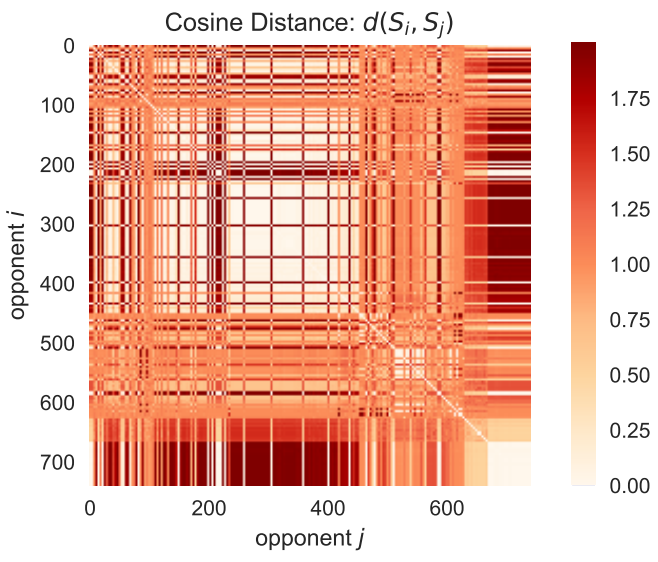

Figure 6.4: Distance Matrix for Cosine Distance

hamming distance between any two in different families could be vastly different. The bottom of the left figure shows many occurrences of the largest group, $C199, 1$ with some that have similar strategies with larger tails. Grouping strategies by their solution sequence seems to be very restrictive for providing relationships between strategies and solutions. It may be beneficial to apply some binary pattern recognition to the solutions, the solutions all seem quite chaotic that could have patterns in other areas of mathematics.

Of the initial sequences that were predicted to occur as solutions and set as our starting population, as described in Section 5.6, 29 of them appeared as actual solution sequences. This is shown below.

- C199,1 with 96 opponents
- C198,2 with 21 opponents
- C196,4 with 3 opponents
- C194,6 with 1 opponent
- C193,7 with 2 opponents
- C100,100 with 8 opponents
- C5,195 with 3 opponents
- C2,1,1,196 with 1 opponent
- C2,198 with 2 opponents
- C1,1,1,1,... with 17 opponents
- C1,1,1,197 with 1 opponent
- C1,2,1,196 with 1 opponent
- C1,199 with 11 opponents
- D1,198,1 with 21 opponents
- D1,196,3 with 1 opponent
- D1,194,5 with 2 opponents
- D1,4,195 with 4 opponents
- D1,3,196 with 4 opponents
- D1,2,197 with 9 opponents
- D2,197,1 with 3 opponents
- D2,196,2 with 2 opponents
- D2,195,3 with 1 opponent
- D2,194,4 with 2 opponents
- D2,193,5 with 1 opponent
- D2,1,197 with 1 opponent
- D3,196,1 with 4 opponents
- D3,194,3 with 1 opponent
- D3,192,5 with 8 opponents
- D200 with 113 opponents

6.4 Clustering Analysis

Here we will look at ways of grouping opponents based on their best response sequences and what that means for scores and potential hidden relationships between strategies. These clustering algorithms are not meant for predictive purposes and are all forms of unsupervised learning; there is not enough metadata present in each strategy definition to use as features for supervised prediction of our best response sequences. These clustering algorithms come from the python library SciKit Learn [43].

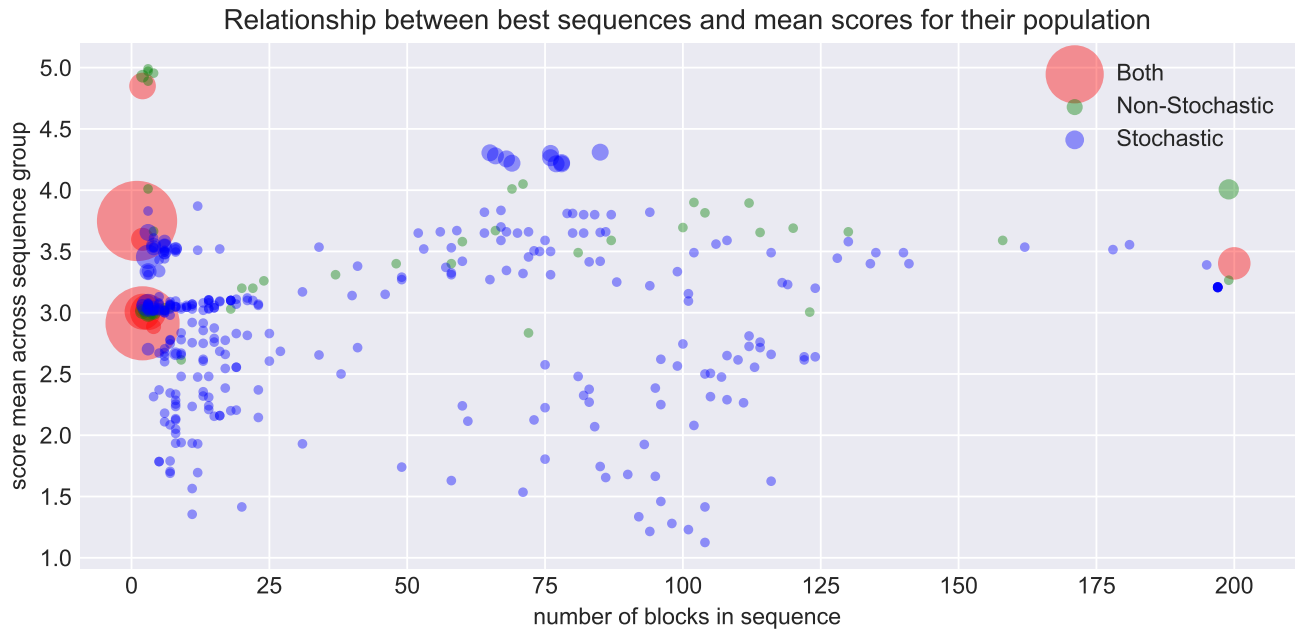

Figure 6.5: Trends for opponents grouped by their best sequence. Dot size represents the number of opponents in the group.

6.4.1 K Means clustering [46]

Here we attempted to label k clusters depending on their parameters. Nothing much was identified from the k means clustering data analysis, typically clusters were found to strongly correlate with parameters such as number of blocks or mean block length. Figure 6.7 shows the clustering for the most correlated variables in 3 dimensions. Its clear that these clusters are layered over the number of blocks of the solutions sequence. Section 6.3 looks in depth as to how opponents are distributed over the solution sequences.

6.4.2 Regression Trees [48]

Regression trees are a way of looking at reducing the variance of a parameter through attributes of observed instances. SciKit Learn has an implementation of regression trees, the code used to generate the data in this subsection is shown in Appendix code B.4.

Our case, we are looking at reducing the variance of the score by looking at which turns tend to cause the largest disparity in score. To do this the algorithm look at reducing the Mean Absolute Error (MAE) of the scores, which is, in effect, reducing the $L1$ norm (sometimes referred to as the L_1 loss of the algorithm) of the scores.

$$\text{MAE} = \frac{1}{n}\sum_{i=0}^{n}|x_i - y_i|$$

Figure 6.8 shows which moves cause the largest variance in resulting score per turn. As you move right on the tree (or move in the false direction on the tree) then playing cooperations at the moves listed. The diagram backs up that by playing C moves consistently (furthest right leaf) we reach a score value of 3.01, and with 325 samples and $MAE = 0.205$ this is a clear way of doing well against a large number of opponents. If we move left on the diagram then the score value increases, which is to be expected; these are the best turns to defect and get a higher score as a result.

The move that dictates the best change in score is move 164 (starting from the 0th turn), if we defect on this move then

Rank	Name	Median score	Cooperation rating	Wins
0	Cycler: C1,199	4.9625	0.005	4.0
1	Tricky Defector	2.23625	0.2475	1.5
2	SolutionB1	1.7675	0.71775	2.0
3	Defector Hunter	1.508125	0.995	0.0
4	Willing	1.5	0.849	0.5

Table 6.3: Table of test opponents

the score value goes up but so does the MAE. Looking at leaf nodes provides an overview of solutions being grouped into minimal MAE after considering 5 moves. These represent which moves are played on the same turn by multiple solutions, all who scored approximately the same. It is clear that there are some paths that, considering one move difference, have clusters of very different scores. For example, the 3rd and 4th groups from the left (with 19 and 4 members respectively) show that, on the turn 156, you can defect and most likely get around a 3.2 or cooperate and get half that score. This shows that even though some solutions are similar, the corresponding opponents are vastly different in operation and will punish at some points others would forgive.

This tree does not have very many useful properties other than displaying the complexities in trying to predict a solution for any opponent. As discussed in Section 6.3 if an opponent does not sit in one of the major groups it becomes incredibly hard to observe some sort of pattern between them. One solution may be to look into combinations of initial handshakes that create the largest variation of responses in the population of opponents; from this we could map these results to a best response sequence to play for the remainder of the game. Section 7.4 looks into this in more detail.

6.5 Conclusion

This chapter includes the most useful results from the analysis conducted on the data output. The results are showing signs of patterns between the complexities in the solution sequence and the strategies they correspond to. However there is no strong correlation or predictor that can be seen to allow accurate prediction of how to beat an opponent

The data produced by this report allows us to easily identify basic responses to certain predetermined tournaments. For example, in a tournament against players Defector Hunter, Solution B1, Willing and Tricky Defector we can play $C1, 199$ and easily come out on top. The code in Appendix Section B.10 produced the results in Table 6.3. It is clear that our solution sequence came out on top for this set of opponents, however if we played this against a different set we would score very low. If all players in the tournament have the same solution then selecting the absolute sequence to play is trivial, however if we have multiple solutions then the problem becomes one of identifying which opponent we are playing in any round. This problem is looked into in more detail in Section 7.4.

We have looked at various methods of grouping and reducing the solutions to try and observe a clear reason for an opponent having a certain solution; yet within the data we have created there seems to be no clear sign of correlation. If the information here is to be assessed further it would be beneficial to gather more data on each opponent to work with. The lack of usable opponent meta-data means that the GA, currently, is the only reasonable method of identifying solution sequences.

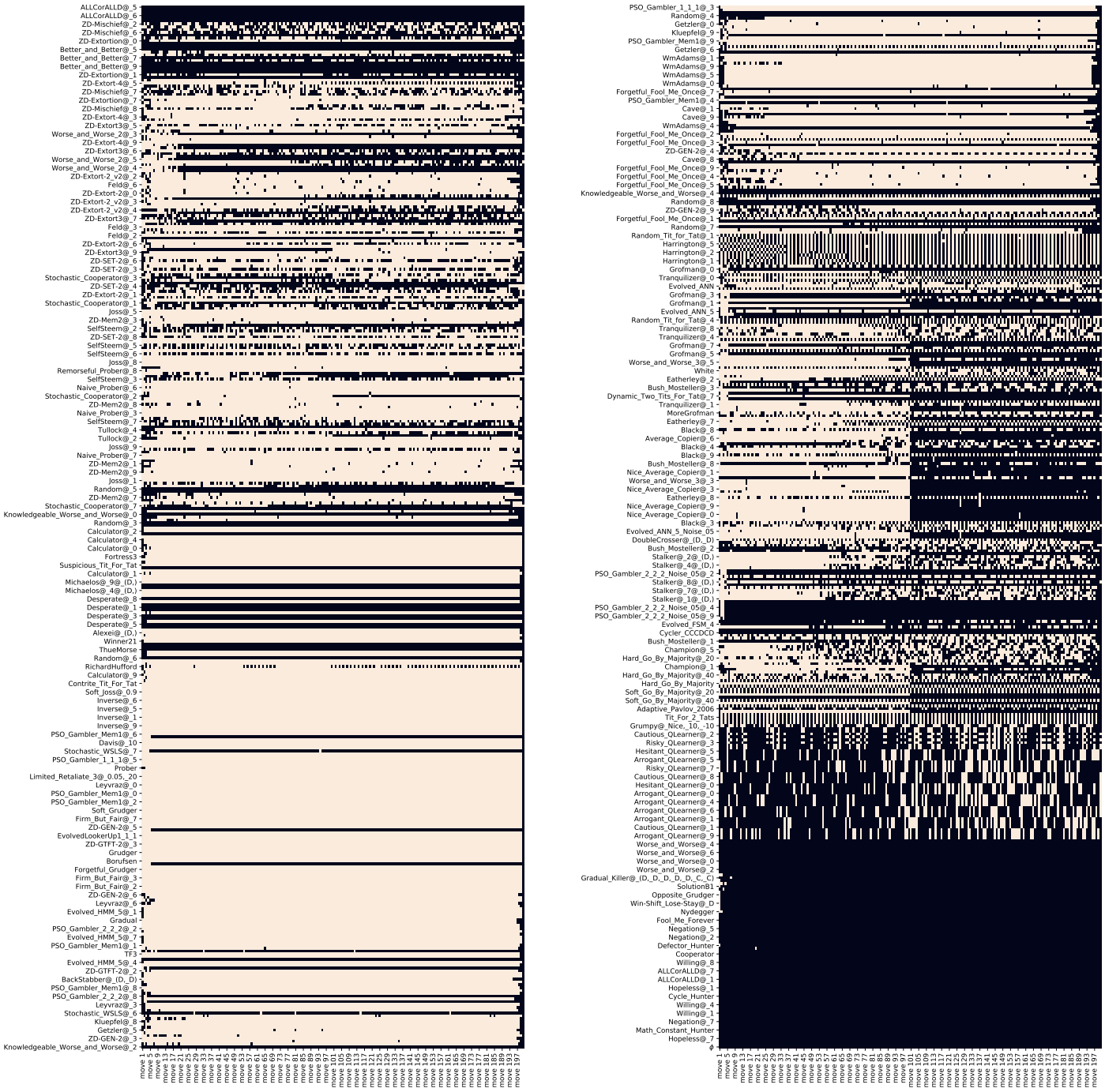

Figure 6.6: Sequence Diagram, sorted by score; High: top left → bottom right: Low. C is light and D is Dark. *NOTE: labels are not complete, approx 1 in 4 shown.*

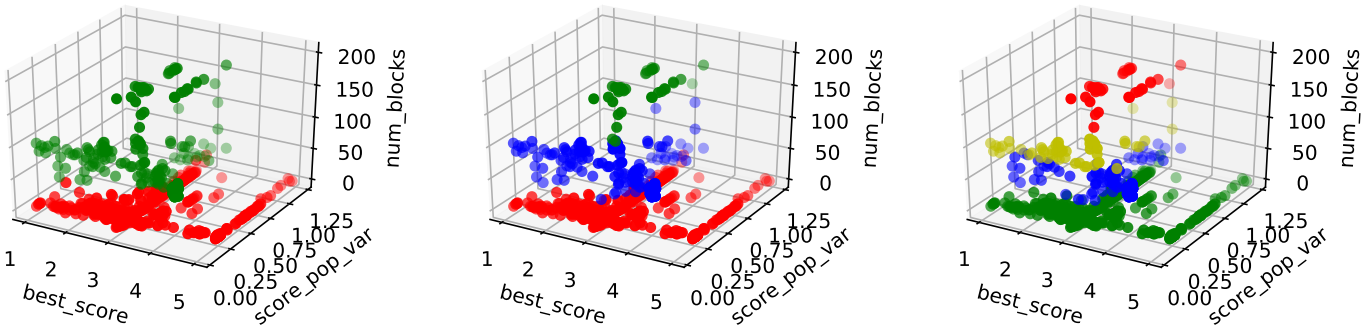

Figure 6.7: K means clustering with 2,3 and 4 clusters

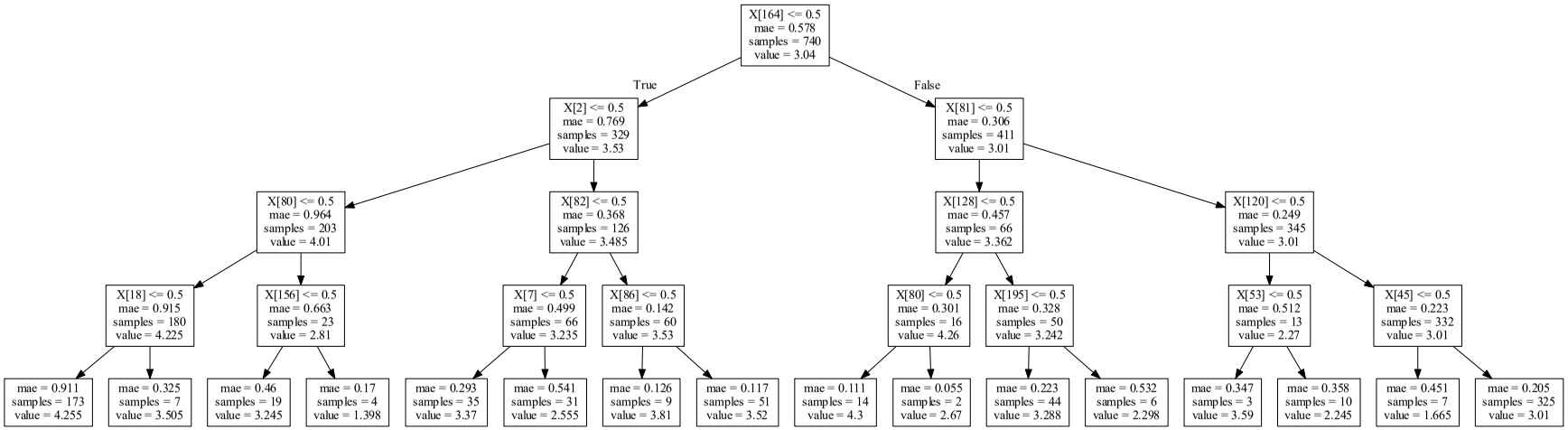

Figure 6.8: A regression tree showing which moves introduce the largest absolute error in the best score. If $X[i] <= 0.5$ is true, it means move i is a Defection move. **TRUE or left** $\Rightarrow D$, **FALSE or right** $\Rightarrow C$

Chapter 7

Conclusions

This report has looked at the concept, structure and generation of solution sequences to strategies within the IPD game. It has identified a successful method of generating these solutions using an evolutionary algorithm and executed analysis to find these solutions to the majority of opponents listed in the Python Axelrod Library. After these solutions were generated analysis was performed into how we could group opponents with similar solutions and considered any patterns that existed within the data we produced.

Each section of this Chapter reflects on a substantive piece of work giving useful content and/or leading to productive discussion. Section 7.1 looks at the first period of the project; researching relevant topics, writing code, creating relevant tools and building a rapport with supervisors. Section 7.2 looks at the execution of the algorithm and how well the problem described in Section 1.3 was solved. Section 7.3 covers where the work completed in this report can be applied in practice. Finally Section 7.4 identifies areas that could be followed up in further work by myself or others.

7.1 Reflection of Approach

Undertaking this project was the first piece of collaborative work I'd undertaken at Cardiff University and, as such, meant learning new skill to apply to what I would be doing. My project supervisors were incredibly helpful and inclusive towards questions I had and suggestions to work on. As with any project there were small teething problems with the scope of work I was expected to complete and the time frame it was expected to be completed in. These were worked out quickly and the work was then accelerated with regular meetings to review progress and set goals. The level of supervision allowed me to stay focused on the goal of the project while also peruse areas of the project I personally found interesting.

The initial part of the project was focused on background to the PD, ML, previous research and work that was to be used further on in the project. Learning Git and the rest of the content in Chapter 4 took less time than expected, this lead to an extended period of analysing the algorithm leading to a successful result overall. This time could have also been used to properly scope and implement a larger expansion of the Axelrod Dojo codebase; I could have picked up more issues that had been previously put forward by the owners on GitHub, increasing this projects contribution to open source research software.

7.2 Summary of Analysis Execution

During the analysis we were finding a sequence that will manipulate our opponent into providing us the most number of cooperation moves we can subsequently defect against without retaliation. The genetic algorithm was just one method of approaching this problem. While executing the genetic algorithm there could be some extra implantations which could improve runtime performance and the results overall.

- We know that the best score per turn we can achieve is a 5.0; adding a check at the end of every generation for the top scoring result for this score per turn may have allowed us to complete the search faster. The LRT strategies, which took over 900 hours of computation, may have found the best result with one of the initial population but were not added to this report due to unneeded extra computation.
- Suboptimal results were given for 57 strategies, as mentioned in Section 6.2. By definition we know we could continue the search until a result with a defection move on turn 200 occurs before terminating the number of generations
- Improving the genetic algorithm further could have improved runtimes and potentially shown better results. This could have been done using more sophisticated crossover and mutation methods or the use of multi population models [49].

Once the data had been collected the analysis was purely descriptive. The extra data, not included in the output, was limited to some classifiers provided in the Axelrod library. From observation these didn't add much to the analysis other than the stochastic variable for each opponent. If any models for predicting solution sequences are to be built, much more data (other than their explicit class structure) would be required from each opponent to identify it from a pool of others. During my research I found no other content I could use as predictor variables; Section 7.4 discusses potential data that could be used to predict sequences in a game environment.

7.3 Applications of results

The application of these results can be leveraged clearly in an IPD instance. As described in Section 7.4 and 6.5 we can 'solve' tournaments to win them. As long as we can identify the opponent (if they are stochastic we would also need to know the seed) in a game we can play our solution sequence to get an optimal result. However this information typically is not provided; Section 7.4 looks at problems with identifying opponents.

Outside of a purely IPD setting the largest take away is understanding that the algorithm was finding methods of manipulating the opponent for the highest cooperation moves we can defect against without encoring a penalty. Along with the fact the solution sequence with the largest number of opponents was $C199, 1$, representing the idea that, for your own highest benefit, you should be cooperative up until your opponent cannot react any more at which point you should defect.

7.4 Potential follow up work

This report has backed up the idea that there is no existence of one universal strategy that will beat every opponent. Theoretically, if a method of identifying an opponent without affecting the games scores could be created, a lookup to the results of this report could be introduced and the solution sequence could be played for the remainder of the game. There are 2 flaws to creating a strategy with this approach:

1. As of writing this report there are 231 strategies listed in the Axelrod library. For simplicity we can create this 'perfect' strategy for only non stochastic opponents[1], all 138 that we analysed. Of these there are 43 solution sequences that contain only non stochastic opponents and another 9 that contain both stochastic and non stochastic, meaning we have a total of 52 sequences to select from as we start a game. From here we have to predict, with the minimal amount of moves in the game, which sequence to play in order to beat our opponent. This however leads us to the second problem:
2. Creating enough variance in order to identify the solution sequence to play may ruin our chances to score well. For example, lets take Collective Strategy as an opponent. When we start our game we wont know were playing Collective Strategy and we will most likely miss the CD handshake, in turn we will have a constant defector to play against for the rest of the game. This will lead to us not being able to play $C1, 1, 197, 1$ and getting the 3.0, we would have to play D to stop our losses.

[1] Using only these we will know the exact solution every time

Because of these reasons, in order to beat any opponent we have to first find a way of identify that opponent without harming our opportunity to play our solution sequence. Further work could be done to investigate a method of predicting which opponent we are playing without impact on the current game being played. This could involve statistical data generated within a match, a package of information players are allowed to analyse before the game or some technique to play each strategy against one another to identify the player the same way some LRT strategies do.

This work also opens up the observation that there is a lack of meta-data about each strategy in the Axelrod Library. The library does a good job of creating a central location for computational IPD tournaments, however the information for separating and identifying strategies, other than their explicit class structure, relies on 5 classifier variables. Expanding on data regarding each strategy, whether it be analyticity or statistically, may allow more sophisticated algorithms to be applied to research on the complexities of how individual IPD strategies are linked.

Finally the results found in this report have weak correlations and patterns with regard to the areas of analysis undertaken. Other areas of mathematics, such as pattern recognition or chaos theory, may be able to provide more clarity in the solution sequence data.

Appendix A

Online Resources

- Personal GitHub Code - https://github.com/GitToby/FinalYearProject
- Axelrod Glossary - https://axelrod.readthedocs.io/en/stable/reference/glossary.html

Appendix B

Code Appendix

```
def runGeneticAlgo(opponent, population_size=150, number_of_game_turns=200, cycle_length=200, generations=250,
                   mutation_probability=0.1, mutation_potency=1, reset_file=True):
    cycler_class = axl_dojo.CyclerParams
    cycler_objective = axl_dojo.prepare_objective(name="score", turns=number_of_game_turns, repetitions=1)
    cycler_kwargs = {
        "sequence_length": cycle_length,
        "mutation_probability":mutation_probability,
        "mutation_potency":mutation_potency
    }

    output_file_name = "data/" + str(opponent).replace(" ","_") + ".csv"
    try:
        if reset_file and os.path.isfile(output_file_name):
            os.remove(output_file_name)
    finally:
        print(str(opponent),
              "|| pop size:", population_size,
              "\tturns:", number_of_game_turns,
              "\tcycle len:", cycle_length,
              "\tgens:", generations,
              "\tmut. rate:",mutation_probability,
              "\t, potency:",mutation_potency)

        axl.seed(1)

        population = axl_dojo.Population(params_class=cycler_class,
                                         params_kwargs=cycler_kwargs,
                                         size=population_size,
                                         objective=cycler_objective,
                                         processes=0,
                                         output_filename=output_file_name,
                                         opponents=[opponent],
                                         print_output=False)
        population.run(generations)
        print("\tAnalysis Complete:",output_file_name)
    # Store the file name and opponent name as a tuple
    return output_file_name, str(opponent)
```

Figure B.1: Code for testing the genetic algorithm in Jupyter notebooks.

]

```
class NewAnalysisRun:
    # default options
    opponent_list = []
    output_files = {}
    save_directory = "output/"
    save_file_prefix = ""
    save_file_suffix = ""
    global_seed = 0
    overwrite_files = True
    stochastic_seeds = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

    def __init__(self, sequence_length=20,
                 population_size=25,
                 generation_length=20,
                 mutation_frequency=0.1,
                 mutation_potency=1):
        self.sequence_length = sequence_length
        self.population_size = population_size
        self.generation_length = generation_length
        self.mutation_frequency = mutation_frequency
        self.mutation_potency = mutation_potency

    def get_pre_made_pop(self, pop_size: int):
        pop = []

        # 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))

        # Random Filler
        while len(pop) < pop_size:
            random_moves = list(map(axl.Action, np.random.randint(
                0, 1 + 1, (self.sequence_length, 1))))
            pop.append(axl_dojo.CyclerParams(random_moves))
```

```
from sklearn import metrics


def CD_map_to_int(x):
    # Returns D:0 C:1
    return 68 - ord(x)


# Itertuples is a list of (index,col1,col2,..) for each row. We sort them by score first
df_vectors = pd.DataFrame()
df_vectors_strings = pd.DataFrame()
df_generation_max = df_generation_max.sort_values('opponent_name')
df_generation_max = df_generation_max.sort_values('best_score')
for tup in df_generation_max.itertuples():
    # index vals mean the indexing starts at 1 so +1 to these:
    #(0, 'generation'), (1, 'score_mean'), (2, 'score_median'), (3, 'score_pop_var'), (4, 'score_range'), (5, 'best_score
    seq_str = tup[7]
    opponent_name = str(tup[8])
    best_score = tup[6]
    bins = tup[15]
    # Mapping to integers 0 &1
    df_vectors[opponent_name] = list(
        map(CD_map_to_int, seq_str)) + [best_score] + [seq_str] + [bins]

    # No Mapping Cs & Ds
    df_vectors_strings[opponent_name] = list(
        seq_str) + [best_score] + [seq_str] + [bins]


move_cols = ['move ' + str(x+1) for x in range(200)]
# Transposing and labeling moves as were forming it in a rotated way.
# (Its easier to just make 2 dfs than map one to the other)
df_vectors = df_vectors.transpose()
df_vectors.columns = move_cols + ["best_score", "best_sequence", "score_bins"]

df_vectors_strings = df_vectors_strings.transpose()
df_vectors_strings.columns = move_cols + \
    ["best_score", "best_sequence", "score_bins"]

# Manhatten Distance == Hamming Distance in this example
# Intresting examples: cosine(??), hamming, jaccard sim
dist_array_ham = metrics.pairwise.pairwise_distances(
    df_vectors[move_cols], metric='hamming')
# add labels
dist_array_ham = pd.DataFrame(
    dist_array_ham, index=df_vectors.index, columns=df_vectors.index)

dist_array_cos = metrics.pairwise.pairwise_distances(
    df_vectors[move_cols], metric='cosine')
dist_array_cos = pd.DataFrame(
    dist_array_cos, index=df_vectors.index, columns=df_vectors.index)

dist_array_jac = metrics.pairwise.pairwise_distances(
    df_vectors[move_cols], metric='jaccard')
dist_array_cos = pd.DataFrame(
    dist_array_cos, index=df_vectors.index, columns=df_vectors.index)

dist_array_other = metrics.pairwise.pairwise_distances(
    df_vectors[move_cols], metric='correlation')
dist_array_cos = pd.DataFrame(
    dist_array_cos, index=df_vectors.index, columns=df_vectors.index)
```

Figure B.3: Distance Matrix generation code using SciKitLearn.

```
# We use mean absolut error, as we want to penalise the magnitude of the error linearly
# (rmse would how how the variance in our scores is reduced move by move)
regressor_tree = tree.DecisionTreeRegressor(criterion='mae',max_depth=4)
#Rerun this for # of cs in a sequence.
regressor_tree.fit(df_vectors[move_cols],df_vectors["best_score"],)

# This will export the graph to a .dot file, use `dot -Tpng .\tree.dot -o tree.png` in cmd to convert to png
# dot_data_clasif = tree.export_graphviz(regressor_tree, out_file='tree.dot')
dot_data_reg = tree.export_graphviz(regressor_tree, out_file='reg_tree.dot')
# Run on CMD line
!dot -Tpng .\reg_tree.dot -o reg_tree.png
!dot -Tpdf .\reg_tree.dot -o reg_tree.pdf

i = Image.open('./reg_tree.png')
plt.figure(figsize=(20,20))
plt.imshow(i)
```

Figure B.4: Regression trees using SciKitLearn.

```
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 B.5: Code to check multiple populations.

```
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 B.6: Code to check multiple generation lengths.

```
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 B.7: The mutation code as given in the axelrod-dojo.

```
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 B.8: Mutation potency code.

```
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 B.9: Mutation frequency code.

```
players = [axl.DefectorHunter(), axl.SolutionB1(),
           axl.Willing(), axl.TrickyDefector(),
           axl.Cycler("C"+"D"*199)]
tournament = axl.Tournament(players,turns=200)
results = tournament.play()

>>> Index, Rank, Name,            Median score, Coop rating, Wins
>>> 0,     0,    Cycler: C1,199,  4.9625,       0.005,       4.0
>>> 1,     1,    Tricky Defector, 2.23625,      0.2475,      1.5
>>> 2,     2,    SolutionB1,      1.7675,       0.71775,     2.0
>>> 3,     3,    Defector Hunter, 1.508125,     0.995,       0.0
>>> 4,     4,    Willing,         1.5,          0.849,       0.5
```

Figure B.10: Tournament code and results

Appendix C

List of Axelrod Opponents

* means Stochastic Opponent, (LRT) means Long Run Time.

- ϕ
- π
- e
- ALLCorALLD*
- Adaptive
- Adaptive Pavlov 2006
- Adaptive Pavlov 2011
- Adaptive Tit For Tat
- Aggravater
- Alexei
- Alternator
- Alternator Hunter
- Anti Tit For Tat
- AntiCycler
- Appeaser
- Arrogant QLearner*
- Average Copier*
- BackStabber
- Better and Better*
- Black*
- Borufsen
- Bully
- Bush Mosteller*
- Calculator*
- Cautious QLearner*
- Cave*
- Champion*
- Colbert
- CollectiveStrategy
- Contrite Tit For Tat
- Cooperator
- Cooperator Hunter
- Cycle Hunter
- Cycler CCCCCD
- Cycler CCCD
- Cycler CCCDCD
- Cycler CCD
- Cycler DC
- Cycler DDC
- DBS (LRT)
- Davis
- Defector
- Defector Hunter
- Desperate*
- DoubleCrosser
- DoubleResurrection
- Doubler
- Dynamic Two Tits For Tat*
- EasyGo
- Eatherley*
- EugineNier
- Eventual Cycle Hunter
- Evolved ANN
- Evolved ANN 5
- Evolved ANN 5 Noise 05
- Evolved FSM 16
- Evolved FSM 16 Noise 05
- Evolved FSM 4
- Evolved HMM 5*
- EvolvedLookerUp1_1_1
- EvolvedLookerUp2_2_2
- Feld*
- Firm But Fair*
- Fool Me Forever
- Fool Me Once
- Forgetful Fool Me Once*
- Forgetful Grudger
- Forgiver
- Forgiving Tit For Tat
- Fortress3
- Fortress4
- GTFT*
- General Soft Grudger
- Getzler*
- Gladstein
- Go By Majority
- Go By Majority 10
- Go By Majority 20
- Go By Majority 40
- Go By Majority 5
- GraaskampKatzen
- Gradual
- Gradual Killer
- Grofman*
- Grudger
- GrudgerAlternator
- Grumpy
- Handshake
- Hard Go By Majority
- Hard Go By Majority 10

- Hard Go By Majority 20
- Hard Go By Majority 40
- Hard Go By Majority 5
- Hard Prober
- Hard Tit For 2 Tats
- Hard Tit For Tat
- Harrington*
- Hesitant QLearner*
- Hopeless*
- Inverse*
- Inverse Punisher
- Joss*
- Kluepfel*
- Knowledgeable Worse and Worse*
- Level Punisher
- Leyvraz*
- Limited Retaliate
- Limited Retaliate 2
- Limited Retaliate 3
- MEM2
- Math Constant Hunter
- Meta Hunter
- Meta Hunter Aggressive
- Meta Majority* (LRT)
- Meta Majority Finite Memory* (LRT)
- Meta Majority Long Memory* (LRT)
- Meta Majority Memory One* (LRT)
- Meta Minority* (LRT)
- Meta Mixer* (LRT)
- Meta Winner* (LRT)
- Meta Winner Deterministic* (LRT)
- Meta Winner Ensemble* (LRT)
- Meta Winner Finite Memory* (LRT)
- Meta Winner Long Memory* (LRT)
- Meta Winner Memory One* (LRT)
- Meta Winner Stochastic* (LRT)
- Michaelos*
- More Tideman and Chieruzzi
- MoreGrofman
- N Tit(s) For M Tat(s)
- NMWE Deterministic* (LRT)
- NMWE Finite Memory* (LRT)
- NMWE Long Memory* (LRT)
- NMWE Memory One* (LRT)
- NMWE Stochastic* (LRT)
- Naive Prober*
- Negation*
- Nice Average Copier*
- Nice Meta Winner* (LRT)
- Nice Meta Winner Ensemble* (LRT)
- Nydegger
- Omega TFT
- Once Bitten
- Opposite Grudger
- PSO Gambler 1_1_1*
- PSO Gambler 2_2_2*
- PSO Gambler 2_2_2 Noise 05*
- PSO Gambler Mem1*
- Predator
- Prober
- Prober 2
- Prober 3
- Prober 4
- Pun1
- Punisher
- Raider
- Random*
- Random Hunter
- Random Tit For Tat*
- Remorseful Prober*
- Resurrection
- Retaliate
- Retaliate 2
- Retaliate 3
- Revised Downing
- RichardHufford
- Ripoff
- Risky QLearner*
- SelfSteem*
- ShortMem
- Shubik
- Slow Tit For Two Tats 2
- Sneaky Tit For Tat
- Soft Grudger
- Soft Joss*
- SolutionB1
- SolutionB5
- Spiteful Tit For Tat
- Stalker*
- Stein and Rapoport
- Stochastic Cooperator*
- Stochastic WSLS*
- Suspicious Tit For Tat
- TF1
- TF2
- TF3
- Tester
- ThueMorse
- ThueMorseInverse
- Thumper
- Tideman and Chieruzzi
- Tit For 2 Tats
- Tit For Tat
- Tranquilizer*
- Tricky Cooperator
- Tricky Defector
- Tullock*
- Two Tits For Tat
- VeryBad
- Weiner
- White
- Willing*
- Win-Shift Lose-Stay
- Win-Stay Lose-Shift
- Winner12
- Winner21
- WmAdams*
- Worse and Worse*
- Worse and Worse 2*
- Worse and Worse 3*
- Yamachi
- ZD-Extort-2*
- ZD-Extort-2 v2*
- ZD-Extort-4*
- ZD-Extort3*
- ZD-Extortion*
- ZD-GEN-2*
- ZD-GTFT-2*
- ZD-Mem2*
- ZD-Mischief*
- ZD-SET-2*

Appendix D

List of Best Solution Sequences

Total sequences:369. Total opponents:740

* means Stochastic Opponent.

@_n means seeded with integer n.

- C199,1 for 96 opponent(s):
  - Aggravater (score: 2.965)
  - Borufsen (score: 3.01)
  - Cave@_5* (score: 3.01)
  - Contrite_Tit_For_Tat (score: 3.01)
  - Davis@_10 (score: 3.01)
  - EvolvedLookerUp1_1_1 (score: 3.01)
  - Feld@_3* (score: 2.29)
  - Firm_But_Fair@_0* (score: 3.01)
  - Firm_But_Fair@_1* (score: 3.01)
  - Firm_But_Fair@_2* (score: 3.01)
  - Firm_But_Fair@_3* (score: 3.01)
  - Firm_But_Fair@_4* (score: 3.01)
  - Firm_But_Fair@_5* (score: 3.01)
  - Firm_But_Fair@_6* (score: 3.01)
  - Firm_But_Fair@_7* (score: 3.01)
  - Firm_But_Fair@_8* (score: 3.01)
  - Firm_But_Fair@_9* (score: 3.01)
  - Forgetful_Grudger (score: 3.01)
  - General_Soft_Grudger@_n=1,d=4,c=2 (score: 3.01)
  - Getzler@_2* (score: 3.01)
  - GraaskampKatzen (score: 3.01)
  - Grudger (score: 3.01)
  - Hard_Tit_For_Tat (score: 3.01)
  - Inverse@_0* (score: 3.01)
  - Inverse@_1* (score: 3.01)
  - Inverse@_2* (score: 3.01)
  - Inverse@_3* (score: 3.01)
  - Inverse@_4* (score: 3.01)
  - Inverse@_5* (score: 3.01)
  - Inverse@_6* (score: 3.01)
  - Inverse@_7* (score: 3.01)
  - Inverse@_8* (score: 3.01)
  - Inverse@_9* (score: 3.01)
  - Inverse_Punisher (score: 3.01)
  - Joss@_0* (score: 2.665)
  - Joss@_1* (score: 2.8)
  - Joss@_2* (score: 2.65)
  - Joss@_3* (score: 2.695)
  - Joss@_4* (score: 2.635)
  - Joss@_5* (score: 2.59)
  - Joss@_7* (score: 2.77)
  - Joss@_8* (score: 2.65)
  - Joss@_9* (score: 2.755)
  - Kluepfel@_1* (score: 3.01)
  - Kluepfel@_4* (score: 3.01)
  - Leyvraz@_0* (score: 3.01)
  - Limited_Retaliate@_0.1,_20 (score: 3.01)
  - Limited_Retaliate_2@_0.08,_15 (score: 3.01)
  - Limited_Retaliate_3@_0.05,_20 (score: 3.01)
  - MEM2 (score: 3.01)
  - Meta_Hunter@_6_players (score: 3.01)
  - Naive_Prober@_0* (score: 2.665)
  - Naive_Prober@_1* (score: 2.8)
  - Naive_Prober@_2* (score: 2.65)
  - Naive_Prober@_3* (score: 2.695)
  - Naive_Prober@_4* (score: 2.635)
  - Naive_Prober@_5* (score: 2.59)
  - Naive_Prober@_7* (score: 2.77)
  - Naive_Prober@_8* (score: 2.65)

- Naive_Prober@_9* (score: 2.755)
- PSO_Gambler_1_1_1@_0* (score: 3.01)
- PSO_Gambler_1_1_1@_2* (score: 3.01)
- PSO_Gambler_1_1_1@_5* (score: 3.01)
- PSO_Gambler_1_1_1@_6* (score: 3.01)
- PSO_Gambler_Mem1@_0* (score: 3.01)
- PSO_Gambler_Mem1@_2* (score: 3.01)
- PSO_Gambler_Mem1@_5* (score: 3.01)
- PSO_Gambler_Mem1@_6* (score: 3.01)
- Punisher (score: 3.01)
- Remorseful_Prober@_1* (score: 2.8)
- Remorseful_Prober@_9* (score: 2.755)
- Resurrection (score: 3.01)
- Retaliate@_0.1 (score: 3.01)
- Retaliate_2@_0.08 (score: 3.01)
- Retaliate_3@_0.05 (score: 3.01)
- Shubik (score: 3.01)
- Soft_Grudger (score: 3.01)
- Soft_Joss@_0.9* (score: 3.01)
- SolutionB5 (score: 2.995)
- Spiteful_Tit_For_Tat (score: 3.01)
- Suspicious_Tit_For_Tat (score: 2.995)
- Thumper (score: 3.01)
- Tideman_and_Chieruzzi (score: 3.01)
- Tit_For_Tat (score: 3.01)
- Two_Tits_For_Tat (score: 3.01)
- ZD-Extort-4@_2* (score: 1.825)
- ZD-Extort-4@_9* (score: 1.945)
- ZD-GEN-2@_2* (score: 3.01)
- ZD-GEN-2@_5* (score: 3.01)
- ZD-GTFT-2@_0* (score: 3.01)
- ZD-GTFT-2@_1* (score: 3.01)
- ZD-GTFT-2@_3* (score: 3.01)
- ZD-GTFT-2@_5* (score: 3.01)
- ZD-GTFT-2@_6* (score: 3.01)
- ZD-GTFT-2@_8* (score: 3.01)
- ZD-GTFT-2@_9* (score: 3.01)

- C198,2 for 21 opponent(s):
  - Alexei@_(D,) (score: 3.0)
  - EugineNier@_(D,) (score: 3.0)
  - Fool_Me_Once (score: 3.02)
  - Kluepfel@_0* (score: 3.02)
  - Kluepfel@_5* (score: 3.02)
  - Michaelos@_0@_(D,)* (score: 3.0)
  - Michaelos@_1@_(D,)* (score: 3.0)
  - Michaelos@_2@_(D,)* (score: 3.0)
  - Michaelos@_3@_(D,)* (score: 3.0)
  - Michaelos@_4@_(D,)* (score: 3.0)
  - Michaelos@_5@_(D,)* (score: 3.0)
  - Michaelos@_6@_(D,)* (score: 3.0)
  - Michaelos@_7@_(D,)* (score: 3.0)
  - Michaelos@_8@_(D,)* (score: 3.0)
  - Michaelos@_9@_(D,)* (score: 3.0)
  - PSO_Gambler_2_2_2@_2* (score: 3.02)
  - PSO_Gambler_2_2_2@_6* (score: 3.02)
  - PSO_Gambler_Mem1@_1* (score: 3.02)
  - PSO_Gambler_Mem1@_3* (score: 3.02)
  - PSO_Gambler_Mem1@_8* (score: 3.02)
  - TF3 (score: 3.02)
- C196,4 for 3 opponent(s):
  - Gradual (score: 3.02)
  - More_Tideman_and_Chieruzzi (score: 3.02)
  - ZD-GEN-2@_3* (score: 3.04)
- C194,6 for 1 opponent(s):
  - BackStabber@_(D,_D) (score: 3.02)
- C193,7 for 2 opponent(s):
  - PSO_Gambler_Mem1@_4* (score: 3.07)
  - PSO_Gambler_Mem1@_9* (score: 3.05)
- C151,1,47,1 for 1 opponent(s):

  - Feld@_2* (score: 2.315)

- C100,26,1,9,1,63 for 3 opponent(s):
  - Dynamic_Two_Tits_For_Tat@_9* (score: 3.545)
  - Nice_Average_Copier@_9* (score: 3.545)
  - Worse_and_Worse_3@_9* (score: 3.545)

- C100,36,1,63 for 1 opponent(s):
  - Average_Copier@_9* (score: 3.535)

- C100,100 for 8 opponent(s):
  - Average_Copier@_5* (score: 3.365)
  - Dynamic_Two_Tits_For_Tat@_3* (score: 3.3)
  - Dynamic_Two_Tits_For_Tat@_5* (score: 3.38)
  - Hard_Go_By_Majority (score: 3.985)
  - Nice_Average_Copier@_5* (score: 3.38)
  - Soft_Go_By_Majority (score: 4.0)
  - VeryBad (score: 4.0)
  - Worse_and_Worse_3@_5* (score: 3.38)

- C99,1,3,1,1,1,2,2,3,1,2,1,1,2,10,2,1,1,1,1,2,1,3,1,1,1,3,1,3,1,1,1,3,1,3... for 1 opponent(s):
  - Level_Punisher (score: 3.4)

- C98,2,1,4,1,9,2,1,3,2,2,2,1,1,1,1,1,1,2,4,1,1,1,2,1,1,2,4,1,5,1,1,2,4,4... for 1 opponent(s):
  - Dynamic_Two_Tits_For_Tat@_2* (score: 3.29)

- C95,3,1,1,1,3,1,3,1,2,1,1,3,1,5,1,1,1,7,3,6,1,1,3,3,2,2,6,1,2,1,1,1,1,1... for 1 opponent(s):
  - Dynamic_Two_Tits_For_Tat@_0* (score: 3.27)

- C92,3,3,1,2,1,2,1,2,1,11,1,1,4,7,1,5,1,1,3,4,1,1,2,4,1,1,1,1,1,1,1,1,2,2... for 1 opponent(s):
  - White (score: 3.4)

- C87,1,1,3,1,1,2,2,1,1,1,3,1,3,2,1,2,2,4,1,1,1,3,3,2,1,4,1,1,3,2,2,2,2,2... for 1 opponent(s):
  - Feld@_8* (score: 2.24)

- C87,3,2,2,6,12,1,87 for 2 opponent(s):
  - Nice_Average_Copier@_6* (score: 3.52)
  - Worse_and_Worse_3@_6* (score: 3.52)

- C87,3,2,2,6,100 for 1 opponent(s):
  - Average_Copier@_6* (score: 3.495)

- C86,1,93,1,18,1 for 1 opponent(s):

  - Remorseful_Prober@_4* (score: 2.645)

- C85,1,12,1,1,74,1,25 for 1 opponent(s):
  - Dynamic_Two_Tits_For_Tat@_8* (score: 3.495)

- C84,1,5,1,1,1,1,3,1,3,1,1,1,3,3,2,1,1,1,2,2,1,1,1,1,7,1,6,1,2,1,4,1,1,4... for 1 opponent(s):
  - Stalker@_5@_(D,)* (score: 3.65)

- C79,1,54,1,63,2 for 1 opponent(s):
  - Adaptive_Tit_For_Tat@_0.5 (score: 3.04)

- C78,1,1,1,1,3,1,2,4,1,6,2,3,1,1,1,1,3,1,1,2,2,1,1,2,1,4,1,1,1,2,3,3,1,5... for 1 opponent(s):
  - Feld@_5* (score: 2.115)

- C71,1,28,26,1,73 for 2 opponent(s):
  - Nice_Average_Copier@_4* (score: 3.585)
  - Worse_and_Worse_3@_4* (score: 3.585)

- C71,1,17,9,1,6,6,1,1,1,2,2,9,2,4,1,1,1,3,1,2,1,2,1,2,10,3,3,2,3,6,1,2,3,3... for 1 opponent(s):
  - Dynamic_Two_Tits_For_Tat@_1* (score: 3.38)

- C71,3,1,2,7,3,2,3,5,1,3,6,6,4,1,1,3,1,1,1,1,1,3,4,1,4,2,6,1,1,4,1,2,1,2... for 1 opponent(s):
  - Stalker@_7@_(D,)* (score: 3.66)

- C67,1,1,1,1,1,1,1,1,1,1,1,3,1,1,1,1,1,1,1,3,1,3,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - Eatherley@_5* (score: 3.445)

- C66,1,1,2,2,1,7,1,5,2,2,1,2,1,1,1,1,2,1,1,1,2,1,5,2,1,4,7,2,3,2,1,3,2,2... for 1 opponent(s):
  - Stalker@_6@_(D,)* (score: 3.65)

- C65,3,1,2,3,2,3,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,3,1,1... for 1 opponent(s):
  - Tranquilizer@_6* (score: 3.2)

- C64,2,34,100 for 1 opponent(s):
  - Average_Copier@_8* (score: 3.53)

- C64,3,33,100 for 1 opponent(s):
  - Average_Copier@_4* (score: 3.61)

- C62,1,2,2,2,1,3,1,7,1,9,1,1,2,3,4,5,5,5,2,1,3,4,1,2,1,6,3,2,1,1,2,2,1,3... for 1 opponent(s):

  - Tranquilizer@_9* (score: 3.37)

- C59,1,116,1,21,2 for 1 opponent(s):
  - Feld@_6* (score: 2.18)

- C58,1,1,1,1,1,1,2,1,2,2,2,2,2,5,5,1,1,1,3,1,2,2,3,2,2,3,1,1,2,1,3,1,2,5... for 1 opponent(s):
  - Stalker@_2@_(D,)* (score: 3.65)

- C57,1,42,8,1,44,1,46 for 1 opponent(s):
  - Average_Copier@_1* (score: 3.52)

- C56,5,4,1,3,2,5,1,2,1,1,1,1,1,1,2,3,1,2,7,2,1,2,1,2,8,2,1,1,1,1,1,1,1,5... for 1 opponent(s):
  - Stalker@_1@_(D,)* (score: 3.66)

- C55,1,2,1,1,1,1,1,1,4,1,1,1,3,2,1,5,1,3,1,2,2,7,1,1,1,1,4,1,1,2,2,2,4,1... for 1 opponent(s):
  - Stalker@_4@_(D,)* (score: 3.65)

- C55,2,2,1,2,1,1,1,1,1,10,1,22,14,1,85 for 1 opponent(s):
  - Dynamic_Two_Tits_For_Tat@_6* (score: 3.52)

- C53,1,3,1,1,2,2,1,1,3,1,5,2,3,3,3,2,1,1,2,1,2,5,1,1,3,1,2,2,1,4,1,4,1,1... for 1 opponent(s):
  - Stalker@_3@_(D,)* (score: 3.66)

- C52,1,32,1,113,1 for 1 opponent(s):
  - Remorseful_Prober@_3* (score: 2.705)

- C52,1,2,1,5,1,2,2,1,2,3,1,1,1,3,1,2,1,2,3,1,2,3,1,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - Eatherley@_9* (score: 3.58)

- C50,1,49,33,1,7,1,58 for 2 opponent(s):
  - Nice_Average_Copier@_1* (score: 3.525)
  - Worse_and_Worse_3@_1* (score: 3.525)

- C50,1,14,2,1,1,4,3,3,1,6,2,2,1,2,1,1,2,3,4,1,1,1,3,1,1,1,2,1,1,1,1,1,2,3... for 1 opponent(s):
  - Eatherley@_6* (score: 3.5)

- C50,1,1,1,7,2,4,1,1,1,1,2,1,1,2,1,1,3,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - Eatherley@_1* (score: 3.49)

- C50,1,1,1,1,1,1,1,1,1,1,2,30,1,4,3,1,2,2,1,3,2,4,3,1,1,4,1,2,7,5,1,1,2,1... for 1 opponent(s):

  - Tranquilizer@_4* (score: 3.345)

- C48,1,15,1,4,1,1,1,11,1,10,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - ZD-Extort-4@_5* (score: 1.625)

- C47,2,1,3,2,2,2,1,27,3,1,3,2,4,1,1,2,1,1,3,6,1,1,3,1,1,3,3,1,3,3,1,1,1,4... for 1 opponent(s):
  - Doubler (score: 3.67)

- C46,1,51,1,1,100 for 2 opponent(s):
  - Nice_Average_Copier@_8* (score: 3.49)
  - Worse_and_Worse_3@_8* (score: 3.49)

- C44,1,55,100 for 3 opponent(s):
  - Average_Copier@_7* (score: 3.55)
  - Nice_Average_Copier@_7* (score: 3.55)
  - Worse_and_Worse_3@_7* (score: 3.55)

- C44,1,2,1,2,3,1,1,1,2,1,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2... for 1 opponent(s):
  - Champion@_3* (score: 3.8)

- C42,4,22,1,5,1,5,2,2,1,5,1,1,1,1,2,3,1,2,3,3,1,3,4,3,1,1,1,6,3,1,1,2,1,1... for 1 opponent(s):
  - Tranquilizer@_1* (score: 3.455)

- C37,4,24,5,1,4,2,2,1,2,2,7,2,3,1,1,6,2,3,1,1,1,1,1,3,1,2,1,2,1,1,2,1,1,1... for 1 opponent(s):
  - Champion@_9* (score: 3.82)

- C35,1,1,1,1,2,3,3,19,1,1,1,2,2,3,1,3,1,1,2,1,2,3,1,1,4,2,1,2,1,1,1,1,1,1... for 1 opponent(s):
  - Eatherley@_0* (score: 3.42)

- C31,1,1,3,1,2,2,1,2,1,1,2,3,1,1,2,2,2,2,1,1,3,3,2,2,2,1,1,2,3,1,2,1,1,4... for 1 opponent(s):
  - Champion@_5* (score: 3.8)

- C31,2,40,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,5,3,1,4,3,2... for 1 opponent(s):
  - Tranquilizer@_5* (score: 3.31)

- C29,1,147,1,21,1 for 1 opponent(s):
  - Remorseful_Prober@_0* (score: 2.675)

- C25,1,23,1,2,1,2,2,1,1,4,1,2,6,1,1,1,1,3,1,1,1,1,1,1,3,2,1,2,1,1,1,1,2,1... for 1 opponent(s):

  - Champion@_6* (score: 3.81)

- C22,1,24,4,4,4,1,1,2,1,1,4,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - Eatherley@_2* (score: 3.4)

- C21,1,28,1,2,1,4,1,15,1,10,1,29,1,5,1,6,1,52,1,8,10 for 1 opponent(s):
  - Forgiving_Tit_For_Tat (score: 3.2)

- C16,1,41,1,2,1,1,1,3,1,1,2,1,2,5,1,1,5,1,2,1,1,4,2,1,2,1,1,2,1,2,1,2,2,3... for 1 opponent(s):
  - Stalker@_0@_(D,)* (score: 3.65)

- C14,1,85,100 for 3 opponent(s):
  - Average_Copier@_3* (score: 3.53)
  - Nice_Average_Copier@_3* (score: 3.53)
  - Worse_and_Worse_3@_3* (score: 3.53)

- C13,1,1,1,1,1,1,181 for 1 opponent(s):
  - Worse_and_Worse_2@_8* (score: 2.255)

- C12,3,2,2,4,2,4,1,2,1,4,1,1,1,1,1,154,4 for 1 opponent(s):
  - Cave@_4* (score: 3.1)

- C11,1,2,1,1,2,1,181 for 1 opponent(s):
  - Worse_and_Worse_2@_4* (score: 2.125)

- C11,3,2,1,1,2,2,3,3,3,2,2,11,2,1,1,1,5,2,1,1,2,1,1,1,8,1,40,1,14,1,31,2,37 for 1 opponent(s):
  - Stochastic_Cooperator@_6* (score: 2.655)

- C10,1,21,1,25,1,20,1,2,1,33,1,4,1,7,1,56,2,2,10 for 1 opponent(s):
  - Forgiver (score: 3.2)

- C10,1,2,1,2,1,2,181 for 1 opponent(s):
  - Worse_and_Worse_2@_3* (score: 1.935)

- C10,3,4,1,1,2,1,178 for 1 opponent(s):
  - Worse_and_Worse_2@_6* (score: 2.05)

- C9,1,90,26,1,73 for 2 opponent(s):
  - Nice_Average_Copier@_2* (score: 3.485)
  - Worse_and_Worse_3@_2* (score: 3.485)

- C9,1,90,57,1,42 for 2 opponent(s):

- Nice_Average_Copier@_0* (score: 3.55)
- Worse_and_Worse_3@_0* (score: 3.55)

- C9,1,2,1,2,1,2,182 for 1 opponent(s):
  - Worse_and_Worse_2@_7* (score: 2.285)
- C8,1,5,2,1,183 for 1 opponent(s):
  - Worse_and_Worse_2@_5* (score: 2.11)
- C7,1,1,1,1,1,1,1,1,1,2,1,2,1,1,1,2,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - DoubleCrosser@_(D,_D) (score: 3.59)
- C6,1,18,1,39,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - Eatherley@_7* (score: 3.49)
- C6,1,6,1,2,1,2,181 for 1 opponent(s):
  - Worse_and_Worse_2@_0* (score: 2.015)
- C6,1,4,1,4,1,2,181 for 1 opponent(s):
  - Worse_and_Worse_2@_1* (score: 2.135)
- C5,1,35,1,1,1,9,1,3,1,1,1,2,2,1,4,1,2,2,1,2,1,1,2,1,3,5,3,2,1,4,1,2,1,1... for 1 opponent(s):
  - Stalker@_9@_(D,)* (score: 3.66)
- C5,1,5,1,55,1,20,2,4,2,1,5,2,1,2,1,2,2,1,1,2,6,1,2,3,1,2,2,1,1,4,4,7,1,3... for 1 opponent(s):
  - Evolved_ANN_5_Noise_05 (score: 3.58)
- C5,1,3,2,1,1,1,1,3,1,1,1,5,1,1,1,1,1,3,3,2,1,1,3,4,1,2,1,3,2,1,4,1,5,7... for 1 opponent(s):
  - Yamachi (score: 3.9)
- C5,1,1,1,2,1,4,1,4,2,5,1,1,1,3,1,2,2,1,3,1,1,1,2,1,1,2,1,2,1,1,2,2,1,1... for 1 opponent(s):
  - Eatherley@_8* (score: 3.535)
- C5,4,2,3,4,1,2,3,1,1,3,1,1,2,86,1,11,1,61,7 for 1 opponent(s):
  - ZD-Extortion@_3* (score: 1.415)
- C5,11,2,8,3,1,1,69,19,1,28,1,50,1 for 1 opponent(s):
  - Stochastic_Cooperator@_3* (score: 2.48)
- C5,195 for 3 opponent(s):

  - Appeaser (score: 3.01)
  - GrudgerAlternator (score: 3.01)
  - Win-Stay_Lose-Shift@_C (score: 3.01)

- C4,1,72,1,2,1,33,1,14,1,31,1,34,1,2,1 for 1 opponent(s):
  - ZD-Mem2@_6* (score: 2.685)
- C4,1,56,1,26,1,1,1,9,55,1,44 for 1 opponent(s):
  - Average_Copier@_0* (score: 3.51)
- C4,1,39,1,55,100 for 1 opponent(s):
  - Dynamic_Two_Tits_For_Tat@_7* (score: 3.44)
- C4,1,5,1,188,1 for 1 opponent(s):
  - Remorseful_Prober@_5* (score: 2.6)
- C4,1,5,1,5,1,5,1,5,1,3,1,5,1,5,1,4,1,1,1,5,1,4,1,5,1,5,1,5,1,5,1,1,1,4... for 1 opponent(s):
  - DoubleResurrection (score: 2.835)
- C4,1,4,2,2,2,2,1,1,1,1,1,7,1,3,1,1,1,2,1,1,4,1,2,2,2,1,2,7,2,2,1,1,1,2... for 1 opponent(s):
  - ZD-GEN-2@_1* (score: 3.14)
- C4,1,2,1,1,3,4,2,1,1,2,1,1,1,1,2,1,1,1,1,2,1,2,1,3,1,4,2,1,1,1,2,1,1,2... for 1 opponent(s):
  - SelfSteem@_9* (score: 2.725)
- C4,89,1,106 for 1 opponent(s):
  - Stochastic_WSLS@_7* (score: 3.01)
- C3,1,44,1,4,1,15,1,12,2,10,1,16,1,12,1,71,4 for 1 opponent(s):
  - ZD-Extort-2@_5* (score: 2.2)
- C3,1,10,1,37,1,32,1,33,1,11,1,16,1,49,2 for 1 opponent(s):
  - Forgetful_Fool_Me_Once@_3* (score: 3.09)
- C3,1,6,1,38,1,2,2,1,1,1,1,4,1,2,3,1,3,1,2,2,1,1,1,1,1,3,1,1,1,1,3,2,1,2... for 1 opponent(s):
  - Stalker@_8@_(D,)* (score: 3.655)
- C3,1,2,1,1,1,2,1,63,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - Eatherley@_3* (score: 3.4)
- C3,1,2,2,1,2,3,3,2,5,3,2,169,2 for 1 opponent(s):

  - Cave@_7* (score: 3.105)

- C3,1,2,2,1,4,4,1,1,2,29,1,148,1 for 1 opponent(s):
  - Cave@_6* (score: 3.035)

- C3,1,1,1,3,1,3,1,1,1,2,1,3,1,1,1,2,1,3,1,1,1,2,1,3,1,1,1,1,1,1,1,2,1,3... for 1 opponent(s):
  - Evolved_FSM_16_Noise_05 (score: 3.66)

- C3,2,45,1,29,1,24,1,6,1,1,1,14,1,63,7 for 1 opponent(s):
  - ZD-Extort-2_v2@_6* (score: 2.16)

- C3,2,26,1,6,1,30,1,13,1,10,1,102,3 for 1 opponent(s):
  - ZD-Extort-2_v2@_5* (score: 2.24)

- C3,2,4,2,1,2,2,2,3,1,4,2,2,1,1,1,3,3,2,1,1,1,3,1,1,1,1,1,4,2,2,2,1,2,2... for 1 opponent(s):
  - Hard_Go_By_Majority@_20 (score: 3.815)

- C3,2,3,3,1,1,1,2,1,2,3,3,5,4,1,5,2,1,1,1,1,1,2,1,1,1,7,1,1,1,2,1,3,1,2... for 1 opponent(s):
  - ZD-Extort3@_5* (score: 1.925)

- C3,2,2,1,3,1,3,1,1,2,3,1,3,1,3,3,4,1,2,3,6,3,3,1,3,1,1,1,2,2,3,1,1,1,2... for 1 opponent(s):
  - Hard_Go_By_Majority@_10 (score: 3.695)

- C3,2,1,4,2,2,4,3,4,3,3,4,1,2,2,1,5,1,3,1,3,5,1,2,1,3,2,2,1,2,3,5,1,1,1... for 1 opponent(s):
  - ZD-Mischief@_5* (score: 1.215)

- C3,6,24,1,25,1,8,1,11,1,9,1,14,1,23,1,4,1,21,1,28,1,4,10 for 1 opponent(s):
  - Evolved_ANN (score: 3.26)

- C3,97,24,1,34,1,40 for 1 opponent(s):
  - Stochastic_Cooperator@_5* (score: 2.345)

- C2,1,196,1 for 3 opponent(s):
  - Remorseful_Prober@_8* (score: 2.655)
  - TF1 (score: 3.0)
  - Tester (score: 3.005)

- C2,1,97,26,1,73 for 1 opponent(s):
  - Average_Copier@_2* (score: 3.495)

- C2,1,61,1,132,3 for 1 opponent(s):

  - Cave@_2* (score: 3.02)

- C2,1,53,1,22,1,9,1,27,1,8,1,14,1,49,1,6,2 for 1 opponent(s):
  - Forgetful_Fool_Me_Once@_4* (score: 3.1)

- C2,1,4,1,125,1,63,3 for 1 opponent(s):
  - Getzler@_1* (score: 3.045)

- C2,1,4,1,18,1,73,1,97,2 for 1 opponent(s):
  - Forgetful_Fool_Me_Once@_7* (score: 3.06)

- C2,1,4,1,4,2,2,184 for 1 opponent(s):
  - Worse_and_Worse_2@_9* (score: 2.235)

- C2,1,2,1,2,1,1,3,2,1,1,2,2,1,2,2,1,1,1,1,1,2,3,1,1,1,1,1,3,2,3,3,3,2,2... for 1 opponent(s):
  - Hard_Go_By_Majority@_40 (score: 3.895)

- C2,1,2,1,1,1,2,1,1,1,3,2,5,2,4,1,2,1,1,1,2,1,1,1,2,1,1,1,2,1,1,1,3,1,3... for 1 opponent(s):
  - Hard_Go_By_Majority@_5 (score: 3.655)

- C2,1,1,1,194,1 for 1 opponent(s):
  - Calculator@_7* (score: 3.0)

- C2,1,1,1,192,3 for 1 opponent(s):
  - ZD-GTFT-2@_2* (score: 3.02)

- C2,1,1,1,72,1,121,1 for 2 opponent(s):
  - Joss@_6* (score: 2.665)
  - Naive_Prober@_6* (score: 2.665)

- C2,1,1,3,192,1 for 1 opponent(s):
  - Omega_TFT@_3,_8 (score: 3.015)

- C2,1,1,196 for 1 opponent(s):
  - EvolvedLookerUp2_2_2 (score: 4.955)

- C2,2,80,5,11,100 for 1 opponent(s):
  - Dynamic_Two_Tits_For_Tat@_4* (score: 3.51)

- C2,2,45,1,5,1,48,1,50,1,35,9 for 1 opponent(s):

  - ZD-Extort3@_0* (score: 1.93)

- C2,2,18,1,26,1,5,1,48,1,50,1,41,3 for 1 opponent(s):
  - ZD-Extort-2@_0* (score: 2.21)

- C2,2,18,1,26,1,5,1,48,1,50,1,21,1,13,9 for 1 opponent(s):
  - ZD-Extort-2_v2@_0* (score: 2.16)

- C2,2,2,2,1,3,2,2,2,2,2,2,2,2,2,2,2,2,14,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2... for 1 opponent(s):
  - Leyvraz@_8* (score: 3.22)

- C2,3,82,3,2,3,2,1,2,3,2,2,3,2,2,1,4,2,1,1,1,1,2,1,4,6,1,1,1,1,1,5,7,1,1... for 1 opponent(s):
  - ZD-Extortion@_9* (score: 1.74)

- C2,3,54,1,36,1,29,1,23,1,31,1,15,2 for 1 opponent(s):
  - Forgetful_Fool_Me_Once@_9* (score: 3.1)

- C2,3,22,1,27,2,1,1,1,2,2,2,2,1,2,3,3,2,1,2,1,1,2,1,2,1,1,1,2,5,3,2,1,1,1... for 1 opponent(s):
  - ZD-Extort-2@_6* (score: 2.325)

- C2,3,22,1,22,1,4,2,1,2,3,1,2,6,1,1,3,2,1,1,3,1,2,1,1,1,3,1,3,3,1,1,1,3,3... for 1 opponent(s):
  - ZD-Extort3@_6* (score: 2.07)

- C2,3,3,2,2,1,1,2,1,1,3,2,175,2 for 1 opponent(s):
  - ZD-GTFT-2@_7* (score: 3.03)

- C2,3,1,2,2,1,5,2,1,2,3,1,2,1,3,1,3,1,1,1,2,1,1,1,2,6,1,1,2,1,3,1,1,4,2... for 1 opponent(s):
  - ZD-SET-2@_9* (score: 2.745)

- C2,6,2,6,1,4,8,1,16,4,1,1,11,9,20,100,8 for 1 opponent(s):
  - Stochastic_Cooperator@_0* (score: 2.66)

- C2,6,1,3,1,1,1,1,1,1,1,1,1,3,1,7,1,7,3,2,3,1,3,1,4,3,1,1,1,1,1,1,2,6,4... for 1 opponent(s):
  - ZD-SET-2@_5* (score: 2.29)

- C2,13,1,21,1,47,1,1,4,3,1,1,4,3,3,1,4,3,2,4,2,6,1,1,1,3,2,1,1,1,2,2,1,1,3... for 1 opponent(s):
  - Bush_Mosteller@_3* (score: 3.42)

- C2,198 for 2 opponent(s):

  - Adaptive (score: 4.88)
  - Nydegger (score: 4.98)

- C1,1,197,1 for 3 opponent(s):
  - CollectiveStrategy (score: 3.0)
  - Gladstein (score: 3.005)
  - Ripoff (score: 3.005)

- C1,1,196,2 for 2 opponent(s):
  - PSO_Gambler_2_2_2@_5* (score: 3.03)
  - PSO_Gambler_2_2_2@_9* (score: 3.03)

- C1,1,5,1,3,2,5,1,2,1,1,4,1,5,2,3,2,1,1,3,3,2,2,1,1,3,1,1,1,1,1,2,2,2,2... for 1 opponent(s):
  - ZD-Extortion@_5* (score: 1.415)

- C1,1,3,1,18,2,2,1,2,1,1,1,1,1,3,3,6,4,4,1,2,1,1,2,1,1,7,1,2,1,2,2,2,1,2... for 1 opponent(s):
  - Champion@_4* (score: 3.82)

- C1,1,2,1,194,1 for 1 opponent(s):
  - Calculator@_1* (score: 3.0)

- C1,1,2,1,162,1,30,2 for 1 opponent(s):
  - PSO_Gambler_2_2_2@_3* (score: 3.05)

- C1,1,2,1,121,1,23,1,47,2 for 1 opponent(s):
  - Kluepfel@_9* (score: 3.045)

- C1,1,2,1,9,1,22,1,11,2,1,1,2,1,4,2,6,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,11,3,2... for 1 opponent(s):
  - Tranquilizer@_3* (score: 3.25)

- C1,1,2,1,6,1,187,1 for 1 opponent(s):
  - Feld@_0* (score: 2.335)

- C1,1,2,1,4,1,12,1,6,1,21,1,9,1,42,1,93,2 for 1 opponent(s):
  - PSO_Gambler_2_2_2@_0* (score: 3.1)

- C1,1,2,1,4,1,7,1,1,1,2,3,2,1,53,1,4,1,2,2,1,2,1,1,1,2,5,2,5,3,2,1,4,1,2... for 1 opponent(s):
  - Tranquilizer@_8* (score: 3.32)

- C1,1,2,1,2,1,17,1,16,1,50,1,104,2 for 1 opponent(s):
  - PSO_Gambler_2_2_2@_7* (score: 3.08)

- C1,1,2,1,1,1,3,1,1,1,2,1,1,1,2,1,3,2,3,1,2,1,1,1,3,2,4,2,3,1,1,1,3,1,3... for 1 opponent(s):

- Soft_Go_By_Majority@_5 (score: 3.69)

- C1,1,2,1,1,1,3,2,3,6,1,2,1,1,2,1,3,1,3,3,2,2,4,2,3,5,2,2,1,2,3,2,2,2,2... for 1 opponent(s):
  - ZD-GEN-2@_9* (score: 3.15)
- C1,1,2,1,1,2,1,1,2,1,1,1,1,2,1,1,2,1,1,4,6,4,4,1,1,1,1,1,1,1,2,2,1,1,1... for 1 opponent(s):
  - SelfSteem@_8* (score: 2.76)
- C1,1,1,1,195,1 for 1 opponent(s):
  - Kluepfel@_7* (score: 3.015)
- C1,1,1,1,115,1,79,1 for 1 opponent(s):
  - Remorseful_Prober@_2* (score: 2.665)
- C1,1,1,1,28,1,1,1,87,1,1,1,57,1,1,1,14,1 for 1 opponent(s):
  - Weiner (score: 3.03)
- C1,1,1,1,10,1,152,1,30,2 for 1 opponent(s):
  - Getzler@_3* (score: 3.045)
- C1,1,1,1,2,1,1,4,1,1,1,1,2,1,1,1,1,1,1,2,1,1,4,1,1,3,3,1,1,3,2,5,1,2,1... for 1 opponent(s):
  - ZD-SET-2@_7* (score: 2.555)
- C1,1,1,1,1,1,1,1,2,1,115,1,68,5 for 1 opponent(s):
  - Feld@_9* (score: 2.31)
- C1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1... for 17 opponent(s):
  - Adaptive_Pavlov_2011 (score: 3.97)
  - GTFT@_0.33* (score: 3.085)
  - Harrington@_5* (score: 3.205)
  - Harrington@_7* (score: 3.205)
  - Random_Tit_for_Tat@_0* (score: 3.055)
  - Random_Tit_for_Tat@_1* (score: 3.205)
  - Random_Tit_for_Tat@_2* (score: 3.13)
  - Random_Tit_for_Tat@_3* (score: 3.16)
  - Random_Tit_for_Tat@_4* (score: 3.31)
  - Random_Tit_for_Tat@_5* (score: 3.25)
  - Random_Tit_for_Tat@_6* (score: 3.235)
  - Random_Tit_for_Tat@_7* (score: 3.31)
  - Random_Tit_for_Tat@_8* (score: 3.37)
  - Random_Tit_for_Tat@_9* (score: 3.355)
  - Soft_Go_By_Majority@_10 (score: 4.0)
  - Soft_Go_By_Majority@_20 (score: 4.0)
  - Soft_Go_By_Majority@_40 (score: 4.0)
- C1,1,1,2,54,1,2,1,17,1,31,1,49,1,33,1,2,1 for 1 opponent(s):

  - Forgetful_Fool_Me_Once@_6* (score: 3.1)

- C1,1,1,2,26,1,16,1,15,7,3,1,4,1,2,8,1,89,1,9,10 for 1 opponent(s):
  - Stochastic_Cooperator@_4* (score: 2.815)

- C1,1,1,4,1,18,7,2,31,21,4,103,6 for 1 opponent(s):
  - Stochastic_Cooperator@_7* (score: 2.855)

- C1,1,1,197 for 1 opponent(s):
  - Evolved_FSM_16 (score: 3.665)

- C1,2,196,1 for 1 opponent(s):
  - TF2 (score: 2.99)

- C1,2,2,1,2,1,1,2,58,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,11,87,1,3,1,8 for 1 opponent(s):
  - Eatherley@_4* (score: 3.535)

- C1,2,2,1,2,1,1,2,1,3,3,3,161,1,14,2 for 1 opponent(s):
  - ZD-GEN-2@_4* (score: 3.095)

- C1,2,2,81,1,30,1,82 for 1 opponent(s):
  - Stochastic_WSLS@_4* (score: 3.05)

- C1,2,1,1,86,1,34,1,2,1,67,3 for 1 opponent(s):
  - ZD-Extort-2_v2@_9* (score: 2.475)

- C1,2,1,1,4,1,1,2,1,1,1,2,1,1,1,3,2,2,1,1,1,4,1,2,1,1,1,2,1,4,1,1,1,2,1... for 1 opponent(s):
  - SelfSteem@_6* (score: 2.65)

- C1,2,1,1,1,1,1,3,1,4,2,3,1,1,1,4,6,3,4,4,3,2,3,3,4,3,8,1,11,3,1,7,4,1,1... for 1 opponent(s):
  - ZD-Extortion@_4* (score: 1.68)

- C1,2,1,1,1,2,3,2,3,1,2,4,1,1,1,2,3,1,4,2,1,1,1,1,2,1,1,2,3,2,3,2,1,1,1... for 1 opponent(s):
  - SelfSteem@_1* (score: 2.81)

- C1,2,1,1,1,2,3,2,1,1,1,2,3,2,1,1,1,2,1,1,1,2,1,1,2,1,1,1,2,1,1,1,1,2,1... for 1 opponent(s):
  - SelfSteem@_5* (score: 2.64)

- C1,2,1,1,1,2,3,2,1,1,1,2,1,1,1,2,2,3,1,1,1,2,1,1,2,3,1,2,1,1,1,1,4,1,2... for 1 opponent(s):

  - SelfSteem@_2* (score: 2.615)

- C1,2,1,1,1,2,1,4,1,1,2,1,1,1,2,1,1,1,1,1,2,1,1,1,2,1,1,2,1,1,1,2,3,1,4... for 1 opponent(s):
  - SelfSteem@_3* (score: 2.66)

- C1,2,1,1,1,4,1,2,1,2,1,1,1,2,1,1,1,1,2,1,1,1,1,2,1,1,1,4,1,1,2,1,1,2,1... for 1 opponent(s):
  - SelfSteem@_0* (score: 2.615)

- C1,2,1,4,1,1,4,1,1,4,1,3,3,4,1,4,1,2,1,3,1,7,1,1,4,2,3,1,1,2,1,1,2,1,1... for 1 opponent(s):
  - ZD-Mischief@_6* (score: 1.28)

- C1,2,1,4,1,1,2,1,1,1,1,4,2,2,2,1,3,1,1,1,165,2 for 1 opponent(s):
  - ZD-GEN-2@_8* (score: 3.1)

- C1,2,1,4,1,1,1,3,2,4,1,4,1,2,4,1,1,1,2,1,1,1,1,2,1,1,1,2,1,1,1,1,1,2,1... for 1 opponent(s):
  - SelfSteem@_7* (score: 2.715)

- C1,2,1,62,1,2,1,2,1,2,4,1,1,3,2,1,1,2,2,4,1,1,3,2,2,1,1,1,2,1,1,3,2,1,1... for 1 opponent(s):
  - Bush_Mosteller@_4* (score: 3.5)

- C1,2,1,196 for 1 opponent(s):
  - Stochastic_WSLS@_8* (score: 3.02)

- C1,3,2,2,18,1,168,5 for 1 opponent(s):
  - Getzler@_7* (score: 3.095)

- C1,3,1,1,42,1,1,1,5,1,43,100 for 1 opponent(s):
  - Champion@_0* (score: 3.87)

- C1,3,1,3,1,2,1,5,4,2,5,2,167,3 for 1 opponent(s):
  - Leyvraz@_7* (score: 3.105)

- C1,3,1,3,1,5,1,5,1,7,1,21,1,6,1,3,3,1,1,8,1,1,1,9,2,1,1,1,1,5,2,4,1,2,1... for 4 opponent(s):
  - Arrogant_QLearner@_7* (score: 4.225)
  - Cautious_QLearner@_7* (score: 4.225)
  - Hesitant_QLearner@_7* (score: 4.225)
  - Risky_QLearner@_7* (score: 4.225)

- C1,4,3,1,3,1,46,1,33,1,103,3 for 1 opponent(s):

  - ZD-Extortion@_7* (score: 1.695)

- C1,7,1,1,2,1,1,1,1,2,1,1,2,3,2,1,1,1,2,1,2,1,3,3,1,1,1,2,2,3,2,2,1,4,3... for 1 opponent(s):
  - ZD-Extortion@_8* (score: 1.63)

- C1,45,1,35,1,6,2,1,1,1,1,5,1,4,1,1,2,7,5,1,2,4,2,2,1,4,1,1,1,5,2,5,2,1,4... for 1 opponent(s):
  - Bush_Mosteller@_8* (score: 3.52)

- C1,140,1,58 for 1 opponent(s):
  - Stochastic_WSLS@_1* (score: 3.06)

- C1,199 for 11 opponent(s):
  - Defector_Hunter (score: 4.99)
  - Evolved_FSM_4 (score: 3.67)
  - Handshake (score: 4.97)
  - Opposite_Grudger (score: 4.975)
  - SolutionB1 (score: 4.955)
  - Tricky_Defector (score: 4.915)
  - Willing@_0* (score: 4.975)
  - Willing@_2* (score: 4.975)
  - Willing@_5* (score: 4.975)
  - Willing@_6* (score: 4.975)
  - Win-Shift_Lose-Stay@_D (score: 4.975)

- D1,198,1 for 21 opponent(s):
  - Calculator@_2* (score: 2.945)
  - Calculator@_3* (score: 2.99)
  - Calculator@_4* (score: 2.975)
  - Calculator@_5* (score: 2.945)
  - Calculator@_6* (score: 3.005)
  - Calculator@_8* (score: 2.975)
  - Calculator@_9* (score: 3.005)
  - Evolved_HMM_5@_0* (score: 3.02)
  - Evolved_HMM_5@_1* (score: 3.02)
  - Evolved_HMM_5@_2* (score: 3.02)
  - Evolved_HMM_5@_3* (score: 3.02)
  - Evolved_HMM_5@_4* (score: 3.02)
  - Evolved_HMM_5@_5* (score: 3.02)
  - Evolved_HMM_5@_6* (score: 3.02)
  - Evolved_HMM_5@_7* (score: 3.02)
  - Evolved_HMM_5@_8* (score: 3.02)
  - Evolved_HMM_5@_9* (score: 3.02)
  - PSO_Gambler_1_1_1@_1* (score: 3.02)
  - PSO_Gambler_1_1_1@_8* (score: 3.02)
  - Winner12 (score: 3.02)
  - Winner21 (score: 3.0)

- D1,196,3 for 1 opponent(s):
  - PSO_Gambler_1_1_1@_3* (score: 3.04)

- D1,194,5 for 2 opponent(s):
  - Feld@_7* (score: 2.38)
  - ZD-GTFT-2@_4* (score: 3.025)

- D1,193,1,4,1 for 1 opponent(s):
  - ZD-Extort-4@_8* (score: 1.785)

- D1,99,1,88,1,8,2 for 1 opponent(s):

  - Remorseful_Prober@_7* (score: 2.78)

- D1,99,100 for 1 opponent(s):
  - Revised_Downing@_True (score: 4.01)

- D1,79,1,117,2 for 1 opponent(s):
  - Remorseful_Prober@_6* (score: 2.67)

- D1,76,1,34,1,86,1 for 1 opponent(s):
  - ZD-Extort-4@_6* (score: 1.69)

- D1,70,1,127,1 for 1 opponent(s):
  - Feld@_1* (score: 2.37)

- D1,68,1,10,1,13,1,101,4 for 1 opponent(s):
  - Getzler@_5* (score: 3.03)

- D1,60,1,95,1,39,3 for 1 opponent(s):
  - Forgetful_Fool_Me_Once@_0* (score: 3.06)

- D1,45,1,35,1,19,1,34,1,60,2 for 1 opponent(s):
  - Forgetful_Fool_Me_Once@_8* (score: 3.07)

- D1,42,1,7,1,5,2,3,1,1,1,2,1,2,1,1,1,1,1,1,4,1,1,1,1,2,3,2,1,1,5,2,2,2,3... for 1 opponent(s):
  - Champion@_2* (score: 3.81)

- D1,21,1,20,1,75,1,6,1,24,1,46,2 for 1 opponent(s):
  - Forgetful_Fool_Me_Once@_2* (score: 3.08)

- D1,14,1,183,1 for 1 opponent(s):
  - ZD-Extort-4@_1* (score: 1.785)

- D1,10,1,25,1,14,1,78,1,16,1,49,2 for 1 opponent(s):
  - ZD-Mem2@_3* (score: 2.605)

- D1,9,1,1,1,1,1,3,1,3,1,7,1,19,1,6,1,13,1,11,1,24,1,24,1,7,1,11,1,44,2 for 1 opponent(s):
  - Forgetful_Fool_Me_Once@_1* (score: 3.17)

- D1,7,1,1,1,7,182 for 1 opponent(s):

  - Worse_and_Worse_2@_2* (score: 2.085)

- D1,7,11,1,1,1,3,1,2,1,28,1,75,1,7,1,11,1,46 for 1 opponent(s):
  - Stochastic_Cooperator@_1* (score: 2.555)

- D1,7,192 for 1 opponent(s):
  - Grofman@_2* (score: 3.31)

- D1,5,1,1,1,1,2,4,2,3,1,1,3,45,1,11,1,113,3 for 1 opponent(s):
  - Cave@_1* (score: 3.07)

- D1,5,2,6,1,1,1,5,2,5,1,2,1,4,1,3,1,2,1,4,1,2,1,7,1,1,1,6,1,2,1,3,1,6,2... for 1 opponent(s):
  - MoreGrofman (score: 3.49)

- D1,5,2,1,2,1,1,2,3,1,1,2,1,2,2,2,3,3,5,7,8,1,5,1,2,1,2,2,1,1,1,1,1,1,1... for 1 opponent(s):
  - ZD-SET-2@_3* (score: 2.385)

- D1,5,194 for 2 opponent(s):
  - Grofman@_3* (score: 3.27)
  - Grofman@_9* (score: 3.41)

- D1,4,1,90,1,29,1,23,1,18,1,24,1,4,1 for 1 opponent(s):
  - ZD-Mem2@_9* (score: 2.79)

- D1,4,1,4,1,21,1,20,1,26,1,13,1,29,1,34,1,34,6 for 1 opponent(s):
  - Forgetful_Fool_Me_Once@_5* (score: 3.11)

- D1,4,1,1,2,4,1,14,2,10,1,1,1,14,1,5,1,11,1,13,1,4,1,14,1,13,1,23,1,15,1,20,1,2,1... for 1 opponent(s):
  - Evolved_ANN_5 (score: 3.31)

- D1,4,2,5,1,2,1,2,2,3,1,1,4,65,1,1,2,5,3,2,2,1,2,3,1,7,2,5,3,3,3,1,3,4,1... for 1 opponent(s):
  - Tranquilizer@_2* (score: 3.325)

- D1,4,20,1,174 for 2 opponent(s):
  - Grofman@_6* (score: 3.31)
  - Grofman@_7* (score: 3.37)

- D1,4,178,1,16 for 1 opponent(s):
  - Grofman@_4* (score: 3.43)

- D1,4,195 for 4 opponent(s):

  - Grofman@_0* (score: 3.22)
  - Grofman@_1* (score: 3.3)
  - Grofman@_5* (score: 3.38)
  - Grofman@_8* (score: 3.44)

- D1,3,1,172,1,21,1 for 1 opponent(s):
  - ZD-Extort-4@_0* (score: 1.705)
- D1,3,1,80,1,1,1,3,1,5,2,2,1,1,1,1,2,1,1,1,2,2,2,1,4,2,1,1,2,3,3,1,1,1,3... for 1 opponent(s):
  - Feld@_4* (score: 2.225)
- D1,3,1,52,1,50,1,24,1,7,1,11,1,43,1,1,1 for 1 opponent(s):
  - ZD-Mem2@_1* (score: 2.775)
- D1,3,1,1,4,2,1,17,3,1,1,1,1,1,1,1,2,2,1,4,1,1,1,3,2,1,3,4,2,2,3,11,1,10,1... for 1 opponent(s):
  - ZD-Mem2@_5* (score: 2.715)
- D1,3,196 for 4 opponent(s):
  - PSO_Gambler_2_2_2_Noise_05@_0* (score: 3.655)
  - PSO_Gambler_2_2_2_Noise_05@_2* (score: 3.655)
  - PSO_Gambler_2_2_2_Noise_05@_5* (score: 3.655)
  - PSO_Gambler_2_2_2_Noise_05@_6* (score: 3.655)
- D1,2,1,18,1,7,1,26,1,13,1,69,1,52,4,1,1 for 1 opponent(s):
  - ZD-Extort-2_v2@_1* (score: 2.385)
- D1,2,1,2,90,1,1,1,2,2,1,1,1,1,1,1,1,1,3,1,3,1,2,3,3,2,1,1,2,2,2,2,2,2,2... for 1 opponent(s):
  - Bush_Mosteller@_9* (score: 3.27)
- D1,2,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - Black@_0* (score: 3.39)
- D1,2,1,1,12,3,1,4,2,1,2,1,1,1,1,1,2,2,4,2,2,1,1,1,1,3,1,2,1,2,1,2,2,1,1... for 1 opponent(s):
  - ZD-Mischief@_7* (score: 1.665)
- D1,2,2,59,1,61,1,24,1,43,5 for 1 opponent(s):
  - ZD-Extortion@_2* (score: 1.565)
- D1,2,2,17,1,7,1,26,1,13,1,69,1,52,4,1,1 for 1 opponent(s):
  - ZD-Extort-2@_1* (score: 2.545)
- D1,2,2,9,1,22,1,47,1,33,1,47,1,30,2 for 1 opponent(s):

  - Kluepfel@_3* (score: 3.04)

- D1,2,2,3,1,1,1,3,4,1,2,1,1,1,1,2,2,2,4,2,1,1,4,1,1,1,1,1,1,2,2,1,2,2,1... for 1 opponent(s):
  - Grumpy@_Nice,_10,_-10 (score: 4.05)

- D1,2,3,2,2,2,1,1,1,1,1,2,1,3,4,2,1,2,1,2,1,3,3,1,1,1,2,2,3,2,2,1,1,20,2... for 1 opponent(s):
  - ZD-Mischief@_8* (score: 1.745)

- D1,2,197 for 9 opponent(s):
  - PSO_Gambler_2_2_2@_1* (score: 3.025)
  - PSO_Gambler_2_2_2_Noise_05@_1* (score: 3.665)
  - PSO_Gambler_2_2_2_Noise_05@_7* (score: 3.665)
  - PSO_Gambler_2_2_2@_4* (score: 3.065)
  - PSO_Gambler_2_2_2_Noise_05@_3* (score: 3.665)
  - PSO_Gambler_2_2_2_Noise_05@_8* (score: 3.665)
  - PSO_Gambler_2_2_2@_8* (score: 3.025)
  - PSO_Gambler_2_2_2_Noise_05@_4* (score: 3.665)
  - PSO_Gambler_2_2_2_Noise_05@_9* (score: 3.665)

- D1,1,1,116,1,11,1,16,1,47,4 for 1 opponent(s):
  - ZD-Extort3@_3* (score: 1.935)

- D1,1,1,82,1,33,1,28,1,47,4 for 1 opponent(s):
  - ZD-Extort-2_v2@_3* (score: 2.235)

- D1,1,1,82,1,33,1,11,1,16,1,47,4 for 1 opponent(s):
  - ZD-Extort-2@_3* (score: 2.32)

- D1,1,1,3,1,6,1,2,1,2,1,37,1,139,3 for 1 opponent(s):
  - Cave@_0* (score: 3.04)

- D1,1,1,3,3,7,1,2,1,1,1,1,3,1,1,3,1,5,1,2,4,2,1,1,1,3,2,2,2,2,2,2,2,2,2... for 1 opponent(s):
  - Black@_1* (score: 3.56)

- D1,1,1,2,2,1,1,2,1,3,5,21,2,24,2,22,1,4,1,29,1,2,1,20,1,31,1,11,1,2,3 for 1 opponent(s):
  - ZD-Mischief@_9* (score: 1.93)

- D1,1,1,1,1,194,1 for 1 opponent(s):
  - Calculator@_0* (score: 2.98)

- D1,1,1,1,1,29,1,13,1,7,1,14,1,14,1,54,1,38,1,6,1,3,1,3,5 for 1 opponent(s):
  - ZD-Mem2@_4* (score: 2.605)

- D1,1,1,1,1,22,1,25,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,29,1,1,1,1,1,1,1,1,1... for 1 opponent(s):

  - RichardHufford (score: 3.005)

- D1,1,1,1,1,20,1,173,1 for 1 opponent(s):
  - ZD-Extort-4@_7* (score: 1.94)

- D1,1,1,1,1,17,1,41,1,54,1,31,1,46,2 for 1 opponent(s):
  - ZD-Extort-2_v2@_2* (score: 2.155)

- D1,1,1,1,1,17,1,41,1,54,1,6,1,24,1,30,1,15,2 for 1 opponent(s):
  - ZD-Extort-2@_2* (score: 2.205)

- D1,1,1,1,1,17,1,21,1,44,1,1,1,1,1,1,3,2,1,3,1,1,2,1,2,2,1,3,3,1,1,4,1,3,2... for 1 opponent(s):
  - ZD-Extort3@_2* (score: 2.125)

- D1,1,1,1,1,3,1,1,1,68,1,4,7,1,2,3,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - Tranquilizer@_7* (score: 3.23)

- D1,1,1,1,1,2,1,1,1,1,2,10,1,2,1,9,1,2,1,60,1,88,1,9,1 for 1 opponent(s):
  - ZD-Mem2@_7* (score: 2.83)

- D1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - Sneaky_Tit_For_Tat (score: 3.265)

- D1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1... for 6 opponent(s):
  - Hard_Tit_For_2_Tats (score: 4.01)
  - N_Tit(s)_For_M_Tat(s)@_3,_2
  - (score: 4.01)
  - Once_Bitten (score: 4.01)
  - Pun1 (score: 3.99)
  - Slow_Tit_For_Two_Tats_2 (score: 4.01)
  - Tit_For_2_Tats (score: 4.01)

- D1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2... for 1 opponent(s):
  - Black@_6* (score: 3.415)

- D1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2... for 1 opponent(s):
  - Harrington@_6* (score: 3.205)

- D1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2... for 1 opponent(s):
  - Harrington@_2* (score: 3.21)

- D1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1... for 1 opponent(s):

  - Harrington@_3* (score: 3.21)

- D1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - Harrington@_4* (score: 3.205)
- D1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - Harrington@_9* (score: 3.21)
- D1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - Harrington@_8* (score: 3.21)
- D1,1,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - Harrington@_0* (score: 3.205)
- D1,1,1,1,1,1,1,1,3,2,1,1,1,1,1,1,2,4,1,2,1,1,1,1,1,4,2,1,2,3,1,1,1,3,2... for 1 opponent(s):
  - Black@_9* (score: 3.515)
- D1,1,1,1,2,1,1,1,2,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - Harrington@_1* (score: 3.21)
- D1,1,1,1,6,4,1,1,1,4,3,2,2,2,4,3,1,2,2,2,2,2,6,2,1,1,3,1,1,2,2,2,1,1,1... for 1 opponent(s):
  - ZD-Mischief@_2* (score: 1.125)
- D1,1,2,143,1,49,3 for 1 opponent(s):
  - Getzler@_9* (score: 3.055)
- D1,1,2,4,1,1,1,2,1,2,1,2,1,1,3,1,1,3,1,1,2,165,2 for 1 opponent(s):
  - Cave@_3* (score: 3.06)
- D1,1,2,1,1,1,1,5,2,9,2,3,2,10,1,40,1,2,1,16,1,71,1,8,1,15,1 for 1 opponent(s):
  - ZD-Mem2@_8* (score: 2.685)
- D1,1,3,2,2,1,1,3,4,1,2,2,1,4,3,1,1,2,4,1,2,2,1,1,4,1,2,4,1,1,1,3,2,7,1... for 1 opponent(s):
  - ZD-Extortion@_6* (score: 1.335)
- D1,1,3,1,1,2,2,1,1,3,2,1,1,1,1,4,1,2,1,1,1,1,1,1,1,2,1,1,1,2,1,1,1,1,1... for 1 opponent(s):
  - Black@_3* (score: 3.555)
- D1,1,4,1,4,1,2,1,2,1,1,1,1,1,9,1,2,1,1,1,10,1,8,1,9,1,3,2,2,1,4,1,3,1,5... for 4 opponent(s):

  - Arrogant_QLearner@_9* (score: 4.31)
  - Cautious_QLearner@_9* (score: 4.31)
  - Hesitant_QLearner@_9* (score: 4.31)
  - Risky_QLearner@_9* (score: 4.31)

- D1,1,9,1,14,1,8,1,1,1,6,1,5,1,3,1,7,1,3,1,14,1,6,1,2,1,3,3,5,1,1,1,11,1,1... for 4 opponent(s):
  - Arrogant_QLearner@_8* (score: 4.255)
  - Cautious_QLearner@_8* (score: 4.255)
  - Hesitant_QLearner@_8* (score: 4.255)
  - Risky_QLearner@_8* (score: 4.255)

- D2,197,1 for 3 opponent(s):
  - Fortress3 (score: 2.99)
  - Prober (score: 3.01)
  - Prober_3 (score: 2.995)

- D2,196,2 for 2 opponent(s):
  - Kluepfel@_2* (score: 3.04)
  - Leyvraz@_3* (score: 3.025)

- D2,195,3 for 1 opponent(s):
  - ZD-GEN-2@_6* (score: 3.015)

- D2,194,4 for 2 opponent(s):
  - Leyvraz@_4* (score: 3.06)
  - ZD-GEN-2@_0* (score: 3.025)

- D2,193,5 for 1 opponent(s):
  - PSO_Gambler_1_1_1@_4* (score: 3.07)

- D2,4,1,3,2,1,1,2,1,2,1,2,1,2,2,1,1,1,1,4,1,4,1,1,6,1,1,1,2,5,4,6,1... for 1 opponent(s):
  - ZD-SET-2@_0* (score: 2.315)

- D2,3,2,2,2,1,1,1,1,1,6,1,1,1,1,3,1,1,1,1,3,3,3,1,1,1,2,2,3,2,2,1,4,3,1... for 1 opponent(s):
  - ZD-Extort3@_8* (score: 2.475)

- D2,3,2,2,4,1,4,3,2,5,1,1,1,1,1,2,1,3,3,1,1,1,2,2,3,2,2,1,2,3,2,1,1,1,4... for 1 opponent(s):
  - ZD-Extort-2@_8* (score: 2.505)

- D2,3,3,2,1,2,1,1,1,1,1,1,1,2,3,2,1,13,1,155,3 for 1 opponent(s):
  - Cave@_8* (score: 3.1)

- D2,3,46,1,125,1,22 for 1 opponent(s):
  - Stochastic_WSLS@_0* (score: 3.09)

- D2,2,1,107,1,1,1,14,1,46,1,19,1,2,1 for 1 opponent(s):

  - Kluepfel@_6* (score: 3.06)

- D2,2,1,17,1,26,1,7,1,97,1,43,1 for 1 opponent(s):
  - ZD-Mem2@_0* (score: 2.62)
- D2,2,1,17,1,7,1,19,1,6,1,13,1,36,1,24,1,7,1,11,1,42,4 for 1 opponent(s):
  - ZD-Extort3@_1* (score: 2.145)
- D2,2,1,4,1,1,2,3,1,1,1,1,3,1,3,1,1,5,2,1,7,5,3,3,1,1,7,2,1,5,1,3,2,2,2... for 1 opponent(s):
  - ZD-SET-2@_6* (score: 2.375)
- D2,2,1,2,1,1,1,1,3,2,1,1,1,3,3,5,1,2,3,4,4,4,4,4,4,4,4,6,2,5,1,1,2,3,1... for 1 opponent(s):
  - Black@_5* (score: 3.335)
- D2,2,2,33,1,42,1,2,1,12,1,3,1,34,1,36,1,19,1,1,1,1,2 for 1 opponent(s):
  - ZD-Extort-2_v2@_8* (score: 2.37)
- D2,1,1,194,2 for 1 opponent(s):
  - Kluepfel@_8* (score: 3.03)
- D2,1,1,58,1,99,1,13,1,20,3 for 1 opponent(s):
  - Getzler@_6* (score: 3.055)
- D2,1,1,47,1,52,1,92,3 for 1 opponent(s):
  - Getzler@_0* (score: 3.045)
- D2,1,1,2,1,1,3,1,1,1,2,1,2,1,1,3,3,1,1,1,2,1,1,4,1,3,1,41,3,2,2,2,3,1,1... for 1 opponent(s):
  - ZD-Extort3@_7* (score: 2.27)
- D2,1,1,1,1,1,1,2,1,3,3,1,1,1,1,1,2,102,1,55,1,14,3 for 1 opponent(s):
  - Cave@_9* (score: 3.07)
- D2,1,1,1,2,2,1,1,1,69,1,15,2,2,1,1,1,1,2,1,1,1,2,2,2,1,4,2,1,1,2,3,3,1,1... for 1 opponent(s):
  - ZD-Extort-4@_4* (score: 1.805)
- D2,1,1,1,2,1,2,2,1,2,1,9,2,17,1,28,1,1,1,2,3,1,2,2,1,2,2,1,1,1,2,2,1,1,1... for 1 opponent(s):
  - ZD-Extort-2_v2@_7* (score: 2.48)
- D2,1,1,1,6,1,73,1,1,2,7,1,3,1,1,1,2,2,4,2,4,2,4,2,2,1,2,1,2,2,6,1,2,1,2... for 1 opponent(s):

  - Bush_Mosteller@_0* (score: 3.31)

- D2,1,2,193,2 for 2 opponent(s):
  - Leyvraz@_1* (score: 3.03)
  - Leyvraz@_2* (score: 3.025)

- D2,1,2,192,3 for 1 opponent(s):
  - Leyvraz@_6* (score: 3.02)

- D2,1,2,186,9 for 1 opponent(s):
  - WmAdams@_7* (score: 3.07)

- D2,1,2,4,1,187,3 for 1 opponent(s):
  - Getzler@_8* (score: 3.03)

- D2,1,2,2,2,1,1,1,1,2,1,2,1,1,1,1,1,1,1,174,1 for 1 opponent(s):
  - PSO_Gambler_1_1_1@_7* (score: 3.12)

- D2,1,2,2,5,1,5,1,9,1,2,1,8,1,1,1,8,1,2,1,5,1,5,1,5,1,9,1,6,2,5,1,9,1,1... for 1 opponent(s):
  - Adaptive_Pavlov_2006 (score: 4.01)

- D2,1,2,1,1,1,24,1,20,1,57,1,88 for 1 opponent(s):
  - Stochastic_WSLS@_5* (score: 3.02)

- D2,1,2,1,2,2,2,2,2,2,2,2,2,2,2,3,1,2,2,2,2,2,2,3,1,2,1,3,2,2,2,2,2,2... for 1 opponent(s):
  - Leyvraz@_5* (score: 3.095)

- D2,1,2,1,2,1,2,4,1,2,2,4,2,1,1,1,1,1,2,1,1,2,1,3,1,3,1,4,1,2,4,1,1,1,1... for 1 opponent(s):
  - Black@_7* (score: 3.59)

- D2,1,3,1,4,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2... for 1 opponent(s):
  - Leyvraz@_9* (score: 3.155)

- D2,1,4,4,1,3,1,1,1,2,2,3,5,1,1,1,1,2,6,1,4,2,1,1,1,1,1,1,2,3,1,1,1,2,2... for 1 opponent(s):
  - ZD-SET-2@_2* (score: 2.5)

- D2,1,7,2,7,4,2,4,1,1,4,1,2,1,3,4,4,1,1,1,3,1,1,4,9,1,12,2,1,2,1,1,1,1,24... for 1 opponent(s):
  - ZD-SET-2@_4* (score: 2.5)

- D2,1,19,1,96,1,31,1,46,1,1 for 1 opponent(s):

  - Stochastic_WSLS@_2* (score: 3.07)

- D2,1,197 for 1 opponent(s):
  - Prober_2 (score: 4.97)

- D3,196,1 for 4 opponent(s):
  - Fortress4 (score: 2.98)
  - Hard_Prober (score: 3.0)
  - Predator (score: 3.0)
  - Raider (score: 3.0)

- D3,194,3 for 1 opponent(s):
  - Stein_and_Rapoport@_0.05@_(D,_D) (score: 3.02)

- D3,192,5 for 8 opponent(s):
  - WmAdams@_0* (score: 3.06)
  - WmAdams@_1* (score: 3.06)
  - WmAdams@_2* (score: 3.06)
  - WmAdams@_3* (score: 3.06)
  - WmAdams@_5* (score: 3.06)
  - WmAdams@_6* (score: 3.06)
  - WmAdams@_8* (score: 3.06)
  - WmAdams@_9* (score: 3.06)

- D3,177,1,2,1,13,3 for 1 opponent(s):
  - Getzler@_4* (score: 3.065)

- D3,97,100 for 1 opponent(s):
  - Champion@_1* (score: 3.83)

- D3,56,1,36,1,32,1,6,1,45,1,9,2,1,5 for 1 opponent(s):
  - Tullock@_9* (score: 2.875)

- D3,24,1,2,1,4,2,1,1,2,2,1,1,1,1,1,1,1,5,2,1,2,3,2,3,3,1,1,1,3,1,1,1,1,1... for 1 opponent(s):
  - Champion@_8* (score: 3.8)

- D3,20,1,1,1,1,1,3,1,1,1,44,1,6,2,1,1,9,1,1,1,1,1,3,1,1,1,1,1,1,1,1,1,1,1... for 1 opponent(s):
  - Tranquilizer@_0* (score: 3.245)

- D3,12,1,151,1,22,10 for 1 opponent(s):
  - ZD-Extort-4@_3* (score: 1.79)

- D3,8,1,1,2,7,21,1,156 for 1 opponent(s):
  - Meta_Hunter_Aggressive@_7_players (score: 2.615)

- D3,4,4,1,2,2,5,1,1,3,2,2,3,2,2,1,5,3,4,2,2,1,1,1,2,1,1,6,2,2,1,2,2,1,1... for 1 opponent(s):

  - ZD-SET-2@_1* (score: 2.565)

- D3,2,9,2,3,2,1,3,2,1,1,7,4,4,1,2,1,2,3,1,1,1,2,1,2,1,2,1,2,2,1,1,4,2,1... for 1 opponent(s):
  - ZD-Mischief@_3* (score: 1.23)

- D3,1,1,86,1,4,1,29,1,2,1,17,1,34,1,5,1,8,3 for 1 opponent(s):
  - ZD-Extort-2@_9* (score: 2.555)

- D3,1,1,54,1,36,1,3,50,1,43,1,5 for 1 opponent(s):
  - Stochastic_Cooperator@_9* (score: 2.75)

- D3,1,1,8,4,1,1,25,1,147,8 for 1 opponent(s):
  - Tullock@_7* (score: 2.92)

- D3,1,1,4,2,2,2,2,1,1,1,2,2,3,2,1,1,1,1,2,2,2,7,28,1,2,2,4,1,1,1,4,1,1,2... for 1 opponent(s):
  - ZD-Mischief@_1* (score: 1.655)

- D3,1,1,3,1,3,1,2,1,2,2,1,2,1,5,1,1,1,1,17,1,32,1,11,5,2,1,2,1,2,8,2,1,1,1... for 1 opponent(s):
  - ZD-Extortion@_1* (score: 1.535)

- D3,1,1,1,1,2,1,1,1,2,1,1,2,1,1,1,2,1,1,1,2,1,1,1,1,2,1,2,1,1,1,2,1,1... for 1 opponent(s):
  - SelfSteem@_4* (score: 2.64)

- D3,1,2,1,6,1,16,1,1,1,2,1,9,1,8,1,5,1,6,1,7,1,6,2,10,1,2,3,3,2,2,2,2,1,3... for 4 opponent(s):
  - Arrogant_QLearner@_6* (score: 4.3)
  - Cautious_QLearner@_6* (score: 4.3)
  - Hesitant_QLearner@_6* (score: 4.3)
  - Risky_QLearner@_6* (score: 4.3)

- D3,1,4,4,5,2,1,1,1,1,3,6,2,1,2,1,1,1,3,2,4,2,3,4,3,1,3,1,1,2,5,2,2,1,1... for 1 opponent(s):
  - ZD-SET-2@_8* (score: 2.62)

- D3,1,81,1,16,1,71,1,25 for 1 opponent(s):
  - Stochastic_Cooperator@_8* (score: 2.48)

- D4,87,1,4,1,29,1,2,1,52,1,14,3 for 1 opponent(s):
  - ZD-Extort3@_9* (score: 2.355)

- D4,39,1,56,19,1,78,1,1 for 1 opponent(s):

  - Stochastic_Cooperator@_2* (score: 2.67)

- D4,18,1,20,1,20,1,54,1,31,1,43,3,1,1 for 1 opponent(s):
  - ZD-Mem2@_2* (score: 2.76)

- D4,5,1,9,1,41,1,42,1,90,5 for 1 opponent(s):
  - ZD-Extortion@_0* (score: 1.355)

- D4,4,4,3,4,3,3,1,3,2,1,3,1,1,1,1,2,3,2,3,2,4,1,2,1,1,1,2,1,1,1,1,1,2,1... for 1 opponent(s):
  - Black@_8* (score: 3.49)

- D4,2,2,3,3,1,1,1,1,1,1,1,3,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,1... for 1 opponent(s):
  - Black@_2* (score: 3.49)

- D4,2,2,1,1,2,2,1,3,3,1,1,1,3,1,4,1,2,1,4,1,2,1,2,1,2,2,5,1,1,1,2,1,1,1... for 1 opponent(s):
  - Black@_4* (score: 3.505)

- D4,2,6,1,12,1,1,2,2,1,11,2,10,1,7,1,2,2,2,1,3,1,2,1,8,2,7,1,1,1,4,3,4,3,4... for 4 opponent(s):
  - Arrogant_QLearner@_3* (score: 4.215)
  - Cautious_QLearner@_3* (score: 4.215)
  - Hesitant_QLearner@_3* (score: 4.215)
  - Risky_QLearner@_3* (score: 4.215)

- D4,1,1,1,2,1,3,2,13,1,1,2,2,1,13,1,8,1,2,2,1,1,4,1,3,2,2,1,16,1,5,2,1,1,1... for 4 opponent(s):
  - Arrogant_QLearner@_2* (score: 4.215)
  - Cautious_QLearner@_2* (score: 4.215)
  - Hesitant_QLearner@_2* (score: 4.215)
  - Risky_QLearner@_2* (score: 4.215)

- D4,1,4,1,14,175,1 for 1 opponent(s):
  - Prober_4 (score: 3.04)

- D4,1,8,1,9,1,5,1,12,1,3,1,5,1,6,1,2,1,10,1,11,2,3,3,2,1,1,3,7,2,1,1,6,1,5... for 4 opponent(s):
  - Arrogant_QLearner@_5* (score: 4.22)
  - Cautious_QLearner@_5* (score: 4.22)
  - Hesitant_QLearner@_5* (score: 4.22)
  - Risky_QLearner@_5* (score: 4.22)

- D5,187,8 for 1 opponent(s):
  - PSO_Gambler_1_1_1@_9* (score: 3.07)

- D5,72,1,51,1,46,1,16,7 for 1 opponent(s):

  - Tullock@_6* (score: 2.78)

- D5,2,1,2,1,1,4,1,1,3,2,3,7,2,3,3,1,2,1,2,1,32,1,12,3,1,1,2,2,3,3,1,1,2,1... for 1 opponent(s):
  - ZD-Extort-2@_7* (score: 2.575)

- D5,1,1,1,1,1,6,2,2,2,2,2,2,2,1,2,1,1,1,5,1,1,1,1,8,1,1,1,4,2,2,3,1,1,2... for 1 opponent(s):
  - ZD-Extort3@_4* (score: 2.08)

- D5,1,3,1,10,1,1,2,16,1,8,2,10,1,5,1,2,1,16,2,6,3,3,1,1,1,2,1,1,1,10,1,4,2,1... for 4 opponent(s):
  - Arrogant_QLearner@_4* (score: 4.28)
  - Cautious_QLearner@_4* (score: 4.28)
  - Hesitant_QLearner@_4* (score: 4.28)
  - Risky_QLearner@_4* (score: 4.28)

- D5,1,188,1,5 for 1 opponent(s):
  - Stochastic_WSLS@_9* (score: 3.08)

- D6,113,1,11,1,16,1,41,10 for 1 opponent(s):
  - Tullock@_3* (score: 2.835)

- D6,1,18,2,17,1,30,4,2,3,6,2,2,1,1,2,1,1,5,2,2,3,1,1,1,1,1,2,3,1,2,2,6,2,6... for 1 opponent(s):
  - Bush_Mosteller@_7* (score: 3.59)

- D6,1,193 for 1 opponent(s):
  - Gradual_Killer@_(D,_D,_D,_D,_D,_C,_C) (score: 4.89)

- D7,64,1,45,1,79,3 for 1 opponent(s):
  - Tullock@_4* (score: 2.745)

- D7,50,1,119,1,16,6 for 1 opponent(s):
  - Tullock@_0* (score: 2.775)

- D7,7,1,1,1,2,1,4,1,1,1,5,1,2,1,2,1,3,1,4,1,4,1,1,1,5,1,1,1,1,1,6,2,2,1... for 1 opponent(s):
  - ShortMem (score: 3.59)

- D7,1,1,1,3,1,4,2,1,1,2,1,1,6,2,1,1,1,1,2,2,6,1,4,1,5,3,1,2,2,1,3,2,4,1... for 1 opponent(s):
  - ZD-Mischief@_4* (score: 1.46)

- D8,186,6 for 1 opponent(s):

  - PSO_Gambler_Mem1@_7* (score: 3.08)

- D8,14,1,103,1,24,1,30,1,14,3 for 1 opponent(s):
  - Tullock@_2* (score: 2.755)

- D8,14,1,7,1,40,1,61,1,7,1,55,3 for 1 opponent(s):
  - Tullock@_1* (score: 2.915)

- D8,4,2,1,3,3,1,1,1,3,1,4,1,1,2,7,3,1,3,3,1,1,1,1,1,1,2,2,1,3,2,1,2,1,1... for 1 opponent(s):
  - ZD-Extort-2_v2@_4* (score: 2.25)

- D8,1,1,9,1,3,4,8,1,5,1,43,1,12,1,38,1,57,5 for 1 opponent(s):
  - Tullock@_8* (score: 2.83)

- D8,1,3,1,3,2,2,2,2,2,2,2,1,2,1,1,1,3,1,2,2,4,5,1,2,1,6,1,3,1,1,3,1,1,1... for 1 opponent(s):
  - ZD-Extort-2@_4* (score: 2.265)

- D8,1,7,1,8,1,13,1,3,1,2,1,12,1,16,1,1,1,2,1,5,1,7,1,5,1,1,1,3,1,1,1,3,1,1... for 4 opponent(s):
  - Arrogant_QLearner@_0* (score: 4.265)
  - Hesitant_QLearner@_0* (score: 4.265)
  - Risky_QLearner@_0* (score: 4.265)
  - Cautious_QLearner@_0* (score: 4.265)

- D9,187,4 for 1 opponent(s):
  - WmAdams@_4* (score: 3.07)

- D9,33,1,8,1,10,2,1,1,1,3,1,3,1,2,7,3,2,3,5,1,3,6,6,4,1,1,3,1,1,1,1,1,3,4... for 1 opponent(s):
  - Champion@_7* (score: 3.835)

- D9,1,24,1,1,2,16,1,11,1,1,1,15,1,3,1,1,2,5,1,1,5,9,1,7,1,2,2,1,1,2,1,3,2,1... for 4 opponent(s):
  - Arrogant_QLearner@_1* (score: 4.305)
  - Hesitant_QLearner@_1* (score: 4.305)
  - Risky_QLearner@_1* (score: 4.305)
  - Cautious_QLearner@_1* (score: 4.305)

- D10,70,1,13,1,64,1,38,2 for 1 opponent(s):
  - Tullock@_5* (score: 2.655)

- D10,15,1,168,6 for 1 opponent(s):
  - ZD-GEN-2@_7* (score: 3.13)

- D14,1,37,1,66,1,80 for 1 opponent(s):

  - Stochastic_WSLS@_3* (score: 3.07)

- D19,1,180 for 1 opponent(s):
  - Eventual_Cycle_Hunter (score: 4.99)

- D27,1,34,1,17,1,33,1,46,1,38 for 1 opponent(s):
  - Stochastic_WSLS@_6* (score: 3.03)

- D48,1,4,1,28,2,14,1,2,3,1,1,1,1,5,2,2,1,1,2,1,2,1,1,1,1,1,2,3,3,1,5,1,1,3... for 1 opponent(s):
  - Bush_Mosteller@_5* (score: 3.53)

- D50,1,6,1,10,2,7,1,2,1,3,1,1,1,8,1,1,2,1,1,8,1,1,1,2,1,2,2,1,3,1,2,2,1,1... for 1 opponent(s):
  - Bush_Mosteller@_1* (score: 3.7)

- D55,1,14,1,2,1,1,1,10,1,9,1,2,1,1,1,2,1,5,2,1,1,1,2,7,2,4,1,1,3,2,2,1,1,1... for 1 opponent(s):
  - Bush_Mosteller@_6* (score: 3.67)

- D57,2,1,1,5,1,1,3,1,1,2,2,13,1,1,1,1,1,6,2,3,2,1,1,2,2,4,1,5,2,3,1,1,1,3... for 1 opponent(s):
  - Bush_Mosteller@_2* (score: 3.59)

- D200 for 113 opponent(s):
  - ϕ (score: 5.0)
  - π (score: 5.0)
  - e (score: 5.0)
  - ALLCorALLD@_0* (score: 1.0)
  - ALLCorALLD@_1* (score: 5.0)
  - ALLCorALLD@_2* (score: 1.0)
  - ALLCorALLD@_3* (score: 5.0)
  - ALLCorALLD@_4* (score: 5.0)
  - ALLCorALLD@_5* (score: 1.0)
  - ALLCorALLD@_6* (score: 1.0)
  - ALLCorALLD@_7* (score: 5.0)
  - ALLCorALLD@_8* (score: 5.0)
  - ALLCorALLD@_9* (score: 5.0)
  - Alternator (score: 3.0)
  - Alternator_Hunter (score: 5.0)
  - AntiCycler (score: 4.6)
  - Anti_Tit_For_Tat (score: 5.0)
  - Better_and_Better@_0* (score: 1.36)
  - Better_and_Better@_1* (score: 1.48)
  - Better_and_Better@_2* (score: 1.52)
  - Better_and_Better@_3* (score: 1.36)
  - Better_and_Better@_4* (score: 1.5)
  - Better_and_Better@_5* (score: 1.4)
  - Better_and_Better@_6* (score: 1.54)
  - Better_and_Better@_7* (score: 1.44)
  - Better_and_Better@_8* (score: 1.32)
  - Better_and_Better@_9* (score: 1.48)
  - Bully (score: 4.98)
  - Colbert (score: 3.06)
  - Cooperator (score: 5.0)
  - Cooperator_Hunter (score: 5.0)
  - Cycle_Hunter (score: 5.0)
  - Cycler_CCCCCD (score: 4.34)
  - Cycler_CCCD (score: 4.0)
  - Cycler_CCCDCD (score: 3.68)
  - Cycler_CCD (score: 3.68)
  - Cycler_DC (score: 3.0)
  - Cycler_DDC (score: 2.32)
  - Defector (score: 1.0)
  - Desperate@_0* (score: 3.0)
  - Desperate@_1* (score: 3.0)
  - Desperate@_2* (score: 3.0)
  - Desperate@_3* (score: 3.0)
  - Desperate@_4* (score: 3.0)
  - Desperate@_5* (score: 3.0)
  - Desperate@_6* (score: 3.0)
  - Desperate@_7* (score: 3.0)
  - Desperate@_8* (score: 3.0)
  - Desperate@_9* (score: 3.0)
  - EasyGo (score: 4.98)
  - Fool_Me_Forever (score: 4.98)
  - Hopeless@_0* (score: 4.98)
  - Hopeless@_1* (score: 5.0)
  - Hopeless@_2* (score: 4.98)
  - Hopeless@_3* (score: 5.0)

- Hopeless@_4* (score: 5.0)
- Hopeless@_5* (score: 4.98)
- Hopeless@_6* (score: 4.98)
- Hopeless@_7* (score: 5.0)
- Hopeless@_8* (score: 5.0)
- Hopeless@_9* (score: 5.0)
- Knowledgeable_Worse_and_Worse@_0* (score: 2.88)
- Knowledgeable_Worse_and_Worse@_1* (score: 3.04)
- Knowledgeable_Worse_and_Worse@_2* (score: 3.04)
- Knowledgeable_Worse_and_Worse@_3* (score: 2.82)
- Knowledgeable_Worse_and_Worse@_4* (score: 3.12)
- Knowledgeable_Worse_and_Worse@_5* (score: 2.74)
- Knowledgeable_Worse_and_Worse@_6* (score: 3.12)
- Knowledgeable_Worse_and_Worse@_7* (score: 3.14)
- Knowledgeable_Worse_and_Worse@_8* (score: 2.94)
- Knowledgeable_Worse_and_Worse@_9* (score: 3.18)
- Math_Constant_Hunter (score: 5.0)
- Negation@_0* (score: 4.98)
- Negation@_1* (score: 5.0)
- Negation@_2* (score: 4.98)
- Negation@_3* (score: 5.0)
- Negation@_4* (score: 5.0)
- Negation@_5* (score: 4.98)
- Negation@_6* (score: 4.98)
- Negation@_7* (score: 5.0)
- Negation@_8* (score: 5.0)
- Negation@_9* (score: 5.0)
- Random@_0* (score: 2.86)
- Random@_1* (score: 3.02)
- Random@_2* (score: 2.96)
- Random@_3* (score: 2.92)
- Random@_4* (score: 3.04)
- Random@_5* (score: 2.82)
- Random@_6* (score: 3.0)
- Random@_7* (score: 3.2)
- Random@_8* (score: 3.14)
- Random@_9* (score: 3.1)
- Random_Hunter (score: 5.0)
- ThueMorse (score: 3.0)
- ThueMorseInverse (score: 3.0)
- Tricky_Cooperator (score: 5.0)
- Willing@_1* (score: 5.0)
- Willing@_3* (score: 5.0)
- Willing@_4* (score: 5.0)
- Willing@_7* (score: 5.0)
- Willing@_8* (score: 5.0)
- Willing@_9* (score: 5.0)
- Worse_and_Worse@_0* (score: 4.6)
- Worse_and_Worse@_1* (score: 4.68)
- Worse_and_Worse@_2* (score: 4.66)
- Worse_and_Worse@_3* (score: 4.58)
- Worse_and_Worse@_4* (score: 4.4)
- Worse_and_Worse@_5* (score: 4.48)
- Worse_and_Worse@_6* (score: 4.52)
- Worse_and_Worse@_7* (score: 4.62)
- Worse_and_Worse@_8* (score: 4.5)
- Worse_and_Worse@_9* (score: 4.6)
- ZD-Mischief@_0* (score: 1.02)

Appendix E

Further Figures

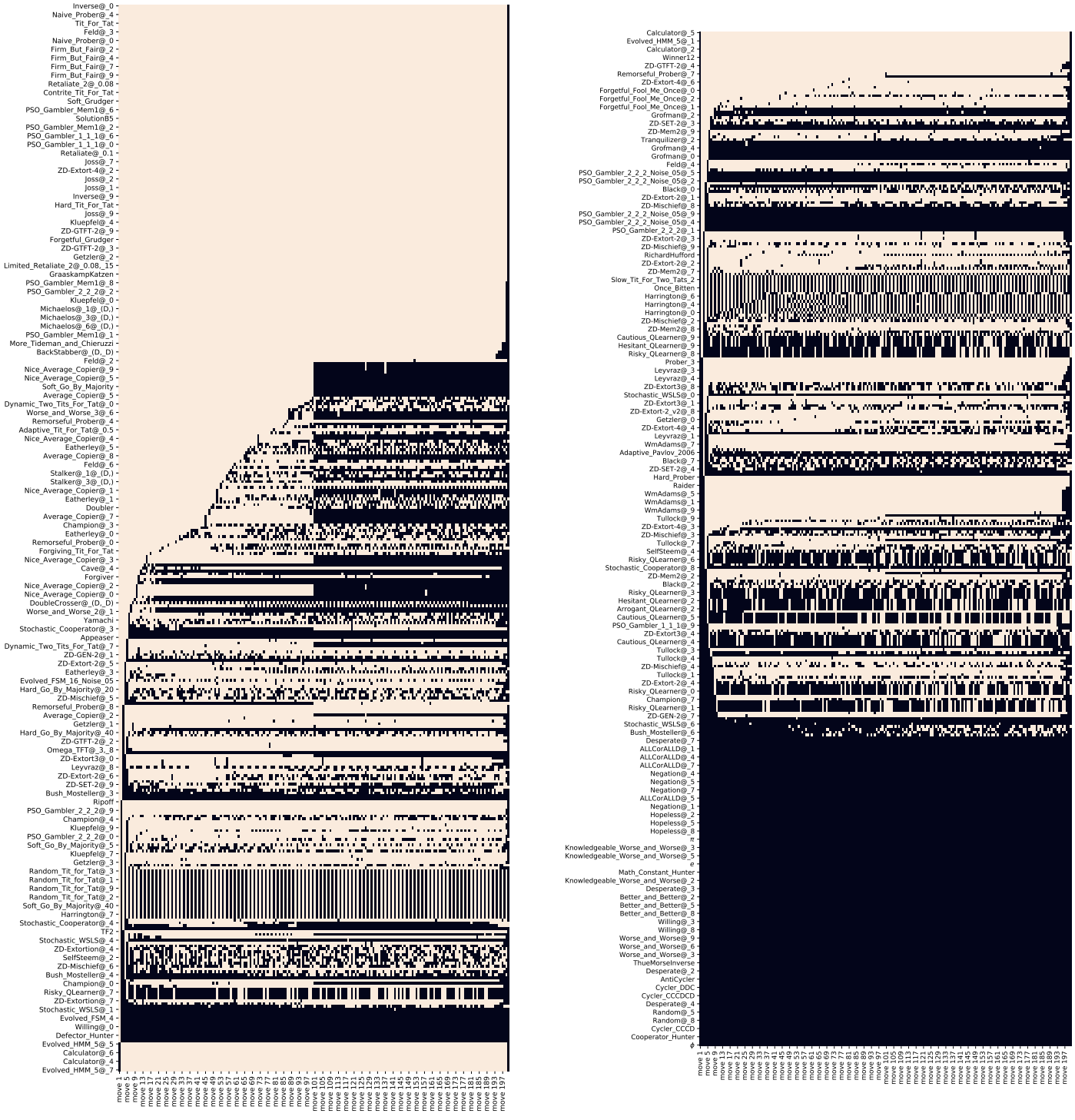

Figure E.1: Sequence Diagram, sorted by move in turns. *note: labels are not complete, approx 1/4 shown.*

List of Figures

1.1 Generic genetic algorithm cycle diagram 6

2.1 High Level Genetic Algorithm Cycle. Extension of Figure 1.1 10

3.1 Finite State diagram of strategy Tit For Tat 13

3.2 Look Up diagram of strategy Tit For Tat 13

4.1 Description and commits for PR on Github 17

4.2 Tail of code commit log as shown on Github 17

4.3 Feature discussions in PR on GitHub 17

4.4 Scope Discussion in PR on GitHub 17

4.5 An Example of a test in the Axelrod Dojo 18

4.6 Code to create a sequence result to optimise best score for 3 opponents 19

4.7 Cells for creating the jupyter instance of a research environment 20

5.1 **Initial Population Size Analysis:** Best score per turn vs generation for different initial population sizes 23

5.2 **Initial Population Size Analysis:** Scatter of mean best score vs different initial populations 24

5.3 **Generation Analysis:** Mean Best Score diff vs total generation lengths 25

5.4 **Generation Analysis:** Max best score vs total number of generations 26

5.5 **Mutation Potency Analysis:** Best score vs generation for different mutation potencies 27

5.6 **Mutation Potency Analysis:** Average best score diff vs mutation potencies 28

5.7 **Mutation Frequency Analysis:** Best score vs generation for different mutation frequencies 29

5.8 Grudger matches against totalities 30

5.9 Old Crossover algorithm 31

5.10 New Crossover algorithm 31

5.11 Example of new crossover algorithm 32

5.12 Best Score vs generations for pre-set initial populations on top of random sequences . . . 33

5.13 Initial Population Code . . . 36

5.14 **New Population:** Non optimal sequence players after changing initial population . . . 37

5.15 **New Population:** Generation Length analysis for best score mean against generation length. . . . 38

5.16 **New Population:** Best score against generation for different mutation potency levels . . . 39

5.17 **New Population:** Best score against generation for different mutation frequencies . . . 40

5.18 A function for wrapping a player with a global seed function call . . . 40

6.1 A histogram showing the distribution of best scores with overlaid KDE . . . 42

6.2 A joint plot of best score vs number of blocks coloured by stochastic boolean . . . 43

6.3 Distance Matrix for Hamming Distance . . . 45

6.4 Distance Matrix for Cosine Distance . . . 45

6.5 Trends for opponents grouped by their best sequence. Dot size represents the number of opponents in the group. . . . 46

6.6 Sequence Diagram, sorted by score; High: top left $\rightarrow$ bottom right: Low. C is light and D is Dark. *NOTE: labels are not complete, approx 1 in 4 shown.* . . . 48

6.7 K means clustering with 2,3 and 4 clusters . . . 49

6.8 A regression tree showing which moves introduce the largest absolute error in the best score. If $X[i] <= 0.5$ is true, it means move i is a Defection move. **TRUE or left** $\Rightarrow D$, **FALSE or right** $\Rightarrow C$ . . . 50

B.1 Code for testing the genetic algorithm in Jupyter notebooks. . . . 55

B.2 AnalysisRun Class for creating structured compute cycles. . . . 56

B.3 Distance Matrix generation code using SciKitLearn. . . . 57

B.4 Regression trees using SciKitLearn. . . . 58

B.5 Code to check multiple populations. . . . 58

B.6 Code to check multiple generation lengths. . . . 59

B.7 The mutation code as given in the axelrod-dojo. . . . 59

B.8 Mutation potency code. . . . 60

B.9 Mutation frequency code. . . . 61

B.10 Tournament code and results . . . 61

E.1 Sequence Diagram, sorted by move in turns. *note: labels are not complete, approx 1/4 shown.* . . . 98

List of Tables

4.1 Functional Python libraries for analysis 18

4.2 Internal built in python modules used 18

5.1 Table of test opponents 21

5.2 Output data table 22

6.1 Raw data from `AnalysisRun.py` output file 41

6.2 Metadata which was merged to Table 6.1 during analysis; joined base on name. 42

6.3 Table of test opponents 47

Bibliography

[1] Robert Axelrod. Effective choice in the prisoner's dilemma. *Journal of conflict resolution*, 24(1):3–25, 1980.

[2] Robert Axelrod. More effective choice in the prisoner's dilemma. *Journal of Conflict Resolution*, 24(3):379–403, 1980.

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

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

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

[6] Jeffrey Horn, Nicholas Nafpliotis, and David E Goldberg. A niched pareto genetic algorithm for multiobjective optimization. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*. Ieee, 1994.

[7] Yahya Rahmat-Samii and Eric Michielssen. Electromagnetic optimization by genetic algorithms. *Microwave Journal*, 42(11):232–232, 1999.

[8] 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.

[9] The Axelrod project developers. Axelrod: v4.0.0, 2017.

[10] Marc, Vince Knight, Martin Jones, T.J. Gaffney, Toby Devlin, Nikoleta, and Georgios Koutsovoulos. Axelrod-python/axelrod-dojo: v0.0.8, March 2018.

[11] James Cambell. Thesis. 2016.

[12] Saul I Gass and Arjang A Assad. *An annotated timeline of operations research: An informal history*, volume 75. Springer Science & Business Media, 2005.

[13] John Tooby and Leda Cosmides. The evolution of war and its cognitive foundations. *Institute for evolutionary studies technical report*, 88(1):1–15, 1988.

[14] Robert J Aumann and Sergiu Hart. *Handbook of game theory with economic applications*, volume 2. Elsevier, 1992.

[15] Daniel M. Cable and Scott Shane. A prisoner's dilemma approach to entrepreneur-venture capitalist relationships. *Academy of Management Review*, 22(1):142 – 176, 1997.

[16] Duncan Snidal. The game theory of international politics. *World Politics*, 38(1):25–57, 1985.

[17] Bobbi S Low. *Why sex matters: A Darwinian look at human behavior*. Princeton University Press, 2015.

[18] William H Press and Freeman J Dyson. Iterated prisoners dilemma contains strategies that dominate any evolutionary opponent. *Proceedings of the National Academy of Sciences*, 109(26), 2012.

[19] Shashi Mittal and Kalyanmoy Deb. Optimal strategies of the iterated prisoner's dilemma problem for multiple conflicting objectives. *IEEE Transactions on Evolutionary Computation*, 13(3):554–565, 2009.

[20] Julie Rehmyer. Game theory suggests current climate negotiations wont avert catastrophe. *Science News*, 2012.

[21] Thomas Osang and Arundhati Nandy. Environmental regulation of polluting firms: Porter's hypothesis revisited. *Revista Brasileira de Economia de Empresas*, 3(3), 2013.

[22] Bruce Schneier. Lance armstrong and the prisoners' dilemma of doping in professional sports. *WIRED*, 2012.

[23] Jonathan Goldman and Ariel D Procaccia. Spliddit: Unleashing fair division algorithms. *ACM SIGecom Exchanges*, 13(2):41–46, 2015.

[24] spliddit.org. http://www.spliddit.org.

[25] Vincent Knight, Marc Harper, Nikoleta E Glynatsi, and Owen Campbell. Evolution reinforces cooperation with the emergence of self-recognition mechanisms: an empirical study of the moran process for the iterated prisoner's dilemma. *arXiv preprint arXiv:1707.06920*, 2017.

[26] Marc Harper, Vincent Knight, Martin Jones, Georgios Koutsovoulos, Nikoleta E Glynatsi, and Owen Campbell. Reinforcement learning produces dominant strategies for the iterated prisoners dilemma. *PloS one*, 12(12):e0188046, 2017.

[27] Daniel Ashlock, Eun Youn Kim, and Warren Kurt. Finite rationality and interpersonal complexity in repeated games. 56(2):397–410, 2004.

[28] Daniel Ashlock and Eon Youn Kim. Fingerprinting: Visualization and automatic analysis of prisoner's dilemma strategies. *IEEE Transactions on Evolutionary Computation*, 12(5):647–659, 2008.

[29] Jiawei Li and Graham Kendall. A strategy with novel evolutionary features for the iterated prisoner's dilemma. *Evolutionary computation*, 17(2):257–274, 2009.

[30] Sander Bakkes, Pieter Spronck, and Jaap Van den Herik. Rapid and reliable adaptation of video game ai. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(2):93–104, 2009.

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

[32] Randy L Haupt and Sue Ellen Haupt. Practical genetic algorithms.

[33] Jason D Lohn, Derek S Linden, Gregory S Hornby, and William F Kraus. Evolutionary design of an x-band antenna for nasa's space technology 5 mission. In *Antennas and Propagation Society International Symposium, 2004. IEEE*, volume 3, pages 2313–2316. IEEE, 2004.

[34] Dragan A Savic and Godfrey A Walters. Genetic algorithms for least-cost design of water distribution networks. *Journal of water resources planning and management*, 123(2):67–77, 1997.

[35] Linus Torvald. Git.

[36] Neil P Chue Hong, Tom Crick, Ian P Gent, Lars Kotthoff, and Kenji Takeda. Top tips to make your research irreproducible. *arXiv preprint arXiv:1504.00062*, 2015.

[37] Andreas Prlić and James B Procter. Ten simple rules for the open development of scientific software. *PLoS Computational Biology*, 8(12):e1002802, 2012.

[38] Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. Ten simple rules for reproducible computational research. *PLoS computational biology*, 9(10):e1003285, 2013.

[39] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90–95, 2007.

[40] Multiple. Pandas.

[41] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[42] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[43] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[44] John W Tukey. *Exploratory data analysis*, volume 2. Reading, Mass., 1977.

[45] Mohammad Norouzi, David J Fleet, and Ruslan R Salakhutdinov. Hamming distance metric learning. In *Advances in neural information processing systems*, pages 1061–1069, 2012.

[46] Mr Bora, Dibya Jyoti, Dr Gupta, and Anil Kumar. Effect of different distance measures on the performance of k-means algorithm: an experimental study in matlab. *arXiv preprint arXiv:1405.7471*, 2014.

[47] Seung-Seok Choi, Sung-Hyuk Cha, and Charles C Tappert. A survey of binary similarity and distance measures.

[48] Leo Breiman. *Classification and regression trees*. Routledge, 2017.

[49] Darrell Whitley and Andrew M Sutton. Genetic algorithmsa survey of models and methods. In *Handbook of natural computing*, pages 628–687. Springer, 2012.