Full Unit - Final Report

Benjamin Stephenson

Royal Holloway University of London, United Kingdom, 2021

Benjamin.Stephenson.2018@rhul.ac.uk

Supervisor: Argyrios Deligkas

Abstract

The Iterated Prisoner's Dilemma game has been a widely used tool, primarily used for modelling complex interactions within groups and discussing strategies and contemplating the existence of a single best stable strategy.

This paper presents my implementation of this problem, discussing not just how the strategies affect each other, but how the tournament environment itself affects the performance of the strategies. A further environment for developing strategies that can evolve and change their behaviour will be introduced, as well as giving strategies the ability to collude with each other to create a group advantage. I will also discuss past tournaments and the strategies that participated in them. Words: 13945

Contents

1 Introduction

The Iterated Prisoner's Dilemma is an extension of the original thought experiment; the Prisoner's Dilemma. In the original game, two prisoners are accused of a serious crime and arrested with no opportunity to converse with each other prior to their arrest. There is not enough evidence for either to be convicted of the serious crime, but there is enough to be convicted of a lesser crime. Each prisoner is then given the choice to implicate the other for the serious crime (defect), if they do this, they will get a reduced sentence. Or they could choose to stay silent (cooperate) and only be convicted of the lesser crime. Cooperate and defect are often referred to as C and D respectfully.

The payoff matrix used to calculate scores in a match is defined as follows, where each strategy is trying to maximise their points.

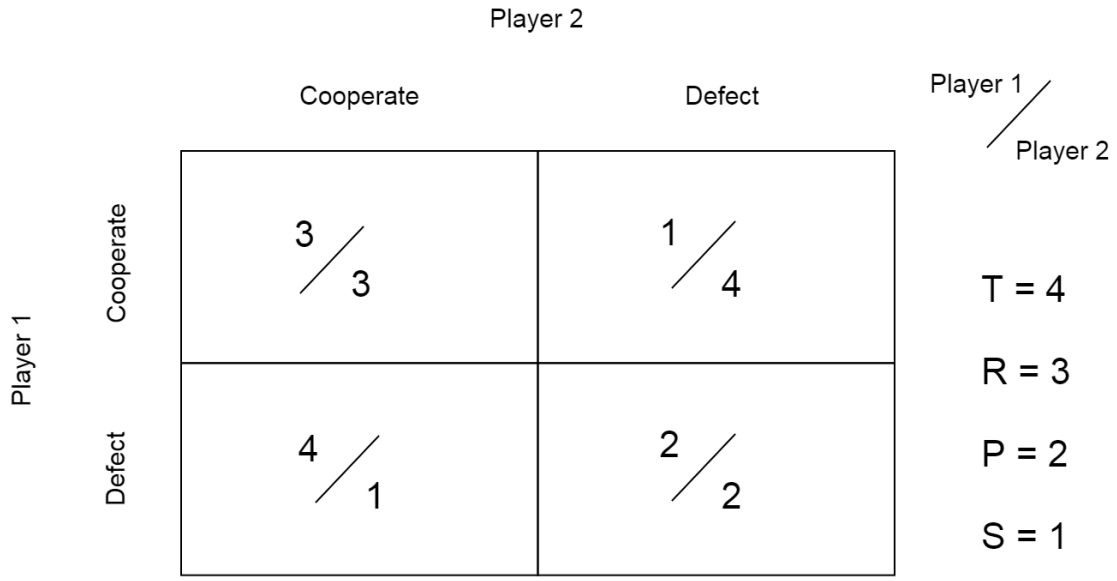

Figure 1: Payoff Matrix

The numbers are place holders for the values, T - the temptation, S - suckers payoff, R - reward for cooperation and P - the punishment. As shown in figure 1. The game is defined by $T > R > P > S$ and $R < (S + T)/2$. So long as these two statements are true, the game remains in a Nash equilibrium, Wikipedia contributors [2020b], and continuous cooperation is better than alternating defection and cooperation. This is why many variations of the game exist but ultimately remain the same.

The worst possible outcome for you is that you remain silent and your accomplice betrays you, in which case you are convicted of the serious crime and serve the maximum sentence with no reductions. The next worst, is that you both betray each other, in which case you are both

convicted for the serious crime and serve long sentences, however both are reduced due to your willingness to divulge the information. The second best scenario, is where you both remain silent and only get convicted of the lesser crime and both serve short jail time *(This outcome has the highest mutual benefit)*. The best option, for purely selfish gain, is that you betray your accomplice, where you get a reduced sentencing of the lesser crime, but they are faced with the worst outcome.

In this scenario, even though the best global outcome for both parties is mutual cooperation. The only Nash equilibrium for two self-interested parties is to defect, as regardless of the other's choice, this will minimise your sentence. However, the Iterated Prisoner's Dilemma introduces reciprocity, which gives an opponent an opportunity to retaliate in future rounds. The goal in this version is to minimise total years spent in prison *(or in the case of my implementation, maximise the payoff)*.

My goals for this project are to explore the implementations of differing strategies as well as the implementation of the tournament system, creating a Round-Robin style tournament as well as a Single-Elimination tournament, to illustrate how a differing tournament structure creates differing results; in the real world this is often the case, as indicated in 2.1.

I also aim to create an environment for genetic algorithms to evolve, to observe whether their is an emergent stable strategy that is reproducible upon duplicating the same parameters. Further aims are to create colluding strategies that work in a team, to artificially boost one of their members as well as using genetic algorithms to tailor bespoke, optimal algorithms for a specific environment of already known strategies. The results of these tests need to be displayable to a user, and the user should be able to specify parameters.

2 History

The first competition was organised by Axelrod in 1979, based on strategies by experts in game theory from the fields of economics, sociology, political science and mathematics.

The contest had mean rounds of 200, with the payoff matrix $S = 0, P = 1, R = 3, T = 5$, as stated in Jurišić et al. [2012]. In this competition one of the simplest strategies won the whole competition. This strategy was tit-for-tat(TFT) sent by Anatol Rapport, this strategy is explained in my report on basic strategies.

What was shown in this tournament was that strategies that achieved good results had a tendency to cooperate and were 'nice'.

Observations of the first competition were:

1. TFT won even being as simple as it is.
2. Strategies that tried to improve on TFT had worse results as they tried to defect occasionally, which resulted in chains of mutual defections. They were 'too clever'.
3. The biggest factor for success was niceness.
4. Strategies that implemented forgiveness performed better.
5. 'kingmaker'(GRAASKAMP, DOWNING) strategies and the environment can massively affect results.

Finally, better strategies than TFT exist, such as those created through genetic algorithms, which was orininally outlined by Axelrod himself in the journal *the Dynamic of Norms* Axelrod et al. [1987]. More recently colluding strategies as shown in Rogers et al. [2007], where strategies cooperate in an additional layer, to boost one team member by purposely losing to them. This presents an interesting example of why the Iterated Prisoner's Dilemma is formatted as a game involving purely self-interested agents, however often in the real world team work like this can result in a greater pay off for the team.

2.1 Importance and Real-life examples

The Prisoner's Dilemma illustrates why sometimes two completely rational individuals may not cooperate, even if it seems to be in their best interest to do so, as mentioned in Wikipedia contributors [2020a]. In contrast to this the Iterated Prisoners Dilemma illustrates why when the most rational decision is to defect in an isolated round, when they are linked, it is more rational to create mutual cooperation.

Interestingly, many natural processes can be abstracted into models where living beings are engaged in endless games of prisoner's dilemma. For example:

In the environment, it is inarguable that all countries would benefit from a stable climate. However, in any country's self-interest of endless economic growth, the choice to maintain current behaviour is often wrongly perceived to be of a larger benefit than the eventual benefit to *all* countries if they changed behaviour. This was reported in The Economist [2007].

It has also been observed in Wikipedia contributors [2020a], that the social structure of guppies can be modeled as iterated prisoner's dilemma, as they inspect predators in groups and when a member doesn't cooperate they are punished.

It is also researched in the field of psychological addiction where the addiction can be abstracted to a prisoner's dilemma of the patient and their future selves, which is explained in further detail in Monterosso and Ainslie [2007].

3 Planning and Timeline

The broad goals for this project are as follows:

1. To create a tournament environment for the strategies that can host multiple strategies.
2. To order the strategies at the end of the tournament based on performance to some win condition:
    - A Round-Robin where points are assigned.
    - A Single-Elimination tournament.
3. To create strategies from varying methodologies:
    - Deterministic strategies.
    - Adaptive Strategies.
    - Colluding Strategies.
    - Strategies cultivated through genetic algorithms
4. To create a simple and intuitive graphical user interface to represent the whole system.

The original plan and the timeline of events is in the Appendix A.3.

3.1 Risks and Mitigations

The potential risks as to why the project may fail to proceed and what should be done to mitigate them:

- Risk of scope creep, as with wanting to have a wide variety of strategies all requiring individual research, there is a possibility that the project may get enlarged to something I cannot deliver in full in the time given. To avoid that, I have tried to 'front load' the important tasks, such as creating the tournament, basic strategies and having a somewhat completed project by the end of the first term. This is so that the second term can be used for extensible goals such as colluding strategies and genetic algorithms.
- Risk of badly designed program structure is extremely important. If the structure for the tournament is not made general enough, my plans to have varying strategies all with wildly differing implementations will ultimately fail as they will not be easily added and removed from the tournament without affecting the tournament. I have tried to minimise this by adding extensive planning for the project structure at the beginning of this term and beginning implementation early to emphasise iteration and improvement, TDD will be used.

- Risk of not fully understanding the theory behind a strategy, resulting in either a faulty or no implementation of it. To mitigate that I have tried to emphasise the writing of a report before implementation to ensure I do adequate research and ensuring that practical implementations are included in that research and not just theoretical.
- Arguably most likely risk is specification and plan break down. For example getting stuck on implementation of genetic algorithms for multiple weeks and not creating a deliverable project. To prevent this I have created the timeline and have created buffer weeks at the end of term to ensure that if I ever lose track there are a few extra weeks to minimise the impact on the final deliverable.

4 Strategies

This section discusses the varying strategies that I researched and implemented. Stating the advantages and disadvantages of each and how you can conclude almost any one is superior than the other; this is reliant on the environment though, which we will come to see has an extremely profound affect on the outcome of a tournament.

What I have found through testing differing strategies in a tournament setting concludes that you can manipulate the winner if you decide who the participants be. For example one ALLD against any amount of ALLC, both of which are explained below, the ALLD will always win. It is also likely that if you add TFT to this environment, which is seen as a 'better' overall strategy, ALLD will still win as it can take advantage of the ALLC strategies. Both strategies are explained below.

4.1 'Always' Strategies

This section discusses the two strategies always cooperate - ALLC and always defect ALLD.

4.1.1 Always Defect

ALLD is the most 'rational' algorithm in an isolated setting where there is no fear of retaliation, as the best outcome for the individual regardless of the opponents choice would be to defect. Therefore, if we assume that the iterated version is nothing more than playing multiple instances of the single prisoner's dilemma game, this would be the optimal choice. Furthermore, As mentioned above in 1 if the amount of rounds in a game is known, this will always be the only rational strategy.

However, the ALLD strategy fails to predict the behaviour of human players and in an environment of superrational strategies which are aware that other superrational strategies know that to defect is the dominant solution, where the number of rounds is large enough and unknown, it is more rational to cooperate, as mutual cooperation creates a larger total than mutual defection. This is discussed in the context of a different game in He et al. [2015].

4.1.2 Always Cooperate

Therefore, leading on from ALLD, if we assume an environment of superrational strategies where the number of rounds is unknown, the most 'optimal' strategy would be to always cooperate.

However, this is dependant on the strategy population, in an unknown environment we cannot assume all actors to be superrational. For example in an environment where you are aware of superrational strategies that will try to always cooperate, you will gain a massive advantage by always defecting, thus always gaining the temptation bonus. Therefore, even though this is the

optimal strategy for a nice environment, most environments are not nice and this strategy will almost always perform less well than ALLD.

4.2 Tit-for-tat and its Variations

The most dominant 'simple' strategy which won the first ever Iterated Prisoner's Dilemma tournament was for tit-for-tat. The premise is simple: TFT will start by cooperating, then do whatever its opponent did the round before it. As shown in Jurišić et al. [2012].

This strategy is 'nice' as it favours cooperation and will never be the first to defect, attempting to set up an endless string of mutual cooperation. It will also not try to score more than its opponent, as in a tournament environment the score of an individual match does not matter so much as the cumulative total amongst many games.

However, it is not a blind optimist such as ALLC , it will not continue to cooperate with an opponent who defects. If the opponent defects, TFT will retaliate by defecting the next round as a punishment. This can result in an endless chain of CD and DC outcomes, for instance, if two TFT strategies played against each other, where one starts with defection.

TFT is also forgiving, if the opponent attempts to re-initiate cooperation after defection, TFT will comply.

4.3 Pavlov

Pavlov is one of the more famous strategies and can often outperform TFT. Also known as win-stay, lose-shift, which describes its method well. It works by categorising a win or a loss for the previous round and then adjusting its strategy based on the previous outcome. As shown in figure 1. If it receives R or T points, this is considered a win and it repeats the same move as before. If it receives P or S points, it changes its strategy. This is also outlined in Jurišić et al. [2012].

The main weakness of Pavlov is that against a constant defector it will try to cooperate every second move, however adapted versions exist to overcome this, such as APavlov in Li et al. [2011]. Another version will be discussed later, though I have called it $StatisticalAdapt$ to avoid confusion with Pavlov.

4.4 p-value

A p-value is a random strategy with a degree of favouritism towards a particular action. A true 50/50 random strategy would have p-value of 0.5. This means you can create a somewhat unique strategy by altering the p-value.

This is important as in testing environments the more strategies the more general the data,

(though this depends on which strategies too). This type of strategy allows for the extra room in tournaments to be padded out by, albeit random, differing strategies. Which was suggested by Deligkas [2020]. Furthermore, in the single elimination tournament type when ALLD is not present, the p-value strategy with the highest defection rate will be the most dominant. As defection is a dominant strategy and ensures that you can only get as bad as your opponent, which means you can never lose a given match.

4.5 Adaptive Strategy

An adaptive algorithm is an algorithm that changes its behaviour over the time that it is run. This change is based on information it collects throughout its run time. In the context of the Iterated Prisoner's Dilemma, the information used will be the previous actions that the strategy and the opponent took.

As discussed in the *pavlov* section in 4.3. We can create an algorithm that adjusts it's play based on the outcome of a previous round, with some condition. Now, in the example Pavlov, it only looks at the last turn and checks if it was a CC or DD, which means it will cooperate next turn. A CD or DC will mean it defects next turn.

We can expand this to not just look at the last round, but all the rounds of the entire match that have been played so far. In addition we can introduce probability, meaning that the *chance* of cooperating or defecting changes. But remains that, a chance.

With this framework we can also define how the history affects the future choice. For example, an instance of an *adapt strategy*, as explained in Kraines and Kraines [1993], would start with some n and some k, where k/n is the probability that the strategy will cooperate on the next turn. Then for each round in the history we review the outcome, for each CC and DD, we increase the probability by some p/n, conventionally $p = 1$. For each CD and DC we decrease the probability that the strategy will cooperate, therefore increasing the probability of defection by some x/n, conventionally $x = 2$. This convention is set this way to make retaliation come quicker, slow to gain trust but quick to lose it. Changing any of these values k, n, p, x will change how the algorithm behaves.

However, other adaptive strategies exist, for example *ScoreAdapt* which Deligkas [2020] suggested; adjusting the strategy based on how the strategy and its opponent are scoring. In this case, if our strategy is ahead or equal it will cooperate, if it is behind it will defect.

4.6 Colluding Strategies

As shown first by Rogers et al. [2007], a somewhat 'cheating' solution exists. The traditional prisoner's dilemma is formulated to involve purely self-interested agents. This statement seems simple, yet in the real world it is not as such. Communication can take place on different levels and meta-communication exists where you make decisions from an extra dimensional perspective. For example, from a single dimension, a football team playing another football team wants to

win. However, if they look at how they can abuse that expectation, in the instance of betting for example, they can purposefully lose if they are expected to lose to increase benefit massively for those who bet against them and they can have a cut of those profits. Thus even though they lost the game, because they are colluding with another, they win more than they would have if they had actually won.

In the iterated prisoners dilemma it is not beneficial to do this in a total score sense, since the temptation plus the suckers payoff is lower than the two of the rewards. However, if like in the real world, winning the whole competition was the goal there is benefit.

Thus, colluding strategies do not operate for individual benefit or for team benefit, but to have one member that is the best. This also means that unlike other strategies, their strategy is as a team and not as an individual. They achieve this by recognising each other through some mechanism, one of the members will then purposefully lose, so that the other team member achieves the maximum score. The strategy that is being boosted as part of the team will play to maximise their own payoff, thus use a TFT strategy when not playing against a team mate. Other team mates will play to minimise all oppositions' score, and maximise each others' score in the team, especially the leader.

5 Tournaments

This section discusses the advantages and disadvantages of different tournament styles in the context of the Iterated Prisoners Dilemma discussed in previous sections. The tournament needs to be fair and robust, though this definition differs for each tournament type outlined below:

5.1 Single-Elimination

A Single-Elimination tournament is the classic tree style tournament. Where strategies are paired, the winners progress and the losers are eliminated. This is a well known, quick and efficient style, having $n - 1$ matches, where n is the amount of strategies. Whereas Round-Robin has n^2 matches.

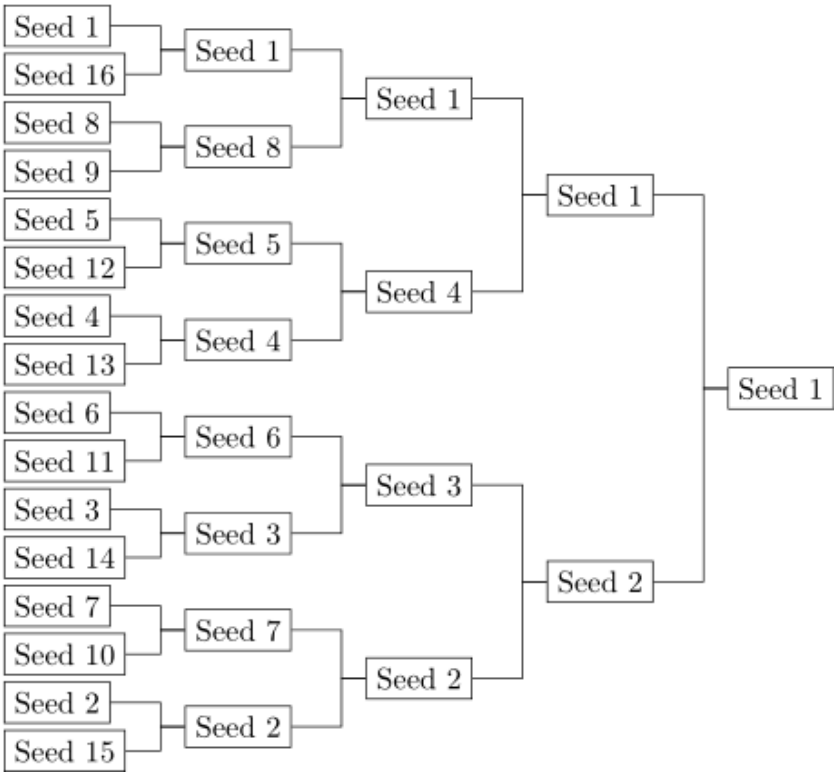

Figure 2: Single Elimination Tournament Structure, from wordpress.com

The main advantage is that the amount of rounds can be random for each match as scores are not cumulative and have no carry over. Therefore total score is independent from the strategy's ranking.

The disadvantages are plentiful however, especially in the context of the Iterated Prisoner's Dilemma. As is the nature with differing strategies, most have strategies they perform well against and those they perform poorly with. In this case if an overall more well rounded strategy, such as TFT, that performs better in many cases than ALLD, always defect. TFT will always lose, as it starts with a cooperate. Thus the overall strengths in a multi-strategy environment are not

expressed in isolated matches.

This problem then expands the problem where seeding would have to be changed to make the tournament fair, but this would assume then that we know how strategies will perform in the environment and can thus place them in a seed where they are more likely to get their intended position. However, in the context of IPD this is akin to match fixing and would make it unfair, not to mention almost impossible to implement when given a set of unknown strategies. This is discussed in depth in Kim et al. [2017]. Furthermore, as defection is a dominant strategy in a single match, any defection dominant strategy will always win. Though this is interesting to see as how differing structures of tournament can affect what we view as the 'best' strategy.

5.2 Round-Robin

A round-robin tournament is where every player plays each other and which ever performs best overall is declared the winner. Round-robin presents some great advantages, mainly, a ranking is produced, not just a winner, so performance of all strategies can be observed. This also gives the chance for more well rounded strategies such as TFT to shine. Thus, can be considered more fair, as every strategy must play all the same strategies as every other one. This creates an environment which is more interesting to observe and strategies can be discussed more generally and not just in the context of isolated games.

A Round-Robin tournament is just a type of directed graph. Although being quite restricted structurally, they are realised in a great many natural phenomena, such as the "pecking structure" of birds and other mammals. Which can represented structurally as a tournament. This article Harary and Moser [1966] discusses the round-robin tournament in a broad context, drawing comparisons to the natural world.

Although in our environment, a Round-Robin tournament is quite easily implemented. Not requiring scheduling such as the circle method explained in Suksompong [2016], which I originally researched, but discovered was redundant as my tournament is synchronous; any two strategies can play each other at any time. A problem presented itself in regards to the implementation however, which will be discussed in section 8.3.

A figure showing the layout of this tournament can also be seen in 8.3.

6 Genetic Algorithms

A genetic algorithm is a way of developing algorithms using natural selection, often used to find good, ideally optimal, solutions to the problem the algorithm will solve. Genetic Algorithms in the context of the Iterated Prisoner's Dilemma, originally proposed in Axelrod et al. [1987], display an interesting dynamic, where they can present the way strategies develop organically. In other scenarios we impose our strategies and cannot see how strategies would develop as they would in the real world. The formulation of a strategy into a genetic one requires two things: a genetic representation of the solution domain and a fitness function. For the genetic representation or Dna, we can use the history of the game so far. Fitness will be assessed based on the individuals performance in an IPD tournament.

The way the history is formulated is as pairs of what the players did for each round in order. For example, a history where I cooperate first round and you defect, then the next round we both defect, this can be written as $(C, D), (D, D))$ or $CDDD$ in this case. This is why a history of 10 rounds can have 2^{20} possible histories, as each round has two contributors, what I played and what you played.

To formulate the Dna, we cannot create a response for every possible history in our environment. Firstly, because it would be infinite; the nature of the IPD is that the length of the match is undefined and potentially infinite. Secondly, the amount of possible formulations of a history of length n is 2^{2n}. Therefore, even if we knew the how many rounds were in the match, the length of the string needed for all possible histories can be formulated below where the length of the string is equal to X.

$$X = \sum_{r=0}^{n} 2^{2r}$$

Which progresses as such, $n = 3$, $X = 85$, $n = 5$, $X = 1365$, $n = 10$, $X = 1398101$, it's exponential and therefore impractical to store strings of that length for any large amount of rounds. So we need to impose a maximum, where the history is equal to the last n rounds where n is small, but greater than 0.

This offers the opportunity for us to make it customisable. Using the method in Bukhari [2005] as a base, consider: A list of lists, A and $A(i)$, respectively. Where each index, i, of A is how large the history is. Therefore, the list $A(i)$ corresponds to all the possible histories of length i. This is because a history of CC and $CCCCCC$ would be considered the same if converted to binary. Converting the history to binary would offer us the ability of instant accessibility to the corresponding response, as we would not have to search, similarly to a hashmap where the hash value is the binary representation of the history. This is better than what is normally used in my opinion, which is to have static responses for the first few rounds before the history reaches a size where the Dna takes effect. What this does create however, is 'redundant' pieces of Dna. Since the Dna includes the choices of the individual it belongs to, sometimes the individual's decision will prevent other parts from being used. For example, if the first move for the individual is to cooperate, then for the next round, the Dna histories where the individual defected will never be used. This needs to remain this way though, as it may evolve back into relevancy later.

Even with this solution we would not be able to store strings of excessive lengths and therefore once the number of rounds exceeds the maximum supported history length, it will simply take the longest supported history it can.

An alternative way of formulating the DNA would be to use the four possible histories for the previous turn: CC, CD, DC and DD. Each one of these would have an associated probability to what the response should be in this situation. This method seems rather limited, and is more akin to a random strategy and though it may display similar behaviour it will be basing decisions only on the last round, and is therefore unable to react to longer patterns.

As mentioned above, crossover and mutation will be used to breed strategies. Crossover is a function that takes two Dna strings and produces a new one using sections of each parent. Different implementations exist, such as using blocks from each parent or uniformly taking each gene randomly from each parent. Mutation will alter a gene from what was inherited, it has a low percentage, but is necessary to create variation to produce new strategies.

6.1 Natural

There are two options for training a set of genetic algorithms. Firstly, we could have an environment of only genetic algorithms where they all compete from a random starting population. With this we can observe if there is a converging outcome across multiple tests or if the randomness of the starting population will mean it diverges, sometimes getting more cooperative based strategies and sometimes getting defecting.

We would have a list of some candidates, they all participate in the RR tournament. The top half are selected and paired off in some way. Each pair is then bred together using some form of crossover and mutation to produce four offspring. The four offspring of each pair will create the next population. This will be repeated for a defined number of evolutions.

I call this method 'natural' as there is no involvement from human defined strategies, this would also be a better representation of what happens in nature, where there are no static individuals and all individuals evolve relative to each other.

6.2 Guided

A second option, is to have environment of already established strategies, then each member of the set of genetic algorithms plays a tournament with themselves and only the other established strategies. In this way we can try evolve a strategy for a given population of established strategies. So the fitness of an individual is not determined by its efficiency against its peers, but by its performance against a defined set of unchanging strategies.

This method is akin to selective breeding where we breed a genetic algorithm for a specific purpose thus it's named 'guided'. This would be helpful in trying to determine optimal strategies to a given environment.

7 Implementation of Strategies

The strategies will all be implemented into the following environment:

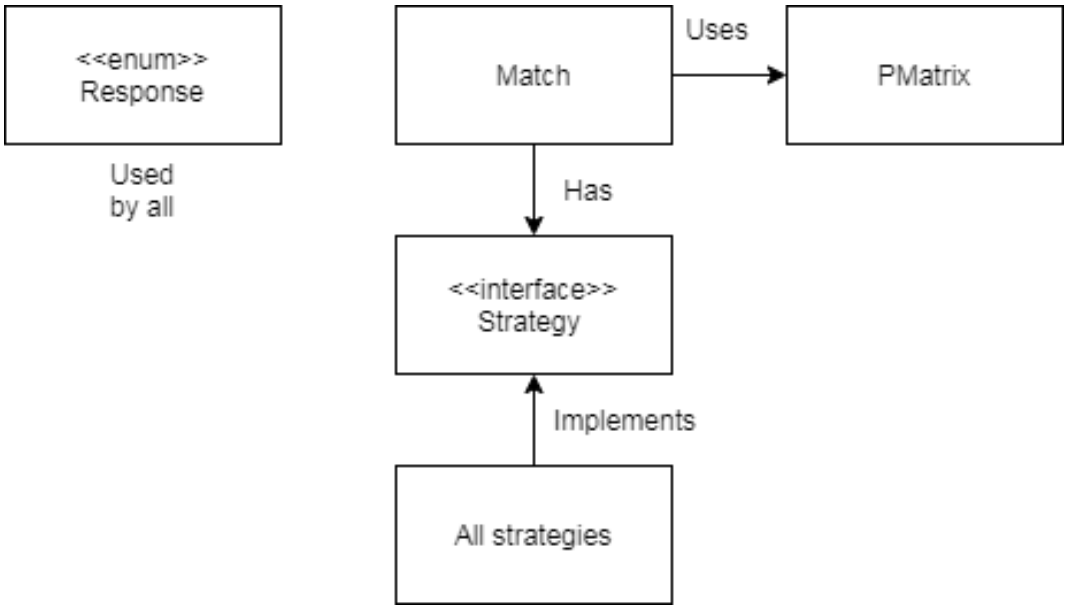

Figure 3: Strategy Class Diagram

The responses are implemented as an *enum* as they are constants and will be simpler to use and more readable this way as opposed to a String, integer or boolean.

This implementation of *Match* allows for any general algorithm that implements the *Strategy* interface to be used as a strategy. This will be important in the future when there will be multiple strategies with differing implementations. The *Strategy* class is implemented as following:

```
package Strategies;

public interface Strategy {
    // interface used as all strategies will have same input and output.

    Response play(int matchCount, Response[][] history, int stratNo);

    String getName();
}
```

Though other strategies will contain other methods, as far as the GUI or any class that deals with simple *strategies*; all they need to have is a *play* function which will return a response given a certain history of a match and a function that returns the name for the GUI.

The payoff matrix is implemented as a singleton as there would be no reason to have multiple. Update was added as a method so that the payoff matrix can be changed for separate matches to observe how a differing payoff matrix may affect the outcome.

7.1 Factory Class

The strategies, tournaments and genetic algorithms are instantiated using a factory class, this is to decouple the code from the GUI controller, so the GUI only has to deal with Strings, this was done using the following method:

```
public class StratFactory {

    //Factory class used to create strategies for the#
    //GUI to decouple the strategies. Also easier to implement.

    public Strategy getStrategy(String stratType){
        if(stratType == null) {
            return null;
        }

        if(stratType.equalsIgnoreCase("TFT")){
            return new Tft();

        } else if(stratType.equalsIgnoreCase("RANDOM")) {
            return new Random();

        } else if(stratType.equalsIgnoreCase("PVALUE")) {
            return new PValue();

            etc...
```

The *getStrategy* method returns the strategy object based on the string input.

7.2 Implementation of *p*-value

The *p*-Value strategies were implemented by creating a random 'PValue' upon instantiation, which will differ across every *p*-Value strategy. Each round that the *p*-Value plays will create a new random number, if it is above the *p*-Value the strategy will cooperate and defect if below. Although this is ultimately a random strategy, and relative to other strategies performs as such. With the best opposition being ALLD.

```
    private float pval = new Random().nextFloat();

    public Response play(int matchCount, Response[][] history, int stratNo) {
        return calcPval();
    }

    private Response calcPval() {
        double random_double = Math.random();
        //randomly chooses one option
        return random_double < pval ? Response.C:Response.D;
    }
```

7.3 Implementation of Statistical Adapt

For the implementation of the *StatisticalAdapt*, outlined in subsection 4.5, $k = 5, n = 10, p = 1, x = 2$. This means the first round has a 50/50 chance of cooperating or defecting.

Then for the next rounds the probability is calculated using the following method:

```
//The probability to cooperate increases by 1 if CC or DD,
//decreases by 2 if CD or DC.
private int getProb(Response[][] history, int stratNo, int matchCount) {
    int change = 0;

    for (int i = 0; i < matchCount; i++) {
        if (increase(history[stratNo][i], history[1 - stratNo][i])) {
            change += cooperateChange;
        } else {
            change -= defectChange;
        }
    }

    //ensures that probability will be between 0 and 1.
    if (k + change > n) {
        return 10;
    } else if (k + change < 0) {
        return 0;
    } else {
        return (k + change);
    }
}
```

The *change* variable is initialised to 0, then as you can see there is a for loop which goes through every outcome in the history of the current match, using the criteria in the method *increase*(), it will return true if the outcome of that round should increase the probability of cooperation, false if it should decrease.

You may think it would be more suitable to only look at the current match and having a state variable which holds the current probability and then update it with one line each round. However, I do not want to introduce variables to strategies. Only parameters of the *play*() method and constants created at instantiation of the strategy such as in the case of the p-Value strategies.

The *if* statement is to 'cap' the change, so it does not go over the bounds of 0 and n, which here is 10. As this would mean negative results or exceedingly greater values, which would make future rounds not affect the the probability outcome.

The *increase*() method as mentioned above is defined as follows:

```
//defining when to increase, when not increase, decrease.
private boolean increase(Response me, Response you) {
    if ((me == Response.C) && (you == Response.C)) {
        return true;
    } else if ((me == Response.D) && (you == Response.D)) {
        return true;
    }
    return false;
}
```

A set of responses that would mean an increase of cooperation are *CC* and *DD* (which is the exact same as the pavlov strategy). This is because with a *CC* outcome the strategy would want to further reinforce the trend of cooperation. With *DD*, this is where these pavlovian strategies differ from strategies such as tit-for-tat, as they try to re-establish cooperation and avoid chains of mutual defection. Therefore, this strategy and pavlov perform much better in IPD environments with noise, which is also mentioned in many articles including Kraines and Kraines [1993].

7.4 Implementation of Colluding Strategies

Colluding strategies are challenging as they require some form of communication, in examples such as Rogers et al. [2007], they used sophisticated methods to dynamically recognise each other without previous communication, which was way above my scope. So I compromised on a communicated code. I also created 'king' strategies and 'pawn' strategies, the pawns would play to minimize their opponents that weren't kings using AllD and maximize kings by purposefully losing. The king strategies exploit pawns and play to maximize their own score when playing other strategies by using tit-for-tat. An alternative is for them to dynamically communicate in a match which one has a higher score and which one is most likely to achieve the highest score, though this too is above my scope.

I created a *Colluder* class which was used to coordinate kings and pawns. It did this by creating a 'code' for each, which is a list of responses that each would perform at the beginning of a match to tell the other that they were a team mate and which type of team mate. Implemented as a singleton so only one is created, to prevent multiple codes from being created.

```
private Colluder() {
    pawnCode = getCode();
    kingCode = getCode();
}

 private Response[] getCode () {
     Response[] code = new Response[codelength];
     java.util.Random random = new java.util.Random();

     for (int i = 0; i < codelength; i++) {
         code[i] = random.nextBoolean() ? Response.C:Response.D;
     }

     return code;
}
```

As shown above, the codes are created when the object if first instantiated, since it is a singleton, this is only done once. It is quite possible for these to be equal, but it has never happened yet. It is also possible for a random strategy to randomly create the code and be misinterpreted as a team member, though this will always be a risk. It may have been more optimal to decide on codes that are designed in such a way that they are unlikely to be produced by another strategy, such as all cooperates or defects or one that would be created by tit-for-tat.

The pawn class and king class operate in the same way except in which strategy they use against which opponent. The logic used to decide who the opponent is, is shown below. It is implemented as part of the *pawnColluder* class.

```
protected int colludeLogic(int matchCount, Response[][] history, int stratNo) {

    //max the depth at the length of the code
    int depth = matchCount;
    if (matchCount > thecolluder.getCodelength()) {
                                        depth = thecolluder.getCodelength();}

    //get the subarray of the opponents history
    Response[] checker = Arrays.copyOfRange(history[1-stratNo], 0, depth);

    //if the opponents history equals either one of the king
    //or pawn codes then it will return 2 or 1
    if (Arrays.equals(checker,
                    Arrays.copyOfRange(thecolluder.getKingCode(), 0, depth))) {
        return 2;

    }else if (Arrays.equals(checker,
                    Arrays.copyOfRange(thecolluder.getPawnCode(), 0, depth))) {
        return 1;
    }
    //if the opponent is a stranger
    return 0;
}
```

The *colludeLogic* method checks the first few rounds to see if they match any of the codes created by the colluder. This is checked from the beginning, because if one of the first responses doesn't match either code, then instead of continuing to try and broadcast the code, the strategy will immediately react. The method will return a 2 if it matches a king, a 1 if it matches a pawn and a 0 otherwise.

Pawn strategies act by cooperating with king strategies to be exploited, cooperating with each other to boost each other and defecting against unknown strategies to try and minimize their score. A side effect to this is that in an environment of lots of strategies that can be taken advantage of by ALLD, the pawn may outperform the king as the king attempts to cooperate with them.

King strategies act by cooperating with each other, defecting against pawns to exploit them and they play tit-for-tat against unknown strategies.

8 Implementations of Tournaments

This section discusses the implementations and observations made of the two tournaments which are implemented in the following environment:

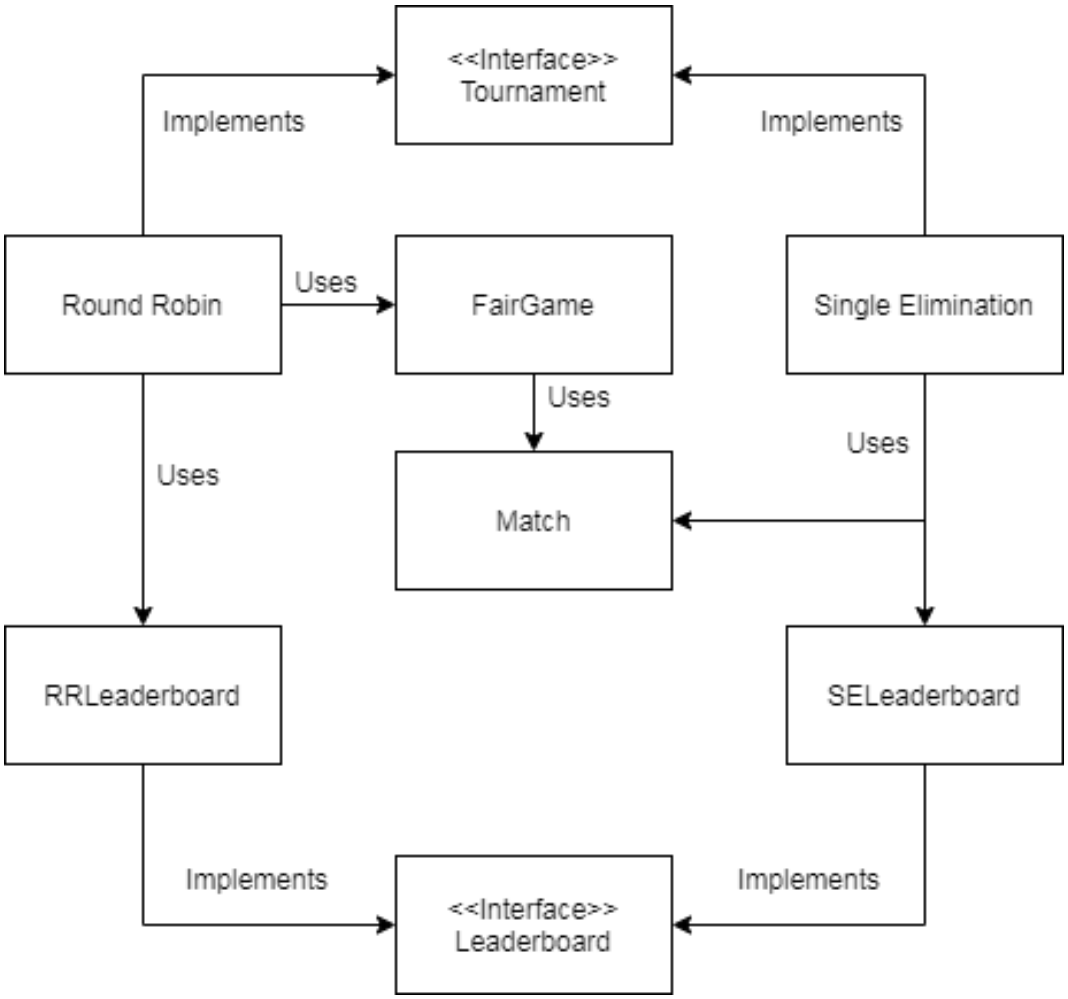

Figure 4: Tournament Class Diagram

8.1 Single-Elimination

As single elimination tournament is of a tree structure, this lends itself to a recursive implementation. Which I have implemented using the following code:

```
//recursively defining the single elimination tree
//(in the case of draw Strategy B is returned).
private Strategy singleElimRec(Strategy[] strategies, SELeaderboard leaderboard) {
    Match theMatch;

    //base case is when there are only two strategies,
    //they play and the winner is returned.
    if (strategies.length == 2) {
        theMatch = new Match(strategies[0], strategies[1],
        ThreadLocalRandom.current().nextInt(roundsLower, roundsUpper));

        //match is added to the leaderboard
        leaderboard.addElement(theMatch.getStrategyA().getName(),
                               theMatch.getStrategyB().getName() );

        //returns the winner
        return (theMatch.getStratAScore() > theMatch.getStratBScore())?
        theMatch.getStrategyA():theMatch.getStrategyB();
    }
```

This section checks if the input array contains only two strategies. This can be considered a leaf as a match requires two strategies to compete. This is the base case. It then creates a match between the two strategies, the parameter for the length of the match can be random as the scoring is not cumulative. There is one winner, who won every match and the rest are simply Losers.

The strategy with the higher score is returned.

```
int halfWay = strategies.length / 2;

    //sub lists are created, splitting the strategies into two pools,
    //which continues until every strategy is paired with only one other.
    Strategy[] strategyListOne = subArray(strategies, 0, halfWay);
    Strategy[] strategyListTwo = subArray(strategies, halfWay, strategies.length);

    //recursively creating a match of the winners of the level below,
    //then returning the strategy that wins
    theMatch = new Match(singleElimRec(strategyListOne, leaderboard),
                         singleElimRec(strategyListTwo, leaderboard),
                         ThreadLocalRandom.current().nextInt(
                         roundsLower, roundsUpper));

    //Leaderboard is updated with the results of the match
    leaderboard.addElement(theMatch.getStrategyA().getName(),
                           theMatch.getStrategyB().getName() );

    //winner is returned
```

```
return (theMatch.getStratAScore() > theMatch.getStratBScore())?
        theMatch.getStrategyA():theMatch.getStrategyB();
```

If the array has more than two strategies, then the list is split into two, down the middle. A match is then created based on the recursive calls of the two sub-lists, which will both return their winner. At completion, the winning strategy is returned.

From this implementation, I have chosen not to use 'BYE"s in order to make the tournament fair. However, this requires that the input list must have 2^n elements, so that sub-lists can always be split in two to create a full tree. a 'BYE' is used in the event that there is an odd number of strategies and one cannot participating, it then gets a 'BYE', allowing it to automatically win that round.

8.2 Round-Robin

The Round-Robin tournament is a basic nested for-loop.

As, like the single elimination strategy, the Round-Robin class implements the tournament interface, it consists of a *play*() method and inherits the fields *roundsLower* and *roundsUpper* which define the range for the random length of the matches.

This is a particularly prominent issue for the Round-Robin tournament and will be discussed in the section below in 8.3.

```
public class Round_Robin implements Tournament {

    public RRLeaderboard play(Strategy[] strategies, int roundsForGame) {
        RRLeaderboard leaderboard = new RRLeaderboard();
        int size = strategies.length;
        int numOfMatches = 3;

        //rounds can be chose to be random if user defines 0.
        if (roundsForGame==0) {
            roundsForGame = ThreadLocalRandom.current().nextInt(
                                        roundsLower, roundsUpper);
        } else {
            roundsForGame = roundsForGame * numOfMatches;
        }

        //FairGame is used here instead of match, as fair game
        //produces multiple matches between two strategies,
        //with random rounds for each match.
        //So all games in the rounds robin will play the same number of rounds.
        FairGame[][] fairGame = new FairGame[size][size];

        //plays matches between strategies all strategies except themselves.
        for (int i = 0; i < size; i++){
            for (int j = 0; j < i; j++) {
                fairGame[i][j] = new FairGame(strategies[i],
                                              strategies[j],
                                              roundsForGame,
                                              numOfMatches);
            }
        }
```

As shown above, a two-dimensional array is initialised to be the same dimensions as the length of the *strategies* parameter. The variable *roundsForGame* can be specified for the purpose of keeping parameters the same during testing. Otherwise, it is initialised as a random number in the range defined by *roundsLower* and *roundsUpper*. This has an important implication that will also be discussed below 8.3.

A nested for loop is run for each strategy in the list *strategies*, it will play a *FairGame* with all other strategies that it has not already played against. Not including itself.

This section calculates the scores of each strategy:

```
int[] totals = new int[size];
//prints out results of games.
for (int i = 0; i < size; i++){

    for (int j = 0; j < i; j++) {
        totals[i] += fairGame[i][j].getStratAScore();
        totals[j] += fairGame[i][j].getStratBScore();
    }

}

//Creates leaderboard and the leaderboard is returned
leaderboard.addRounds((int)Math.floor(roundsForGame/numOfMatches));
for (int i = 0; i < size; i++) {
    leaderboard.addElement(strategies[i], totals[i] );
}

return leaderboard;
```

An array called *totals* is initialised to the same length as *strategies* parameter. Where each element of $strategies[i]$, where the index is i, has the score $totals[i]$. As each pairing is unique in the $fairGame$ array, it allows us to run the same loop as used to create the matches again this time adding to the related total of each strategy in the $fairGame$. Thus the pairing of *strategies* and *totals* is the strategy and the score it accumulated in this specific round-robin tournament.

8.3 Implications of Random Match Length

The reason a random match length is needed is to uphold the Nash equilibrium of the game that is the Iterated Prisoner's Dilemma. If a match has a known length then the logical outcome for a strategy would be to defect on the last turn as there is no more rounds for the opponent to retaliate. As defecting in a unique round will maximise your gain regardless of the opponents decision, which is the Nash equilibrium of the standard prisoner's dilemma. Therefore, as the optimal choice for the last go is to defect, then it will be optimal to defect on the second last go, thus by induction, the only Nash equilibrium would be to always defect. This is illustrated in Wikipedia contributors [2020a].

In avoidance of this, the match length must be large and random. However, this is where we meet the first implication. The implication of fairness, which is explained on the following page.

Strategies	S1	S2	S3	S4	S5
S1					
S2	M(S1, S2, R)				
S3	M(S1, S3, R)	M(S2, S3, R)			
S4	M(S1, S4, R)	M(S2, S4, R)	M(S3, S4, R)		
S5	M(S1, S5, R)	M(S2, S5, R)	M(S3, S5, R)	M(S4, S5, R)	

Figure 5: Round-Robin Tournament Scheduling

As shown in the figure above, Where a set of five (though this number can be arbitrary) strategies play a Round-Robin tournament. Each white cell is where a match is created using the correlating strategies, and a third parameter R, which is a randomly generated. Each match must have an unpredictable and *random* R, so one R is probably not equal to another R.

Thus the total, T, amount of rounds each strategy plays in all there matches is the total of all their associated R values. This is where the problem is found, in order for it to be random, all R values must differ. However, to be fair, all T values must be the same. This is because score is correlated with rounds played, i.e. the more rounds played the higher your score will be. So, all we need to do is find a set of R values that will have each strategy's total T be equivalent. But this, is only possible if all R values are equal. Thus, to the best of my knowledge it is impossible to have both fairness and randomness. This was avoided in the past by having a large enough set of strategies that there was such a large amount of matches that it would mostly average out. However, we do not have a very large set of strategies, so we must confront this.

Randomness and fairness can be achieved using a different structure, as I have used in my implementation using the class *FairGame*:

```
public class FairGame {
    private Match[] matches;
    private int indexTotal;

    public FairGame(Strategy stratA, Strategy stratB,
                    int numOfTotalRounds, int numOfMatches) {

        matches = new Match[numOfMatches];
        int[] rounds = getRounds(numOfTotalRounds, numOfMatches);
        this.indexTotal = numOfMatches;

        for (int i = 0; i < numOfMatches; i++) {
            matches[i] = new Match(stratA, stratB, rounds[i]);
        }
    }
```

As we can see above, a *FairGame* has two strategies, total number of rounds and number of matches as parameters. It creates multiple matches between the same two strategies, each with a random number of rounds, where $match[i]$ has number of rounds $rounds[i]$. Thus each element of *rounds* is random, but the sum of each element equals $numOfTotalRounds$. The array *rounds* is found using the method *getRounds*():

```
//creates an array of integers which represent the number of rounds per match.
private int[] getRounds(int total, int num) {
    int[] rounds = new int[num];
    int whatsLeft = total;

    //each integer in the array is calculated randomly
    //and the bounds change at each iteration
    //to ensure the total of rounds is always the same.
    for (int i = 0; i < num - 1; i++) {
        rounds[i] = ThreadLocalRandom.current().nextInt(1, whatsLeft - (num - i));
        whatsLeft -= rounds[i];
    }

    rounds[num - 1] = whatsLeft;

    return rounds;

}
```

Given a total and number of rounds it loops through the array *rounds* and creates a random value in the range of 1 and how many rounds are still needed, minus an amount so that all rounds are not taken up by one game. This is then returned.

The score is calculated as an average of all matches in a *FairGame*.

Of course some concerns can arise because of this. First of which we will call the 'marathon-sprint implication' which can be defined by the spread of the rounds in the matches created by *FairGame*. For example, we have two strategies, M and S. M performs better the more rounds are in a game, such as an adapting strategy. S performs better the less rounds there are, such as always defect as the less rounds there are the less the opposing strategy can retaliate and the greater the difference in score relative to the amount of rounds.

As implemented, *FairGame* is called with these two strategies, M and S, with $numOfMatches = 5$ and $numOfRounds$ some relatively large, random number. Thus, the array *rounds* is created with 5 elements, the spread of these can vary massively, in terms of percentage of rounds, $(90\%, 2\%, 4\%, 1\%, 2\%)$ and $(30\%, 30\%, 30\%, 8\%, 2\%)$. In the first example four of the five values are 'sprints', thus S will win more matches, and in the second example it is more likely M will win more matches.

My comments on this are two-fold, firstly, in a standard IPD match, the number of rounds is random anyway, which will favour one of M or S anyway. Secondly, is the assumption that winning more matches means getting more points. As the points are an average across all games, in the first example of one marathon and four sprints. The amount of points lead S has over will be much smaller than the amount of points lead M will have in the one marathon, as number of rounds is correlated with score. In the second case, the highest number of rounds is 30% which would not give such great an advantage to M than if the percentage was higher. Thus I believe my implementation to be an example of a random and fair way of structuring matches.

The final implication of having a random number of rounds in the Round-Robin, is the 'absolute value comparison implication'. As each individual round-robin tournament has a random number of rounds, you cannot compare performance of strategies from two different tournaments based on score alone. Each strategy's score must be viewed in the context of the tournament it is played in, meaning the number of rounds total, as well as who the opponents were.

9 Implementation of Genetic Algorithms

The genetic algorithms are implemented into the following environment:

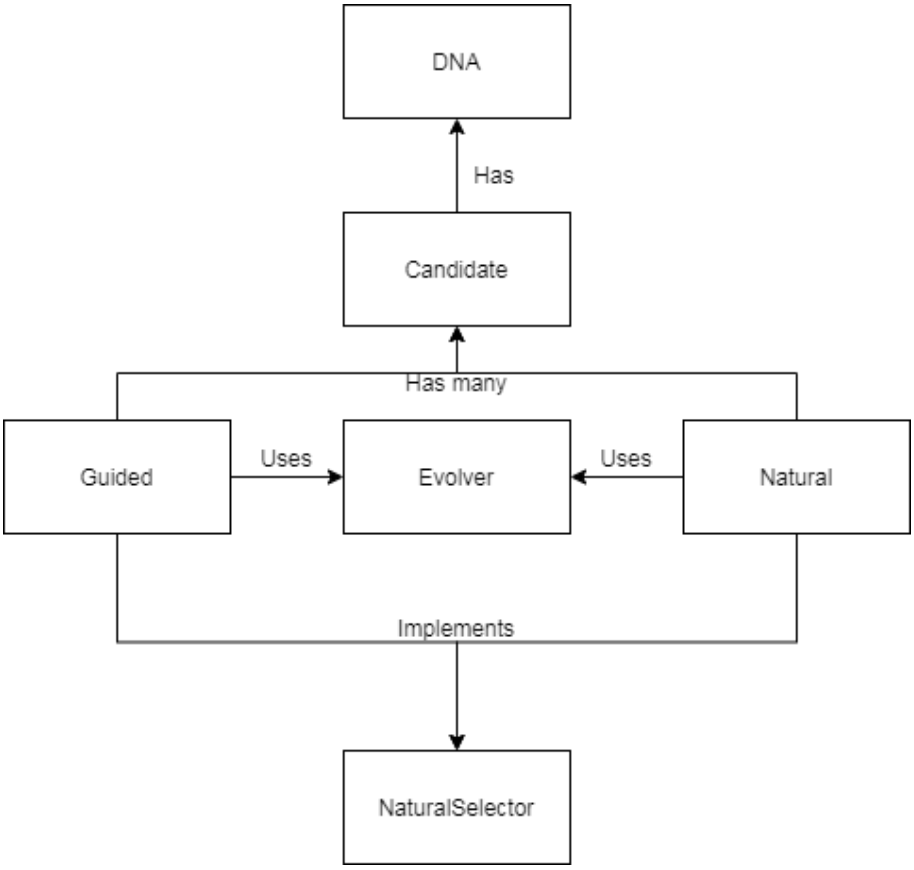

Figure 6: Genetic Algorithm Class Diagram

The Dna class was originally going to be a list within the Candidate class. However, it was far easier to implement as its own class and hide the data structure behind methods.

```
private ArrayList<Response []> wholeDna = new ArrayList<>();
private static int depth = 4;

public Dna() {

    //initialises list of response arrays to size relative to depth.
    for (int i = 0; i < depth; i++) {
        wholeDna.add(new Response[(int) Math.pow(2, (2*i))]);
    }
}
```

As you can see the Dna is simply a list of arrays of responses. When the a new Dna object is created, the arrays in the list are instantiated to the appropriate size. The other two methods that are used mostly are set and get methods, which hide the implementation of the data structure from the other classes.

Another consideration for storing Dna was a tree. The tree would offer a much smaller data size, as redundant data is not stored, as you know what you did in previous moves and therefore do not need to store possibilities where you made a move which is not in your Dna to do. The reason I chose not to use this, is that it only works if the depth of the tree is equal to the amount of rounds. For example in the list method I have, the final sub-list is used for the whole rest of the game and because it holds every possible combination it can deal with any pattern. In the tree example you cannot put a cap on the depth and keep using it as before since the choices made would shift, and then the first node in the tree would not correspond to the first round but the first round relative to the history depth of the candidate. E.g. with a history depth of 5, a candidate can see the last 5 rounds of history. This is perfect for the first 6 rounds, but once the 7th round is reached, the first round according to the candidate will be the actual 2nd round.

Two ways to deal with this is to create a tree which is equivalent to the list solution which stores all possible combinations for each depth up to some max, or to have the max depth equal the number of rounds. For the first, it would require more extensive program structuring and I could just use an array as before, for the second, we do not know the max and if we did it would be in the hundreds which would require roughly 2^{100} nodes in the tree which is way too large.

Candidates are the individuals that have Dna and are evolved. They are strategies themselves and therefore implement the Strategy interface so they can participate in the already existing tournament environment seamlessly. They have a single Dna object field. The Dna is assigned upon creation of the object, thus you cannot change the Dna of a candidate. Having a set Dna method may have been useful to create a set of candidates as the starting population and those candidate objects remain the same and each iteration you alter the Dna of the individuals to be that of their children, but I felt it to be more appropriate to create new candidates for each iteration as I felt it better reflected real life and was therefore easier to understand.

The 'Play' method for the candidate is quite simple. It checks the size of the history, which is match count, and caps it at the Dna depth. It then gets the relevant match history from the history array as a single list where the elements are alternating between my response and your response for that specific round.

```
public Response play(int matchCount, Response[][] history, int stratNo){

    int depth;

    //sets depth to matchcount, unless the matchount
    //is higher than the max depth of the dna.
    //dna stores two dimensional array,
    //depth corresponds to the index of the final sub array.
    //i.e with max depth 5, depth = 4.

    if (matchCount >= Dna.getDepth() - 1) {
        depth = Dna.getDepth() - 1;
    } else {
        depth = matchCount;
    }

    return generateResponse(getList(matchCount, depth, history, stratNo));
}
```

The *generateResponse* method is then called on the resulting list. This method was originally in a different calculator object, but I felt it was better encapsulated as part of the candidate and didn't need its own object. As seen below, it converts the history into binary by creating a boolean array and setting defects as 1's and cooperates as 0's. This binary value is then converted to decimal using left shifts. This decimal number corresponds to the index of the appropriate history, the *getDna* method is called, which returns the relevant response for the index and history size. This solution is far superior to exhaustively searching the history.

```
private Response generateResponse(Response[] list) {
    //convert list to binary then decimal
    int listSize = list.length;

    //convert list to boolean array, representing binary C = 0, D = 1
    boolean[] boolArray = new boolean[listSize];

    for (int i = 0; i < listSize; i++) {
        boolArray[i] = list[i].equals(Response.D);
    }

    //convert binary to appropriate decimal
    int value = 0;
    for (int i = 0; i < listSize ; ++i) {
        value = (value << 1) + (boolArray[i] ? 1 : 0);
    }

    return myDna.getDna(listSize/2, value);
}
```

The Dna and Candidate classes are mainly used to hold and access data. The logic that makes the genetic algorithms work is the *Evolver* class, which is used to create offspring for future generations. It is implemented statically as it only contains methods and one static field, so it holds no state and is used as a 'helper' class. It consists of one public method called *breed* which takes and ordered list of candidates and produces the next generation. As you can see below, the method takes a list of strategies and returns a list of strategies, even though not all strategies can be bred. This is due to the fitness function which is a system that can be used by all strategies, not just candidates. This means the individuals are of the strategy type even though they are instantiated as candidates.

```
public static Strategy[] breed(List<Strategy> individuals) {
    int splitNumber = 2;

    //splits the population in half
    int subSize = individuals.size()/splitNumber;

    //takes the first half to breed
    List<Strategy> firstHalf = individuals.subList(0, subSize);

    //resets individuals list
    individuals = new ArrayList<>();

    //for every pair of individuals (0,1), (2,3) etc.
    for (int parentCounter = 0; parentCounter < subSize;
                                parentCounter = parentCounter+2) {

        //makes babies relative to how large the split was.
        for (int babyCounter = 0; babyCounter < (splitNumber*2); babyCounter++ ) {

            //adds a child of the parents to the new individual list.
            individuals.add(uniformCrossover(
                    (Candidate) firstHalf.get(parentCounter),
                    (Candidate) firstHalf.get(parentCounter+1)));
        }
    }

    //also converts list to array of strategy.
    return individuals.toArray(new Strategy[individuals.size()]);
}
```

As the list of individuals is ordered, the top half are the best performers and are selected to be bred. The outer for-loop goes through the top half of the individuals and pairs them off with the next partner. I.e. 1 goes with 2, 3 with 4 etc. The inner loop is iterated through twice the split number which ensures the population size remains constant. The *uniformCrossover* method is called on the two selected parents, this method is the method that creates a child from two candidates. It is here where the parents are cast as candidates so that their candidate exclusive methods can be used. Although I do not like casting, the alternative was to refactor the tournament code in some way, but would ultimately require some form of casting, so I thought best not change what I already had working.

The *uniformCrossover* private method takes two candidates and returns a single 'child' strategy. Uniform crossover is a method of crossover where each gene in the Dna is selected uniformly as a fifty-fifty chance from each parent. Alternatives are to crossover blocks to keep groupings of genes together. However, in this case uniform crossover sufficed and resulted in more variation, it was also far easier to implement.

```
//uniform crossover considers each bit seperately
//with a chance that it becomes one parent or the other.
private static Strategy uniformCrossover(Candidate p1, Candidate p2){

    int depth = Dna.getDepth();
    Dna babyDna = new Dna();
    Candidate baby;

    //for each level of dna.
    for (int level = 0; level < depth; level++) {

        //for each chromosome in the level of dna;
        for (int actual = 0; actual < Math.pow(2, (2*level)); actual++) {

            java.util.Random random = new java.util.Random();

            //randomly select the baby dna from one of the parents.
            if (random.nextBoolean()) {
                babyDna.setDna(level, actual,
                    p1.getDna().getDna(level, actual));
            } else {
                babyDna.setDna(level, actual,
                    p2.getDna().getDna(level, actual));
            }


            //babyDna has a small chance of mutating
            if (random.nextDouble() < rateOfMutation) {
                babyDna.setDna(level, actual,
                    mutate(babyDna.getDna(level, actual)));
            }

        }

    }

    baby = new Candidate(babyDna);
    return baby;
}
```

As seen above, the for loops traverse the structure of the Dna. The outer loop goes through the outer list, the inner loop goes through each item in that sub-list. A random boolean is created which decides from which parent that gene shall come from. A mutation method is then called for that gene, which has a small chance of becoming the opposite of what it was set as, which

creates a small chance for new genes to be created so that the system is not completely reliant on the initial population. There was an issue I did not realise until much later where I had set the rate of mutation to 0.1, however because the *nextDouble* method is between 0 and 1, this resulted in a 10% chance of mutation.

There are two forms of evolution that I implemented, Natural and Guided. Both of which share many similarities to each other, so I created a parent class to hold some common fields and methods called *NaturalSelector*. It contains two abstract methods used to get the plottable list of the best performers from each generation and a method to get the leaderboard from the last iteration.

```
public abstract List<XYChart.Data<Number, Number>> getBestList();
public abstract Leaderboard getLastLeaderboard();
```

There are two further methods used to create the initial population, since both Guided and Natural require a starting population, formulated in the exact same way. The main method, *getInitialPopulation* takes the user defined population size as a parameter and returns an array of candidates. It then loops through to each new candidate and creates Dna for them, similarly to the *breed* method except the response for that gene is determined randomly and not from parents. The other method is *getRandom*, which simply returns a random response.

```
protected Candidate[] getInitialPopulation(int population) {
    Candidate[] strategies = new Candidate[population];
    Dna.setDepth(4);

    //for each member of the initial population
    for (int member = 0; member < population; member++) {

        Dna myDna = new Dna();
        //creates dna for each candidate and assigns that dna to the candidate.
        for (int level = 0; level < Dna.getDepth(); level++) {

            for (int actual = 0; actual < Math.pow(2, (2*level)); actual++) {
                myDna.setDna(level, actual, getRandom());
                strategies[member] = new Candidate(myDna);
            }

        }
    }

    return strategies;
}
```

Though I tried to avoid it, the only way to make the data comparable is to have a static amount of rounds, so that each iterations can be compared against each other. This resulted in the ability for the user to specify the amount of rounds or to have them remain as random.

9.1 Implementation of Natural

Natural evolution as stated before, is simply a method of selecting effective individuals. In the natural environment they are scored based on performance against each other.

The evolution loop will then begin using the starting population of random individuals:

1. RR-tournament played with starting individuals.
2. Using the returned leaderboard the breed function will be called.
3. With the new list, which has been bred from the best half and mutated will become the next population.
4. Then Loop

Natural evolution consists of just a constructor. Thus you cannot have a stateless Natural Selector object. A list of starting strategies are created, as well as an array of leaderboards, this was meant to allow for the ability to select the leaderboard for any iteration, however I decided to only show the last one as it was much simpler to implement. The loop then begins as defined above and they play a tournament which determines their fitness score, which is their tournament score. This is stored in the leaderboard for the tournament. The best performer for that iteration is saved in a list of the best performers which will be plotted. The *breed* function is called and the new population of individuals are created, then the process repeats.

```
public  Natural(int population, int iterations, int roundsForGame) {

        Strategy[] individuals = getInitialPopulation(population);

        Round_Robin tourney = new Round_Robin();
        RRLeaderboard[] results = new RRLeaderboard[iterations];

        for (int i = 0; i < iterations; i++) {
            results[i] = tourney.play(individuals, roundsForGame);

            //List from the leaderboard used to get the ordered
            //list of strategies based on performance
            List<Strategy> newList = results[i].getGeneticList();

            if (i == iterations - 1) {
                lastLeaderboard = results[i];
            }

            int bestScore = results[i].getStrategiesList().get(0).getKey();
            bestList.add(new XYChart.Data<>(i, bestScore));

            //the evolver returns the next population
            individuals = Evolver.breed(newList);
        }
```

9.2 Implementation of Guided

Guided evolution is similar to Natural in most respects including implementation except with one change, that the fitness of individuals is not determined relative to each other, but relative to the defined list of static strategies. Thus the code is identical as above except the first line of the loop is replaced by what is below:

```
for (int j = 0; j < population; j++) {
    Strategy[] current = addToArray(strategies, individuals[j]);

    //play tournament between the selected strategies and current candidate
    results[j] = tourney.play(current, roundsForGame);

    //List from the leaderboard used to get the ordered list of strategies
    //based on performance
    Pair<Integer, Strategy> geneticIndividual =
        getGeneticStrategy(results[j].getStrategiesList(), individuals[j]);

    geneticResults.addElement(geneticIndividual.getValue(),
                              geneticIndividual.getKey());

    if (geneticIndividual.getKey() >= overallBest.getKey()) {
        overallBest = geneticIndividual;
    }

}
```

Here the list of candidates is iterated through, for each individual a new array is created which consists of just them and all the other user defined strategies. They participate in a tournament, a method called *getGeneticStrategy* is called on the resulting list of strategies from the tournament to find the genetic algorithm strategy in the list to get its score. The leaderboard system works using the pair class, the key of the pair is the score and the value is the strategy. This is then added to a new leaderboard. Once all individuals have participated, the result is a leaderboard of these individuals, which then progresses the same way as *Natural* evolution does, this is the first iteration.

10 Implementation of the GUI

The GUI was created using JavaFX with scene builder to create an FXML file which dictates the layout. The *Main* class shows the scene on start up, which appears as below:

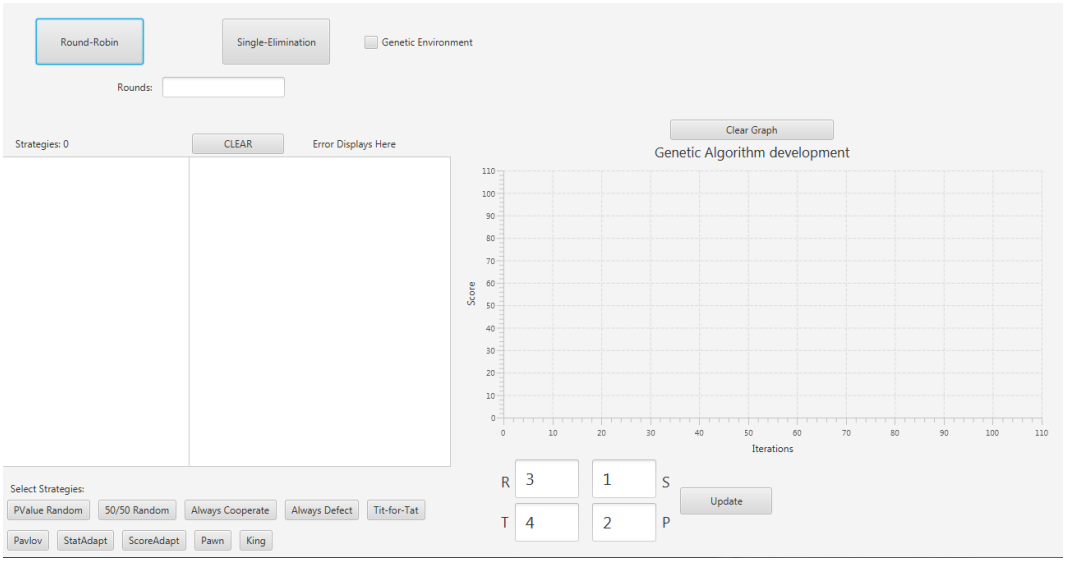

Figure 7: GUI Display

A controller class called *MainScreenControl* is used to interface between the UI and the rest of the system. The main screen controller handles all button clicks, text-field input and displaying output. For standard use, the buttons at the bottom left when pressed, will call their relevant handling method in the controller. All of the button handlers for adding strategies call the same function.

```
private void updateStratAdded(String stratType) {
    strategyList.add(stratFactory.getStrategy(stratType));
    updateListView();
    updateStratCount();
}
```

This function takes a string, which is specified in the handler function, and uses the factory class to produce the appropriate strategy type which is then added to the list of selected strategies. The *UpdateListView* function updates the left menu panel to display the newly added strategy, and the number of strategies displayed via the label is increased by one.

Tournaments function in the exact same way except using the tournament factory. The single elimination handler has some additional checks to see whether the number of strategies is a power of 2. This is done by using the binary representation of numbers. Any power of 2 in binary form has a single 1, so if you subtract 1 from the number you will essentially get the reverse, by using a logical AND operator you will receive 0 if the original number was a power of 2 and a number greater than 2 otherwise.

The genetic environment checkbox is used to toggle between a standard environment and genetic environment, as each presents different options and additional parameters. The genetic algorithms *Natural* and *Guided* are also created using a factory, as they both share the same interface of natural selector, they both share the same method:

```
private void doGeneticAlgorithm(String type) {

    int population;
    int iterations;
    int rounds;

    try {
        rounds = Integer.parseInt(roundsField.getText());
        population = Integer.parseInt(geneticPop.getText());
        iterations = Integer.parseInt(iterationField.getText());


        if ((population%4) != 0) {
            //because top half of the population has to be paired.
            errorLabel.setText("population must be mod 4.");

        } else {
            NaturalSelector nsElector = nSFactory.getNaturalSelector(type, population,
                    iterations, rounds,
                    getStratArray());

            ObservableList<String> list = FXCollections.observableArrayList();
            list.setAll(nsElector.getLastLeaderboard().getLeaderboardList());
            leaderboardListView.setItems(list);

            updateGraph(nsElector.getBestList(), type + " best results");
        }

    } catch (NumberFormatException e) {
        errorLabel.setText("textfields must be integers.");
    }
}
```

The three primary parameters are population, iterations and rounds, which are used by both classes. A secondary parameter which is only used by the *Guided* class is a list of strategies. To accommodate this, the factory class takes all parameters and only uses the relevant ones for the appropriate class. There is a check to ensure that the population is a multiple of four for the sake of keeping the method used to breed them simple, it easier to check on this level. The observable list which displays the leaderboard is updated and the graph is updated.

11 Findings and Experiments

The results of a standard environment with strategies discussed are often very predictable. I categorise strategies into four main categories by which there performance is decided. exploiters(E.g. ALLD), exploitable(E.g. ALLC or Random), exploiters with awareness (E.g. StatAdapt), exploitables with awareness (E.g. TFT). Most strategies fall under these categories and the way they interact with each other is rock, paper, scissor like. Therefore the outcome is determined by the ratio of one group to another, with the ratio between the payoffs accounted for.

If there are enough exploitable strategies, exploiters will overtake cooperators. Or if their are enough exploitable strategies that can retaliate, they can overpower an exploiter. Therefore, because of this reliance upon the environment for performance, there is no universally optimal strategy, though some are better than others.

11.1 Findings of Natural Evolution

From observing the plots, what I can see is that the top scores do not fluctuate too much, however there seems to be a trend of dropping in score for the first few iterations and then slowly building up like an s-curve. An example of this is shown below in an environment of 100 individuals participating in 50 rounds per game with 500 iterations:

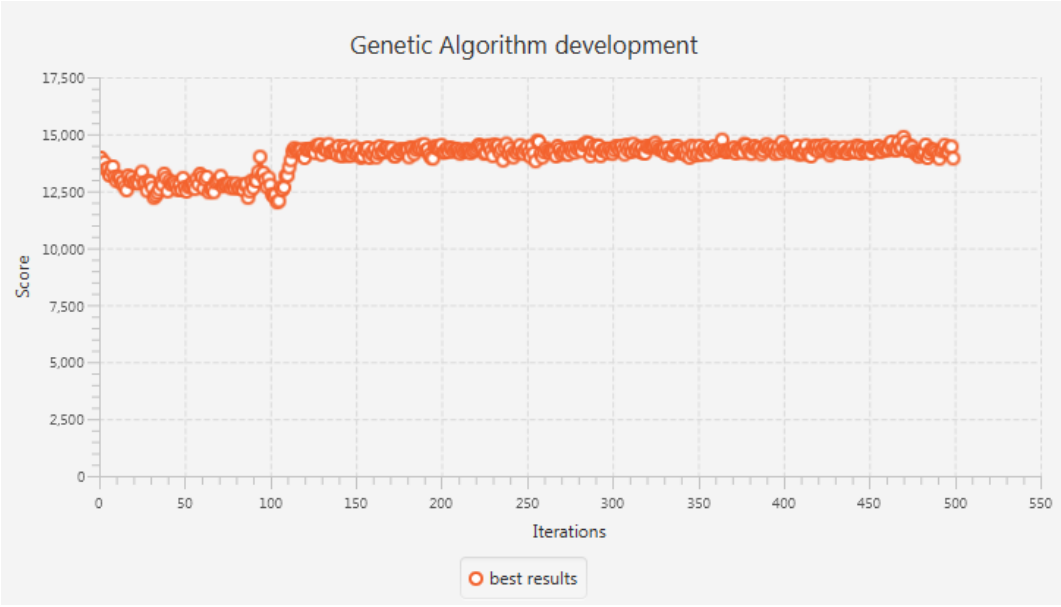

Figure 8: Example Results Diagram

Very interestingly, an emerging behaviour is mutual cooperation, this can be seen by looking at the first chromosome of each genetic sub list.In the beginning it seems to favour defection but is mostly random. But quite quickly it becomes almost exclusively cooperation in the first chromosome, this is the chromosome which decided what to do when their has been nothing but mutual cooperation. However, the ration of D to C seems to favour D. out of 85 chromosomes the ratio of

D seems to be between 45 and 55 and C seems to be 30 to 40. Which tells us that most winning strategies seem to defect most of the time. However, we do not know which genes are responsible for an individuals performance. We can infer based on the fact that most genes for producing a response when the history has only cooperation tends to massively favour cooperation. The gene that dictates the response for solely defection in the history is about fifty-fifty. This lack of a trend tells us one thing, it does not matter as much. The reason for this is that it's probably used minimally, the fact that mutual cooperation produces almost always cooperation tells us that that this gene is being selected for. Which probably means that the individuals get to a stage where they are almost always cooperating. A further explanation for having more defection genes overall, is many of the histories could only be produced by random individuals, we know that the optimal response to a random strategy is to defect. Thus, defection is selected for in these cases.

Through normalising the graph, I can observe a pattern in a new environment of a population of 40 individuals participating in 50 rounds per game with 1000 iterations:

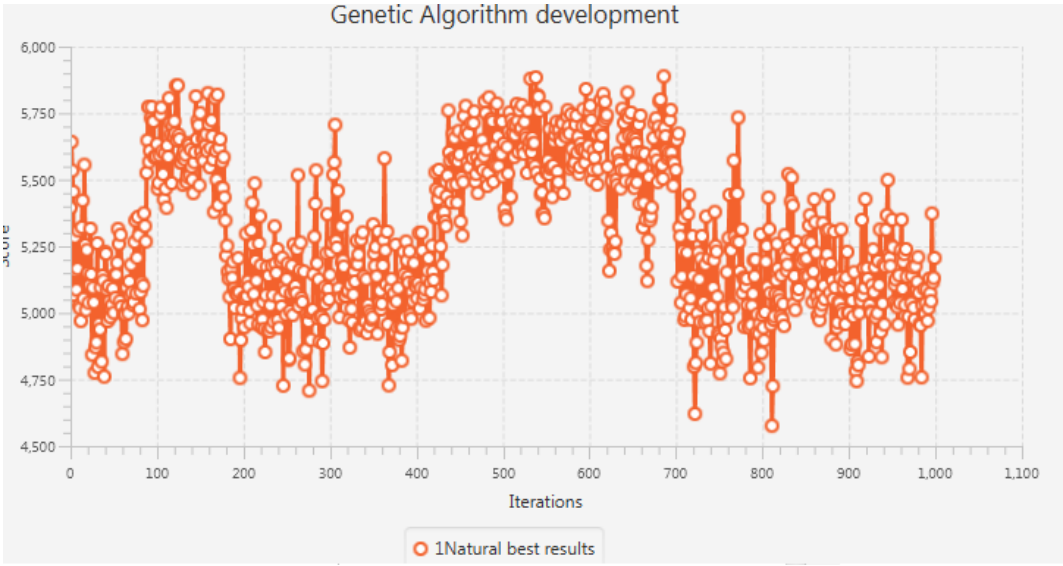

Figure 9: Example Fluctuations Diagram

There is a fluctuation, and this can be attributed to the changing environment. In the beginning there is always always a negative gradient. This is because, as we know the best strategy against random strategies is ALLD, thus in the beginning when all the strategies are random the dominant strategies will be the ones that exploit the randomness. However, as these ALLD-like strategies take over the gene pool, the scores drop as they can no longer exploit their opponents as almost all opponents will be ALLD-like. An example of an ALLD strategy is shown below, by observing the last element of each sub-list we can see that when met with ALLD it will defect and will also defect against ALLC:

: D : CDDD : DCCCCCCCDCDCCDDD :
DCDDDDDDDDCDDDCDDCCDCDCDDCCCDDCCDDCCDCDCDDDDCDDCDDCDCCDDCCCDDDDCDD

This is where the s-curve comes in and there is a sharp increase to often higher than the starting point, through looking at the dna of strategies during this point mostly they start with cooperation:

: C : CDCD : CCDDDCDCDDCDDDCC :
CDCDCCCCDDCCCDDCCCCDDDDDDCDDDDCDCDCDDDCDDDDDCCDCCDCCCDDDDCDDCDDD

Unlike before. The fact that these strategies can dominate in an environment where shortly before the majority was ALLD-like is that they evolve some system akin to TFT. They learn not to be taken advantage of so easily. This is often stable for some time, but eventually it will decline again.

The reasoning for this, though it is hard to see from the DNA, is that the prominence of TFT strategies in an environment with no ALLD strategies, facilitates the evolution of ALLC. This was shown to me when reading Hizak [2016]. As in an environment with no predators TFT and ALLC are functionally identical. However, the rise of ALLC then allows for exploitation once again and the cycle continues. To further this the predator strategies can be more sophisticated than ALLD and probe the opponent to see if it reacts, for example:

: D : CCDC : CDDCDCDDCCDCDDDD :
CCDCDDCCCCCDCCDDDCCCCDDCCCCCCCDCDDCDDCDDCDCDDCCDDDCCCCDCCDDDDDCDD

As you can see, it begins with a defection. If the opponent cooperates it will defect again, however if it defects it will cooperate. From the final section, we see that from the first cell, if the history has only been cooperation, it will cooperate, from the 41st cell which is the history where the opponent has been cooperating and you have been exploiting, it will keep exploiting and if the history has been only defection it will continue that.

This pattern is mainly observable at lower populations sizes, around 50 is where strategies become somewhat stable to a TFT-like strategy where a much greater number of mutations are required to overcome the TFT strategies which is unlikely.

A further observation I made, was how the parameters such as round length and the population affected the results. I expected both to contribute to some form of stability, which they both did, though in different ways. In a test done with 1000 iterations and a population of 80 with a variable number of rounds, as I increased the number of rounds the variability between the highest scoring members of different iterations decreased. The amount that each sample deviated from the previous sample was lower, this is due to more of the Dna being represented in a game as there are more chances for different genes in the Dna to be used.

Similarly in a test done with an environment of 1000 iterations, 100 rounds with a variable population, the stability increases and the fluctuations between defecting and cooperative strategies decreases sometimes to a single stable strategy. The single stable strategy being cooperation and defecting strategies were never observed to be stable and would always revert to cooperation at some point. From starting at a population of 20 and increasing by 8 each time to 100, the maximum spread from the mean normalised to the population size went from 110 and decreased quickly at first then becoming stable in the high 50's as the population increased above 60, with minimal difference thereafter. The reasoning behind this increase in stability is that it takes a larger quantity of invading strategies to displace the current environment. E.g. in an environment of 10 cooperative strategies, if one became a defector they would gain more of a benefit than if they were one defector in an environment of 50 cooperators. This holds so long as atleast half of

the population of cooperators retaliate to defection.

11.2 Findings of Guided Evolution

Guided evolution is extremely successful and will almost always find an optimal solution for a given set of strategies within a hundred iterations. An example scenario of 18 strategies including the genetic algorithm in just 500 iterations the results are below:

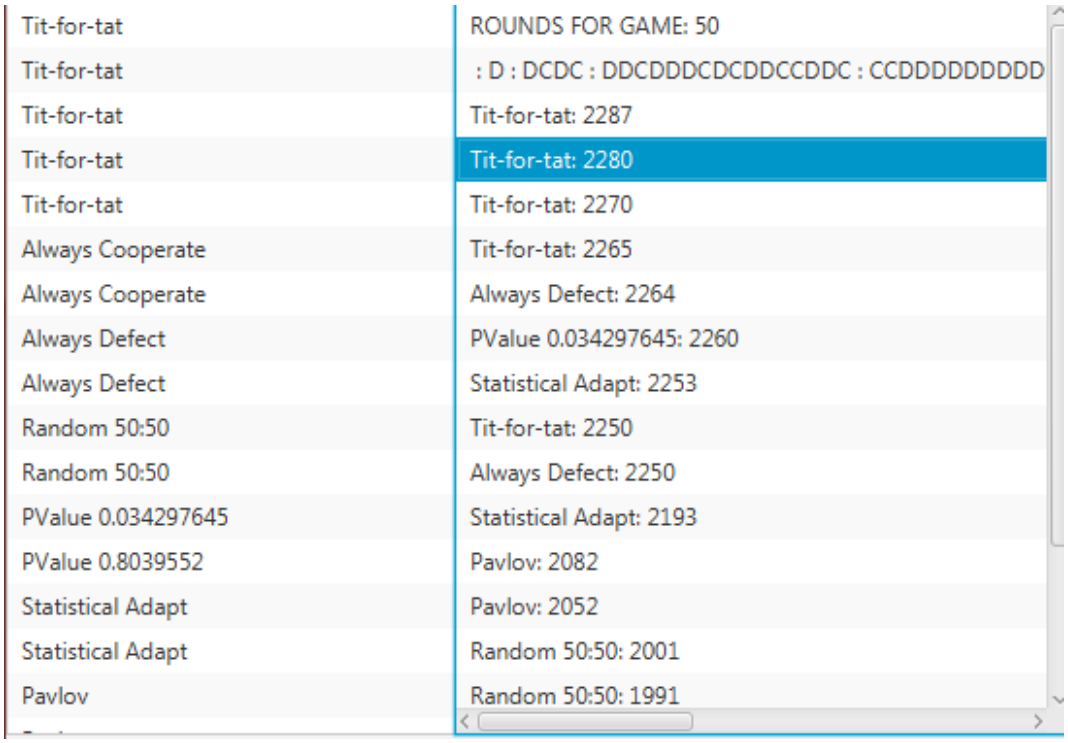

Figure 10: Example Guided Leaderboard Diagram

As we can see it outperformed all others and achieved a score of over 2500, which is 10% higher than TFT. We can also see that it begins with defections so we know it does not like being taken advantage of, however in a history of ALLC it will cooperate and we know it must cooperate with some strategies to score higher than ALLD, so it acts like a probe and tries to take advantage of its opponent if it can and if it cannot it will attempt to cooperate and if the opponent does not cooperate it will defect.

A few things to be careful of, I thought 'Interestingly, in an environment of only random strategies and one ALLD, it will almost always perform slightly better than ALLD by a few points.' This conclusion is incorrect. The reason that it appeared as such is because of the way I implemented the selection of the leaderboard. I chose the leaderboard where the genetic algorithm performed the best, the problem with this is that because the random strategies produce completely random results and when the same random strategy plays again it will not produce the same responses at the same round. This means that the amount of cooperations and defections are not completely half and half. Thus the leaderboard that tends to be chosen is the one in which the random

strategies were cooperating more against the genetic algorithm than the ALLD strategy. Thus the genetic algorithm could score higher by taking by getting the temptation payoff more.

A few example situations are:

- In any extreme environment of either only ALLD or only ALLC, it will evolve to always defect. Which we know is the optimal strategy for this environent.
- In the same environment as above, but where tft is introduced, it will evolve to take advantage of ALLC, always defect against ALLD, but it will evolve to cooperate with TFT since it retaliates. This is more subtle than TFT as it takes advantage of the opponent when it can.
- The main environment where it does not always produce an optimal solution is when a form of adaptive randomness is used. Such as 'statadapt', because it starts very random and becomes less random as the game progresses, the genetic algorithm cannot easily recognise it and therefore has a less stable progression.
- Against colluding strategies it will discover the opponents code and abuse it to appear as a king.

12 Professional Issues

The observation Hizak [2016] and I made of the cyclical behaviour of genetic algorithms can reflect the often-said statement that history repeats itself. All humans have experienced war across history, and most would consider it bad, yet it continues to be a trend. Furthermore, if we consider war to be defection, we would expect after lots of war, peace to eventually take over with a retaliation mechanism such as with Tft which implies forgiveness. However, this is not the case, as stated in Summerfield [2002], "In 1999, a survey of 600 households of Kosovo Albanians by the Centers for Disease Control and Prevention found that 86% of men and 89% of women had strong feelings of hatred towards the Serbs. Overall, 51% of men and 43% of women had a desire to seek revenge most or all of the time". This shows that the desire to continue violence is still very much present after the fact.

People generally think of this as destructive, yet it continues to be an individual's feelings. So why do individuals generally feel anger and the need to continue violence, yet as a society we believe that the 'healthiest' option is forgiveness and not to retaliate?

Some argue that the prevalence of war is due to the cooperation between citizens and unity in the face of the common enemy. So cooperative agents conducting uncooperative behaviour towards others? Real life behaviour is not rational and not easily modeled Thus, we cannot use mathematical models such as the one I have created to dictate what behaviour to expect from one another or from groups. Though that is not to say they aren't relevant, but less accurate than we may like them to be. Furthermore, some people don't like the idea that their behaviour can be modeled and you often see people complaining about the conclusions of studies because their personal experiences or beliefs contradict it. So as scientists, we need to be careful about how others may react to what we make claims about, though I still believe it's 100% necessary to make those claims in the interest of discovery.

Another caveat to groups of individuals is what Rogers et al. [2007] and I illustrated through colluding strategies. Most of human and animal societies are made of groups and it has been observed that self-sacrifice and needlessly altruistic behaviour which seems illogical on an individual level is beneficial at the level of a group. For example if two groups go to war it may be beneficial for the group that a few sacrifice themselves for the good of the group. Thus awareness of the level of abstraction needs to be taken into account when modelling a situation.

13 Self-Evalutation

Overall the project went well, I found the topic very interesting and I was excited that I got to use genetic algorithms specifically. The most difficult part, that I wish I had done better, is coordinating what I need at lower levels of the program for higher levels. When I had a plan of what I needed for the top levels it made it far easier to begin from the bottom, for example when I began adding the genetic algorithm environment I did not know how it would work when I first began the tournament structure and the leaderboards. Because of this, I had to refactor types in

the leaderboard so that I could retrieve the whole strategy object and not just the name, which resulted in storing multiple lists of essentially the same information, but in different types. I was also unsure of which design patterns to use and which methods to use where, this resulted in similar things being done slightly differently across different classes. Such as using static fields and methods or a singleton.

I also had difficulties managing the branches in the repository and often encountered errors, so I began committing straight to the trunk, which I wish I hadn't done. I also did not test everything and didn't always use TDD which sometimes resulted in things going wrong and me not knowing that it was until much later or not knowing what was causing it, oftentimes an off-by-one error in some for-loop.

Something I believe I did well was writing reports for the parts of the program I was working on as I was working on them, so everything I did was documented. As a whole, I think I structured the program well and had minimal difficulties when dealing with the structure, especially when I fully thought things through prior to implementation, so I've definitely learnt the importance of planning. I feel far more confident in creating larger projects and I enjoyed it so much I will probably continue in my free time afterwards.

There were a few things I wish I had added, mainly for ease of use and easier experimentation, namely: saving of genetic algorithms so that you can use specific ones as normal strategies, the ability to save leaderboards and some way of easily analysing the Dna of genetic algorithms to see if they resemble a know strategy or presenting what their general behaviour is in a more readable way.

Based on my original risk evaluation, I was very cognoscente of the risks throughout. There was minimal scope creep as I had a clear idea of what I wanted to achieve from the beginning and I stuck to it throughout. The risk of badly designed program structure was largely avoided, I spent probably double the time thinking about how to structure and implement classes as I did actually implementing them. This resulted in much faster implementation than I originally expected and when I began a task, it was completed relatively quickly. Another risk was not understanding the theory behind what I was implementing, resulting in an inaccurate implementation. I spent a lot of time researching each section, specifically the genetic algorithms as I was unsure of how to implement it. Through seeing how others did it, I managed to use that knowledge to create my own implementation that is somewhat unique and that I'm proud of. The risk I feared to be most likely was plan breakdown, and not following the timeline. In the beginning I followed the timeline I had set closely, in the second term due to exterior issues I did not follow it as closely, but as I had done a majority in the first term, it was far easier in the second term.

References

Robert Axelrod et al. The evolution of strategies in the iterated prisoner's dilemma. *The dynamics of norms*, pages 1–16, 1987.

S. Bukhari. Using genetic algorithms to develop strategies for the prisoners dilemma. 12 2005.

Argyrios Deligkas. private communication, November 2020.

Frank Harary and Leo Moser. The theory of round robin tournaments. *The American Mathematical Monthly*, 73(3):231–246, 1966.

Jun-Zhou He, Rui-Wu Wang, Christopher XJ Jensen, and Yao-Tang Li. Asymmetric interaction paired with a super-rational strategy might resolve the tragedy of the commons without requiring recognition or negotiation. *Scientific reports*, 5:7715, 2015.

Jurica Hizak. Genetic algorithm applied to the stochastic iterated prisoner's dilemma, 08 2016.

M. Jurišić, D. Kermek, and M. Konecki. A review of iterated prisoner's dilemma strategies. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 1093–1097, 2012.

Michael P Kim, Warut Suksompong, and Virginia Vassilevska Williams. Who can win a single-elimination tournament? *SIAM Journal on Discrete Mathematics*, 31(3):1751–1764, 2017.

David Kraines and Vivian Kraines. Learning to cooperate with pavlov: An adaptive strategy for the iterated prisoner's dilemma with noise. *Theory and Decision*, 35:107–150, 1993.

Jiawei Li, Philip Hingston, and Graham Kendall. Engineering design of strategies for winning iterated prisoner's dilemma competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, 3:348–360, 12 2011. doi:10.1109/TCIAIG.2011.2166268.

John Monterosso and George Ainslie. The behavioral economics of will in recovery from addiction. *Drug and alcohol dependence*, 90:S100–S111, 2007.

Alex Rogers, Rajdeep K. Dash, Sarvapali D. Ramchurn, Perukrishnen Vytelingum, and N. R. Jennings. Coordinating team players within a noisy iterated prisoner?s dilemma tournament. *Theoretical Computer Science*, 377(1-3):243–259, May 2007. URL https://eprints.soton.ac.uk/263238/.

Warut Suksompong. Scheduling asynchronous round-robin tournaments. *Operations Research Letters*, 44(1):96–100, 2016.

D. Summerfield. Effects of war: moral knowledge, revenge, reconsiliation, and medicalised concepts of "recovery". 2002.

The Economist. Playing games with the planet. https://www.economist.com/finance-and-economics/2007/09/27/playing-games-with-the-planet, 2007. [Online; accessed 21-October-2020].

Wikipedia contributors. Prisoner's dilemma — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Prisoner%27s_dilemma&oldid=983089784, 2020a. [Online; accessed 12-October-2020].

Wikipedia contributors. Nash equilibrium — Wikipedia, the free encyclopedia, 2020b. URL https://en.wikipedia.org/w/index.php?title=Nash_equilibrium&oldid=992436119. [Online; accessed 6-December-2020].

A Appendix

A.1 Plan

My goals for this project in order of importance are :

1. To create a tournament environment for the strategies that can host multiple strategies.
2. To order the strategies at the end of the tournament based on performance to some win condition, one of:
   - A Round-Robin where points are assigned.
   - A Single-Elimination tournament.
3. To create strategies from varying methodologies.
   - Deterministic strategies.
   - Adaptive Strategies.
   - Colluding Strategies.
   - Strategies cultivated through genetic algorithms
4. To create a simple and intuitive graphical user interface to represent the whole system.

1) Virtual tournament environment: This part of the research is to produce a scale-able and generalised environment where no knowledge of the strategies is required, I will write reports on possible design patterns to implement this and produce proof of concept programs for any promising candidates.

2) Leader-board: The focus for this part of research will be to define a win condition for the strategies. Current candidates is the mentioned round-robin style and knockout. I will write a report on each weighing up the pros and cons and creating proof of concept programs to see how each would function in a real life scenario.

3) Strategies: This part will probably require the most research as my goal is to create a wide variety of strategies with different methodologies and complexities. Research will need to be

done on different deterministic and adaptive strategies, as well as simple static strategies to use as comparisons. As an extensible goal, I would like to include genetic algorithms and colluding strategies, each will have an associated report.

4) User-Interface: The priority for the user interface is use ability, starting with a simple graphical interface. Differing implementations of this will require more research into java GUI frameworks.

A.2 Timeline

Timeline for deliverables, keeping tasks short and not too reliant on others. Beginning and ending within one week, unless 'begin' is mentioned, indicating a longer more substantial task.

A.2.1 Term 1

Week	Task
2	Write reports on different tournament types and discuss implementations.
	Create UML diagrams of program structure for the basic game.
	Write reports on basic strategies.
3	Implement basic strategies into one vs one environment.
	Write report on deterministic and adaptive strategies.
	Begin implementation of whole software structure using UML from week before using TDD.
4	Create basic but use-able GUI for the software.
	Write report and implement interface for strategy object.
5	Implement decided tournament type keeping game general.
	Implement the more complex strategies researched in previous weeks, ideally into the tournament e
6	Write report on genetic algorithms and potential implementations.
	Implement leader-board system that displays the results of a completed tournament.
7	Join all code fragments together to create a full user story,
	the user should be able to, using the GUI, select algorithms to compete in a tournament
	then be presented with a leaderboard based on the outcome of the strategies.
8	Finalize interim report
9-10	Buffer weeks to ensure all tasks completed.

A.2.2 Term 2

Week	Task
1	Begin implementation of genetic algorithm suite where genetic algorithms can be 'evolved'.
2	Implement saved leader-boards so previous tournaments can be reviewed.
	Write report on 'colluding' strategies, where multiple strategies 'boost' one to win.
3	Implement 'colluding' strategies.
	Write report discussing all the different strategies and their performance.
4	Write report discussing the parallels between the real world and the project and how multi-agent systems are used in society.
5	Add genetic algorithm suite to project and allow for saving particular past algorithms that have been 'evolved' to be used in future tournaments
6	Write report discussing programming techniques used, program structure and design patterns used.
	Write report discussing professional issues and a self evaluation.
7	Finalize GUI
8-9	Buffer weeks to ensure all tasks completed, consider beginning on tasks for term 2.

A.3 Diary

WC - 5 October: Completed the original project plan and began a report discussing the basic strategies I could implement to create a basic match system. My supervisor and I agreed to start here specifically, to work from the ground up.

WC - 12 October: I implemented a basic match system where two strategies can compete, this was a simple environment, the UML for which can be seen below. I started implementing this early to spread out the implementation process to allow for any possible delays as well as insuring I would have as much as possible as a deliverable for this report. However, only implemented ALLD and ALLC at this point as they are the easiest to implement. There was no point developing further strategies when I had no way of testing whether they worked in a match yet.

WC - 19 October: I added more complex strategies, TFT, pavlov etc. Which allowed for more interesting matches, especially with the random strategy as this made the results non-deterministic.

I began research on tournaments, I thought it would be interesting to do two variations of tournaments, Round-Robin and Single Elimination to see how the structure of the tournament can affect the results.

I implemented Single-Elimination tournament first, namely because at this point I had over-complicated the research for a round-robin by looking at algorithms for an asynchronous environment, such as real life where one team can't play multiple matches at the same time and trying to complete in as small time as possible; this was unnecessary as my Round-Robin environment was synchronous.

WC - 26 October: I implemented p-value strategies so I could create a larger variation of strategies for the tournaments. Before I had only five strategies which wasn't enough to host a larger tournament unless I used multiples.

I implemented the Round-Robin tournament as a nested loop where each strategy plays each other exactly once, not including themselves. During implementation I discovered an issue where I couldn't make each match random as scores are cumulative and therefore each player in a round-robin must play the same total number of rounds. However, I could think of no way to make each match random and total the same number of rounds for each strategy. So, I left the rounds as a constant number that was the same for all matches.

WC - 2 November: I started a basic GUI using javaFX, which I restarted multiple times as there are many ways of implementing a GUI in javaFX. Eventually, I found *scenebuilder* which made the process graphical and was far easier to use.

I started the report on adaptive strategies. As all the strategies besides the random ones were deterministic and there wasn't much variation in outcomes. So I wanted to expand the strategy pool a bit.

WC - 9 November: I refactored Round-Robin tournament to be more fair. I did this using the *FairGame* class, which took a pair of strategies and played multiple matches between the two each match had a random number of rounds but the total number of rounds was uniform across each pairing of strategies. This is discussed below in further detail.

I compared my current progress to my original plan. I was a week or so behind in some areas but had started programming differently and earlier so was ahead in regards to implementation.

I began research into the history and importance of the Iterated Prisoner's Dilemma.

WC - 16 November: I finished the GUI so you could play any of the select strategies in a tournament. Either a single elimination or round-robin. The results were printed in console.

I refactored some code structure, moving scoring system from the *match* class to the $payoffmatrix$ class. I also created factory classes to decouple the GUI from strategy and tournaments implementation.

I implemented an adaptive strategy which changed strategy based on all previous outcomes, similarly to *pavlov* but performed better in almost all scenarios.

WC - 23 November: I implemented a leader-board system which would be displayed onto the GUI. I tried to think of one leader-board that would work for both tournament types, but settled on creating a leader-board interface with two different implementations as the results from each

tournament were in a very different style.

I began collating all my reports into the Interim Report.

I implemented another adaptive strategy which takes into account the current score.

WC - 30 November: I focused on completing the Interim report and the presentation. I also went through the code adding comments where it felt lacking.

WC - 14 December: I went through the code and began planning how I would implement colluding strategies and genetic algorithms and began researching methods for genetic algorithms in the IPD.

WC - 11 January: I had fully implemented colluding strategies using a pawn and king system with a shared colluder class that would coordinate them. I began writing a report on them and continued researching to see if there was an alternative way to implement them.

WC - 25 January: I had found many articles on genetic algorithms and had written reports on their implementation and I created my own way of storing the Dna so that the first few rounds were not static, but dynamic by creating a list of lists where each sub-list referred to the size of the history.

WC - 1 February: I implemented my Dna system, candidate class and created a calculator class to convert the history to an address for the Dna. I began working on a system to evolve from one generation to another. I had multiple errors with off-by-one errors in indexing the Dna with loops that I took a long time to discover and delayed me by a few days.

WC - 1 February: The evolver class was implemented and could take an ordered list of candidates and produce the next generation. I had accidentally set the mutation rate to 10% without knowing. I implemented the natural class that would evolve a group of candidates relative to each other.

WC - 8 February: I began expanding the GUI to accommodate genetic algorithms and found a way of creating a graph which would display test data. I had to refactor the natural class to not be a single method that would play the tournament but to okay the tournament on instantiation and store the display data in multiple fields that could be queried by additional methods.

WC - 15 February: I refactored that natural class to be a child of a parent class called 'natural selector', in anticipation for implementing guided genetic algorithms so some methods could be shared between the two and so they could share the same methods of retrieving data so that the UI controller only needs access to the 'natural selector' class and not both. I discovered the issue where I had set the mutation rate to 10%.

WC - 22 February: I implemented guided genetic algorithms that would be trained against user specified strategies. I had to refactor the leaderboard classes to fix the data being displayed on the GUI. I completed my report on my implementation of genetic algorithms.

WC - 1 March: I started testing both types of genetic algorithms and collating my reports to produce the final report. I improved the GUI so it was easier to use by adding extra error messages

and creating help pop-ups when the user hovers over buttons.

WC - 8 March: I focused on the final report and began writing the self-evaluation sections and professional issues sections.

WC - 15 March: The final report was approaching completion and I began reviewing my code to make sure it was commented and written to the best of my ability.

A.4 Manual

The program was made using Java 1.8 and JavaFX, so no additional plugins should be required.

When running the program you will be met with this UI:

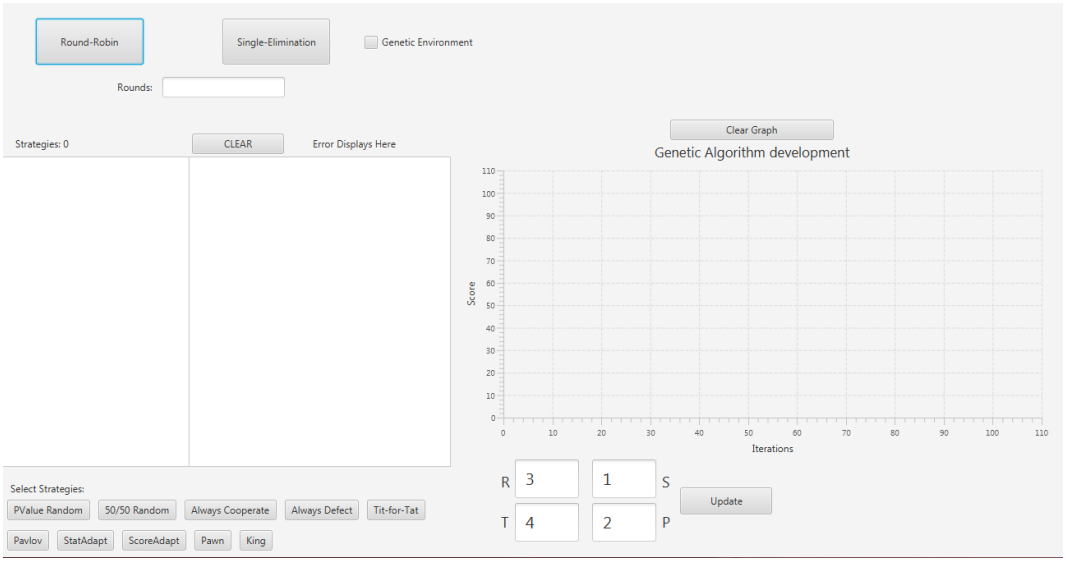

Figure 11: GUI Display

A.4.1 Standard Tournaments and Selecting Strategies

Firstly, by hovering over labels and buttons a prompt should inform you of their purpose.

The top two buttons are used to play tournaments once the strategies are selected from the buttons at the bottom.

Each strategy button correlates to the strategy they are named after, each time you click one, a correlating strategy will be added to the menu display on the left and the number of strategies will be displayed on the top left. The clear button above that will clear ALL selected strategies. THere is no way to remove a single strategy.

Next to the clear button, is an error display that will tell you an error message if something is incorrect.

Once you have selected strategies, you can either press Round Robin or single elimination, Round-Robin can be used with any number of strategies, single elimination has to have a power of two number of strategies.

You must enter a number of rounds, if you wish for it to be random then enter 0.

Once either is pressed, a leaderboard will be displayed on the right menu, scroll can be used to navigate vertically and shift-scroll can be used to navigate horizontally.

On the bottom left you can observe the payoff matrix for the Prisoner's Dilemma, by changing the numbers in the text-fields and pressing 'update', the payoff matrix will be updated. If you want to return to the original matrix and have forgotten the values, the program must be restarted.

The graph is not used for the standard tournaments.

A.4.2 Using the Genetic Environment

A checkbox labelled as 'Genetic Environment' should be in the top center. When checked the UI will swap to the genetic environment.

The two tournament buttons will swap to Natural and Guided, which function as explained in the report.

Additional text boxed for the number of iterations and the population will appear. Ensure that the number of rounds is not set to 0 when using the genetic environment as the graph will not be indicative of the true relationships.

The number of iterations should be large, the largest number I have done is 10000. The populations size must be a multiple of four, if this number is too large it will use a lot of memory. A population of around 100 should suffice.

Natural can be pressed without selecting any additional strategies, Guided requires you select additional strategies, as done in the standard environment.

The results of the most successful iteration will appear in the leaderboard menu, and the scores of the best performers for each iteration will be displayed on the graph. The graph is normalised to the minimum and maximum values.

This can be repeated and a new plot will be placed on the same graph. If you wish to clear the graph, the 'clear graph' button should be pressed.

A.5 Further Tests

Further testing results and test outputs will be included in a separate folder.