Pedro Barbosa Duarte dos Santos Freire Full Unit Project 2017

Supervisor: Kostas Stathis

Cooperative Strategies in Multi-agent Systems
Full Unit Interim Project Report

Table of Contents

1 Introduction

1.1 Motivation

One of the key factors for humanity to have reached the top of the food chain was cooperation. It can also be examined in nature how animals that live or hunt in a group tend to prosper more than animals that strike out on their own. Wolves are predators that are able survive in a world with much larger and more dangerous predators simply due to their pack mentality. Tigers, though one of the strongest predators on Earth, have become endangered due to their solitary nature. Human are seemingly one of the most physically inept creatures in regard to their size, boasting no anatomical self-defence or self-preservation traits. Yet, humanity rose to the top by cooperating and building large civilizations and instruments to make-up for the physical ineptitude.

However, game theory and, more specifically, the Prisoner's Dilemma states that rational decision making obliges an individual not to cooperate. It states that acting selfishly will ensure that an individual in never get the worst end of a deal and may sometimes get the best end of a deal. But somewhere along the line entities began to cooperate, which turned out to be more prosperous than acting selfishly. Yet, blind cooperation can also be fruitless, which is why strategies were developed in order to maximise cooperation without risking heavy loss.

1.2 Aim & Objective

The goal of this project is to find the best strategy for playing the Prisoner's Dilemma and, hence, the best strategy for cooperation between two arbitrary entities. In order to achieve a conclusion, ten distinct strategies will be analysed on how they interact with one another, and how they thrive in multiple environments. A multi-agent system will be built where agents are attributed a strategy and play the Prisoner's Dilemma in an attempt to obtain the best outcome in multiple environments. The system will consist of three steps. The first step is to create a program where an agent can play against another agent to examine how each strategy prospers against each other strategy. The second step is to create a tournament where multiple agents and strategies are selected to play against each and every other agent in the tournament. The final is step is to create an environment where agents have the free will to choose who to play against based on past experience or communication between agents. The most prosperous strategy in the final program will be identified as the best cooperative strategy for the Prisoner's Dilemma.

2 Background Work

2.1 Prisoner's Dilemma

The Prisoner's Dilemma is one of the games analysed in game theory in which two individuals are set against each other and forced into defecting or cooperating with the opponent, while remaining ignorant to the other player's choice. The game was first formalized into the format of a prison sentence, in which two criminals were placed in separate interrogation room's and were forced to either stay silent or blame the other for the crime, in order to determine the length of their prison sentence, hence the name, Prisoner's Dilemma. However, the game has since then been used in many other mediums such as science and business. Therefore, the game has adopted a more universal measurement by adopting the choices of cooperating and defecting, and counting utility/points. Using the latter method of measurement, the outcomes for each possible choice are determined by a standard matrix (Figure 1).

		Player 2	
		Cooperate	**Defect**
Player 1	**Cooperate**	3 Reward 3	Temptation 5 0 Sucker's Pay-off
	Defect	Temptation 5 0 Sucker's Pay-off	1 Punishment 1

Figure 1 – $T > R > P > S$ *and* $R > (S + T)/2$

The matrix shows that there are four possible outcomes. The first outcome is when both players cooperate and are each rewarded with three points; this is called the "reward" because it is the best global outcome. The second and third outcome is when one player cooperates and the other defect, in which case the player that defects will get five points and the player that cooperates will get no points; this is called the "temptation" and the "sucker's payoff", respectively, because it is the best possible outcome for one player and the worst possible outcome for the other. The final outcome is when both players defect and are only rewarded with one point; this is called the "punishment" because it is the worst global outcome.

Upon analysing the different outcomes it's possible to see why it proves to be a dilemma. In order to get the best global outcome, a player has to risk getting the worst individual outcome, and in order to get the best individual outcome it risks getting the worst global outcome. If the game were to be played by two completely rational players that only act on self-interest, the rational choice is to always defect since it will yield the best possible outcome (temptation) and will never yield the worst possible outcome (sucker's payoff). However, if both players are rational and always defect, they will never reap from the temptation and will, instead, always receive the worst possible global outcome, which is why this becomes a dilemma.

2.2 Iterated Prisoner's Dilemma

A Prisoner's Dilemma game usually plays out only once, hence, it will always be a dilemma and there will never be best option. However, the game can also be played multiple times by the same individuals, this is called the Iterated Prisoner's Dilemma. Though it is essentially the same game, this version has very different applications and solutions. Contrary to the original mode, an iterated version of the game allows a player to learn the behavioural tendencies of the opponent or environment. While the goal is still to obtain the most amount of utility by the end, the risk of losing an iteration becomes a lot less significant, which allows players to experiment with cooperation and study the opponent's response rather than playing it safe by defecting. This allows a number of strategies to be developed in order to optimize a player's utility whilst avoiding being exploited by selfish players.

2.3 Strategies

There are a number of standard strategies that have been developed for the Iterated Prisoner's Dilemma. This project will focus on analysing ten distinct strategies. In order to measure the effectiveness of each strategy three characteristics will be evaluated: robustness, stability and initial viability. Robustness takes into account the ability for the strategy to thrive in an environment with many additional strategies. Stability examines the ability for the strategy to resist being "invaded" by another strategy. A strategy is considered "invaded" if its final result is lower than the result of being "rewarded" (i.e. cooperate/cooperate) every round, and the result of the opponent is higher. Initial viability is the ability for the strategy to gain an initial foothold in an environment without being exploited or discriminated.

2.3.1 Always Defect

Always Defect is the rational strategy of a normal Prisoner's Dilemma. It will possibly yield the best outcome and will never yield the worst outcome. When this strategy is put against any other opponent, it will never yield a worst result than its opponent, therefore, it naturally has the best foothold on any environment. However, it also fails to maximise the total utility it gains in an environment with robust and stable strategies, as it lacks any willingness to cooperate with the opponent.

2.3.2 Always Cooperate

Always Cooperate is an interesting strategy as it's hypothetically the approach every player should have to ensure the best global distribution, however, it is the strategically the worst strategy. The robustness of this strategy is limited solely to environments with cooperative strategies and its stability is null as it lacks any form of defending itself from invasion by another strategy. Most strategies tend to build around this objective, but also seek to create some sort of defence against invasive strategies.

2.3.4 Tit for Tat

Tit for Tat is a strategy where the player begins by cooperating and then copying the opponent's previous choice, called cooperation by reciprocity. The strategy has historically been held as one of the most robust and stable strategies as it will always immediately reward the opponent for being cooperative and immediately punish them for defecting. The only way of guaranteeing a result greater than Tit for Tat is to always defect since the first move in this strategy is to cooperate. But if the number of rounds played between each player is large enough the difference between each result is almost insignificant. This strategy has a version that starts by defecting first (Suspicious Tit for Tat). Though it impedes fully selfish players from having a better result, it proves to be more ineffective in most circumstances as it will prevent any possibility of fully cooperating with an opponent that also adopts some sort of reciprocal strategy.

The main issue with this strategy is when there is a chance of error (i.e. defects when it should cooperate or vice versa). For example, if this were to happen in a game between these same strategies, the players would end up in a "death spiral" where they would alternate between "temptation" and "sucker's payoff" as opposed to always cooperating.

2.3.5 Tit for Two Tats

Tit for Two Tats is a version of Tit for Tat, but it's more forgiving of defection as it requires two sequential defection to begin defecting. On the other hand it is also more suspicious of cooperation if the opposing strategy were to start by defecting. The strategy was mainly created to avoid the "death spiral" of the Tit for Tat, however it is more susceptible to uncooperative strategy that can take advantage of its more forgiving nature.

2.3.6 Grudger

Grudger is a strategy that begins by cooperating and does so until the opponent defects, after which it will always defect. Though this strategy may seem very prone to defection, it's actually a cooperative strategy. The Grudger never actively tries to compromise cooperation, it just reciprocates. However, unlike Tit for Tat, its reciprocation will not change for the rest of the game. Therefore, though the strategy is stable, it lacks the robustness of more flexible strategies such as Tit for Tat.

2.3.7 Soft Grudger

Soft Grudger is another version of the Grudger that seeks to be more forgiving. Like its parent strategy, it starts by cooperating and reciprocates when the opponent defects. Instead of defecting until the end, it defects a constant four times then cooperates twice, after which it will begin reciprocating. Though this strategy is more forgiving of single defection, it can also be exploited by invasive strategies due to its fixed cooperation. Typically, any strategy that has constant/non-reciprocating cooperation is vulnerable to strategies that try to achieve "temptation".

2.3.8 Gradual

Gradual is similar to the Soft Grudger strategy. Instead of having a constant defection stream, it defects the amount of times the opponent has defected throughout the whole game. This ensures the strategy is more forgiving to cooperative opponents, and punishes uncooperative opponents. Yet, much like the Soft Grudger, it always cooperates twice after ending its defecting iterations. This allows strategies to exploit "temptation", however it also adds defections the next reciprocation. Therefore, this strategy is, theoretically, equally stable to the Soft Grudger, but more robust.

2.3.9 Pavlov

Pavlov is a strategy that attempts to fix the fault in Tit for Tat. The problem with Tit for Tat is that it would enter a "death spiral" if there was a defection. Pavlov uses a strategy also known as win-stay, lose-shift, meaning that when it gains "reward" or "temptation" it will repeat the choice, otherwise change. This will avoid the "death spiral" in a game between two Pavlov strategies. This strategy is believed to be as robust as Tit for Tat, but not as stable since it will keep taking the "sucker's payoff" when the opponent always defects. Since no outcome is optimal when the opponent defects, this strategy will keep switching between cooperation and defection.

2.3.10 Adaptive

Adaptive is the most exploitative strategy on this list. It begins by cooperating six times, defecting five times. From there it will choose the option that has yielded the best average outcome. The average is recalculated after every iteration. Unlike all the other strategies, this strategy will favour defecting for "temptation" over cooperating. For this reason, the Adaptive strategy can be seen as one of the most flexible as it can exploit vulnerabilities of unstable strategies while still being able to fully cooperate. This is most effective on long iterations where it can analyse the behaviour of the opponent.

2.3.3 Random

Random is not a strategy, but it is a good method of testing the robustness of a real strategy. The core of a strategy is best tested for robustness when it cannot learn or anticipate the opponent's choices. If a strategy fails to have a better result than a random methodology, it can be concluded that it is not robust enough to thrive in any kind of uncooperative environment.

3 Framework

3.1 Program Architecture

The program will have three sub-programs: one vs one, tournament and environment. The one vs one program is used to play a game of the Prisoner's Dilemma between two agents, over an arbitrary number of rounds. Its purpose is mainly used to test whether a strategy is functioning properly and to test which strategies are most compatible.

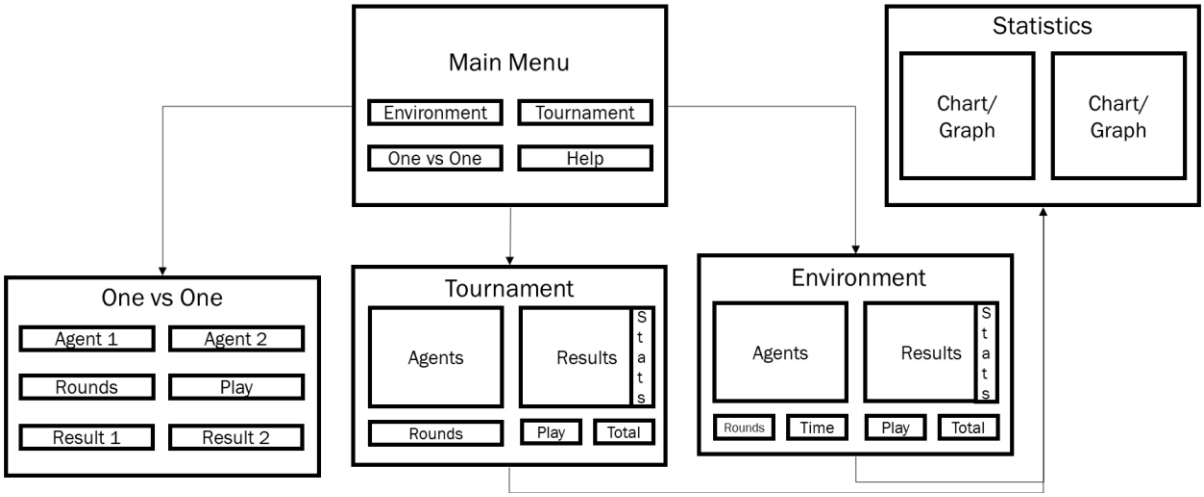
The tournament program is used to have up to ten agents, with individual strategies, play against each and every other agent twice for an arbitrary number of rounds. It will display the result of each individual agent and the total utility gained over all agents. The combined utility of all agents helps determine what set of strategies are more cooperative, even if their individual scores are somewhat mediocre. With this program, agents have a fixed strategy, are forced to play every other agent, and have no chance of error. The program can start to identify what strategies are most robust and most stable. A set of statistics will be developed in order to better evaluate each strategy. Each agent will be able to display their own table for these statistics.

The environment program is similar to the tournament, except agents may choose their opponent based on past experiences. The environment accepts a list of agents and runs the environment for a certain amount of time. During this time agents look for suitable opponents and run the Prisoner's Dilemma for an arbitrary number of iterations. Agents are not allowed to keep playing against the same agent. After each game the user interface should display the current utility of each agent. At the end of the duration, each agent will have several statistics available for display.

The program is built with a Model View Controller (MVC) architecture. This architecture ensures the main functionality of the program is decoupled from the Graphical User Interface. Decoupling helps mitigate errors with every update, and facilitates code debugging by allowing the source of a bug to be easily identified.

The program is divided into three main packages: model, view and controller. The model package contains the classes managing each sub-program. The one vs one and tournament managers only have one core function that is called by the respective controller. The environment manager will be returning information after each iteration over its timespan. Therefore, it will be observable by the respective controller, notifying it every time an iteration is completed so the output can be displayed on the user interface. The model will also have a class to create agent objects. The model has a sub package, strategies, containing the object class for a strategy along with its child classes for each unique strategy. The view package contains, for each sub-program, the respective JavaFX file along with the view class used to initialize the GUI. The controller package will contain the controllers of each sub-program. The controllers manage the interaction between the GUI and the programs main functionality (i.e. the model). It accepts input from the GUI, calls the appropriate processes in the model, and finally sends the output back to the GUI.

3.2 Program Design



The program will begin with a main menu. From the main menu a new window can be opened to play either the one vs one, tournament or environment, or a help window that contains instructions. The one vs one window allows two players to play. The tournament and environment windows allow multiple agents to play. While the tournament has a fixed number of games, the environment allow a timespan to be defined. Each agent can open a window containing its unique statistics.

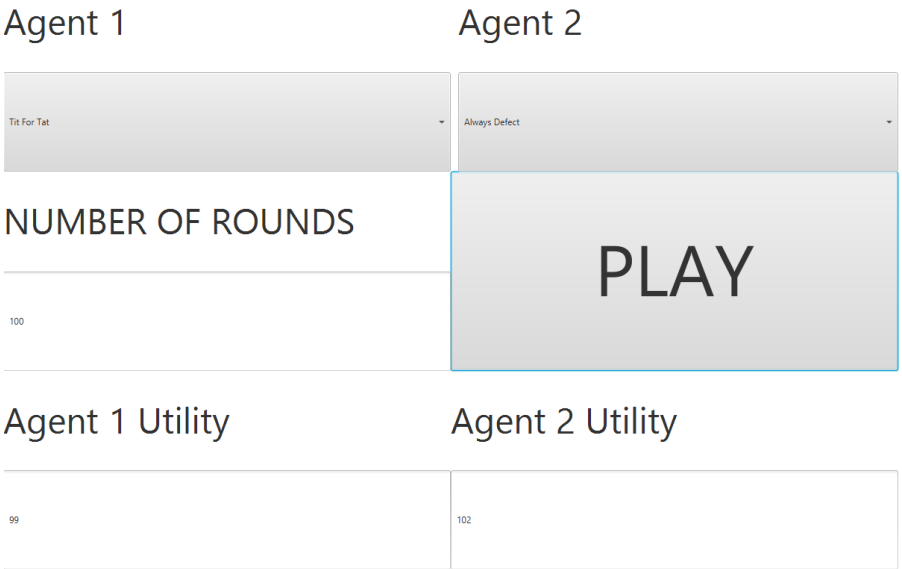
4 Implementation

4.1 Model View Controller

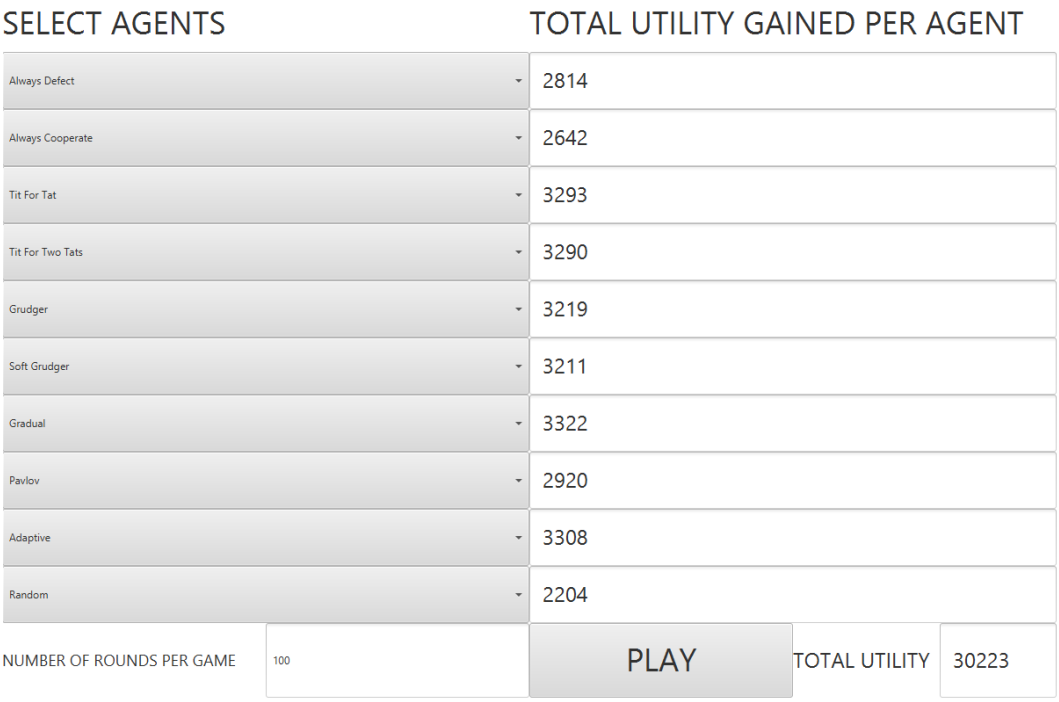
4.1.1 View

The user interface for this program was designed with JavaFX. The appearance of the GUI is fully designed with Scene Builder, a graphical tool to create .fxml files. This components of this file is then initialised by a Java class, so it can interact with the rest of the program.

The one vs one program contains two choice boxes, one for each agent. The boxes drop down the available strategies. There are also three text fields. One text field requires the user to submit the number of rounds for the game. The other text fields will display the utility each agent acquired. The final component is a button to start the game, which is handled by the controller.



The tournament program contains ten choice boxes, which are initialised into an array. Likewise to the one vs one program, the choice boxes drop down a menu with the available strategies. There are also ten text fields that display the utility of its respective agent after a game. Another text field requires user input to determine the number of rounds per game. The last text field displays the combined amount of utility gained. Finally there is a button to start the game, which is handled by the controller.



4.1.2 Controller

The controller is the class that manages the interaction between the GUI and the main program behaviour. The controllers for both the one vs one and the tournament are very similar in their structure. Each of them contain an object for the view and the manager (model). The constructor receives instances of the view and manager, adds itself as a handler for the components in the view, and adds itself as an observer of the game manager.

```
public TournamentController(TournamentManager manager, TournamentView view) {
  this.manager = manager;
  this.view = view;
  view.addHandlers(this);
  manager.addObserver(this);
}
```

The controller in both programs only has one method called handle. This method detects when the “play” button was pressed. Once the event occurs, the game manager is called to run the game. The controller sends the manager a list of strategies gathered from the view and the number of rounds for each game. Once the game has been processed in the model, the controller loops through the list of agents to gather their utility and display it in the respective text field. Inside the loop, the program checks if there is a null value in the array, in case the user did not select some of the available ten agents. After the loop this method also set the value for the total utility gained. The only difference between the two controllers is that the tournament loops through an array of agents, whilst the other uses two pre-set agents/strategies.

Included in the controller package is the driver class (i.e. the main method). The driver calls the loader class which then starts up the application/GUI. The loader creates the GUI and links it to its respective controller along with the model. The display uses a StackPane so it can save the parent display, when windows are opened from other windows. This help individual windows interact properly without needing to keep all of them open. On the current build it's not noticeable as it lacks interaction between the two programs. But it will make the interaction with the main menu and statistics windows more fluid once they are implemented.

4.1.3 Model

The model is the most complex package in the program. It contains the central manager class, but it also contains the object classes for agents, strategies and the game matrix. The manager is similar in both programs, the only difference being the array used in the tournament as opposed to only two variables in the one vs one. The manager has one central method `rungame()` that processes an entire game of the Prisoner's Dilemma. Since the controller sends the strategies in string format, the method begins by initialising each agent with the respective strategy. The tournament manager creates a list of agents and initialises them in a loop. The agents are then told about the opponent. Once the agents know their opponent the manager runs a loop over an iteration of the Prisoner's Dilemma for the given number of rounds. In an iteration: the agents make a choice, the combination is evaluated, the utility of each agent is incremented and the choice they made is saved.

In the tournament version of this method, there are nested loops that go through the entire list of agents for each agent in the list. The loop over the number of rounds is nested beneath those two. The code verifies that the agent does not play against itself and it resets the various conditions the agent may have acquired from the previous game before moving to a different opponent.

```
public void runGame(String[] stratList, int numOfRounds) {
  agentList = new Agent[stratList.length];
  for (int num = 0; num < agentList.length; num++) {
    agentList[num] = stringToAgent(stratList[num]);
  }
  for (Agent agent1 : agentList) {
    for (Agent agent2 : agentList) {
      if (agent1 != null && agent2 != null && agent1 != agent2) {
        agent1.setOpponent(agent2);
        agent2.setOpponent(agent1);
        for (int round = 1; round <= numOfRounds; round++) {
          agent1.getStrat().choose();
          agent2.getStrat().choose();
          matrix.evaluate(agent1.getStrat().getCurrChoice(), agent2.getStrat().getCurrChoice());
          matrix.printRound();
          agent1.incUtility(matrix.getResult1());
          agent2.incUtility(matrix.getResult2());
          agent1.getStrat().setLastChoice();
          agent2.getStrat().setLastChoice();
        }
        agent1.reset();
        agent2.reset();
        System.out.println();
      }
    }
  }
}
```

The agents are initialised in a separate method stringToAgent(). This method takes the strategy name and runs it through a switch statement to identify the correct object to initialise. For example:

```
case "Tit For Tat":
  return new Agent(new TitForTat());
```

The result of each iteration is evaluated by a separate class `GameMatrix`, which contains the method `evaluate()`. This method receives two string signifying the choices made by each agent. It appends the strings and runs it through a switch statement to determine the outcome of each agent. The result for each agent is stored in separate fields that the manager can access. For example:

```
case "DD":
  result1 = 1;
  result2 = 1;
  break;
```

4.2 Object Oriented Programming

The program uses Object Oriented Programming to handle individual agents and strategies.

4.2.1 Agent

The Agent object consists solely of a strategy and the total amount of utility gained. The constructor just receives and initialises a strategy and initialises utility to zero. It has a function to increment utility, to be called after each iteration. A function to set its opponent since most strategies need to know the opponents previous move. Finally it has a function to reset the status of the strategy in order to have a fresh start against the next opponent. Otherwise a strategy like the Grudger would never cooperate with any other agent if the first agent defected against it.

In the current build there is little reason to have a separate object for the agent and the strategy. However, in the future an agent should be able to assess its environment and have free will. A strategy object does not need any of those functionalities.

4.2.2 Strategy

The Strategy object is parent to all ten unique strategies. It has fields for the opponent strategy, the current choice it will make, and the last choice it made since a lot of strategies use reciprocation. The class has the function `choose()`, which changes the current choice. This function is overwritten by every unique strategy. The function is usually composed of several conditional statements based on the opponent's last choice on strategies that reciprocate (e.g. Tit for Tat). Other strategies use conditional statements based on the result of the last iteration (e.g. Pavlov). Some strategies have unique variables that keep track of the number of times it should cooperate/defect (e.g. Gradual).

This class also contains a function to set its opponent, which it receives from the Agent object. It holds a function that changes the current choice into the last made choice, which is called at the end of each iteration of the game. Finally, it has a function that resets all the variables (except the opponent) to their default values. This function is overwritten by strategies that have unique variables, such as the adaptive method that keeps track of all its choices and the average score of each choice. In these cases the function will also set unique variables to their default value.

5 Project Diary

- *Pedro Freire* ***1:22 pm on September 30, 2017***
  This past week I gathered the information from a couple of sources that explained the evolution of cooperation in relation to the prisoner's dilemma, and looked into a number of common strategies used in an iterated prisoner's dilemma. With this information I decided the aim of the project was to find the most effective strategies in certain environments, and built my project plan around this idea.

- *Pedro Freire* ***9:06 pm on October 11, 2017***
  This past week I sketched a brief design of each agent and the tournament build they will use to play against each other. This design is intended to be the basic version of the program, so I made it in a style I currently know how to produce. I also laid down some general ideas of how I could improve the program, but these concepts may further research into multi-agent programming. I also briefly worked on the introduction of the project (problem and objectives).

- *Pedro Freire* ***4:06 pm on October 25, 2017***
  The past couple of weeks I have delved further into the each individual strategy and developed functions to describe the robustness of each strategy. Further along the line I started developing the initial step of the program. I have added a menu that allows a user to choose two strategies and play them against each other and output their results. This functionality will allow me to test if each strategy is functioning as it should be.

- *Pedro Freire* ***10:34 pm on November 20, 2017***
  The past few weeks, after having some difficulties getting the GUI of the program to function correctly, I have finished the graphical display that allows two agents to play against each other. I also developed the second stage of the program, which is to create a tournament in which agents can play against each other agent involved. For now it is fixed to 10 agents of less but this may change in the future. I also updated the design of the program as I ran into issues that were not previously foreseen, such as the need to create an additional menu and a class that loads each individual view.

- *Pedro Freire* ***8:24 pm on December 1, 2017***
  This past week I have added all the remaining strategies to the tournament and refactored some less efficient code in the tournament manager. After having the initial stage of the program complete, I began to put past research and designs together to form the interim report. Once the planning phase was all written down on the report I began to detail the implementation of the program, and final finish the report for delivery.

6 Bibliography

Davis, Wayne. *Prisoner's Dilemma Strategies*. n.d. http://www.iterated-prisoners-dilemma.net/prisoners-dilemma-strategies.shtml. 2017.

Prisoner's Dilemma. n.d. https://www.investopedia.com/terms/p/prisoners-dilemma.asp. 2017.

Prisoner's Dilemma. n.d. https://en.wikipedia.org/wiki/Prisoner's_dilemma.

Robert Axelrod, William D. Hamilton. *The Evolution of Cooperation*. Basic Books, 1981.

Tit for Tat. n.d. https://en.wikipedia.org/wiki/Tit_for_tat. 2017.