Final Year Project Report

Full Unit – Final Report

Cooperative Strategies in a Multi-Agent System

Pedro Freire

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Kostas Stathis

Department of Computer Science

Royal Holloway, University of London

March 28, 2018

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count:

15135

Student Name:

Pedro Freire

Date of Submission:

28/03/2018

Signature:

Table of Contents

Abstract

Cooperation is an essential quality in many aspects of society, yet the prisoner's dilemma teaches us that being selfish is the safest method of ensuring we gain the most utility on an individual scale. However, it also tells us that cooperating will always render the most utility on a global scale. This creates a problem because in order for individuals to use the safest (and possible most rewarding) strategy they must opt out from taking the most publically rewarding option.

The purpose of this project is to identify the best cooperative strategy that will guarantee the best public reward, whilst keeping individual risk to a minimum. In order to reach this objective we will find the most effective strategies under various forced environments. Environments will be composed of many agents which individual strategies that all have a specific degree of cooperation. Strategies will be tested in environments with a high degree of cooperation and a low degree of cooperation. By the end of the project we should find the most effective strategy over all types of environments.

1 Introduction

1.1 Motivation

One of the key factors for humanity to have reached the top of the food chain is 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 will never get the worst end of a deal and may sometimes get the best outcome. 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 program will consist of three modules. 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. In order to find the most prosperous strategy, agents will be set in a tournament and forced to play against other agents in a multitude of different environments. The strategy that demonstrates the highest utility gain among these tests, will be identified as the best strategy for the Prisoner's Dilemma.

1.3 Structure of Thesis

The report is divided into seven chapters. The first chapter will introduce the problem that will be tackled and provided a brief insight into how this project will attempt to solve the problem. The second chapter will cover the research and background work done prior to the development of the program. The third chapter will discuss the programs design and the framework in which it will be built. This chapter will also discuss how the program evolved over time in terms of structure, refactoring and visual allure. The fourth chapter will discuss the specific details about the features of the program, how the work and how they interact. The fifth chapter will demonstrate two different sets of tests that were run with the program in order to obtain a concrete answer to the initial problem. The sixth chapter will summarize the whole process involved in the creation of this project, conclude a suitable answer for the problem, and discuss how the program could be improved. The final chapter will discuss how the project might overlap on some ethical issues regarding artificial intelligence.

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.

2.3.3 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.

2.3.4 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.

2.3.5 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.

2.3.6 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".

2.3.7 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.

2.3.8 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.

2.3.9 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.

3 Framework

3.1 Program Architecture

The program will consists of three main modules: One V One, Tournament and Environment. The One V One module will be used to play a game of the Prisoner's Dilemma between two agents, over an arbitrary number of rounds. Its purpose is mainly to test whether a strategy is functioning properly and to test which strategies are most compatible.

The tournament module will be allow a list of agents with individual strategies to be generated, and allow every agent to play against each and every other agent for an arbitrary number of rounds. The GUI 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 is more profitable, even if their individual result is subpar. Each agent will also have an option to open a new window with more detailed statistics that will be discussed below. The original design of this module only allowed the user to choose a maximum of ten agents, but the end product allows for any number of agents.

The Environment module 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 games specified by the user. For the specified number of games, agents play against each other with the option to skip over another agent. After all the games have been played, the user should be prompted with the results for every agent. Each agent will also have the option to display more detailed, personal, statistics on the last run of the environment.

The statistics should open on a new small window. The window will display a pie chart that indicates how many games the agent has played against every other agent, and a second pie chart that indicates how much utility the agent has gained in matches against every other agent.

The program will be built on a Model View Controller (MVC) architecture. MVC divides programs into three parts. The model is the class/set of classes that provides all of the algorithmic computation; the view is the class that initialises the GUI of the program; and the controller is the class that listens for events in the GUI and calls functions in the model in order to provide the user with the full functionality of the program. This architecture ensures the main functionality of the program is decoupled from the GUI. Decoupling helps mitigate errors with every update to the code, and facilitates code debugging by allowing easy identification of the source of a bug. Finally, this architecture will help external programmers understand the code more easily.

The program was originally to be divided into three main packages: model, view and controller. The model package would contain the classes managing each module. This package would also include the class for agent objects and a sub-package, strategies, containing the object class for a strategy along with its child classes for each unique strategy. The view package would contain the view class and respective JavaFX file for each of the modules. The controller package would contain the controllers of each module.

However, the packaging of the program underwent changes during development for the sake of implementing a more coherent structure. As the code was developed and new classes were added, it became apparent that the three packages from the design were not enough to separate all the individual features. Instead, a package was created for each of the modules (including the main menu and statistics), containing the model and controller classes. Another package, driver, was given the three classes responsible for initializing the program. Finally, another package was created to contain all the objects and classes that assist in managing those objects. The package contains the agent and statistics objects, and a sub package that contains the strategy object and its ten child objects. The only package that was kept was view, which contains the view classes of every module and their respective fxml file. This package was kept so it is be easier to load each GUI.

3.2 Program Design

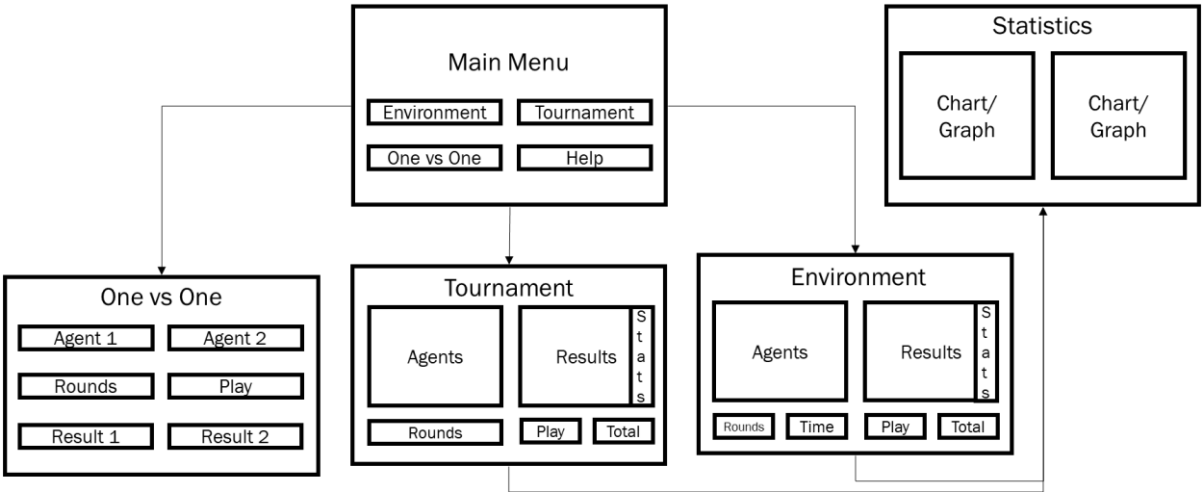
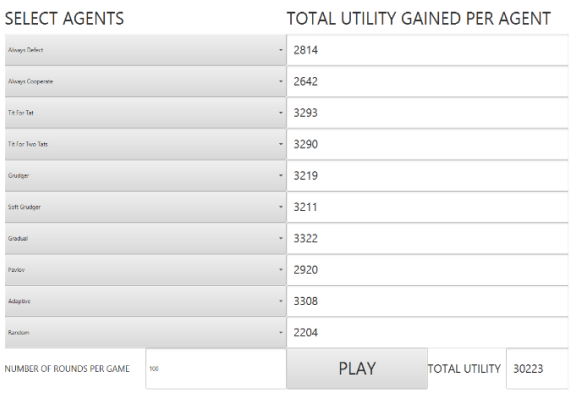


Figure 1 – Initial design for the program.

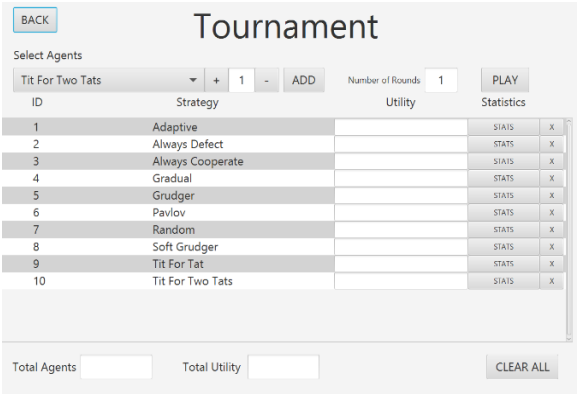
The program will begin by opening a main menu. From the main menu the user can choose to open one of the available game modes, One V One, Tournament or Environment. The One V One screen allows the user to select two agents and a number of rounds, press play and receive the results. The Tournament and Environment windows allow multiple agents to play. The design of these game modes underwent some changes through out development.

The original design of the GUI for the tournament was crude and uninviting compared the end product. It had too many choice boxes and the positioning of the components was suboptimal because it asked the user to submit information on the left and move down the list. The second version asks the user to submit inputs at the top of the screen, from left to right which is more conventional. Furthermore, it displays all of the output at the bottom of the screen. The new GUI allows agents to be distinguished more easily with an ID number, and has the added feature to open each agent's unique statistics.

Before:



After:



The GUI for the environment underwent the same evolution, with only a single additional input text field for the number of games placed next to the right of the number of rounds text field.

4 Implementation

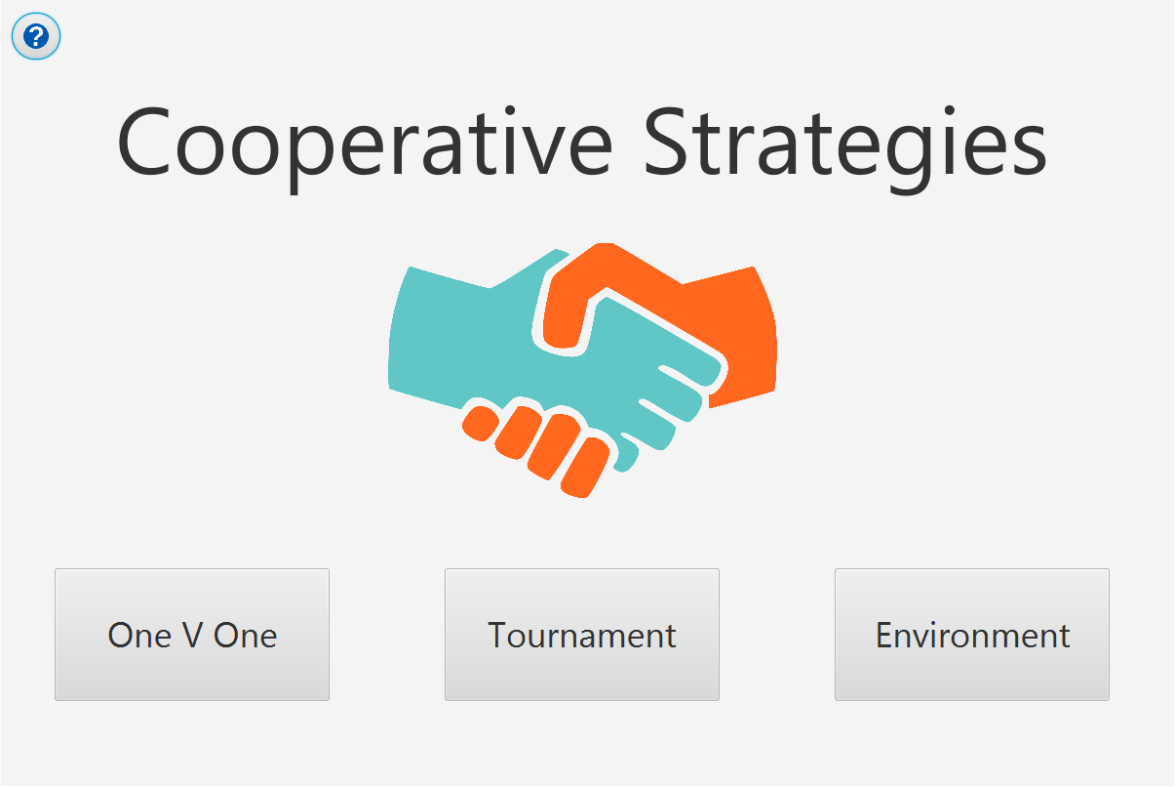
4.1 Model View Controller

The program was implemented with an MVC architecture in order to reduce coupling between different functionalities in the program. The program is divided into three main modules: One V One, Tournament and Environment. The main modules contain most of the functionality of the program, however, it also features to additional modules of reduced dimension: Main Menu and Statistics. These modules have minimal functionality and are used to either structure the program or complement the functionality of the main modules.

As the MVC architecture dictates, each module contains a Model class (called Manager in the program) that is responsible for evaluating all the algorithmic functionality required for the program. Each module then contains a View class that displays the GUI. Finally, each module contains a Controller class that manages the interaction between the Model and the View.

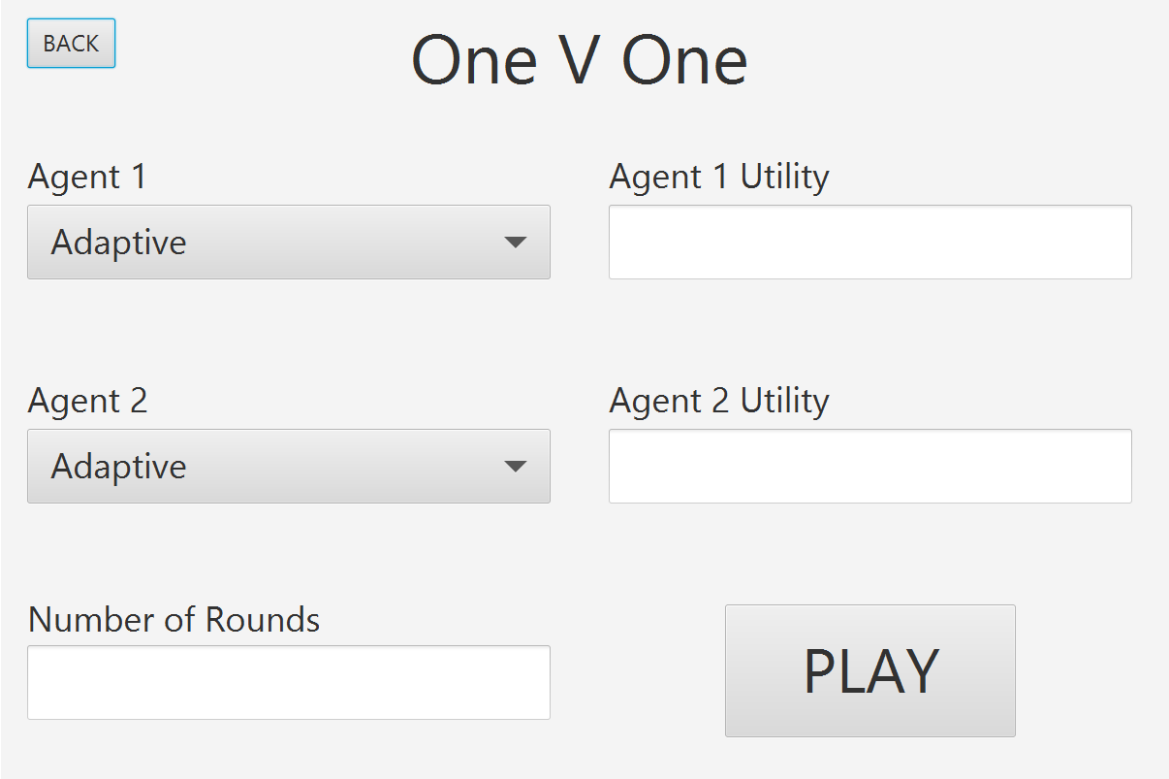
The user interface for this program was designed with JavaFX. The appearance of the GUI is fully designed with Scene Builder, a graphical tool to create .fxml files. The components of this file is then initialised by a Java class (i.e. View), so it can interact with the rest of the program.

4.1.1 Main Menu Module



The Main Menu program contains displays the title of the program and displays three buttons that will take the user to each of the three separate programs/game modes. It also features a help button at the top that will explain the general purpose of the program and explain, specifically, how each of the game modes work. The controller class of this module simply detects button clicks and loads the respective game mode after the user presses one of the buttons. The controller will also spawn a set of dialog boxes with the necessary explanation for the program and each individual game mode. This is the only module that does not feature a Model class since it does not require any arithmetic evaluation.

4.1.2 One V One Module



The One V One module is used to play the Prisoner's Dilemma between two unique agents for an arbitrary number of rounds. The view contains two choice boxes, one for each agent. Pressing the choice boxes will display all the available strategies that the agent can adopt. The user can choose any strategy in order to assign it to the agent. Beside each of the choice boxes, the view displays a text field. Upon iterating through one game, these text fields will display the total utility gained for the respective agent. The view also features a text field labelled "Number of Rounds", where the user can input a valid integer in order to determine how many times the agents will iterate over the course of one game. At the bottom of the screen there is a "PLAY" button which will initiate a game between the two agents as long as there is a valid input for the number of rounds. The final component in this module's view is the "BACK" button which is used to take the user back to the main menu.

The functionality of the GUI in this module is handled by an independent controller class. The controller is the only class that handles and uses functionality from its respective view and model. The controller has three fields: the view, the manager (i.e. the model) and the loader (See 4.3.1). The controller contains a constructor that takes a manager object and a view object, and initializes them to its respective field. The constructor will also add the listeners for the buttons in the view.

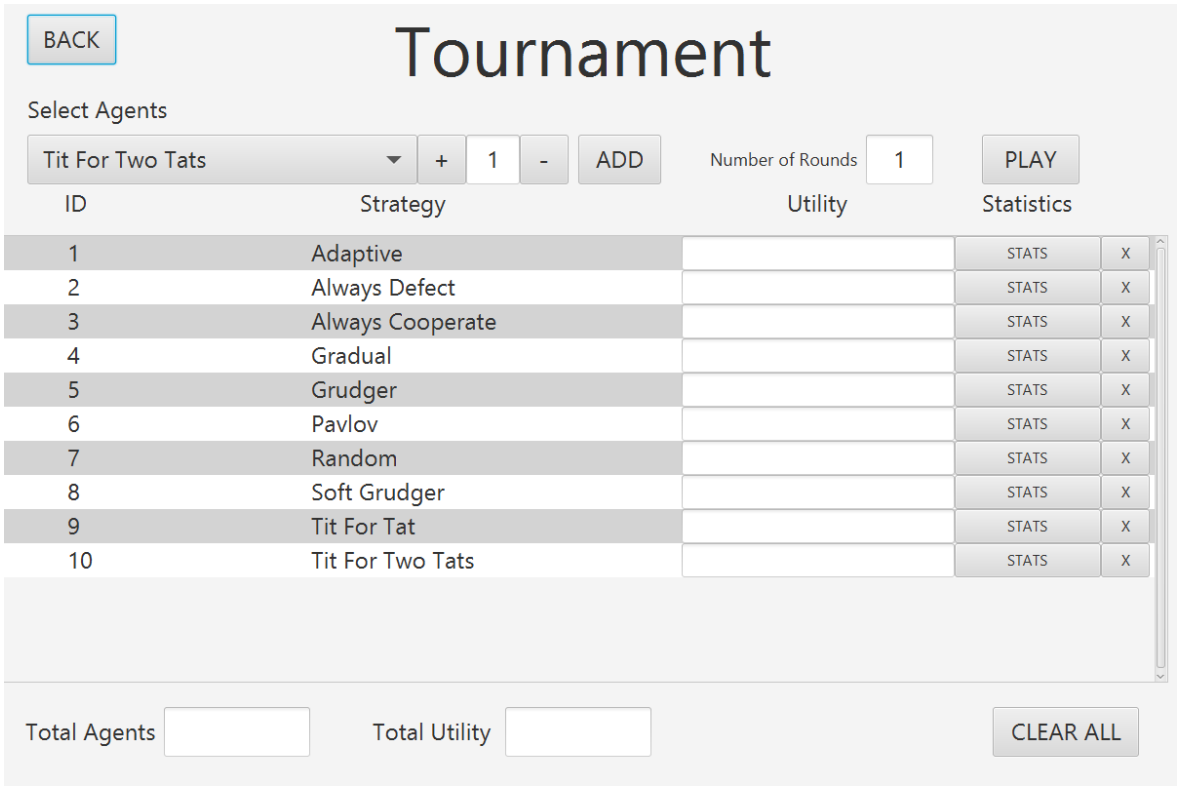
The controller only has one method which listens for events in either of the two buttons. If the "PLAY" button is pressed, it will call the method `runGame()` inside the manager. The method takes three parameters: agent 1, agent 2 and the number of rounds. After the manager evaluates the result of the game, the controller will retrieve the output from the manager and update the respective text fields in the view with the final result. If the user presses the "BACK" button, the controller will call the loader class in order to change displayed screen back to the main menu. Changing screens will overwrite the current stage, therefore, preventing many windows from open concurrently. Any changes made to the current view will also be discarded upon returning to the main menu.

The manager class is where the main functionality of the program is evaluated. It contains four fields: the agent manager (See 4.3.2), the game matrix (See 4.3.3), agent 1 and agent 2. The agent manager and the game matrix are instantly initialised, but the agents are only initialised when a game is run. The model then feature three methods. The two accessor methods for the agents, and the method that runs a game of the Prisoner's Dilemma.

The method `runGame()` takes three parameters: the strategy for agent 1, the strategy for agent 2, and the number of rounds for the game. The number of rounds is received as an integer and the strategies are received as a string. The first step in the method is to translate the string value of the strategy into and an Agent object (see 4.2.1). The method `stringToAgent()` is called from the agent manager. This method will determine which strategy was attributed to the agent and return an initialised agent object with said strategy attributed to it also as an object. The manager then checks whether the agents have been successfully initialised, otherwise it will return an error message.

After checking for possible errors, the function makes an iteration over the Prisoner's Dilemma. It asks the agent to choose their next move, either "D" (defect) or "C" (cooperate). After the agent decides on its choice, the manager will call the `evaluate()` function in the game matrix. This function will receive the choices from both agents and run them through the matrix for the Prisoner's Dilemma (See 2.1). The manager will retrieve the result for each of the agents inside the game matrix and increment it to the utility each agent has gathered throughout the game. Finally, tells the agents that the iteration is over so they can set memorize all the information they need in order to run their strategy's algorithm for its next choice. This whole process is iterated for the number of rounds attributed in the function's parameters. After completing the loop the agents are reset, meaning they will reset the utility they gained and any information they memorized from the previous game in order to have a fresh start for the next game.

4.1.3 Tournament Module



The Tournament module contains one choice box at the top, which displays the list of every available strategy when pressed. The "+" and "-" buttons will increment and decrement, respectively, the number in the text field between them. The "ADD" button next to the choice box will add agents to a list by the number specified in the text field. The display then features a labelled text field where the user can input the number of rounds played per game. Finally, there is the "PLAY" button is used to trigger the tournament between every agent in the list. These components were organised specifically in this format to facilitate the input sequence of the user; beginning at the top, and from left to right.

Below these components, there is a scrollable pane that displays every agent currently in the list. Each agent has a label with its ID to help identify it in the statistics window, a label with the name of the strategy it's using, a text field with the total utility it gained throughout the tournament, a button to open its respective statistics window, and a button that removes the agent from the list. Since each agent consists of various components, the View class initialises five array lists (one for each component). The list of agents is the most important information for the tournament, therefore it is placed at the center and occupies most of the screen.

Upon going through some revision of the code, these multiple arrays could have been better structured and more efficient when dealing with large lists of agents. Since every agent always has the same components, this list could have been reduced into one single list consisting of arrays with a fixed size of five. Those arrays would then contain each individual component that make up the agent. This will ensure the system only has to go through one array list whenever an agent is updated.

At the bottom of the page there are two labelled text fields. One field displays the total amount of agents that participated in the most recent tournament, and the other field displays the total amount of utility gained over every agent. The view also includes a button that will clear every agent from the list. These components were placed purposefully at the bottom since it used to output the result of a tournament or use a functionality mainly used at the end of a tournament. This placement ensures that these components are the last items the user sees. The screen also features a "BACK" button that returns the user to the main menu.

Aside from the basic accessors and mutators in the view class for each components, there are a few more complex functions worth mentioning. As mentioned previously, agents are added at run time, therefore there is a method that creates a JavaFX Hbox to represent the agent. The function appends each of the five components to the Hbox and adds them to the respective array list. There is another function that takes a string with the name of a strategy and adds a new agent to list after initialising it with a call to the previous function. There are two similar functions used to remove and clear the agent list. The `deleteAgent()` function takes an index as a parameter to determine which agent to delete and removes each of its components from their respective list. In order to maintain the ordering of the agents the function will also loop through every agent above the indicated index and decrement its ID by one. It will also change the background so every odd ID has a grey background and every even ID has a white background. The `clearAgentList()` function functions in a similar way, but does not an index parameter, instead, it removes every entry from every list. It also does not need to go through the loop in the `deleteAgent()` function.

The functionality of the GUI in this module is handled by an independent controller class. The controller is the only class that handles and uses functionality from its respective view and model. The controller has three fields: the view, the manager and the loader. The controller contains a constructor that takes a manager object and a view object, and initializes them to its respective field. The constructor will also add the listeners for the buttons in the view.

Similarly to the One V One controller, the Tournament controller only has one method which listens for events in either of the multiple buttons of the GUI. If the "+" button is pressed, the controller will call a method in the view that increments the value of its text field, and if the "-" button is pressed, the decrementing function will be called instead. When the "ADD" button is pressed, the controller will call the previously mentions function that adds a new agent to the list along with the necessary listeners for the newly created buttons used to display statistics and delete the agent. When the "CLEAR ALL" button is pressed, the previously mentioned `clearAgentList()` method is called from the view class. If the "PLAY" button is pressed, it will call the method `runGame()` inside the manager. However, this method only takes two parameters: the list of agents and the number of rounds. After the manager evaluates the result of the game, the controller will retrieve the output from the manager and update the respective text fields in the view with the final result for each agent. If the user presses the "BACK" button, the controller will call the loader class in order to change displayed screen back to the main menu.

At the end of the method there is also a pair of loops that iterate through the list of "STATS" buttons and "X" buttons to determine whether any of them were pressed. The loops were placed at the end of the method in order to avoid going through them as much as possible. Pressing a "STATS" button will create a new stage for a small window to pop up, containing the various statistics measured for the specific agent (See 4.2.3). Unlike the "BACK" function, this function will create a new window for the statistics so that the information on the tournament is not lost and so the various statistics may be compared side by side. Running a new tournament will not update the statistics window, it needs to closed and reopened. Pressing an "X" button will call the previously mentioned function that deletes a single agent from the list.

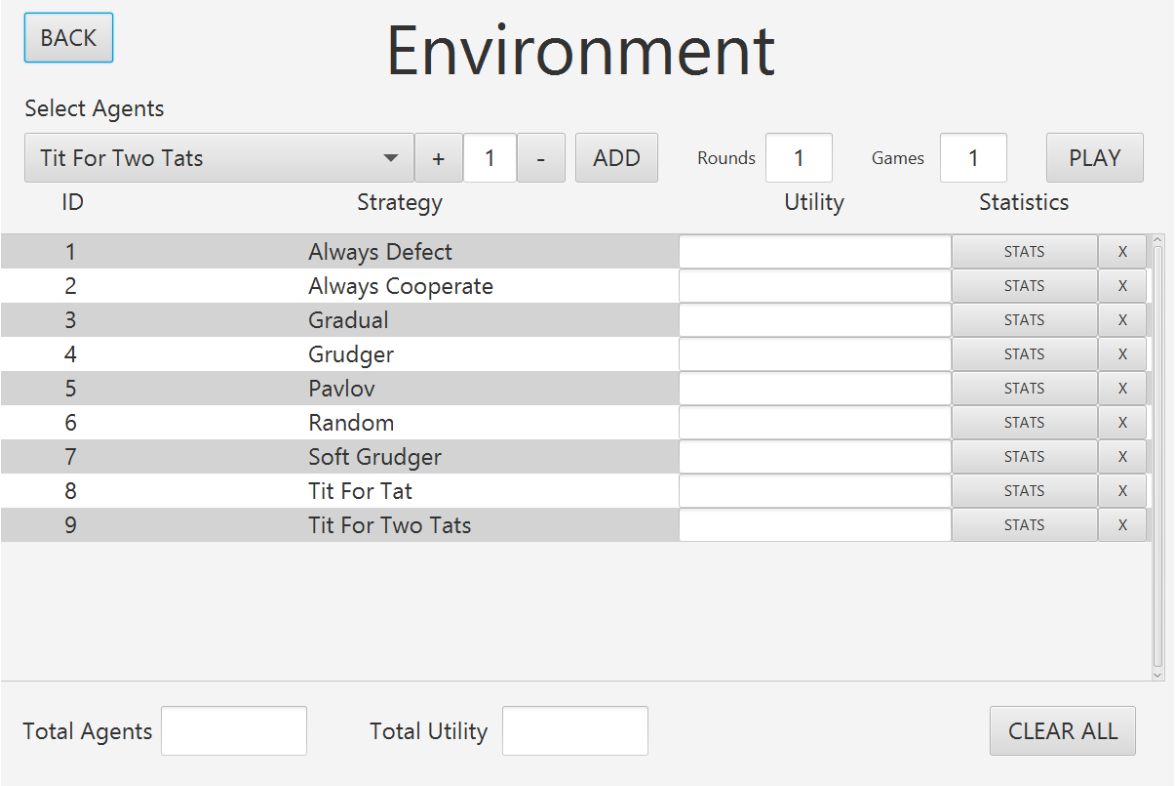
The manager class is where the main functionality of the program is evaluated. It contains three fields: the agent manager, the game matrix, and the list of agents. The agent manager and the game matrix are instantly initialised, but the agent list is only initialised when a game is run. The model then feature three methods. An accessor method for the list of agents, a method that clears the list, and the method that runs a tournament of the Prisoner's Dilemma.

The method `runGame()` takes two parameters: the list of strategies for the list of agents, and the number of rounds for the game. The number of rounds is received as an integer and the strategies are received as an array list of strings. The first step in the method is to translate the string value of each strategy in the list into and an Agent object. Much like the One V One module, the method `stringToAgent()` is called from the agent manager in order to determine which strategy was attributed to the agent and return an initialised agent object. Upon creation, each agent will also be given an ID in accordance to the ID displayed in the GUI.

After every agent is successfully instantiated, the function makes its iterations over the Prisoner's Dilemma. It asks the agent to choose their next move, either "D" (defect) or "C" (cooperate). After the agent decides on its choice, the manager will call the `evaluate()` function in the game matrix. This function will receive the choices from both agents and run them through the matrix for the Prisoner's Dilemma. The manager will retrieve the result for each of the agents inside the game matrix and increment it to the utility each agent has gathered throughout the game. Finally, tells the agents that the iteration is over so they can set memorize all the information they need in order to run their strategy's algorithm for its next choice. This whole process is iterated for the number of rounds attributed in the function's parameters. After completing the loop the agents are mapped to each other (See 4.2.1). Finally, the agents are reset, meaning the utility they gained against their last opponent will be reset to zero after adding it to the total amount of utility they gained over the course of the tournament.

Unlike the One V One module, the set of iterations mentioned above is looped through yet again so that each agent can play once with every other agent, not matter the size of the agent list. In order to achieve this, there is one loop that iterates through every agent of the list. Inside this loop, there is yet another nested loop that iterates through every agent on the list. However the nested loop will increment its starting index every time the outer loop is completed so that agents don't play against each other twice. The output of each of these loops are the agents that run through set of iterations previously explained.

4.1.4 Environment Module



The Environment module contains one choice box at the top, which displays the list of every available strategy when pressed. The "+" and "-" buttons will increment and decrement, respectively, the number in the text field between them. The "ADD" button next to the choice box will add agents to a list by the number specified in the text field. The display then features a labelled text field where the user can input the number of rounds played per game. A second labelled text field allows the user to choose how many games the agents will iterate through. Finally, there is the "PLAY" button is used to trigger the tournament between every agent in the list.

Below these components, there is a scrollable pane that displays every agent currently in the list. Each agent has a label with its ID to help identify it in the statistics window, a label with the name of the strategy it's using, a text field with the total utility it gained in the environment, a button to open its respective statistics window, and a button that removes the agent from the list. The components are initialised similarly to the Tournament and suffer from the same problem with inefficiency.

At the bottom of the page there are two labelled text fields. One field displays the total amount of agents that participated in the most recent environment, and the other field displays the total amount of utility gained over every agent. The view also includes a button that will clear every agent from the list. The screen also features a "BACK" button that returns the user to the main menu. Every component was organised in exactly the same format as the Tournament (with only one additional text field) so the user doesn't need to adapt to a different display.

This view class has most of the same functions mentioned in the Tournament view. The only addition to this view is the field for the number of games, and its accompanying mutator. Every other function is essentially the same as the Tournament view, since most the differentiation between the two game modes are in the model where the algorithmic evaluation takes place.

The functionality of the GUI in this module is handled by an independent controller class. The controller is the only class that handles and uses functionality from its respective view and model. The controller has three fields: the view, the manager and the loader. The controller contains a

constructor that takes a manager object and a view object, and initializes them to its respective field. The constructor will also add the listeners for the buttons in the view.

Much like its view, the Environment controller is very similar to the Tournament controller since the view has most of the same components. The main difference between the two classes is the fact that the Environment controller initialises the Environment view and manager instead of the equivalent Tournament classes. The event handling method listens for clicks on all the buttons previously mentioned. Every button event, except for the "PLAY" button, functions exactly the same as the Tournament equivalent. If the "+" button is pressed, the controller will call a method in the view that increments the value of its text field, and if the "-" button is pressed, the decrementing function will be called instead.

When an event for the "PLAY" button is triggered, the controller first clears the text fields containing the results of previous iterations of the Environment, and it asks the manager to clear the list of agents it contains from the possible previous iteration. These two simple steps help resolve any conflicts between the current iteration and the previous iteration of the Environment. After resetting the information it calls the `runGame()` from the manager class. This version of the function takes the same parameters as the Tournament version: the list of agents and number of rounds, along with an additional parameter: the number of games. This extra parameter determines how many times each agent will attempt to play against every other agent. After running the Environment, the controller updates the text fields containing the individual results of each agent,

The manager class is where the main functionality of the program is evaluated. It contains three fields: the agent manager, the game matrix, and the list of agents. The agent manager and the game matrix are instantly initialised, but the agent list is only initialised when a game is run. The model then feature four methods. An accessor method for the list of agents, a method that clears the list, the method that runs an environment for the Prisoner's Dilemma, and a sub-method that runs a single game between two agents for the designated number of rounds.

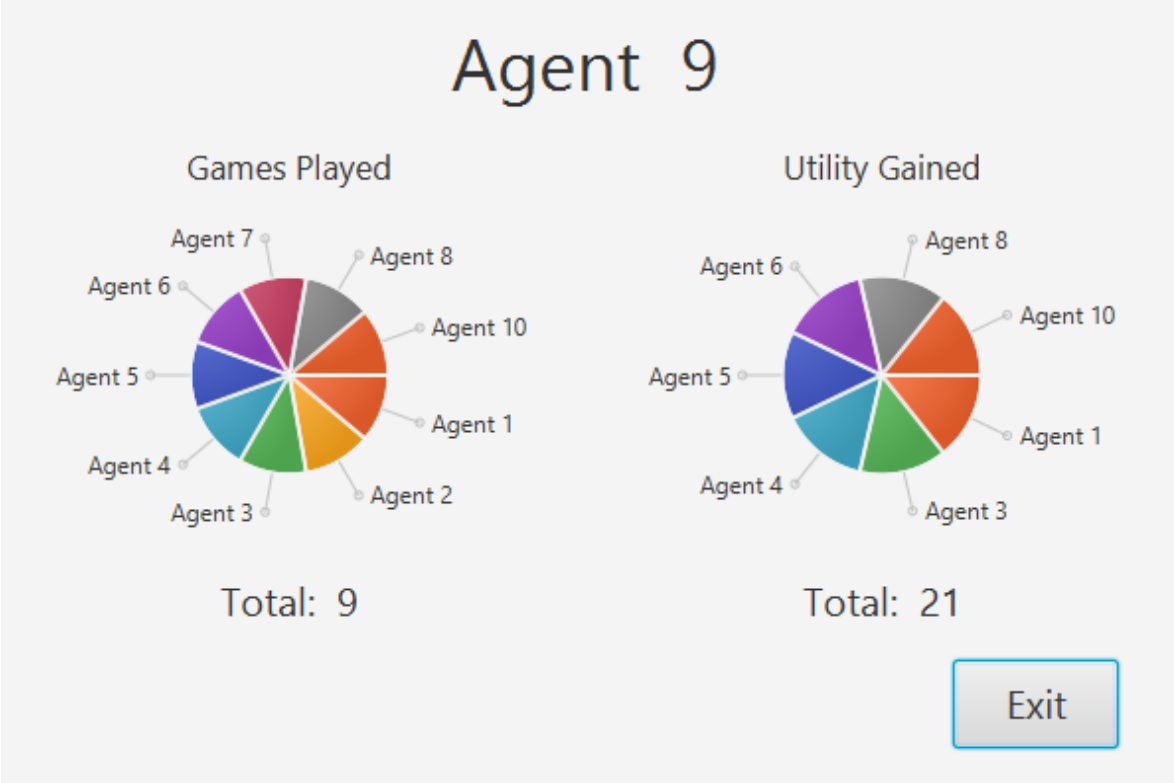
The method `runGame()` takes three parameters: the list of strategies for the list of agents, the number of rounds for a game, and the number of games. The number of rounds and games are both received as an integer and the strategies are received as an array list of strings. The function first calls the method `stringToAgent()` from the agent manager in order to determine which strategy was attributed to the agent and return an initialised agent object. Upon creation, each agent will also be given an ID in accordance to the ID displayed in the GUI.

After every agent is successfully instantiated, the function makes its iterations over the Prisoner's Dilemma. The function creates three nested loops. The outer most loop iterates through the number of games designated in the parameter. The second loop iterates through the list of agents and attributes the current agent to the variable "agent 1". At the end of every iteration, the average utility gained for agent 1 is updated. The third loop iterates through the list of agents again and attributes the current agent to the variable "agent 2". However, the starting index of this loop is incremented every time its parent loop completes an iteration. This prevents agents from duplicating games against agents that come first in the list. Inside the inner most loop the function: checks whether the agents are not null (to avoid exceptions), checks whether each of the agents wants to play against the current opponent, and calls its sub-function `runRounds()` in order to run a game between agent 1 and agent 2 if both conditionals are met. If one of the agents does not want to play against the current opponent the between these agents will not occur, no matter what the other agent might want.

The method runRounds() was created upon refactoring the code in order to make the code more legible. Since runGame() already involves many loops and conditionals, creating a new method with a standalone functionality proved to make the code more comprehensible, even though it is only used once in a single method. The function asks the agent to choose their next move, either "D" or "C". After the agent decides on its choice, the manager will call the `evaluate()` function in the game matrix. This function will receive the choices from both agents and run them through the matrix for the Prisoner's Dilemma. The manager will retrieve the result for each of the agents inside the game

matrix and increment it to the utility each agent has gathered throughout the game. Finally, tells the agents that the iteration is over so they can set memorize all the information they need in order to run their strategy's algorithm for its next choice. This whole process is iterated for the number of rounds attributed in the function's parameters. After completing the loop the agents are mapped to each other. The manager asks each agent whether they want to play again against the opposing agent based on the average utility gained over the last set of iterations. Finally, the agents are reset.

4.1.5 Statistics Module



The statistics module contains two labelled pie charts that express the two statistics measured in both the Tournament and the Environment. The title of the window details which agent is being analysed (agents are identified by their ID). The first chart shows how many games the agent has played in regard to every other agent on the list, and the total number of games is stated below the graph. The second graph shows how much utility the agent has gathered in regard to every other agent on the list, along with the total amount of utility gained labelled below the graph. Finally, there is a button "EXIT" at the bottom of the screen, which is used to close the window.

Opening the GUI for the statistics will not overwrite the GUI of the Tournament or Environment, therefore, there can be multiple windows open displaying the statistics of multiple agents. This feature allows the user to place statistics of different agents side by side for an easier analysis of the results. However, running a new game will not update the results on already open statistics. The windows need to be closed and reopened from the list of agents.

There are two noteworthy methods in the view class that initialize the pie charts. Both methods receive a map of strings as a key (agent ID) and integers as the value (number of games or amount utility gained). Both functions iterate through every entry in the map and add it to the graph, as well as sum up every value in order to later display the total in the label below the graph.

The functionality of the GUI in this module is handled by an independent controller class. The controller is the only class that handles and uses functionality from its respective view and model. The controller has three fields: the view, the manager and the loader. The controller contains a constructor that takes a manager object and a view object, and initializes them to its respective field. The constructor will also add the listeners for the buttons in the view.

The Statistics controller consists of three fields, the constructor and a single event handling method. The fields are divided into the respective view and manager, along with a stage due to its nature as a separate window from the other modules. The event handler only listens for clicks on the "EXIT" button, after which it closes the stage defined in its field. Unlike the previous modules, the constructor contains most of the functionality because the module is not interactive after initialization. The constructor receives an instance of the view, the stage where the view is open and the agent whose statistics are to be displayed. It then calls a method in the manager that modifies the agent statistics into a map that can be read by the GUI; changes the title of the GUI to display the correct agent ID; and, finally, initializes the two graphs with the translated maps saved in the manager.

The manager of this module only contains one main method whose purpose is to adapt the statistics of an agent into maps the view can read. The function receives an agent as the parameter, and iterates over the maps inside the Statistics object (See 4.2.3) within the agent. The maps inside the object have agents as a key. This function creates a new map with the agent's ID as a key and saves the result to its fields. This allows the view to interpret the statistics without exposing it to the agent object.

4.2 Object Oriented Programming

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

4.2.1 Agent

The Agent object has a variety of fields and methods, however some of them are only used for one of the main modules. Each Agent object has a field dictating its ID, its strategy, the total utility gained and the utility gained over one game with one opponent. All of these fields are used in all three main modules. In case of the One V One, the total utility is equal to the utility gained over one game. The object then has fields that are only used for the Tournament and Environment. These are: a map that stores every agent this agent has played against along with the respective average utility it gained against them, and the statistics for the agent (See 4.2.3). Finally, there are the fields that are only used in the Environment to determine whether the agent will want to repeat a game against a mapped opponent. These are: the average utility gained over a game with every other agent, a list of agents that this agent is willing to play against in a future game, and the threshold of average utility gain that determines whether an agent goes on the previous list or not.

The constructor for this object attributes the agent with the strategy given in its parameter, and initializes the total utility and average utility to zero in order to avoid null pointer exceptions. Aside from the accessors and mutators for its multiple fields, the object consists of some small but noteworthy functions that allow the agents to keep updating their utility and decisions for future iterations of the game.

The Agent has the function `incUtility()` which takes an integer and adds it to the utility gained over one game. This function is called on every iteration of the game. It features the function `setOpponent()`, which attributes an opponent to its strategy in order to allow it to store the previous moves the opponent made. Finally, it has the reset() function that adds the utility gained over one game and adds it to the total utility, sets the game utility back to zero, and resets any variables involved in its strategy's algorithm.

The class also consists of a method that adds, a given agent and the average utility gained over a game with said agent, to the map of known agents previously mentioned in the fields. It also calls a method in its statistics object that adds the same information to its own map.

Finally, there are the methods that are currently uniquely used for the Environment. First is the method `setAvgUtility()`, which, unlike a traditional mutator, uses the set of values in the map

of known agents in order to determine the average utility after running through a single game with every other agent. The last function worth mentioning is the `isInUtilityThreshold()`, which determines whether utility given in the parameter is over the threshold determined in the agents field, and returns either true or false.

4.2.2 Strategy

The Strategy object contains the information and functionality required to use a strategy to play the Prisoner's Dilemma. This feature makes use of inheritance to allow the creation of multiple unique strategies. The parent class has fields for the name, opponent, current choice and last choice. These are variables that every strategy uses. It has a method `choose()`, which changes the current choice of the strategy. The parent class does not make any calculations on this method because it should be overridden by ever child object. It also contains a method that changes the last choice to the current choice, and a method, `reset()`, that set the last choice back to null (the more complex strategies override this method). The package is further divided into ten objects that extend the parent class.

The class Always Cooperate overrides choose(), which always sets the current choice to "C".

The class Always Defect overrides choose(), which always sets current choice to "D".

The class Random overrides choose(). The method generates a random number between 0 and 1, if the number is below 0.5 it sets the current choice to "C", else it sets it to "D".

The class Tit for Tat overrides choose(). The method chooses "C" if it's the first choice of the game, else it will choose the opponent's last choice.

The class Tit for Two Tats overrides choose() and adds an extra field to save the value of the opponents second to last choice. The method chooses "C" if it's the first choice of the game, else if the opponent's choice is the same as the opponent's second to last choice it will reciprocate, otherwise it will set the save the opponents last choice to its field.

The class Pavlov overrides choose() and initialises a game matrix. It sets up the matrix with its last choice and the opponent's last choice, if the result is 3 or 5 it keeps the same choice, otherwise it changes its choice.

The class Grudger overrides choose(). It choose "C" for the first round, and "D" if the opponents last choice is "D", after which it will never change again for the rest of the game.

The class Soft Grudger overrides choose(), reset() and adds two new fields to keep track of how many times it still need to defect or cooperate. It starts by choosing "C" for the first round. Afterward it checks whether the opponent defects, if true then it will choose "D" and set its fields to 3 and 2, meaning it has to defect thrice and then change its choice to "C" and cooperate twice. While the fields are not back to zero it will keep decrementing every round. The reset() set every field to zero.

The class Gradual overrides choose(), reset() and adds three fields, two that measure how many times it still needs to defect and cooperate, and a field that keeps track of the opponent's total defections. The choose() function works the same way as the Soft Grudger, except that it sets the defect field to the opponent's total number of defections. The reset() function also sets all of the fields back to zero.

The class Adaptive overrides choose(), reset() and adds an additional eight fields. Two fields that determine how many times it still has to cooperate then defect at the start of the game; a set of three fields that determine the amount of utility gained for cooperating, the amount of times it has cooperated and the average utility gain for cooperating; and a set of three fields that determine the same information for defections. The method choose() plays the initial ten rounds, then checks which average is higher and sets the current choice accordingly. It also recalculates the average at the start of each iteration over a game. The reset() method sets all the fields back to zero, except the two fields first mentioned, that are both set to five.

Statistics

The Statistics object is used to store the statistics of each agent in a separate object. Though the final product of this program only consists of two unique measurements, having an independent object will make the program more structurally sound and will facilitate the possible addition of new statistics in the future with minimal modifications to the agent object.

The object consists of two fields: a map with every known agent and the total utility gained against each agent, and a map with every known agent and total number of games played against each agent. In a tournament the latter field is redundant since every agent is forced to play against every other agent once, however the same object and accompanying view is used for both the Tournament and the Environment.

The constructor initialises both fields into an empty hash map. The class is comprised by an accessor method for each field and the method `mapAgent()`. This method's parameters include the current opposing agent and the utility gained over the last game. If the agent is not already in the map, agent is added as a key and the utility is added as its value. However, if the agent is in the map, the utility will be added to the current value of that key, and the sum will be added as the new value. Similarly, the map of total games played will be initially added with a value of one, and every time a key is already contained in the map its value will increment by one.

4.3 Miscellaneous

Aside from the classes involved in the MVC architecture and the objects used to run the game, the program consists of a few miscellaneous classes. These are classes used to initiate the program with as a JavaFX application, control the GUIs, and refactor common functions for a clearer structure and more efficient use.

4.3.1 Driver

The Driver is a package that consists of three classes, responsible for initialising the program and controlling the flow of each screen. The Driver class is the main method of the program, and all it does it launch the loader.

The Loader class contains all the functionality required to initialize the JavaFX application and switch between screens or create new windows. It extends Application in order to be executable as a JavaFX application. The class consists of one field, the stage (i.e. the current open window), and five methods, one of which is a simple accessor for the stage field. The first method is the `start()` method for the application. The method initializes the stage of the application and adds the GUI of every module into a map into a DisplayControl object, which will be discussed later.

The loader class then features a method `changeDisplay()`, which changes the current stage into a new GUI. This method will overwrite the current window with the new screen. After initialising displaying new screen, the method will also initialise the respective controller class by using the switch statement in the private function, `handleController()`. This method takes the name of desired module, and initialises the relevant controller. The last method in this class is `newDisplay()`, which, instead of overwriting the current stage, it creates a new stage (i.e. open a new window) for the GUI named in its parameter.

The final class in this package is the DisplayControl. This class saves a map with the simplified names of each of the available modules, and the respective JavaFX pane object that is required to instantiate the GUI. It contains a method that adds a module to the map, a method that removes an element for the map, and finally a method that activates the required GUI by inputting the simplified name of the module.

4.3.2 Agent Manager

The Agent Manager was a class created for functions that are used across multiple other classes. The final product of this class only contains one method, `stringToAgent()`. As previously mentioned, this method receives a string with the name of a strategy and instantiates an Agent object with the attributed strategy. This process requires the function to go through a switch statement to find the correct strategy. This method is called by every manager of the three main modules, therefore, having the code in a single separate class will make it much easier to add new strategies to the program. It also helps with finding possible errors, such as a misspelled name for the strategy.

Though the class only has one method, it could have many more. Some of the code in the manager of the main modules was very similar, but there were slight differences that made it hard to create a common method in this class for all three modules. However, on the event of adding additional functionality to the program, the class can be used to help keep the structural integrity of the program and avoiding having to write the same code over multiple independent classes.

4.3.3 Game Matrix

The Game Matrix class is a very simple class with the sole purpose of evaluating the result of a single iteration of the Prisoner's Dilemma. It consists of three fields: the utility gained for agent 1, the utility gained for agent 2, and the string representation of the last iteration. The class then features an accessor for each result field, the `evaluate()` method, and the `printRound()` method which was initially used to verify whether the strategies were applying the correct algorithm in response to previous rounds.

The `evaluate()` method receives two choices in string form, either "D" or "C" and concatenates the two strings. It uses the resulting string and runs it through a switch statement in order to determine the result for each of the agents. The switch statement consists of the available options in the standard Prisoner's Dilemma matrix (See 2.1), and updates the result field for each agent also in accordance to the standard matrix.

Creating an independent class for the matrix evaluation was one of the first refactoring processes made in the development of the program, since it's a function used over every main module of the program. It's also a function that will be used on the possible creation of a new module, and a function that requires little to no modification in any program involving the Prisoner's Dilemma.

5 Evaluation

5.1 Evaluation Procedure

The amount of tests that can be run with this program are essentially infinite, since there a lot of variables involved: number of agents, combination of agents, number of rounds in an encounter, and number of games in an Environment. The variation that can be given to these variables allow for infinite tests (limited only by the CPU's processing capacity). Therefore, it is important to select a set of tests that can determine the answer to the thesis statement: what is the best strategy for the Prisoner's Dilemma, and under what circumstance.

First we will test the quality of each strategy in a tournament with fixed list of agents and an incrementing number of rounds per encounter. We will begin by testing a tournament with one round, followed by two rounds, then ten rounds, twenty rounds, and finally one hundred rounds. This test will show us how each strategy evolves with the increase in number of rounds, which determines how well each strategy optimizes that utility gained from each opponent.

Secondly, we will test the quality of each strategy in a tournament with a fixed large number of rounds and a randomly generated list of agents. The program does not include a function to randomly generate a list of agents, so we will be using a random number generator to determine the size of the list. We will fill in the list by randomly selecting a strategy based on the order in which it is displayed on the GUI (alphabetical). Each test will generate twenty random lists; the strategy being tested will always be included at least once at the top of the list; and there will be one of these tests for every strategy (except Random). This test will show us which strategies adapt the best to the opponents they are presented with. Random generation can always give uncertain results in small samples, but it should be enough to give us an idea of the quality of each strategy.

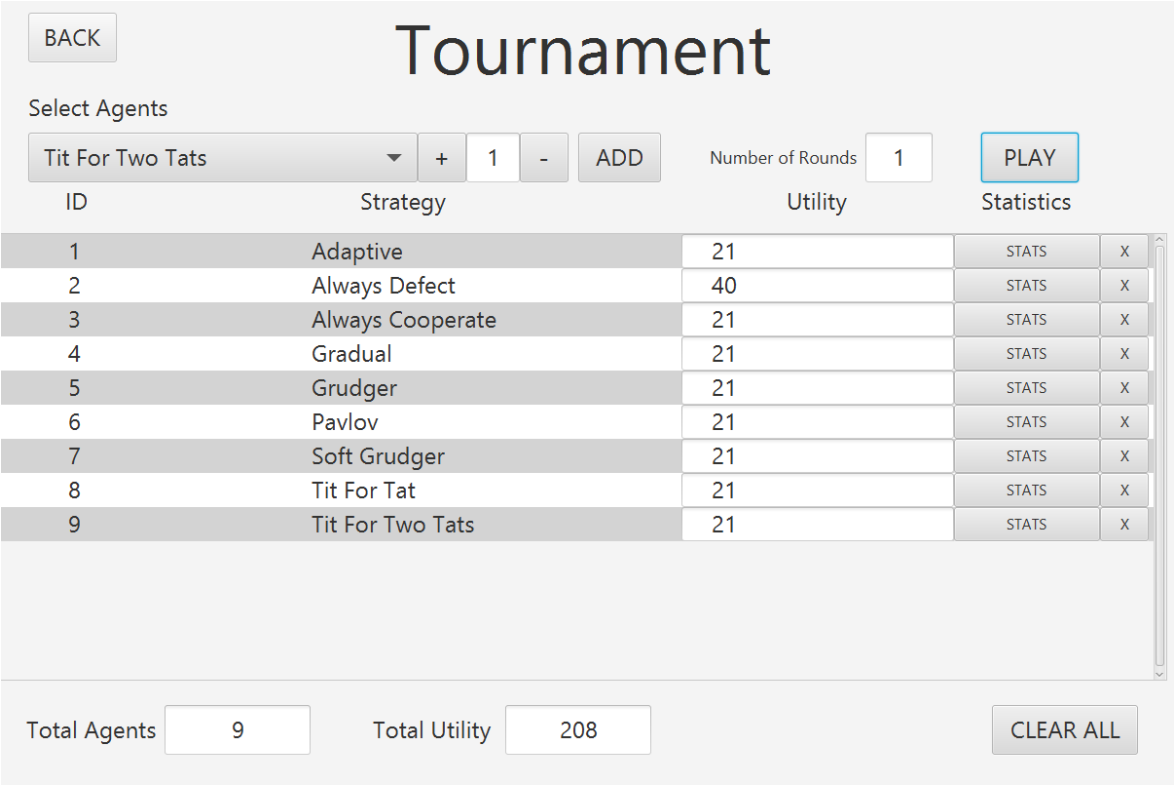
The One V One module was mostly used to test whether strategies were functioning properly during development. Though it may be interesting to see how strategies play individually against other strategies, the results would not help us find suitable answers for the thesis. Therefore, we will not use this module to run tests for the evaluation.

5.2 Number of Rounds Tests

On Test Set 1 we will set tournament with a list of nine agents, where every agent has a unique strategy. The Random strategy is not included because it would only help to create variance in the results. Each test will run a tournament where every agent will have a single encounter with every other agent. With every new test, the number of rounds per encounter will increment.

5.2.1 Test 1

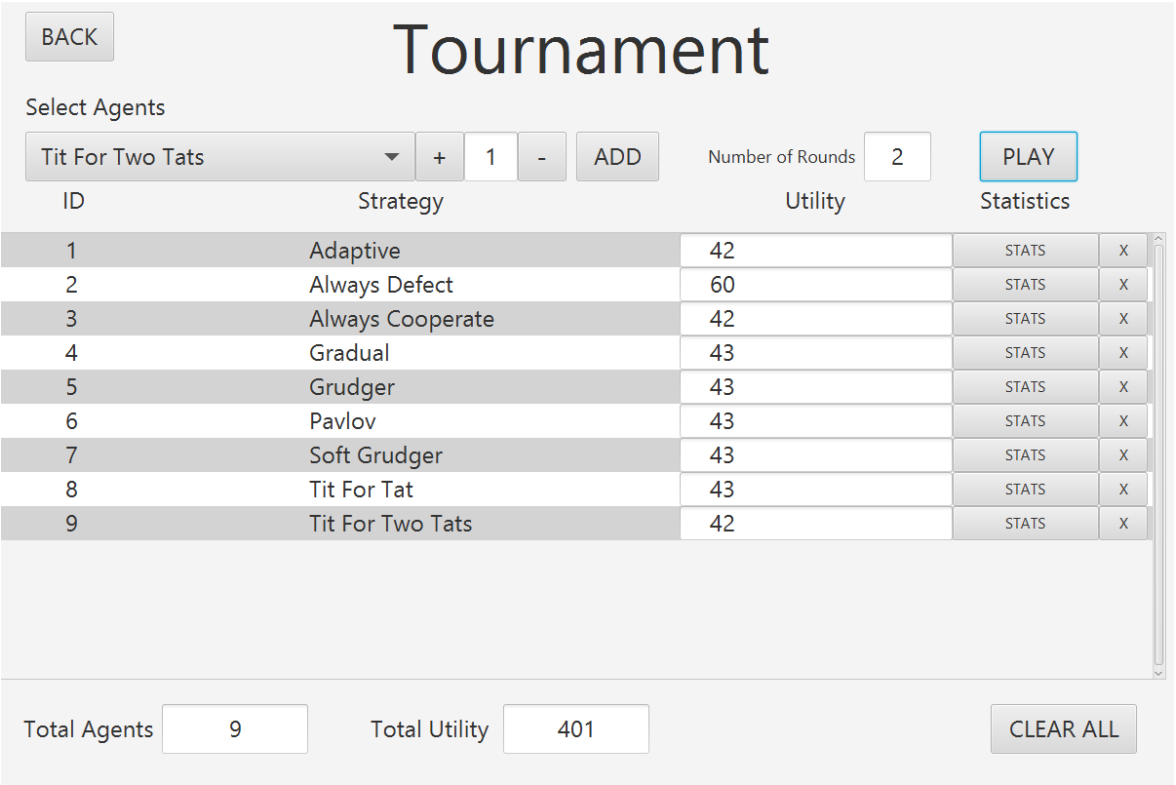
The first test will run the tournament with only one round per encounter to show which strategy is best when playing a non-iterated version of the Prisoner's Dilemma.



The results of Test 1 show that Always Defect was the most successful strategy because every other opposing agent begins by cooperating. Every other agent gained the same amount of utility. These results indicate that defecting is always the most efficient option when playing for a single round of the Prisoner's Dilemma.

5.2.2 Test 2

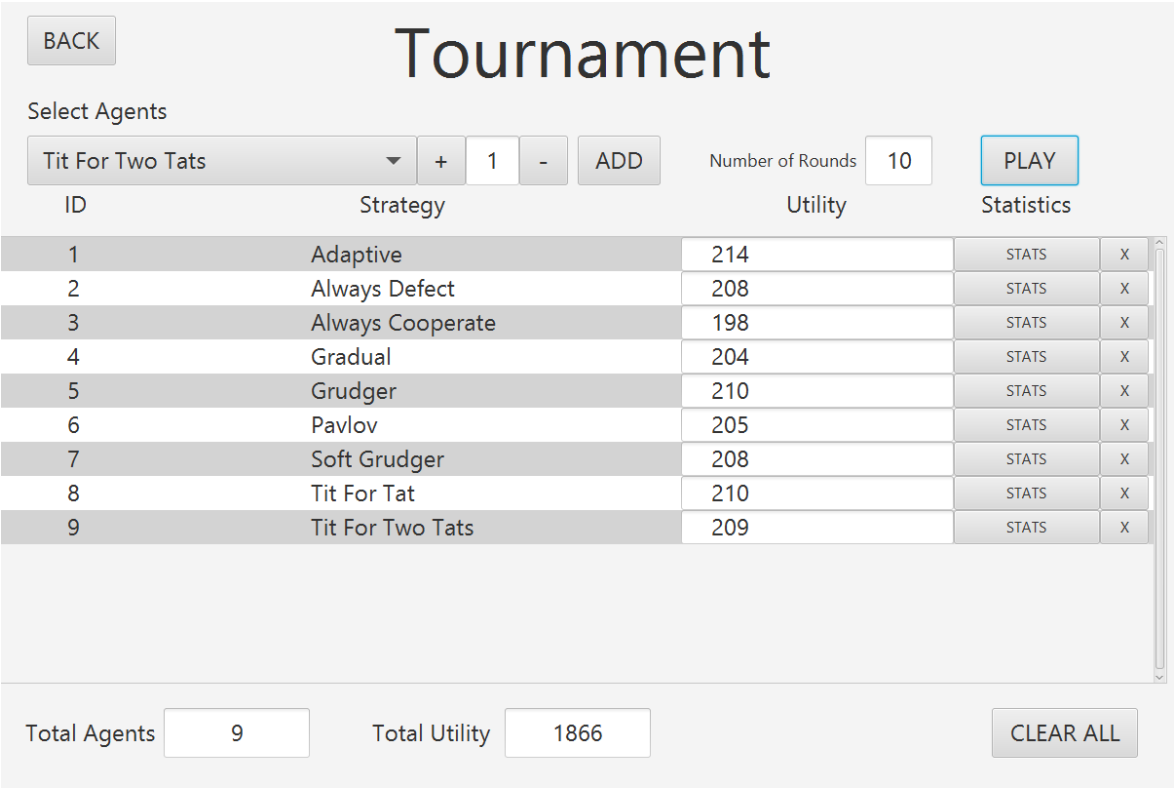
The second test will run the tournament with two rounds per encounter to see how well strategies immediately react to the opposition.



The results of Test 2 still show us that Always Defecting is still the best strategy, however, we can also see that some of the strategies adapted to the second defection by also defecting. These strategies gained one more utility over strategies that kept cooperating with the defector. We can also see that the amount of utility Always Defect has over the other strategies has been reduced in terms of ratio. While it had almost twice the amount of utility as the other agents in Test 1, it only has one third more utility after an additional round. This begins to tell us that this strategy will begin to lose its lead the more rounds that are played in each encounter.

5.2.3 Test 3

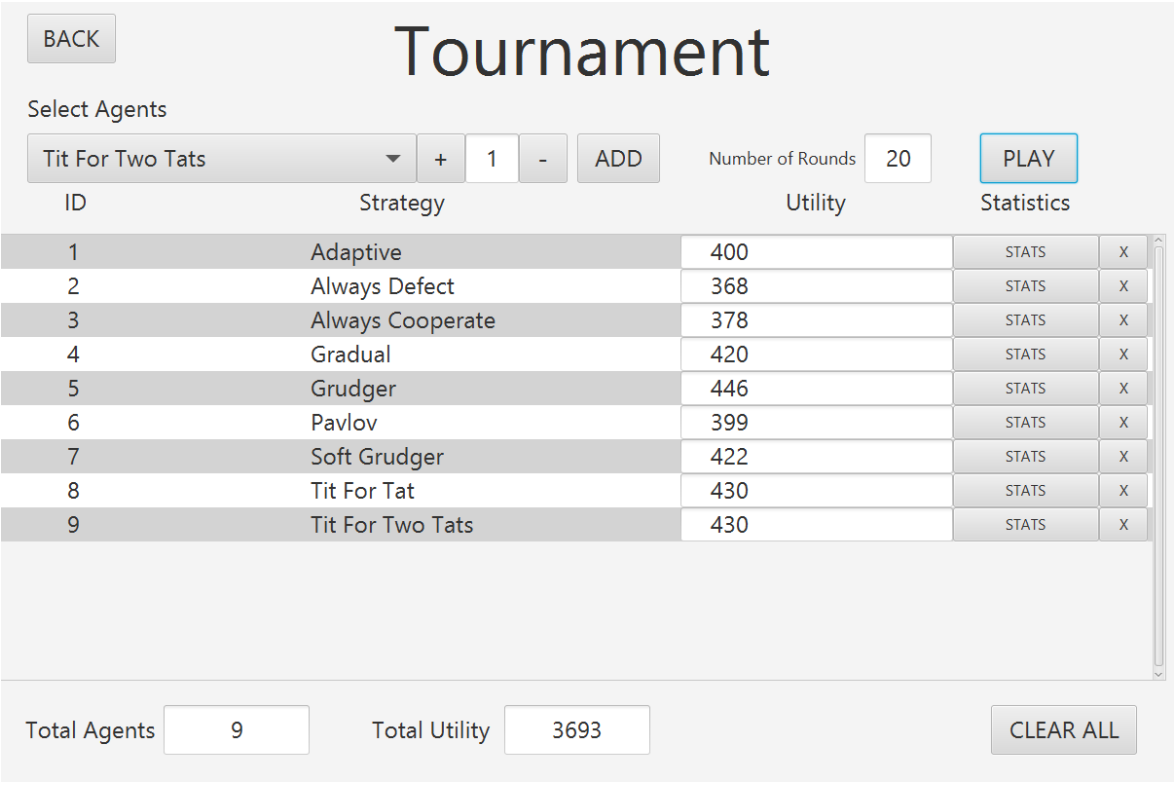
The third test will run the tournament with ten rounds per encounter to see which strategies have the most efficient reactions to the opposition.



The results of Test 3 already show us that Always Defecting no longer has the most utility. This happens because most of the other strategies have learned no to cooperate with it, while still cooperating with the other strategies. Always Cooperate still has the least amount of utility because it does not reciprocate defection. The strategy with the most utility at the point is Adaptive. The Adaptive strategy is the only strategy that actively tries to defect in order to get the most utility, but will still cooperate if it recognizes the opponent is reciprocating defection. The strategies with the second most utility are Tit for Tat and Grudger because the reciprocate defection and never try to cooperate again until the opponent decides to cooperate first.

5.2.4 Test 4

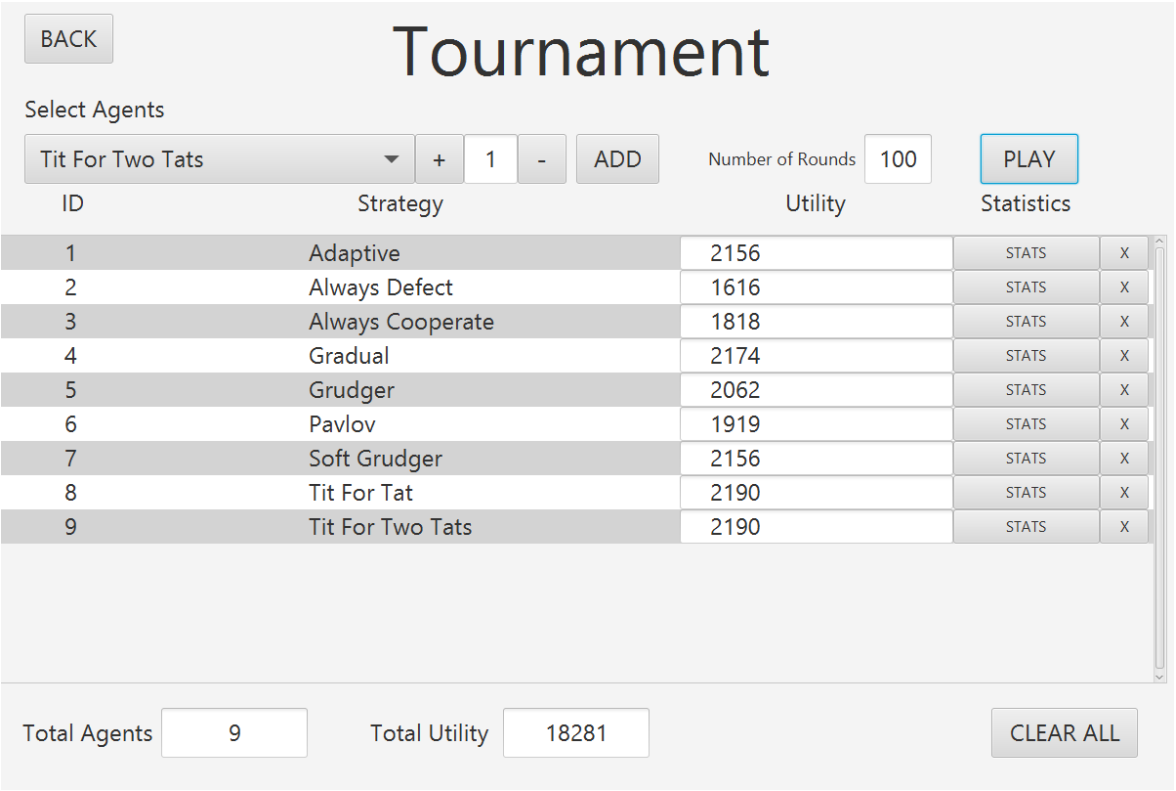
The fourth test will run the tournament with twenty rounds per encounter to see whether the top strategies keep their lead on a game that is twice as long.



The results of Test 4 show us that the Adaptive strategy is no longer favoured because the more cooperative strategies have played enough rounds against each other to ensure a selfish strategy does not have the chance to take the lead. The test also shows us that Always Cooperate has overtaken Always Defect because most of the opposing strategies prefer to cooperate over defecting. The strongest strategy at this point is the Grudger, which tells us that willingness to cooperate and an absolutely unforgiving attitude on defection might be the best strategy on long iterations of the game. At this point it is also worth mentioning that the Pavlov strategy proves to be the weakest of the complex strategies, perhaps because it is unable to optimize its decisions when the opponent keeps defecting. To recap, the Pavlov strategy sees whether it has received either 3 or 5 utility on the previous round and repeats it if so, otherwise it changes its move. If the opponent always defects its conditional will never be satisfied, hence, it will keep changing its move.

5.2.5 Test 5

The fifth and final test will run the tournament with one hundred rounds per encounter in order to see whether the same trend keeps showing up at a much higher iteration count.



The results of Test 5 show us that the best overall strategies were the two versions of Tit for Tat, followed by Gradual, then both Adaptive and Soft Grudger (which is interesting because the strategies have very different goals), Grudger, Pavlov, Always Cooperate, and, lastly, Always Defect. At this amount of iterations, the Grudger strategy has fallen in grace and the Adaptive strategy climbed of the tournament ladder, the rest of the strategies remained with approximetely the same ratio as the previous test.

In Appendix A eight similar tests can be found running from 200 to 900 iterations with the same list of agents. These tests show that the rankings described no longer changes, with the exception of the Adaptive strategy that overtakes the Soft Grudger on longer iterations.

5.2.6 Conclusion:

The outcome of these tests tell us four main points of information:

- Defecting is the most efficient strategy in very short iterations of the game.
- Immediate reciprocity is the best strategy in long iterations of the game.
- It is always best to have a complex strategy in medium to long iterations of the game.
- Pavlov is the worst complex strategy at any number of iterations.

5.3 Random Agent List Test

On Test Set 2 we will run a tournament with a list of thirteen agents for 100 rounds per encounter. The first agent in the list will be the test target, every other agent will be randomly generated. In order to maintain a balance between cooperative and selfish strategies, the Adaptive and Always Defect strategies will be given a higher chance of being picked for the list. Each of these tests will be performed ten times for each agent, and, in order to reduce uncertainty, each of those sets will be generated three additional times. This results in a total of 270 tests. The report will only display one example run for each agent but each section will include a table that displays the utility gained over each run and the overall average utility gained over each test set. Furthermore, the strategies will be tested in order of worst to best based on the previous test set.

5.3.1 Always Defect Test

Example Run:

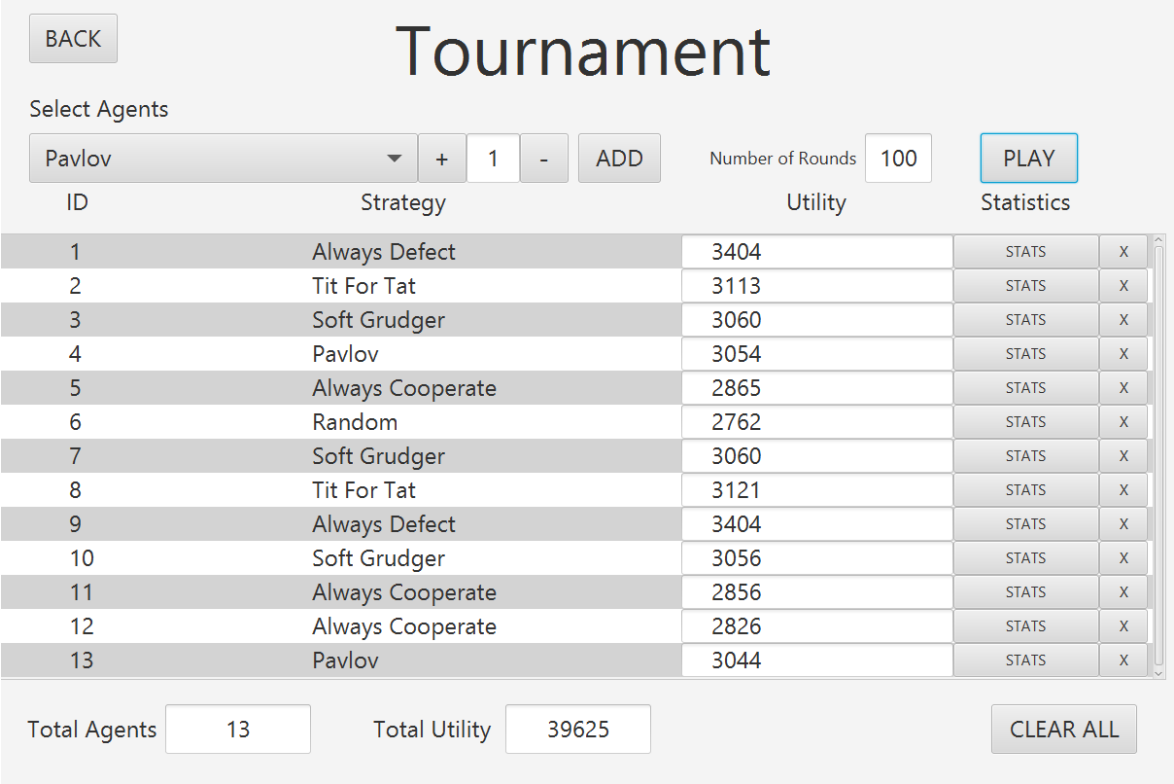


Table of Results:

	UTILITY		
	Test Set 1	**Test Set 2**	**Test Set 3**
TEST 1	3404	2292	2452
TEST 2	1996	1812	2248
TEST 3	1572	1436	1712
TEST 4	1788	2444	2036
TEST 5	2104	1620	1556
TEST 6	1828	2060	1644
TEST 7	3500	2936	2748
TEST 8	1808	2624	2648
TEST 9	1860	2232	2144
TEST 10	2548	2516	2152
AVERAGE	**2241**	**2197**	**2134**

5.3.2 Always Cooperate Test

Example Run:

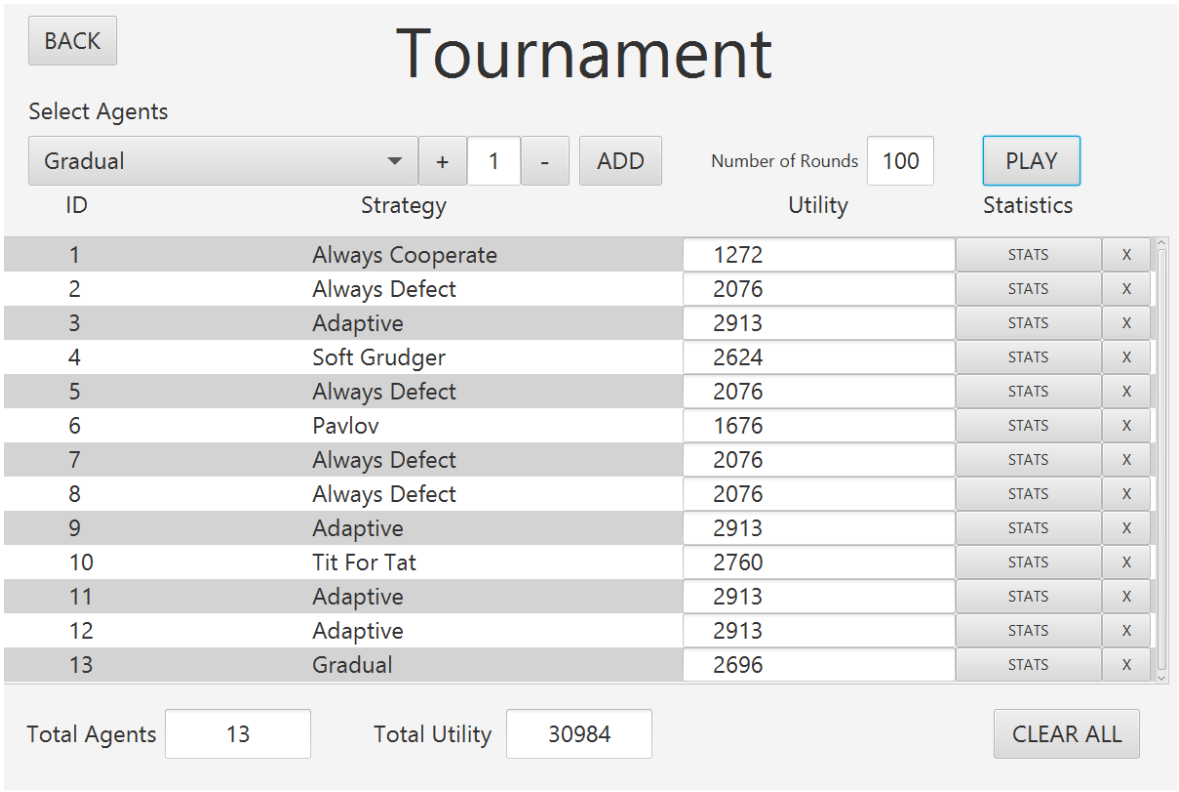


Table of Results:

	UTILITY		
	Test Set 1	**Test Set 2**	**Test Set 3**
TEST 1	1272	600	1407
TEST 2	2508	2700	2982
TEST 3	1854	2754	2400
TEST 4	2289	1554	1278
TEST 5	2154	1564	2136
TEST 6	2586	1734	2454
TEST 7	972	819	1854
TEST 8	1218	1422	1722
TEST 9	2568	1536	2301
TEST 10	1983	2136	1962
AVERAGE	**1707**	**1682**	**2050**

The results of the first two tests shows us that the average utility gained for the Always Defect strategy is substantially higher than the utility gained by Always Cooperate. This likely occurs due to the increased probability of facing a selfish strategy. Whilst the tests with a fixed agent list had more naturally cooperative strategies than selfish strategies, this set of tests attempted to balance this weight. When there are more cooperative strategies, the reward for cooperating will eventually outweigh the loss the more iterations are performed. But when there are just as much probability of defecting as there is of cooperating, the reward will no longer outweigh the loss.

Ultimately this proves that when the ends are balanced, it is always a safer bet to defect rather than to blindly cooperate. However, following a complex strategy that reacts to the opponent's choices, will substantially increase overall utility gained as the next tests will demonstrate.

5.3.3 Pavlov Test

Example Run:

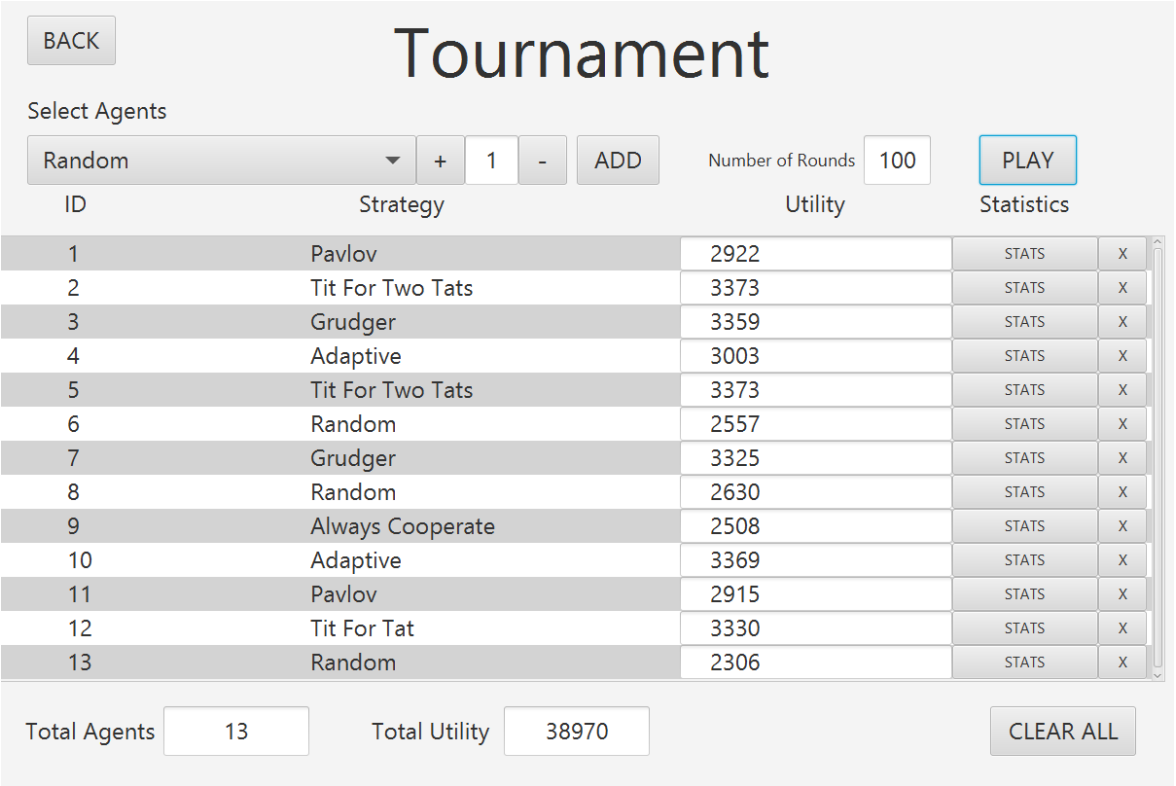


Table of Results:

	UTILITY		
	Test Set 1	**Test Set 2**	**Test Set 3**
TEST 1	2922	2042	2801
TEST 2	2869	2283	2507
TEST 3	1676	2516	1907
TEST 4	2157	1280	1828
TEST 5	2783	2850	2053
TEST 6	2488	2638	1464
TEST 7	2187	1251	2329
TEST 8	2504	1907	2532
TEST 9	2376	2907	1888
TEST 10	2821	1907	2411
AVERAGE	**2478**	**2158**	**2172**

The results of the Pavlov Test display that any reactive strategy (even the worst on the list), is always a better option than to stick solely to one choice. The first set of tests shows a large increase in utility from the previous strategies. However, the other tests show a much lower utility gain, to the point that the average is lower than the first two tests, for Always Defect.

When samples for randomized test are not large enough, there is always a chance of getting unreliable results, which is why the thirty test were divided into three sets. The average over all sets is 2269 for Pavlov, and 2190 for Always Defect. Though the results are approximate, the Pavlov strategy proves to be slightly stronger than the defecting strategy. It is also worth noting that the Pavlov results display lower lows (1251) and also lower highs (2922), than Always Defect (1436 and 3500). Which shows that Always Defect might be a better strategy when playing in high risk situations.

5.3.4 Grudger Test

Example Run:

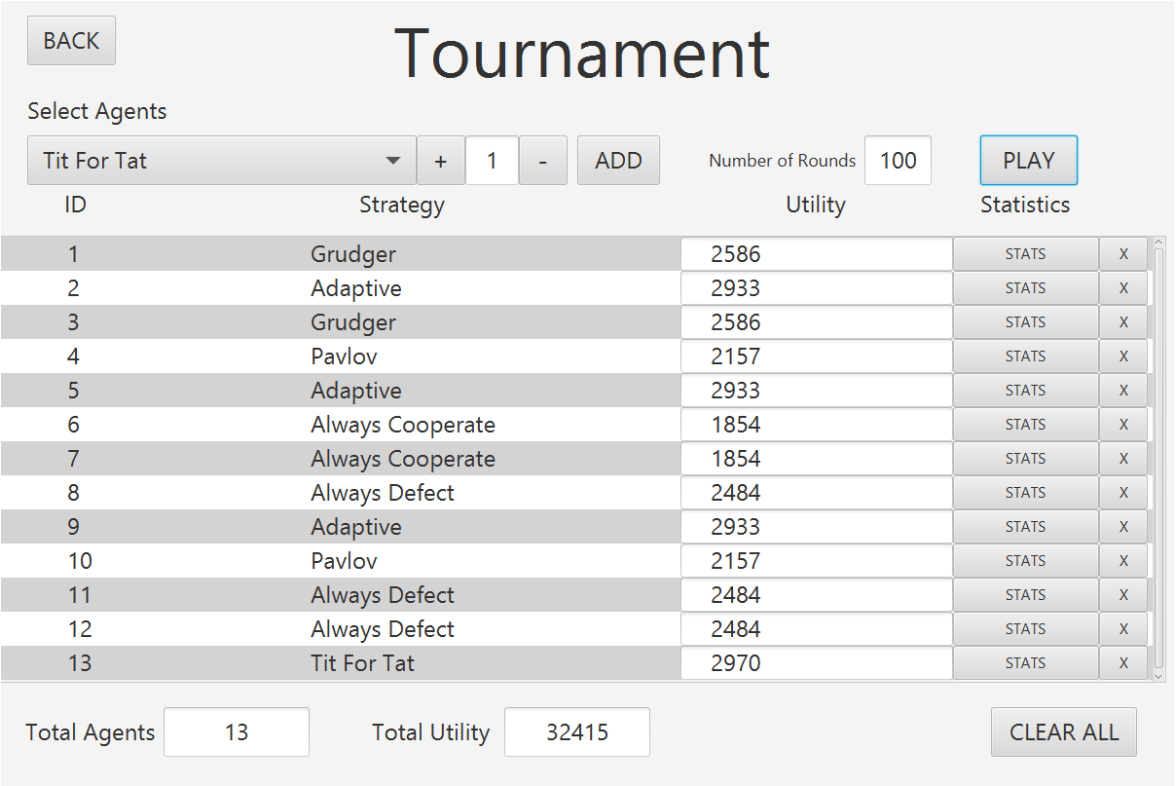


Table of Results:

	UTILITY		
	Test Set 1	**Test Set 2**	**Test Set 3**
TEST 1	2586	2944	2078
TEST 2	3061	2997	2897
TEST 3	3359	2924	2789
TEST 4	2929	2385	2630
TEST 5	2787	3189	2723
TEST 6	2882	2385	2470
TEST 7	2974	3084	2429
TEST 8	2275	2972	2988
TEST 9	2907	2272	2586
TEST 10	2837	2676	2659
AVERAGE	**2860**	**2783**	**2625**

The results of the Grudger Test begin to show a much more significant increase in utility in comparison to the previously analysed strategies. It has an average utility gain of 2752 over three sets, and none of the sets have an average below the previous strategies. The increase in utility has a suitable explanation.

The Pavlov strategy is never satisfied if it gets the reward for double defection (1), therefore, against strategies that frequently defect, it will keep changing its choice and lose utility every time it cooperates with the defector. On the other hand, The Grudger understands it should cooperate with cooperative strategies, and defect against selfish strategies. However, it proves to be the weakest of the cooperative strategies because it does not give the opponent a second chance to cooperate (e.g. Adaptive or Random, hence, there will be no more double cooperation for the encounter.

5.3.5 Soft Grudger Test

Example Run:

Table of Results:

	UTILITY		
	Test Set 1	**Test Set 2**	**Test Set 3**
TEST 1	3073	3202	2791
TEST 2	2624	3031	2453
TEST 3	3101	2901	2413
TEST 4	3265	3112	2932
TEST 5	2751	2823	3101
TEST 6	2391	3567	2879
TEST 7	2461	3036	2771
TEST 8	3086	2803	3251
TEST 9	2573	2523	2668
TEST 10	2728	2835	3334
AVERAGE	**2805**	**2983**	**2859**

The results for the Soft Grudger Test show a slight increase in utility from the Grudger, with an average utility of 2882 over the three sets. The difference between the Soft Grudger, and the Grudger is that the Soft Grudger allows the opponent a second chance to cooperate. This makes the strategy more compatible with the Adaptive strategy. Since the Adaptive strategy had a 200% higher chance of being generated to the list, the average utility over multiple tests proved to be moderately higher than its less forgiving counterpart. It might also be a more efficient strategy against the Random strategy, much such a claim is hard to prove due to the uncertainty.

There is one other strategy that has a similar approach to the grudgers, Gradual. However, this strategy proved to be more efficient, and the reason will be discussed when we look at its respective test.

5.3.6 Adaptive Test

Example Run:

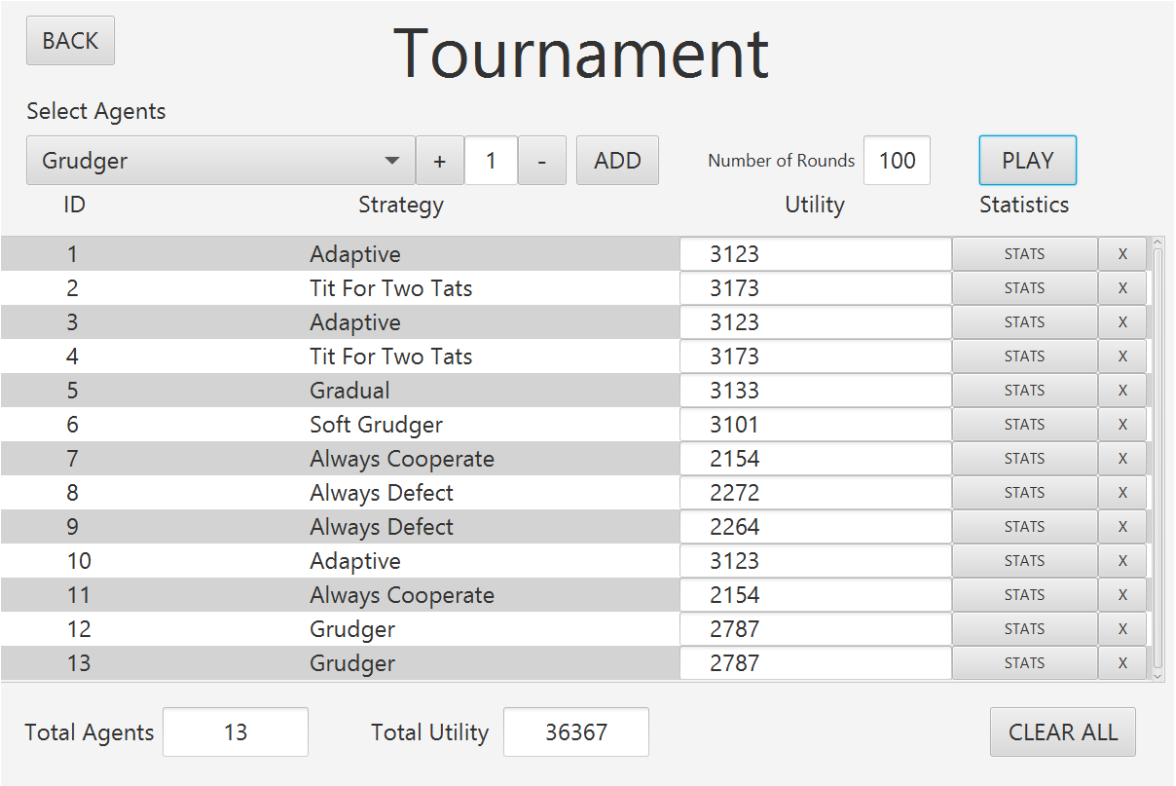


Table of Results:

	UTILITY		
	Test Set 1	**Test Set 2**	**Test Set 3**
TEST 1	3123	2731	2111
TEST 2	3003	3749	2538
TEST 3	2913	2728	3530
TEST 4	2933	2914	2532
TEST 5	3281	3152	2752
TEST 6	3526	2926	3095
TEST 7	2584	3461	3112
TEST 8	3002	2331	3124
TEST 9	2904	2938	2595
TEST 10	2825	3088	2723
AVERAGE	**3009**	**3002**	**2811**

The results for the Adaptive Test were the most surprising in regard to the expectations set before the running the tests. The Adaptive strategy is the most unconventional because it is the only strategy that actively tries to exploit cooperative strategies to gain the most utility possible, while still being able to recognize when the opponent has a robust defensive strategy.

The results show an average of 2941 over all thirty tests, which is a moderate improvement on the previous tests and only slightly below the next strategy's average. It is interesting that such an exploitative strategy can be placed at the top of the ranking because it means that being selfish can be profitable, as long as there is a robust strategy in place. However, this strategy will mostly likely lose a lot of effectiveness if it is solely set up against robust strategies such as Gradual of Tit For Tat.

5.3.7 Gradual Test

Example Run:

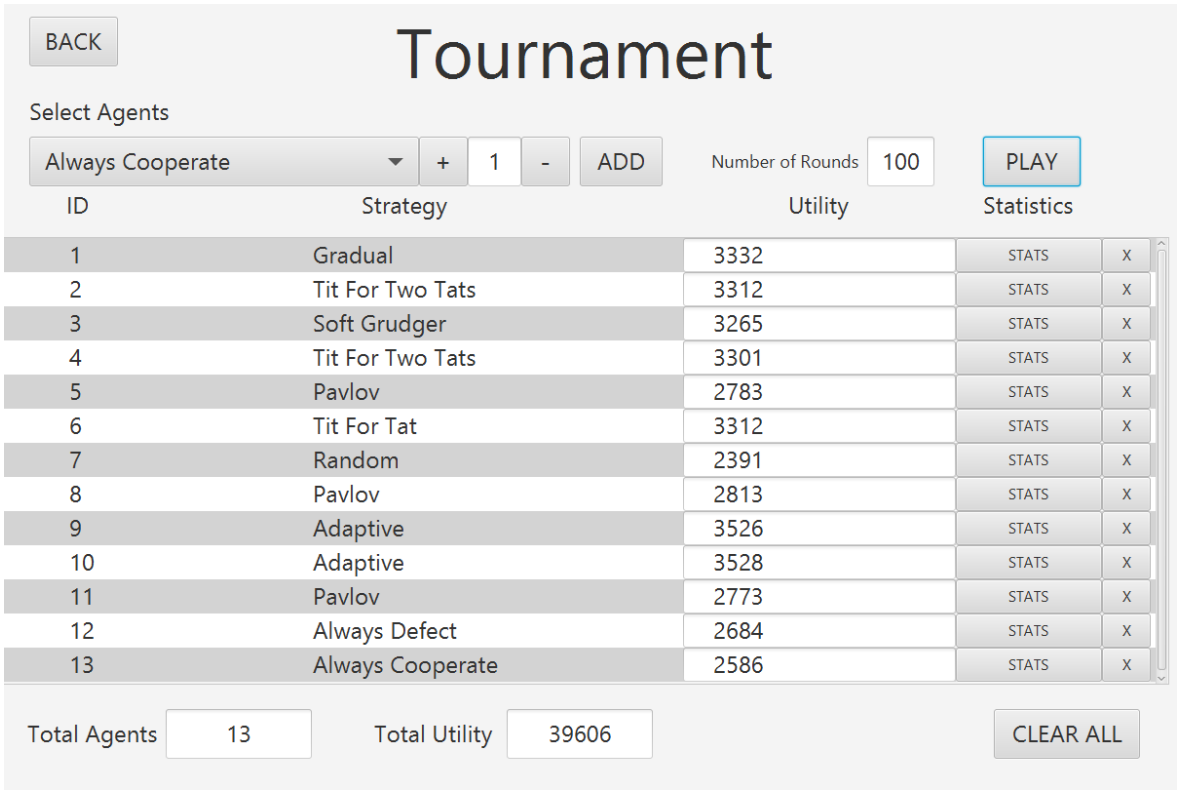


Table of Results:

	UTILITY		
	Test Set 1	**Test Set 2**	**Test Set 3**
TEST 1	3332	3300	2864
TEST 2	2696	3148	2442
TEST 3	3133	2711	2855
TEST 4	2907	3555	2664
TEST 5	3210	3158	2515
TEST 6	3290	3280	3024
TEST 7	2625	2711	2937
TEST 8	2496	2263	3369
TEST 9	3254	2897	2756
TEST 10	3113	2877	3297
AVERAGE	**3005**	**2990**	**2872**

The results for the Gradual Test displayed an average just above the Adaptive strategy at 2956. As it was briefly mentioned previously, this strategy has a similar approach to the grudger strategies. It still gives the opponent multiple chances to cooperate which makes it better than the Grudger, but it also increments the amount punishment it offers at the opponent if they keep defecting, which makes it better than the Soft Grudger. It takes the advantages of both strategies and balances their strengths to create a more effective version.

5.3.8 Tit for Two Tats Test

Example Run:

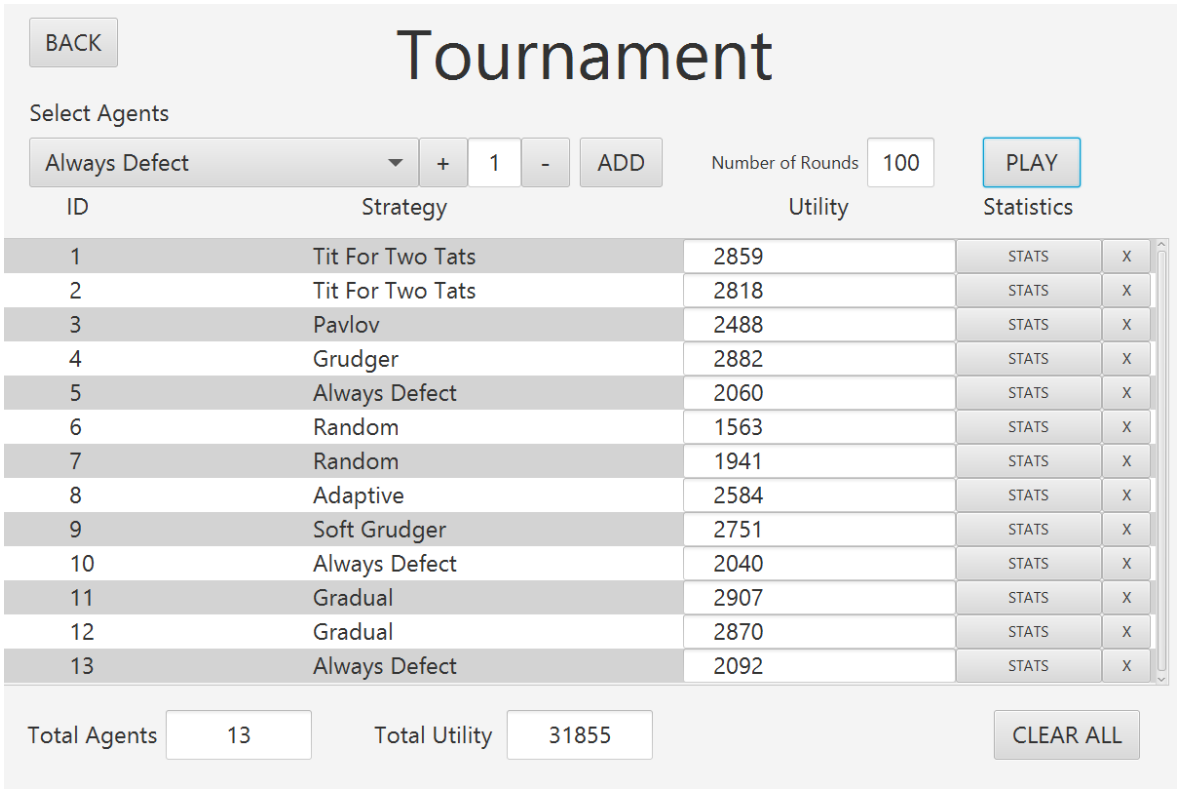


Table of Results:

	UTILITY		
	Test Set 1	**Test Set 2**	**Test Set 3**
TEST 1	2859	3228	2490
TEST 2	3373	3100	2560
TEST 3	3098	3180	3009
TEST 4	3173	2895	3172
TEST 5	3312	2770	2889
TEST 6	3223	3106	3221
TEST 7	3108	2876	3216
TEST 8	2694	2902	3374
TEST 9	2759	3118	2970
TEST 10	3198	2888	3100
AVERAGE	**3080**	**3006**	**3000**

The results for the Tit for Two Tats test proved to be the most successful at an average utility of 3029, the only strategy to average above 3000. This average is the highest average utility of every test, including Tit for Tat (discussed below).

Just like Tit for Tat, this strategy uses an algorithm of reciprocity, where it mimics the opponent's previous choice. This type of strategy was regarded as the most effective during the research done prior to the development of the program. Although this strategy scored the highest utility, Tit for Tat ended with an average utility below that of both Gradual and Adaptive. This shows that, perhaps, there is a flaw in the application of immediate reciprocity.

5.3.9 Tit for Tat Test

Example Run:

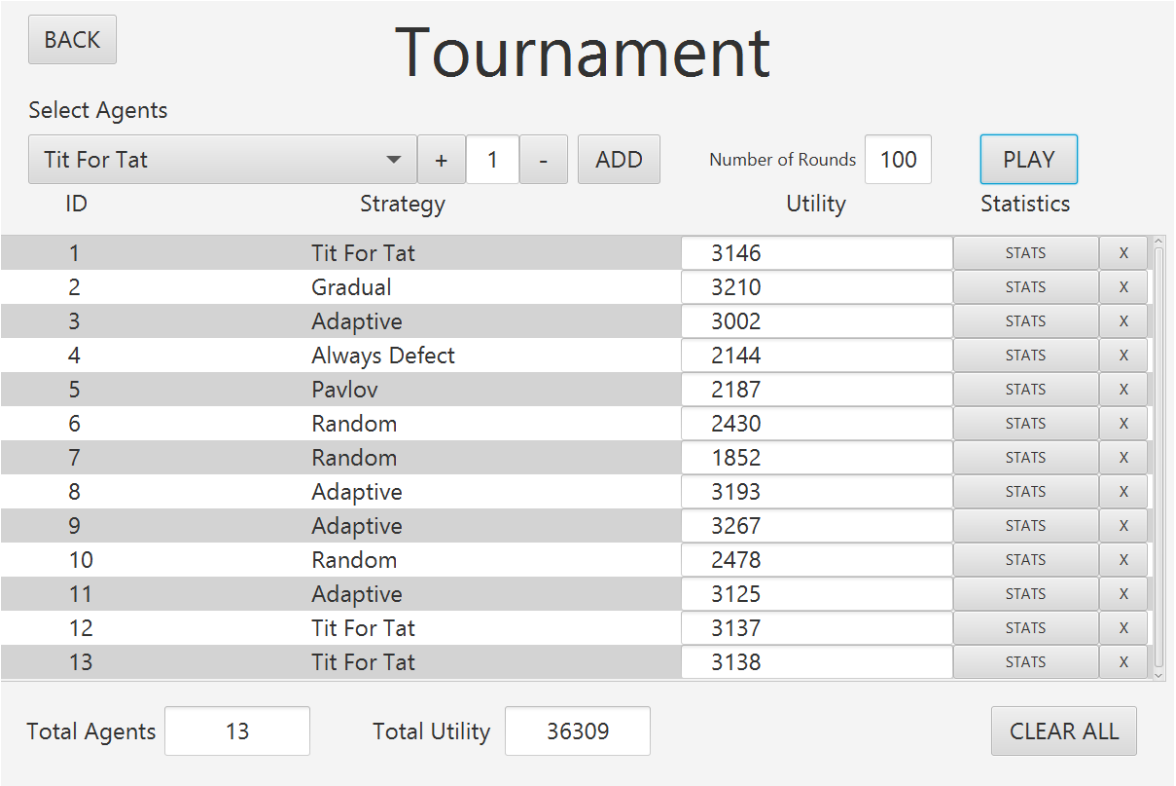


Table of Results:

	UTILITY		
	Test Set 1	**Test Set 2**	**Test Set 3**
TEST 1	3146	3215	3186
TEST 2	3330	3221	2769
TEST 3	2760	3100	2317
TEST 4	2970	2997	2631
TEST 5	3098	3180	2905
TEST 6	3312	2900	2662
TEST 7	3194	2017	3104
TEST 8	2576	2769	2887
TEST 9	3119	3573	2577
TEST 10	2690	2769	3171
AVERAGE	**3020**	**2974**	**2820**

The results of the Tit for Tat Test finish with an average result of 2938. This average is significantly lower than the average of Tit for Two Tats, which questions whether these results are truly reliable, or if the sample size is too small to remove uncertainties. Both strategies have a very similar algorithm. In the first set of tests (with the fixed agent list) the two strategies had the exact same amount of utility for most of the example runs. Also, most of the randomly generated lists that contained both of these strategies and did not contain the Random strategy, displayed a very approximate amount of utility.

Ultimately, this result either shows: that using random generation to perform test with relatively small samples might be unreliable; or that Tit for Two Tats is more effective at dealing with opponents that choose randomly. If the second option is true, then Tit for Two Tats is undeniably the most effective strategy for the Prisoner's Dilemma.

5.3.10 Final Ranking

1. Tit for Two Tats
2. Gradual
3. Adaptive
4. Tit for Tat
5. Soft Grudger
6. Grudger
7. Pavlov
8. Always Defect
9. Always Cooperate

6 Conclusion

6.1 Summary

The aim of this project was to evaluate the best method of cooperating in any given environment. In order to find a solution we delved into the concept of game theory and came across the Prisoner's Dilemma. The dilemma is a game played by two entities that are given a result based on whether they chose to cooperate or defect as well as the opponent's choice. The idea behind the dilemma is that having mutual cooperation will yield the best global outcome and mutual defection will yield the worst global outcome, however, defecting against an opponent that cooperates will yield the best personal outcome and vice versa. Assuming the entities are in no way connected, it was concluded that defecting is always the safest bet. However, the game can be iterated a number of times so entities have the chance to understand how the opponent plays. Background research also led to the discovery of already existing strategies used in iterated versions of the dilemma.

After having a clear understanding of the Prisoner's Dilemma and the strategies used in the iterated version, we sought to create a platform where we can establish which strategy is the most effective. In this platform, completely unbiased agents could implement these strategies and play against each other in a number of tests. First we developed a platform that allows two agents to play the game iteratively. This allowed us to, both, test whether the algorithms were functioning properly, and to have some idea of which strategies were most compatible.

The most important development for the program was the tournament. This game mode allowed a list of agents, with specific strategies, to be forced to play against every other agent on the list for a specified number of rounds. This was the module most used during the evaluation stage. The first set of tests allowed us to analyse which strategies work best at a specific number of iterations. From the results, we gathered that defecting is the best option at a low number of iterations. However, the strategy was quickly surpassed by strategies that react to the opponents choices. The best strategies at a high number of iterations proved to be the reciprocating algorithms, Tit for Tat and Tit for Two Tats.

With these results in place, we created a new set of tests. This time the number of iterations was kept constant, and, instead, we attempted to generate completely random lists of agents. In order to maintain a balanced list, the chances of generating a selfish strategy were increased to match the amount of naturally cooperative strategies. The final results of this second test matched the previous test with a few exceptions. Since the list was balanced for the second set of tests, always defecting proved to be more profitable than always cooperating, however, this information is almost irrelevant considering every other strategy was more effective. The most profitable strategy proved to be Tit for Two Tats. Surprisingly, its result was significantly higher than Tit for Tat (a very similar strategy). The reason for this outcome might be because Tit for Two Tats allows the opponent to defect once without immediately punishing them. The weakness could be exploited, but no such strategy had that ability.

Finally, we created a platform where agents could choose which agents it wants to play against based on previous experience. The results of the tests run on the environment were not discuss during evaluation because the outcome did not provide any unexpected information. Some example runs of these tests can be found in Appendix B. The tests showed that the utility gained by selfish strategies (i.e. Always Defect and Adaptive), was greatly reduced and that Tit for Two Tats is still the most effective strategy.

By applying the results of this project to real life, we can conclude that reciprocating an opponent's move is the most effective strategy, and allowing a small amount of leniency and forgiveness will yield the highest return. This conclusion proves that cooperating is the best way guaranteeing success as long as there is a robust defence mechanism to prevent exploitation.

6.2 Future Work

Though the final product of the program was able to provide a reasonable conclusion to the thesis, there is a list of possible improvements and added features that could have provided a more reliable conclusion to this project.

One feature that was considered very early on was to add a strategy that could identify the opposing strategy and adopt a strategy to guarantee the best results, the Prober. This strategy would play a certain set of moves at the beginning until it could identify which one, of the unique strategies it was playing against. Once the strategy was identified, it would begin making choices that could take advantage of the opponent's weaknesses. There was a short attempt at creating this unique strategy, however, due to the amount of strategies and the similarities between them, it was hard to create a set of initial steps that could uniquely identify the opposing strategy. This problem was only aggravated when it played against the Random strategy, in which case it could not identify the opponent and did not know how to proceed properly. Nonetheless, it is a feature that could yield interesting results if implemented properly.

Another feature that was considered was to have agents change their strategy when they are rejected in the environment. There were two problems with the execution of this feature. Firstly, only two of the strategies are ever rejected in the environment, therefore, whenever the agent changed to a new strategy it did not necessarily mean it changed to the optimal strategy. Secondly, the purpose of the project was to find the most effective strategy, so if agents keep switching their strategy no reasonable solution can be interpreted.

Finally, there is the option to have modelled the tournament in a different way, a knockout tournament. In this tournament agents would be matched against another agent, iterate over one game and the agent that gained most utility would move on to the next stage. Just like the previous feature there are two major problems with this tournament design. First, always defecting guarantees that the agent will never end with less utility than the opponent, meaning Always Defect strategy would win every tournament. The second problem is that most of the strategy combinations would end with a tied result. Most agents would always cooperate with each other, thus ending in a tie. Also, since every tournament is won by always defecting, every tournament would also end in a tie (if both sides of the bracket had a defecting agent). Nevertheless, this feature could yield interesting results if the agents were set up in a manner that allowed them to deviate from the strategy, however, this would also not provide a concrete evaluation for this project's thesis.

7 Professional Issues

The development of this project was not highly afflicted by professional issues, however, there are certain definitely some issues that are worth discussing upon having completed the project. The issues relevant to this project revolve around the topic of management, safety and usability.

Due to the lack of any previous experience, management was undoubtedly the biggest issue during the development of the project. During the planning phase it was difficult to allocate time and effort to each task due to the lack of knowledge on how to produce the program. The initial plan stated that the development of the program would take four months. Of those four months, seven were attributed solely to the development of all the strategies. Once it came to the development phase, the strategies were developed and fully tested for completion in one week. The plan also failed to attribute time to testing and debugging, which took the most time and effort of every task. As it turned out, the initial idea of the program turned out to be easier to program than expected, which led to the idea of creating the environment feature. This only led to another issue.

The initial idea behind creating an environment was to have agents behave concurrently, in a space where they actively sought out opponents and communicated their thoughts to every other agent. After some research and discussion with the project supervisor, this idea proved to be too ambitious for development in just a couple of months. With this, the end product of the environment turned out to be lacklustre, and did not provide any meaningful answers that the tournament could not already provide.

Safety is the second topic that is worth mentioning for the issues involved in this project. Whereas management was an issue during the development, safety is an issue regarding the final product. The program developed for this project delivers a system that could possibly be used for measurements in professional environments, for example, economy. The Prisoner's Dilemma is a system used to measure the effectiveness of certain actions in the market. We can imagine two competitive companies that sell a similar product, and after some discussion they agree on a price for the product. When they release their product they can choose to sell at the agreed upon price (cooperate), or sell it at a lower price (defect). In order to establish a strategy that optimizes their utility gain under these circumstances, a company might want to employ a similar program to that which was developed in this project. However, if strategies are not properly implemented, this becomes a risk for the company. Therefore, it is essential that the program is 100% tested to be fully functional. In order to offer such a guarantee, the program would have to go through multiple lines of testing and most likely revised by skilled programmers that can guarantee there are no problems with the system.

The last issue in this project is in regard to usability. Artificial intelligence is one of the hottest ethical topics in recent years. There are multiple aspects of concern on the topic, but for this specific project, the main concern is how artificial intelligence is replacing people. When there is a system that can find the ideal method of cooperation, companies will no longer be required to hire analysts who seek this information. The development of an environment as was previously described is a better example of how artificial intelligence could be more efficient than a person. If agents could run in such an environment, in one minute each agent could have run more simulations than a person could ever possibly hope to accomplish. This abundant amount of information can then be used to determine which strategy is best in a certain circumstance. Humans also make mistakes and computers do not (as long as they are well programmed), which is another reason why such systems have begun to replace human efforts.

Bibliography

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

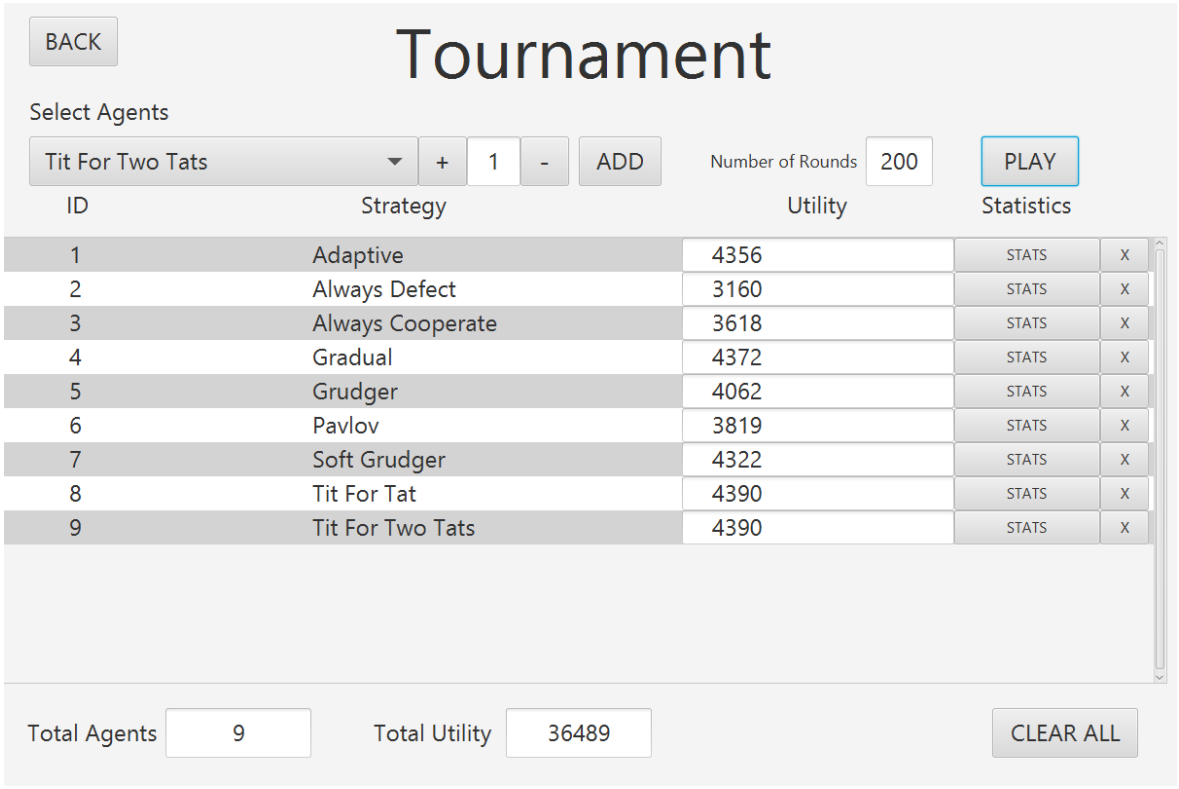
[2] *Prisoner's Dilemma*. n.d. https://www.investopedia.com/terms/p/prisoners-dilemma.asp. 2017.

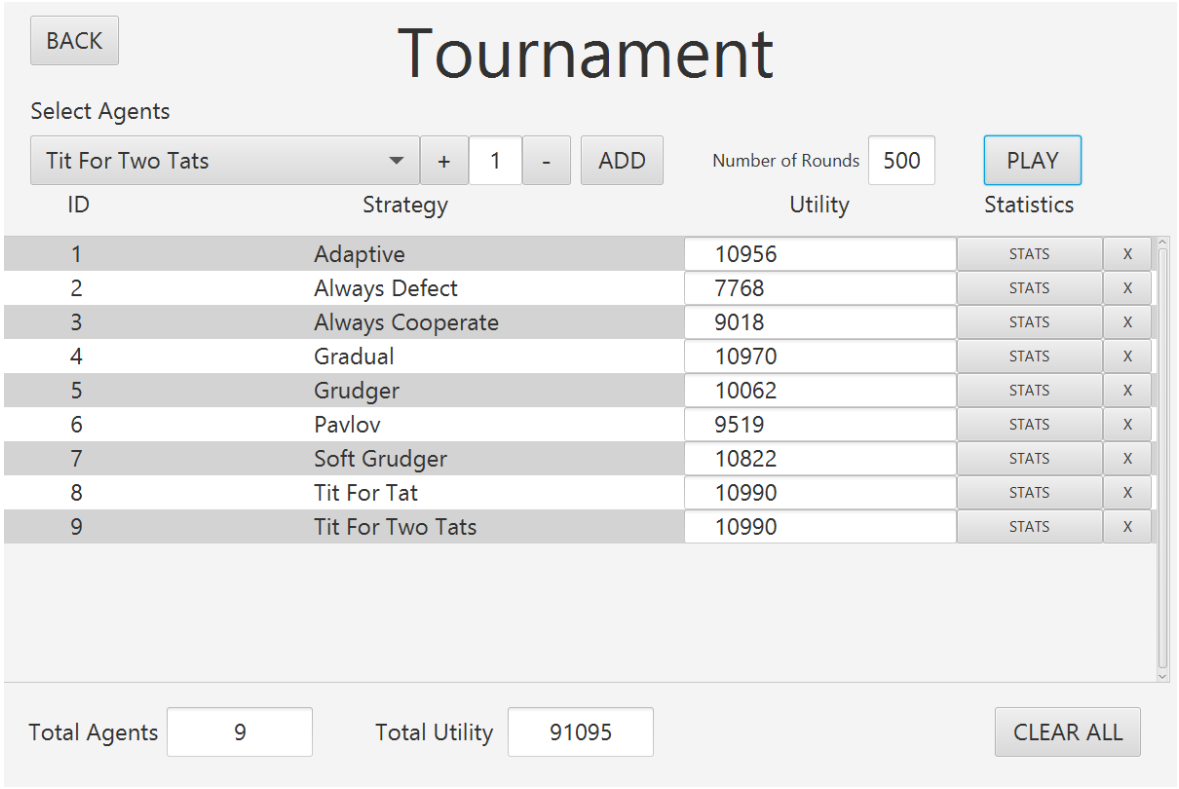
[3] *Prisoner's Dilemma*. n.d. https://en.wikipedia.org/wiki/Prisoner's_dilemma.

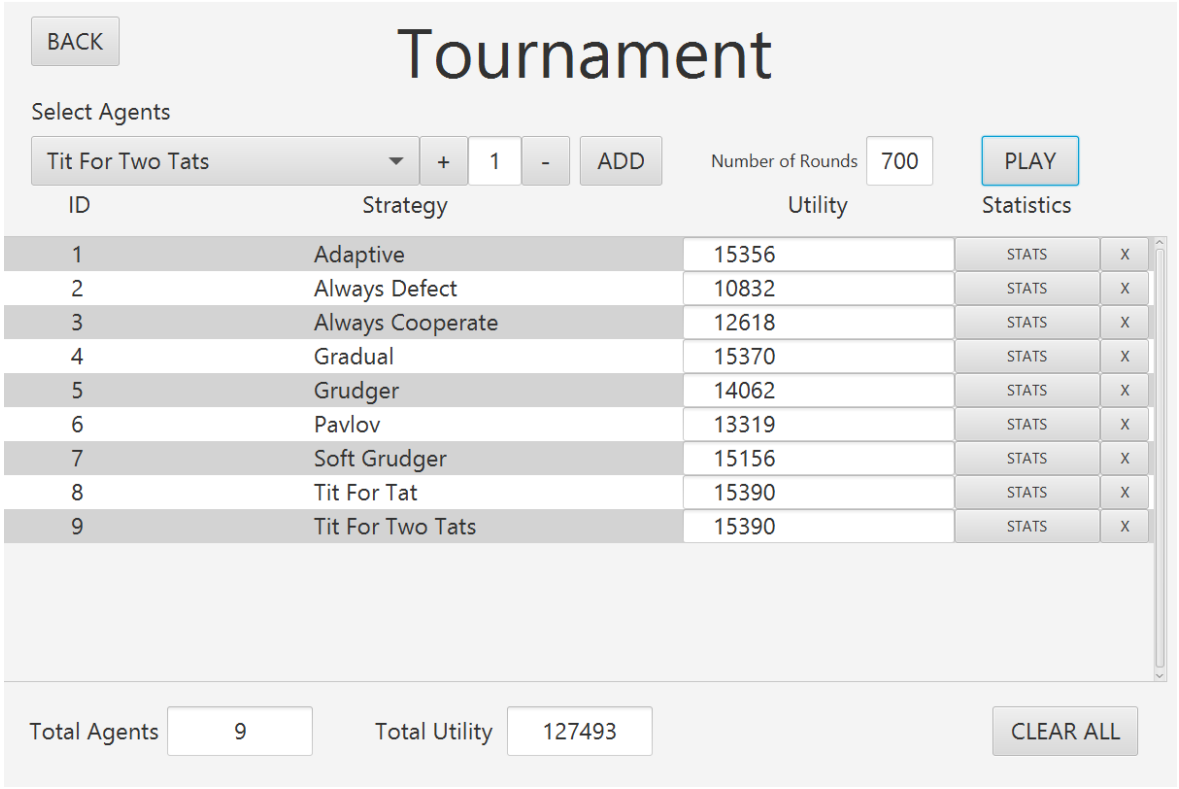
[4] *RANDOM.ORG*. (1998). Retrieved from RANDOM.ORG: https://www.random.org

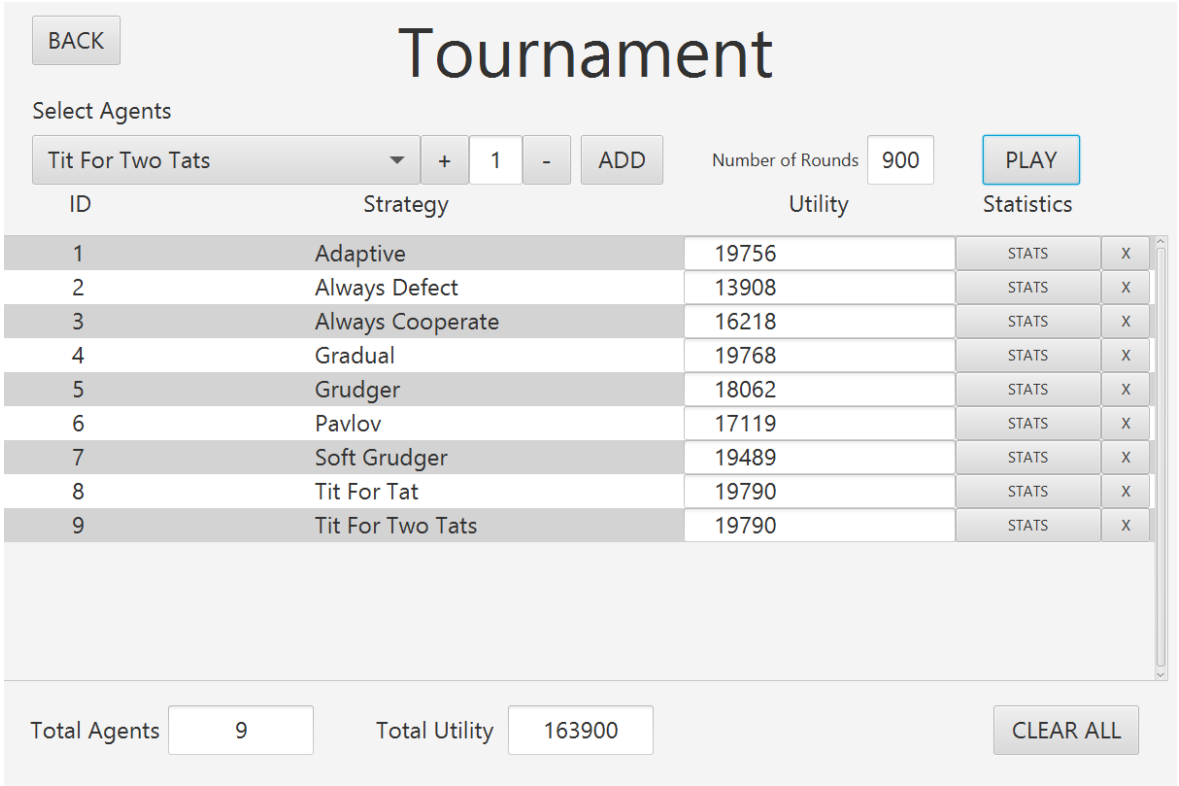
[5] Robert Axelrod, William D. Hamilton. *The Evolution of Cooperation*. Basic Books, 1981. *Tit for Tat*. n.d. https://en.wikipedia.org/wiki/Tit_for_tat. 2017.

Appendix A

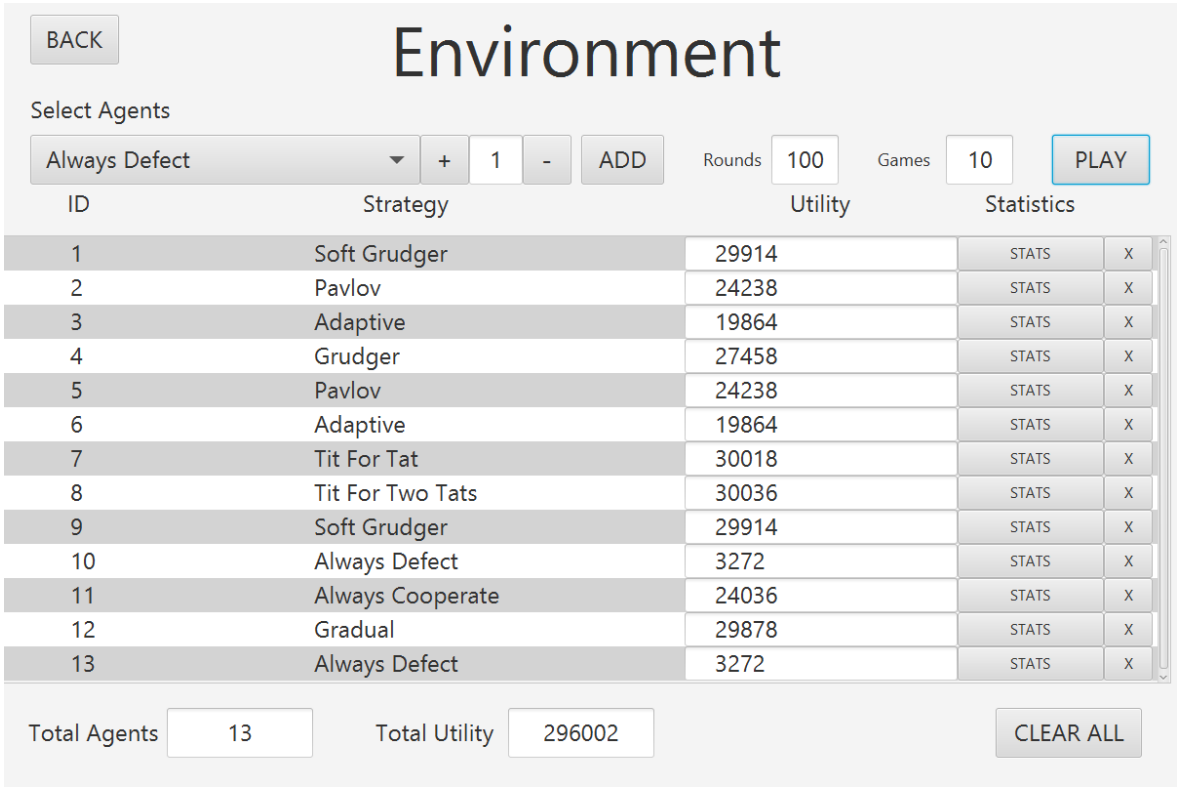

BACK
Tournament
Select Agents
Tit For Two Tats
+
1
-
ADD
Number of Rounds
200
PLAY
ID
Strategy
Utility
Statistics
1 Adaptive 4356 STATS X
2 Always Defect 3160 STATS X
3 Always Cooperate 3618 STATS X
4 Gradual 4372 STATS X
5 Grudger 4062 STATS X
6 Pavlov 3819 STATS X
7 Soft Grudger 4322 STATS X
8 Tit For Tat 4390 STATS X
9 Tit For Two Tats 4390 STATS X
Total Agents
9
Total Utility
36489
CLEAR ALL

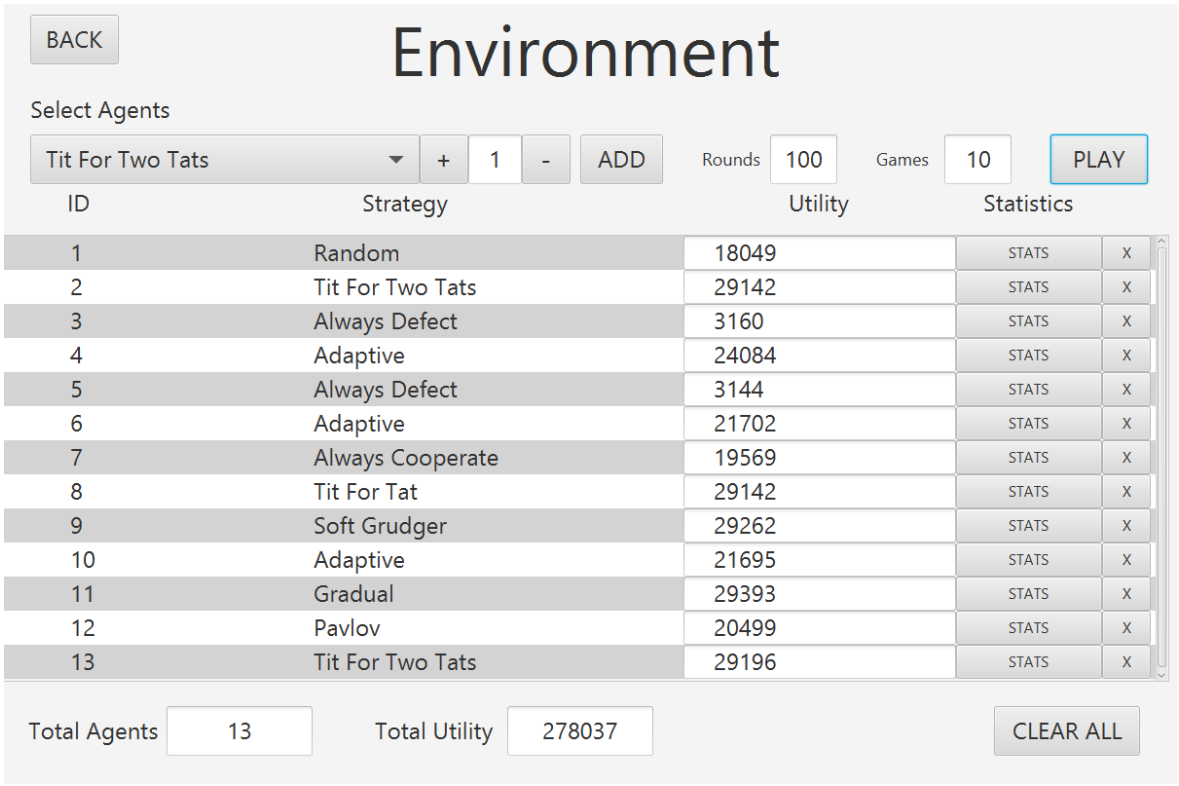

BACK
Tournament
Select Agents
Tit For Two Tats
+
1
-
ADD
Number of Rounds
500
PLAY
ID
Strategy
Utility
Statistics
1 Adaptive 10956 STATS X
2 Always Defect 7768 STATS X
3 Always Cooperate 9018 STATS X
4 Gradual 10970 STATS X
5 Grudger 10062 STATS X
6 Pavlov 9519 STATS X
7 Soft Grudger 10822 STATS X
8 Tit For Tat 10990 STATS X
9 Tit For Two Tats 10990 STATS X
Total Agents
9
Total Utility
91095
CLEAR ALL

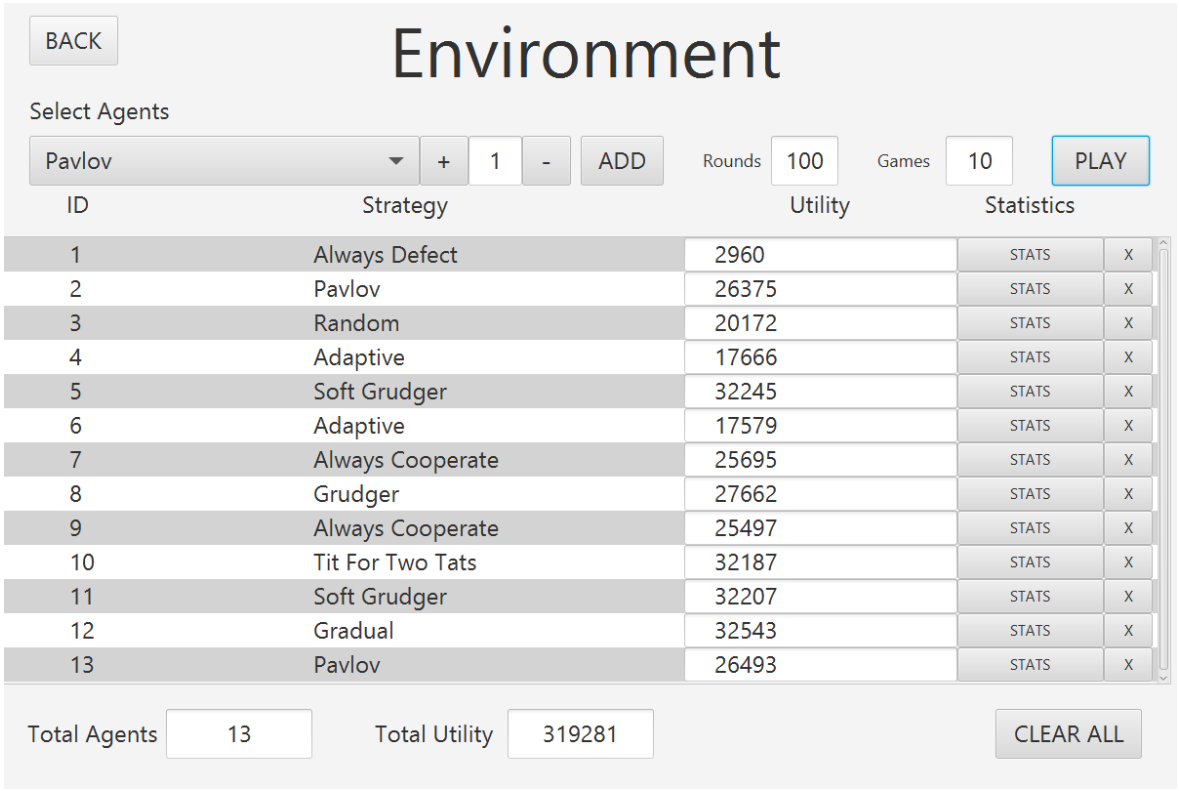

BACK
Tournament
Select Agents
Tit For Two Tats
+
1
-
ADD
Number of Rounds
700
PLAY
ID
Strategy
Utility
Statistics
1 Adaptive 15356 STATS X
2 Always Defect 10832 STATS X
3 Always Cooperate 12618 STATS X
4 Gradual 15370 STATS X
5 Grudger 14062 STATS X
6 Pavlov 13319 STATS X
7 Soft Grudger 15156 STATS X
8 Tit For Tat 15390 STATS X
9 Tit For Two Tats 15390 STATS X
Total Agents
9
Total Utility
127493
CLEAR ALL

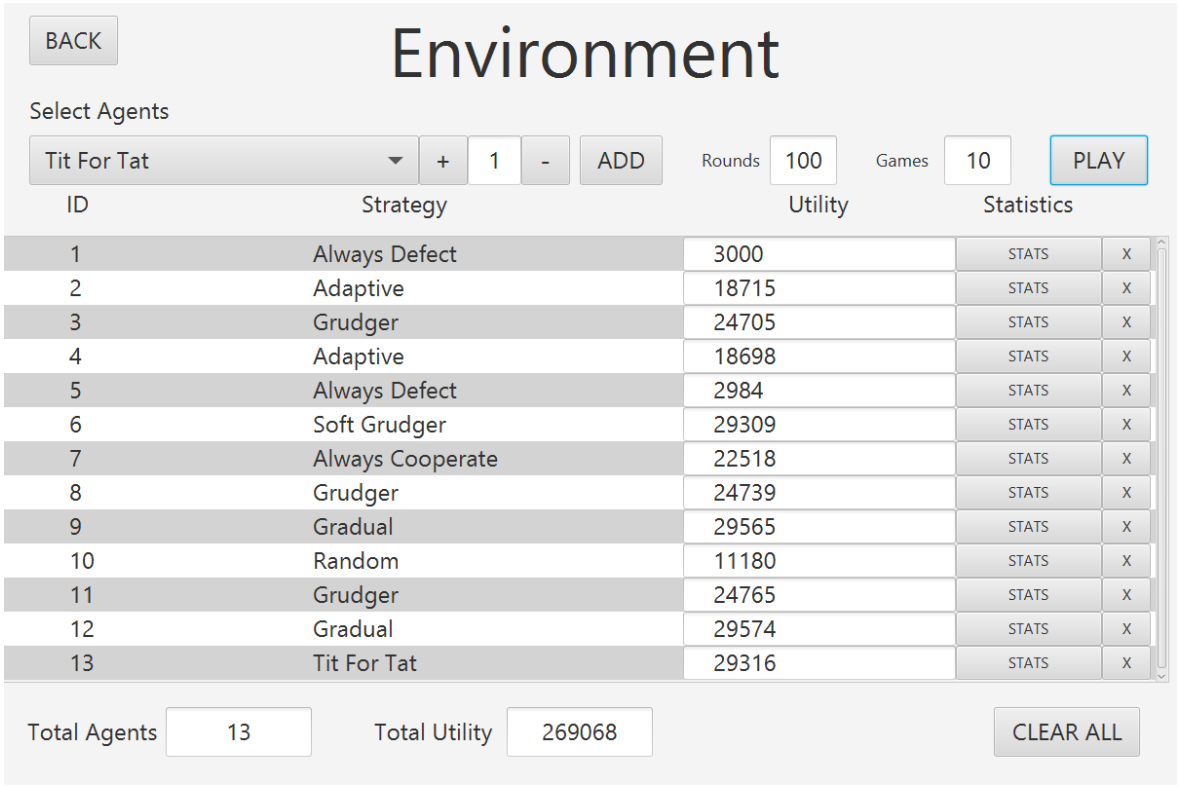

BACK
Tournament
Select Agents
Tit For Two Tats
+
1
-
ADD
Number of Rounds
900
PLAY
ID
Strategy
Utility
Statistics
1 Adaptive 19756 STATS X
2 Always Defect 13908 STATS X
3 Always Cooperate 16218 STATS X
4 Gradual 19768 STATS X
5 Grudger 18062 STATS X
6 Pavlov 17119 STATS X
7 Soft Grudger 19489 STATS X
8 Tit For Tat 19790 STATS X
9 Tit For Two Tats 19790 STATS X
Total Agents
9
Total Utility
163900
CLEAR ALL

Appendix B


BACK
Environment
Select Agents
Always Defect
+
1
-
ADD
Rounds
100
Games
10
PLAY
ID
Strategy
Utility
Statistics
1 Soft Grudger 29914 STATS X
2 Pavlov 24238 STATS X
3 Adaptive 19864 STATS X
4 Grudger 27458 STATS X
5 Pavlov 24238 STATS X
6 Adaptive 19864 STATS X
7 Tit For Tat 30018 STATS X
8 Tit For Two Tats 30036 STATS X
9 Soft Grudger 29914 STATS X
10 Always Defect 3272 STATS X
11 Always Cooperate 24036 STATS X
12 Gradual 29878 STATS X
13 Always Defect 3272 STATS X
Total Agents
13
Total Utility
296002
CLEAR ALL


BACK
Environment
Select Agents
Tit For Two Tats
+
1
-
ADD
Rounds
100
Games
10
PLAY
ID
Strategy
Utility
Statistics
1 Random 18049 STATS X
2 Tit For Two Tats 29142 STATS X
3 Always Defect 3160 STATS X
4 Adaptive 24084 STATS X
5 Always Defect 3144 STATS X
6 Adaptive 21702 STATS X
7 Always Cooperate 19569 STATS X
8 Tit For Tat 29142 STATS X
9 Soft Grudger 29262 STATS X
10 Adaptive 21695 STATS X
11 Gradual 29393 STATS X
12 Pavlov 20499 STATS X
13 Tit For Two Tats 29196 STATS X
Total Agents
13
Total Utility
278037
CLEAR ALL


BACK
Environment
Select Agents
Pavlov
+
1
-
ADD
Rounds
100
Games
10
PLAY
ID
Strategy
Utility
Statistics
1 Always Defect 2960 STATS X
2 Pavlov 26375 STATS X
3 Random 20172 STATS X
4 Adaptive 17666 STATS X
5 Soft Grudger 32245 STATS X
6 Adaptive 17579 STATS X
7 Always Cooperate 25695 STATS X
8 Grudger 27662 STATS X
9 Always Cooperate 25497 STATS X
10 Tit For Two Tats 32187 STATS X
11 Soft Grudger 32207 STATS X
12 Gradual 32543 STATS X
13 Pavlov 26493 STATS X
Total Agents
13
Total Utility
319281
CLEAR ALL


BACK
Environment
Select Agents
Tit For Tat
+
1
-
ADD
Rounds
100
Games
10
PLAY
ID
Strategy
Utility
Statistics
1 Always Defect 3000 STATS X
2 Adaptive 18715 STATS X
3 Grudger 24705 STATS X
4 Adaptive 18698 STATS X
5 Always Defect 2984 STATS X
6 Soft Grudger 29309 STATS X
7 Always Cooperate 22518 STATS X
8 Grudger 24739 STATS X
9 Gradual 29565 STATS X
10 Random 11180 STATS X
11 Grudger 24765 STATS X
12 Gradual 29574 STATS X
13 Tit For Tat 29316 STATS X
Total Agents
13
Total Utility
269068
CLEAR ALL