Final 2022 Project Report

Full Unit – Interim Report

Cooperative Strategies in Multi-agent Systems

Matthew Lowe

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

October 28, 2022
Matthew Lowe, 2022

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 Where other published and unpublished source materials have been used, these have been acknowledged.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: 14953

Student Name: Matthew Lowe

Date of Submission: 25/03/2023 (5 day extension taken)

Your extension application Ref 93441 has been considered and the decision is now available.

- Your application has been **Approved.** Your submission deadline is **31 March 2023 10:00.**
- You can view your application here

Signature:

Table of Contents

Abstract

The prisoner's dilemma is a well-known application within the field of mathematics, more specifically the field of game theory. In the original problem two players are isolated from each other and they must decide whether to deny or confess given the four possible outcomes; [1] They both remain silent, [2] They both defect, [3] Player A defects and Player B remains silent, [4] Player A remains silent and Player B defects.

The original prisoner's dilemma was first introduced by Melvin Dresher and Merrill Flood in 1950. Later in 1992 it was named by Albert Tucker who was a Princeton mathematician (References [3]). The prisoner's dilemma is a single round event however, the iterated prisoner's dilemma is a multiround process where players must devise a strategy that aims to maximize their reward.

There are many different strategies that a player can decide on, amongst others

- Tit for tat: Cooperates on the first round and then copies the opponent's previous choice for the remaining iterations.

- Random: Player randomly defects
- Unconditional Co-operator/Defector: The player will always cooperate or always defect
- Grudger: Will always cooperate until the opponent defects and will then always defect thereafter

(Reference [4])

A real-life application of the iterated prisoner's dilemma could be found within the oligopolistic markets such as soft drinks, fast food restaurants, and oil producers. For instance, in the soft drink market if we take Coca Cola and Pepsi, they might decide to run a marketing campaign 6 times throughout the year. If they both decide to advertise then their profits will both be inferior to the scenario where they both decide not to advertise. However, if one decides to advertise and the other not then the first one will see a larger increase in profits than the competitor. The best outcome for both Coca Cola and Pepsi would be to not advertise which is an unstable scenario as each company gets swayed by seeing the benefits of advertising when the other does not which ultimately creates the prisoner's dilemma.

Project Specification

Aims: To implement a virtual tournament of a set of programmable players engaging in a round robin for the Iterated Prisoner's Dilemma game, test different strategies for these players to follow, identify winning strategies and evaluate the final results.

Early Deliverables

1. Familiarisation with the idea of strategies and the identification of set of them.
2. Implementation of strategies and the definition of an interface that allows agents to be selected and play against each other
3. Definition of a tournament
4. Report on the design of a tournament.
5. Implementation of a tournament, where a user selects the number of agents, associates them with existing strategies, specifies the number of iterations and deploys a tournament
6. Report on the Iterative Prisoner's dilemma problem
7. Report on previous work with the Iterative Prisoner's dilemma

Final Deliverables

1. A program that allows us to simulate tournaments of the iterative prisoner's dilemma.
2. A GUI that allows a user to create and manage tournaments.
3. GUI should provide summary statistics that are tournament specific, i.e. a summary for each player/strategy and how it is compared with the other players/strategies in the tournament.
4. The report should describe some of the theory behind strategies in cooperative settings.
5. The report should provide an analysis and an evaluation of the different strategies, including an indication of which one is the best strategy, if any.
6. The report should contain a design of the program and a discussion of any software engineering issues encountered.

7. The report should describe any useful and interesting programming techniques that have been employed to develop the final prototype.

Chapter 1: **Introduction**

1.1 Introduction

Cooperative Strategies in Multi-agent Systems is the implementation of the iterated prisoner's dilemma game in the form of a round robin tournament. Agents in the system function as players that will choose a strategy with the goal of gaining the most points possible to win the tournament.
Looking into the game theory of the prisoner's dilemma, it causes an interesting scenario where the most points possible is gained when both players choose to split however, the element of trust needed between the players causes a doubt which often leads to players deciding on stealing. In a round robin tournament where multiple rounds take place stealing becomes less effective depending on chosen strategies by other agents. This creates a dynamic situation in the prisoner's dilemma game where well thought out strategies reap the rewards.

1.2 Project Motivation

My interest in this topic sparked during Economics at A-level when my teacher showed us the prisoner's dilemma and the split or steal game show. Our class became fascinated with the topic, and we spent several lessons discussing it further. Even though this project was not my first choice I was not disappointed to be researching it further and implementing it into code. This project will allow me to use all the java knowledge that I have gained over the last few years and implement a topic that has endless research surrounding it. Working independently on a large project will prepare me for working in the software development industry as I must figure out solutions to problems and fix errors without direct help. Also, I can use my own imagination on how I want to apply the project to a real-life industry and how I want to design the interface to efficiently display my research.

1.3 Aims and Objectives

1. Research iterated prisoner's dilemma strategies
2. Implement a set of strategies that can be chosen by agents in the system a. Tit for tat
    b. Splitter
    c. Stealer
    d. Random
3. Research Round Robin Tournaments
4. Create a simple to use GUI for creating the Tournament
    a. Select amount of players
    b. Assign players to strategies
    c. Show matches
    d. Show results
5. Identify winning strategies

Chapter 2:**Background Theory**

2.1 Theory of Cooperation

2.1.1 Description

Here I will discuss how cooperation occurs in different forms of life and discuss research that dates back decades.

2.1.2 Cooperation in Evolution

A co-operator is someone who "pays a cost for another individual to receive a benefit" (Reference [6]). Evolution is based on the fight for survival where only the fittest survive, which should benefit selfishness. This means that every organism should aim to promote its own evolutionary success rather than help its competitors at the expense of self-succession. However, cooperation takes place between members of the same species and even between different species; genes cooperate in genomes, cells cooperate in multi-cellular organisms.

Humans have engaged in cooperation dating back to the begging of the species in hunter gatherer societies that cooperated to hunt food, build shelters, and protect their community. Humans have cooperated to create infrastructure that no other life form on earth has ever achieved. Everything around us was created through cooperation, taking a moment to look around can be mind-blowing.

2.1.3 History of Cooperation

The concept of working for mutual benefit is present in all forms of daily interaction, even in everyday household interactions as simple as washing the dishes to mutually benefit house mates. You pay the cost of effort cleaning up for others to receive the benefit of clean plates.

Cooperation fascinated Charles Darwin who wrote "the small strength and speed of man, his want of natural weapons, are more than counterbalanced by his social qualities, which lead him to give and receive aid from his fellow-men" (Reference [5]), explaining that self-interested is balanced by the willingness to cooperate. The counter to Darwin's point made by others was that cooperation only took place because of the "struggle for survival" meaning his idea that desires are counterbalanced by a willingness to give and receive aid was false, with every act being done out of self-interest.

In 1902 a Russian prince called Kropotkin wrote a book to back up Darwin's claim. He explained that cooperation took place dating back centuries among Siberian herds, Polynesian islanders, and medieval guilds. The book was written to prove that cooperation did not need to be strictly enforced to take place.

2.1.4 Motivations behind Cooperating

The main questions posed to people willing to cooperate are Why would anyone share common interested rather than cheat the others? What motivation is preventing a switch to cheating others? How can motivation shape behaviour to pay a cost for another individual to receive a benefit? The two main approaches to these questions are Kin selection and Reciprocal aid.

Kin selection takes the example of genetics. A gene will look past costs to itself when the outcome is a possible immortal set of its replicas. If the ones who benefit from the act are closely related, then the costs to the individual are significantly outweighed by the benefits of the outcome. JBS Haldane said 'I will jump into the river to save two brothers or eight cousins', which was later known as Hamilton's rule r>c/b (Reference [6]). This explained that the coefficient of relatedness, r, must

exceed the cost to benefit ratio of the altruistic act. This can relate back to the genes as the probability of two brothers sharing the gene is 1/2 , cousins 1/8. The more related the genes are, the more they benefit from the acts of the predecessor. Hamilton's rule is now known as Kin selection.

Reciprocal aid provides a theory that looks at individuals with no relation to each other. This is a more economic styled exchange where both parties benefit through cooperation. However, each party will gain significantly more by defecting. This leads to cooperating being more vulnerable to defecting.

(All Above using References [5],[6],[7])

2.2 The Prisoner's Dilemma

2.2.1 Description

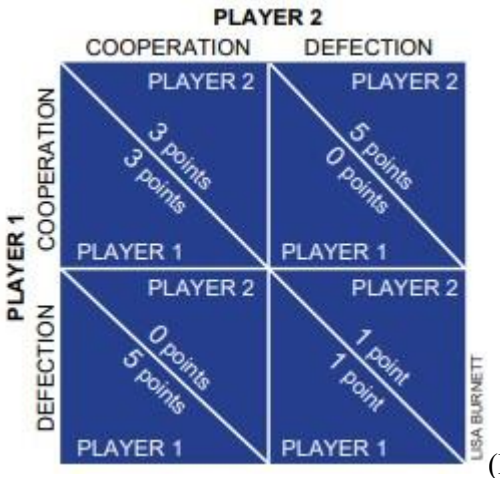
In this section I will discuss the idea of the Prisoner's dilemma to demonstrate the above-mentioned findings.

2.2.2 The Original Dilemma

The original problem takes place between two prisoners. Each prisoner is separated and taken into a room and asked the question of whether the other committed a crime. The punishment they receive is dependant on if one, both or neither admit to the other person's actions. The ideal scenario for the prisoners is that both stay silent and cooperate because they will both receive less jail time due to the lack of evidence of the crime committed. However, if one of them admits the other has committed the crime they get let free whilst the other goes down for more time. This floats the idea in each of their heads if they can trust the other not to speak. If both prisoners admit the crime of the other prisoner, then they get significantly more time than if they had both remained silent.

2.2.3 Prisoner's Dilemma Game

An adaptation of the original dilemma which sees jail time replaced with points, but the concepts remain. The game can best be displayed in the form of a matrix:



(Reference [7])

The game strongly poses the question of Will they cooperate? If player 1 chooses to defect, then player 2 has no choice but to also defect otherwise they will receive 0 points. Player 1 has the choice to cooperate first in hope that player 2 will also cooperate and possible continue to cooperate in the future. However, even if player 1 cooperate player 2 is still better of defecting as they receive all the

points and can gain a bigger advantage in the game. So, no matter what the first player chooses, the second player will always have a better outcome when defecting. The first player is in the exact same situation, they should always defect because if the second player chooses to cooperate then the first player gets all the points, and if the second player attempts to steal the points by defecting, then they both end up with 1 point each.

2.2.4 Individual vs Collective viewpoint

The Prisoner's dilemma game showcases decisions made from an individual point of view. Each player always chooses to defect because they are thinking about the outcome from a selfish point of view. If the players looked at the game from a collective view, they would see that in the long term always cooperating with each other will lead to a higher points tally. The reward for mutually cooperating outweighs the reward for mutually defecting, but the reward for a one-sided defection creates a temptation that presents itself as a better option than cooperation.

The idea that always defecting being the best strategy does not sit well with most people. People feel uneasy that both parties can not conclude cooperating with each other so that both benefits. When the game is carried out in real life scenarios it becomes even more interesting as this feeling of unease takes over. Taking the game show 'Golden Balls', two players are given the option to split or steal a cash prize. Each player can attempt to convince the other of the reasons why they are going to split the cash. The face-to-face aspect will affect how players act and they will often cooperate out of selflessness, pressure from the outside world, and even out of guilt from the opponents reasoning as to why they need the money. Smart players could play on different emotional factors to sway the other into splitting when they ultimately steal all the money for themselves.

2.3 Game Theoretic Tournament

2.3.1 Description

In this part I will explain how implementing Prisoner's Dilemma in tournaments creates more complex, strategized outcomes and which strategies I implemented thus far.

2.3.2 Tournament Play

In tournament play agents will select a strategy to stick with for the whole tournament. The tournament runs as a round robin format. A round robin format is where a number of rounds take place so that everyone plays each other once. This means that compared to the Prisoner's Dilemma where it is always better to defect, there are strategies that exist to gain more points that someone who always defects.

2.3.3 Implementing Round Robin in code

The way I implemented Round Robin in Java was making use of nested for loops and the java utility import ArrayList. To make sure everyone plays each other I had to use a clockwise motion so that each player rotated after each round. The players were split into two lists with the first player fixed in place to be rotated around. Then, increment the index by 1 and match that team with the player fixed in the first position. For the other match ups in the round you match the player at the same index in the other list. The loop will run for the total amount of players – 1. (reference [1])

2.3.4 Tit For Tat

Tit for Tat is a strategy that has been identified by many as a proven winning strategy. It works by always cooperating on the first move and then copies the opponents last move for the remainder of the game. The strategy in a single round will never win on points but when used in a tournament setting it return he most consistent amount of points against all the different strategies being used. This strategy can be used to convince other players into cooperating because the opponent knows that they will get rewarded for also cooperating. It also provides punishes people who always try to defect because a tournament filled with tit for tat users and only one always defector will always result with the defector finishing with the most points.

Implementing the strategy in java involved the use of an object to store a player's choice of action and then using a separate class containing the strategy as a method that takes two parameters, the player that chose tit for tat and the opponent. Then, setting the players first choice as co-operator and from then on copying the opponents previous move stored in the opponents choice of action object.

2.3.5 Unconditional Splitter

An unconditional splitter is a strategy that will cooperate with the opponent every single round. This strategy will be punished by a variety of different strategies and will come away with no points against someone who always defects. The idea of an unconditional splitter is that their opponent will see their past choices and decide to also cooperate which will return the best possible points for an unconditional splitter. When this strategy comes up against Tit For Tat it is the same as if two unconditional splitters came up against each other because Tit For Tat will copy the opponents last move which is always cooperate in this case.

Implementing this strategy in Java was simple as the decision is always to cooperate. The object that stores the player's choice of action needs to be populated with the decision to cooperate using a for loop of the length of the total rounds taking place between the match ups.

2.3.6 Unconditional Stealer

An unconditional stealer is a strategy that always defects every single round. This strategy will never lose a match up however over the duration of a tournament they will not always win the tournament due to other strategies chosen by other players. This is because the points given when two people decide to steal is 1 point whereas the decision to cooperate is 3. An unconditional stealer will punish unconditional splitters because the return when you cooperate and the other steals is 0 to 5. A stealer will be punished in a tournament environment if all the other players are co-operators and they are the only stealer because although they will win the head to head match, they will not gain as many individual points as the people who are constantly cooperating with each other.

Implementing this in Java is simple as the decision is always to defect. The player's choice object is just populated with defect for the iteration of the rounds taking place in the match up.

2.3.7 Random

A Random player is a strategy where the decision to split or steal is completely random. There is no set idea behind the strategy, and it will usually place poorly in the tournament rankings. It will often win some individual match ups but just like an unconditional stealer, it will be punished in the overall rankings when other players decide to cooperate with each other unconditionally.

Implementing random in Java uses the Random class. I created an object of class Random and created a string array containing split and steal. I then used the nextInt method to randomly choose between either position 0,1 of the array and then assigned the players choice to the randomly chosen position of 0 or 1.

2.4 GUI

2.4.1 Description

Here I will discuss the aims of a good interface and how design patterns effect the interface

2.4.2 Purpose of a GUI

A graphical user interface gives the user a display that they can interact with and hide the complexity of the code behind it. My interface allows the user to select an even number of players from a drop down menu, create the players with their name and strategy, run the round robin tournament with the created players, and finally display the results of the tournament in descending order of points.

2.4.3 Goals of a GUI

The goal of a GUI is to provide the user with an easy to understand display so that they interact with it as intended by the creator. The user should be given feedback based on their interactions. To help the user understand how to navigate the interface labelling as a guide for what needs to be done next.

My interface contains buttons that are labelled to guide users for what needs to be clicked on next. I also labelled a button 'help' that will explain what each strategy does as well as a welcome page that describes the process of creating the tournament

2.4.4 Architectural Design Pattern

An architectural pattern captures the design structures of the software so that they can be used. This is especially useful for Game architectures, like the Round Robin Prisoner's Dilemma tournament. This pattern will describe the different layers of the system but not how each individual class interacts with each other.

2.4.5 Model View Controller (MVC)

The MVC pattern is an architectural design pattern that decouples Model and View to reduce complexity. MVC does this by separating the frontend with the code in the backend. This makes it easier to changes without them interfering with each other. The project gets separated down into three parts; A view, A model, and A controller. The view creates the display for the user to interact with, the model processes and stores data, and the controller responds to user actions and interacts with the model and the view. (Reference [8])

I have used this pattern in my program with my interface using fxml. The view class displays the GUI made with scenebuiler to the user. The user then interacts with the interface and the controller for that view will interact with the classes in the program, the model, which will store interactions in an object or return data to the controller to be displayed by the view.

2.4.6 Behavioural Design Pattern

Behavioural design patterns identify common communication patterns to increase flexibility and reduce coupling for the communication. For my program I made use of the Observer pattern

2.4.7 Observer

The observer pattern is a behavioural design pattern where an object can register itself interested in the events caused by another object. This is typically used by even listeners in JavaFX. I made use of this pattern because it provides the back end code with the input made by the user. An example of this in my program is when the user enters the player's name, the player object will be notified when the name has been entered so it can create a new player using the user's inputted name.

Chapter 3:**The Iterated Prisoner's Dilemma**

3.1 Round Robin

3.1.1 Implementation in Java

```
public static ArrayList<String> CreateMatches(ArrayList<String> Players){
int numRounds = (Players.size() - 1);          int halfSize =
(Players.size()/2);

        ArrayList<String> list = new ArrayList<>();
        ArrayList<String> matches = new ArrayList<>();

list.addAll(Players);
list.remove(0);

        int teamSize = list.size();
         for (int i = 0; i<numRounds;
i++){             int teamIdx = i %
teamSize;
            matches.add(Players.get(0) + " vs " + list.get(teamIdx));
             for (int idx = 1; idx < halfSize; idx++){
int firstTeam = (i + idx) % teamSize;                  int
secondTeam = (i + teamSize - idx) % teamSize;
    matches.add(list.get(firstTeam) +     "     vs    "     +
list.get(secondTeam));
            }
}
        return matches;
    }
```

The algorithm above is how I create all the match ups for the tournament. The parameter contains the names of each player participating in the game. The main concept behind creating the match ups is to split the player list in half and rotate the players 'clockwise' so that they each player each other once. A pivot player is used, in my case player 1 at position 0, to rotate around.

Looking into the code now, the method starts off by storing the number of rounds that need to be player in order for everyone to play each other once. The halfSize variable is used to 'split' the list in half. A new list is created to store all the players except the player at position 0 which is used as the pivot.

For the matches against player at position 0, they match up against whichever player is at the next index. For the next player they will match up against the player who is at the same index from the other end of the list.

Each time a match is created it gets added to an ArrayList matches which will be return once all the matches have been created. This Array List is then used to run the game's head to head code to tally the points of the match based on their chosen strategy.

(Above code is an adaptation of Reference [1] to work for my specific program)

3.2 Iterating over a player's choice's

3.2.1 Description

One of the main concepts behind my program is being able to iterate over the player's choices created from the chosen strategy and tally their points based on a head to head with the opponent's choices. I have implemented this by making recursive calls to a method that run's the basic Prisoners Dilemma.

3.2.2 Implementation in Java

```
public static void tallyRoundPoints(Player p1, Player p2){
String p1Choice,p2Choice;         int[] points = new
int[2];          int total1 = 0;          int total2 = 0;

        int arrLength = p1.getChoices().size();

        for (int i = 0; i<=arrLength-1; i++){
if ((arrLength-i)<=5){
                p1Choice = p1.getChoices().get(i);
p2Choice = p2.getChoices().get(i);

                points = Dilemma.compareChoice(p1Choice,p2Choice);

                total1 = total1 + Array.getInt(points,0);
total2 = total2 + Array.getInt(points,1);
p1.setRoundPoints(total1);
p2.setRoundPoints(total2);
            }
        }
    }
```

This method is my recursive algorithm to tally the points for a head to head between the players in the match up. The players are passed into the method as Player object parameters.

Looking at the code, it loops through the players choices and only looks at the 5 most recent choices made by the player because my player object stores all the choices that have been made by the player from every match up, which is something I will improve in the second term.

The choice at position i for each player is stored and then put through the Dilemma method. The method returns an int array of size 2, position 0 is the points for player1 and position 1 is the points for player2. These points are then added to a running total for the specific player which is then stored in the player object method setRoundPoints().

This method is important to the running of my program as it does not only tally the points it also is used to return the specific points gained by players from the head to head matchup to the controller which will then be displayed on the interface.

3.3 Prisoner's Dilemma

3.3.1 Description

This section of the code is the basic prisoner's dilemma implemented in java. It has changed over the weeks of my program to work with new classes and their methods.

3.3.2 Implementation in Java

```
public static int[] compareChoice(String choice1, String choice2){
int[] points = new int[2];

                                                                    &&
if(choice1.equalsIgnoreCase(choice2)
choice1.equalsIgnoreCase("split")){
points[0] = 3;              points[1] = 3;
        }else     if    (choice1.equalsIgnoreCase(choice2)          &&
choice1.equalsIgnoreCase("steal")){             points[0] = 1;
points[1] = 1;
        }


                                                                    &&
if(choice1.equalsIgnoreCase("split")
choice2.equalsIgnoreCase("steal")){
points[0] = 0;              points[1] = 5;
        }else     if(choice1.equalsIgnoreCase("steal")              &&
choice2.equalsIgnoreCase("split")){             points[0] =
5;              points[1] = 0;
        }
        return points;
    }
```

This method implements the basic prisoner's dilemma in java by taking the two choices made by the two players and comparing them. The correct amount of points is then given to the players based on the theory behind the dilemma.

Looking at the code, an integer array of size 2 is created and will be returned. As mentioned in the previous section 3.2, this array will contain the specific points of each player. The first If else statement is for when the player's choices match each other, so split split / steal steal. The points are then set for the players, with position 0 being for player 1 and position 1 in the int array for player 2. The second If else statement is for when the player's choices are opposite to each other. Points are rewarded specifically for the player's choice as a split steal scenario only gives points to the stealer, as known from the basic prisoner's dilemma

Chapter 4: **My Program**

In this section I will discuss how I created my program on the Iterated Prisoner's Dilemma using software engineering. Each part of this section will be a breakdown of the main functions in my program and discuss what techniques I used to write the code.

4.1 Software Engineering

4.1.1 Description

Here I will discuss the software engineering techniques that I used for my program.

4.1.2 Development Process

I decided to use an agile methodology to develop this program. I broke down the goals I had for the functionality of the software and I worked on each section in sprints. Using an agile development methodology meant that I was able to find bugs early on when developing a new segment. This meant that in the long run I would not run into any conflicting parts of code and allowed me to scale the program as I originally intended. Agile also allowed me to adapt to different problems that arose and meant that if some of my goals were not deliverable then the final product would not be effected by that.

The sprints that I carried out were: Creating a player agent, Talley points with the dilemma, Round Robin Tournament, Replay ability, GUI. Within each sprint I had different user stories that broke down the goals of each sprint. Then, I created a priority of tasks list within each story to keep on track with the main aspects that I wanted to implement for each story.

4.1.3 Testing

I used TDD when developing my program to improve the quality of my code and find bugs early in the development process. My tests focused on the core elements; Player Agent, Strategies, Scoring System. These elements were crucial to the implementation of my code. The player agent had to be functioning properly as they are the ones using the program. The strategies are also important in the Iterated Prisoner's Dilemma to provide uniqueness and logic to each tournament. Finally, the scoring system was the logic of the Prisoner's Dilemma which returned points to each player in a match depending on their choices. Without ensuring each of these core concepts worked correctly the rest of the code would not be able to function.

4.1.4 Risk Mitigation

When I began my project, I made sure to look forward down the line at potential risks. Keeping these risks in mind during development allowed me to be prepared for them and to have a plan in place to navigate the problems. These plans proved useful when I ran into issues of overthinking certain tasks complexities, the plan I had in place for this risk allowed me to navigate it correctly rather than wasting time had I not planned ahead.

4.1.5 Design

The main design technique that I used was object-oriented programming. This technique was vital to my program as behaviours in each class impacted how the program functioned. My player object was the location where information about each agent participating in the tournament was held. This object would get populated by the behaviours of different classes throughout the program. Objectoriented programming also allowed for reusability of code which was useful when it came to implementing the infinite replay ability of the Round Robin Tournament.

Next, I used design patterns to decouple my code and reduce complexity. I made use of MVC design pattern in my program to separate the front and backend of my program. Using this design pattern made testing easier as I was able to test each component separately. Furthermore, design patterns improved the maintainability of my code as it was easier to edit code without affecting unnecessary areas.

4.1.6 Branching

Branching allowed me to work on specific areas of code which meant that I could isolate any problems that were occurring without them leaking over into the main version of my program. Branching also allowed me to experiment with different ideas and allowed me to carry out research on tests that I carried out on the Iterated Prisoner's Dilemma as a whole. In this branch I carried out more stressing tests with a large amount of players in a tournament along with a high number of rounds which I never intended to be part of the main program itself as it would be hard to run on lower end systems. Branching also proved useful in the first term when presenting as I was able to go back to early versions of my program and display my program's progression to the examiner and other students.

4.2 Player Agents

4.2.1 Description

I'm going to discuss how Player Agents interact with the system.

4.2.2 Player Object

To store information about each agent I used a class called Player which acted as an object to hold all the information about each player such as name, strategy, total points, choices. This object allows players to interact with each other as the information stored is called in other classes that control the logic of the software.

4.2.3 Player Strategies

Each agent was able to choose from a list of strategies when beginning the tournament. Different strategies work better in different scenarios depending on the number of people participating in the tournament and also the number of iterations that take place in the individual match ups. When choosing a strategy agents must keep these factors in mind. Agents can see a help page with an explanation on what each strategy does. In the long run Agents must factor in how the opponents are acting and they are able to alter their strategy if wanted before the next tournament starts.

4.2.4 Head-to-head

Agents in the system participate in a Round Robin Tournament format. In each match up the players use the chosen strategy and play multiple iterations of the Prisoner's Dilemma against each other. The number of iterations gets randomised at the beginning of each tournament because if players knew this exact numbers then it would lead to a Nash equilibrium of everyone defecting.

```
  public static void run(Player p1, Player p2, int numDilemmaRounds){
for (int i = 0; i<numDilemmaRounds; i++){
Strategies.Choice(p1.getStrategy(),p1,p2);
Strategies.Choice(p2.getStrategy(), p2, p1);
        }
```

```
        p1.setChoicesHistory(p1.getChoices());
p2.setChoicesHistory(p2.getChoices());
Iterator.tallyRoundPoints(p1, p2);
    }
```

The above code defines how a head to head of a player works. The players from a created matchup are passed into a method along with the randomised number of rounds to be played. Inside Strategies, the players choices are populated for the round. After this, the player's choices for the entire round are added to a history which can then be used in later tournaments to allow other agents to understand which players are cooperating or defecting more than others. The player's points for the match are then tallied and stored in the player object.

4.2.5 Learning

In my program different metrics are used so that players can see how well they are performing in different tournament settings. Total points are the overall metric to determine the tournament winner. This allows players to see how well certain strategies perform depending on strategies chosen by opponents. As mentioned previously, players will notice that stealing pays off in small sized tournaments as it punishes co-operators however, they will also notice that in a large pool this is not the outcome. I also implemented a trust score mechanism that gives players a tell on how trust worthy opponents have been so far in the previously played tournaments. Finally, I added a gossip mechanism that allows players to talk to each other in between tournaments. This allows players to learn characteristics of how the different players act, maybe a certain player is claiming to change to a specific strategy but never does.

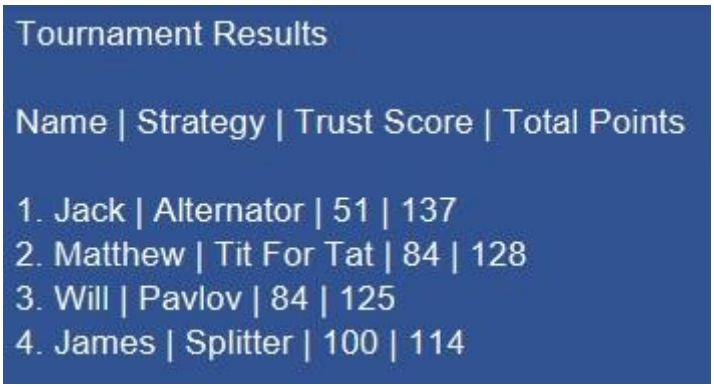
4.2.6 Style

Each player has their own though processing when approaching a tournament. Some players may prefer to be consistent in their choice of strategy and stick to the same strategy every tournament. Others may be more reactive to the outcome of a tournament and try to change strategy if they are unhappy with their outcome. This uniqueness makes the Iterated Prisoner's Dilemma so complex and interesting as there are no rules on how a player should act and everyone seems to devise different though processes on how they want to act and what outcome satisfies them.

4.2.7 Results

At the end of a tournament results are displayed at the click of a button. The results are shown in descending order of points. Each player can see these results to see what strategies other players chose. In my code I used a comparator [2] to compare points and sort the players in correct order.

```
 public static Comparator<Player> Comparator = new Comparator<Player>(){
public int compare(Player p1, Player p2){             int time1 =
p1.getJailTime();             int time2 = p2.getJailTime();
return time2-time1;
        }
    };
```



4.3 Strategies

4.3.1 Description

In this section I'm going to discuss the logic of my strategies and what each strategy does. [10]

4.3.2 Finding the strategy

During the second term I spent a large amount of time redesigning the logic behind finding which strategy was chosen by a player. In the first term I only had 4 strategies as the method I used was way too complex and not scalable. To improve this, I added a method inside the strategy class called Choice:

```
public static void Choice(String choice, Player p, Player opponent){
if (choice.replaceAll("\\s+","").equalsIgnoreCase("Stealer")){
            Stealer(p,opponent);
        }
```

The method takes in a string of the choice and both players. Each strategy takes in the opponent parameter even if they are not needed. This made it more consistent to add complex strategies that did use the opponent's choices to decide an outcome. The if statement was used to format the string so that strategies with multiple words like 'Tit For Tat' were consistently formatted to match other strategies length and spacing. This further decoupled my code compared to the first term. Without implementing this decoupling I would not have been able to scale my amount of strategies. The lines of code in term 1 for this same logic for 4 strategies was the same length as it is now for 12 strategies.

4.3.3 Populating Player's choices

Populating a player's choices was another change I made during the second term compared to first term. I wanted my program to be scalable to a large amount of rounds and large amount of replay ability. Furthermore, I wanted the number of rounds player to be random rather than fixed. To integrate this I set a players choice one at a time rather than all at once.

```
 public static void Stealer(Player p, Player opponent){
        p.setChoice("steal");
    }
```

Each time the strategy is called, the result of the outcome is set inside the Player object setter setChoice(String choice) which will add the string to an array list of type string called choices.

4.3.4 Stealer and Splitter

The Stealer strategy will constantly repeat the choice steal every iterated round against the opponent, regardless of what the opponent does. Although the strategy is very basic it is effective in a tournament against a small pool of opponents. The strategy will punish opponents splitting as a steal vs split choice results in 5 points to the stealer and 0 to the splitter.

The Splitter strategy is just the opposite of the Stealer. Splitter will constantly split regardless of the opponents' choices. Again, this strategy is very basic but is also effective in its own ways. This strategy is effective in a tournament against many opponents. This is because it encourages other opponents to also split.

4.3.5 Random

The Random strategy will randomly choose between splitting and stealing each round. The strategy is basic however can punish strategies that promote splitting by randomly throwing in a steal to gain points on the opponent.

4.3.6 Alternator

The Alternator strategy will randomly choose its first move of splitting or stealing, after that it will alternate between splitting and stealing each round. This strategy also tends to fare well in a small pool of players but in a larger group will score poorly.

4.3.7 Grudger

The Grudger strategy works by splitting every move until the opponent decides to steal. Once the opponent steals the Grudger will begin to steal every round then on until the end effectively punishing the opponent for their previous steal.

4.3.8 Tit For Tat

The Tit For Tat strategy is the most well renowned strategy in the Iterated Prisoner's Dilemma. In my program I implemented four different versions of tit for tat: Tit For Tat, Suspicious Tit For Tat, Forgiving Tit For Tat, Two Tits For Tat.

Regular Tit For Tat works by splitting on the first turn and then copy the opponent's previous move after that. This strategy is very adaptive to the opponent's strategy as it will make the most of opponents willing to split, and at the same time punish opponents who are attempting to steal to gain an advantage.

Suspicious Tit For Tat works by stealing on the first turn and then copy the opponent's previous move after that. Suspicious Tit For Tat will perform better in a tournament where opponents are stealing the majority of the time as the strategy can't be punished on the first move.

Forgiving Tit For Tat works by splitting on the first turn. The strategy will continue to copy the previous move of the opponent however, if the opponent steals it will continue to split for a few moves to try and convince the opponent to go back to splitting. If the opponent does not switch back to splitting it will begin stealing.

Two Tits For Tat works by splitting on the first turn and then copy the opponent's previous move after that. However, at one instance of stealing by the opponent the strategy will steal for the next two turns regardless if what the opponent does. This strategy attempts to punish strategies trying to sneak in a random steal to gain points.

4.3.9 Pavlov

The Pavlov Strategy is another more complex but popular and proven strategy in the Iterated Prisoner's Dilemma. In my program I implemented two versions of the Pavlov: Pavlov and Score Based Pavlov.

Regular Pavlov works by starting by splitting and then repeating the previous choice if it resulted in an equal payoff. If the payoff was not equal, it will switch to the opposite move. Pavlov is a consistent strategy in all forms of tournament sizes.

Score Based Pavlov works by randomly splitting or stealing on the first move. After that it will check the result of the previous iteration, if the points gained was more or equal than the opponent's points then the move will be repeated, otherwise it will switch to the opposite move if the payoff was worse.

4.3.10 Hard Majority

The Hard Majority Strategy works by stealing on the first move and the looking at the opponents history to decide the next move. The strategy will choose split or steal depending on which action makes up the majority of the opponent's move history.

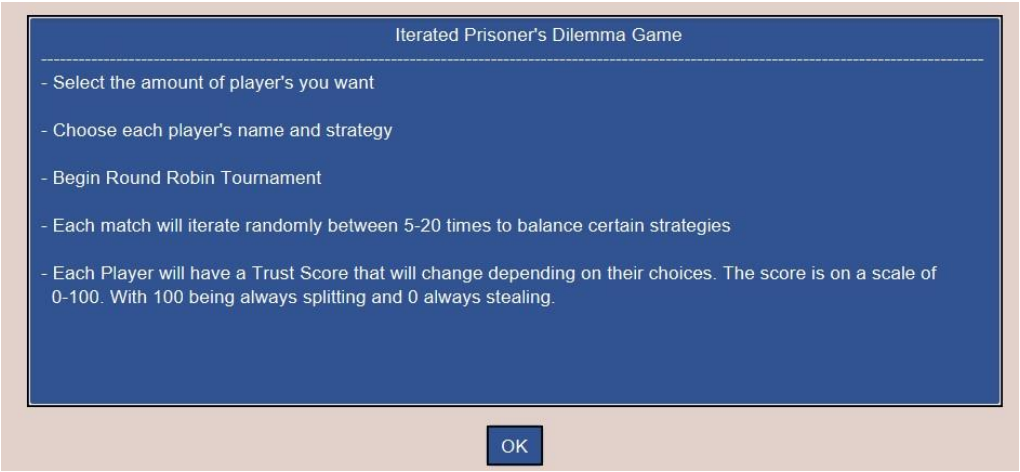
4.4 Graphical User Interface

4.4.1 Description

Here I discuss the interface of my program and the ways JavaFX helped to create the interface.

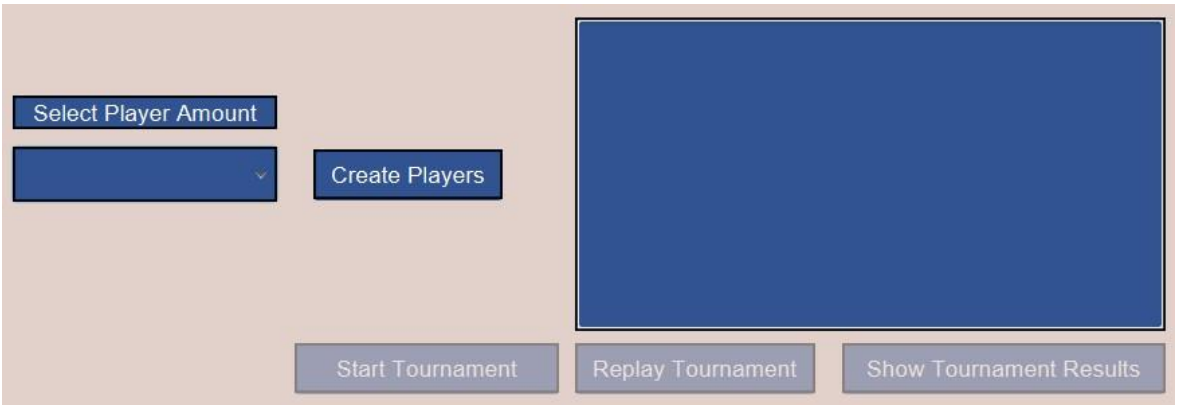
4.4.2 Design and Layout

When creating my GUI, I wanted it to be easy to follow and be consistent throughout each pop up. I chose a colour palette to make the interactive parts of my GUI stand out. The colour palette was also important to make the text easily readable. The text font and size were also kept consistent throughout the GUI.

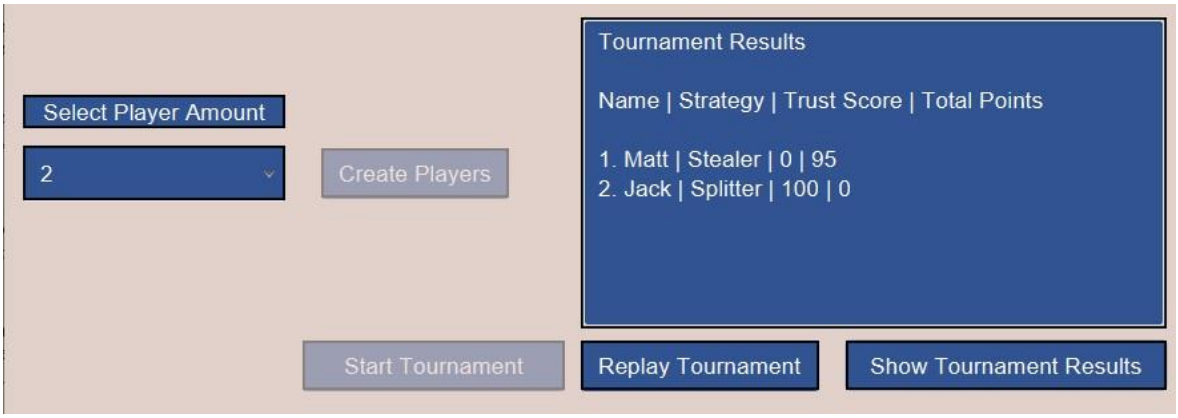


I used borders to help visually separate different GUI components. This allows users to distinguish between different parts of the interface. Consistent bordering allows users to identify which parts of the interface can be interacted with and which will display information.

The main focus of any GUI is how the user interacts with it. I wanted to keep the placement of buttons as consistent as possible and have them in natural positions on the interface, that way users would intuitively interact with the system. Furthermore, the placement of buttons is important to ensure that users interact with the system as intended. I did put some stop gap measures in place to force users to click on buttons in the intended ordering:



I set buttons to disabled until the user has interacted with the system as intended. Before starting a tournament the creator has to select the amount of players, and those players must enter their name and strategy. Once this is complete the tournament can be started. The show tournament results is made available after the tournament has finished and the replay tournament is available after the tournament results have been viewed:



Overall, with the design and layout I believe I achieved my goal. I wanted to keep the design simple and easy to use for anyone who wanted to run tests on the Iterated Prisoner's Dilemma so the minimalistic design that I went for achieved this.

4.4.3 User Interaction

Users interact with my GUI through text areas, buttons, and choice boxes. The text areas are used for entering their name or for entering a message to send into the Gossip Box. Buttons are labelled with what action they do for easy user understanding. The choice boxes will store a list of decisions that the tournament creator or players can make such as; how many players to create, which strategy the player wants to choose.

User feedback is also important to give the users understand on how well they have performed in the tournament. The tournament results will display the players in descending order of points and will show their name, strategy, trust score, and total points. This information can then be used to the player's benefits when it comes to playing again and updating their strategy based on other peoples trust score and total points.

Finally, I used help pages and error pop ups to guide users on how to correctly interact with the system and to explain what each strategy does. When the game begins the welcome page explains the steps to take to create a tournament with selected players to produce and outcome. When players are entering their name there is a strategy help button which will open up a separate window to explain what each strategy does in simple detail. This strategy help button is available to players whenever they are selecting or updating their strategy. To reduce errors, if a player tried to confirm themselves without writing in their name or selecting a strategy then an alert is generated to tell them to enter whatever is missing.

4.4.4 Cascading Style Sheets

Cascading Style Sheets (CSS) is a simple design language intended to simplify the process of designing interfaces. When creating my interface I wanted to make use of CSS as it is an efficient way of designing an interface. JavaFX allows you to use CSS which was important.

CSS allows you to style individual components of the interface as you like. For example, when designing my choice box with CSS I wanted it to be in the same style as the buttons but I wanted the list within it to visually tell the user what they are selecting.

When a user is hovering over the item that they want to select from the list it highlights in red. This allows the user to easily distinguish which item they are about to select. Also, once the item has been selected if they re open the list there will be a check mark next to the current selection for further easy recollection on their choice.

```
.choice-box .context-menu { -fx-background-color: #325391;}

.choice-box .menu-item:focused { -fx-background-color: #325391;}

.choice-box .menu-item > .label { -fx-text-fill: white; -fx-border-color:
#325391}

.choice-box .menu-item:focused > .label { -fx-text-fill: red;}
```

CSS made this function easy as you can see in the above code, you can specifically target sections within the choice box to change its looks [13].

CSS saved me lots of time when designing my interface as you are able to load the same style sheet in the code so that the new pop-up design is kept consistent. CSS makes it so much easier to edit the style of an interface and at the same time does not interfere with the underlying code behind the interface.

4.4.5 Technical Implementation

To create my interface I used JavaFX and Scene Builder. JavaFX is a package in Java that can create interfaces of applications. Scene Builder is a tool that allows you to design the layout and types of components you want to use for the interface. To use JavaFX I had to add it to the dependencies of my pom file which is made easy using visual studio code and maven.

In order to create functionality for the components of my interface I had to give them an fx:id and an On Action name. The fx:id allowed me to control the contents of the component and the On Action allowed me to provide functionality to the buttons.

I made use of the Initializable controller in JavaFX to populate choice boxes. Initializable allows the contents of the choice box to be there as soon as the display is shown. This controller was also useful for disabling buttons on the display to prevent user error. It allowed me to guide users through their first tournament creation to avoid mistake with interaction.

A prominent factor to my program functioning was being able to pass user input to other interfaces.

```
GameView gameView = loader.getController();
gameView.displayName(Matches,playerList,selected);
```

This allowed my program to be continuous and was very beneficial later down the line when I implemented replay ability. It was also useful for when the creation of players was finished. It allowed me to pass information from the pop ups back to the main view page of the interface which could then be used to run the tournament.

4.5 Iterations on outcome

4.5.1 Description

The importance of iterations in the Iterated Prisoner's Dilemma is fundamental as it can completely change the outcome of the results. I will break this down using tests that I carried using my program's strategies.

4.5.2 Relationship between Iterations and Tournament Results

The relationship between the number of iterations and the results is an important factor that I took into consideration when creating the game. In my first term I had a set amount of iterations taking place between players. The problem I found with this was that if players realised the amount of rounds taking place it would incentivise them to steal as the benefits of stealing are greater than splitting in the short term. This in turn mean the outcome of the tournament usually resulted in a player choosing the Stealer strategy coming out on top. Furthermore, there was only one tournament taking place at a time and scores did not tally over time like they do now in my program which further incentivised players to steal.

I came up with an idea to randomise the number of rounds that take place during each tournament. The randomised number was generated at the start of each tournament to keep the number of rounds consistent for each match up. The more iterations meant that players were more inclined to split as they realised a strategy like Tit For Tat would provide a more beneficial outcome in the long run. This change caused a shift in the outcome of the tournament results. I saw that strategies that focused on splitting over stealing performed much better than before.

4.5.3 Payoff Equilibrium

In the original Prisoner's Dilemma studies have found that the Nash equilibrium is always to defect. I also found that in a low number of iterations that the Nash equilibrium was also to steal (defect). In my game the points of a steal vs steal is 1 point each, whereas split vs split is 3 points each. So, the lower the number of iterations, the lower the average payoff for each player was.

When I began to increase the number of iterations this equilibrium began to change. I found that the average payoff increased as there was more incentive to split with each other and players developed

a pattern of splitting which led to a high average payoff. This is again correlated to the previous point of the relationship of iteration and tournament results.

4.5.4 Strategy Choices

The relationship of iterations and tournament outcomes also depends on the strategy choices from the players. For example, if a tournament of 6 players is created and 5 of the 6 players choose the stealer strategy then the payoffs in the matches will be significantly lower and in turn the iterations did not actually increase the stability of the points outcome. Although this scenario is an extreme it still has to be taken into consideration as players can be unpredictable especially when their aim is to win.

This scenario became more realistic when the tournament had the capability to be played again. Allowing players to see what strategies others are choosing meant that there was a higher chance of players changing their strategy to punish players using 'nice' strategies that focused on splitting.

4.6 Gossip

4.6.1 Description

Now I'm going to discuss how gossip influences play.

4.6.2 Communication

Gossip in my program works by allowing players to communicate with each other after a tournament ends. Communication between players can create dynamic scenarios and adds another layer of strategy that a player can adopt. In my code I use JavaFx functions to create a chat box where players can gossip. The use of a VBox inside a scroll pane simulates a messaging system like text messaging.

The messages then get stored in a list to be passed into the next pop up window, this way previous messages can be shown to the next player.

```
public void createName(ArrayList<Player> playerList, int i, List<String>
messageList) {
```

```
        if (messageList != null && !messageList.isEmpty()){
this.messageList.addAll(messageList);
        }else{
            this.messageList = new ArrayList<>();
        }
```

To create this gossip box with the correct formatting I took inspiration from reference [9].

4.6.3 Reputation

The fundamental behind gossip is to create a reputation that differs per player. Players can claim that they are going to change their strategy to manipulate others into attempting to counter them, and instead can counter them back. If a player is trustworthy then they will be able to convince other players into splitting to punish other players who have been stealing consistently.

The trustworthy players can use this to their advantage in other ways as well. For example, they can build an alliance with certain players to manipulate them, and then in the future they can use this to steal against those same players to gain an advantage.

4.6.4 Interactivity

In my game the gossip system is in the form of one-way communication. This means that players can send many messages at one time, but once they confirm their new strategy they can no longer gossip to other players. I chose this form of communication as it reduces the risk of manipulation from other players. Furthermore, players can just focus on their own strategy and plan how they are going to act in future rounds.

Messages in the box are labelled by the player's name so that everyone can see who has said what. This also helps the user to remember who is trustworthy and who is not.

Users interact with the gossip box using a text field where they can enter the message they want to send to other players. The Send button will then format and add the message to the VBox which gets displayed on screen. I kept the interaction with the gossip box simple to understand and mimicked a classic text messaging format for user recognition.

4.6.5 Manipulation

Manipulation is a massive contribution factor as to why I added gossip to my game. Manipulation can be an effective strategy in itself even without the chosen strategy for the actual tournament. Players can intentionally spread false information to try and deceive their opponents in gossip. For example, a player could spread a rumour that 'Jack' is going to change his strategy to Stealer to exploit the rest of the players. This could in turn influence everyone into Splitting and possibly even make Jack split, then the rumour spreader could exploit the splitter strategy with Suspicious Tit For Tat.

In the short term manipulation can be effective, but can negatively impact a player in the long run. This returns to the concept of reputation, manipulation will damage a players reputation if others catch on to their games. If manipulation is done smartly and not too frequently it can deceive other players without them even catching on.

4.6.6 Cognitive Bias

Cognitive bias refers to a thought process caused by the tendency of the human brain to simplify information processing through a filter of personal experience and preferences [11]. This can cloud players judgment of the messages being sent by other players. Cognitive bias causes players to interpret information received through the gossip chat differently. This in turn affects decision making and the outcome of their score in the tournament.

The main form of cognitive bias is when someone tends to read the information is ways that confirms their beliefs about a player. Player Matt could be someone who is sarcastic but player Jack could be someone who tends to take peoples word for granted. Matt could say he is going to change to a Splitter, Jack could believe him and also change to Splitter, but in the end Matt actually chose Stealer. Jack's cognitive bias caused him to make a bad judgment and affected his tournament score.

Cognitive bias also plays into long term strategy for the tournament. You could be a player who has decided to base their strategy on the discussions in the gossip box. This is called the anchoring effect which is when cognitive bias causes people to rely too heavily on information given [12]. This effect also determines a player's outcome as if they had just stuck to a plan from the start rather than false gossip information then they could have performed better in the tournament.

4.7 Trust Score

4.7.1 Description

Trust score is a component that I added to create another layer of strategy behind player's decisions in the tournament. Allowing players to see tendencies of other players is a key component behind strategizing in tournaments.

4.7.2 Calculation of a Player's Trust Score

The Trust Score was an idea that I thought could also contribute to a players decision making and create a psychological component that can be used as information on other players history which can then be brought up in the gossip box between players.

The calculation for getting a player's trust score was simple, it was the ratio of total splits vs total splits & steals. The higher the trust score, the more a player has been cooperating and vice versa.

```
 public static void TrustScore(Player p) {
int totalSplits = 0;         int
totalSteals = 0;         int trustScore =
0;
        for(int i = 0; i < p.getChoicesHistory().size(); i++){
String choice = p.getChoicesHistory().get(i);
if(choice.equals("split")){                 totalSplits++;
            }else{
                totalSteals++;
            }         }         trustScore  = (int) (100 * ((double)
totalSplits / (totalSplits + totalSteals)));

        p.setTrustScore(trustScore);

    }
```

A players Trust Score is updated after every tournament and is set inside the Player class. I wanted the Trust Score to be calculated on a players entire history of every tournament that has taken place to far. This was players could get a better understanding in the long run on what each player has been acting like.

4.7.3 The use of Trust Score

Trust score can influence a player's decision making when it comes to choosing a strategy. Players can use Trust Score as either a form of manipulation or to show that they are willing co-operators. For example, in the first tournament a player could choose to be a splitter to get a high trust score to manipulate people into splitting in the following tournament, and then punish them by switching up completely to a stealer strategy to gain points.

Trust Score does actually give players incentive to split in the long run. For example, if a tournament consists of lots of players and they see that your trust score is low, then they might create an alliance to punish the player with a low trust score by choosing strategies like Tit For Tat or Pavlov that will split with each other but steal against players that are stealing often.

4.7.4 Trust Score and Game Theory

Trust score relates into game theory concepts such as the Nash equilibrium. Trust score will help players trust each other more which can lead to a Nash equilibrium of players splitting with each other more often.

Trust score is also a measure of how successful different strategy patterns can. Finding a balance of choosing between a strategy that focuses on splitting and a strategy that focuses on stealing can lead to a higher outcome in the long run.

4.8 Replay-ability

4.8.1 Description

The ability to replay the tournament using the same players creates more depth to a players strategy outside of just the chosen strategy. I allowed players to have the option to change strategy after each tournament so they can adapt based on results.

4.8.2 The Need for Replay-ability

Replay-ability is once again another layer of strategy to take into account for the players. It also allows players to react and adapt to feedback given in the previous tournament such as trust score, total points. In my program once a tournament is replayed players can update their strategies. They can also chat to each other in a gossip box which can then be used again and again each replay.

The more replays, the more of a profile each player can build up. It allows other players to gain greater knowledge on the behaviours of other players which can then be used to strategize accordingly. Replay-ability also allows for research into more complex strategies that can consist of more than just Tit for Tat. It could be possible to find a combination of different strategies that if ordered and used correctly, could outscore Tit For Tat making the research into replay-ability endless.

4.8.3 How I implemented Replay-ability

In my code I implemented replay-ability by using a method to pass in updated player information created from the previous tournament. The controller for each replay was the same with just the updated information passed into it. Furthermore, I used an updated pop up that contained a gossip box and allowed players to change strategy.

I also added a button that shows the overall scores of all the tournaments combined so far. This also displays the past strategies chosen by players and their trust score for all the tournaments combined as well. Displaying previous strategies is part of the need for replay ability as players can take this into consideration when updating their strategy for the next tournament.

```
   public void createReplay(int selected, ArrayList<Player> playersList,
ArrayList<String>  playersNames){                replay.setDisable(true);
display.setDisable(true);          overall.setDisable(true);

        this.selected = selected;
this.playerList.addAll(playersList);
this.Players.addAll(playersNames);
    }
```

The above method is how I pass information into the newly created tournament. The new information is passed in and the variables inside the controller are set to the values being passed into the method. This allows for an infinite number of replays to be carried out by the tournament creator.

4.8.4 Benefits for Researchers

Replay-ability opens the door to lots of different research that can take place. Researchers might conduct an experiment about how Strategies can evolve depending on changes to incentives, like points. For example, maybe the punishment for steal vs steal is now 0 points to both players, this should lead to even more people splitting and make stealing strategies very ineffective.

Furthermore, it allows researchers to observe how effective strategies really are in the long run. The ability to replay the tournament over and over proved useful in my research to find the most consistent strategy. I was able to prove that Tit For Tat is the best in the long run, even though a Stealer strategy might out perform it in the odd tournament in the end it will all balance out due to replay ability.

Finally, researchers can further understand the impact of human decision making has on the outcome of tournaments. The longer a group of players play against each other, the more likely cognitive biases is going to impact outcome. As players tendencies start to become known then other players are likely to adapt which provides some interesting research.

Chapter 5: **Analysis**

The most interesting part of the Iterated Prisoner's Dilemma is the infinite amount of research that can take place. In this chapter I will cover the research that I was able to analyse throughout the development of my program. I will also judge how I implemented my interface to make this analysis possible.

5.1 How Number of Agents Effects Outcome

5.1.1 Description

The number of Agents in the system is a catalyst for the shifts in results of the tournament. I will investigate this concept in more detail and give examples of some results that I found.

5.1.2 Theoretical analysis

In the original Prisoner's Dilemma, it took place between two players. This made the outcome easier to analyse as there were only 4 possible outcomes. When the game is played between two or more players the outcome becomes more complex as there are multiple different choices and payoffs to analyse. This is because there are more and more opportunities for players to split or steal, with a large amount of different combinations of choices.

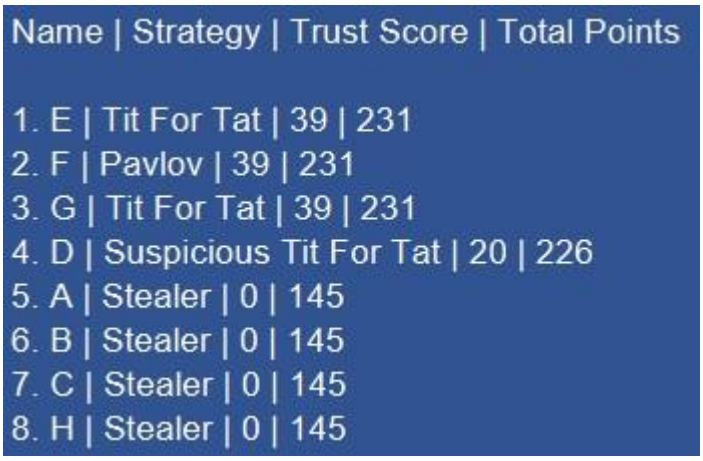
To research into this more I decided to carry out some tests based on some questions I posed about the game, How is splitting or stealing is influence by the number of agents? How do strategies success rates differ depending on the number of agents?

5.1.3 Splitting or Stealing

To understand the question, you must look at what the players are trying to achieve. Each player wants to finish the tournament with the most points possible. We can look at the question from two pools of players, a small group and a large group. In a small group of players, players will be more likely to steal as they are not bothered about their reputation. In a small group individuals can just be selfish and think about themselves because there is not many people to out strategize them.

For example, in a group of 2 the Stealer Strategy will always come out with more points or the same points as the opponent worst case scenario. This incentivises players to always steal in a group of 2.

As the number of Agents increases, Stealing becomes less and less popular. This is because other players know that if they split with each other but steal against the Stealers, then they will outscore the stealers. This gives people less incentive to steal as they run the risk of being punished by other players who have allied with each other to split.

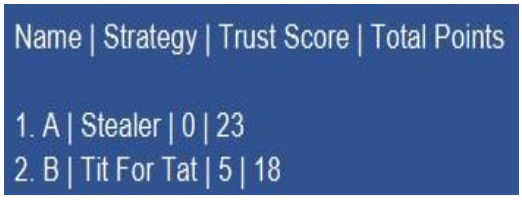


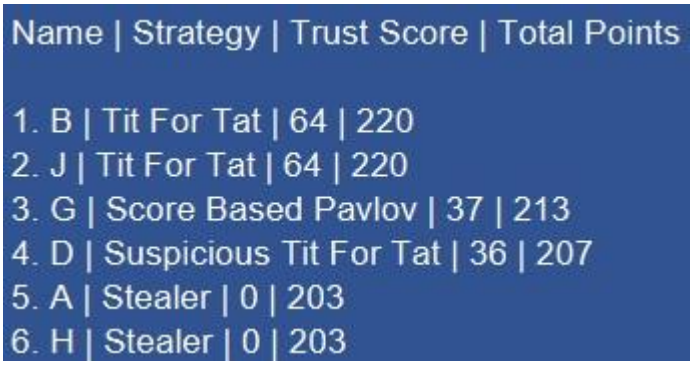
As you can see above, as the number of players increase the less effective stealing becomes. Players are now going to be more likely to split than steal in a large group of participants which shows how splitting or stealing is influenced by the number of Agents.

5.1.4 Success Rate of Strategies

Certain strategies may be successful in a small group of players and others may be more successful in a larger group of players. Strategies like Stealer and Suspicious Tit For Tat are very effective in a small group of 2/4 players but become less effective in a larger group.

This links back to what was mentioned previously where in a large group players tend to split more often or choose strategies that will adapt to other players, like Tit For Tat. Tit For Tat however is not successful in small groups like 2/4 even though research has shown that this is the 'best' strategy in the Iterated Prisoner's Dilemma.





With this in mind, players will tend to shift to more adaptive strategies when the number of agents increases and shift to more punishing strategies as the number of agents decreases. This is how a strategies success rate differs depending on the number of agents in a tournament.

5.2 Strategies in Different Scenarios

5.2.1 Description

Strategizing in the Iterated Prisoner's Dilemma becomes very complex when different scenarios are created. I am going to explore a few different scenarios and find the best and worst strategies in each: When the game is played for a fixed number of rounds, When the game is played infinitely, When there is gossip included in decision making.

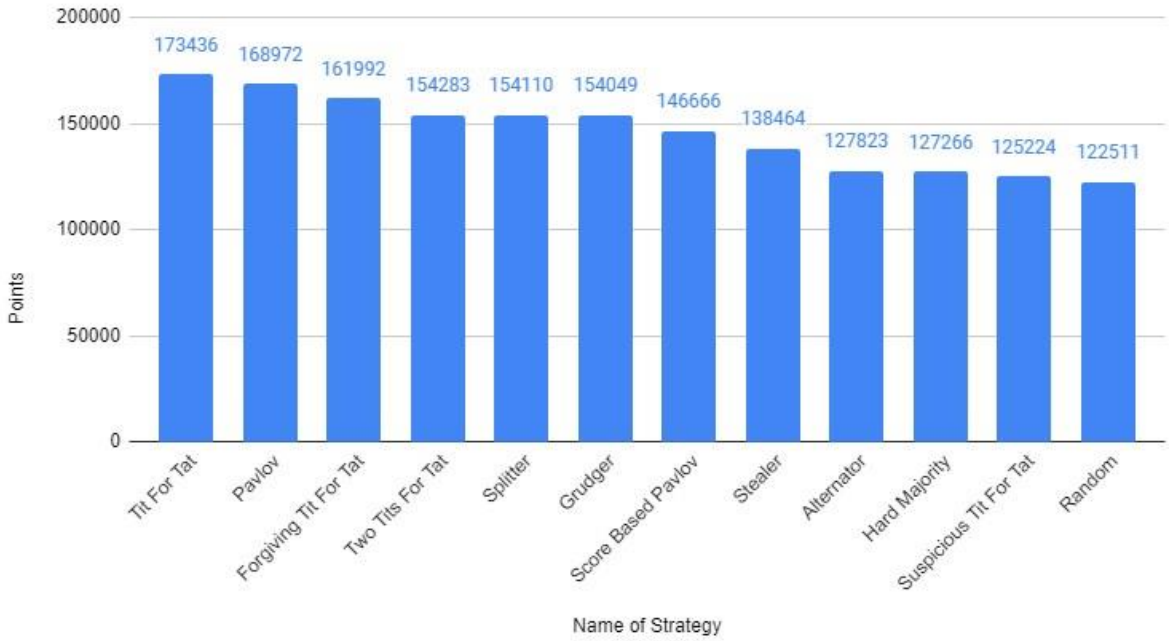
5.2.2 Fixed number of Rounds

Early on in my program I used as fixed number of rounds for the number of iterations that players would play against each other. What I found was that when players know the number of rounds taking place, they can use specific strategies to gain the most points and best result. Aggressive strategies like Stealer performed very well in a low number of fixed rounds as they always outperformed or equalled the points as the opponent. Whereas when the number of rounds got randomised later in my program, I noticed that Stealer began to perform much worse than the other strategies that I had implemented.

Proven strategies like Tit For Tat and Pavlov struggled with a fixed number of rounds as they could be punished by more simple strategies like Random, Stealer and even Suspicious Tit For Tat. Stealing is very effective against these strategies in a fixed number of rounds as they are not able to make use of their copying opponents feature to punish the opponent.

5.2.3 Infinite repetitions

A feature that I planned to implement from the start was my replay tournament ability. I knew that this feature would be useful to determine the best strategies in the Iterated Prisoner's Dilemma. The use of infinite repetitions is the most common way researchers have tried to find the best strategies. What I found was that the same strategies always seemed to emerge at the top after a long period of repetitions of the tournament. I ran the tournament with the same players with a randomised number of rounds 30 times:



In the early stages the results were scattered in terms of which strategy was at the top, but in the long run this balanced out and every time Tit for Tat, and on rare occasions Pavlov, was always at the top of the leader board. As you can see in the chart, strategies that had previously performed well in a low number of rounds are now at the bottom of the leader board in this scenario.

So we can see that in this scenario more complex strategies performed better than the simple strategies. It seems that complex strategies and complex scenarios go hand in hand with each other in the Iterated Prisoner's Dilemma.

5.2.4 Gossip's effect on Strategies Success

Gossip is a complex concept that adds a layer of strategy on top of the already available strategies. Judging Gossip's effect on success is very difficult as what tends to happen is gossip causes players to change strategy rather than stick with the initially chosen strategy.

For that reason we must try and look at gossip's effect on strategy success differently. Instead of considering just the available strategies as a strategy we must look at a player's style of play as a form of strategy. Gossip plays with the emotion of players to influence their style of play. Gossip tends to give success to deceitful players who use gossip to trick others into changing strategies that can then be punished.

5.3 Interface

5.3.1 Description

I'm now going to analyse the features of my interface and look at what goals I followed to create my interface.

5.3.2 My Goals

When I started the project I knew that condensing a complex concept like this into an interface could prove difficult. I wanted the interface to have an easy learning curve so that someone who is completely unfamiliar with this topic would be able to use my interface without issues. I also wanted to provide complex functionality to those who want to run tests on the game that require deeper knowledge of the topic. Finally, I wanted to keep the layout and colour scheme consistent throughout so that the interface is visually simple to understand.

5.3.3 The Features

To make the learning curve of my interface easy I used specific features in JavaFX.

The two main features to achieve this was Pop Ups and the ability to disable buttons until they are meant to be used. I added help pop ups to explain how to run the tournament and another to explain what each strategy does in simple detail. Furthermore, I added error prevention pop ups to stop people from trying to create a player that was missing a name or strategy. Disabling buttons was aa key way to guide new users through the program as I intended it to be used. Preventing users from clicking on buttons that were not meant to be used until another part had been populated first was key to making sure that my program ran the same for everyone, regardless of their previous knowledge.

To provide more complex research I added the ability to replay the tournament for as many times as you like. This was made possible thanks to the controller classes that I implemented. The interface for the replay function is adapted to make it easy to use and the tournament results part of the interface is also adapted to show the results of every tournament run so far.

Finally, the colour scheme of my interface is clear and consistent throughout every pop up. I wanted the background of my interface to be a simple and unintrusive colour and then I wanted to draw users attention to the main features like the buttons and text outputs. To draw attention I used consistent thickness of bordering and coloured the buttons and outputs with a darker blue to make it stand out.

Chapter 6: **Professional Issues**

I am now going to discuss the professional issues that I ran into when developing my program and how they resulted in time wastage which could have been spent on implementing more complexity to my program.

The first main problem occurred in term 1 when I was developing the logic behind different strategies pitted off against each other. I ended up using lots of different unnecessary if statements to determine how each strategy played against each other.

```
        String p1Strategy = p1.getStrategy();
        String p2Strategy = p2.getStrategy();

if(p1Strategy.equals("Stealer")){
            Strategies.alwaysSteal(p1);
        }else if(p1Strategy.equals("Splitter")){
            Strategies.alwaysSplit(p1);
        }else if(p1Strategy.equals("Random")){
            Strategies.Random(p1);
        }
if(p2Strategy.equals("Stealer")){
            Strategies.alwaysSteal(p2);
        }else if(p2Strategy.equals("Splitter")){
            Strategies.alwaysSplit(p2);
        }else if(p2Strategy.equals("Random")){
            Strategies.Random(p2);
        }

                        if(p1Strategy.equals("Tit     For     Tat")      &&
!p1Strategy.equals(p2Strategy)){
            Strategies.titForTat(p1, p2);
        }else if(p2Strategy.equals("Tit For Tat") &&
!p2Strategy.equals(p1Strategy)){
            Strategies.titForTat(p2, p1);
        }else if(p1Strategy.equals("Tit For Tat") &&
p1Strategy.equals(p2Strategy)){             Strategies.alwaysSplit(p1);
            Strategies.alwaysSplit(p2);
        }
```

As you can see there were many lines of code and this was only for 4 strategies. If I wanted to implement another 8 strategies on top of this it was going to get messy. The problem I realised that I was having was that each strategy took in either 1 player or 2 players into the method. This caused a problem of inconsistency. I realised that if I just made very strategy take in 2 parameters, even if 1 was not needed, then I could delete all of this messy code and just create some simple if statement in my Strategies class to check the name of the strategy and then run the strategy accordingly. If I had tackled this problem in term 1 rather than leaving over to term 2 then I could have saved significant time that was wasted redesigning this entire logic. The new logic behind this is much simpler and is scalable to implement an exhaustive amount of strategies if I was to go back and improve this program in the future.

Seeing as I wanted to make my program scalable not only in terms of strategies but also in the sense of the number of iterations, I had to revamp how a players choices were populated. In the first term I had a set number of iterations that players would play against each other. I wanted to change this due to me becoming aware of research where if players knew the number of rounds being played, then they would always defect on the last round regardless of what the opponent was going to do as it was beneficial to their overall score [14]. I figured out that if I set a players choice 1 at a time and

controlled the iterations elsewhere then I would be able to scale the program infinitely. This feature was also something that I should have taken into consideration more in first term and had I gone down this path from the start then again, more time would have been saved for second term to implement additional features.

The next problem that I had was when I had implemented the two above features I began implementing many different strategies. This in turn caused some hidden problems that were a headache to fix. When running my TDD tests I found some strange problems where if two strategies were run in one order and then swapped and run again, the results would be different. This was not what I wanted to see happen as I knew that this could have devastating effects on the research and testing that I wanted to carry out to determine which strategies are the best in my game. I spent a long time figuring out how to fix this problem and in the end I had to re code some of my strategies completely. This problem really occurred to me when I implement different versions of the strategies I had already implemented. The problem I was having with Tit For Tat was that when the strategy looked at the previous move of the opponent, it was actually looking at the move just played rather than the move at the previous position. For example, if P1 choices were [Split,Steal] and P2 using Tit For Tat was [Split], the strategy was making its next move Steal rather than Split. It was a small logic error which caused lots of time to be wasted figuring out a way around it. Sometimes the small errors like this take the most time to fix as they can be very hard to locate.

A small problem that I encountered was when I began creating the CSS for my GUI. The issue I had was changing the design of my choice boxes. The choice boxes started to use the formatting of my labels which caused a problem where the contents of the choice boxes had borders around them and looked very messy. I managed to find a useful post to format specific parts of the choicebox: [13].

The final problem that I had was more of a culmination of all the previous ones. With the time wasted on the previous problems it meant that I was not able to implement some more complex features to my interface and program as I whole. I would have liked to have a feature where the tournament creator could have typed in the amount of players that they wanted to use rather than having a set number of players incrementing in even numbers only. This would have allowed for any researcher to use my program to carry out more complex tests on the Iterated Prisoner's Dilemma rather than being limited to a smaller range of tests that are possible. I could have made this possible by spending the wasted time to adapt my creation of matches in the Round Robin Algorithm to include a 'by' system where players get a free range of points when an odd number of players is entered by the tournament creator. This would have added another dynamic layer to my program.

Looking back I ran into some issues when creating my program which could have been handled much more efficiently had I laid more solid foundations early on in the development of my program. Some issues caused more damage than others but ultimately all contributed to inefficient time usage which could cost me some marks that were easily gainable had I planned features more thoroughly. However, these issues that I ran into have given me greater understanding on how to deal with problems in future project and will help me be more efficient with my programming work.

Chapter 7: **Conclusion**

To conclude, I am very pleased with the outcome of my project and I believe I implemented the main core concepts behind the Iterated Prisoner's Dilemma. Considering the infinite scope of this topic I am happy with the features that I implemented in the period of time. I also was able to learn from the mistakes that I made with my initial planning that I will be able to use for future projects. I developed a greater independence of working on a large project on my own which is a useful skill to take into

the industry for future employment. I also gain a greater knowledge on Java which is a programming language I already enjoyed previous to this project.

Although my project has not come up with some ground-breaking research for the Iterated Prisoner's Dilemma, I do believe that it has captured the concepts interesting aspects and can be used to educate people with little knowledge on the topic. Just like I mentioned previously, the scope of this subject is infinite and the scalability of a program like this could be worked on for years and would still only just scratch the surface of this amazing concept.

Chapter 8: **User Manual**

How to use the program:

Follow the instruction of the main page carefully, create the players, run the tournament, show the results and then choose if you want to play again or not. Strategy help is available each time a player is created/updated.

Video: https://youtu.be/A3MdkUYG8aU

Chapter 9: **Project Diary**

Project Diary

10/10/22:

Begun the project. My aim for the next 2 weeks is to get the basic dilemma implemented and

have a command line to enter choices. - created an initial player object with tdd

11/10/22:

- added more setters getters to the player object

13/10/22:

Today I implemented the basic prisoner's dilemma by comparing users input. There

are some small problems that I will fix like being able to enter defect/cooperate in

the command line in any format

18/10/22:

I now have a basic terminal interface to prompt the players to enter their

name and choice of defect/cooperate which then gets stored in the

player object

31/10/22:

Had family over the last week so had little time to work on the project but I did manage to get a simple GUI working with javaFX and scenebuilder

01/11/22:

Up to this point I am a little behind on the milestones but over the next week I will make an initial GUI, create the iterated dilemma and possibly add some strategies

09/11/22:

I have now make the dilemma iterated for 5 rounds and added some strategies that players can select from a drop down menu. I will now work on getting the strategies to work for head to head game and update the GUI to incorporate it all over the next week

15/11/22:

Head to Head is now working and I also updated the gui so that players can be created using a drop down menu to select the amount of players the user wants 16/11/22:

I am now on schedule for the milestones and over the last 2 weeks I will make updates to the GUI to implement the tournament feature and finish the javadoc

23/11/22:

The program is finished for this term and I will spend the remainding time writing my report and checking over the program to find any hidden errors

26/11/22:

Fixed remaining small errors, code is finished. Overall I am happy with the progress I made this term and I completed all the milestones that I set for myself. I would have liked to implement more strategies than just the select 4 that I chose but I decided that it would be more beneficial to get the core concepts implemented as adding new strategies should be straight forward.

30/01/23

Started to implement more strategies, starting with the grudger strategy. I need to make it more efficient to play strategies against each other as I ran into some problems today trying to implement another strategy that relies on other people actions rather than a constant splitter/stealer that always does the same thing each round

07/02/23

Implemented an alternator strategy that randomly chooses its first choice and then alternates between splitting and stealing after that. This was quite simple to implement. My next steps will be to redesign how the head to head works depending on what strategies are chosen to make it easier to implement more complex strategies.

20/02/23

Spent last few weeks planning the redesign of the head to head. Started implementing it by updating the strategy names, and adding the opponent parameter to every strategy even if they are not needed. This will make it easier when adding new strategies that take the opponent into consideration. I also added a new method in Strategies.java which will determine the players strategy choice. This will mean that the massive chunk of code in Game.java can be deleted making the overall program cleaner and more efficient to add an exhaustive amount of strategies.

27/02/23

Fully implemented my updated way to face players against each other. Now the points are tallied 1 at a time. The arraylist gets updated 1 round at a time which makes it easier to face a tit for tat vs stealer as tft will split first round regardless, and wont need the splitters code to run before it anymore. I'm happy with how I have implemented the code that I spend a long time planning out. My next aims for the project are to add loads of strategies, make the number of rounds played random, possibly add a different tournament style like swiss system, and to make charts for determining the best strategies.

04/03/23

Today took longer than expected as there was some really small problem with the strategy that was well hidden. I eventually found it and everything works how its inteded to work.

13/03/2023

Smooth day for implementing two more strategies, Pavlov and Hard majority. The changes that I made are proving to make it way easier to implement more complicated strategies with less issues.

14/03/2023

Implemented more strategies. Next im going to work on a chat box where players can gossip at the end of a tournament. Also going to add a trust score so players know if other players have been stealing more than splitting.

15/03/2023

Added a replay tournament ability. Two tournament can be played in a row between the same players. They can then change their strategy when the new tournament

starts. I had some issues with carrying scores over but mangaged to fix it. Tomorrow im going to make the replayability inifinite so the tournament creator can make new tournaments over and over as many times as they like.

16/03/2023

Tournaments can now be played for an infinite amount of times in a row now. This will be useful to gather data on which strategies are the best over a long period.

I also began making a chat box where players will be able discuss the previous tournament. I need to find a way to pass the text to other players so that they can chat to each other.

17/03/2023

Fully implemented a chat box ability. Credit to this amazing youtube video I found https://www.youtube.com/watch?v=_1nqY-DKP9A&t=2718s&ab_channel=WittCode

refrence [9] in my report. Was useful to understand how to format the GUI chat box neatly and mimics a text messaging service like on your phone. I also

implmented a trust score that just checks the amount someone splits compared to steals. The higher the score the more the person splits and vise versa.

I should be finished with the code tomorow and I only need to make some charts to compile data to prove which strategies are the best. Finally I will look over all my code and make it neat and improve my GUI a little.

18/03/2023

My project is now finished from a GUI and coding stand point. I just need to go over everything and add any missing javadoc and it is complete. Today I completely revamped the GUI with colour and css.

Chapter 10: **Bibliography**

[1]https://stackoverflow.com/questions/26471421/round-robin-algorithm-implementation-java

[2]https://beginnersbook.com/2013/12/java-arraylist-of-object-sort-example-comparable-andcomparator/

[3] https://www.sciencedirect.com/topics/social-sciences/prisoners-dilemma

[4] https://plato.stanford.edu/entries/prisoner-dilemma/strategy-table.html

[5] Robert Axelrod and William D. Hamilton. The evolution of cooperation. Science, 211:1390-1396, 1981.

[6] Nowak, Martin A. "Five rules for the evolution of cooperation." science 314.5805 (2006): 1560-1563.

[7] M. Nowak, R. May and K. Sigmund. The Arithmetics of Mutual Help, Scientific American, June 1995.

[8]https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvcarchitecture-and-frameworks-explained/

[9] https://www.youtube.com/watch?v=_1nqY-DKP9A&t=2718s&ab_channel=WittCode

[10] https://plato.stanford.edu/entries/prisoner-dilemma/strategy-table.html

[11] https://www.techtarget.com/searchenterpriseai/definition/cognitive-bias

[12] https://thedecisionlab.com/biases/anchoring-bias

[13] https://stackoverflow.com/questions/43530178/style-choicebox-list-with-css-injavafx

[14] https://ncase.me/trust/notes/#:~:text=%22You%20won't%20know%20in,so%20they'd%20both%20cheat.