Cooperative Strategies in Multi-agent Systems Final Report

Lauren Moore
Supervisor: Argyrios Deligkas
Full Unit Project 2020/2021
Word Count: 13,736

Abstract

The Prisoner's Dilemma is a game in which two players have the option to either cooperate with or betray each other, with differing scores earned depending on the choices made by each player. It has been studied since its invention in the 1950s and is still incredibly relevant to the fields of not only game theory but business, psychology and even politics. I have implemented two types of Iterative Prisoner's Dilemma tournament (in which players play multiple times and attempt to accumulate the highest score) - the traditional round robin and an evolutionary tournament. During this process I have performed research in order to fully understand the background theory and applications of this type of program. The literature I have read relevant to the topic, the professional issues that arise from use of programs like mine and the experiments I have performed are also detailed in this report. Finally, I have evaluated the success of my implementation when compared to the aims I set out at the beginning of the project.

Contents

1 Project Aims and Motivation

The overall aim of my project was to produce a program that runs two types of tournament of the Iterative Prisoner's Dilemma - round robin and evolutionary. In order to achieve this there were multiple smaller aims, such as creating a GUI to allow the user to configure and run the tournaments, and outputting results in as meaningful a way as possible. These aims were supported by the early deliverables set out in the project brief, which were all completed by the end of 2020:

- Identification of a set of strategies - this has allowed me to create a program with a good number of relevant strategies.
- Reports on the Iterative Prisoner's Dilemma and Tournaments - this has cemented my knowledge and ensured my implementation of the IPD and tournaments is correct.
- Report on Previous work with the IPD - research for this has provided options for extensions to the project. After discussion of my research with my advisor I decided that running evolutionary tournaments was the avenue I wanted to investigate with my project work in the second term.
-
- A program that allows the user to simulate tournaments of the Iterative Prisoner's Dilemma - this deliverable had been produced by the end of 2020 and I extended my project by adding functionality that simulated evolutionary tournaments as well.

The final deliverables set out in the project brief are:

- A GUI that allows a user to create and manage tournaments - this deliverable had also been produced by the end of 2020 but was extended by implementing a GUI for both creating and outputting results of evolutionary style tournaments.
- 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 - this deliverable had also been produced by the end of 2020 but was extended to include outputting the round robin results to a CSV file and also by including the evolutionary tournament, which by definition shows how strategies perform compared to each other as changing colours in the graph.

The aims I set as extensions to the original brief were:

- Add functionality to allow the user to create and run an evolutionary tournament - including both back end support and a GUI which allows the user to intuitively assign strategies to each node in a graph.
- Allow the user to select different types and sizes of graphs to run their evolutionary tournaments on, the final list of which was grid, circle, star, complete, path and bipartite.
- Allow the user as much control of the tournament variables as possible including whether or not to randomise their values where appropriate.

- Allow the user to export the details/results of their tournament to allow for further analysis and saving of results.

The work involved in my project has the potential to be massively beneficial to my future career. Firstly, it has given me the opportunity to learn and improve a wide array of skills, from researching a topic to managing a large scale project to implementing a GUI using JavaFX. These more generalised skills will be useful in a software development work environment, particularly the ability to set and achieve goals while working on a project. Additionally, the topic of the Prisoner's Dilemma and simulation has such a wide variety of applications that the knowledge is likely to be incredibly useful, particularly in the context of performing data analysis or modelling for a business.

2 Project Theory

2.1 Introduction to the Prisoner's Dilemma

The Prisoner's Dilemma is a game in which two participants play against each other in an attempt to either minimise or maximise their own score. While the analogy used to explain the game states both prisoners are attempting to minimise their score (prison sentence), most tournaments (including my own) has strategies attempting to maximise their score. Each participant or agent is given the option to either cooperate or defect, with scores determined by comparing the two participants' decisions. While the specific amount of points for each combination of decisions varies depending on the implementation, the participants always get an equal but small score when both cooperate, an equal but smaller score if both defect and uneven scores if one defects and one cooperates. The defecting agent will receive the largest amount of points possible while the cooperator will receive the smallest possible amount of points, traditionally 0. The Iterative Prisoner's Dilemma is a variation on this where the agents play the game multiple times in a row. A strategy is a set of logic or rules for how an agent will act normally depending on how their opponent has acted in previous rounds. The normal Prisoner's Dilemma only allows for very basic strategies as it is only played once (either cooperate, defect or random), while the iterative version allows for the deployment of much more complicated strategies as the agents can base their decisions on their opponents previous actions. When this iterative version is played with more than two strategies it is referred to as a tournament.

The method of evaluating strategies is generally how effectively they manage to minimise or maximise their score (depending on the version of the game being run) over multiple rounds of the game during a tournament. However it is necessary to run multiple tournaments to obtain an accurate analysis as factors such as the ordering and what other strategies are in the tournament will affect how effective a strategy appears to be. Study of the Prisoner's Dilemma and comparison of strategies is incredibly important as it has an incredibly large number of real world applications depending on the strategies entered into a tournament and how they are analysed. For example it can be used to attempt to maximise profits for a singular business (identifying a strategy that performs optimally against many others), or to maximise profits for two competing businesses (identifying two strategies that maximise payoff for both parties when played against each other). It also has important implementations with large numbers of players, such as the tragedy of the commons, which describes how when faced with a finite resource it is in players' best interest overall to all use as little of the resource as possible to avoid depleting it, which can be applied to concepts such as how power and resource consumption affect climate change.

2.2 Strategies

A strategy is the method a player or agent uses to determine which decision to make in each round. These can be as simple as always cooperating or as complicated as analysing all an opponents previous moves to attempt to work out which strategy it is following and react accordingly. There are two main types of strategy - basic and adaptive.

Basic strategies are the simplest type of strategies, as they do not consider any history or perform any analysis in order to make their decision. There is therefore an incredibly limited amount of these strategies, which are listed below.

- alwaysCooperate - A strategy that cooperates regardless of the game history and any other factors.
- alwaysDefect - A strategy that defects regardless of the game history and any other factors.
- alwaysRandom - A strategy that has a 50% chance of making each decision on each turn.
- varyingRandom - A strategy that has a random chance, assigned by the runner of the tournament, of cooperating.

Adaptive strategies are more sophisticated as they can perform analysis on the match's history. This allows for many more possibilities for strategies as they are able to react to their opponent's moves. The vast majority of strategies are adaptive.

- titForTat - A strategy that cooperates as its first choice, then copies its opponent's previous move. [17]
- Spiteful - A strategy that always cooperates until its opponent defects, in which case it defects on every subsequent turn. [3]
- softMajority - A strategy that begins by cooperating and then continues to cooperate as long as the opponent has cooperated a number of times equal to or greater than its number of defections. [3]
- hardMajority - A strategy that begins by defecting and then continues to defect as long as the opponent has defected a number of times equal to or greater than its number of cooperations. [3]
- varyingMajority - A strategy that begins by cooperating then if the opponent has cooperated more times than it has defected it will cooperate. The strategy only considers a set number of previous moves, which is assigned by the runner of the tournament at the beginning.
- periodicDDC - A strategy that plays defect, defect, cooperate on a continuous cycle.
- periodicCCD - A strategy that plays cooperate, cooperate, defect on a continuous cycle.
- periodicCD - A strategy that plays cooperate, defect on a continuous cycle.
- mistrust - A strategy that is similar to tit for tat (and is sometimes called suspicious tit for tat because of this). It begins by defecting and then copies its opponent's previous move.
- pavlov - A strategy that begins by cooperating, then cooperates only if both players made the same move in the previous round. [20]
- titForTwoTats - A strategy that cooperates on the first two moves, then defects if its opponent defected on both of the two previous moves.
- hardTitForTat - A strategy that cooperates on the first two moves, then defects if its opponent defected on one or more of the two previous moves.

- gradual - A strategy cooperates until its opponent defects. Whenever its opponent defects the strategy will defect the total number of times its opponent has defected, then cooperate twice in a row. [4]
- prober - A strategy that plays the moves defect, cooperate, cooperate then always defects if the opposing strategy cooperated in moves 2 and 3. If this condition is not met it plays the same as tit for tat. [12]
- mem2 - A strategy that makes the same decisions as tit for tat in the first two moves. The strategy then chooses a new strategy to follow every two moves based on the outcome of the last two moves:
  - if the result was both cooperate each time, follow tit for tat
  - if the result was cooperate defect or defect cooperate each time follow titForTwoTats
  - if the result was any other then play always defect
  - if always defect has been chosen twice then it will always choose always defect

  [11]
- titForTatWithForgiveness - A strategy similar to tit for tat in that it begins by cooperating then copies its opponent's previous move in every subsequent round. However, it has a random chance (normally quite low) of cooperating even if its opponent defected on the previous round.
- scoreBased - A strategy that makes its decision based on the difference between its score and its opponent's score. If the strategy's score is higher by a certain amount it will cooperate, otherwise it will defect.

List of strategies based on - [13], with some strategies added from my other research and some suggested by my advisor.

2.3 Theory behind strategies

There are many different strategies and while they are mostly just classified as either basic or adaptive, there are other ways of analysing and grouping strategies. One of these methods is grouping them by shared properties, such as niceness and whether or not they are forgiving. [1] These properties provide a possible extension to my project as I could use my program to run tournaments to investigate how they affect a strategy's performance. In addition, I could also investigate how the properties of its competitors' affect a strategy's performance.

A strategy is considered to be nice if does not defect before its opponent does. This is very important as many adaptive strategies look to punish opponents that defect, so defecting first could cause a cycle of retaliatory defecting.

A strategy is considered to be forgiving if it does not continue defecting long after its opponent has began or returned to cooperating. This is important as not entering into mutual cooperation is likely to cause the opponent to begin defecting again. If a strategy's opponent is defecting then its maximum possible score is severely limited, as its options are cooperate and get the lowest

possible round score or defect and get the second lowest possible round score (if the traditional payoffs are being used).

A strategy is known as retaliating if it begins defecting for some period of time (not necessarily forever) once its opponent defects. This is important as strategies without this feature are likely to be exploited by some strategies as they provide no punishment for defecting against them.

A strategy is non-envious if it is not trying to score more than its opponent. This can be beneficial as a strategy attempting to defect in order to gain more points than its opponent could cause patterns of mutual defection, minimising the scores of both itself and its opponent.

2.4 Iterative Prisoner's Dilemma

The Iterative Prisoner's Dilemma is a variation on the Prisoner's Dilemma in which multiple rounds of the game are played with the same two players. This allows for far more sophisticated and interesting strategies as they can access and make decisions on their opponent's previous moves. The consequences of this are that players are able to make predictions about what their opponent will decide in the next round and, arguably more significantly, it encourages players to choose strategies that cooperate often as otherwise their opponent is able and likely to retaliate.

Which strategies perform the best in the Iterative Prisoner's Dilemma depends on multiple factors, including some outside of the control of the strategy itself. A good strategy must obviously be able to read and analyse the decisions made by its opponent in the past, however its performance will also be affected by which strategies it is playing against as well as the order in which it plays against them. This is explored in multiple articles on the subject, including Axelrod's article discussing his famous first tournament. In this article he mentions how strategies that are inclined to be cooperative do well when playing in a tournament against each other as they often establish mutual cooperation.

2.5 Introduction to tournaments

Tournaments are incredibly significant when discussing and researching the Iterative Prisoner's Dilemma as they are the main method for evaluating the performance of algorithms/strategies. They work by allowing multiple people to enter one or more strategies, which will play a number of rounds of the Prisoner's Dilemma against other participants. The type of tournament dictates whether or not they play every other strategy or just some, and many also include a rule that strategies play against a copy of themselves.

2.6 Types of tournament

Round Robin. A round robin style tournament works by having each player/strategy play a game of the Iterative Prisoner's Dilemma against every other player. Depending on the rules of the tournament they may also play against a copy of themselves. It is also possible to change the payoffs for cooperation and defection, the number of rounds and whether or not the number of rounds is finite. All of these changes have been observed (and in the case of finite rounds proven mathematically [7]) to impact the results of a tournament. While this type of tournament is still studied today, many tournaments around the beginning of the Prisoner's Dilemma's popularity were round robin tournaments. This includes the most famous one in the field - Axelrod's 1980 tournament. [1]

Evolutionary tournaments. Evolutionary tournaments are a type of tournament where the score of a strategy in a particular generation determines how many instances of the strategy appear in the subsequent generation. These can take place in a similar fashion to the round robin or on a graph such as a grid in order to better represent actual populations of organisms. In this case the strategy plays in a round robin tournament with all of its neighbours (the nodes in the graph that are adjacent to it) with the strategy with the largest score taking over the space of its less successful neighbours. This leads to strategies that are more successful being represented by more nodes in the graph.

The results of these kind of tournaments are particularly significant for researchers who wish to apply their results to real life behaviour of both humans and animals. They are mainly concerned with attempting to explain how cooperative behaviours persist in species despite more selfish behaviours having better short term results for an individual. Multiple theories have emerged from or been researched by these tournaments, including direct and indirect cooperation and group selection. Direct cooperation is the theory that repeated cooperation with someone you've met before is likely if they have previously cooperated and indirect cooperation is the same concept but regarding someone who you've only heard cooperates. Group selection, originally proposed by Darwin, states that natural selection can be applied to a group as a whole as well as just individuals. The application of this in the Prisoner's Dilemma is that a group of nice strategies all interacting with each other is likely to thrive and be better represented in subsequent rounds. ([15])

Evolutionary tournaments played on a grid, referred to as a Spatial Prisoner's Dilemma, are still the subject of ongoing research. As well as being interested in how cooperation evolved researchers are investigating how it can be encouraged, particularly among players who are naturally inclined to defect. There are multiple mechanisms for this still being investigated, including rewarding strategies for cooperating [8] and a reputation system [19]. The reputation system seems to be a good way of mirroring the real world, as we have multiple systems that perform the same function, such as credit scores. There are also more unusual and modern examples of this system of reputation driving cooperation such as how social media influencers use the size of their following to acquire collaborations. The more followers (reputation in this case) the person has the more likely another influencer is to be willing to engage in mutual promotion (cooperation) with them.

Knockout tournaments. Knockout tournaments work by playing pairs of strategies against each other. The strategy that gains the least points in the round is then eliminated, with this pattern continuing until only one strategy remains. This type of tournament is not commonly used as it has multiple flaws, such as the seeding (which strategies play against which) affecting which strategy wins. In addition, it is not fair to use this type of tournament to declare a second place due to the fact that any strategy played by the winning strategy could theoretically be the second best, even if it was knocked out early in the tournament. There are, however, ways of improving the traditional knockout tournament in order to gain more useful results such as holding a double-elimination tournament instead. In this tournament an entrant can lose a single round without being eliminated, which reduces the risk of an effective strategy being knocked out due to bad luck (e.g. unlucky seeding or unlucky random number generation in a match).

Another decision made by the designer of a tournament that will affect the outcome is whether or not the game lasts a set number of rounds. A large number of strategies are designed to

cooperate where possible in order to minimise or avoid retaliation from their opponent. However if the players know when the last round is there is no incentive to cooperate on that round as there is no chance of retaliation. The problem that this causes is that there is also no reason to cooperate on the second to last round (or any round before that), as the opponent is nearly guaranteed to defect on the next round regardless. In order to combat this most modern tournaments are of indeterminate length rather than being fixed, meaning that at the end of every round there is a possibility that the game will end. [5]

2.7 Winning strategies and notable tournaments

The most notable tournament in the field of the Prisoner's Dilemma is the first tournament run by Robert Axelrod in 1980. It consisted of 14 entered strategies as well as one that randomly chose between cooperating and defecting, which played in a round robin style tournament (every strategy played against every other strategy and a copy of itself). The winning strategy in this tournament, tit for tat, is still discussed today and many different versions and possible improvements have been suggested since its invention. For example, the a strategy commonly referred to as tit for tat with forgiveness follows the basic principles of tit for tat, but also has a small chance of cooperating on a turn where it is meant to defect. The benefit of this is that it can (depending in the strategy it is playing against) allow the strategy to re-establish cooperation with its opponent if they are stuck in a cycle of both defecting. While it was not the first appearance of tit for tat, Axelrod's first tournament is responsible for its popularisation within the field. The impact of this is that many people have attempted to improve on this, with strategies such as tit for tat with forgiveness and slow tit for tat. ([1])

Axelrod ran more than one tournament, with his second introducing the idea of the tournament having a random probability of ending rather than running for a set number of rounds. Tit for tat won this tournament again, further cementing its place as a very significant strategy in the field. The whole results table also impacted the field as it partially disproved some of the conclusions drawn from the first tournament. Axelrod originally theorised that strategies which were nice and forgiving did better overall. However in the second tournament some of these strategies, in Axelrod's words, 'fell victim to' strategies which were designed to take advantage of their cooperative natures. Additionally, in his article regarding this tournament Axelrod began considering the evolutionary type of tournament instead of the round robins run in his first two. He studied cooperation through the lens of evolutionary biology in order to explain why it persists in nature, becoming so interested in this that he released an entire book on the subject, entitled the evolution of cooperation. ([2])

In 2004 and 2005 the IEEE Congress on Evolutionary Computing ran Prisoner's Dilemma tournaments to mark the 20th anniversary of Axelrod's publication. They had two significant differences to the original - the concept of noise (a small probability that a strategy's actual move would be different to their chosen move) was introduced and teams/researchers were invited to submit more than one strategy. The second change proved to be the most significant, as strategies from the University of Southampton won both tournaments by entering 'teams' of strategies that intended to recognise each other and work together. They did this by employing a pre-agreed series of moves in order to recognise each other, then either all cooperate or act preferentially towards one member of the team to maximise their individual score. ([18])

3 Literature Review

[13] - This paper has a similar aim to my project in that they are attempting to use tournaments in order to analyse the performance of prominent strategies in the area of research. In relation to my project aims it was particularly useful when gathering the list of strategies I wanted to implement. The list featured in this paper formed the basis of my list, along with suggestions from my advisor and some other strategies I discovered during my research.

[1] - This paper by Robert Axelrod describes his famous 1980 tournament and also draws some conclusions based on the data produced by it. Possibly the most significant impact it had on my work is the fact that it popularised the tit for tat strategy, which led to both that and multiple variants and attempted improvements being included in my list of strategies. I also found Axelrod's theories on properties of strategies affecting their performance incredibly interesting as a method of analysis and have therefore included it in my report, as well as considering it in the experiments I ran using my completed program. This research has clearly had a large impact on both my work and the field in general as most of the other papers cited in my report either directly reference or build upon the work done by Axelrod. However, given how long ago this research was performed I have needed to look at other more, recent sources to ensure the theory presented in my report is as accurate and up to date as possible.

[2] - This paper, again by Robert Axelrod, details his second attempt at running a tournament. Tit for tat was victorious again, proving that it is a strategy worth including and studying in my own work. It also expanded on his previous theories regarding properties of strategies but altered some conclusions, such as nice strategies seemingly gaining more points. This tournament showed that while nice strategies are likely to be successful when interacting with each other, they will become exploited if they are faced with many not nice strategies. As mentioned previously this research is now 40 years old and this means that it cannot form the entirety of my research. Many breakthroughs and changes have happened in the field since then, including the rise of the pavlov (or win-stay, lose-shift) strategy and the interest in evolutionary tournaments instead of round robins. In the experiments section of the report I have actually used my evolutionary tournament implementation to expand on Axelrod's conclusion in this report by investigating the validity of his conclusion on nice strategies in an evolutionary tournament.

[15] - This article shows one of the ways that research in the field developed after the initial interest in Axelrod's tournaments. It illustrates one of the ways in which an evolutionary tournament can be run, using a 2d grid and having each strategy play against its immediate neighbours. This ended up being the extension I chose for my project because the graph view of the results each round seemed interesting to simulate and possibly more meaningfully communicates the results of a game when compared to just a score. While I did implement a 2d grid with the same tournament rules as in this research I extended beyond it by offering multiple graph configuration options. The main impact this research had on my project was sparking my interest in and explaining the basics of the concept of evolutionary tournaments. Like a lot of the significant research in this field, this article was written a number of years ago and I had to perform more research to find newer research to supplement my knowledge.

[18] - This paper was authored by the team that won the Prisoner's Dilemma tournaments run by the IEEE Congress on Evolutionary Computing. It describes the unusual method of entering multiple strategies into a tournament which will function similar to a team by identifying each other. These strategies are split into what is known as master and slave strategies. Once they

recognise each other a master strategy will defect and a slave strategy will cooperate in order to maximise the score of the master strategy. The ability to recognise other strategies is discussed in other papers, particularly ones focused on evolutionary tournaments, but had not been achieved and utilised this effectively in round robin tournaments before. Both this recognition-based cooperation and the introduction of noise could be used as further extensions to my project. While they would both be possible extensions to the round robin style of tournament they may have more significance and allow me to draw more context-based conclusions if I chose to implement them in conjunction with an evolutionary tournament. This is because they both have real world counterparts in organisms (trait recognition and miscommunication/mistakes), which is the context that evolutionary tournaments are generally applied to. I did not implement these features during my project but they would make good extensions if I were to continue working on it.

[8] - This paper discusses the Spatial Prisoner's Dilemma and how strategies taking part can be encouraged to cooperate. It looks at a possible mechanism for this where strategies that cooperate are given a reward which is taken equally from the points of strategies which have chosen to defect. The research concludes that this method of rewarding cooperators and punishing defectors can be successfully used to promote cooperation. This is interesting for researchers in psychology and biology looking to investigate how society can promote cooperation for the greater good. I feel that extending my project to allow for this kind of system to be used may produce interesting results given that my program allows for different types of graph to be used. This would be beneficial as different types of graph more accurately represent different scenarios - conclusions from a grid may be relevant to interactions between a society of organisms but a complete graph may better model interactions between businesses.

[16] - During my research I came across a program named Oyun which was designed for teaching students about the Prisoner's Dilemma. While there are strong similarities in our programs, for example they both run round robin and evolutionary tournaments, the way in which these problems are approached is relatively different.

There are very few design decisions to be made when implementing a round robin tournament and the two final products are very similar because of this. The only significant difference is that the results of each game are output in Oyun as a table containing a small summary (players, scores earned and winner) and in my game this information is displayed in a matrix. I feel that both of these choices are clear and easy to read, but that my detailed results screen may be improved by adding summary information such as total score earned in a game and which strategy was the winner like in Oyun.

There are far more options when implementing an evolutionary tournament and I have made significantly different decisions to the creators of Oyun in this regard. Oyun allows the user to select which strategies will be entered into the tournament but only allows for equal amounts of each strategy to be used. It then runs the tournament in the back end, outputting the results as a plot of the amount of each strategy in each generation. My program, on the other hand, allows the user to run evolutionary tournaments on a graph through the GUI, choosing the size, players and placement of players each time. In my opinion both of these implementation choices are equally valid as both have benefits and drawbacks. The advantage of Oyun is that a large number of players can be used over a large number of generations and the program will still output clear, analysable results in the format of a graph. My program limits the number of players and generations based on what can be clearly presented through the GUI and because of this can handle significantly fewer of both. The main advantage of my approach is that it offers far

more customisation options, making it more suitable for running experiments regarding things like unequal representation of strategies and how their placement affects their performance. This shows that both of our programs are suited to the audience they were developed for, as Oyun is intended to teach the concepts of the Prisoner's Dilemma, whereas mine is intended for a more academic research based context.

4 Software Engineering and Program

4.1 Program

I have created a small demonstration of the main functionality of my program. It can be viewed at: https://youtu.be/OPNnuQap2Sw

My final program allows the user to run their choice of either round robin or evolutionary tournaments of the Prisoner's Dilemma. As one of my aims for this project was to allow the user as much control as possible the main/first screen is populated entirely with all the configuration options the user has. This includes the strategies that will play, the payoffs and the number of rounds per game and a button for each type of tournament. The user should press one of these tournament buttons once they are satisfied with the configuration values they have chosen. Additionally, there is a menu item at the top of the window labelled help, opening this creates a sub-item named user guide. When selected this will open a user guide pdf I have created, which is included in appendix D.

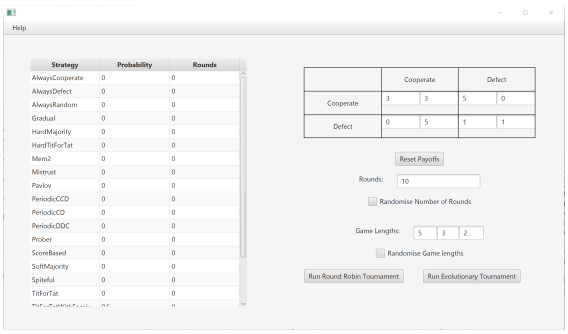


Figure 1: The first screen of the program as it appears upon launch (i.e. no values have been changed).

The front screen contains all the information necessary for a round robin tournament so selecting its corresponding button will simply run the tournament in the background and launch a screen containing its results. From this screen the user can launch a window containing more detailed results, export the results to a CSV file or return to the first screen.

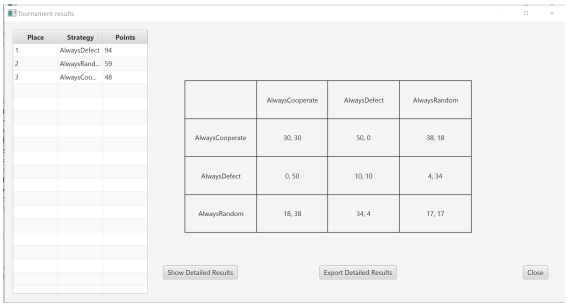


Figure 2: The screen of the program that shows the results of a round robin tournament.

If the user chooses to view the detailed results of the tournament a new window is launched

containing details of each round, namely the decisions made and points earned.

Figure 3: The screen of the program that shows the results of a round robin tournament.

Evolutionary tournaments require more set up than round robins so selecting that button will launch another configuration screen. This one will allow the user to select graph type and size, the program will generate the graph in the GUI then the user is able to select a strategy from the list and then select a node to assign that strategy to the node for the tournament. It is also possible to set the amount of generations the tournament will run for in this screen.

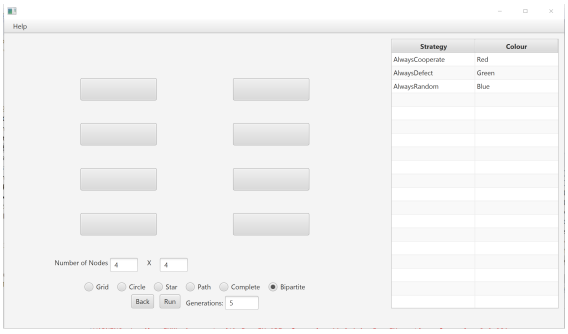


Figure 4: The screen of the program that allows the user to set up the evolutionary tournament.

The next step is to press the run button and the program will run the tournament in the back end and launch the results screen. This is very similar to the previous screen with the graph remaining as that is where the results are displayed and with the table of strategies/colours remaining to act as a key. The user can view the results of the tournament through increasing or decreasing the generation number with the next and previous buttons. It is also possible to enter a number into the generation number box to jump straight to that generation.

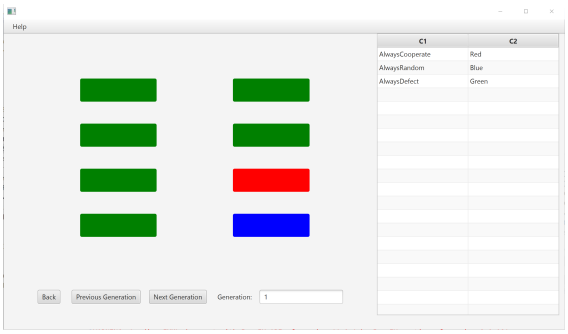


Figure 5: The screen of the program that allows the user to view the results of the evolutionary tournament.

4.2 Development

The logical first step during my development was to implement all the strategies listed in this report so they could be used to run games of the Iterative Prisoner's Dilemma. This program refers to a round as a single decision made by each strategy and a game as a set of rounds played between the same two strategies. The round robin plays three games with each pair of strategies. In terms of an evolutionary tournament, a generation is where each pair of neighbours have played a set of three games with the scores being totalled.

Initially I implemented the most basic three strategies mentioned in the report - alwaysCooperate, alwaysDefect and alwaysRandom. These were enough to allow me to decide the best way to implement these strategies, which I decided was having each strategy be its own class which was a subclass of an overall strategy superclass. This allowed me to ensure each strategy had a method that returned its decision in each round as well as associating the amount of points earned with the instance of a particular strategy.

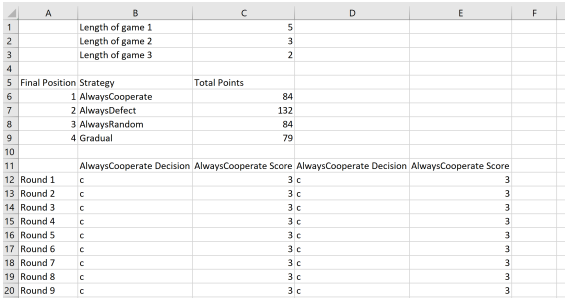
In order to implement the more complex adaptive strategies I needed to then add a data structure storing the history (each player's moves) of a game so the strategy could analyse it. This led to the development of the game class, which took a number of rounds and two strategies as parameters and played the number of rounds specified in the parameter. The history could then be stored as two variables in that class (one for each strategy) and accessed by the two strategies playing that game.

I was then able to build upon that by implementing functionality that runs a round robin tournament. A round robin tournament is formed of multiple games of the Iterative Prisoner's Dilemma so my program works by calling the playGame function in my game class multiple times. I began by creating a new tournament class named roundRobin which takes a list of all the participating strategies and the total number of rounds in each game. The runTournament function in this class calls the runGame function for each possible pair of strategies, including making strategies play against a dummy version of themselves.

It was necessary to add a function that ensures that the length of each game is not fixed. As discussed in the types of tournament section, this is important because fixed-length tournaments have an optimal strategy (alwaysDefect), whereas infinite or randomly ending tournaments do not. After discussion with my advisor I implemented a function that takes the total number of rounds and runs three games of random length. This is done by first generating a random value

x1 between 1 and the total number of rounds, which is the length of the first game. The length of the second game x2 is then generated, which is between 1 and the total number of rounds - x1. The length of the final game is the total number of rounds - x1 - x2. While the values of x1 and x2 are randomised, they are the same for every pairing in order to maintain fairness. In my proof of concept program the values of x1,x2 and x3 were always randomised by the program, however I have now extended the functionality of my final program to allow the user to specify these values if they wish.

Once all the logic for the running of the tournament was implemented, I began to implement a GUI. This allows the user to configure the tournament by selecting the participating strategies, payoff values and the number of rounds in each game. It also attempts to present the results in a readable, concise format (ordered results list and matrix showing points earned in each game) as well as the full, verbose results in a separate screen. After displaying the results through the GUI I began thinking about ways to export these results out of the program to facilitate further analysis. Eventually I decided on exporting all the data included in the verbose results screen into a CSV file so it could be accessed as a spreadsheet. In order to create this spreadsheet all the user needs to do is press the button marked 'Export Detailed Results' in the results screen, the program handles the generation, naming, and saving of the file in the background.



	A	B	C	D	E	F
1		Length of game 1	5			
2		Length of game 2	3			
3		Length of game 3	2			
4						
5	Final Position	Strategy	Total Points			
6	1	AlwaysCooperate	84			
7	2	AlwaysDefect	132			
8	3	AlwaysRandom	84			
9	4	Gradual	79			
10						
11		AlwaysCooperate Decision	AlwaysCooperate Score	AlwaysCooperate Decision	AlwaysCooperate Score	
12	Round 1	c	3	c	3	
13	Round 2	c	3	c	3	
14	Round 3	c	3	c	3	
15	Round 4	c	3	c	3	
16	Round 5	c	3	c	3	
17	Round 6	c	3	c	3	
18	Round 7	c	3	c	3	
19	Round 8	c	3	c	3	
20	Round 9	c	3	c	3	

Figure 6: The CSV file produced by the program opened as a spreadsheet.

The next section of the program to be built was the evolutionary tournament and the first step towards this was to implement a node class. The purpose of this was to allow me to bundle together an instance of a strategy as well as a list of their neighbouring nodes in order to pass this information to the class responsible for running the tournament. This seemed to be the more efficient implementation as the neighbouring nodes don't change, only the strategy they are playing changes so there is no need to identify the neighbours each time a generation is run.

I then began to create the actual evolutionary tournament class, which takes the set of all nodes in the graph and uses that to run the tournament. This is done by essentially running multiple small round robin tournaments (one for each generation), with the program iterating through the list of all nodes and each node playing a set of rounds with each of its neighbours. Once all games have been played the scores are normalised to maintain fairness in the tournament. This means that a node's score is divided by the number of neighbours it has in order to remove the advantage given to nodes with more neighbours, as the more neighbours you have the more games you get to play. Finally, the strategy associated with each node is updated if it has a higher performing neighbour and the results of the generation are saved so they can be presented using the GUI.

As part of the tournament, when a generation ends each node takes on the strategy of its most successful neighbour (the adjacent node with the highest score). I had some trouble working out an efficient solution to this as the score accumulated is stored in the instance of the strategy. This meant that I could not have multiple nodes associated with a single strategy object even if they were following the same strategy as their scores would be added together. The solution I came to was to use reflection to examine the strategy followed by the most successful node and create a new instance of it to be assigned to the node being taken over by that strategy.

After all necessary back end code was created I was able to begin thinking about designing and creating the front end. Initially I created everything but the graph itself, namely the list of strategies to choose from and a set of radio button which allow the user to select a type of graph. Implementing these components was simple as radio buttons have very simple functionality and I had learnt how to use the TableView to present lists of strategies during the creation of my proof of concept program in the first term.

Successfully generating and displaying different types of graph was a significant challenge as it involved learning new skills related to UI creation and developing small algorithms for the generation of each graph. Firstly I had to investigate how to change sections of the GUI based on choices made by the user (in this case which radio button is selected). This was done by adding a pane to the UI where the content should be then setting the contents of that pane through the code when the user makes a selection.

Next, I began trying to actually generate the graphs, which took multiple attempts to implement successfully. The implementation that seemed to work best for what I wanted was to generate the graph from buttons. The main advantage of this was that they have inbuilt functionality that allows for users to interact with them (by pressing them) and assign them a strategy. Additionally, their colour can then be changed in order to represent the strategy they have been assigned. Each button in the graph is associated with an instance of the node class which keeps track of which strategy it is assigned and which nodes/buttons are its neighbours.

Code tailored to each graph shape was required to generate the different types of graph in the UI. While generating a grid, a path and a bipartite graph were all relatively straightforward, I found it much more difficult to generate circular graphs. In particular it is difficult to generate round looking circles using a small number of nodes. The solution I came to was to generate slightly more rectangular circles, as shown in figure 7, in order to write an algorithm that worked for any number of nodes.

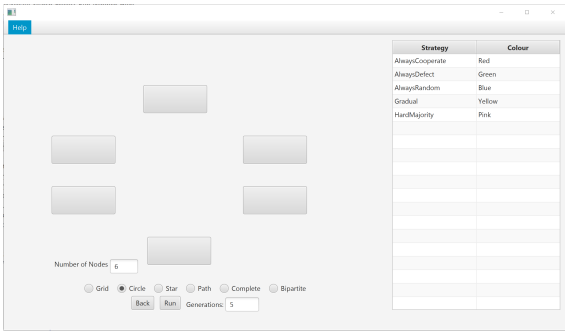


Figure 7: An example of a circular graph generated by the program.

Each type of graph then required a corresponding method to convert the buttons and strategies assigned to them into instances of the node class. This required tailored methods because each type of graph has different criteria for deciding which nodes are neighbours. For example, circle and complete graphs are the same shape and are generated by the same algorithm but in a circle graph adjacent nodes are neighbours and in a complete all nodes are neighbours. This was achieved by assigning each node a numerical id on creation then performing different mathematical operations and comparisons on these ids based on the graph type.

Once the ArrayList of nodes has been created the program is able to launch the results screen and run the evolutionary tournament in the back end. The results of every generation of the tournament are calculated and saved in an ArrayList made up of multiple smaller ArrayLists of Nodes so the user can move between generations as they choose. This is done through the next generation and previous generation buttons on the UI, with the results being displayed in the same way the configuration was set - a graph made up of buttons representing nodes.

4.3 Decision Making

Over the course of my project there were many occasions where I had to try multiple methods of implementing a feature or changed my mind on the best way to solve a problem. The majority of these situations occurred when I was attempting to implement a feature or concept I had never used before, showing just how much I have learnt over the course of the project.

Deciding what was an appropriate scope for my project was difficult. Obviously the aims set out in the brief formed the basis - implement a round robin tournament with a list of in-built strategies along with a relevant GUI to run the tournament and output the results. Given that I successfully implemented the round robin tournament in the first term I felt that was a good indication that I would be able to successfully implement the evolutionary tournament during the second term. To support this I produced a list of milestones to give myself the best possible chance of producing a program with functionality that worked and that I was confident in.

- Implement all back end functionality to run an evolutionary tournament from an ArrayList of nodes that know their strategy and their neighbours.
- Implement a GUI that allows the tournament to be configured on a simple grid, including allowing the user to assign strategies to each square/node in the grid
- Add a screen that outputs the results of an evolutionary tournament using a button to advance generations
- Add GUI functionality allowing the user to select from a list of types of graph to run their tournament
- Allow the user as much control as possible, including choosing graph size and both a next and previous button in the results screen
- Add functionality to export results for both tournaments

I felt that the above list was a reasonable amount of goals for the second term, and knew that if for some reason it turned out to be too many I would still have a program with basic evolutionary tournament functionality as long as I achieved the first three.

The design philosophy of making it as easy as possible to add functionality to my code was incredibly important to me. One example of this is the fact that I decided to use the model view controller design pattern when creating my program. This was the best choice as it allowed for the addition of new functionality and corresponding GUI screens with minimal if any changes required to the existing code. It also allowed me to reuse existing screens and logic where possible, such as how all the choices made in the first screen can be passed to either the round robin or evolutionary tournament. Additionally, I implemented both strategies and tournaments as superclasses and subclasses in order to make my program as modular as possible. This would allow for new strategies and tournaments to be added quickly and efficiently, with no need for repeated code and the super/sub class system in place to ensure all necessary methods were added in each new class.

I initially began by implementing my tournament so it would be run on a two dimensional grid of variable size represented by a two dimensional array of my node class. However, after discussion with my advisor I changed my approach to instead use just the system of each node containing an ArrayList of its neighbours. The benefit of this is that my program is now able to apply the evolutionary tournament to multiple types of graph instead of just grids.

Once I completed the class responsible for running an evolutionary tournament I realised there was a significant amount of repeated code between the evolutionary and round robin classes. My solution to this was to create a superclass for the tournaments that contained getters for variables used in all tournaments such as the decisions made and points earned during, as well as the code run at the end of every game to increase scores and save data needed for the GUI. While the main benefit was to increase the readability of my code it also provides the benefit of making it easier to extend my program in the future by adding more types of tournament.

One of the biggest challenges of my project was producing a front end that clearly displayed the graphs for evolutionary tournaments and allowed the user to intuitively assign each node a strategy. Initially I began by creating a separate FXML file for each graph type that would be displayed in the GUI. Unfortunately this did not easily allow for dynamic generation of graphs meaning that the user would not have been able to specify the number of nodes in a graph. I felt that giving the user as much control over the tournament as possible is important so I decided to investigate other potential implementations. The fact that I was unable to use a pre-made FXML file meant that I needed to dynamically generate the graph within the UI. My first thought was to research whether JavaFX had any inbuilt support for building graphs but it does not so I realised that I had to build the graphs from the components available in JavaFX. My final unsuccessful attempt was to use a GridPane, however this did not allow the user to intuitively assign strategies as the squares in a GridPane cannot be interacted with, they can only hold other components. My successful implementation, as described earlier in the program section, was to use buttons to represent the nodes of the graph.

4.4 Methodology

Testing

All of the back end development was done using test-driven design (TDD). Test-driven design is a software engineering methodology that works by first writing a unit test for the intended functionality then writing code in order to make the program pass that test. The benefit of this is that you can be confident that each small section of code written functions correctly, which is

particularly important given just how many strategies and other classes are necessary to run a tournament. While it is possible to automate testing for a GUI this is something I had no previous knowledge of or experience in. Ultimately I decided that it would not be beneficial to utilise this type of testing as the time taken to learn and successfully implement it would be more than the time taken to perform testing myself through the front end. I was also aware that the way the designer of a program interacts with it can be different to the way an end user does and that this meant I may not find all the errors in my GUI. To combat this problem I enlisted others to test the program for me and report any issues they encountered. This was a worthwhile and successful exercise as it uncovered small things I had overlooked, such as how the buttons should be disabled once the user has selected the run tournament button so they cannot change the strategy of a node after the tournament has concluded.

Version Control

Use of a version control system (in my case SVN), was incredibly important for my project. While in most cases it is only useful for checking when and how code has been altered, I had to use it as a method of recovering my project. A few weeks into the project my laptop became unusable, and without the use of SVN all my work would have been lost, setting my progress back significantly.

Adapted Code

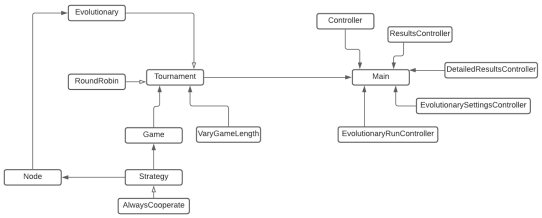
There are a small number of online resources that I adapted code from in order to solves problems and implement functionality.

[6] - I adapted code from this website in order to export the results of my tournament to a CSV file. I was unsure how to create and add data to a CSV file and the code from this tutorial performed that function while I wrote code that formatted the data from my tournament so it could be written to the file.

[9] - This tutorial was useful to me when learning to create a JavaFX GUI. Mainly, I adapted code from it when learning to implement TableViews to display lists of strategies. After I had learnt how to do this I was able to match the functionality to the requirements of my program, such as outputting the position of a strategy in the tournament or allowing the user to set values by editing cells in the TableView.

[14] - I adapted code from this website in order to launch a pdf from the menu of my program. This pdf is the user guideand I felt it was important that this be accessible from within the program itself. While the code I used had error detection the response it had to these errors was to print error messages. My program is GUI based so I had to adapt the code to instead create alerts in the case of an error.

UML



The UML Class diagram above shows the structure of the final program, including the classes used to create and control the GUI. In the diagram an arrow with a filled in arrowhead represents an association between two classes and a hollow arrowhead represents a super and sub class relationship. I decided to exclude most of the detail (variables and methods) from the main diagram in order to make it more readable and useful. However I did not want to exclude this information from the report completely so it is instead included in table format in the appendix - A. Additionally, I excluded all but one of the strategy classes as they are identical for the purposes of the UML and some labels and TextFields in the controller class as there were a large amount that performed similar functions.

Diary Summary

I chose to make diary entries that corresponded to a week each. In each entry I felt that the most important information was what I have achieved that week, the questions I had for my advisor (and his responses) and everything else discussed in the weekly meeting. The summary of my completed work that week is important to document because it shows the evolution of my program each week and shows that I have achieved everything that I wanted to. The list of questions and answers provides an easily accessible reference for me (if my paper notes are not accessible) as well as showing the decisions I have made and options I have considered. The summary of the discussion with my advisor is important as he often makes suggestions or gives feedback on what I have worked on that week.

Timeline

As part of my project plan I created an intended timeline for both the programs and reports that I needed to produce over the course of this project. These can be found in appendix B. However, I very quickly fell out of step with this as I worked at a quicker pace than planned in the first term. Overall this is a good thing as I actually managed to complete goals before they were due but it did mean that some of the project aims that I (and the project brief) had assigned to the second term had been completed by the end of the first term.

The good progress I made in the first term gave me the opportunity to extend my project by adding the functionality to run an evolutionary tournament in the second term. As this was not part of my original intentions for this project it was not included in my original timeline, rendering it essentially useless. Due to how quickly I deviated from the original timeline and the fact I was not confident in my ability to predict how long tasks would take I decided not to make

a revised timeline for the second term's work. Instead, I used the list of milestones detailed in section 4.3 as a way to plan my time. They were deliberately written in the order they should be completed but not assigned time values as I was unsure which sections were most likely to present problems and therefore take the most time.

5 Experiments

Once I had completed the programming section of my project I had a system capable of running experiments using both round robin and evolutionary tournaments. There is an incredibly large number of possible configurations of strategy, graph choice, payoffs and round lengths that could have been investigated so in this section I have included a small number of interesting results and suggested further experiments that could be performed. In the interest of keeping this section easy to follow I have included written descriptions of the starting configurations and results here and screenshots of the results in appendix C. The default values referred to in this section are:

- Rounds: 10
- Game Lengths: 5, 3, 2
- Payoffs:
  - Mutual Cooperation: 3 points each
  - Mutual Defection: 1 point each
  - Defection vs Cooperation: 5 points and 0 points respectively

Experiment 1 - How does changing the ratio of nice to not nice strategies affect which strategies survive in an evolutionary tournament?

This experiment was inspired by Axelrod's conclusion from his second tournament that while a group of nice strategies will succeed when interacting with each other their niceness will be exploited and cause them to lose if there are too many not nice strategies. [2] I decided to run this experiment as an evolutionary tournament on a 4 by 3 grid, with the default number of rounds, default payoffs and randomised game lengths. Strategy placement was random but different for each tournament to prevent a placement that benefitted one strategy more than others from skewing the results.

In order to test Axelrod's conclusion I began by selecting two nice strategies (alwaysCooperate and titForTat) and one not nice strategy (periodicDDC). I deliberately chose two nice strategies that differed in other ways (such as titForTat being retaliatory and alwaysCooperate not being retaliatory) to try and prevent another unconsidered property from impacting the results.

In the first tournament run I placed an equal amount of each strategy on the grid - 4 of each (figure 10). By the second generation of the tournament periodicDDC has taken over the whole board, showing that nice strategies are quite vulnerable to not nice strategies given that 2/3 of the board began by following a nice strategy. It also showed that being retaliatory did not prevent titForTat from being eliminated.

In the second tournament I reduced the representation of periodicDDC to two and increased the amount of the other strategies by one to see if a smaller amount of not nice strategies would allow the nice strategies to thrive (figure 11). Similarly to the last tournament the graph ended up being taken over completely by periodicDDC, this time within three generations instead of two. This does support the theory that nice strategies do better when surrounded by nice strategies but also showed that only a very small amount of not nice strategies is enough to exploit nice strategies into extinction.

My final tournament for this experiment was to populate the graph with only one instance of periodicDDC, five instances of alwaysCooperate and six instances of titForTat (figure 12). Unlike the other two tournaments, with this ratio the nice strategies are able to completely eliminate periodicDDC from the graph. periodicDDC is only present in the first two generations, after which the graph stabilises to eight instances of alwaysCooperate and four instances of titForTat. Interestingly they are split into two separate groups, not mixed together throughout the graph (figure 13).

Overall my experiment did support Axelrod's conclusion on the role niceness plays in a strategy's survival. It is either a great asset (in the case of lots of nice strategies) or such a weakness that even other properties such as being retaliatory can't compensate for it (in the case of lots of not nice strategies).

Experiment 2 - How much do you need to disincentivize defection before it ceases to be an effective strategy?

The purpose of this experiment was to investigate how much of an impact changing the payoffs used in a tournament impacts its results. I decided to run this experiment as an evolutionary tournament on a 4 by 4 grid, the default number of rounds, randomised game lengths and an equal amount of each strategy. Strategy placement was random but kept consistent for each tournament so the only difference between tournaments was the payoffs (figure 14).

I decided to use four strategies in this experiment (alwaysDefect, titForTat, alwaysRandom and Pavlov) so that there was variation in the types of strategies that were included. alwaysDefect was chosen because its performance is the best indicator of how successful a strategy of defection is. I also chose to include alwaysRandom because most strategies aim to disincentivize defection themselves by being either retaliatory or not forgiving and I wanted to include at least one strategy that did not discourage defection.

In the first tournament I used the default payoffs provided by the program which are taken from Axelrod's original tournament and from my research seem to be the most commonly used values. By the second generation the entire graph was populated by alwaysDefect, showing these payoff values massively benefit strategies that defect.

As the first tournament was won by alwaysDefect so quickly I decided to lower the points earned for defection against a cooperating opponent from 5 to 2. In the first generation both alwaysDefect and alwaysRandom were completely eliminated, leaving only the two strategies that choose to cooperate unless defected against (figure 15). This shows that it is possible to lower the reward for defection dramatically enough to cause the elimination of strategies that look to exploit nicer strategies.

I then decided to increase the reward for defection up to 3 to find the exact point at which defection is no longer a viable strategy. In this tournament I found that within three generations all strategies were either Pavlov or titForTat, meaning that lowering the payoff value to 3 still discourages defection to the point of making it not viable as a strategy.

As the previous tournament did not uncover the point at which defection is not viable I ran the last tournament in this experiment, one where a successful defection earns four points. The results of this part of the experiment were much less conclusive than the others. While every other payoff value seemed to produce the same results every time, this value has produced several different outcomes. The most common seems to be that titForTat eliminates every other strategy

but this takes far longer than in the other tournaments (approximately 8 generations), however it is also possible for alwaysDefect to eliminate every other strategy or titForTat and Pavlov to eliminate the other two strategies then coexist.

Overall my results show that it is possible to completely disincentivize defection in a tournament. To completely ensure that defection is not a viable strategy the payoff for a successful defection must be three or below, but to just give nice strategies a better chance then lowering the payoff to four is sufficient.

Experiment 3 - How does the placement of strategies affect their performance in an evolutionary tournament?

During the course of my program's development I realised that strategies with fewer neighbours would be at a disadvantage because they got to play fewer games and rectified this with score normalisation. However there is another disadvantage of having fewer neighbours - there are less nodes that you could possibly take over. As this cannot be fixed without breaking the rules of the evolutionary tournament (nodes can only take over nodes adjacent to them), I wanted to investigate just how much (if any) impact this has on the results of a tournament. I felt the best graph configuration to test this on was a path as it is relatively simple, all nodes have two neighbours apart from the end nodes which have one. I chose a graph made up of four nodes, four different strategies (alwaysCooperate, alwaysDefect, alwaysRandom, and hardMajority) and used the default number of rounds, game lengths and payoffs. The only variable changed between tournaments was the placement of each strategy.

In the first tournament I ran the graph stabilised to three instances of hardMajority and one of alwaysDefect (figure 16).

In the second tournament hardMajority was moved to be one of the end nodes (figure 17). The results of this tournament stabilised to two instances of hardMajority and two of alwaysDefect.

In the third tournament I rearranged the starting configuration again (figure 18) and this resulted in the graph stabilising to three instances of hardMajority and one of alwaysDefect.

The conclusions I have drawn from this experiment is that the starting configuration of the tournament does affect how well strategies perform. hardMajority was clearly the most successful strategy of this group but its representation in the stabilised graph was less when it started on an edge node and therefore had less opportunity to take over other nodes.

Experiment 4 - How much do the results of evolutionary tournaments and Round Robins with the same players differ?

The purpose of this experiment was simply to investigate how similar the results of evolutionary and round robin tournaments are when run with as similar a configuration as possible. I had not come across any research on this during my project and therefore did not have any expectation of what results this experiment should or would produce. I decided to keep all the variables the same, using the program defaults, with only the type of tournament changing.

The strategies I have chosen to use for this experiment are: titForTat, Spiteful, Mistrust and periodicCD. I attempted to choose strategies with a wide variety of properties and approaches, such as being forgiving and nice (titForTat), nice but not forgiving (spiteful), not nice but forgiving (mistrust) and not affected by its opponent in any way (periodicCD).

The results of the round robin tournament were that spiteful and titForTat were the winners with equal scores (figure 19). Mistrust performed the worst, which is not surprising as its initial defection means it becomes locked into a cycle of constant mutual defection with spiteful, titForTat and itself. This is a demonstration of the problems that can occur for a strategy that is not nice if it is playing against strategies that are retaliatory, particularly if they are also unforgiving.

I then constructed what I felt to be the evolutionary tournament most similar to a traditional round robin - one held on a complete graph. This appears to be the most similar as in a complete graph every node is connected to every other node (and therefore plays against every other strategy every generation). A single instance of each of the four strategies was used like in a round robin tournament (figure 20). Surprisingly, the results of this tournament were massively different to the round robin, with mistrust taking over the whole graph after the first generation. After thinking about what differences in the tournaments existed I have come to the conclusion that the only difference was the inclusion of an extra rule in the round robin tournament - a strategy must play against a copy of itself. As a nice strategy is benefitted by playing against itself and a not nice strategy is negatively affected (often entering into a large number of mutual defections) this rule caused a massive difference in the performance of mistrust in the two tournament types.

After observing the results of the first evolutionary tournament I decided to attempt to produce a tournament with closer results to the initial round robin by increasing the number of instances of strategies used. While it would be unfair to add uneven numbers of strategies to a graph, adding an equal number of each strategy would theoretically give each strategy an equal chance of success while forcing strategies to play against themselves like in a round robin (figure 21). As I suspected, the results of this tournament were much more similar to the round robin. The graph stabilised to a mixture of spiteful and titForTat - although not an equal amount of each like their equal scores in the round robin would suggest (figure 22). This proves that while evolutionary tournaments can come to the same or similar conclusions to a round robin, it depends heavily on the amount and placement of the competing strategies. If I were to continue investigating this line of thought I would run the same strategies on different graphs that are less connected and observe how much the results differed to those of the original round robin.

Possible Future Experiments

While I have detailed experiments in this section that I felt were interesting and used the functionality of my program effectively there is such a large number of variables presented to the user that I could not reasonably investigate them all as part of my project. I have instead included a brief list of some more experiments I would have performed had my project been longer or more focused on research using my program.

- Expand upon experiment one by investigating the placement of strategies - Experiment one proved that the ratio of nice to not nice strategies affects which strategies survive to subsequent generations. I would like to expand on this further by investigating whether the placement of these strategies also affects this - i.e. does grouping nice strategies together instead of placing them randomly give them a better chance of survival?
- Investigate how changing the number of total rounds impacts the performance of different strategies in a round robin tournament - There are multiple strategies available in my program that could be affected by how many rounds they get to play. For example, prober

plays three set moves at the beginning of a game to try and discern what strategy its opponent is following and react accordingly in future rounds. If prober were to only play between one to three rounds in a game it would not be able to enact its whole plan and may possibly perform badly because of this.

- Investigate the properties of starting configurations that don't stabilise to a single colouring - All the configurations involved in the experiments I have run so far stabilise to a single colouring. It would be interesting to investigate whether there are starting configurations that lead to graphs that either change forever or constantly fluctuate between a set of states. Once these configurations have been identified they could be used to attempt to work out if there is a set of criteria for creating configurations of this type.

6 Professional Issues

The Prisoner's Dilemma can be applied to an enormous number of real-life disciplines and scenarios. While the experiments I have run using my implementation are from a computer science or mathematical perspective, simply investigating how changing the initial configuration impacts the results, there are many subject areas that use similar experiments to model real-life scenarios. For example, while in my experiments the payoffs for each decision are simply an arbitrary number of points, business people may have them represent profit earned and biologists may use them as resources gained or an indication of how likely an organism is to survive and reproduce. Using the Prisoner's Dilemma to run experiments and draw conclusions about the real world allows for the possibility for ethical issues to arise, particularly when the data and results collected influence future decision making.

Running tournaments of the Prisoner's Dilemma does not necessarily encourage unethical behaviour, in fact it can be used to illustrate the importance of cooperation and acting for the greater good. For example, it can and has been used in classroom environments to demonstrate how an individual's actions affect everyone around them, particularly how selfishness (or unethical behaviour) negatively affects the group overall. [10] This is particularly significant for business students (or anyone working in business management) specifically as there are many opportunities for businesses to act unethically in order to maximise their profits.

Unfortunately, there are certain scenarios that when modelled as the Prisoner's Dilemma show that the optimal decision for an individual player is to act unethically. For example, psychologists and biologists are incredibly interested in how humans and other animals have evolved to cooperate with each other despite it not being the optimal action in a single game of the Prisoner's Dilemma. One example used to explain why it is advantageous to cooperate (and therefore why we have evolved to do it) is to look at the way people in a town interact. They perform actions that have a cost but no benefit attached to them in order to help others such as tipping when eating at a restaurant or driving a neighbour to a hospital because there is a high possibility that they will need similar assistance in the future. This is an example of all players making ethical, non-selfish decisions for the good of the collective, including themselves. However, if a person visited the theoretical town there would be no logical reason to engage in this collective behaviour as they will not be around to benefit from it in the future. This could be used to justify non-ethical behaviour that will have direct (and possibly severe) consequences for the other player - the waitress that doesn't get tipped may not be able to afford food or the person needing medical assistance may die if they do not reach the hospital. This is essentially an example illustrating how the Nash equilibrium of a finite game is to defect (mentioned earlier in this report) and the consequences that making the optimal decision (defect) could have.

There are multiple applications of the Prisoner's Dilemma that could be viewed as unethical that occur when businesses are the players. Firstly, if two businesses are direct competitors and by far the two biggest businesses in their field, such as the Sony PlayStation and Microsoft Xbox competing for the home console market, then they will play to maximise their profits which is not good for consumers. This shows that it is important to examine the wider impact of the results of these games and not just the impact on the participants. The concept of the free market is intended to encourage competition in businesses in order for consumers to receive the best possible products, services and prices. However, the two companies can engage in a Prisoner's Dilemma game in order to avoid cutting prices and profits. Sony knows that if they cut their

prices in order to compete with Microsoft then the logical retaliation by Microsoft is to lower their prices an equal or higher amount in order to maintain competitive pricing. This would leave them both in essentially the same position they started in, except for with a smaller profit margin. While this is good for customers, it not what the businesses want. Instead they both maintain their prices (cooperate) for the good of themselves and each other but not consumers.

A business market that has more than two successful companies can be modelled as a tournament of the Iterative Prisoner's Dilemma with multiple entrants. A successful (but unethical) strategy in tournaments of this kind would be to enter master and slave strategies like the winning team of the IEEE Congress on Evolutionary Computing's tournament [18]. While in the tournament this strategy was regarded as an intelligent and innovative approach, it raises ethical questions when applied to the real world. In the context of businesses the equivalent would be for a business to directly buy smaller competitors and either utilise their capacity/resources to further its own brand or shut them down completely. While the former is still unethical and encourages companies to attempt to establish monopolies, in my opinion the latter is a larger ethical problem as it not only encourages monopolies but would also lead to a large amount of people losing their employment when the company is shut down.

7 Project Evaluation

Upon reflection I feel that my project was very successful in terms of both achieving the aims set out and my educational growth. Over the course of my university education I have taken part in several successful team projects but have not produced work of this scale as an individual project before. I am proud of what I have managed to achieve, especially as I was able to formulate my own extension to the project based on which areas of research in the field I found to be personally interesting.

Once all the round robin functionality and associated GUI components were implemented (and therefore all those project aims completed) I began to think about how I wanted to extend my program beyond the initial project brief. I considered possibilities including allowing the user to play in a tournament, proposing improvements to strategies based on results from my experiments and implementing a different type of tournament. Eventually I decided on implementing an evolutionary tournament as the sheer number of different possible applications was incredibly interesting to me, particularly how we can use such a mathematical model to examine our own psychology and behaviour. Additionally, I had come across multiple research papers concerned with it, some of which were very recent, and I liked the idea of extending the project in a way that was relevant to research currently being performed.

In the end I achieved every goal except for the final one - Allow the user to export the details/results of their tournament to allow for further analysis and saving of results. It was very simple to export round robin tournament data as a CSV file, however due to the fact that evolutionary tournament results are displayed on a graph I struggled to come up with a satisfactory format/method of saving them. After discussing this with my advisor he felt that simply including the previous and next generation buttons along with the ability to jump to any generation was enough as there is no way of exporting the results in a useful format. If I were to have more time I may have added functionality to save and load graph configurations/tournament results to make running and repeating experiments with the program easier.

There are some small fixes and improvements I would make to the program if I were to continue its development. The GUI in particular has some small issues, mostly in the sections that need to display graphs. For example, the buttons generated for the graph can overlap with other components on the screen for certain graph sizes and because of this the maximum size of a graph has been limited. Working more on this and utilising the GUI space better would allow me to increase the limit on the number of nodes per graph and allow the user to run more useful simulations for research, as these are normally done on much larger graphs than currently available in my program. In terms of quality of life improvements I think it would help the user to better understand the results of an evolutionary tournament if the edges of the graph were also generated by the GUI instead of just the nodes. The reason these two changes were not made during my project is that I decided it was better to prioritise ensuring that my program's intended functionality was all present and implemented correctly, especially as I still have relatively little experience in GUI programming and find it to be time consuming.

During the course of my research into the field of the Prisoner's Dilemma, particularly the spatial version, I found many possible features and extensions for my program. As I have focused so heavily on allowing the user as much control as possible it would be a good extension to allow users to add their own strategy classes to the program either through writing their own class or through a system like Oyun's finite state machines [16]. Alternatively, if I wanted to improve my

evolutionary tournament functionality I feel it could be useful to allow the user to create custom graphs, connecting nodes however they wish instead of based on the rules of a particular graph type. This could allow the user to better model specific scenarios, such as how trade between cities works, with nodes representing cities and edges only being assigned to cities with trade routes or agreements. While there is a large number of possible improvements to my program I do not feel this means that my program is inadequate or lacking functionality in any way. This was cemented by the fact that upon comparing my program to a similar system (Oyun) I actually found my program to include a much larger number of customisation options/features for the user. Instead, I feel that these possible extension ideas show just how large the field of study is and how many different ideas and approaches are still being researched.

A Class Diagram Tables

Strategy		
+	points	int
+	tournamentPosition	int
+	rounds	int
+	probability	int
+	addPoints	
+	getPoints	int
+	getDecision	char
+	setProbability	
+	setRounds	
+	setPoints	
+	positionProperty	StringProperty
+	nameProperty	StringProperty
+	probabilityPropery	StringProperty
+	roundsProperty	StringProperty
+	colourProperty	StringProperty

Game		
+	historyStrategy1	ArrayList Character
+	historyStrategy2	ArrayList Character
+	roundScores	ArrayList Integer
+	strategy1	Strategy
+	strategy2	Strategy
+	strategy1move	char
+	strategy2move	char
+	length	int
+	dummyStrat	boolean
+	payoffs	ArrayList Integer
+	player1InitialScore	int
+	player2InitialScore	int
+	decisions	ArrayList Character
+	points	ArrayList Integer
+	getLastMove	char
+	setLastMove	
+	playGame	
+	calculateScores	ArrayList Integer
+	isDummy	boolean
+	getplayer1Score	int
+	getplayer2Score	int
+	getAllDecisions	ArrayList Character
+	getAllScores	ArrayList Integer

VaryGameLength		
+	totalRounds	int
+	random	Random
+	first	int
+	second	int
+	third	int
+	getFirstSet	int
+	getSecondSet	int
+	getThirdSet	int

Tournament		
+	totalRounds	int
+	payoffList	ArrayList Integer
+	scores	ArrayList Integer
+	decisions	ArrayList ArrayList Character
+	points	ArrayList ArrayList Integer
+	lengths	ArrayList Integer
+	setGameLengths	ArrayList Integer
+	first	int
+	second	int
+	third	int
+	player1Score	int
+	player2Score	int
+	returnScores	ArrayList Integer
+	returnDecisions	ArrayList ArrayList Character
+	returnPoints	ArrayList ArrayList Integer
+	returnGameLengths	ArrayList Integer
+	endOfGame	

Evolutionary		
+	nodes	ArrayList Node
+	genScores	ArrayList Integer
+	generations	int
+	allGenerations	ArrayList ArrayList Node
+	setUpTournament	
+	runWholeTournament	
+	runGeneration	
+	updateNodes	
+	returnResults	ArrayList Node
+	setGenerationScores	
+	returnGenerationScores	ArrayList Integer
+	normaliseScores	
+	addNodesToMasterList	
+	returnAllGenerationResults	ArrayList ArrayList Node

Round Robin		
+	strategies	ArrayList Strategy
+	runTournament	
+	setUpTournament	
+	writeToCsv	
+	returnResults	ArrayList Strategy

Node		
+	strategy	Strategy
+	neighbours	ArrayList Node
+	playedAllGames	boolean
+	nodeId	String
+	newStrategy	Strategy
+	getStrategy	Strategy
+	setStrategy	
+	getNeighbours	ArrayList Node
+	addNeighbour	
+	getPlayedAllGames	boolean
+	setPlayedAllGames	
+	getNodeId	String
+	setNodeId	

Main		
-	primaryStage	Stage
-	rootLayout	BorderPane
+	dialogStage	Stage
+	resultsStage	Stage
-	strategyData	ObservableList Strategy
+	tournament	RoundRobin
+	getStrategyData	ObservableList Strategy
+	start	
+	initRootLayout	
+	showOverview	
+	showEvSettings	
+	showEvRun	
+	getPrimaryStage	Stage
+	main	
+	displayResults	
+	closeResults	
+	launchDetailedResults	
+	exportDetailedResults	
+	setTournament	

Controller		
-	strategyTable	TableView Strategy
-	strategyColumn	TableColumn Strategy, String
-	probabilityColumn	TableColumn Strategy, String
-	roundsColumn	TableColumn Strategy, String
-	cc1	TextField
-	cooperate1	Label
-	rounds	TextField
-	game1	TextField
-	game2	TextField
-	game3	TextField
-	roundsCheckBox	CheckBox
-	gamesCheckBox	CheckBox
-	resetPayoffsButton	Button
+	random	Random
+	tournament	RoundRobin
+	gameLengths	ArrayList Integer
+	mainn	Main
+	initialize	
+	setMain	
-	handleRunTournament	
-	launchEvolutionary	
-	cc1Edited	
-	probabilityEdited	
-	roundsEdited	
-	roundsCheckBoxSelected	
-	gamesCheckBoxSelected	
-	resetPayoffs	
-	helpSelected	

ResultsController		
-	resultsTable	TableView Strategy
-	placeColumn	TableColumn Strategy, String
-	strategyColumn	TableColumn Strategy, String
-	pointsColumn	TableColumn Strategy, String
-	resultsGrid	GridPane
-	resultsList	ObservableList Strategy
+	initialize	
+	setResults	
+	setMain	
-	handleBackButton	
-	detailedResultsButton	
-	exportDetailedResultsButton	
+	constructGrid	

DetailedResultsController		
-	resultsLabel	Label
-	scrollPane	ScrollPane
+	initialize	
+	setResults	

EvolutionarySettingsController		
-	mainn	Main
-	graphType	ToggleGroup
-	gridButton	RadioButton
-	circleGraphButton	RadioButton
-	starGraphButton	RadioButton
-	pathGraphButton	RadioButton
-	completeGraphButton	RadioButton
-	bipartiteGraphButton	RadioButton
-	stratsTable	TableView Strategy
-	strategyColumn	TableColumn Strategy, String
-	colourColumn	TableColumn Strategy, String
-	stratsList	ObservableList Strategy
-	graphPane	AnchorPane
-	nodesPane	AnchorPane
-	tf	TextField
-	tf2	TextField
+	limit1	int
+	limit2	int
-	generationField	TextField
+	strategiesForTable	ArrayList Strategy
+	buttons	ArrayList Button
+	nodes	ArrayList Node
+	tfText1	String
+	tfText2	String
+	minnWidth	double
+	minnHeight	double
+	nodeHorizontalDistance	double
+	nodeVerticalDistance	double
+	roundsParam	int
+	payoffsParam	ArrayList Integer
+	gameLengthsParam	ArrayList Integer
+	initialize	
+	setMain	
+	backButton	
+	runCorrectAnchorMethod	
+	setNodesPaneChildren	
+	setSecondNodesPaneChildren	
+	setAnchorGrid	

+	gridRun	
+	setAnchorCircle	
+	circleRun	
+	starRun	
+	setAnchorStar	
+	setAnchorPath	
+	pathRun	
+	completeRun	
+	setAnchorBipartite	
+	bipartiteRun	
-	buttonPressedChangeColour	EventHandler ActionEvent
+	buttonToNode	
+	setButtonData	
+	setTableData	
+	runButton	

EvolutionaryRunController		
+	buttons	ArrayList Button
+	nodes	ArrayList Node
+	buttonsPane	AnchorPane
+	previousButton	Button
+	allGens	ArrayList ArrayList Node
+	evoTournament	Evolutionary
+	generationNumber	int
-	stratsTable	TableView Strategy
-	strategyColumn	TableColumn Strategy, String
-	colourColumn	TableColumn Strategy, String
-	stratsList	ObservableList Strategy
+	totalGens	int
-	chosenGenNumber	TextField
+	mainn	Main
+	rounds	int
+	payoffs	ArrayList Integer
+	gameLengths	ArrayList Integer
+	allStrategies	ArrayList Strategy
+	initialize	
+	showButtons	
+	nextGen	
+	previousGen	
+	setGen	
+	updateNodes	
+	setButtonHandlers	
-	newButtonPressedChangeColour	EventHandler ActionEvent
+	backPressed	

B Timelines

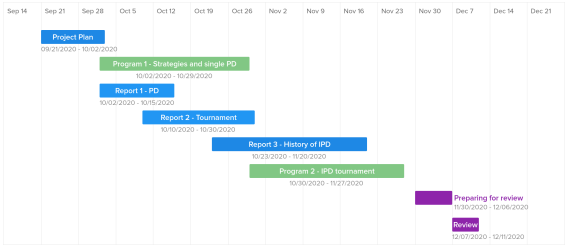


Figure 8: The prospective timeline for my first term of work made as part of my project plan.

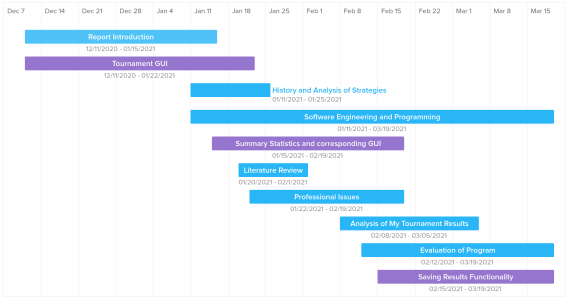


Figure 9: The prospective timeline for my second term of work made as part of my project plan.

C Experiment Screenshots

Experiment 1

Figure 10: The starting configuration of the first tournament I ran as part of experiment 1, showing equal amounts of each strategy in a random placement.

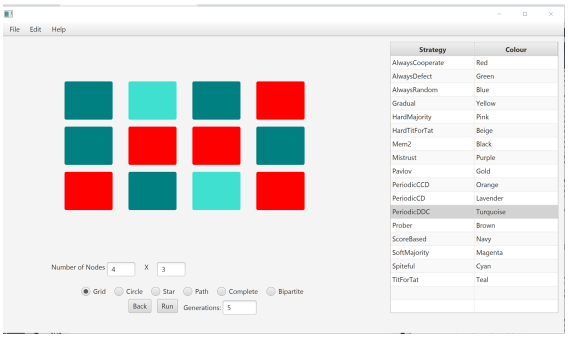


Figure 11: The starting configuration of the second tournament of experiment 1 showing only two instances of periodicDDC.

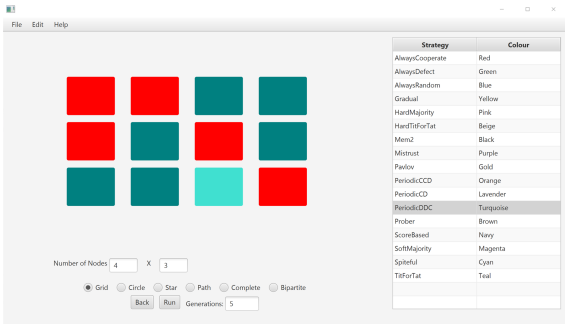


Figure 12: The starting configuration of the third tournament of experiment 1 showing only one instance of periodicDDC.

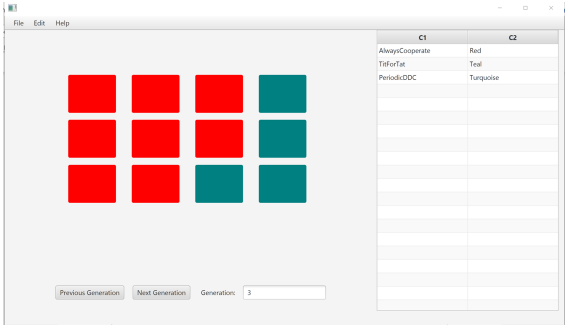


Figure 13: The colouring that the third tournament stabilised to, showing two coexisting strategies - titForTat and alwaysCooperate.

Experiment 2

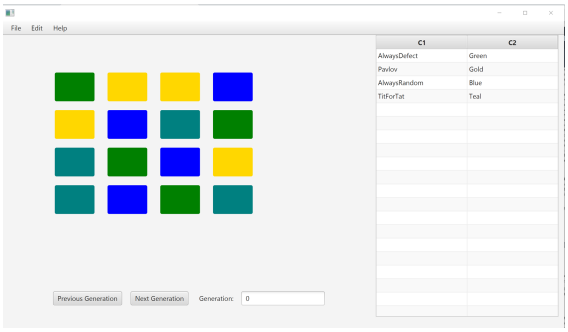


Figure 14: The starting configuration of the first tournament I ran as part of experiment 2, showing equal amounts of each strategy in a random placement.

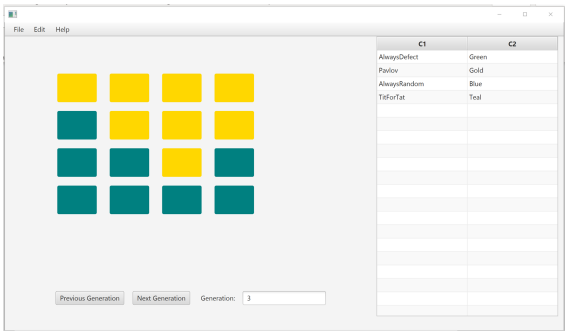


Figure 15: The colouring that the second tournament stabilised to, showing two coexisting strategies - pavlov and titForTat.

Experiment 3

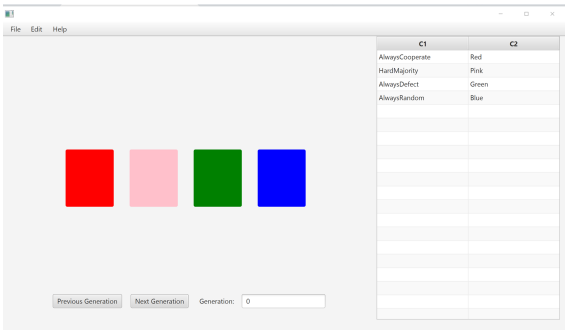


Figure 16: The starting configuration of the first tournament I ran as part of experiment 3, showing one of each strategy in a random placement.

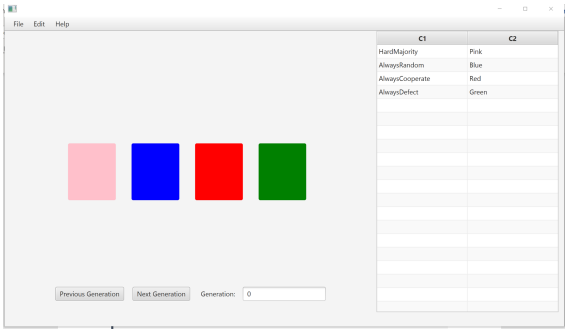


Figure 17: The starting configuration of the second tournament I ran as part of experiment 3, showing one of each strategy, with hardMajority moved to be an end node.

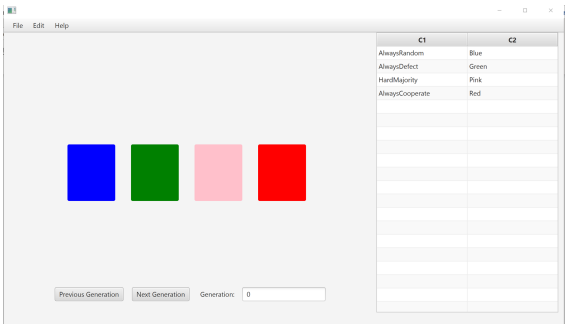


Figure 18: The starting configuration of the third tournament I ran as part of experiment 3, showing one of each strategy, with hardMajority moved back to be a middle node and the positions of other strategies changed as well.

Experiment 4

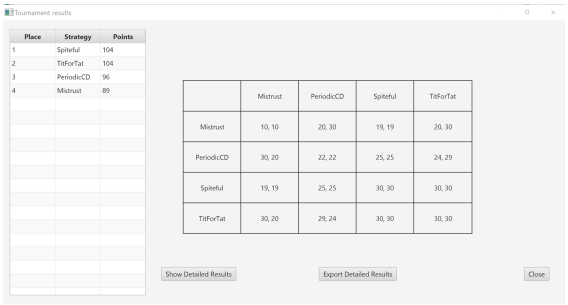


Figure 19: The results of the round robin tournament I ran as part of experiment four.

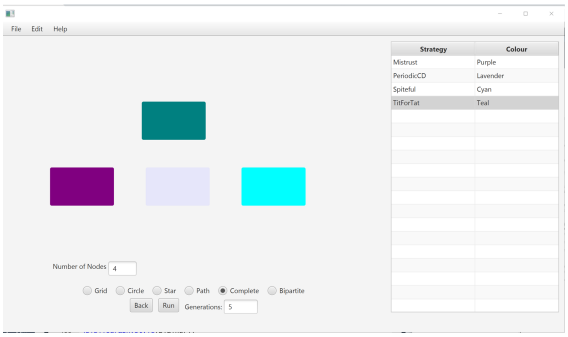


Figure 20: The starting configuration of the first evolutionary tournament I ran as part of experiment 4, showing one of each strategy on a complete graph.

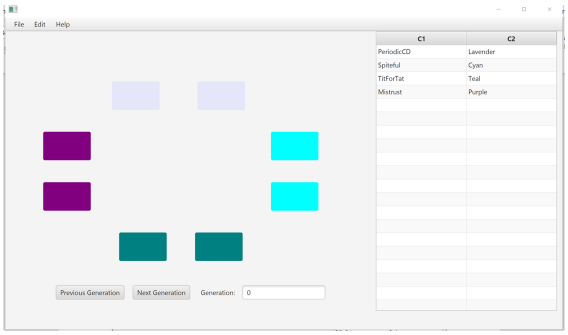


Figure 21: The starting configuration of the second evolutionary tournament I ran as part of experiment 4, showing two of each strategy on a complete graph.

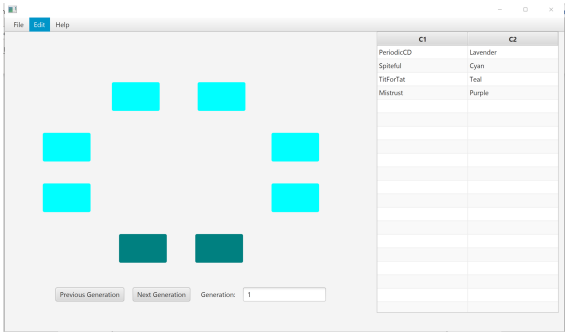


Figure 22: The colouring that the results of the second evolutionary tournament I ran stabilised to, showing 6 nodes following titForTat and 2 following Spiteful.

D Running Instructions and User Guide

Running Instructions

The project is made up of three packages:

- src contains all strategy classes and those necessary to run tournaments.
- testsrc contains all JUnit test classes.
- guisrc contains both java and fxml files used to create and control the GUI.

In my program directory, in the folder labelled Executable, I have created an executable jar file of my program. Running the program requires installation of the Java Runtime Environment which can be found at https://www.oracle.com/uk/java/technologies/javase-jre8-downloads.html but is likely to already be installed on the user's computer. The JavaFX package must be installed - https://gluonhq.com/products/javafx/ contains versions for Windows, MacOS and Linux.The version present in the build path of this program is JavaSE 1.8. The jar file can then be run by navigating to the folder containing it in the command line and entering the command: java –module-path PATH –add-modules javafx.fxml –add-modules javafx.controls -jar ZFAC043.final.jar where PATH is replaced with the path to the JavaFX SDK which should end in \lib.

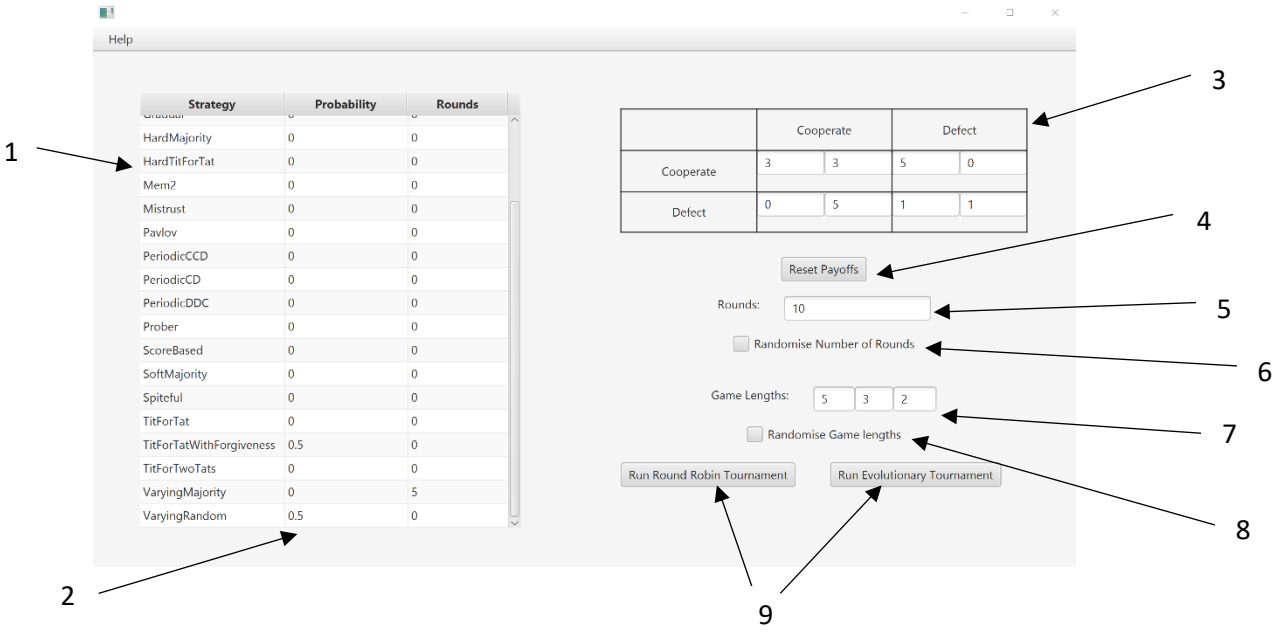
I have created a user guide to explain the functionality of my program in case the user does not have pre-existing knowledge of the prisoner's dilemma or anything is unclear. This can be accessed through the help menu at the top of the program and I have also included it here.

User Guide

User Guide

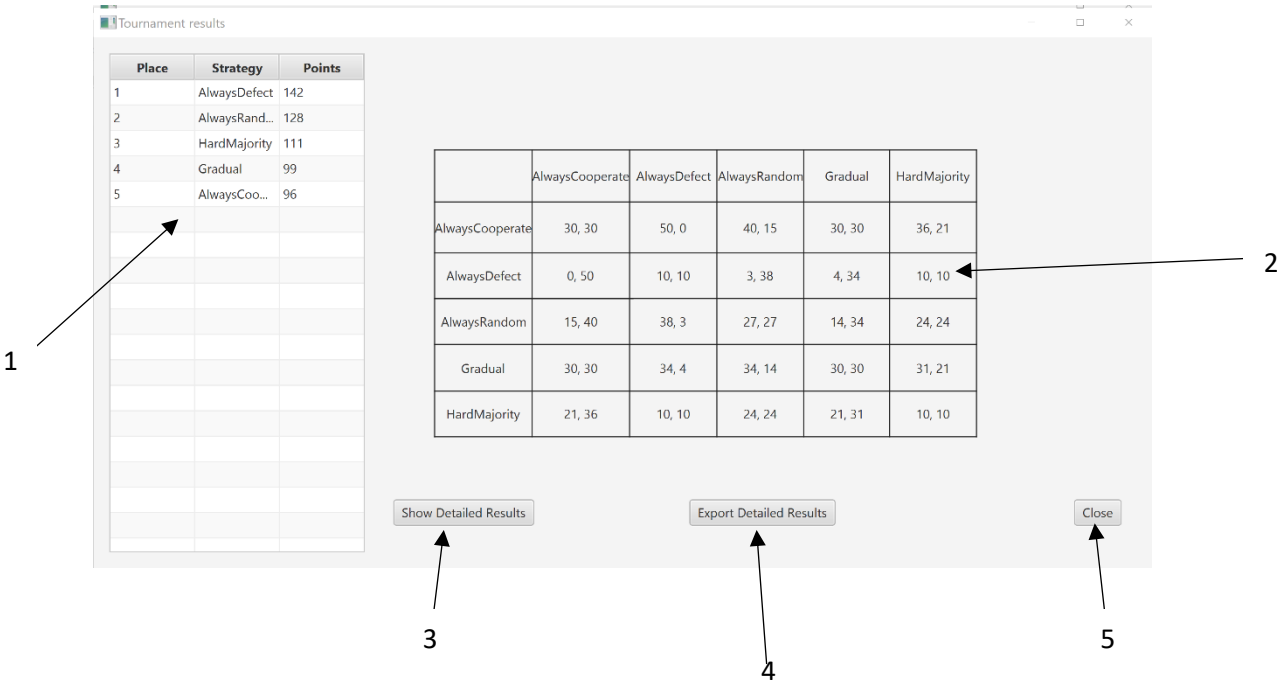
Main Screen

This screen allows the user to configure the details of their tournament and choose which type of tournament they wish to run.



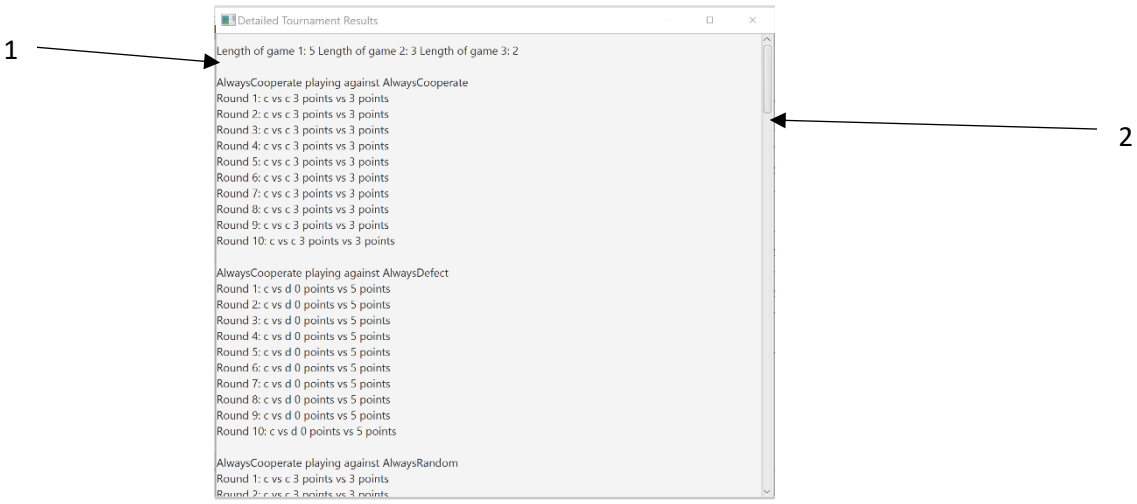
1. This table is used to select which strategies will participate in the tournament by highlighting them.
2. If the value in either the probability or rounds column is not 0 it can be changed by typing in the column.
3. This matrix shows the payoffs for each combination of decisions, they can be changed by typing in the boxes.
4. Pressing this button will return the payoffs to the default values (the ones pictured). The value can be changed by typing in the box.
5. This is the total number of rounds played between each pair of strategies.
6. This will set total rounds to a value between 1 and 100.
7. This is the number of rounds per game (each pairing plays three games together). The total of these values must be the same as the total rounds. The values can be changed by typing in the boxes.
8. This will randomise the length of each game, ensuring they still total to the number of total rounds chosen.
9. These buttons can be used to select which type of tournament will be run.

Round Robin Results Screen



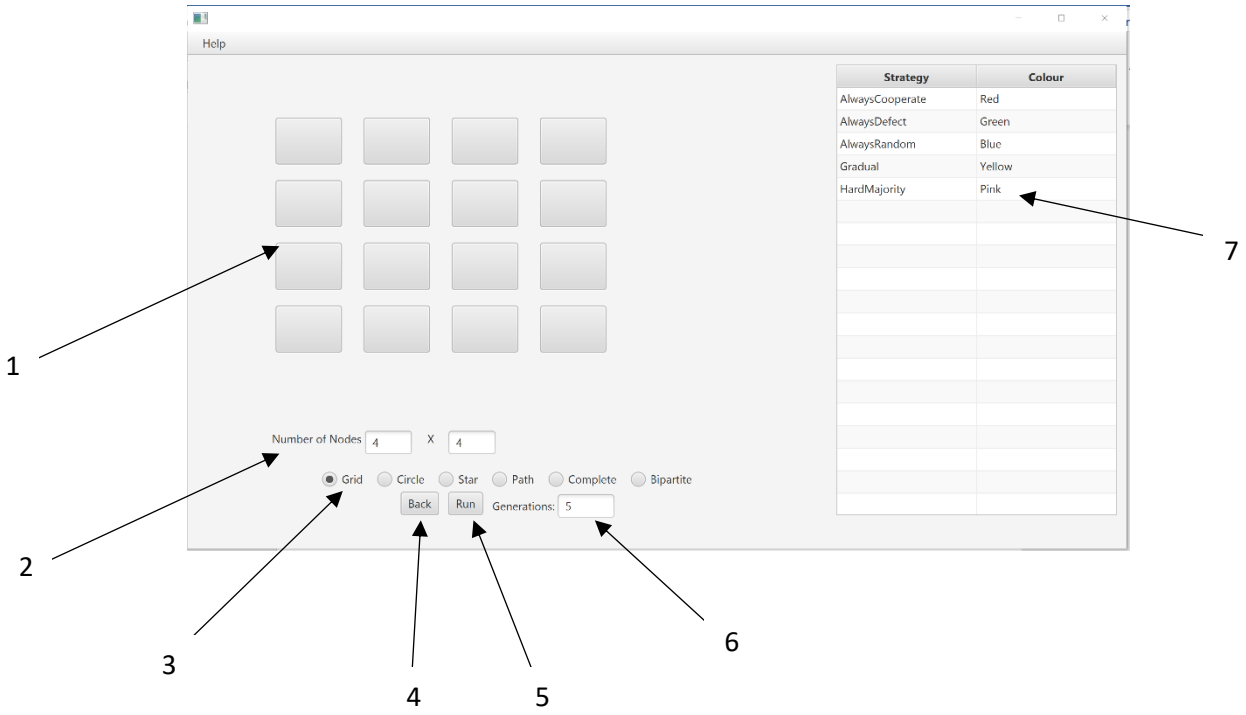
1. This table shows the overall results of the tournament, the first entry is the winner, the second the runner up, etc. Points is the total points a strategy accumulated across the whole tournament.
2. This matrix shows the scores achieved by each strategy in each match.
3. This button launches the program window showing the detailed results of the tournament - the game lengths and the decisions made and points earned in each round of each match.
4. This button takes the information in the detailed results window as well as the score board and exports it to a CSV file.
5. This button closes the screen, returning the user to the main screen.

Detailed Results Screen



1. This screen shows the game lengths and the decisions made and points earned in each round of each match.
2. As there is a lot of information in this window it is scrollable.

Evolutionary Tournament Configuration Screen



1. This is the graph that the tournament will be played on. Clicking on a node with a strategy highlighted in table 7 will assign that strategy to the node. The node will change colour to show this.
2. The number of nodes in the graph can be changed by typing in these boxes.
3. The type of graph used in the tournament can be changed by selecting one of these options.
4. This button takes the user back to the main screen.
5. This button runs the tournament and takes the user to the results screen.
6. The total number of generations run in the tournament can be changed by typing in this box.
7. This table contains all the strategies that can be entered into a tournament. Highlighting a strategy by clicking on it will allow the user to assign it to nodes in the graph.

Evolutionary Tournament Results Screen

1. This is the graph that the tournament results will be displayed on. It will show the state of the nodes at the generation shown in the generation box (4).
2. This button will return the user to the evolutionary tournament configuration screen.
3. This will change the generation results being displayed on the graph, decreasing the generation by 1.
4. This will change the generation results being displayed on the graph, increasing the generation by 1.
5. This will show the generation results currently being displayed on the graph. The user can skip to a specific generation by typing in this box.
6. This table acts as a key, showing the strategy that each colour corresponds to.

References

[1] Robert Axelrod. Effective choice in the prisoner's dilemma. *The Journal of Conflict Resolution*, 24(1):3–25, 1980. URL: http://www.jstor.org/stable/173932.

[2] Robert Axelrod. More effective choice in the prisoner's dilemma. *The Journal of Conflict Resolution*, 24(3):379–403, 1980. URL: http://www.jstor.org/stable/173638.

[3] Robert Axelrod. *The evolution of cooperation*. Basic Books, 2006.

[4] Bruno Beaufils and Delahaye Jean-Paul. Our meeting with gradual: A good strategy for the iterated prisoner's dilemma. 5, 04 1996. URL: https://www.researchgate.net/publication/2697047_Our_Meeting_With_Gradual_A_Good_Strategy_For_The_Iterated_Prisoner's_Dilemma.

[5] Morton D Davis and Steven J Brams. The prisoner's dilemma. *Encyclopædia Britannica*, Jul 1999. URL: https://www.britannica.com/science/game-theory/The-prisoners-dilemma.

[6] Jean Fernando. Reading and writing csvs in java. URL: https://stackabuse.com/reading-and-writing-csvs-in-java/.

[7] Mingjian Fu, Jinbing Wang, Linlin Cheng, and Lijuan Chen. Promotion of cooperation with loyalty-based reward in the spatial prisoner's dilemma game. *Physica A: Statistical Mechanics and its Applications*, page 125672, 2021. URL: https://www.sciencedirect.com/science/article/pii/S0378437120309705, doi:https://doi.org/10.1016/j.physa.2020.125672.

[8] Mingjian Fu, Jinbing Wang, Linlin Cheng, and Lijuan Chen. Promotion of cooperation with loyalty-based reward in the spatial prisoner's dilemma game. *Physica A: Statistical Mechanics and its Applications*, page 125672, 2021. URL: https://www.sciencedirect.com/science/article/pii/S0378437120309705, doi:https://doi.org/10.1016/j.physa.2020.125672.

[9] Marco Jakob. Javafx tutorial. URL: https://code.makery.ch/library/javafx-tutorial/.

[10] Harvey James. Using the prisoner's dilemma to teach business ethics when personal and group interests conflict. *Teaching Business Ethics*, 2:211–222, 06 1998. doi:10.1023/A:1009781017593.

[11] Jiawei Li and Graham Kendall. The effect of memory size on the evolutionary stability of strategies in iterated prisoner's dilemma. *IEEE Transactions on Evolutionary Computation*, 18, 10 2013. doi:10.1109/TEVC.2013.2286492.

[12] Philippe Mathieu, Bruno Beaufils, and Jean-Paul Delahaye. *Studies on Dynamics in the Classical Iterated Prisoner's Dilemma with Few Strategies*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[13] Philippe Mathieu and Jean-Paul Delahaye. New winning strategies for the iterated prisoner's dilemma. *Journal of Artificial Societies and Social Simulation*, 20(4):12, 2017. URL: http://jasss.soc.surrey.ac.uk/20/4/12.html, doi:10.18564/jasss.3517.

[14] Mkyong. How to open a pdf file in java, Apr 2011. URL: https://mkyong.com/java/how-to-open-a-pdf-file-in-java/.

[15] Martin Nowak and Robert May. Evolutionary games and spatial chaos. *Nature*, 359:826–829, 10 1992. doi:10.1038/359826a0.

[16] Charles H. Pence and Lara Buchak. Oyun: A new, free program for iterated prisoner's dilemma tournaments in the classroom. *Evolution: Education and Outreach*, 5(3):467–476, 2012. doi:10.1007/s12052-012-0434-x.

[17] Anatol Rapoport, Albert M. Chammah, and Carol J. Orwant. *Prisoner's dilemma: a study in conflict and cooperation*. Univ. of Michigan Press, 1970.

[18] A. Rogers, R.K. Dash, S.D. Ramchurn, P. Vytelingum, and N.R. Jennings. Coordinating team players within a noisy iterated prisoner's dilemma tournament. *Theoretical Computer Science*, 377(1):243 – 259, 2007. URL: http://www.sciencedirect.com/science/article/pii/S0304397507001831, doi:https://doi.org/10.1016/j.tcs.2007.03.015.

[19] Chengjiang Wang, Li Wang, Juan Wang, Shiwen Sun, and Chengyi Xia. Inferring the reputation enhances the cooperation in the public goods game on interdependent lattices. *Applied Mathematics and Computation*, 293:18–29, January 2017. doi:10.1016/j.amc.2016.06.026.

[20] C Wedekind and M Milinski. Human cooperation in the simultaneous and the alternating prisoner's dilemma: Pavlov versus generous tit-for-tat. *Proceedings of the National Academy of Sciences*, 93(7):2686–2689, 1996. URL: https://www.pnas.org/content/93/7/2686, arXiv:https://www.pnas.org/content/93/7/2686.full.pdf, doi:10.1073/pnas.93.7.2686.