Extended Essay – Mathematics

Application of the Prisoner's Dilemma in a Genetic Algorithm

Upper Canada College

EE Advisor: Ms. Evans

Word Count: 3941

Abstract

The Iterated Prisoner's Dilemma (IPD) has for a long time been a topic of interest to Game Theorists. In this essay, I will incorporate the Iterated Prisoner's Dilemma in a Genetic Algorithm (GA). The GA will attempt to simulate an IPD tournament, where after each iteration, the population of people using each strategy will be changed based on the fitness level of each strategy. The research question is: "What is the difference between the Static Environment and the Growth Environment in terms of the performance of strategies, as well as the pattern of population change?" To assess the question, I will begin by explaining the mechanism of the game. Then, I will introduce 11 well-known strategies of various characteristics. Next, I will outline the structure of a GA, and explain how the program modifies the population of strategies. After conducting the experiment, I reached the following conclusion. First, the population of the strategies diverge almost immediately after the 1st iteration in the Static Environment, whereas the population diverges after the 700th iteration in the Growth Environment. Second, there is a noticeable difference in the pattern of growth between the Static Environment and the Growth Environment. In the Static Environment, the curves for Tit-for-tat and Suspicious Tit-for-tat can be represented by two square-root functions. The populations of these strategies continue to grow, even in the long-term. The curves for strategies like Grudger and Tit-for-two-tats are flat.

The curves for strategies like Hard Major and All Defect exhibit asymptotic behaviour, suggesting that the populations for these strategies will eventually die out. In the Growth Environment, all the strategies exhibit exponential growth in population.

Word Count: 272

Table of Contents

1. Introduction

1.1. Research Motivation

The Prisoner's Dilemma is a standard example of a game analyzed in the field of Game Theory that shows why two completely "*rational"* individuals might not cooperate, even if it appears that it is in their best interests to do so. In Game Theory, the term "rationality" refers to the state in which participants act in a way that maximizes their benefits.[1] The Prisoner's Dilemma can be used to model many real-world situations involving behaviour including climate-change, nuclear arms races, animal behaviour, etc.… It is this wide applicability of the Prisoner's Dilemma that captured my attention. I believe that the Prisoner's Dilemma has many potential usages in the future in the field of economics, psychology, and biology. In this essay, I have applied the repeated version of the Prisoner's Dilemma to a Genetic Algorithm, which models Charles Darwin's Theory of Evolution.

1.2. Scholarly Works

There are two main areas in which I conducted extensive research: the mechanism of the Iterated Prisoner's Dilemma (IPD) and the structure of the Genetic Algorithm (GA). Three primary scholarly works were utilized for the first area of research. "Game Theory: A Critical Introduction" by Shaun P. Hargreaves Heap discusses the propositions proposed to the prisoners in the game, as well as the outcomes as a result of the

[1] "Internet Encyclopedia of Philosophy." Internet Encyclopedia of Philosophy. Web. 14 May 2016.

individuals' actions. "Economics and the Theory of Games" by Fernando Vega-Redondo discusses past experiments conducted with regard to the IPD. Finally, "The Evolution of Cooperation" by Robert Axelrod discusses a well-known IPD tournament, which inspired me to create the static and growth models in this investigation.

For the second area of research, I utilized an academic journal named "Using the Genetic Algorithm to Generate Lisp Source Code to Solve the Prisoner's Dilemma" by Cory Fujiki and John Dickinson at the University of Idaho. This scholarly work outlines the functions and structure of a GA, as well as the results from a past experiment.

1.3. Significance of the Research Question

I believe that the topic of investigation is significant due to the fact that the program can simulate real-life situations in business, such as two firms in the mobile phone industry competing for market share. The Static Environment can be used to model competition in the short-term, where there is no significant growth in the industry. The Growth Environment can be used to model the long-term competition, where an industry might grow due to the increase in the number of sales. The GA will determine which types of strategies would gain the most market share in the Static and Growth Environment, as well as how strategies perform differently in the two distinct environments.

2. Explanation of the Game

2.1. The Prisoner's Dilemma

The Prisoner's Dilemma is a traditional and elegant model for studying decision-making and self-interest. The structure of the game was originally framed by Merrill Flood and Melvin Dresher working at the RAND Corporation in 1950.[2] Albert W. Tucker formalized the game with prison sentence rewards, presenting it as follows:

> Two members of a criminal gang are imprisoned. Each prisoner is in solitary confinement with no means of communicating with the other.[3] Each prisoner is given the opportunity to either defect by testifying that the other committed the crime, or to cooperate with the other by remaining silent.[4] The offer is:
>
> - If Prisoner A and Prisoner B both defect, each of them serves two years in prison[5]
> - If A defects but B cooperates, A will be set free and B will serve three years in prison (and vice versa)[6]
> - If A and B both cooperate, both of them will only serve one year in prison[7]

[2] Kuhn, Steven. "Prisoner's Dilemma." Stanford University. Stanford University, 1997. Web. 10 Apr. 2016.
[3] "Prisoners' Dilemma." : The Concise Encyclopedia of Economics. Web. 10 Apr. 2016.
[4] Ibid
[5] "Prisoner's Dilemma Definition | Investopedia." Investopedia. 2009. Web. 10 Apr. 2016.
[6] Ibid
[7] Ibid

Prisoner B (PB)

Prisoner A (PA)

	Cooperate	Defect
Cooperate	PA: 1, PB: 1	PA: 3, PB: 0
Defect	PA: 0, PB: 3	PA: 2, PB: 2

Table 2.1 *The Prisoner's Dilemma with payoffs as time (# of years) in jail*

In this version of the Prisoner's Dilemma, each prisoner is trying to minimize his or her time in jail. However, Prisoner A has no knowledge of what Prisoner B's move will be because they are not allowed to communicate with each other. Prisoner A knows that if Prisoner B defects, his or her response is to defect as well – he or she receives only two years in jail as opposed to three years if he or she cooperates. Prisoner A also knows that if Prisoner B chooses to cooperate, his or her best response is to defect – he or she receives no jail time if he or she defects as opposed to one year if he or she cooperates. No matter what Prisoner B does, it is in Prisoner A's best interests to defect. Thus, defection is the dominant strategy for Prisoner A. Since a similar analysis holds true for Prisoner B, the *dominant strategy equilibrium* is for prisoners to defect. A dominant strategy equilibrium is defined as the combination of strategies in a game in such a way, that there is no incentive for players to deviate from their choice.[8]

[8] "Policonomics." Policonomics. Web. 14 May 2016.

2.2. The Prisoner's Dilemma in the context of this essay

Player A (PA)	Player B (PB): Cooperate	Player B (PB): Defect
Cooperate	PA: R=3, PB: R=3	PA: S=0, PB: T=5
Defect	PA: T=5, PB: S=0	PA: P=1, PB: P=1

Table 2.2 *The Prisoner's Dilemma with payoffs as points*

Table 2.2 is a strategic form illustration of another classic example of the Prisoner's Dilemma game formulated by political scientist Robert Axelrod. Note that in this example (and from this point forward), Prisoner A and B are replaced by Player A and B, and that the two individuals are trying to maximize their *payoff* as opposed to minimize it as in the previous formulation. A payoff is a number that represents a player's benefits or profits, as well as the motivations of a player.

The game consists for four possible payoffs, represented by letters "R", "S", "T", and "P". "R" represents the reward for mutual cooperation. "S" represents the sucker's payoff. "T" represents the temptation payoff. "P" represents the punishment for mutual defection.[9] In order for the Prisoner's Dilemma to be present, two relationships must be satisfied.

[9] Kuhn, Steven. "Prisoner's Dilemma." Stanford University. Stanford University, 1997. Web. 10 Apr. 2016.

First, the order of decreasing payoff value is as follows: T>R>P>S.[10] The ordering maintains the proper incentive structure, as the temptation to defect must always be greater than the reward for cooperation. Second, the reward for cooperation must be greater than the average of the temptation payoff and the sucker's payoff.[11] Mathematically, this relationship is represented by the following inequality:

$$R > \frac{1}{2}(T + S).$$

This relationship removes the ability for players to take turns to exploit each other to do better than if they played the game egoistically.

A further technical analysis shows how the [Defect, Defect] dominant strategy is undesirable. [Defect, Defect] is not considered a *Pareto efficient* outcome. An outcome in a game is considered Pareto efficient if it is impossible to make any individual better off without making at least one individual worse off.[12] Thus, [Cooperate, Cooperate] is the only Pareto efficient solution to the Prisoner's Dilemma. However, the [Defect, Defect] outcome is not easily avoidable because it is the *Nash equilibrium*. A Nash equilibrium is an outcome in a game with the property that no single player can obtain a higher payoff by changing his or her current move.[13] It follows

[10] Kuhn, Steven. "Prisoner's Dilemma." Stanford University. Stanford University, 1997. Web. 10 Apr. 2016.
[11] Ibid
[12] "Pareto Efficiency Definition | Investopedia." Investopedia. 2009. Web. 10 Apr. 2016.
[13] "Game Theory and Economic Analysis." - 2002, Pg. 42 by Christian Schmidt, Robert W. Dimand, Mary Ann Dimand, Sylvain Sorin, Hervé Moulin, Sébastien Cochinard, Jean-Louis

that [Defect, Defect] is the Nash Equilibrium since a player's best choice is to defect when he or she knows that his or her opponent will defect. Thus, it is impossible for two rational egoists who play the game to end up in any other state than the Nash equilibrium.

2.3. The Iterated Prisoner's Dilemma (IPD)

In the IPD, two players play the normal Prisoner's Dilemma more than once in succession. They remember previous actions of their opponents and change their strategies accordingly. In this essay, a finite version of the IPD will be utilized; two individuals will play the game for a finite number of rounds. Research shows that the best strategy for playing the repeated game is much different from the best strategy for playing just one round. Repeating the game introduces the possibility of *reciprocity* – players are now able to reward or punish each other based on past moves.[14]

The repeated version is a better representation of real-life situations, as two parties usually interact with each other multiple times, and after each interaction, each party will adjust its decision accordingly. An example would a business partnership between two firms. If one firm betrays the other by gaining a significant amount of market share in one interaction, the disadvantaged firm can retaliate in the next interaction.

Rullière, Bernard Walliser, Claude D'aspremont, Louis-André Gérard-Varet, Jean-Pierre Ponssard, Sébastien Steinmetz, Hervé Tanguy. Web. 10 Apr. 2016.

[14] "Reciprocity." Reciprocity. Web. 14 May 2016.

3. Strategies for The Iterated Prisoner's Dilemma (IPD)

3.1. Use of Different Types of Strategies

In this investigation, 5 different categories of strategies are utilized. They are: Cooperative, Aggressive, Self-Interested, Randomizer, and Passive.

3.2. Cooperative Strategies

3.2.1. Tit-for-tat (TFT)

The player using this strategy will choose to cooperate on the first move. From the second move onwards, the player will mimic the opponent's previous move.[15] The strategy is careful, but cooperative. Below are the results of a game of IPD between a player using TFT and a player using a customized strategy (Defect (D), Cooperate (C), Cooperate, Defect, Cooperate).

Round	Tit-for-tat	Customized Strategy
1	C	D
2	D	C
3	C	C
4	C	D
5	D	C

Table 3.1 *Outcomes of a simulated game between TFT and a customized strategy*

[15] "Genetic Algorithms and Their Applications: "Proceedings of the Second International Conference on Genetic Algorithms : July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, Ma"", Pg. 238

3.2.2. Tit-for-two-tats (TFTT)

The player using this strategy will choose to cooperate on the first move. From the second move onwards, the player will defect only when the opponent defects twice in a row. Otherwise, the player will cooperate.[16] TFTT is more forgiving than TFT, as it retaliates after two defections as opposed to one. Below are the results of a five-round game of IPD between a player using Tit-for-two-tats and a player using a customized strategy (C, C, D, D, D).

Round	Tit-for-two-tats	Customized Strategy
1	C	C
2	C	C
3	C	D
4	C	D
5	D	D

Table 3.2 *Outcomes of a simulated game between TFTT and a customized strategy*

[16] "Genetic Algorithms and Their Applications: "Proceedings of the Second International Conference on Genetic Algorithms : July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, Ma"", Pg. 238

3.2.3. Suspicious Tit-for-tat (STFT)

The player using this strategy will choose to defect instead of cooperate on the first move. From the second move onwards, the player will use TFT. STFT does not completely trust its opponent, so it defects first. Below are the results of a five-round game of IPD between a player using STFT and a player using a customized strategy (D, C, C, D, C).

Round	Suspicious Tit-for-tat	Customized Strategy
1	D	D
2	D	C
3	C	C
4	C	D
5	D	C

Table 3.3 *Outcomes of a simulated game between STFT and a customized strategy*

3.2.4. Prober

The player using Prober will start with [D, C, C]. If the opponent has cooperated in the second and third move, the player will defect permanently. Otherwise, the player will utilize TFT for the remaining rounds. Prober is careful and adaptive. It exploits the opponent when the right time comes. This strategy is particularly effective against pure

cooperative strategies such as ALLC, as it can exploit by defecting from the fourth move onwards. Below are the results of a five-round game of IPD between a player using Prober and a player using a customized strategy (C, C, C, D, C).

Rounds	Prober	Customized Strategy
1	D	C
2	C	C
3	C	C
4	D	D
5	D	C

Table 3.4 *Outcomes of a simulated game between Prober and a customized strategy*

3.3. Aggressive Strategies

3.3.1. Grudger

The Grudger stimulates a player who is provocable and unforgiving. The player will choose to cooperate on the first move. He or she continues to cooperate until the opponent defects. Then, he or she will defect permanently. Below are the results of a five-round game of IPD between a player using Grudger and a player using a customized strategy (C, D, C, C, C).

Round	Grudger	Customized Strategy
1	C	C
2	C	D
3	D	C
4	D	C
5	D	C

Table 3.5 *Outcomes of a simulated game between Grudger and a customized strategy*

3.3.2. All Defect (ALLD)

The ALLD strategy is completely an aggressive strategy. The player using this strategy defects on every move. Below are the results of a five-round game of IPD between a player using ALLD and a customized strategy (D, C, D, C, D).

Round	ALLD	Customized Strategy
1	D	D
2	D	C
3	D	D
4	D	C
5	D	D

Table 3.6 *Outcomes of a simulated game between ALLD and a customized strategy*

3.3.3. Hard Major (HM)

The player using Hard Major will defect on the first move. For the next move, if the number of defections of the opponent is greater than or equal to the number of times he or she has cooperated, the player using HM will defect. Otherwise, the player will cooperate. HM is adaptive, as it calculates the ratio between the number of defection and cooperation by the opponent every move. Below are the results of a five-round game of IPD between a player using HM and a customized strategy (C, D, D, D, D).

Round	Hard Major	Customized Strategy
1	D	C
2	C	D
3	D	D
4	D	D
5	D	D

Cooperations: 1
Defections: 2
2 > 1

Table 3.7 *Outcomes of a simulated game between HM and a customized strategy*

3.3.4. Soft Grudger (SG)

The player using SG will cooperate until the opponent defects. He or she will punish his or her opponent with the sequence [D, D, D, D, C, C]. He or she will cooperate after the punishing sequence. This process will be repeated if the fist condition is met. This strategy is a slightly more forgiving strategy compared to the normal Grudger; the opponent is given a second chance (two C's at the end of cycle). Below are the results of a six-round game of IPD between a player using SG and a customized strategy (D, D, D, D, D, D).

Round	Soft Grudger	Customized Strategy
1	C	D
2	D	D
3	D	D
4	D	D
5	D	D
6	C	D

Table 3.8 *Outcomes of a simulated game between SG and a customized strategy*

3.4. Self-Interested Strategies

3.4.1. Pavlov

The Pavlov strategy is an intuitive “win-stay, lose-switch” strategy. The player using this strategy cooperates on the first move. If the player received 5 or 3 points in the last round, he or she will repeat the last choice. Below are the results of a five-round game of IPD between a player using Pavlov and a player using a customized strategy (C, D, C, C, C).

Round	Pavlov	Customized Strategy
1	C	C
2	C	D
3	D	C
4	D	C
5	D	C

3
Repeat C

5
Repeat D

Table 3.9 *Outcomes of a simulated game between Pavlov and a customized strategy*

3.5. Randomizer Strategies

3.5.1 Random (RAND)

The Random strategy is added to better simulate the real world, as certain outcomes occur due to chance. The player using RAND has a 50% chance of choosing “Cooperate” every move. Below are the results of a five-round game of IPD between a player using RAND and a player using a customized strategy (C, D, C, C, C).

Round	Random	Customized Strategy
1	D	C
2	D	D
3	C	C
4	D	C
5	C	C

50% Chance of Defecting every move

Table 3.10 *Outcomes of a simulated game between RAND and a customized strategy*

3.6 Passive Strategies

3.6.1 All Cooperate (ALLC)

The ALLC strategy is completely a passive strategy. The player using this strategy cooperates on every move. This strategy is vulnerable to aggressive strategies such as ALLD. Below are the results of a five-round game of IPD between a player using ALLC and a customized strategy (D, C, D, C, D).

Round	ALLC	Customized Strategy
1	C	D
2	C	C
3	C	D
4	C	C
5	C	D

Table 3.11 *Outcomes of a simulated game between ALLC and a customized strategy*

3.7. Characteristics of Certain Strategies

In the book "Evolution of Cooperation" by political scientist Robert Axelrod, he describes an experiment he conducted to analyze certain strategies. He invited academic colleagues all over the world to devise computer strategies to compete in an IPD tournament. By analyzing top-scoring strategies, he concluded that several conditions must be reached:

- Nice[17]

  The most important criterion is that the strategy must be "nice", that is, it must not defect before its opponent does.

- Retaliating[18]

  Axelrod concluded that the strategy must not be a "blind optimist"; it must retaliate by defecting in some situations. Ruthless strategies can easily exploit these players.

- Forgiving[19]

  Though players defect, they will fall back to cooperating if the opponent does not continue to defect.

- Non-envious[20]

  The strategy must not be selfish. In other words, the strategy does not to try to get more points than the opponent.

[17] Heap, Shaun Hargreaves, and Yanis Varoufakis, Pg. 165.
[18] Ibid
[19] Ibid
[20] Ibid

TFT and TFTT follow nearly all of the guidelines. As a result, I can predict these will be the highest-performing strategies. Conversely, aggressive strategies such as ALLD will likely to perform poorly. The experimental results are going to be presented and analyzed in the rest of the essay.

4. Genetic Algorithm (GA)

The Genetic Algorithm is an adaptive learning system inspired by evolution and heredity. In this particular application, the Genetic Algorithm will be used with a program generator to create source programs.

The GA is used to produce a set of *programs* (called a generation), which attempt to solve a particular problem.[21] In this case, the program is a type of strategy, and the problem is the Iterated Prisoner's Dilemma. Each program is evaluated based on its "*fitness*" level, which is a measure of the optimality of a strategy.[22] The best programs are manipulated to create a new set of programs (the next generation). This process is repeated for a finite number of times.

[21] "Genetic Algorithms and Their Applications: "Proceedings of the Second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, Ma"" - 1987, Pg. 236 by James Edward Baker, Craig G. Shaefer, John H. Holland, Darrell Whitley, G. Deon Oosthuizen, Rick L. Riolo, John J. Grefenstette, Kenneth De Jong, Stewart W. Wilson. Web. 10 Apr. 2016.
[22] "Documentation." Genetic Algorithm Terminology. Web. 14 May 2016.

There are four main sections in a Genetic Algorithm:

- Initialization[23]

  The GA generates a population of a finite number of individuals that are represented by their respective strategies for the IPD.

- Evaluation[24]

  The individuals are evaluated based on their fitness. The fitness is the numerical value of the score an individual received using a particular strategy.

- Modify population
  - Static Environment
    - The number of individuals does not change.
  - Growth Environment
    - The number of individuals grows.

- Iterate[25]

  The program will go back to the Evaluation process. The program will iterate a finite number of times.

[23] "Genetic Algorithms and Their Applications: "Proceedings of the Second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, Ma"", Pg. 237
[24] Ibid
[25] Ibid

5. Methodology

5.1. The Experiment

In the experiment, the payoff structure for the IPD will be defined as [R, T, S, P] = [3, 5, 0, 1]. Initially, there are 2,200 individuals in a population. There are 200 individuals using each strategy. 100 individuals are randomly picked from this population to play the IPD with an outside individual (not within the initial population) using TFT for ten rounds each. The scores are recorded for the outside individual using TFT. Next, the outside individual will use TFTT against 100 random individuals within the population to play the same game for ten rounds each. The scores are recorded for the outside individual using TFTT. This process will be repeated until the outside player has used all 11 strategies.

At the end of the first round, there will be 1,000 scores for each of the 11 strategies. The 1,000 scores for each strategy will be added to generate a total score for each strategy. The score will be the basis for the Evaluation step of the GA.

Static Environment

For the strategies with scores under 1,900, the population of that strategy will be decreased by 3%. The population subtracted will be added to the strategy with the highest score. Mathematically, this could be represented by:

$$POP_{new\_best\_strategy} = [POP_{new\_best\_strategy\_original} + 0.03(POP_{new\_total\_worst\_strategies})].$$

Growth Environment

For the strategies over 2,300, the populations of those strategies will increase by 2% each. Mathematically, this could be represented by:

$$POP_{new\_good\_strategy} = 1.02(POP_{new\_good\_strategy\_original}).$$

Data Recording

The population of each strategy will be recorded every 100 iterations.

5.2. Configuring the GA

To run the GA, the Terminal application on an Apple computer was first launched. Then, the directory that contains the files needed to run the GA was navigated to using the “cd” command. These files include:

Strategies	**Main Body**	**Data Collection**
ALLC.java	EChoice.java	experimentResult.log
ALLD.java	EStrategy.java	
Grudger.java	Experiment.java	
HardMajor.java	Player.java	
Pavlov.java	RandomPlayer.java	
Prober.java	ScoreResolution.java	
Random.java	Strategy.java	
SoftGrudger.java		
SuspiciousTitForTat.java		
TitForTat.java		
TitForTwoTats.java		

Table 5.1 *List of files needed to run the Genetic Algorithm*

```
Robins-MacBook-Air:~ robinlin$ cd Desktop
Robins-MacBook-Air:Desktop robinlin$ cd Experiment2
Robins-MacBook-Air:Experiment2 robinlin$
```

Figure 5.1 *Navigating to directory of ".java" files*

After navigating to the directory, the command "make" is used.

```
Robins-MacBook-Air:Experiment2 robinlin$ make
javac -g RandomPlayer.java
Robins-MacBook-Air:Experiment2 robinlin$
```

Figure 5.2 *Compiling java files to configure GA*

5.3. Input

For the Static Environment, the input of the Genetic Algorithm is the following command:

```
lobins-MacBook-Air:Experiment2 robinlin$ java Experiment 1
```

Figure 5.3 *Running the GA under the Static Environment*

For the Growth Environment, the input of the Genetic Algorithm is the following Command:

```
Robins-MacBook-Air:Experiment2 robinlin$ java Experiment 2
```

Figure 5.4 *Running the GA under the Growth Environment*

In both environments, the number of iterations is automatically set to 1,000, and the number of individuals using each strategy will be printed every 100 iterations.

5.4. Output

The output can be displayed by launching experimentResult.log. The number of players using each strategy will be outputted every 100 iteration. Below is a sample output:

```
Iteration 1
200 of All Defect, 200 of Tit-for-tat, 200 of Tit-for-two-tats, 200 of Pavlov, 200
of Grudger, 200 of Random, 200 of Suspicious-tit-for-tat, 200 of Soft-grudger, 200
of All Cooperate, 200 of Hard Major, 200 of Prober in basket.
```

Figure 5.5 *Sample output for experiment under Growth Environment using 1,000 iterations for the first iteration*

5.5. Evaluating Fitness

The GA decides how much the population of each strategy changes by evaluating the fitness of each strategy. As mentioned previously, the score of the strategy is a measure of its fitness.

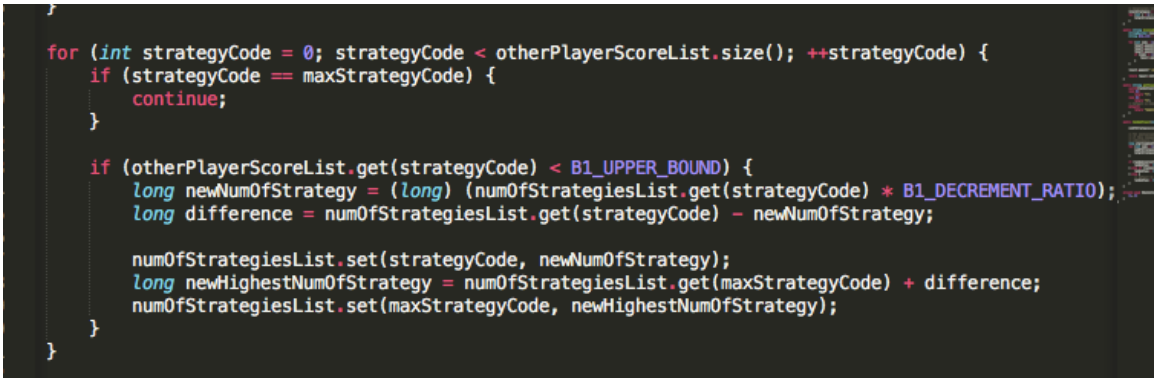
The scores of each strategy will be compared to the score boundaries. For the Static Environment, the variable “B1_UPPER_BOUND” is the score boundary. This is set to 1,900. For the Growth Environment, the variable “B2_LOWER_BOUND” is the score boundary. This set to 2,300.

The other two parameters are the growth factor and the decrement factor. As previously mentioned, the decrement percentage for the Static Environment is 3%. As a result, the decrement factor is 0.97. The increase percentage for the Growth Environment is 2%. As a result, the growth factor is 1.02.

Below are the sections of RandomPlayer.java that decides how the population will be altered:

```
break;
B2:
for (int strategyCode = 0; strategyCode < otherPlayerScoreList.size(); ++strategyCode) {
    if (otherPlayerScoreList.get(strategyCode) > B2_LOWER_BOUND) {
        long newNumOfStrategy = (long) (numOfStrategiesList.get(strategyCode) * B2_INCREMENT_RATIO);
        numOfStrategiesList.set(strategyCode, newNumOfStrategy);
    }
}
break;
```

Figure 5.6 *Growth operator for the Growth Environment*



```
for (int strategyCode = 0; strategyCode < otherPlayerScoreList.size(); ++strategyCode) {
    if (strategyCode == maxStrategyCode) {
        continue;
    }

    if (otherPlayerScoreList.get(strategyCode) < B1_UPPER_BOUND) {
        long newNumOfStrategy = (long) (numOfStrategiesList.get(strategyCode) * B1_DECREMENT_RATIO);
        long difference = numOfStrategiesList.get(strategyCode) - newNumOfStrategy;

        numOfStrategiesList.set(strategyCode, newNumOfStrategy);
        long newHighestNumOfStrategy = numOfStrategiesList.get(maxStrategyCode) + difference;
        numOfStrategiesList.set(maxStrategyCode, newHighestNumOfStrategy);
    }
}
```

Figure 5.7 *Decrement operator for the Static Environment*

6. Results

Below is the list of parameters:

# of Iterations	Frequency of Data Collection	Lower Boundary	Upper Boundary	Decrement Ratio	Increment Ratio
1,000	Every 100 iterations	2,300	1,900	0.97	1.02

Table 6.1 *Summary of parameters used for the experiment*

The results are summarized below in two tables:

Static Environment

Iterations	TFT	TFTT	GRUDGER	PAVLOV	ALLD	SG	STFT	RAND	ALLC	PROBER	HM
1	200	200	200	200	200	200	200	200	200	200	200
101	364	205	97	200	121	109	284	194	254	230	142
201	434	208	58	200	76	48	373	194	254	229	126
301	503	215	28	200	46	24	446	201	254	194	89
401	539	218	14	200	26	4	479	201	254	197	68
501	565	219	0	200	11	0	499	201	254	200	51
601	584	221	0	200	0	0	510	201	254	195	35
701	596	222	0	200	0	0	519	201	254	184	24
801	622	222	0	200	0	0	526	201	254	156	19
901	644	226	0	200	0	0	532	201	254	129	14

Table 6.2 *# of players using each strategy in the Static Environment*

Growth Environment

Iterations	TFT	TFTT	GRUDGER	PAVLOV	ALLD	SG	STFT	RAND	ALLC	PROBER	HM
1	200	200	200	200	200	200	200	200	200	200	200
101	1160	885	200	344	200	200	1116	357	758	689	364
201	8251	6318	208	2124	212	208	7932	2389	5346	2343	1053
301	59625	44309	240	15225	236	224	57310	17143	38578	9290	3605
401	431796	320838	262	110142	287	244	415030	124034	279313	36354	13523
501	3128054	2324198	314	797760	371	282	3006589	898404	2023357	142472	56187
601	22661495	16837831	414	5779338	422	332	21781522	6508470	14658349	547614	243162
701	164174353	121983968	508	41869103	508	378	157799264	47151408	106194389	2466455	934699
801	1189384939	883730537	689	303326676	602	508	1143199673	341595100	769340617	10677819	3813076
901	8616672822	6402314835	885	2197494288	730	626	8282076915	2474735481	5573600349	42706485	16507678

Table 6.3 *# of players using each strategy in the Growth Environment*

Static Environment

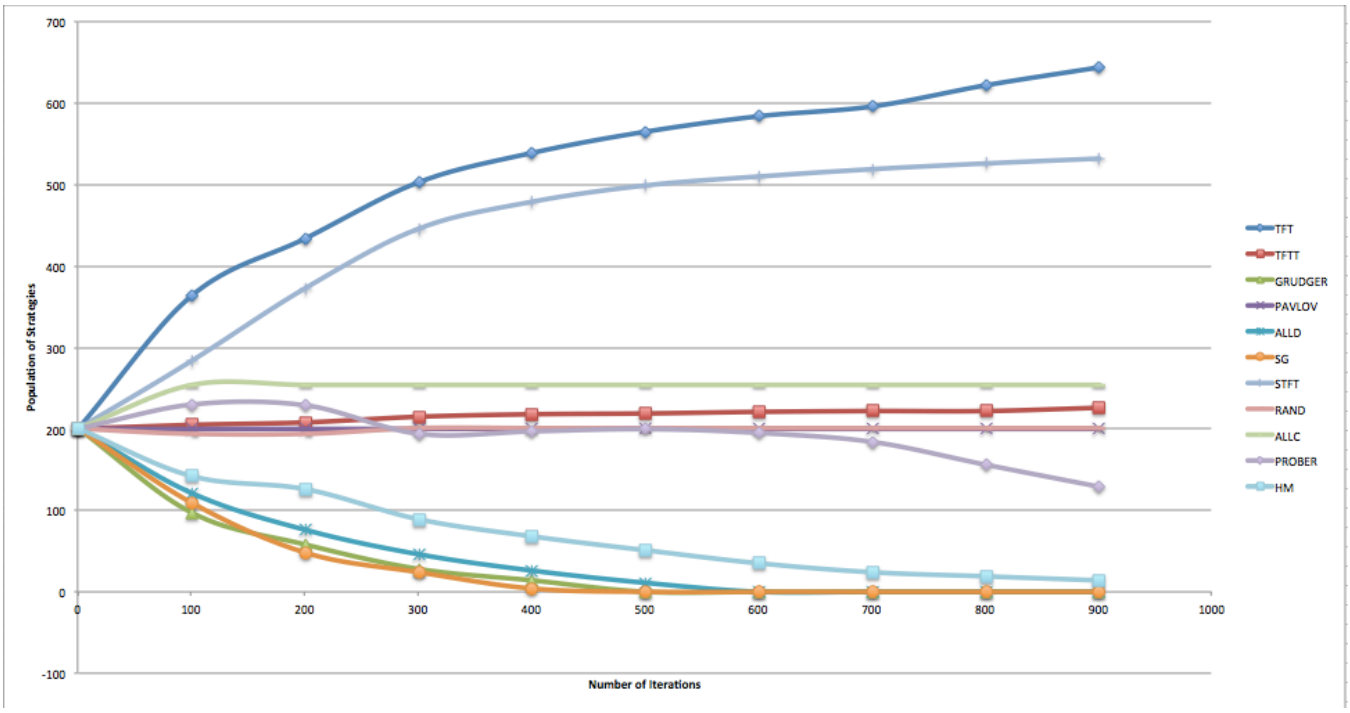


Figure 6.1 *# of players using each strategy in the Static Environment*

Growth Environment

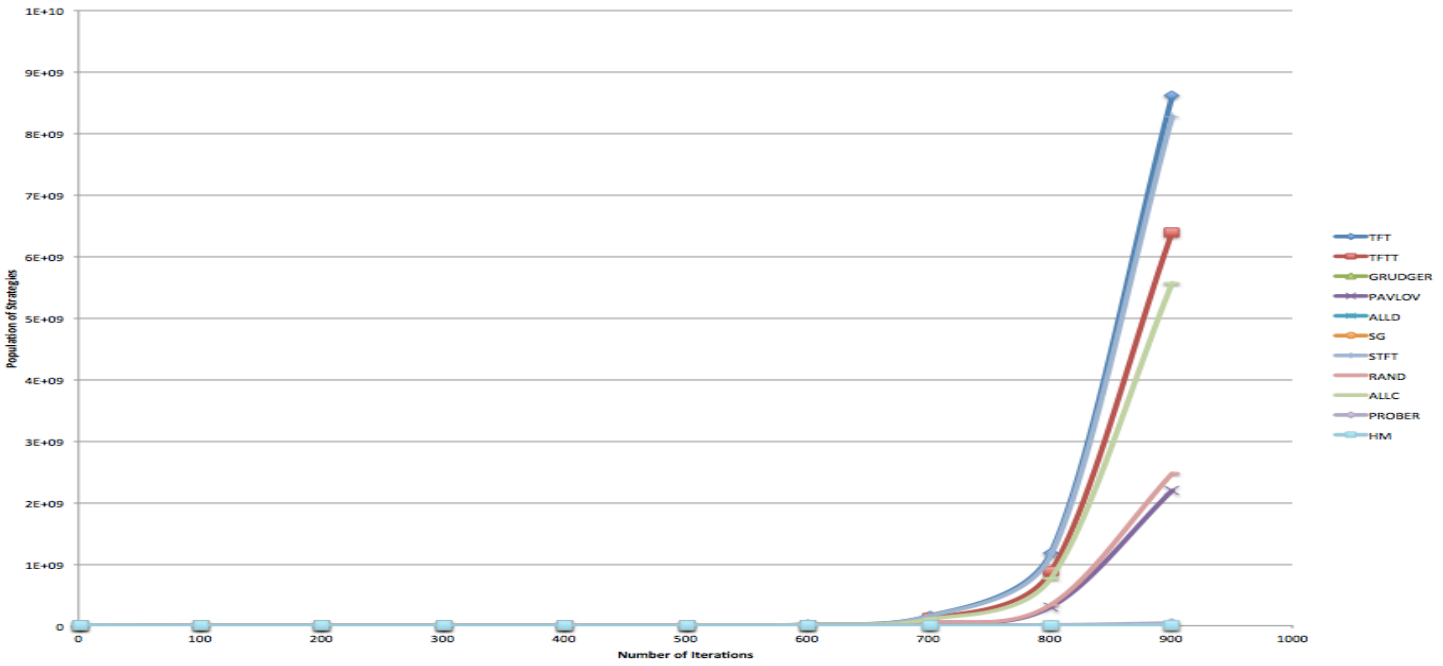


Figure 6.2 *# of players using each strategy in the Growth Environment*

7. Analysis

Static Environment

In the Static Environment, the overall best strategy was TFT, as it had the highest population at the 901th iteration. TFT and STFT saw an overall growth, whereas SG, ALLD, Grudger and HM suffered an overall decline.

The two best strategies had a common characteristic: they were "retaliating". In other words, these two strategies will retaliate by defecting when the opponent defects.

These two strategies are different from ALLC, as TFT and STFT follow a "if you hurt me, I will hurt you back" mentality. In other words, TFT and STFT have a method of defense against aggressive strategies that are attempting to exploit others.

On the other hand, aggressive strategies such as Grudger and HM performed poorly in the Static Environment. These strategies do not possess the characteristics of a successful strategy proposed by Robert Axelrod. For example, both of these strategies are "unforgiving." In the case of HM, when the opponent seems to be exploiting by defecting more often than cooperating, HM punishes the opponent until the rounds end; the opponent is not given a second chance.

Interestingly enough, the population of strategies such TFTT and ALLC stayed relatively the same. An explanation for this phenomenon would be that these are strategies that do not have an effective method of defense. As opposed to TFT, TFTT allows two exploitations in a row. ALLC does not have a method of defense at all. This trend is significant as it shows that players should not be overly “nice”.

As mentioned previously, this model could represent a short-term business competition between two firms. Given the results, we can see that firms that are cooperative tend to do better than firms that are aggressive, as the cooperative firms dominant a larger market share.

In Figure 6.1, it is evident that the population of strategies diverged immediately after the first iteration. This is significant, as it shows that the early stages of a competition can decide the fate of a firm in the short-term; firms that saw growth in the first 100 iterations performed very well at the 901^{th} iteration.

It is now evident that a strategy that possesses the list of characteristics proposed by Robert Axelrod does in fact to better than strategies that do not.

Finally, it is interesting to see that the two best strategies continue to grow even in the 901th iteration, whereas mediocre and poor strategies converged between the 200th and 600th iteration. This is important, as it shows that the best-performing strategies continue to perform well in the long-term. This corresponds with Charles Darwin's concept of "survival of the fittest"; firms utilizing poor strategies will eventually seize to exist in the long-run.

Strategy	**Cooperative**	**Aggressive**	**Other**	**Growth**	**Decline**
Tit-for-tat	√			High	
Tit-for-two-tat	√			Minimal	
Suspicious Tit-for-tat	√			High	
Prober	√			Minimal	
Grudger		√			High
ALLD		√			High
Hard Major		√			High
Soft Grudger		√			High
Pavlov			√		Minimal
Random			√	Minimal	
ALLC			√	Minimal	

Table 6.3 *Summary of the growth and decline of the population of the strategies in the Static Environment*

Growth Environment

In the Growth Environment, the best-performing strategies were TFT, STFT, and TFTT. However, all the strategies seem to exhibit exponential growth. As a result, an effective comparison cannot be made.

However, in Figure 6.2, we can see that the different strategies seem to grow at different rate. Cooperative strategies such as TFT, STFT, and TFTT grew at a faster rate (starting from the 700^{th} iteration), whereas aggressive strategies such as HM at a slower rate. This is significant as it shows that cooperative strategies start to do well early in the competition.

8. Conclusion

In this essay, I have analyzed the difference between the Static Environment and the Growth Environment in terms of the performance of strategies, as well as the overall population change. When comparing the graphs for both environments, it is evident that there is a noticeable difference. First, the population of the strategies diverge almost immediately after the 1st iteration in the Static Environment, whereas the population diverges after the 700th iteration in the Growth Environment. Second, the there is a noticeable difference in the pattern of growth between the Static Environment and the Growth Environment. In the Static Environment, the curves for TFT and STFT can be represented by two square-root functions. The populations for these strategies continue to grow, even in the long-term. The curves for strategies like Grudger and TFTT are flat. The curves for strategies like HM and ALLD seem to be exhibit asymptotic behaviour, suggesting that the populations for these strategies will eventually die out. In the Growth Environment, the curves for all the strategies are exponential in nature.

A limitation to this experiment is the frequency of data collection. If the frequency is set too low, the number data points will decrease. With less data points, the growth between these data points will be less concrete. For example, it is possible that the population first decreases, then increases between two different data points, which might go unnoticed

given a low frequency. To correct this limitation, we can simply increase the frequency.

The conclusions reached in this investigation can be applied to business competition, where two firms are competing for market share. Different firms exhibit different behaviours. However, only a few of them will ever succeed. A prime example is the competition between Apple Inc. and Samsung Group in the mobile industry. These companies interact frequently, as these companies change prices of their manufactured goods according to the current market outlook.

In the future, I believe that the Genetic Algorithm can be applied to wide-range of situations, such as finding the most optimal UPS shipping route and the most effective packing method.

Bibliography

- "Genetic Algorithms and Their Applications: "Proceedings of the Second International Conference on Genetic Algorithms : July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, Ma"" - *1987, by James Edward Baker, Craig G. Shaefer, John H. Holland, Darrell Whitley, G. Deon Oosthuizen, Rick L. Riolo, John J. Grefenstette, Kenneth De Jong, Stewart W. Wilson.* N.p., n.d. Web. 14 May 2016.
- "Game Theory and Economic Analysis." - *2002, by Christian Schmidt, Robert W. Dimand, Mary Ann Dimand, Sylvain Sorin, Hervé Moulin, Sébastien Cochinard, Jean-Louis Rullière, Bernard Walliser, Claude D'aspremont, Louis-André Gérard-Varet, Jean-Pierre Ponssard, Sébastien Steinmetz, Hervé Tanguy.* N.p., n.d. Web. 14 May 2016.
- "Prisoners' Dilemma." *: The Concise Encyclopedia of Economics*. N.p., n.d. Web. 14 May 2016.
- "Prisoner's Dilemma Definition | Investopedia." *Investopedia*. N.p., 2009. Web. 14 May 2016.
- "Pareto Efficiency Definition | Investopedia." *Investopedia*. N.p., 2009. Web. 14 May 2016.
- Kolodny, Niko. "Instrumental Rationality." *Stanford University*. Stanford University, 2013. Web. 14 May 2016.
- Kuhn, Steven. "Prisoner's Dilemma." *Stanford University*. Stanford University, 1997. Web. 14 May 2016.
- "Policonomics." *Policonomics*. N.p., n.d. Web. 14 May 2016.
- "Reciprocity." *Reciprocity*. N.p., n.d. Web. 14 May 2016.

Appendix A: ALLD.java

```
public class AllD extends Strategy {
	@Override
	public EChoice getDecision(EChoice prevOpponentChoice, EChoice
prevPlayerChoice, int playerPrevScore) {
		return EChoice.DEFECT;
	}
	public AllD() {
		strategyType = EStrategy.ALL_D;
	}
}
```

Appendix B: EChoice.java

```
public enum EChoice {
	COOPERATE, DEFECT;
	@Override
	public String toString() {
		switch(this) {
			case COOPERATE:
				return "Cooperate";
			case DEFECT:
				return "Defect";
			default:
				throw new IllegalArgumentException();
```

```
		}
	}
}
```

Appendix C: EStrategy.java

```
import java.util.HashMap;

public enum EStrategy {
	ALL_D(0),
	TIT_FOR_TAT(1),
	TIT_FOR_TWO_TATS(2),
	PAVLOV(3),
	GRUDGER(4);

	private final int m_strategyCode;
	private static HashMap<Integer, EStrategy> infoMap = new
HashMap<Integer, EStrategy>();

	static {
		for (EStrategy s : EStrategy.values()) {
			infoMap.put(s.m_strategyCode, s);
		}

	}

	private EStrategy(int strategyCode) {
		m_strategyCode = strategyCode;
	}

	public int getStrategyCode() {
		return m_strategyCode;
	}

	public static EStrategy valueOf(int strategyCode) {
		return infoMap.get(strategyCode);
	}

	@Override
	public String toString() {
		switch(this) {
			case ALL_D:
				return "All Defect";
			case TIT_FOR_TAT:
				return "Tit-for-tat";
			case TIT_FOR_TWO_TATS:
```

```
                        return "Tit-for-two-tats";
                    case PAVLOV:
                        return "Pavlov";
                    case GRUDGER:
                        return "Grudger";
                    default:
                        throw new IllegalArgumentException();
                }
        }
}
```

Appendix D: Experiment.java

```
import java.io.*;
// import java.util.Scanner;
import java.util.ArrayList;

public class Experiment {

        // private Scanner sc = new Scanner(System.in);
        private PrintWriter logger;

        private Player playerA = null;
        private RandomPlayer playerB = null;
        // each element's index corresponds to EStrategy strategyCode.
        private ArrayList<Integer> scoreList = null;

        private int printPerIteration;
        private int numOfIterations;
        private int numOfGamesPerStrategy = 1000;

        private static boolean verboseMode = false;

        public void run() {

                logger.println("Number of Iterations: " + numOfIterations);
                logger.println("Print Basket every " + printPerIteration + "
iterations");
                logger.println("Player B behaviour: " +
playerB.getPlayerBehaviour());

                for (int iteration = 0; iteration < numOfIterations; ++iteration) {
                        logger.println("\nIteration " + (iteration + 1));

                        if (iteration % printPerIteration == 0) {
```

```
                        logger.println(playerB.displayBasketDetails());
                }

                for (int strategyCode = 0; strategyCode < 5; ++strategyCode)
{

                        playerA.changeStrategyTo(strategyCode);
                        if (verboseMode){
                                logger.println("\nPlayer A switched to strategy "
+ playerA.getCurrentStrategy());
                        }

                        for (int i = 0; i < numOfGamesPerStrategy; ++i) {
                                // Change PlayerB's strategy every 5 games
                                if (i % 5 == 0) {
                                        playerB.reconsiderStrategy();
                                        if (verboseMode) {
                                                logger.println("Player B switched
to strategy " + playerB.getCurrentStrategy());
                                        }
                                }

                                EChoice aChoice = playerA.decideChoice();
                                EChoice bChoice = playerB.decideChoice();

                                int aScore =
ScoreResolution.resolveDecisions(aChoice, bChoice);
                                int bScore =
ScoreResolution.resolveDecisions(bChoice, aChoice);

                                if (verboseMode) {
                                        logger.println("Player A chose: " +
aChoice);
                                        logger.println("Player B Chose: " +
bChoice);
                                        logger.println("Player A got score: " +
aScore);
                                        logger.println("Player B got score: " +
bScore);
                                }

                                playerA.updateGameResult(bChoice, aScore);
                                playerB.updateGameResult(aChoice, bScore);
                        }

                        scoreList.add(playerA.getTotalScore());
```

```
                        playerA.resetPlayer();
                        playerB.resetPlayer();
                        logger.flush();
                }
                for (int i = 0; i < 5; ++i) {
                        logger.println("Strategy " + EStrategy.valueOf(i) + "
score: " + scoreList.get(i));
                }

                playerB.recalculateStrategyProbability(scoreList);
                scoreList.clear();
        }

        logger.flush();
    }

    public Experiment(int[] parameters) throws Exception {

        numOfIterations = parameters[1];
        printPerIteration = parameters[2];

        logger = new PrintWriter("experimentResult.log");

        scoreList = new ArrayList<Integer>();

        playerA = new Player();
        playerB = new RandomPlayer(parameters[0]);

        //playerA.changeStrategyTo(4);
    }

    public static void main(String argv[]) throws Exception {

        if (argv.length > 4) {
                throw new IllegalArgumentException("Number of arguments
exceeded 4");
        }

        int startIndex = 0;

        int[] defaultParameters = new int[3];
        // playerB behaviour
        defaultParameters[0] = 1;
        // numOfIterations
        defaultParameters[1] = 1000;
```

```
		// printEveryXIterations
		defaultParameters[2] = 100;

		if (argv.length > 0) {
			if (argv[0].equals("-v")) {
				if (argv.length < 4) {
					throw new IllegalArgumentException("Must
specify all arguments for verbose mode");
				}
				Experiment.verboseMode = true;
				startIndex++;
			}
			int j = 0;
			for (int i = startIndex; i < argv.length; ++i) {
				defaultParameters[j] = Integer.parseInt(argv[i]);
				++j;
			}
		}

		new Experiment(defaultParameters).run();
	}
}
```

Appendix E: Grudger.java

```
public class Grudger extends Strategy {

	boolean opponentChoseDOnce = false;
	boolean firstTime = true;

	@Override
	public EChoice getDecision(EChoice prevOpponentChoice, EChoice
prevPlayerChoice, int playerPrevScore) {

		EChoice choice = null;

		if (prevOpponentChoice == EChoice.DEFECT) {
			opponentChoseDOnce = true;
		}

		if (firstTime) {
			firstTime = false;
			choice = EChoice.COOPERATE;
```

```
                } else {
                        choice = opponentChoseDOnce ? EChoice.DEFECT :
EChoice.COOPERATE;
                }

                return choice;
        }

        public Grudger() {
                strategyType = EStrategy.GRUDGER;
        }
}
```

Appendix F: Pavlov.java

```
public class Pavlov extends Strategy {

        private boolean firstTime = true;

        @Override
        public EChoice getDecision(EChoice prevOpponentChoice, EChoice
prevPlayerChoice, int playerPrevScore) {
                if (firstTime) {
                        firstTime = false;
                        return EChoice.COOPERATE;
                } else {
                        if (playerPrevScore == 3 || playerPrevScore == 5) {
                                return prevPlayerChoice;
                        } else {
                                return (prevPlayerChoice == EChoice.COOPERATE)
? EChoice.DEFECT : EChoice.COOPERATE;
                        }
                }
        }

        public Pavlov() {
                strategyType = EStrategy.PAVLOV;
        }
}
```

Appendix G: Player.java

```
public class Player {

	protected int totalScore;
	protected int prevScore;
	protected EChoice prevChoice = null;
	protected EChoice prevOpponentChoice = null;
	protected Strategy currentStrategy;

	public Player() {
		prevChoice = EChoice.COOPERATE;
		currentStrategy = Strategy.chooseStrategy(EStrategy.ALL_D);
	}

	public void changeStrategyTo(int strategyCode) {
		changeStrategyTo(EStrategy.valueOf(strategyCode));
	}

	public void changeStrategyTo(EStrategy strat) {
		currentStrategy = Strategy.chooseStrategy(strat);
	}

	public EChoice decideChoice() {
		EChoice choice =
currentStrategy.getDecision(prevOpponentChoice, prevChoice, prevScore);
		prevChoice = choice;

		return choice;
	}

	public void updateGameResult(EChoice opponentChoice, int
receivedScore) {
		prevOpponentChoice = opponentChoice;
		prevScore = receivedScore;
		totalScore += receivedScore;
	}

	public EStrategy getCurrentStrategy() {
		return currentStrategy.getStrategyType();
	}

	public int getTotalScore() {
		return totalScore;
	}

	public void resetPlayer() {
```

```
                totalScore = 0;
                prevScore = 0;
                EChoice prevChoice = null;
                EChoice prevOpponentChoice = null;
                changeStrategyTo(EStrategy.ALL_D);
        }
}
```

Appendix H: RandomPlayer.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.concurrent.ThreadLocalRandom;

import java.io.*;

public class RandomPlayer extends Player {

        // Array that contains the number of each strategies in the basket
        // the index corresponds to the strategyCode in EStrategy
        private ArrayList<Long> numOfStrategiesList = null;
        private long totalStrategies = 0l;
        // Randomization type
        private ERandomType randomType = ERandomType.B1;

        // Experiment Specific parameters
        private final int B1_UPPER_BOUND = 2200;
        private final double B1_DECREMENT_RATIO = 0.8;
        private final int B2_LOWER_BOUND = 2350;
        private final double B2_INCREMENT_RATIO = 1.02;

        public void reconsiderStrategy() {

                int randomStrategyIndex = -1;

                long randomNumber = (long)
(ThreadLocalRandom.current().nextDouble() * totalStrategies);

                for (int i = 0; i < numOfStrategiesList.size(); ++i) {
                        randomNumber -= numOfStrategiesList.get(i);
                        if (randomNumber <= 0l) {
                                randomStrategyIndex = i;
                                break;
                        }
                }
```

```
                changeStrategyTo(randomStrategyIndex);
        }

        public void recalculateStrategyProbability(ArrayList<Integer>
otherPlayerScoreList)
        {
                switch (randomType) {
                case B1:
                        // Get the max score and find out which Strategy it is
                        int max = 0;
                        int maxStrategyCode = -1;
                        for (int strategyCode = 0; strategyCode <
otherPlayerScoreList.size(); ++strategyCode) {
                                if (max < otherPlayerScoreList.get(strategyCode)) {
                                        max = otherPlayerScoreList.get(strategyCode);
                                        maxStrategyCode = strategyCode;
                                }
                        }

                        // All 5 score are smaller than B1_UPPER_BOUND, then
don't change the population
                        if (max < B1_UPPER_BOUND) {
                                return;
                        }

                        for (int strategyCode = 0; strategyCode <
otherPlayerScoreList.size(); ++strategyCode) {
                                if (strategyCode == maxStrategyCode) {
                                        continue;
                                }

                                if (otherPlayerScoreList.get(strategyCode) <
B1_UPPER_BOUND) {
                                        long newNumOfStrategy = (long)
(numOfStrategiesList.get(strategyCode) * B1_DECREMENT_RATIO);
                                        long difference =
numOfStrategiesList.get(strategyCode) - newNumOfStrategy;

                                        numOfStrategiesList.set(strategyCode,
newNumOfStrategy);
                                        long newHighestNumOfStrategy =
numOfStrategiesList.get(maxStrategyCode) + difference;
                                        numOfStrategiesList.set(maxStrategyCode,
newHighestNumOfStrategy);
                                }
```

```
                        }

                        break;
                case B2:
                        for (int strategyCode = 0; strategyCode <
otherPlayerScoreList.size(); ++strategyCode) {
                                if (otherPlayerScoreList.get(strategyCode) >
B2_LOWER_BOUND) {
                                        long newNumOfStrategy = (long)
(numOfStrategiesList.get(strategyCode) * B2_INCREMENT_RATIO);
                                        numOfStrategiesList.set(strategyCode,
newNumOfStrategy);
                                }
                        }
                        break;
                }

                // Recalculate totalStrategies
                totalStrategies = 0l;
                for (int i = 0; i < numOfStrategiesList.size(); ++i) {
                        totalStrategies += numOfStrategiesList.get(i);
                }
        }

        public String displayBasketDetails() {
                StringBuilder result = new StringBuilder();
                String NEW_LINE = System.getProperty("line.separator");

                for (int code = 0; code < 5; ++code) {
                        result.append(numOfStrategiesList.get(code));
                        result.append(" of ");
                        result.append(EStrategy.valueOf(code));
                        if (code != 4) {
                                result.append(", ");
                        }
                }

                result.append(" in basket.");

                return result.toString();
        }

        public String getPlayerBehaviour() {
                switch(randomType) {
                case B1:
                        return "1";
```

```
                case B2:
                        return "2";
                // Default is Undefined
                default:
                        return "Undefined";
                }
        }

        public RandomPlayer(int randomizationType) {

                numOfStrategiesList = new ArrayList<Long>();

                // For initialization, all strategies start equal.
                // if this changes in the future, then split initialization
                // individually for each strategy
                long startingNumOfStrategies = 200l;
                for (int code = 0; code < 5; ++code) {
                        numOfStrategiesList.add(startingNumOfStrategies);
                        totalStrategies += startingNumOfStrategies;
                }

                if (randomizationType == 1) {
                        randomType = ERandomType.B1;
                } else if (randomizationType == 2) {
                        randomType = ERandomType.B2;
                } else {
                        // Default to randomType 1
                        randomType = ERandomType.B1;
                }
        }

        private enum ERandomType {
                B1, B2
        }
}
```

Appendix I: ScoreResolution.java

```
public class ScoreResolution {

        // Returns the score of player1 for one game
        public static int resolveDecisions(EChoice player1Choice, EChoice
player2Choice) {

                int score;
```

```
                if (player1Choice == EChoice.COOPERATE) {
                        score = (player2Choice == EChoice.COOPERATE) ? 3 : 0;
                } else {
                        score = (player2Choice == EChoice.COOPERATE) ? 5 : 1;
                }

                return score;
        }
}
```

Appendix J: Strategy.java

```
public abstract class Strategy {

        protected EStrategy strategyType;

        public abstract EChoice getDecision(EChoice prevOpponentChoice,
EChoice prevPlayerChoice, int playerPrevScore);

        public EStrategy getStrategyType() { return strategyType; }

        public static Strategy chooseStrategy(EStrategy eStrat) {
                switch (eStrat) {
                        case ALL_D:
                                return new AllD();
                        case TIT_FOR_TAT:
                                return new TitForTat();
                        case TIT_FOR_TWO_TATS:
                                return new TitForTwoTats();
                        case PAVLOV:
                                return new Pavlov();
                        case GRUDGER:
                                return new Grudger();
                        default:
                                return null;
                }
        }
}
```

Appendix K: TitforTat.java

```
public class TitForTat extends Strategy {

        private boolean firstTime = true;
```

```
	@Override
	public EChoice getDecision(EChoice prevOpponentChoice, EChoice
prevPlayerChoice, int playerPrevScore) {
		if (firstTime) {
			firstTime = false;
			return EChoice.COOPERATE;
		} else {
		  return prevOpponentChoice;
		}
	}

	public TitForTat() {
		strategyType = EStrategy.TIT_FOR_TAT;
	}
}
```

Appendix L: TitforTwoTats.java

```
public class TitForTwoTats extends Strategy {

	private boolean firstTime = true;
	private boolean opponentChoseDBeforeThis = false;

	@Override
	public EChoice getDecision(EChoice prevOpponentChoice, EChoice
prevPlayerChoice, int playerPrevScore) {

		EChoice choice = null;

		if (firstTime) {
			firstTime = false;
			choice = EChoice.COOPERATE;
		} else {
			if (prevOpponentChoice == EChoice.COOPERATE) {
				choice = EChoice.COOPERATE;
			} else {
				choice = opponentChoseDBeforeThis ?
EChoice.DEFECT : EChoice.COOPERATE;
			}
		}

		opponentChoseDBeforeThis = (prevOpponentChoice ==
EChoice.DEFECT) ? true : false;

		return choice;
```

```
	}

	public TitForTwoTats() {
		strategyType = EStrategy.TIT_FOR_TWO_TATS;
	}
}
```