AN EXAMINATION ON CONVERGENCE TIME OF THE
ITERATED PRISONER’S DILEMMA GAME USING THE
PAVLOV STRATEGY IN THE DIFFERENT TYPES OF
THE INTERACTION GRAPHS

by

Zhi Hao Lin
B.Sc., Shanghai University of Engineering Science, China, 1995
B.Sc., Queen’s University, Canada, 2004

PROJECT SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

In the
School
of
Computing Science

© Zhi Hao Lin 2007

SIMON FRASER UNIVERSITY

Spring 2007

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without permission of the author.
APPROVAL

Name: Zhi Hao Lin
Degree: Master of Computing Science
Title of Project: An Examination on Convergence Time of the Iterated Prisoner's Dilemma Game Using the Pavlov Strategy in the Different Types of the Interaction Graphs

Examinig Committee:

Chair: Dr. Joseph G. Peters
Professor of School of Computing Science

Dr. Petra Berenbrink
Senior Supervisor
Assistant Professor of School of Computing Science

Dr. Funda Ergun
Supervisor
Associate Professor of School of Computing Science

Dr. Arthur L. Liestman
Internal Examiner
Professor of School of Computing Science

Date Defended/Approved: September 21st, 2006
DECLARATION OF PARTIAL COPYRIGHT LICENCE

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the "Institutional Repository" link of the SFU Library website <www.lib.sfu.ca> at: <http://ir.lib.sfu.ca/handle/1892/112>) and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author’s written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, BC, Canada
ABSTRACT

We embed Iterated Prisoner's Dilemma problem into a graph system. Each vertex in the graph corresponds to a player in the IPD game. At each round, an edge in the graph is picked at random, and two players, who are connected by this edge, will play one round of PD game using Pavlov strategy. After that, another edge will be chosen and corresponding players will play again. This game cycle repeats infinitely, until all of the players in the system choose cooperation as their strategy. We call this state as "convergence state", and the number of rounds used to reach convergence state as "convergence time". Our interest is the relationship between the sizes of various types of graphs and their convergence time. This project explored this relationship by computer simulations. In addition, it also provided observations on other related issues.

Keywords: Iterated Prisoner's Dilemma, Pavlov strategy, Prisoner's Dilemma game
DEDICATION

To my dear family who supports me all the time.
ACKNOWLEDGEMENTS

I wish to express my greatest appreciation to my senior supervisor Dr. Petra Berenbrink, for her guidance, support, suggestions, friendship and encouragement in the completion of this thesis during the last two years. I also want to thank Dr. Arthur L. Liestman who exceeded his duty as a committee member by carefully reviewing my work and offering his precious suggestions. Likewise, I would like to thank Dr. Funda Ergun and Dr. Joseph G. Peters for sharing their astounding expertise through their services as committee members. This thesis would have been impossible for me to finish without their professional advice.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approval</td>
<td>ii</td>
</tr>
<tr>
<td>Abstract</td>
<td>111</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>v</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xi</td>
</tr>
<tr>
<td>Glossary</td>
<td>xii</td>
</tr>
<tr>
<td>1 Introduction and Motivation</td>
<td>i</td>
</tr>
<tr>
<td>1.1 Game Theory</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Brief History and Influence</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2 Game Model Constitution and Types</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Iterated Prisoner’s Dilemma</td>
<td>4</td>
</tr>
<tr>
<td>1.2.1 Prisoner’s Dilemma Problem</td>
<td>4</td>
</tr>
<tr>
<td>1.2.2 Importance of the Prisoner’s Dilemma Problem</td>
<td>6</td>
</tr>
<tr>
<td>1.2.3 Iterated Prisoner’s Dilemma Problem</td>
<td>7</td>
</tr>
<tr>
<td>1.3 Typical Strategies for the Iterated Prisoner’s Dilemma Problem</td>
<td>10</td>
</tr>
<tr>
<td>2 Game Simulation</td>
<td>13</td>
</tr>
<tr>
<td>2.1 Project Summary</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Simulation Plan and Game Environment</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1 Simulation Plan</td>
<td>16</td>
</tr>
<tr>
<td>2.2.2 Programming Language</td>
<td>21</td>
</tr>
<tr>
<td>2.2.3 Regression Analysis</td>
<td>22</td>
</tr>
<tr>
<td>2.3 Experimental Results</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1 Data Analysis and Discussions</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1.1 Data Stability</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1.2 Observations About Cycle</td>
<td>27</td>
</tr>
<tr>
<td>2.3.1.3 Observations About Tree</td>
<td>33</td>
</tr>
<tr>
<td>2.3.1.4 Observations About Grid</td>
<td>40</td>
</tr>
<tr>
<td>2.3.1.5 Observations About Complete Graph</td>
<td>50</td>
</tr>
<tr>
<td>2.3.1.6 Impact of the number of initially cooperative players</td>
<td>54</td>
</tr>
<tr>
<td>3 Summary and Future Works</td>
<td>62</td>
</tr>
<tr>
<td>3.1 Summary of Experimental Results</td>
<td>62</td>
</tr>
</tbody>
</table>
3.2 Possible Future Works

Appendices

Reference List
LIST OF FIGURES

Figure 2.1: Interaction graph examples: Cycle, Grid and Complete graph.........................14
Figure 2.2: 0 - 3 tree, min size: 7, max size: 1978, increment: 1....................................24
Figure 2.3: Two trials comparison for case: Cycle, min size: 100, max size: 100000, increment: 100..........................................................25
Figure 2.4: Two trials comparison for case: Binary tree, min size: 7, max size: 4099, increment: 2.........................................................26
Figure 2.5: Two trials comparison for case: 2-row grid, min size: 4, max size: 134, increment: 2................................................................26
Figure 2.6: Two trials comparison for case: Complete graph, min size: 4, max size: 41, increment: 1.........................................................27
Figure 2.7: Cycle, min size: 100, max size: 100000, increment: 100..................................28
Figure 2.8: The average convergence time in proportion to Dyer's lowest upper bound in theorem 1........................................................................................................29
Figure 2.9: Cycle, min size: 100, max size: 53400, increment: 100, ex-edges: 1%...............................30
Figure 2.10: Cycle, min size: 100, max size: 57100, increment: 100, ex-edges: 2%..............30
Figure 2.11: Cycle, min size: 100, max size: 57200, increment: 100, ex-edges: 10%................31
Figure 2.12: Cycle, min size: 100, max size: 57200, increment: 100, ex-edges: 100%...........31
Figure 2.13: Cycle, min size: 100, max size: 4100, increment: 100, noise: 0.1%.................32
Figure 2.14: Cycle, min size: 100, max size: 2200, increment: 100, noise: 0.2%..................33
Figure 2.15: Binary tree, min size: 7, max size: 1999, increment: 2.................................34
Figure 2.16: Ternary tree, min size: 7, max size: 262, increment: 3.....................................35
Figure 2.17: 4-ary tree, min size: 5, max size: 157, increment: 4........................................35
Figure 2.18: 5-ary tree, min size: 6, max size: 96, increment: 5.........................................36
Figure 2.19: Binary tree, min size: 7, max size: 797, increment: 2, noise: 0.1%.................37
Figure 2.20: Ternary tree, min size: 4, max size: 247, increment: 3, noise: 0.1%..................38
Figure 2.21: 4-ary tree, min size: 5, max size: 145, increment: 4, noise: 0.1%......................38
Figure 2.22: Binary tree, min size: 7, max size: 545, increment: 2, noise: 0.2%.................39
Figure 2.23: Ternary tree, min size: 4, max size: 211, increment: 3, noise: 0.2%.................39
Figure 2.24: 4-ary tree, min size: 5, max size: 121, increment: 4, noise: 0.2%.....................40
Figure 2.25: 2-row grid, min size: 4, max size: 150, increment: 2
Figure 2.26: 3-row grid, min size: 9, max size: 150, increment: 3
Figure 2.27: 5-row grid, min size: 16, max size: 148, increment: 4
Figure 2.28: 5-row grid, min size: 25, max size: 150, increment: 5
Figure 2.29: Square, min size: 4, max size: 144, increment: ($\sqrt{n} + 1)^2 - n$
Figure 2.30: 2-row grid, min size: 4, max size: 120, increment: 2, noise: 0.1%
Figure 2.31: 3-row grid, min size: 9, max size: 120, increment: 3, noise: 0.1%
Figure 2.32: 4-row grid, min size: 16, max size: 120, increment: 4, noise: 0.1%
Figure 2.33: 5-row grid, min size: 25, max size: 120, increment: 5, noise: 0.1%
Figure 2.34: Square, min size: 4, max size: 100, increment: ($\sqrt{n} + 1)^2 - n$, noise: 0.1%
Figure 2.35: 2-row grid, min size: 4, max size: 126, increment: 2, noise: 0.2%
Figure 2.36: 3-row grid, min size: 9, max size: 126, increment: 3, noise: 0.2%
Figure 2.37: 4-row grid, min size: 16, max size: 124, increment: 4, noise: 0.2%
Figure 2.38: 5-row grid, min size: 25, max size: 125, increment: 5, noise: 0.2%
Figure 2.39: Square, min size: 4, max size: 121, increment: ($\sqrt{n} + 1)^2 - n$, noise: 0.2%
Figure 2.40: Comparison of complete graphs with various number of initially cooperative players
Figure 2.41: Complete graph, min size: 4, max size: 49, increment: 1
Figure 2.42: Complete graph, min size: 4, max size: 41, increment: 1, noise: 0.1%
Figure 2.43: Complete graph, min size: 4, max size: 40, increment: 1, noise: 0.2%
Figure 2.44: Cycle, min size: 100, max size: 37500, increment: 100, ini_cooped: 50%
Figure 2.45: Binary tree, min size: 7, max size: 3267, increment: 2, ini_cooped: 50%
Figure 2.46: Ternary tree, min size: 4, max size: 307, increment: 3, ini_cooped: 50%
Figure 2.47: 4-ary tree, min size: 5, max size: 145, increment: 4, ini_cooped: 50%
Figure 2.48: 2-row grid, min size: 4, max size: 124, increment: 2, ini_cooped: 50%
Figure 2.49: 3-row grid, min size: 9, max size: 123, increment: 3, ini_cooped: 50%
Figure 2.50: 4-row grid, min size: 16, max size: 124, increment: 4, ini_cooped: 50%
Figure 2.51: 5-row grid, min size: 25, max size: 125, increment: 5, ini_cooped: 50%
Figure 2.52: Square, min size: 4, max size: 121, increment: \((\sqrt{n} + 1)^2 - n\).
ini_cooped: 50%........................................................................................................60

Figure 2.53: Complete graph, min size: 4, max size: 41, increment: 1, ini_cooped:
50%.........................................................................................................................61
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1.1</td>
<td>Payoff matrix of Prisoner’s Dilemma Problem (T &gt; R &gt; P &gt; S, 2R &gt; T+S)</td>
<td>5</td>
</tr>
<tr>
<td>Table 2.1</td>
<td>Game environment table</td>
<td>17</td>
</tr>
<tr>
<td>Table 2.2</td>
<td>Trend line functions when interaction graph is Cycle with various extra edges</td>
<td>29</td>
</tr>
<tr>
<td>Table 2.3</td>
<td>Trend line functions when interaction graph is Cycle, under various noise conditions</td>
<td>32</td>
</tr>
<tr>
<td>Table 2.4</td>
<td>Trend line functions when interaction graph is Tree, in the basic game environment</td>
<td>35</td>
</tr>
<tr>
<td>Table 2.5</td>
<td>Trend lines when interaction graph is Tree, under various noise conditions</td>
<td>37</td>
</tr>
<tr>
<td>Table 2.6</td>
<td>Trend line functions when interaction graph is Grid, in the basic game environment</td>
<td>41</td>
</tr>
<tr>
<td>Table 2.7</td>
<td>Grid trend line functions, under various noise conditions</td>
<td>45</td>
</tr>
<tr>
<td>Table 2.8</td>
<td>Complete Graph trend line functions, under various noise conditions</td>
<td>52</td>
</tr>
<tr>
<td>Table 2.9</td>
<td>“50% ini_coop” vs. “all-defect” trend line function comparison</td>
<td>55</td>
</tr>
</tbody>
</table>
GLOSSARY

0-3 tree: In this project, a 0-3 tree denotes a tree each of whose nodes has a random number between 0 and 3 of children.

Converge: An Iterated Prisoner’s Dilemma system is called “converged” when it reaches the mutual-cooperation status.

Convergence time: The number of game rounds used for the system to converge.

Cooperate: In the Prisoner’s Dilemma game, “Cooperate” is one of two action options that players can choose. If a player chooses to cooperate means that he will keep silent to the prosecutor, and not betray his accomplice.

Defect: In the Prisoner’s Dilemma game, “Cooperate” is one of two action options that players can choose. If a player chooses to defect means that he will confess to the prosecutor, and betray his accomplice.

Ex_edges: An abbreviation of “the number of extra edges”, which is represented by a percentage of the number of edges in the original graph.

Game environment: Any cell in Table 2.1 represents a game environment.

Game model: A game model consists of three factors: a set of players, a set of possible actions (or moves, strategies) they can take, and a set of specified payoffs for each combination of strategies.

Graph size: In this project, the size of a graph refers to the number of vertices in the graph.

Ini_cooped: An abbreviation of “the number of initially cooperative players”, which is represented by a percentage of the number of total players.

Max size: In the charts, it denotes the size of the biggest interaction graph.

Increment: In the charts, it denotes increment on the number of players between two adjacent testing graphs.

Min size: In the charts, it denotes the size of the smallest interaction graph.

Mutual-cooperation status: a status of the Iterated Prisoner’s Dilemma game, in which everybody cooperates.
Noise: The possibility of making mistakes at each action-taking moment.

Pavlov strategy: A strategy in the Iterated Prisoner’s Dilemma game. It will cooperate if it played the same action as its opponent’s in the last round, and will defect otherwise.

Interaction graph: A graph used in this project to help decide which two players can play against each other in the Iterated Prisoner’s Dilemma game. Each vertex of this graph represents a player in the game. Any two players can play against each other only if they have an edge linked in between.

Payoff: In game theory, a payoff is the payment a player gets for his action. It could be a reward or penalty, decided by the actions taken by the payoff receiver and other players.

Payoff matrix: A payoff matrix is a table that shows the payoffs for every possible action by each player for every possible action by each other player.

R-squared value: A number between 0 and 1. It’s used to represent the closeness between the trend line generated by the Microsoft Excel “trendline” feature and the data curve in the chart. The higher this number is, the better the trend line fits the data curve. “1” represents perfect fit.

Square grid: A grid which has the same number of rows and columns.

Strategy: In the Iterated Prisoner’s Dilemma game model, a strategy is a set of rules which give instructions to the players for their next moves based on the history of actions taken by the strategy followers and the corresponding payoffs they receive.

Tit-for-Tat strategy: A strategy in the Iterated Prisoner’s Dilemma game. It will cooperate in the first round, and play the same action his opponent played in the last round for all the rest rounds.

X-row grid: In this project, a grid is defined by its size and the number of its rows. For example, if the size of a 2-row grid is 20, then this grid is a 2×10 grid. Since a 2×10 grid is isomorphic to a 10×2 grid, we assume that the number of rows never exceeds the number of columns in all grids in this project.
1 INTRODUCTION AND MOTIVATION

1.1 Game Theory [1]

1.1.1 Brief History and Influence

Game theory is a theory of decision-making. It studies how one should make decisions and, to a lesser extent, how one does make them. The goal of game theory is to help us understand situations in which decision-makers are involved. In the Oxford Dictionary, a game is defined as “an activity providing entertainment or amusement”. However, its scope in game theory is much larger, since we apply game models in a wide range of disciplines, including economics, politics, and biology etc., to illustrate typical real life phenomena.

Game theory is important, because its models can be applied in lots of situations. For instance: business companies compete for customers; election candidates battle for votes; jury members decide the verdict for a trial; animals cooperate to fight against their enemies; people place bids in an auction; the evolution of siblings’ behaviour towards each other; country leaders’ decision behaviour under pressure from various interest groups; and the long term consequences of one’s action. All of these problems can be better addressed if we have a game model simulating the situations, and have research on the consequences of each possible action option.

Although some very primitive game-theoretic ideas can be traced back until the 18th century, the major development of this discipline started in the 1920’s, by two
famous mathematicians - Émile Borel (1871-1956) and John von Neumann (1903-1957). During 1921 – 1927, Borel published his four notes and first defined the games of strategy. In 1944, Neumann and his co-worker Oskar Morgenstern published a book called “Theory of games and economic behaviour” [2], which symbolizes the establishment of this theory. In 1950, the American mathematical genius John Forbes Nash published two very important articles [3][4], which were summarised as “Nash equilibrium” later on. Being recognized by more and more people, this “Nash equilibrium” finally developed into a key concept in equilibrium theory, and initiated the game-theoretic study of bargaining. Along with more and more game situations being studied, game theory is now involved in many academic fields, ranging from philosophy to sociology and political science etc. In addition, even psychologists began to use game theory models to study how human subjects behave in certain experimental games. In the 1970s, evolutionary biology also joined the party, and began to use the game models as a tool. Now, game-theoretic methods have occupied a dominant role in not only micro-economic sciences, but also a wide range of other social and behavioural sciences.

In this project, we are going to study a game model called “Iterated Prisoner’s Dilemma”, which will be discussed in details later.

1.1.2 Game Model Constitution and Types

A game model consists of three factors: a set of players, a set of possible actions (or moves, strategies) they can take, and a set of specified payoffs for each combination of strategies. At each step, each player chooses an action, and accepts a payoff. A payoff is the payment a player gets for his action. It could be a reward or penalty, decided by the actions taken by the payoff receiver and other players.
Depending on various categorizing criteria, games can be grouped into different types.

1. Symmetric vs. Asymmetric: a symmetric game is a game where the payoffs for playing a particular strategy depend only on the other strategies employed, not on who is playing them. If the identities of the players can be changed without changing the payoffs to the strategies, then a game is symmetric. Otherwise, it is asymmetric.

2. Zero-sum vs. Non-zero-sum: if for any combination of strategies, the total payoff lost by some player, is equal to the total payoff won by another, then it is a zero-sum game; otherwise it is a non-zero-sum game.

3. Simultaneous vs. Sequential: simultaneous game is a game in which players choose their strategies at the same time, while in sequential game the later player already has some knowledge about earlier actions.

4. Perfect information vs. Imperfect information: sequential game can be further divided into perfect information and imperfect information game. A game is perfect information if all of players know the actions taken by all previous players; while imperfect information is the contrary.

5. Infinite vs. Finite: games in real life must finish at some point of time, however, game models with infinite moves are often established to research if there is a winning strategy under certain circumstances.
The game model, Iterated Prisoner’s Dilemma, which we will be focusing in the following part of this paper, is a symmetric, non-zero-sum, simultaneous and infinite game model.

1.2 Iterated Prisoner’s Dilemma

1.2.1 Prisoner’s Dilemma Problem

Two criminals Hans and Becker have been arrested as suspects of a crime. They are placed in separated cells with no communication with each other. The prosecutor does not have enough evidence to charge them, but he knows that they are both selfish persons who care about their own benefits much more than the other’s. The smart prosecutor then makes an offer to them: “If one person confesses while the other remains silent, the one who confesses will be released, and the other person will be charged for more than what he has committed. If both persons confess, then both will get charged for crimes they actually committed; if both persons remain silent, then I will have to charge you both with insufficient evidence.” Now, Hans and Becker both have two possible actions:

1. “Cooperate”, which means to cooperate with his accomplice, and keep silent to the prosecutor. Or,

2. “Defect”, which means to betray his accomplice, and confess to the prosecutor.

Table 1.1 is the payoff matrix of this Prisoner’s Dilemma game model. A payoff matrix is a table that shows the payoffs for every possible action by each player for every possible action by each other player. As we can see, Table 1.1 lists all the possible combinations of actions that might be taken by both players in the “Action” column, and the “Payoff” column shows the payoff to each player if they choose to play the corresponding actions.
For example, if Hans chooses to cooperate (i.e. to remain silent), and Becker chooses to defect (i.e. to confess), then Hans will receive “S”, while Becker will receive “T” as payoff. We have four different payoffs in this game model, namely “T”, “R”, “P”, and “S”. These four payoffs can be anything in real life situations, but two conditions must be met in order for a situation to be considered an application of the Prisoner’s Dilemma model:

1. \( T > R > P > S \)
2. \( 2R > T + S \)

Note: A greater payoff means better benefit, and players always adopt the action which leads to maximum benefit.

Table 1.1: Payoff matrix of Prisoner’s Dilemma Problem (\( T > R > P > S, 2R > T + S \))

<table>
<thead>
<tr>
<th>ACTION</th>
<th>PAYOFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hans</td>
<td>Becker</td>
</tr>
<tr>
<td>Cooperate</td>
<td>Cooperate</td>
</tr>
<tr>
<td>Cooperate</td>
<td>Defect</td>
</tr>
<tr>
<td>Defect</td>
<td>Cooperate</td>
</tr>
<tr>
<td>Defect</td>
<td>Defect</td>
</tr>
</tbody>
</table>

The name of this game, “Prisoner’s Dilemma”, comes from the following observation: whatever the other player chooses, each player is better off confessing than remaining silent. However, the overall outcome obtained when both confess is worse than the outcome that could have been obtained had they both remained silent. Since both players are assumed selfish, they would take the action which would lead to the maximum benefit. However, due to the no-communication rule, they are facing a dilemma when making the decision.
The outcome of this game will always be both players to play “defect”. Since nobody will take the risk of getting the biggest penalty S, so to defect will be their only choice. Unfortunately, this result leads to a group benefit sub-optimal situation, since \(2P < 2R\).

1.2.2 Importance of the Prisoner’s Dilemma Problem

The Prisoner’s Dilemma Problem has been attracted widespread attention in a variety of disciplines ever since it was published. The reason for its high popularity is due to its abundant real life applications.

For instance, in political science, this Prisoner’s Dilemma scenario is often used to illustrate the situations where two countries engage in an arms race. In such cases, both countries have two options: either to increase their military power or to make agreements to achieve mutual security. It is clear that the latter option has better overall benefits. However, in real life, most countries prefer the military option because of the uncertainty of whether the other country will abide by the agreement. As a consequence of this game, we now all live in a world full of nuclear weapons, which is definitely a sub-optimal situation.

We can also find the Prisoner’s Dilemma scenario in sports. Consider two players halfway in a marathon race, with a group of other players trailing behind them at a great distance. The two leading players often work together by sharing the tough load of the first position, where there is no shelter from the wind. If neither of them would like to take the lead, then they will soon be caught up by the following players. However, if one person takes the lead position all the time, then chances are he will be defeated by the
person who follows him. In this case, to cooperate is to take the lead and suffer from the
wind resistance, and to defect is to refuse to take the lead.

Although we can find many applications of the Prisoner’s Dilemma, most
situations in our lives usually involve multiple players, and these players have to play this
Prisoner’s Dilemma game for multiple rounds. That is why in the game theory discipline,
much research now focuses on a more advanced version of the Prisoner’s Dilemma,
namely the Iterated Prisoner’s Dilemma game model.

1.2.3 Iterated Prisoner’s Dilemma Problem

In the Iterated Prisoner’s Dilemma model [5], the number of players is more than
two, and they play the Prisoner’s Dilemma game for multiple rounds. In each round, one
pair of players are picked to play the Prisoner’s Dilemma game. Everybody remembers
the history of his own actions and the corresponding payoffs, and tries to use this
information to help with the decision when the next time he is chosen to play. Like the
Prisoner’s Dilemma game, no communications are allowed among the players during the
entire game time. Therefore, each player doesn’t have any information on how any other
players played in previous rounds, and what their payoffs are, and what they will play in
the next round. Unlike the Prisoner’s Dilemma game, in which players play to maximize
their payoffs in just one round, in the Iterated Prisoner’s Dilemma, players play trying to
maximize their accumulated payoffs when the game is over. The number of rounds that
they play for can be a finite or infinite number, so there are two types of the Iterated
Prisoner’s Dilemma, one is finite and the other is infinite accordingly.
As discussed above, in the Prisoner’s Dilemma situation, the only possible outcome is that both players choose to defect. In fact, not only in the one-round Prisoner’s Dilemma game, but also in any Iterated Prisoner’s Dilemma game, the outcome will be the same as long as the game proceeds for a finite number of rounds.

Why is that? Because in any finitely repeated Prisoner’s Dilemma game, players will all defect in the last round, because the Iterated Prisoner’s Dilemma becomes the Prisoner’s Dilemma at that time. Since all of players know that the last round will be an all-defect round, they will defect in the second last round. This is because players are all selfish, and to defect gives them more benefit than to cooperate. The only reason that will make them cooperate is that they hope by cooperating they will get cooperation back from their opponents and hence create a win-win situation. When they know that in the next round (i.e. the last round) they won’t get the cooperation back, they have no choice but to defect. Therefore, everybody will defect in the last and the second last rounds. This chain-action could be tracked back round by round until the first round, and makes the finite version of Iterated Prisoner’s Dilemma game nothing but a series of independent Prisoner’s Dilemma games. However, if we play the infinite version of Iterated Prisoner’s Dilemma game, then by applying some strategies, the outcome of the game might be mutual-cooperation, which means that everybody cooperates at a certain round, and all rest rounds from that point on. The reason why the infinite Iterated Prisoner’s Dilemma game could have different outcome with the finite version is simply that players don’t need to worry about being retaliated at the last round, so that they could use some strategy which will instruct players to cooperate under certain conditions and finally reach the mutual-cooperation status.
The “strategy” that we are referring here means a set of rules which give players instructions on which action they should take in the next move based on the actions they have taken in the history and the payoffs they have received as a consequence of these actions. One research goal of the Iterated Prisoner’s Dilemma game is to find good strategies. A good strategy is supposed to:

1. Maximize the benefit of the players who execute it.
2. Help eliminate all the defects in the system.

In order to discover new strategies, Robert Axelrod, a professor of Political Science and Public Policy at the University of Michigan, conducted an Iterated Prisoner’s Dilemma Tournament [6]. He invited academic colleagues all over the world to design their strategies, and compete in the tournament. Although all of these strategies vary in complexities, initial action, degree of generosity etc., professor Axelrod summarized several important characteristics out of those which performed well in the contest:

1. Nice: the strategy must be nice, which means a player will not defect before his opponent does. Almost all of top-scoring strategies have this character.

2. Retaliating: although being nice is necessary, a good strategy must also retaliate. An always-cooperate strategy will easily be exploited by other strategies.

3. Forgiving: in order for the system to eventually remove all of defects, the strategy also need to include some mechanism to
forgive opponents who defected before, and cooperate to them again.

4. Non-envious: which means the strategy is not aiming to obtain more scores than opponents in each round.

1.3 Typical Strategies for the Iterated Prisoner’s Dilemma Problem

Professor Robert Axelrod has held two times of the Iterated Prisoner’s Dilemma Tournament. In Axelrod’s first competition tournament, every program played with each other, and the winner was the one with the highest cumulative payoffs. The winning strategy of that contest was Tit-for-Tat, which was invented by Anatol Rapoport [7].

After the results of the first tournament were announced, Axelrod ran a second tournament, this time he added in a probability that the game would end after each round. Among all of 62 entrants, the winner was again Anatol Rapoport and his Tit-for-Tat.

The Tit-for-Tat strategy is very simple. It will cooperate in the first round, and play the same action his opponent played in the last round for the rest rounds: if its opponent cooperated to him, then it will cooperate in the next round; otherwise, it will defect in the next round.

Although Tit-for-Tat won two tournaments, it does not mean that Tit-for-Tat is the best strategy [8] [9] [10]. First, Tit-for-Tat never beat any other strategy in a one-on-one contest. It won by never losing much to each opponent, and adding up the scores to be the highest. In other words, if this were an elimination competition, Tit-for-Tat would have been eliminated very early. Second, the success of Tit-for-Tat was partially decided by other players’ strategies, since its scores were gained through competing with each
other players. In fact, some strategies, which did poorly in the second tournament, could have won the title if they were in the first tournament. Third, Tit-for-Tat always retaliates a defect-play by playing the same in the next round. Therefore, Tit-for-Tat never eliminates defect-plays but only forward them to other players. If all players in the system use the Tit-for-Tat strategy, then once a defect-play occurs in the system, the system will never reach the mutual-cooperation status. You may have noticed that Tit-for-Tat strategy doesn’t generate defect-plays by itself, since it cooperates at its first move, and then repeat its opponent’s last play. As long as all players use the Tit-for-Tat strategy, we should never be worried about a defect-play being introduced into the system. However, this ideal situation only exists in the experimental environments, where it’s error free. In the real world, people make mistakes. If some player mistakenly takes a defect-play, then this defect-play will linger in the system, until another player takes a cooperate-play by mistake.

After these flaws have been discovered, many revised versions of Tit-for-Tat have been created. The Tit-for-Two-Tat strategy offers a certain amount of leniency by always cooperating unless the opponent defects twice in a row. Another similar variant is General-Tit-for-Tat [9], which always cooperates, and only defects with a certain probability every time the opponent defects. These modified strategies improved in quickly eliminating the defect-plays in the system by distinguishing occasional bad behaviour and systematic bad behaviour, and not punishing occasional bad behaviour as much as systematic bad behaviour. However, they cannot maximize the benefit when they are playing against some soft strategies which lack mechanism to retaliate defect-plays. For example, when playing against the always-cooperate strategy, which will
always cooperate in all rounds unconditionally, the always-defect strategy will obviously perform better than any of the above Tit-for-Tat related strategies in terms of maximizing the players’ benefit. The Pavlov strategy, which is a completely Tit-for-Tat irrelevant strategy, on the other hand, can perform much better in such situations. [11]

The Pavlov strategy is a so-called “win-stay, lose-shift” strategy, which means it does not choose “cooperate” or “defect” directly, but only considers whether it needs to switch the decision based on the benefit received in the last round. If the player was rewarded by receiving R or T points, then he will repeat the last play; if he was punished by receiving P or S points, then he will switch to the opposite of the last play. By case analysis, we can find that the Pavlov strategy can also be summarized as “Player will cooperate if and only if he took the same action as his opponent’s in the last round”. So if player A and his last opponent both cooperated or both defected, A will cooperate in the next round, otherwise A will defect in the next round.

Although Pavlov surpasses all the Tit-for-Tat related strategies when facing soft strategies like “always cooperate”, it actually will be exploited by some tough strategies, for instance: “always defect”. When the Pavlov strategy plays against the “always defect” strategy, it will switch its play in every round, and lose benefit every time it switches to cooperate. Therefore, although the Pavlov strategy is one of the best strategies found by far, it is still not a perfect one.
2 GAME SIMULATION

2.1 Project Summary

In this project, we are going to use the computer program to simulate the situation where a group of players play the infinite version of the Iterated Prisoner’s Dilemma, and all players adopt the Pavlov strategy to decide their actions.

As discussed above, the Iterated Prisoner’s Dilemma game proceeds round by round, and one pair of players are chosen to play the Prisoner’s Dilemma at each round. The way how two players are chosen in each round is not totally by randomness. Instead, we use a graph, which is called the “interaction graph” in this project (We drew some interaction graph examples in the Figure 2.1, and represented each player in the game with a black dot.), to decide which two players can be chosen to play together. In this interaction graph, the number of vertex is equal to the number of players, and each vertex uniquely represents a particular player in the game. We randomly choose an edge in the interaction graph at each round of the game, and two players who are connected by this edge are the ones who play the Prisoner’s Dilemma game at that round. Any two players who don’t have an edge linked in between will never play together. On the other hand, we also make sure that each vertex has at least one edge linked to it, so that every player can get his chances to play. The last requirement to the interaction graph is that any two players can only be linked by no more than one edge.
As pointed out in previous research [14][15], no matter what shape the interaction graph has, the game system will finally reach the mutual-cooperation status, as long as it is a connected graph. We say the system is “converged” when it reaches the mutual-cooperation status, and the number of the rounds played until reaching the mutual-cooperation status is called the “convergence time”. Although we are simulating an infinite game, our program won’t run forever, and it will stop when the system is converged. Since we use randomness to choose the playing edges, the convergence time for any given game system obviously will be different from game to game. However, we should also expect that in reality the convergence time would fall into a certain range with high probability for a given game system. For example, if the game system consists of 4 players and the interaction graph is of type circle, then it’s very unlikely that the convergence time can reach a big number, say 1000, although it’s theoretically possible. Here, the number of players clearly has a significant impact on the convergence time. The more players we have, the more time is needed to converge. And we can also see that the shape of the interaction graph is another decisive factor to the convergence time. Suppose we changed the interaction graph of the above example to a complete graph, then there will be more edges involved in the play, and the result will probably be different. In fact,
Martin Dyer and his colleagues have already proved that the convergence time is polynomial in the number of players when the interaction graph is a cycle, and it is exponential in the number of players when the interaction graph is a complete graph [15]. However, by far nobody has found a way to prove the relationship between the convergence time and the shape of the interaction graph for other commonly used graph types, e.g. Tree, Grid, or some variations of these basic graph types.

The main purpose of this project is to use computer programs to construct all the above interaction graphs, and simulate the Iterated Prisoner’s Dilemma game with the Pavlov strategy on each of the above graph types. By analyzing the convergence time obtained from the experiments, we try to summarize the relationship between the convergence time and the shapes of the interaction graphs for some popular graph types, namely Cycle, Tree, Grid, Cycle with certain amount of extra edges and Complete Graph. Although the Cycle and Complete Graph situations have already been formally proved by Dyer, we would still like to run the experiments on these two graphs to check the tightness of his two theorems, based on which his conclusions were drawn. The cycle-with-extra-edges graph type is not a standard graph type, the reason why we want to add it into our experiments is that we want to check if the polynomial relationship to the convergence time is a unique characteristic belonging to Cycle, or it can also belong to some other graph structures with similar number of edges.

Besides the shape of the interaction graph, we will also try to analyze the impact on the convergence time from some other factors, which are:

1. The number of players who will cooperate in their first round. Since the Pavlov strategy needs the information of the last round to decide the action of the next round,
and it doesn’t give specific rules on how to play the first round, we want to test what kind of impact the number of initiative cooperative players has on the convergence time.

2. Noise. According to our game rules, players should abide by the Pavlov strategy when they take actions. If a player made a decision which was against the Pavlov strategy, we consider that move was a mistake. Noise refers to the possibility of players making mistakes at each action-taking moment. If, for example, the “noise” parameter is 1%, then every player has 1% chance to make mistake every time he’s chosen to play. As we know, people make mistakes in the real world. Therefore, in order for a game strategy to be successful in reality, it must have the ability to tolerate a certain amount of mistakes, and should not let these mistakes slow down the converging speed significantly. We will test how well the above game model works in a noisy environment.

We will test and analyze the impact of these two factors on all the interaction graphs mentioned above.

Although the conclusions drawn in this project come from experiments instead of theoretical proof, our intention of this project is trying to point out a direction to the future formal proof.

2.2 Simulation Plan and Game Environment

2.2.1 Simulation Plan

In order to test the impact on the convergence time from different factors introduced above, we will create a series of game environments. We illustrate these game environments by the following table.
Table 2.1: Game environment table

<table>
<thead>
<tr>
<th></th>
<th>Basic</th>
<th>Noise</th>
<th>ini_cooped</th>
<th>ex_edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Tree</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Grid</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Complete Graph</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
</tr>
</tbody>
</table>

In this table, the rows represent the graph types. Among all the standard graph types discussed in this project, the Cycle and the Complete Graph both have a unified form, which means we can define a cycle or a complete graph simply by its size. On the other hand, the Grid and the Tree have various forms. For example, a tree could be a binary-tree, ternary-tree, or 2-3 tree etc; and a grid could be a 2-row grid, 3-row grid, or 4-row grid etc. Obviously, all these different forms will result in different convergence time, since they have different edges. It's not possible for us to test on all these innumerable forms. Therefore, we will only test on some typical forms of the Tree type and the Grid type interaction graphs. Specifically, we will test on the binary, ternary, 4-ary trees and 5-ary trees (for some cases) for the Tree type interaction graphs, and the 2-row, 3-row, 4-row, 5-row, and square grids for the Grid type interaction graphs.

The columns in the above table represent the game conditions. Here is an explanation to what each column stands for:
1. Basic: the games will proceed in a noise-free environment, and everybody will
   defect in his first round. In addition, the interaction graph will be a regular graph with no
   extra edges.

2. Noise: the games will proceed in the same environment as the “Basic”
   condition, except for the “Noise” parameter: at each round of the plays, players have a
   chance to take the opposite action of the Pavlov strategy. As stated above, this “Noise”
   factor is to simulate the “people making mistakes” situation in the real world. In our
   research, we assume that every player has the same probability to make mistakes, and this
   probability represents the strength of the noise in the game environment. For example, if
   we set the “Noise” parameter to 2% in our computer program, then each of both players
   who are picked to play at each round has 2% of chances to play the opposite move to the
   Pavlov strategy.

3. ini_cooped: the games will proceed in the same environment as the “Basic”
   condition, except for the “the number of initially cooperative players” parameter. This
   parameter is used to check whether the players’ first moves will have a significant impact
   on the final convergence time, and we use a percentage of the total number of players to
   represent the number of players who will choose to cooperate in their first rounds (of
   course, this percentage can’t be 100%, otherwise we already have a converged system
   before the game starts). For example, if the “ini_cooped” parameter is equal to 40%, then
   40% of the total players will cooperate in their first rounds. We choose these initially
   cooperative players at random when we construct the interaction graph.

4. ex_edges: the games will proceed in the same environment as the “Basic”
   condition, except that the interaction graph in the experiment is not a standard type of
graph (e.g. cycle, tree, grid), but a standard type of graph with some extra edges (of course, we can’t add any extra edges to a complete graph). We use a percentage of the number of original edges to represent the number of extra edges. For example, if the “ex_edges” is equal to 30%, and the original graph has 100 edges, then we will add in 30 more edges to the original graph. The method of choosing the corresponding vertices for these extra edges is again by randomness on one condition: the pair of vertices which are chosen to attach an extra edge should have no edge linked in between already. This condition is to prevent the occurrence of the situation where a pair of vertices are linked by more than one edge.

Every cell in Table 2.1 is a game environment, however we will only do experiments on those marked “Yes”. Although our program supports the “ex_edges” tests on any graph types except for Complete Graph, we will omit this test on the Grid type and the Tree type of interaction graphs. The reason of omission is that according to our experimental results (see 2.3.1.3, 2.3.1.4), the convergence time for Grid or Tree is exponential in the number of players, except for a few cases (i.e. Binary tree, 2-row grid and 3-row grid). Since the purpose of this project is just to find out the magnitude of the convergence time, and the exponential relationship is already the slowest converging speed we can possibly get. By adding more edges to the interaction graph, the converging speed will not be any faster. Therefore, we can skip these tests knowing that the result would still be exponential if we had done these tests.

In order to draw the convincing conclusions, we need to test on the interaction graphs with size (i.e. vertices / number of players) ranging as widely as possible. Therefore, for each of the above game environments, our plan is to start with a small-
sized interaction graph, and obtain the convergence time for this graph from the average value out of 100 repetitive games. After that, we increase the size of the graph by a fixed amount, and do the same experiments on the new graph. We keep enlarging the interaction graph and testing the new graph, until the interaction graph becomes too big to converge in reasonable time. By choosing the starting size and the increment carefully, we shall have enough data to do a convincing analysis when we reach such a hard-to-converge size. Since the converging speed is different from graph type to graph type, the actual value of starting size, increment and finishing size also vary from graph type to graph type. However, we will always obtain the convergence time of a given-sized interaction graph by averaging the convergence time of 100 games. We decided to use 100 as the repeating times, because we observed in our tests that it is big enough to represent the typical convergence time, while also small enough to finish the experiments efficiently under our computer hardware condition.

Besides the “starting size”, “increment”, and the “finishing size”, we also need to set the appropriate value for the “noise”, “ini_cooped” and the “number of extra edges” parameters when we do experiments on the corresponding game environments.

For the “noise” parameter, we decided to test on the 1% and 2% two cases, so that we can analyze the impact of noise by comparing the convergence time in noise free, 1% noise and 2% noise three cases. If the convergence time is not significantly different in these three cases, then we can safely claim that the Pavlov is a good strategy to operate in a noisy environment, because 2% is already a huge number for noise parameter comparing to 0.1% set in other Pavlov and Iterated Prisoner’s Dilemma research [11].
As for the “ini_snoped” parameter, we tested the 50% case for all graph types mentioned above, moreover, we especially tested various values from 50% along the way until 95% for the complete graph only to check the tightness of Dyer’s theorem 2.

As stated above, we will do the “ex_edges” test on Cycle-typed interaction graphs only. We will run experiments for 1%, 2%, 10%, and 100% four cases respectively. We won’t increase the value further, since we consider that if the number of extra edges is more then the number of original edges, the new graph should hardly be considered a Cycle-based graph any more.

After we have done all the experiments described above, we will use charts with the size of interaction graph on the x-axis and the convergence time on the y-axis to compare the difference of the convergence time under various game conditions, and furthermore conclude impact from different factors. Since data curves in most charts are exponential, in order for all of the data points to be seen clearly, we use logarithmic scaling for both x-axis and y-axis in all of our data charts.

2.2.2 Programming Language

The simulation program was written in C# (see appendix for code). We use the “Random” class in C# to generate the necessary random numbers. Although it is not possible to acquire the genuine random numbers in a deterministic machine, the pseudo-random numbers generated by this class are good enough for the practical purposes [16]. In order to achieve the best randomness, we use only one random number generator to create all of numbers that need to have random relation with each other [17].
2.2.3 Regression Analysis

We need a mechanism to judge the functions of the data curves in the experimental charts, so that we can learn the relationship (e.g., polynomial, exponential etc.) between the size of the interaction graph and the convergence time from these functions. We use the “trendline” feature in the Microsoft Excel to accomplish this task: After we have plotted a chart, we use the “trendline” to apply all types of trend lines respectively, namely Linear, Logarithmic, Polynomial (order 2), Power, and Exponential, and the one whose “R-squared value [18]” is closest to 1 will be chosen to be the trend line of the data curve. Moreover, the magnitude of this trend line function will be considered the magnitude of the data curve, meaning that we believe that the data curve is polynomial if the trend line function is polynomial, or the data curve is exponential if the trend line function is exponential ... etc. One big advantage of using the “trendline” function as our data regression analysis tool is that research shows that the trend line function obtained by the “trendline” can actually be used for the prediction of future data, if the R-squared value is at least 0.98 [18]. As we will see later, in most of our experimental results, the R-squared values are above 0.98.

2.3 Experimental Results

In this section, we will present our research results. We found that the impact of the “int. coopered” factor is consistent for all types of interaction graphs, so we will dedicate one sub-section (2.3.1.6) to discussing this issue. Other test results will be categorized by the types of interaction graphs involved in and put into the corresponding sub-sections (2.3.1.2 – 2.3.1.5). We will prove each conclusion by a series of charts showing the convergence time as a function of the graph size. In each chart, we will also
include the corresponding trend line, the function of the trend line and its R-squared value along with the data curve, so that readers can see how close the trend line fits the data curve.

In the following charts, “min size” denotes the size of the smallest interaction graph; “max size” denotes the size of the biggest interaction graph; “increment” denotes increment on the number of players between two adjacent testing graphs. The data curve in each chart consists of a series of dots, each of which represents the convergence time of an interaction graph with the size indicated by its x-axis value. Unless otherwise indicated in the titles of the charts, the values of the “noise”, “ini_cooped” and “ex_edges” three parameters are all 0’s in all the charts. The functions showed in the charts are the functions of the corresponding trend lines.

2.3.1 Data Analysis and Discussions

2.3.1.1 Data Stability

Before we look into the results of our experiments, we should first check the stability of data we gathered, which means to see how consistent the convergence time is over repetitive games under the same game condition. Our conclusions will be convincing only when supporting data is typical, rather than accidental. The convergence time we are referring to here, is again the average convergence time out of 100 games.

We found in our experiments that the stability of the convergence time really depends on the shape of the interaction graph. If, for example, the interaction graph is a 0-3 tree, which means that each node of this tree has a random number between 0 and 3 of children, then the convergence time could change dramatically from game to game. As
we can see from Figure 2.2, the convergence time could vary by as much as 5-6 times from game to game when the size of the interaction graphs reaches 1000. Moreover, Figure 2.2 also shows that instability of the convergence time grows while we increase the size of the interaction graph. The reason for this instability of the convergence time is the uncertainty of the shapes of the interaction graphs. Since any node in a 0-3 tree has a random number of children, the exact shape of the tree differs from game to game. Although this difference is minor, since they are all 0-3 trees, and the total numbers of nodes are the same among all 100 games for each graph size, it turned out to cause significant inconsistency in the convergence time. This is why we decided to only experiment on trees which have a fixed number of children on each node for Tree type interaction graphs.

Figure 2.2: 0–3 tree, min size: 7, max size: 1978, increment: 1

\[ y = 0.1967x^{2.2152} \]
\[ R^2 = 0.9777 \]

1 In all the trend line functions in this project, "x" denotes the size of the interaction graph, and "y" denotes the convergence time.
Our experiments showed that the convergence time would be much more stable as long as the shape of the interaction graphs could remain the same over all 100 games for each graph size. The following charts are the comparison of two trials for some typical game environments. Every convergence time drawn in each chart is an average value out of 100 games. These comparison results showed below are typical, but not the best among all of the experiments that we have done.

Figure 2.3: Two trials comparison for case: Cycle, min size: 100, max size: 100000, increment: 100
Figure 2.4: Two trials comparison for case: Binary tree, min size: 7, max size: 4095, increment: 2

Figure 2.5: Two trials comparison for case: 2-row grid, min size: 4, max size: 134, increment: 2
2.3.1.2 Observations About Cycle

In this section, we're going to discuss our observations on the convergence time when the interaction graph is of type Cycle.


Martin Dyer's Theorem 1 says: Let G be a cycle on n vertices and let $\varepsilon > 0$ be given. With probability at least $1 - \varepsilon$, the Pavlov process reaches the absorbing state\(^2\) in

\[
\frac{49}{2} \cdot n \log\left(\frac{49n}{94\varepsilon}\right) \text{ steps}^3.
\]

This theorem basically says that the convergence time should be less than or equal to $\frac{49}{2} \cdot n \log\left(\frac{49n}{94\varepsilon}\right)$ with the probability of $1 - \varepsilon$, if the interaction graph is a cycle. The

\(^2\) Absorbing is equal to convergence in this paper.
\(^3\) One step is equal to one round of play.
correctness of this theorem was verified by our experimental data. Moreover, our experimental results even showed that the actual convergence time would be much smaller than this upper bound. Let \( \varepsilon = 1 \), \( y = \frac{49}{2} n \log\left(\frac{49e}{94}\right) \), then according to the theorem, there is at least zero probability that the system will converge in \( y \) steps, which means that there’s no guarantee at all that the system will converge in \( y \) steps. However, our experiments (see Figure 2.7 for details of the game conditions) showed that 100% of all \( 100 \times 1000 = 10^7 \) games had finished within \( y \) steps, and the average convergence time was consistently around 40% of the upper bound. We show the comparison between the real average convergence times and the Dyer’s theoretical lowest upper bounds in Figure 2.8, where “\( n \)” denotes the size of the interaction graphs, and “\( \text{Avg} (x) \)” denotes the average convergence time for cycles with size \( x \).

Figure 2.7: Cycle, min size: 100, max size: 100000, increment: 100

\[
y = 11.114x^{1.1271}
\]

\[
R^2 = 0.9996
\]
Since the average convergence times for all of the tested cycles reached only at roughly 40% of the steps that according to Theorem 1, there is no guarantee at all to converge, we can see that Dyer’s Theorem 1 is loose.

Observation 2: Cycles with up to 100% extra edges still converged in polynomial time (see Table 2.2 for the comparison of trend line functions, and Figure 2.9 – 2.12 for experimental data).

Table 2.2: Trend line functions when interaction graph is Cycle with various extra edges

<table>
<thead>
<tr>
<th>Extra edges</th>
<th>0%</th>
<th>1%</th>
<th>2%</th>
<th>10%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trend line</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>function</td>
<td>$y = 1.114x^{1.277}$</td>
<td>$y = 10.234x^{1.139}$</td>
<td>$y = 10.151x^{1.198}$</td>
<td>$y = 10.03x^{1.232}$</td>
</tr>
<tr>
<td>Extra edges</td>
<td>100%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trend line</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>function</td>
<td>$y = 11.086x^{1.078}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 2.9: Cycle, min size: 100, max size: 53400, increment: 100, ex_edges: 1%

\[ y = 10.234x^{1.136} \]
\[ R^2 = 0.9996 \]

Figure 2.10: Cycle, min size: 100, max size: 57100, increment: 100, ex_edges: 2%

\[ y = 10.151x^{1.188} \]
\[ R^2 = 0.9995 \]
Figure 2.11: Cycle, min size: 100, max size: 57280, increment: 100, ex_edges: 10%

![Graph showing cycle time and size relationship with ex_edges at 10%](image)

\[ y = 10.103x^{1.372} \]
\[ R^2 = 0.9995 \]

Figure 2.12: Cycle, min size: 100, max size: 87200, increment: 100, ex_edges: 100%

![Graph showing cycle time and size relationship with ex_edges at 100%](image)

\[ y = 11.086x^{1.273} \]
\[ R^2 = 0.9996 \]
Observation 3: The convergence time became exponential to the size of the interaction graph when noise was no less than 0.1% (see Table 2.3 for the comparison of trend line functions and Figure 2.7, 2.13 and 2.14 for experimental data).

Table 2.3: Trend line functions when interaction graph is Cycle, under various noise conditions

<table>
<thead>
<tr>
<th>Noise</th>
<th>0%</th>
<th>1%</th>
<th>2%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trend line function</td>
<td>$y = 1.114x^{1.271}$</td>
<td>$y = 2660.5e^{0.003t}$</td>
<td>$y = 1224.8e^{0.002t}$</td>
</tr>
</tbody>
</table>

Figure 2.13: Cycle, min size: 100, max size: 4100, increment: 100, noise: 0.1%
2.3.1.3 Observations About Tree

Observation 1: According to our data, the convergence time for the binary trees was polynomial to the size of the interaction graph. However, there is a high possibility that the relationship will eventually change to exponential as the graph size grows. This suspicion is originated by the observation that the last few data points in the data curve has already started to deviate from the trend line and go up in a way which looks very like exponential (see Figure 2.15). Unfortunately, due to the restriction of our computer hardware condition, we could not test larger graphs and collect enough data to prove our suspicion.
Observation 2: Except for binary, the convergence time for any n-ary tree was exponential to its size. We have tested Ternary, 4-ary, and 5-ary tree in our experiments, and all of them converged in exponential time to their sizes. Moreover, when we were comparing the trend line functions of these cases, we found that the degrees of their exponential functions went up along with the increase of the degrees of trees (i.e. $0.1615 > 0.1006 > 0.0427$ in Table 2.4). This means that a tree with higher degree is more difficult to converge than a tree with lower degree. Since the 5-ary tree is already exponential, we can reasonably summarize that any n-ary tree with degree larger than 5 will also converge in exponential time to its size (see Table 2.4 for trend line comparison, and Figure 2.16 - 2.18 for experimental data).
### Table 2.4: Trend line functions when interaction graph is Tree, in the basic game environment

<table>
<thead>
<tr>
<th>Tree type</th>
<th>Binary</th>
<th>Ternary</th>
<th>4-ary</th>
<th>5-ary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trend line function</td>
<td>$y = 1.1135x^{0.7946}$</td>
<td>$y = 544.66x^{0.9273}$</td>
<td>$y = 212.92x^{0.8064}$</td>
<td>$y = 114.29x^{0.3015}$</td>
</tr>
</tbody>
</table>

### Figure 2.16: Ternary tree, min size: 7, max size: 262, increment: 3

- Trend line function: $y = 544.66x^{0.9273}$
- $R^2 = 0.9803$

### Figure 2.17: 4-ary tree, min size: 5, max size: 157, increment: 4

- Trend line function: $y = 212.92x^{0.8064}$
- $R^2 = 0.9896$
Observation 3: The higher degree a tree has, the more difficult for it to converge. As stated in Observation 2, we have only experimented trees with degrees up to 5 to support this conclusion, however there is no reason for us to suspect that trees with any higher degree will violate this rule (see Table 2.4 for trend line comparison, and Figure 2.16 – 2.18 for experimental data).

Observation 4: Noise had a noticeable impact on the convergence time of Tree type interaction graphs, especially when it was a binary tree. We found in our experiments that noise greater than or equal to 0.1% was big enough to turn the convergence time of the binary trees from polynomial to exponential (see the Table 2.5 for trend line comparison, and Figure 2.15 – 2.17 and 2.19 – 2.24 for experimental data).
Table 2.5: Trend lines when interaction graph is Tree, under various noise conditions

<table>
<thead>
<tr>
<th>Tree type</th>
<th>0% noise</th>
<th>0.1% noise</th>
<th>0.2% noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>( y = 1.135x^{0.916} )</td>
<td>( y = 1276.2e^{0.0132x} )</td>
<td>( y = 627.03e^{0.1121x} )</td>
</tr>
<tr>
<td>Ternary</td>
<td>( y = 544.66e^{0.0427x} )</td>
<td>( y = 324.47e^{0.0351x} )</td>
<td>( y = 213.54e^{0.026x} )</td>
</tr>
<tr>
<td>4-ary</td>
<td>( y = 114.29e^{0.106x} )</td>
<td>( y = 152.9e^{0.1127x} )</td>
<td>( y = 127.03e^{0.1129x} )</td>
</tr>
</tbody>
</table>

Figure 2.19: Binary tree, min size: 7, max size: 797, increment: 2, noise: 0.1%
Figure 2.20: Ternary tree, min size: 4, max size: 247, increment: 3, noise: 0.1%

![Ternary tree graph](image)

\[ y = 324.47e^{0.052x} \]

\[ R^2 = 0.9842 \]

Figure 2.21: 4-ary tree, min size: 5, max size: 145, increment: 4, noise: 0.1%

![4-ary tree graph](image)

\[ y = 152.9e^{0.1127x} \]

\[ R^2 = 0.9921 \]
Figure 2.22: Binary tree, min size: 7, max size: 545, increment: 2, noise: 0.2%  

\[ y = 627.03e^{0.015x} \]
\[ R^2 = 0.9873 \]

Figure 2.23: Ternary tree, min size: 4, max size: 211, increment: 3, noise: 0.2%  

\[ y = 213.51e^{0.061x} \]
\[ R^2 = 0.9869 \]
2.3.1.4 Observations About Grid

In this project, a grid is defined by its size and the number of its rows. For example, if the size of a 2-row grid is 20, then this grid is a 2×10 grid. Since a 2×10 grid is isomorphic to a 10×2 grid, we assume that the number of rows never exceeds the number of columns in all grids in this project.

Observation 1: The 2-row grid converged in polynomial time to its size. The 3-row grid looked like a polynomial time model, however like the Binary tree case, the last few data points went beyond the trend line, and had the potential to change the relationship to exponential. Like the Tree case, by observing the trend lines of all grid types we have tested up to 5-row, we can reasonably conclude that for the grids with row number greater than 3, it takes exponential time for them to converge (see the Table 2.6 for trend line comparison, and Figure 2.25 – 2.29 for experimental data).
Observation 2: Like the tree case, the convergence time not only depended on the size of the graph, but also depended on the shape of the grid. When the size is fixed, the smaller the difference between the number of rows and the number of columns, the harder for the graph to converge. A square, in which case the number of rows and the number of columns are equal, was most difficult to converge. In our experiments, an 11×11 square (size 121), took more time to converge than a 5×30 grid (size 150) (see the Table 2.6 for trend line comparison, and Figure 2.25 – 2.29 for experimental data).

<table>
<thead>
<tr>
<th>Grid type</th>
<th>2-row</th>
<th>3-row</th>
<th>4-row</th>
<th>5-row</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trend line function</td>
<td>(y = 1.631x^{0.963})</td>
<td>(y = 0.0918x^{-0.234})</td>
<td>(y = 326.08e^{0.072x})</td>
<td>(y = 284.76e^{0.189x})</td>
</tr>
<tr>
<td>Grid type</td>
<td>Square</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trend line function</td>
<td>(y = 33.432e^{1.29x})</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.6: Trend line functions when interaction graph is Grid, in the basic game environment
An x-row grid is a grid with x rows. Since an x times y grid is isomorphic to a y times x grid, we assume that the row number never exceeds the column number.
Figure 2.27: 4-row grid, min size: 16, max size: 148, increment: 4

\[ y = 326.08e^{0.0736x} \]
\[ R^2 = 0.9885 \]

Figure 2.28: 5-row grid, min size: 25, max size: 150, increment: 5

\[ y = 284.76e^{0.0895x} \]
\[ R^2 = 0.9963 \]
Observation 3: The convergence time noticeably increased when we raised noise. This was the case for all grid types we had tested, namely, 2-row grid, 3-row grid, 4-row grid, 5-row grid and square. However, the 2-row grid was still able to hold up its polynomial relationship when we raised noise up to 0.2%, and the 3-row grid was able to hold up its polynomial relationship when noise was no greater than 0.1%. For the rest types of grids, since they were already in exponential relationship even when noise was zero, the increase of noise would just increase the degrees of their exponential functions (see the Table 2.7 for trend line comparison, and Figure 2.25 – 2.39 for experimental data).

\[ y = 33.432e^{0.129x} \]

\[ R^2 = 0.9884 \]

\( n \) denotes the size of the current square.
Table 2.7: Grid trend line functions, under various noise conditions

<table>
<thead>
<tr>
<th>Grid type</th>
<th>normal</th>
<th>0.1% noise</th>
<th>0.2% noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-row</td>
<td>$y = 1.631e^{0.667}$</td>
<td>$y = 0.6573e^{2.665}$</td>
<td>$y = 0.4576e^{2.2738}$</td>
</tr>
<tr>
<td>3-row</td>
<td>$y = 0.0918x^{2.928}$</td>
<td>$y = 0.0304x^{3.330}$</td>
<td>$y = 0.1692e^{0.7724}$</td>
</tr>
<tr>
<td>4-row</td>
<td>$y = 326.08x^{0.5734}$</td>
<td>$y = 185.78x^{0.4994}$</td>
<td>$y = 151.67x^{0.3972}$</td>
</tr>
<tr>
<td>5-row</td>
<td>$y = 284.76x^{0.2895}$</td>
<td>$y = 162.74x^{0.1025}$</td>
<td>$y = 141.82x^{0.0491}$</td>
</tr>
<tr>
<td>squares</td>
<td>$y = 33.432x^{1.241}$</td>
<td>$y = 23.516x^{1.2464}$</td>
<td>$y = 23.769x^{1.3468}$</td>
</tr>
</tbody>
</table>

Figure 2.30: 2-row grid, min size: 4, max size: 120, increment: 2, noise: 0.1%
Figure 2.31: 3-row grid, min size: 9, max size: 120, increment: 3, noise: 0.1%

3-row grid with 0.1% noise

Trendline

\[ y = 0.0304x^{3.3299} \]

\[ R^2 = 0.9794 \]

Figure 2.32: 4-row grid, min size: 16, max size: 120, increment: 4, noise: 0.1%

4-row grid with 0.1% noise

Trendline

\[ y = 185.78e^{0.089x} \]

\[ R^2 = 0.992 \]
Figure 2.33: 5-row grid, min size: 25, max size: 120, increment: 5, noise: 0.1%

\[ y = 162.74e^{0.029x} \]

\[ R^2 = 0.9972 \]

Figure 2.34: Square, min size: 4, max size: 100, increment: \((\sqrt{n} + 1)^2 - n\), noise: 0.1%

\[ y = 23.516e^{0.1416x} \]

\[ R^2 = 0.977 \]

\(n\) denotes the size of the current square.

47
Figure 2.35: 2-row grid, min size: 4, max size: 126, increment: 2, noise: 0.2%

\[ y = 0.4576x^{2.2019} \]
\[ R^2 = 0.9945 \]

Figure 2.36: 3-row grid, min size: 9, max size: 126, increment: 3, noise: 0.2%

\[ y = 169.29e^{0.0738x} \]
\[ R^2 = 0.9842 \]
Figure 2.37: 4-row grid, min size: 16, max size: 124, increment: 4, noise: 0.2%

![4-row grid graph with trendline and equation: $y = 151.67e^{0.0972x}$, $R^2 = 0.9952$.]

Figure 2.38: 5-row grid, min size: 25, max size: 125, increment: 5, noise: 0.2%

![5-row grid graph with trendline and equation: $y = 141.82e^{0.1303x}$, $R^2 = 0.9981$.]
2.3.1.5 Observations About Complete Graph

Observation 1: Dyer’s condition for a complete graph to converge in exponential
time to its size is higher than necessary.

Martin Dyer’s Theorem 2: Let $T_e$ denote the absorption time$^4$ of Pavlov process
on the graph $K_n$ starting from a configuration$^5$ $X_0$ with fewer than $\frac{3n}{5}$ defeaters$^6$. With
probability $1 - O(1)$ we have $T_e \geq (1.1)^n$.

This theorem means that if a complete graph starts the game with initially
cooperative players fewer than 60%, then there’s a constant chance that the system will

---

$^4$ "n" denotes the size of the current square.
$^5$ Absorption time is the same as convergence time.
$^6$ A “configuration” denotes a state of the interaction graph. It could be anything from the all-defect to the
all-cooperate.
$^7$ Plus here stands for playing cooperate at first move.
converge in exponential time. According to this theorem, 60% is the necessary condition for us to predict a probably exponential convergence. However, in our experiments, we found that even if we increased the number of initially cooperative players up to 95%, the convergence time was still exponential. This discovery might lead to a formal prove that the convergence time for a complete graph is exponential to its size, unless the initial state is already very close to the converging state (see Figure 2.40 for experimental results).

**Observation 2:** Noise had a noticeable impact on the convergence time, and increased the degrees of the exponential functions (see Table 2.8 for the comparison of trend line functions, and Figure 2.41 – 2.43 for experimental data).
Table 2.8: Complete Graph trend line functions, under various noise conditions

<table>
<thead>
<tr>
<th>type</th>
<th>normal</th>
<th>0.1% noise</th>
<th>0.2% noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete graph</td>
<td>$y = 5.8968e^{0.441x}$</td>
<td>$y = 5.4723e^{0.440x}$</td>
<td>$y = 4.9404e^{0.438x}$</td>
</tr>
</tbody>
</table>

Figure 2.41: Complete graph, min size: 4, max size: 49, increment: 1

$$y = 5.8968e^{0.441x}$$

$R^2 = 0.9977$
Figure 2.42: Complete graph, min size: 4, max size: 41, increment: 1, noise: 0.1%

\[ y = 5.4723e^{0.4482x} \]
\[ R^2 = 0.999 \]

Figure 2.43: Complete graph, min size: 4, max size: 40, increment: 1, noise: 0.2%

\[ y = 4.9404e^{0.4506x} \]
\[ R^2 = 0.9987 \]
2.3.1.6 Impact of the number of initially cooperative players

We have proceeded two experiments to discover the impact of the number of initially cooperative players on the convergence time. The first one was specifically designed to check the tightness of Dyer's theorem 2, and has already been explained in section 2.3.1.5. The second experiment was designed to check the impact of the number of initially cooperative players in a more general way, and we have conducted this experiment on all types of the interaction graphs discussed in this project. In this experiment, the purpose was to test the case in which all players make random choices for their first moves. The scenario how people make their first moves could be any case from all-defect to all-cooperate. The reason why we are particularly interested in this "random start" case is based on the observation that in the real life applications, people usually will make random choices for their first moves, since it's human's intuition to make random choices when one doesn't have any experience / preference in either option. If every player in the system makes random choice as his first move, then it will become the "random start" case in the systematic point of view. Since the scenario is a common scene in reality, our research on it will be more meaningful than research on any other first-move scenario.

In our game simulation program, this random start scenario can be simulated by setting the "ini_cooped" parameter to 50% in the game model. To evaluate the impact of the number of the initially cooperative players, we compare the convergence time of the random start scenario to the convergence time of the all-defect scenario, and see if the converging speed will improve if we initially set the system to half-converged.
The experimental result showed that the random start scenario did not converge undisputedly faster than the all-defect scenario. In fact, except for a few cases (Ternary tree, and 3-row grid), the random start scenario actually converged slower than the all-defect scenario. However, we also noticed that the random start scenario didn’t slow down the converging speed much either in all the cases we have tested. Therefore, it seemed that the number of initially cooperative players was a minor factor to the convergence time (see Table 2.9 for the comparison of trend line functions, and Figure 2.7, 2.15 – 2.17, 2.25 – 2.29, 2.41 and 2.44 – 2.53 for experimental data).

Table 2.9: “50% ini_cooped” vs. “all-defect” trend line function comparison

<table>
<thead>
<tr>
<th>Graph type</th>
<th>All-defect</th>
<th>50% ini_cooped</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle</td>
<td>$y = 11.114x^{1.271}$</td>
<td>$y = 9.3383x^{1.453}$</td>
</tr>
<tr>
<td>Binary tree</td>
<td>$y = 1.1135x^{1.784}$</td>
<td>$y = 0.4311x^{0.911}$</td>
</tr>
<tr>
<td>Ternary tree</td>
<td>$y = 544.66x^{0.5427}$</td>
<td>$y = 556.8x^{0.4412}$</td>
</tr>
<tr>
<td>4-ary tree</td>
<td>$y = 212.92x^{0.9398}$</td>
<td>$y = 190.52x^{0.3126}$</td>
</tr>
<tr>
<td>2-row grid</td>
<td>$y = 1.631x^{0.905}$</td>
<td>$y = 1.1705x^{0.905}$</td>
</tr>
<tr>
<td>3-row grid</td>
<td>$y = 0.0918x^{2.328}$</td>
<td>$y = 0.1349x^{2.415}$</td>
</tr>
<tr>
<td>4-row grid</td>
<td>$y = 326.08x^{0.6738}$</td>
<td>$y = 267.33x^{0.7788}$</td>
</tr>
</tbody>
</table>

55
Figure 2.44: Cycle, min size: 100, max size: 37500, increment: 100, ini_cooped: 50%
Figure 2.45: Binary tree, min size: 7, max size: 3267, increment: 2, ini_cooped: 50%

![Binary tree graph](image)

**Equation:** $y = 0.431x^{1.911} 
R^2 = 0.988$

Figure 2.46: Ternary tree, min size: 4, max size: 307, increment: 3, ini_cooped: 50%

![Ternary tree graph](image)

**Equation:** $y = 556.8e^{0.011x} 
R^2 = 0.9736$
Figure 2.47: 4-ary tree, min size: 5, max size: 145, increment: 4, ini_cooped: 50%

Figure 2.48: 2-row grid, min size: 4, max size: 124, increment: 2, ini_cooped: 50%

\[ y = 190.52 e^{1.025x} \]

\[ R^2 = 0.9877 \]

\[ y = 1.1705x^{1.8979} \]

\[ R^2 = 0.9919 \]
Figure 2.49: 3-row grid, min size: 9, max size: 123, increment: 3, ini_cooped: 50%

![Graph showing trendline and exponential equation.]

\[ y = 0.1349x^{0.8339} \]

\[ R^2 = 0.9939 \]

Figure 2.50: 4-row grid, min size: 16, max size: 124, increment: 4, ini_cooped: 50%

![Graph showing trendline and exponential equation.]

\[ y = 267.33e^{0.0708x} \]

\[ R^2 = 0.9851 \]

59
Figure 2.51: 5-row grid, min size: 25, max size: 125, increment: 5, ini_cooped: 50%

\[ y = 204.23e^{0.0943x} \]
\[ R^2 = 0.9967 \]

Figure 2.52: Square, min size: 4, max size: 121, increment: \((\sqrt{n} + 1)^2 \cdot n\), ini_cooped: 50%

\[ y = 29.171e^{0.3318x} \]
\[ R^2 = 0.9826 \]
Figure 2.53: Complete graph, min size: 4, max size: 41, increment: 1, ini_cooped: 50%
3 SUMMARY AND FUTURE WORKS

3.1 Summary of Experimental Results

In this project, we have setup an Iterated Prisoner’s Dilemma game model by computer programs, and simulated the impact on the convergence time from various interesting factors, which were the shape of the interaction graph, noise, the number of initially cooperative players, and the number of extra edges (for Cycle only).

From our experimental results, we can see that among all these factors, the one that influences the convergence time the most is the shape of the interaction graph. In other words, when the number of players is fixed, the convergence time of a given system is mostly decided by the number of edges in the interaction graph. For example, given the same amount of players, a complete graph will converge much slower than a tree, while a tree will converge much slower than a cycle with high probability.

Noise has a clear impact on the convergence time. A noisier environment usually takes longer to converge. However, as we discussed, the Pavlov strategy has a strong ability to recover from a mistake. Our experimental data also verified this point: the delay caused by noise was trivial in all testing cases.

The factor of the number of initially cooperative players has an unclear impact on the convergence time in our experiments. For some types of the interaction graphs, adding more initially cooperative players slowed down the convergence progress; while for other types of the interaction graphs, it speeded up the convergence progress. However, the change on the convergence time is minor in all cases.
By testing the convergence time of the cycle with extra edges, we proved that the polynomial relationship between the interaction graph size and the convergence time is not an exclusive character of Cycle. Any interaction graphs with similar complexity (or similar number of edges) to Cycle will have polynomial convergence time.

We have also tested Martin Dyer’s two theorems respectively. For his theorem 1, we found that his upper bound of the convergence time of Cycle was much higher than the actual results we obtained from our simulations. For his theorem 2, we found that in order for Complete Graph to converge in exponential time, the highest percentage of the initially cooperative players could be allowed much higher than 60% – the upper bound stated in theorem 2.

3.2 Possible Future Works

The first future work could be to formally prove the relationship between the convergence time and the number of initially cooperative players. Our experiments showed that the random start scenario doesn’t converge faster than the all-defect scenario for most of the graph shapes. One intuitive explanation to this scenario is that any graph can achieve the 50% convergence rate very quickly. The time consuming part is to converge the rest 50%. If this is true, then we can even claim that in a complete graph, it is the last 5% players who consume most of the convergence time, since our experiment has shown that complete graphs with 95% initially cooperative players take almost the same time to converge as the ones with 0% initially cooperative players (see Figure 2.40). Another possible explanation is that how the convergence rate changes depends on how the current cooperative players are distributed in the system. Because the way how we
chose these initially cooperative players was totally by randomness, this probably caused these initially cooperative players to be evenly spread in the graph, and therefore generated considerable pairs of “cooperate-defect” players, instead of “cooperate-cooperate” players. Since the number of cooperative players will decrease by one every time such a “cooperate-defect” pair is chosen to play, the existence of high volume of these “cooperate-defect” pairs will actually decrease the converging speed, instead of increase it, since the system needs to first convert these “cooperate-defect” pairs to “defect-defect” pairs, and then start from the “all-defect” situation.

In order to verify the correctness of the first conjecture, one approach is to design another simulation program which is capable of keeping track of the convergence rate and time used to reach such rate at the same time, so that we can have a clear view of how the whole convergence time is spent during the whole converging process. If the first 50% of players does converge in trivial time in the experiments, then our first conjecture is correct; otherwise, it’s wrong. To verify the correctness of the second conjecture, we can design another computer simulation program which allows users to designate the number of the “cooperative-defect” player pairs when the game starts. By doing experiments through this simulation program, we can check the relationship between the convergence time and the number of “cooperate-defect” pairs.

The second future work could be to give a formal proof to tighten up Martin Dyer’s two theorems.

The third possible work could be to give formal proofs on the convergence time for the interaction graphs of type Grid, Tree and other popular graph shapes, the same work as what Martin Dyer has done for Cycle and Complete graph in his two theorems.
APPENDICES

C# source code:

```csharp
using System;
using System.Collections;
using System.Collections.Generic;

namespace Project
{
    /// <summary>
    /// Main class of the project
    /// </summary>
    public class MainClass
    {
        // Static field of the project
        static string projectName = "ExampleProject";

        // Static field of the project
        static int numPlayers = 5;

        // Static method of the project
        static void Main()
        {
            // Call the function to process the graph
            ProcessGraph(graphType);
        }
    }
}
```

65
Bong max = 0; // the max round number of all test runs in each graph
Bong min = 0; // the min round number of all test runs in each graph
Reader r = new BrokenFileInputStream();
Graph g = null; // testing graph

String fileName = "Grafique " + min_Players + " " + max_Players + " " increment + " " testing_times + " repeat_Times " + startTimeMonth + " startTime Year + " startTime Hour + " startTime Min + " startTime Second + " int ";

try
{
    StringWriter sw = new StringWriter(fileName);
    // begin testing
    for (Int k = min_Players; k < max_Players; k += increment) :
    // display progress on screen
        Console.WriteLine("Final graph size is: " + max_Players + " " increment + " " currently at size: " + k);
    startTime = DateTime.Now;
    // test 200 possible players in row is the square root of the number of players
    int start = Tester.TopLevel.Start(k);
    int end = Tester.TopLevel.End(k);
    for (Int i = 1; i <= start; i++)
    {
        if (k == k / i + 1)
        {
            // record the number of players on file
            sw.WriteLine(" " + players + " " + " rows " + k / i + " columns ");
            for (int j = 0; j < repeat_Times; j++)
            {
                g = new Grille(k, x / i);
                if (extra_Edges > 0) // user requested an irregular graph
                    CustomizeGraph(g, extra_Edges * g.getEdges().Length / 100);
            }
        }
    }
}
if (startTime.Year <= 2000) // user requested some players' initial
    to be cooperative
    InitializeGraph(g, (Int)Convert.ToInt32(g.numberOfPlayers.Length / 100), r);

planGame(g, r, noise);
temp = g.getRound(entered);
totalRounds = temp;
testResult[i] = temp;
sw.WriteLine(temp + " ");

// calculate average value
avg = totalRounds / repeat_Times;
sw.WriteLine("avg = " + avg + " ");

// calculate standard deviation
for (Int p = 0; p < testResult.Length; p++)
    stddev = Math.Pow(testResult[p] - avg, 2);

    stddev = Math.Sqrt(stddev / repeat_Times);
    sw.WriteLine("standard deviation = " + stddev + " ");

// find the Max and Min value
max = Math.Max(testResults[0], testResult[1])
min = Math.Min(testResults[0], testResult[1])

66
for (int i = 2; i < testResults.length; i++)
    max = Math.max(max, testResults[i]);
    min = Math.min(min, testResults[i]);

sw.WriteLine("\tmax = ");
sw.WriteLine(max);
sw.WriteLine("\tmin = ");
sw.WriteLine(min);

// Reset all counters
totalRounds = 0;
testResults = new long[repeatTimes];
temp = 0;
avg = 0;
stdDev = 0;
max = 0;
min = 0;
sw.WriteLine();

// Display progress on screen
DateTime startTime = DateTime.Now;
Console.WriteLine("Timer started" - (DateTime - startTimer));
sw.WriteLine();
sw.WriteLine();
sw.Close();

catch (Exception e)
{
    Console.WriteLine("Exception popped up" + e.GetType().Name);
    Environment.Exit(1);
}

// Summary
// Test a non-grid graph

public static void TestNonGrid()
{
    DateTime stopTime = DateTime.Now;
    long[] testResults = new long[repeatTimes];
    // Store the result of each run being tested
    long totalRounds = 0;
    // Store the total number of all tests run in all repeatTimes tests, need to calculate the "mean" value for each type of cycle
    long temp = 0;
    double avg = 0;
    // Store the average results for each type of cycle
    double stdDev = 0;
    // Store the standard deviation
    long max = 0;
    // The max round number of all test runs in each graph
    long min = 0;
    // The min round number of all test runs in each graph
    Random r = new Random(int) 
    Stop g = null;
    // Testing graph.

    // Set dimensions
    if (gridType == 1)
    { graphName = "Circle"; 
    else if (gridType == 2)
    { graphName = "Tree"; 
    else if (gridType == 3)
    { Console.WriteLine("This method can’t be used to test Grids");

76
return;

else

graphName = "Complete Graph":
string filename = graphName + " time " + minPlayers + " maxPlayers " + increment + " testing times " + repeatTimes + " intGraph" + intGraphNode + " noise " + K + " extra edges " + extraEdges;
if (graphType = 2)
fileName = "K degree " + minDegrees + " maxDegrees " + time;
startTime.Year = " " + startTime.Month + " " + startTime.Day + " " + startTime.Hour + " " + startTime.Minute + " " + startTime.Second + " Ext";
else
fileName = " time " + startTime.Year + " " + startTime.Month + " " + startTime.Day + " " + startTime.Hour + " " + startTime.Minute + " " + startTime.Second + " Ext";
try
    StreamWriter sw = new StreamWriter(fileName);
    //Begin timer:
    for (int k = minPlayers; k < maxPlayers; k = increment) :
        //display program on screen
        Console.WriteLine("Final graph size is: " + maxPlayers + ", increment: " + increment + ", currently at size: " + k);
        startTime = DateTime.Now;
        sw.WriteLine( " players: ");
        //out repeat Times times for each graph
        for (int j = 0; j < repeatTimes; j++) |
            if (graphType = 1)
                g = new Circle(g);
            else if (graphType = 2)
                g = new Triangle(g, minLength, minLength);
            else
                g = new CompleteGraph(g);
            if (extraEdges > 0) // user requested on irregular graph
                extraEdges = g.addEdge().Length / 100, y;
            if (intGraphNode > 0) // user requested some players' initial location to be computed
                intGraphNodes = g.getRandomNodes().Length / 100,
                p;
            playGame(g, r, noise); temp = g.getNodes().Length;
            testResults[j] = temp;
            sw.WriteLine( " ");

    //calculate avg avg = totalResults / repeatTimes;
    sw.WriteLine(" avg:");
    //calculate the standard deviation
    for (int r = 0; r < testResults.Length; r++)
        stdDev = Math.Pow(testResults[r] - avg, 2);
        stdDev = Math.Sqrt(stdDev / repeatTimes);
    sw.WriteLine("standard deviation: " + stdDev);

    //Find out the Max and Min value

78
max = Math.Max(testResults[0], testResults[1]);
min = Math.Min(testResults[0], testResults[1]);
int i = 2;i < testResults.Length; i++)
max = Math.Max(max, testResults[i]);
min = Math.Min(min, testResults[i]);

sw.WriteLine("Max: ", max);
sw.WriteLine("Min: ", min);
// Display progress on screen
time = time + 500;
Console.WriteLine("Time spent: ", time + start line);
// reset all counters
totalCounts = 0;
totalRounds = 0;
testResults = new long[repeat_times];
temp = 0;
avg = 0;
stdDev = 0;
max = 0;
min = 0;
sw.WriteLine(
sw.Flush();
sw.Close();
)
}
catch (Exception e) {
Console.WriteLine("Exception detected: ", e.GetType().Name);
Environment.Exit(1);
}

} // (Summary)
// CONSTRUCT the graph using Prim's algorithm
} // (Summary)
// (Query name = "") (Query)
} // (Query name = "") (Query)
// (Query name = "") (Query)
// (Query name = "") (Query)
public static void playGame(graph g, Random r, double noise) {
if (done) return;
// (Query)
// g.getComponents(Customer) is not used
while (true)
int p = g.getUnvisited() * g.getUnvisited().Length; // pick a random number between 0, g.getUnvisited().Length
// get the current status of the players attached to the picked edge
int p1 = g.getUnvisited().getEdge(p, 0).getPlayed();
int p2 = g.getUnvisited().getEdge(p, 0).getPlayed();
// get the new status
int newEdge = p1 + p2;
// add in noise
if (r.NextDouble() < noise) {
// update the g.getComponents(Customer)
if (p1 < p2) g.getComponents(Customer)(i++) = 1;
else if (done = true) g.getComponents(Customer)(i++) = 1;
} else {
if (p1 < p2) g.getComponents(Customer)(i++) = 1;
else if (done = true) g.getComponents(Customer)(i++) = 1;
}
newState = newState + (-1);

else {
  // update the cooperatePlayers counter
  if ((parity == positive && posParity == -1) || (parity == negative && posParity == 1)) {
    g.cooperatePlayers_COUNTER = g.cooperatePlayers_COUNTER + 1;
  } else if (newState == -1) {
    g.cooperatePlayers_COUNTER = g.cooperatePlayers_COUNTER - 1;
  }
}

// update the status of the players
if (g.getState(newState)) {
  g.setEdges(g.getActiveEdges());
}

// check if converged
if (g.getCooperatePlayers_COUNTER() == g.getNumPlayers().Length) {
  converged = true;
  // double-check the correctness of convergence, can be errors when passes the debug phase
  for (int i = 0; i < g.getNumPlayers().Length; i++) {
    if (g.getPlayers()[i].getState() == 1) {
      g.increaseRoundCounter(); // Increase iteration counter
    }
  }
}

}<comment>
  set ini_gos players' initial status to "cooperate", and update the cooperated counter
</comment>
</comment>

public static void initializeGraph(Graph g, int ini_gos, Random r) {
  int temp = 0;
  ArrayList<Character> newArrayList();
  for (int i = 0; i < g.getNumPlayers().Length; i++) {
    temp = (int) (r.nextDouble() * ArrayList.size());
    g.setPlayers()[i].setState(1);
    g.increaseRoundPlayersCounter();
    newArrayList.add();
  }
}

}</comment>
</comment>
</comment>
</comment>

// comment: add extra edges into the given basic graph, and return (// comments)
public static Graph customizeGraph(Graph baseGraph, int i, boolean r) {
    // safety check
    if (i < baseGraph.getPlayerCount() || baseGraph.getPlayerCount() % 2 == 1) {
        return baseGraph;
    }
    // copy the edges from the base graph to the new graph
    Graph newGraph = new Graph(baseGraph.getPlayerCount(), baseGraph.getNumberOfEdges()); // copy the edges from the base graph to the new graph
    newGraph.copyFrom(baseGraph, baseGraph.getPlayerCount());
    // Link 1 extra edge to random chosen i pair of players
    int k = 1;
    int done = false;
    // randomly choose two players in the base graph to link with each other
    while (!done) {
        int i = (int) (Math.random() * baseGraph.getPlayerCount());
        int j = (int) (Math.random() * baseGraph.getPlayerCount());
        // condition: h is the edge (i,j) doesn't exist and edge (j,i) doesn't exist
        if (baseGraph.getEdge(i, j) == null) {
            LinkedList<Node> e = new LinkedList<Node>()
            done = true;
            foreach (Edge e in e) {
                if (i == getPlayer(e).getId() && j == getPlayer(e).getId()) {
                    done = false;
                    break;
                }
            }
            newGraph.addPlayerToPlayer(i, j, new Edge(i, j, k));
        } else {
            newGraph.addPlayerToPlayer(newGraph.getPlayer(i), newGraph.getPlayer(j), new Graph.newEdge(i, j, k));
        }
    }
    return newGraph;
}

// request basic testing parameters from user
public static void main(String[] args) {
bool done = false;
// ask user for graph type
while (!done) {
    Console.WriteLine("Please choose the graph type, and press enter:
1. Tree
2. Grid
3. Complete Graph.");
    graphType = Convert.ToInt32(Console.ReadLine());
    if (graphType > 3 || graphType < 0)
        done = true;
    else
        Console.WriteLine("Please enter an integer between [1,4].");
}

done = false;
// ask for parameter minPlayers
while (!done) {
    Console.WriteLine("Please enter an integer to specify the minimum size of the
testing graphs (at least 4): ");
    minPlayers = Convert.ToInt32(Console.ReadLine());
    if (minPlayers > 4)
        done = true;
    else
        Console.WriteLine("The size must be \> 4.");
}

done = false;
// ask for parameter maxPlayers
while (!done) {
    Console.WriteLine("Please enter an integer \( \leq \) : maxPlayers : "
    to specify the maximum size of the testing graphs.");
    maxPlayers = Convert.ToInt32(Console.ReadLine());
    if (minPlayers > maxPlayers)
        done = true;
    else
        Console.WriteLine("The size must be \( \geq \) : minPlayers.");
}

if (maxPlayers < minPlayers)
    increment = 0;
else 
    done = false;
// ask for parameter increment
while (!done) {
    Console.WriteLine("Please enter an integer between [0, "
    increment = Convert.ToInt32(Console.ReadLine());
    if (increment < maxPlayers - minPlayers && increment > 0)
        done = true;
    else
        Console.WriteLine("The increment must be between [0, "
    increment = 0;
}

done = false;
// ask for parameter repeat_Times
while (done)
{
    Console.WriteLine("Please enter an integer to specify the testing times for each graph (e.g., the converting times for each site of graph). ":)
    repeat_times = getInt();
    if (repeat_times < 0)
        done = true;
    else
        Console.WriteLine("The repeat times must be > 0.");

    done = false;
    // ask for parameter init_plus
    while (done)
    {
        Console.WriteLine("Please enter an integer: 0, 100) to indicate how many players should take cooperate move in their first move in percentage of the whole group. For instance, enter 20 if you want 20% of players should cooperate in 2222 move.");
        init_plus = getInt();
        if (init_plus < 0 || init_plus < 0)
            done = true;
        else
            Console.WriteLine("The number must be between 0,100.");
    }

    done = false;
    // ask for parameter noise
    while (done)
    {
        Console.WriteLine("Please enter a double number [0, 100) to indicate the ratio of making a mistake in each move. For instance: enter 0.1 for 10% of probabilities a mistake will be made in each move.");
        noise = getDouble();
        if (noise > 0 && noise < 100)
            done = true;
        else
            Console.WriteLine("The number must be between 0,100.");
    }

    done = false;
    // ask for parameter extra_edges
    while (done)
    {
        Console.WriteLine("If you need an irregular graph, please enter a double number > 0 to indicate the percentage of extra edges over the regular ones. For example: enter 20 if you need 20% extra edges; and 0 if you want a regular graph.");
        extra_edges = getDouble();
        if (extra_edges > 0)
            done = true;
        else
            Console.WriteLine("The number must be > 0.");
    }

    if (graphType == 2) // if user requested a tree
        done = false;
    // ask for parameter min_degrees
    while (done)
    {
        Console.WriteLine("Please enter an integer > 0 to indicate the minimum degrees of the tree.");
    
}
min_Degrees = getMin();
if (min_Degrees >= 0 && min_Degrees < min_Players)
done = true;
else
Console.WriteLine("The number must be between 0, " + min_Players + ", ");
done = false;
// ask for parameter max_Degrees.
while (!done)
Console.WriteLine("Please enter an integer (" + Math.Max(1, min_Degrees) + ", " + (min_Players - 1) + ") to indicate the maximum degrees of the tree.");
max_Degrees = getInt();
if ((max_Degrees > min_Degrees && max_Degrees <= min_Players) || (max_Degrees == min_Degrees && max_Degrees != 0))
done = true;
else
Console.WriteLine("The number must be between ");
// ask user for a double number:
public static double getDouble(string message)
{ double done = false;
double returnvalue = 0; // return value
while (!done)
try
{ returnvalue = Convert.ToDouble(Console.ReadLine());
done = true;
}
catch (Exception e)
{ if (message == "FormatException")
Console.WriteLine("Please enter a double number.");
return returnvalue;
}
// ask user for an integer:
public static int getInt()
{ int done = false;
int returnvalue = 0; // return value
while (!done)
try
{ returnvalue = Convert.ToInt32(Console.ReadLine());
done = true;
}
catch (Exception e)
{ if (message == "FormatException")
Console.WriteLine("Please enter an integer.");
return returnvalue;
}
```csharp
// Player.cs

using System;
using System.Collections.Generic;

namespace BoardGame
{
    public class Player
    {
        // This class represents one player of the game.
        // which is also be considered as a vertex of the graph.
        public int id;
        public string name;
        public string color;
        public List<int> moves;
        public List<int> edges;
        public List<int> nextMoves;
        public List<int> prevMoves;
        public List<int> incidentEdges;
        public int state;

        public Player(int id, string name, string color, List<int> moves, List<int> edges, List<int> nextMoves, List<int> prevMoves, int state)
        {
            this.id = id;
            this.name = name;
            this.color = color;
            this.moves = moves;
            this.edges = edges;
            this.nextMoves = nextMoves;
            this.prevMoves = prevMoves;
            this.state = state;
        }

        public void SetState(string state)
        {
            if (state == "H" || state == "L")
            {
                this.state = Convert.ToInt32(state);
                this.incidentEdges = incidentEdges;
            }
        }

        public void PrintPlayer()
        {
            Console.WriteLine("Player ID: "+this.id);
            Console.WriteLine("Player Name: "+this.name);
            Console.WriteLine("Player Color: "+this.color);
            Console.WriteLine("Player Moves: "+this.moves);
            Console.WriteLine("Player Edges: "+this.edges);
            Console.WriteLine("Player Next Moves: "+this.nextMoves);
            Console.WriteLine("Player Prev Moves: "+this.prevMoves);
        }
    }
}
```
```java
public int getSite0()
{
    return state;
}

public int getSite1()
{
    return id;
}

public void setEdges(List<Edge> e)
{
    this.incidentEdges = e;
}

public List<Edge> getEdges()
{
    return incidentEdges;
}

public void addEdge(Edge e)
{
    incidentEdges.Add(e);
}

/// "Edge.cs"
using System;
using System.Collections.Generic;
using System.Text;

namespace Player {
    public class Edge {
        private Player player1;
        private Player player2;

        public Edge(Player player1, Player player2) {
            this.player1 = player1;
            this.player2 = player2;
        }

        public void setPlayer1(Player player1) {
            this.player1 = player1;
        }

        public void setPlayer2(Player player2) {
            this.player2 = player2;
        }

        public Player getPlayer1()
        {
            return player1;
        }
    }
}
```

public class Graph {
    protected Player[] players;
    protected Edge[] edges;
    protected long round_counter;
    protected int coop_players_counter;

    public void setPlayers(Player[] players) {
        this.players = players;
    }

    public void setEdges(Edge[] edges) {
        this.edges = edges;
    }

    public void setRound_COUNTER(long round_COUNTER) {
        this.round_COUNTER = round_COUNTER;
    }

    public void setCoop_Players_COUNTER(int coop_Players_COUNTER) {
        this.coop_Players_COUNTER = coop_Players_COUNTER;
    }

    public Player[] getPlayers() {
        return players;
    }

    public Edge[] getEdges() {
        return edges;
    }

    public long getRound_COUNTER() {
        return round_COUNTER;
    }

    public int getCoop_Players_COUNTER() {
        return coop_Players_COUNTER;
    }

    // Other methods...
}
public void increaseRoundCounter()
{
    this.roundCounter++;
}

public void increase_coopPlayersCounter()
{
    this.coopPlayersCounter++;
}

public void decrease_coopPlayersCounter()
{
    this.coopPlayersCounter--;
}

using System;
using System.Collections.Generic;
using System.Collections;

namespace RockPaperScissors
{
    public class Grid : Graph
    {
        // This function constructs a Grid with specified rows, columns, and initial specified player numbers.
        public Grid(int numPlayersRow, int numPlayersCol)
        {
            this.numPlayersRow = numPlayersRow;
            this.numPlayersCol = numPlayersCol;
            this.RoundCounter = 0;
            this.decrease_coopPlayersCounter();
        }

        private void drawGrid(int width, int length)
        {
            for (int i = 0; i < length; i++)
            {
                Console.WriteLine("\" /n");
            }
        }
    }
}

78
// Compose the grid by the indices of players and edges
for (int i = 0; i < length; i++) {
    for (int j = 0; j < width; j++) {
        // Index of the current player
        int curIndex = i * width + j;
        // Not the last row
        if (i < length - 1) {
            // Not the last column
            if (j < width - 1) {
                // Link the player to the right one
                linkRight(curIndex, width, i, j);
                // Link the player to his below one
                linkBelow(curIndex, width, i, j);
            } else {
                // Not the last row, but in the last column
                // In this case, only link to the left player
                linkLeft(curIndex, width, i, j);
            }
        } else {
            // In the last row, but not the last column
            // In this case, only link to the right player
            if (j < width - 1) {
                linkRight(curIndex, width, i, j);
            } else {
                // The player in last column and row doesn’t have any outgoing link
            }
        }
    }
}

// Current
// Link the specified edge and player, assume the parameters are NOT NULL
private void linkEdge(e, Player p1, Player p2) {
    p1.addEdge(e);
    p2.addEdge(e);
    e.setPlayer1(p1);
    e.setPlayer2(p2);
}

// Current
// Link the current player to his right one, assume the current player is NOT NULL, and
// the right one is also NOT NULL
private void linkCurrentRight(Player curPlayer) {
    // Index of edge (taking the current player and the one to his right
    int curRightIndex = (curIndex + 2 * width - 1) % width;
}
// Instantiate the edge between the current player and the one to the right
eges[curRightIdx] = new Edge();
// Link the current player to his right one
linkEdges[curRightIdx], players[curIdx], players[curIdx + 1];

// <summary>
// Link the current player to the one below him, assume the current player is NOT NULL.
// hint: the one below IS NULL
// </summary>
// (param: curIdx):the index of the current player
// (param: width):the number of players in each row
// (param: row):"the current row index
// (param: col):"the current column index
private void LinkBelow(int curIdx, int width, int row, int col) {
    // Instantiate the edge below the current one
    players[curIdx + width] = new Player(curIdx + row * width, -1);
    // Index of edge linking the current player and the one to his right
    int curRightIdx = row * width + 1;
    // Index of edge linking the current player and the one below him
    int curBelowIdx = row * (width + 1);
    // Instantiate the edge between the current player and the one below him
    edges[curBelowIdx] = new Edge();
    // Link the current player to his below one
    linkEdges[curBelowIdx], players[curIdx], players[curIdx + width];
}

namespace Fucow
public class CompleteGraph : Graph {

    // <summary>
    // Construct a complete graph, preset the status of initCars player as "cooperate"
    // </summary>
    // (param: numPlayers):the number of players in the graph
    // (param: initCars):the number of initialized "cooperate" players
    public static completeGraph() {
        this.setPlayers(numPlayers); // initCars
        this.setEdges(numPlayers * (numPlayers - 1) / 2); // initCars
        this.setRound(0);
        this.setPlayers(Counter(0));
        drawCompleteGraph();
    }
}

80
public class Tree {
    public Tree(int maxPlayers, int minPlayers, int maxDegrees, int minDegrees) {
        this.maxPlayers = maxPlayers;
        this.minPlayers = minPlayers;
        this.maxDegrees = maxDegrees;
        this.minDegrees = minDegrees;
    }

    public void generateTree() {
        // Generate a random number of players
        Random rand = new Random();
        int numPlayers = rand.nextInt(maxPlayers) + 1;

        // Create a graph with the generated number of players
        Graph graph = new Graph(numPlayers);

        // Add edges between players
        for (int i = 0; i < numPlayers; i++) {
            for (int j = i + 1; j < numPlayers; j++) {
                graph.addEdge(i, j);
            }
        }
    }
}

// Tree class implementation
// less than the min degree. These numbers of players will be discarded.
while CanPlayers > 1 { lastPos >>= shift; }
// assign children for the next player in the tree who hasn't been assigned.
for max = maxChildren - 1; max >= min; max-- {
    if (lastPos == noChildren) {
        root = (max < min) ? max : min; // min, max
    } else { // assign children for a player who is not the last in the tree
        root = finalNode.NextDouble() ? max - min : 1; // min, max
    }
    nextChildren.AddChildren(root, lastPos); // insert "rank" players into the tree, as children of the node indexed
    nextChildren.AddChildren(root, lastPos); // nextChildren
    for (int i = 1; i <= rank; i++) {
        // add a new node to the tree
        newNodes.AddNode(lastPos, i, i);
        // generate an edge to link between new node and the node indexed
        nextEdges.AddEdge(newNodes[Player] + nextChildren[Node] + i, lastPos);
        // record this edge in new node
        nextNodes[lastPos].nextNodes = i, addEdge(newNodes[Player], lastPos, i, Count - 1); // record this edge in node indexed "nextChildren".
        nextNodes[lastPos].nextNodes = i, addEdge(newNodes[Player], lastPos, i, Count - 1); // update the smallest index of players who haven't been assigned children
        nextNodes[lastPos].nextNodes = i, addEdge(newNodes[Player], lastPos, i, Count - 1); // update the index of the player who is the last inserted into tree.
        lastPos = root;
    }
}
// copy "nextPlayers" to "this.players[]".
this.players = new Player[nextPlayers.Count];
for (int i = 0; i < nextPlayers.Count; i++) { players[i] = nextPlayers[i];
// copy "nextEdges" to "this.edges[]".
this.edges = new Edge[nextEdges.Count];
for (int i = 0; i < nextEdges.Count; i++) { edges[i] = nextEdges[i];

// System
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;

82
```java
package Paraiso;

public class Cycle : Graph {
    public Cycle(int numPlayers) {
        this.setPlayers(new Player[numPlayers]);
        this.setEdges(new Edge[numPlayers]);
        this.setRoundCounter(0);
        this.setOpponentsCounter(0);
        drawCycle();
    }

    private void drawCycle() {
        // Initialize players
        for (int i = 0; i < this.players.length; i++) {
            this.players[i] = new Player(i, 1);
        }
        // Initialize edges
        for (int i = 0; i < this.edges.length; i++) {
            this.edges[i] = new Edge(this.players[i], this.players[i + 1]);
            this.edges[i].addEdge(edges[i]);
            this.edges[i].addEdge(edges[i + 1]);
        }
    }
}
```
REFERENCE LIST

[8] Robert Boyd, Jeffrey P. Lorberbaum, No Pure Strategy is Evolutionarily Stable in the Repeated Prisoner’s Dilemma Game