PROGRAMMING II PROJECT

SOCIAL GAME SYSTEMS
(A SIMULATION ON REPEATED PRISONER'S DILEMNA)

JOAN JOSEPH THOMAS (2106939)
FATHIMA SHANA CHIRATHODI (2027546)
MIAH MARIAM AKTER (2107051)

JULY 2024

TABLE OF CONTENTS

INTRODUCTION

Project Background:

The Prisoner's Dilemma is a fundamental problem in game theory that demonstrates why two rational individuals might not cooperate, even if it is in their best interest to do so. In the context of repeated interactions, the dynamics can shift significantly, providing a rich ground for studying strategic decision-making. This project simulates a population of players employing various strategies to repeatedly play the Prisoner's Dilemma game repeatedly, offering insights into the conditions under which cooperation or defection dominates. The simulation sets up a set number of players randomly assigned the given strategies, and they are pit against each other for a number of interactions and then their playoff score is taken to calculate the relative fitness and based on the relative fitness score of each strategy their reproduction probabilities are calculated.

The Repeated Prisoner's Dilemma Simulation project aims to explore the dynamics of strategic decision-making among players in a controlled environment. This simulation models various strategies such as Always Cooperate, Always Defect, Tit for Tat, Grim Trigger, and others, to study their interactions and outcomes over multiple iterations of the Prisoner's Dilemma game. The successful strategies over one generation will see their prevalence grow while the unsuccessful strategies will see themselves decline. By analyzing the emergent behaviors and equilibrium states, this project provides insights into the effectiveness and stability of different strategies in repeated interactions. The simulation is implemented in Java using JavaFX for the user interface, allowing real-time visualization of strategy performance and population dynamics.

Objectives:

The primary objectives of this simulation are:

- To model and simulate various strategies in the Repeated Prisoner's Dilemma.
- To analyze the outcomes and effectiveness of these strategies over multiple iterations.
- To understand the conditions leading to equilibrium states and the prevalence of certain strategies.
- To provide a user-friendly interface for visualizing the simulation in real-time.

LITERATURE OVERVIEW

The study of game theory, and particularly the Repeated Prisoner's Dilemma, has been a cornerstone of understanding strategic interactions in various fields. One of the most influential figures in this area is Robert Axelrod, whose experiments in the early 1980s provided profound insights into the dynamics of cooperation.

Axelrod's Experiments

Robert Axelrod conducted a series of computer tournaments to investigate which strategies would perform best in the Repeated Prisoner's Dilemma. He invited game theorists to submit strategies that his computer would then pit against each other in a round-robin format. The winning strategy from these tournaments was Tit for Tat, a simple yet effective strategy that cooperates on the first move and then mimics the opponent's previous move. Axelrod's findings highlighted several key principles for promoting cooperation:

1. **Clarity:** Strategies should be easy to understand.
2. **Niceness:** Strategies should begin with cooperation and retaliate only if provoked.
3. **Forgiveness:** Strategies should be willing to return to cooperation if the opponent does.
4. **Retaliation:** Strategies should not be too forgiving to avoid exploitation.

These principles from Axelrod's experiments demonstrated that even in competitive environments, cooperation could emerge and stabilize if these simple rules were followed.

Importance of Studying Game Theory and Repeated Prisoner's Dilemma

Game theory, and specifically the Repeated Prisoner's Dilemma, is crucial for understanding strategic interactions in various domains such as economics, politics, and social sciences. The Repeated Prisoner's Dilemma models situations where individuals or entities must decide between cooperation and defection over multiple interactions. This scenario is highly relevant to many real-world problems, including:

- **Economic Transactions:** Businesses and consumers repeatedly interact in markets, and trust is essential for long-term cooperation and mutually beneficial exchanges.
- **International Relations:** Countries face decisions on whether to cooperate or compete in areas such as trade, environmental agreements, and military alliances.
- **Social Behavior:** Individuals in communities regularly choose between selfish behavior and cooperative behavior, influencing the overall social harmony and collective well-being.

This project builds upon these foundational studies by implementing a range of strategies and allowing for their interaction in a dynamic, controlled simulation. By extending the analysis to multiple iterations and incorporating real-time visualizations, this project aims to provide deeper insights into the strategic behaviors and their outcomes.

IMPLEMENTATION

Technology Stack

The simulation is implemented using Java with **JavaFX** for the graphical user interface. **Maven** is used for project management and dependency handling. The simulation uses **threads** for various tasks for running it parallelly. Specifically, it also uses **executor service** framework extensively to manage the pool of threads and run the tasks in parallel and make the simulation very efficient.

Project Structure

```
StrategySimulation/
│
├── src/
│   └── main/
│       ├── java/
│       │   └── com/
│       │       └── example/
│       │           └── strategysimulation/
│       │               ├── Main.java
│       │               ├── models/
│       │               │   ├── Player.java
│       │               │   └── Interaction.java
│       │               ├── strategies/
│       │               │   ├── Strategy.java
│       │               │   ├── AlwaysCooperate.java
│       │               │   ├── AlwaysDefect.java
│       │               │   ├── TitForTat.java
│       │               │   ├── GrimTrigger.java
│       │               │   ├── RandomStrategy.java
│       │               │   ├── Pavlov.java
│       │               │   ├── TitForTwoTats.java
│       │               │   ├── GenerousTitForTat.java
│       │               │   ├── Prober.java
│       │               │   └── TitForTatWithRandomForgiveness.java
│       │               ├── utils/
│       │               │   └── Strategies.java
│       │               └── Simulation.java
│       └── resources/
│           └── application.css
├── pom.xml
└── README.md
```

Key Classes:

- **Main:** The main class of the program, it handles entry point of the simulation and all the UI elements and sets up the main scene of the simulation, it uses JavaFX.
- **Player:** The main class representing a player in the simulation. Each player has a strategy and can interact with other players.
- **Strategies:** An abstract class defining the general behavior of strategies. Specific strategies like Always Cooperate, Always Defect, Tit for Tat, and others extend this class.
- **Simulation:** Manages the overall simulation, including initializing players, running interactions, and updating the user interface.
- **Interaction:** Represents a single round of the Prisoner's Dilemma between two players, determining the outcomes based on their strategies.

Inputs

The simulation allows users to adjust various parameters through the user interface, including:

- Number of players (value range 50 - 1000)
- Distribution of strategies (value range 50 - 1000)
- Number of Interactions (value range 50 - 1000)

Users can modify these parameters to explore different scenarios and observe how the outcomes change. The default values are set to provide a balanced starting point at 50 for the simulation.

Execution Flow

1. **Starting the Application:**
   - When the application is launched, the **main method** in the Main class is executed.
   - This calls the launch(args) method, which starts the JavaFX application and calls the start method.

2. **Setting Up the UI:**
   - The **start method** in the Main class sets up the UI components, including the line chart, sliders, buttons, and legend box.
   - It displays the main window with these components.

3. **Starting the Simulation:**
   - When the user clicks the "Start" button, the **startSimulation** method is called.
   - This method reads the values from the sliders and initializes a new Simulation instance with these values.
   - It starts the simulation in a separate thread by calling **simulation.runSimulation().**

4. **Running the Simulation:**
   - The **runSimulation** method in the Simulation class sets simulationRuning to true and starts the executor.
   - It run a loop for the specified number of rounds.
   - In each round, it performs interactions between players by calling the interact method.
   - It calculates the fitness of each player by calling the **calculateFitness** method.
   - It reproduces new players based on fitness by calling the **reproduce** method.
   - It tracks the strategy prevalence for the round by calling the **trackStrategyPrevalence** method.
   -

5. **Interactions:**

   - The **interact** method creates tasks for interactions between players.
   - Each task randomly selects two players, gets their decisions, and calculates their payoffs.

- It adds the interaction to both players' histories.
- The tasks are executed in parallel using the executor.

6. **Fitness Calculation:**
   - The **calculateFitness** method calculates the total payoff for all players.
   - It calculates the fitness of each player by dividing their payoff by the total payoff.

7. **Reproduction:**
   - The **reproduce** method creates tasks for reproducing new players based on fitness.
   - Each task selects a parent player proportional to their fitness.
   - creates a new player with the parent's strategy.
   - The tasks are executed in parallel using the executor.
   - The old players are replaced with the new players.

8. **Updating the UI:**
   - The **updateUI** method in the Main class continuously updates the chart with the latest strategy prevalence data from the simulation.
   - It runs in a separate thread and periodically updates the UI.

9. **Stopping the Simulation**
   - When the simulation finishes, the simulationRunning flag is set to false and the executor is shut down.
   - If the user clicks the "Reset" button, the **resetSimulation** method is called.
   - This stops the simulation if it is running and clears the chart and legend.

Fitness Determination and Reproduction Mechanism

Fitness Calculation

1. **Interaction Payoffs:**
   - Each player i participates in multiple interactions. Let p_{ij} represent the payoff player i receives from interaction j.
   - The total payoff for player i, P_i, is calculated as:

$$P_i = \sum_{j=1}^{n_i} p_{ij}$$

where n_i is the number of interactions player i participates in.

2. **Total Population Payoff:**
   - The total payoff for the entire population, P_{total}, is the sum of the total payoffs of all players:

$$P_{\text{total}} = \sum_{i=1}^{N} P_i$$

where N is the number of players in the population.

3. **Fitness Assignment:**
   - The fitness f_i of player i is the ratio of their total payoff to the total population payoff:

$$f_i = \frac{P_i}{P_{\text{total}}}$$

   - This normalizes the fitness values so that the sum of all fitness values equals 1:

$$\sum_{i=1}^{N} f_i = 1$$

Reproduction Process

1. **Total Fitness:**
   - The total fitness of the population, F_{total}, is the sum of all individual fitness values:

$$F_{\text{total}} = \sum_{i=1}^{N} f_i$$

2. **Fitness Proportional Selection:**
   - To select a parent for a new player, a random value r is generated uniformly from the range $[0, F_{\text{total}})$.
   - Iterate through the players, accumulating their fitness values until the cumulative fitness exceeds r.

Mathematical Steps for Reproduction

1. **Calculate total fitness F_{total}:**

$$F_{\text{total}} = \sum_{i=1}^{N} f_i$$

2. **Select a parent for each new player:**
   - For each new player:
     1. Generate a random value r such that $0 \leq r < F_{\text{total}}$.
     2. Initialize a cumulative fitness C to 0.
     3. Iterate through the players j:

$$C = C + f_j$$

     4. When $C \geq r$, select player j as the parent.
     5. The new player inherits the strategy of the selected parent j.

1**. Fitness Calculation:**

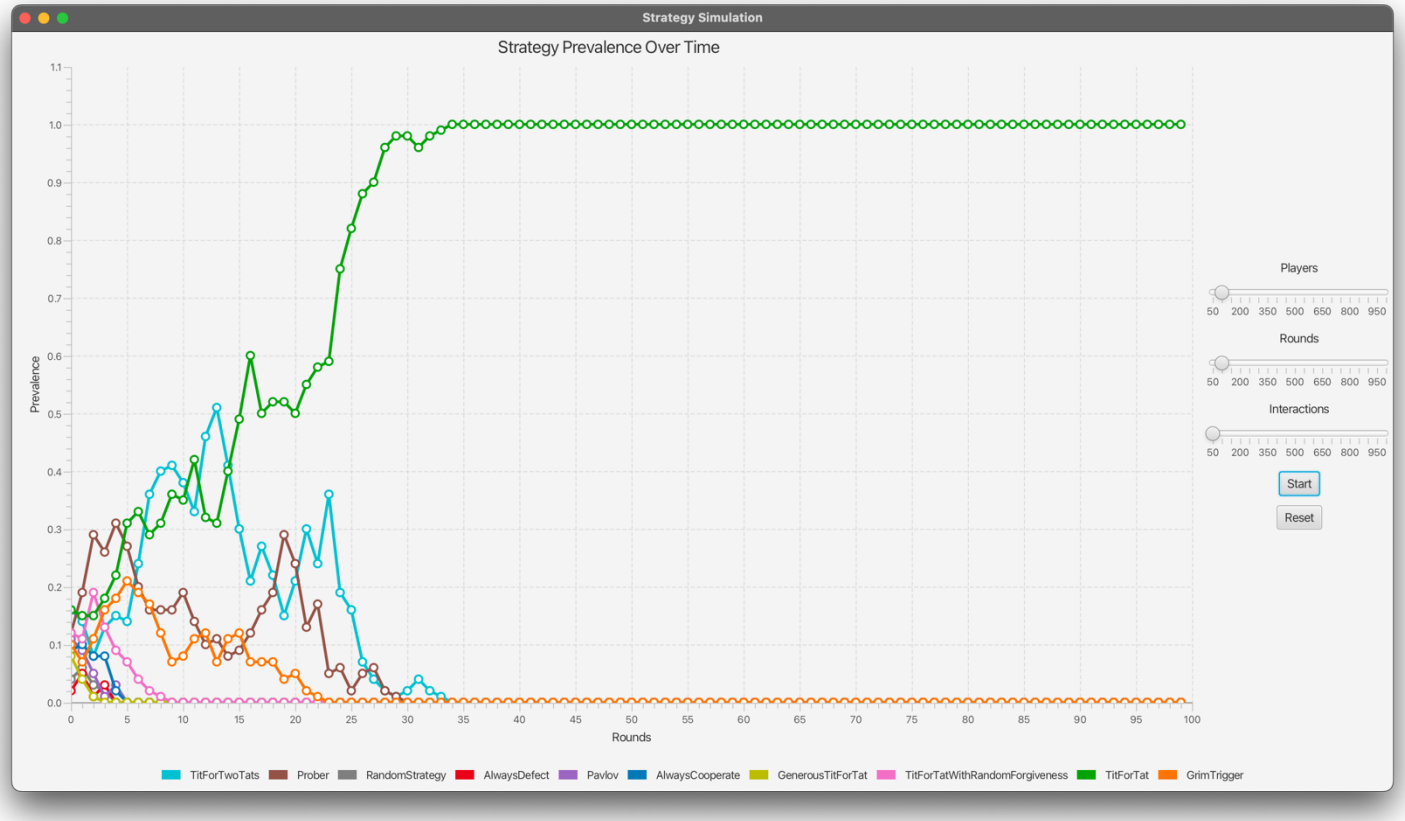
- Interaction Payoffs: Each player engages in multiple interactions per round. The outcomes of these interactions are stored in the Interaction objects.
- Total Payoff Calculation: For each player, the total payoff is calculated by summing the payoffs from all their interactions.
- Fitness Assignment: The fitness of each player is determined by dividing their total payoff by the sum of the total payoffs of all players. This normalizes the fitness values so that they are proportional to the relative success of each player in the population.

2. **Reproduction:**

- Fitness Proportional Selection: New players are created based on the fitness of existing players. The probability of an existing player being chosen as a parent is proportional to their fitness.
- New Generation Creation: For each new player, a parent is selected using the fitness proportional selection method. The new player inherits the strategy of the selected parent.
- Population Update: The entire population is replaced by the new generation of players.

This process simulates natural selection, where, Fitness Calculation, Players' success in interactions (analogous to an organism's ability to survive and reproduce) determines their fitness. Reproduction: Players with higher fitness are more likely to pass on their strategies to the next generation, simulating the concept of "survival of the fittest

RESULTS & CONCLUSION



Results

The simulation was run with various strategies competing over multiple rounds to observe their prevalence over time. The strategies included Always Cooperate, Always Defect, Tit for Tat, Grim Trigger, Random Strategy, Pavlov, Generous Tit for Tat, Prober, Tit for Two Tats, and Tit for Tat with Random Forgiveness.

Note - The result is a representation of a single simulation run over an average ran of simulations. During several simulation run, a variety of different strategies showed final prevalence including unforgiving strategies like grim trigger and prober etc. but on average the forgiving strategies like Tit for Tat, Tit for Two Tats, Generous Tit for Tats show more prevalence while purely nasty strategies like Always Defect always go extinct after just a few rounds.

From the simulation graph, several observations can be made:

1. **Initial Strategy Prevalence**: In the early rounds, a diverse set of strategies show varied prevalence. No single strategy dominates initially, indicating a competitive environment where multiple strategies can coexist.

2. **Rise of Tit for Tat and Tit for Two Tats**: Around the 15th round, the Tit for Tat strategy begins to rise in prevalence. Grim Trigger also shows significant prevalence, demonstrating the effectiveness of reciprocal and punitive strategies in maintaining cooperation and punishing defection.

3. **Decline of Always Defect and Other Strategies**: Strategies like Always Defect and Random Strategy show initial fluctuations but eventually decline to near zero prevalence. This indicates that purely selfish strategies or those lacking a clear pattern of reciprocity are less successful in the long term.

4. **Dominance of Tit for Tat**: By the 30th round, Tit for Tat becomes the dominant strategy, maintaining a prevalence close to 1.0. This suggests that a strategy based on reciprocity and forgiveness (to a limited extent) is highly successful in evolutionary terms within the parameters of this simulation.

5. **Extinction of Other Strategies**: By the 40th round, other strategies such as Pavlov, Prober, and Tit for Two Tats, although initially competitive, are driven to extinction. This reinforces the dominance of Tit for Tat in a stable environment where its principles of cooperation and retaliation are optimal.

Conclusion

The results of the simulation clearly demonstrate the evolutionary advantage of the Tit for Tat strategy in the context of the repeated Prisoner's Dilemma. Tit for Tat's success can be attributed to its simple yet effective mechanism of cooperation, retaliation, and forgiveness, making it robust against both cooperative and defective strategies.

Key Takeaways:

Cooperative Strategies Prevail: Strategies that incorporate cooperation, such as Tit for Tat and Generous Tit for Tat etc., tend to be more successful over the long term compared to purely selfish strategies like Always Defect.

Importance of Reciprocity: The success of Tit for Tat highlights the importance of reciprocity in maintaining cooperation within a population. This is consistent with Axelrod's findings and emphasizes the relevance of reciprocal strategies in both biological and social contexts.

Evolutionary Stability: The simulation underscores the concept of evolutionary stability, where strategies that foster mutual benefit and deter exploitation tend to dominate the population over time.

These findings not only validate classical theories of game theory and evolutionary biology but also provide insights into modern problems of cooperation and competition, from economics and politics to social behavior and evolutionary dynamics. By understanding the principles that govern successful strategies, we can better design systems and policies that promote cooperative behavior and mitigate conflict.