|  |
| --- |
|  |
| ipd-sim: A Simulation Framework for Prisoner’s Dilemma Scenarios  CSC400 |
| |  |  |  | | --- | --- | --- | | Jessica Boe | 12/1/18 | Dr. Antonios | |

Contents

[Introduction 2](#_Toc518254999)

[Architecture 3](#_Toc518255000)

[Type of System 3](#_Toc518255001)

[Platform 3](#_Toc518255002)

[Inputs and Outputs 3](#_Toc518255003)

[User Interface 7](#_Toc518255004)

[Use Cases 8](#_Toc518255005)

[Sequence Diagrams 9](#_Toc518255006)

[Structural Design 10](#_Toc518255007)

[Class Diagrams 10](#_Toc518255008)

[Data 11](#_Toc518255009)

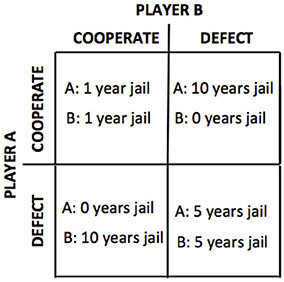
[User Interface Design 12](#_Toc518255010)

# Introduction

## The Prisoner’s Dilemma: Background

The prisoner’s dilemma is widely regarded as one of the foundations in the area of game theory. With applications in diverse areas from environmental protectionism, to economics, to even foreign relations and defense, the prisoner’s dilemma is an interesting case study in human interactions across a host of disciplines.

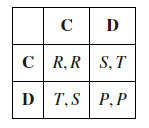
In a basic, single game prisoner’s dilemma, there are two participants with a simple, four square matrix that allows each player two options: to cooperate with the other player, or to defect. This was originally set up in the context of an actual ‘prisoner’s dilemma’, where the two participants are criminals who have had no prior interaction with one another, with the following payoff matrix:



From this example, we can see that the least risky move to take would be for each player to cooperate. However, if a player is selfish, they would probably be more likely to defect under the assumption that their opponent will cooperate, so that they get off with no time spent in jail while their opponent gets the maximum sentence.

Additionally from this matrix, we can view what is referred to as the Nash Equilibrium, or the solution to a game in which each player has nothing to gain by changing their strategy. In this case, the Nash Equilibrium is for each player to betray one another, because even if mutual cooperation leads to a better outcome, if one player chooses to cooperate and the other doesn’t the outcome for the other player is worse.

From the classic prisoner’s dilemma game, we can derive the matrix into a simpler form to apply it to a broader spectrum of areas. The new matrix is:



Here, the parameters have taken variable forms, with R pertaining to the reward payoff, formerly -1, T pertaining to the temptation payoff, formerly 0, S pertaining to the sucker payoff, or -10 before, and P pertaining to the punishment payoff, formerly -5. These parameters can be tweaked to suit different models and purposes, and the parameters will have a drastic effect on the way that a player attempts the game.

The final thing to keep in mind when experimenting with the prisoner’s dilemma is that players have unique strategies, which must be modeled when making simulations.

## The Iterated Prisoner’s Dilemma

While the single game prisoner’s dilemma is interesting,

Mining for cryptocurrencies such as bitcoin in a proof-of-work environment is deliberately consumptive of energy. Bitcoin and Bitcoin Cash currently account for use of about 0.31% of the world’s energy consumption. Due to these costs, the second-largest cryptocurrency by market cap, Ethereum, is switching to a hybrid of a proof of work environment with what is known as a proof-of-stake environment. Ethereum will rely on virtual miners, or validators, who stake Ether to the system. Validators are rewarded when they follow the rules and lose some or all of their deposit when they break them. Proof of stake is used as a checkpoint every 50 blocks to offer evidence that the blocks hold valid transactions.

eth-sim is a simulator that shows how a crypto economy may look with different user types, population numbers, time periods, incentives, and disincentives for a proof-of-stake-based system, and whether the current incentive structure is playing out fairly or is being exploited. eth-sim will allow its user to set parameters and run a program via a console-based system that enables them to test whether a sample population of a crypto economy is adequately incentivized into validating transactions on the blockchain. The requirement for the amount of Ether to be staked can be tweaked, as a smaller population will have a large enough stake of Ether and thus be able to validate transactions. If the “rich” keep getting “richer”, what effects will that have on the overall economy if an actor behaves selfishly? If only a relatively small amount of Ether needs to be put at stake and the pool for potential validators grows larger, will the “bad” actors outweigh those acting fairly?

The simulations will act according to a decision matrix in order to choose the “best” possible outcome for themselves. eth-sim will ultimately allow you to compute results, such as annual return on an actor’s stake in the system with a series of validators with different preprogrammed strategies. They will behave in various ways, according to how incentivized they are to act selfishly or to contribute to the system, which is one such parameter. The way an actor behaves will be based on a variant of the prisoner’s dilemma.

# Architecture

## Type of System

eth-sim is a simulation software that interacts with its users via a simple GUI system. The user will be prompted to input parameters that grant access to various simulations, with the end goal showing how a crypto economy may react to incentivization structures, with a Moran Process-styled generation mechanic to model which type of actor responds best through multiple generations of birthing and eliminating bots. The results will be analyzed by a class that allow a human being to make use of the simulation data through charts and text summaries, outputted through files in a user-specified location.

## Platform

The application is standalone for PC, potentially for Mac if the build allows it. It is built in Python 3 with the help of numerous plugins to implement important features without “rebuilding the wheel”. Such examples include Axelrod to help model the prisoner’s dilemma process and multiple actor types, simulation helpers, GUI incorporation, and more elegant file creation for displaying results. Plugins to integrate the process of handling blockchain implementation may also be used, although this part could be too difficult to complete by the deadline given the associated technical challenges. Regardless of whether this project gets to this step by then, the simulation should still give its user a good idea of the outcomes had the transactions been on a live blockchain. The blockchain would only serve to record the transactions that are being simulated, and it may be too computationally intensive to set up and run simulations efficiently. However, if this can be implemented, that would be ideal.

## Inputs and Outputs

Transactions on the blockchain are built on trustworthiness of the participants. For that reason, a decision matrix is used to see if the actors in the simulation are credible in their market transactions. This is done through cooperation with the network, as if the network clouds up with too many bad actors, good actors are eventually not compelled to be a part of the system anymore. To assess the actors that can be chosen randomly for the validation pool, users of the simulation are encouraged to play with parameters that incentivize good actors to cooperate with the system, and disincentivize bad actors from manipulating it. These actors must maintain a certain amount of tokens in order to be a part of this pool, so it’s important to encourage cooperation.

Actors are pitted against each other in a generic Prisoner’s Dilemma setup that is multiplied against positive and negative incentives and disincentives. In every generation, participants are pitted against a surviving opponent with four possible outcomes:

* Mutual cooperation: (C, C)
* Defection: (C, D) or (D, C)
* Mutual defection: (D, D)

These correspond to payouts (both positive and negative) as shown:

|  |  |  |
| --- | --- | --- |
|  | **Cooperate** | **Defect** |
| **Cooperate** | (R, R) | (S, T) |
| **Defect** | (T, S) | (P, P) |

For this to constitute a Prisoner’s Dilemma, these parameters must be true:  
 **T > R > P > S**

* **R:** Reward payoff
* **P:** Punishment payoff
* **S:** Sucker payoff
* **T:** Temptation payoff

The parameters that eth-sim’s user can tweak are:

1. Incentive (floating point) – The multiplier by which a bot’s Ether at stake is subject to keep for cooperating in a transaction
2. Disincentive (floating point) – The multiplier by which a bot’s Ether at stake is subject to lose for getting caught cheating in a transaction
3. Number of Generations (Integer) – The number of generations that a simulation is subject to for analysis. As the average validation time for 50 blocks is about 11.66 minutes, this would constitute one generation. The user will be alerted to what could be constituted as normal generations, but simulations are framed to these approximate common intervals:
   1. One day: 123 generations
   2. One week: 864 generations
   3. One month: 3,759 generations
   4. Six months: 22,554 generations
   5. One year: 45,107 generations
   6. Three years: 135,321 generations
   7. Five years: 225,535 generations
   8. Ten years: 451,071 generations
4. Number of games per generation (Integer) – This is the number of games that each actor must play with another market participant to make or lose it to the next generation.
5. Types of simulation participants – These are the various types of strategies that market actors could use in their attempt to gain the most Ether and become validators. While this list could potentially be expanded, these are the basic types of bots that a user can experiment with. These parameters are also subject to noise, a changeable parameter that can make an actor’s decision subject to mistakes (i.e., a bad actor accidentally cooperates, and a good actor unintentionally acts badly):
   1. Cooperator – Actor will always cooperate
   2. Defector – Actor will always defect
   3. Copycat – Actor will cooperate with probability *p* if the opponent’s cooperation ratio is p, starting with a random decision
   4. Bully – Starts by defecting and does the opposite of opponent’s previous move
   5. Random – Actor will always make completely random decisions
6. Number of each actor (Integer array) – For each actor type in the game, the user can specify the number represented in the genesis generation. As the generations increment, if an actor’s Ether stake is too small for the validation pool, they are removed and a member of the class of the last player that “battled” and defeated them is replicated in their place
7. Mistake probability (Floating point) – The probability that an actor will act against the decision that they implemented
8. Ether validator minimum (Floating point) – The amount of Ether an actor must have to continue to the next generation
9. Mode (String) – While sandbox would likely the most useful mode for the simulator, the simulator contains a few sample scenarios as well:
   1. Sandbox – User can tweak parameters
   2. One-year Mixed Population Sample – 20 of each actor sample, 10 games per generation, one years’ worth of generations
   3. One-year Highly Cooperative – 84 Cooperators, with 4 of each of the other actor samples, 10 games per generation, one years’ worth of generations
   4. One-year Highly Defective – 84 Defectors, with 4 of each of the other actor samples, 10 games per generation, one years’ worth of generations
   5. One-year Highly Copycat – 84 Copycats, with 4 of each of the other actor samples, 10 games per generation, one years’ worth of generations
   6. One-year Copycat versus The World – 40 Copycats and 15 of the other population types, 10 games per generation and one years’ worth of generations

The user can run different scenarios, most notably, a sandbox mode that lets the user test whatever parameters they would like to set. There are also preset scenarios that are useful for demoing the capabilities of the simulator for a newbie.

This information is generated and the computer may have to run a potentially computationally-intensive process to crunch the numbers. Data for each generation is summed to find statistics of interest, such as how much Ether the richest and poorest population members contain and which actor classes fared the best and worst as far as longevity through multiple generations, through charts and CSV files. Additionally, this will be saved into a file that keeps track of summary data for previous scenarios, which the user can view to think about how they would build out their incentivization structures.

## User Interface

The UI will be a simple GUI with textboxes that allow for easy input and output of simulations. See the “User Interface” section for more information and a mockup.

# Use Cases

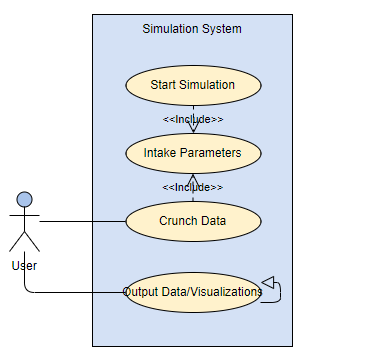
Because this is a somewhat niche-y project, it is somewhat difficult to look at this project in terms of typical use cases.

**Scenario One:**

*“A user wishes to design fair parameters to allow for people to validate token transactions on his online reputation service blockchain, which allows them to earn more tokens. They use a simulation tool to check to see if the incentivization structure could potentially be misused by bad actors over a five-year timeframe. Using his desired setup structure of giving a relatively low amount of tokens for honest transactions but barely disincentivizing cheaters, he his “Run”, and lets the computer crunch data. After several minutes go by, a CSV file and chart is generated with the results of billions of operations to test the incentivization structure, which reveals it is highly likely to be exploited.”*

**Scenario Two:**

*“A user wishes to exploit the current parameters a known proof-of-stake system has in place and emulate using a simulator. By inputting the parameters as closely as possible to how they are currently in place and choosing the strategy she thinks would make her richer in placing transactions with other people, she is able to confirm that over a one-year timeframe, acting as a “Bully” is the best plan of action for the parameters the system currently has in place.”*



# Sequence Diagrams

TODO

# Structural Design

## Class Diagrams

TODO

# Data

The information generated is more or less entirely dependent on the input parameters, so there is no need for a database. However, the output data is saved in a CSV file that lets the users see the results of previous experiments for which they can use for their own various purposes.

# User Interface Design

