Simulating Economic Learning in Dynamic Strategic Scenarios with a Genetic Algorithm

Authors:
Ferraz, Vinícius
Pitz, Thomas

Affiliation:
Alfred Weber Institute of Economics, University of Heidelberg
Faculty of Society and Economics, Rhine-Waal University of Applied Sciences

Contact:
Correspondence should be addressed to visferraz@gmail.com

Version: 1.0.0

Introduction (abstract)

This paper introduces an experimental and exploratory approach, combining game theory and Genetic Algorithms to create a model to simulate evolutionary economic learning. The objective of this paper is to document the implementation of a genetic algorithm as a simulator for economic learning, then analyze how strategic behavior affects the evolution towards optimal outcomes, departing from different starting points and potentially transforming conflict into harmonious scenarios. For this purpose, the introduced construct aimed at allowing for the evaluation of different strategy selection methods and game types. 144 unique 2x2 games, and three distinct strategy selection rules: Nash equilibrium, Hurwicz rule and a Random selection method were used in this study. The particularity of this paper is that rather than changing the strategies themselves or player-specific features, the introduced genetic algorithm changes the games based on the player payoffs. The outcome indicated optimal player scenarios for both The Nash equilibrium and Hurwicz rules strategies, the first being the best performing strategy. The random selection method fails to converge to optimal values in most of the populations, acting as a control feature and reinforcing that strategic behavior is necessary for the evolutionary learning process. We documented also two additional observations. First, the games are often transformed in such a way that agents can coordinate their strategies to achieve a stable optimal equilibrium. And second, we observed the mutation of the populations of games into sets of fewer (repeating) isomorphic games featuring strong characteristics of previous games.

Simulation Model Structure

From a holistic perspective, the simulation model is mainly divided in three parts, defined below:

1. Population – the populations are pre-defined to include all 144 classes of 2x2 games, and the subdivisions in layers and families. Any other combination is possible and also new games are possible, the only rule is to respect the payoff structure thresholds, in which all payoffs are represented by integer values between 1 and 4.

2. Evolution Environment – in this step, the games are played, that is, strategies are selected for both players according to de desired strategy selection rules. Utilities are computed and the fitness scores are assigned to games based on the utility acquired by the players.

3. Genetic Operators – Games are selected from the population based on their fitness scores using the fitness proportionate selection rule (roulette wheel), to become parents of a new game, generated by the crossover and mutation operators. This new game is re-inserted in the population for the next round.

This process is outlined in the figure below. Details of each module will be provided in subsequent chapters.

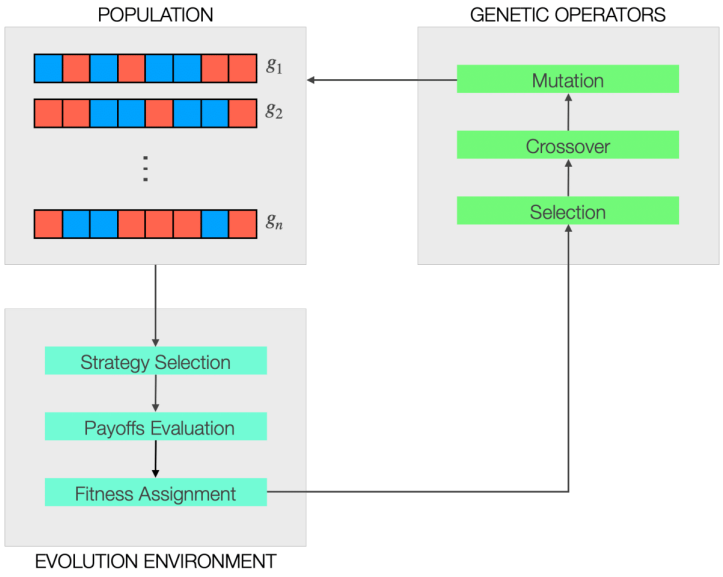

Figure 1 - algorithm structure

Data Inputs (Population):

The inputs consist of 144 2x2 strategic form games converted to vector format, the game vectors are based in the structure of (nested) “list objects”, containing the payoff schemes, the games’ names and IDs. The holistic data structure is based in the “dictionary feature”, the game vectors (list structure) are nested within the dictionary and can be accesses by key names. Populations’ names and games are matched by key-value pairs. This data is stored in a separate file and it is necessary for the simulation model to run. The input file has been given the standard name: `complete_pools_edit.py`, and it is imported in the main code file by the following command line `import complete_pools_edit as cp`.

Parameters Definition

In this step, the main parameters for a simulation round can be defined by the practitioner. The parameters for Strategy Selection allow for variations between populations and strategy selectors (methods and sub-methods) and the parameters for the Genetic Algorithm defines the mains aspects of the evolutionary model.

Strategy Selection Parameters

The `pool_of_games` variable defines which population (game pool) is taken for the simulation process. Based on the layers and families divisions, this value can take the following parameters (added inside the brackets (`[]`) in the data call function `cp.data['population']`):

Layer 1 – `L1`
Layer 2 – `L2`
Layer 3 – `L3`
Layer 4 – `L4`
All layers (entire space) – `complete`
Family: Biased – `biased`
Family: Cyclic – `cyclic`
Family: Prison – `prison`
Family: Second Best – `secondbest`
Family: Unfair – `ufair`
Family: Win-Win – `winwin`

The `strategy` variable takes a numerical value between 1 and 3 and it defines the strategy selector used in the present round. The values correspond to the following strategy selectors:

Nash Equilibrium – `1`
Hurwicz Rule – `2`
Random – `3`

The `eq_sel` variable only affects the code when the Nash Equilibrium selector is chosen. This variable defines the equilibrium selection rules (when more than one is available), based on two distinct methods:

Payoff dominant Nash equilibrium – `1`
Random Nash equilibrium – `2`

The `alpha` variable defines the value of the α parameter when the Hurwicz Rule selector is chosen. The alpha can be set to any value between 0 (pessimistic) and 1 (optimistic).

The `n` variable defines the replacement rate for the genetic algorithm. That is, how many new individuals are generated and added to the next population.

Genetic Algorithm Parameters

The `evolution_target` variable defines the stopping criterion. That is, for how many periods the simulation model should run. It can take any integer value as input, however, the higher the number, the longer the program needs to finish running.

The `mutation_rate` variable defines the probability of the mutation operator taking place within a period. This rate can be set to any value between 0 and 1.

The `crossover_points` variable defines if the crossover operator takes one or two slicing points as standard. The inputs work as follows:

1 point crossover – `1`
2 points crossover – `2`

Core Functions:

The core functions read the input parameters and process the input populations in the logical order as presented below.

Strategy Selection Functions (Evolution Environment)

The following functions belongs to the strategy selection block. They are applied for playing the games and calculating the payoffs for each player in each game. The fitness scores are generated from the utility figures and assigned to the respective games at the end of the procedure.

`NashEquilibrium()` – reads the game vectors and performs the selection of strategies for both players based in the Nash equilibrium rule, followed by the equilibrium selection rule, then calculates the utilities for both players based on the received payoffs.

`HurwiczSelection()` – reads the game vectors and calculates the Hurwicz coefficients for each player and strategy option, then calculates the utilities for both players based on the received payoffs.

`HurwiczRule()` – compares the Hurwicz coefficients and select the pair of strategies with the higher values.

`RandomSelection()` – reads the game vectors and performs a random selection of strategies for both players, then calculates the utilities for both players based on the received payoffs.

Genetic Algorithm Functions (Genetic Operators)

`BinaryConversion()`– converts the game vectors to the corresponding binary sequences, from 8 integers to 16 bits. Each pair of bits represent an integer from 1 to 4 in the following scheme:

```
00 = 1
01 = 2
10 = 3
11 = 4
```

`GeneratePopulation()` – combines the binary strings with their respective position indexes and fitness scores achieved in the game-playing stage.

`WeightsSelection()` – calculates the probabilities of all games for being selected for reproduction, based in relative performance of the individual fitness scores compared to the games in the same population.

`FitnessProportionateSelection()` – performs a weighted random choice by computing a probability distribution based on the individual selection weights, then selects two games from the respective pool to become parents of new games.

`Crossover()` – selects one or two position(s) within the binary strings at random. The binary strings are sliced in the selected position and recombined into a new binary string, the offspring.

`Mutation()` – computes if mutation takes place based on the given probabilities. If negative, the offspring binary string remain unchanged. If positive, a random bit withing the offspring's binary string is selected and flipped, that is, 1 turns to 0 or 0 turns to 1.

`WeightsDeletion()` – calculates the probabilities of all games for being removed from the population, based in the relative performance of the individual fitness scores compared to the games in the same population. This selection works inversely to the `WeightsSelection()` function.

`PopulationAdjustment()` – performs a weighted random choice by computing a probability distribution based on the individual weights, then removes the selected game and inserts the new one (offspring) into the population, keeping the population size constant.

`StrategyVectorConversion()` – converts the binary strings back into the vector format, so they can be played in the next round.

`GeneticLoop()` – controls the execution iterations, while storing the results in a .csv format file. This function also prints the progress and results of the simulation model in the console.

Process Flow

The core functions take as input the results of the previous functions and are executed in sequence. The diagram below illustrates the step-by-step workflow of the algorithm.

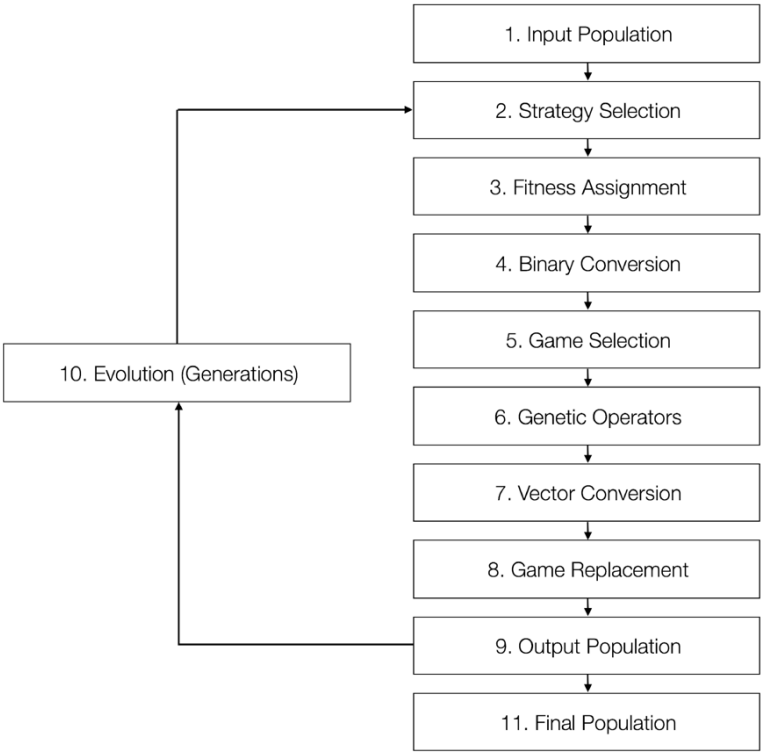

Figure 2 - simulation flowchart

The evolutionary loop consists of taking an input population and iteratively executing steps 1-9 until the stopping criterion is satisfied, that is, `evolution_target`. The results of a full round will be saved in a ".csv" file, with the following name structure, dependant on the strategy selector applied:

Nash: nash + variant (payoff dominant = pdom; random = rand) + pool name.
Hurwicz: hurwicz + pool name + alpha value*100
Random: random + pool name

The results of a simulation round are saved in tabular format with one record (row) for each game in each generation

Output Structure

Each of the strategy selection methods generate a particular output file. The reason behind that is to have the program output all information that is changing across each iteration. At the end of a complete run, the program will generate a ".csv" file, containing

all data points selected by the practitioner within the respective run. The data structure is defined as follows:

Nash Equilibrium Strategy Output

Variable	Definition
generation	Generation number for the data record in this row
game	Game in vector structure
fitness	Fitness values for this game
counter	Simple index counter for each game in a population
eq_count	Count of Nash equilibria found by the Support Enumeration method
nash_list	List of probabilities (arrays) for the one or many equilibria found
eq_list	List of payoffs resulting from the recorded Nash equilibria
R	List of payoffs for the row player from the recorded Nash equilibria
C	List of payoffs for the column player from the recorded Nash equilibria
r_utility	Payoffs yielded for the row player after equilibrium selection
c_utility	Payoffs yielded for the column player after equilibrium selection
r_eq	Strategy probabilities for the row player after equilibrium selection
c_eq	Strategy probabilities for the column player after equilibrium selection
eq_name	Differentiates between payoff dominant (pdom) and random (rand)
poolname	Name of the population used in this simulation

Hurwicz Rule Strategy Output

Variable	Definition
generation	Generation number for the data record in this row
game	Game in vector structure
fitness	Fitness values for this game
counter	Simple index counter for each game in a population
hc_prs1	Hurwicz coefficient value for player row's first pair of strategies
hc_prs2	Hurwicz coefficient value for player row's second pair of strategies
hc_pcs1	Hurwicz coefficient value for player column's first pair of strategies
hc_pcs2	Hurwicz coefficient value for player column's second pair of strategies
prsp_prob_dist	Strategy probabilities for the row player after Hurwicz rule
pcsp_prob_dist	Strategy probabilities for the column player after Hurwicz rule

r_utility	Payoffs yielded for the row player after equilibrium selection
c_utility	Payoffs yielded for the column player after equilibrium selection
alpha	Value selected for the Hurwicz alpha parameter
poolname	Name of the population used in this simulation

Random Strategy Output

Variable	Definition
generation	Generation number for the data record in this row
game	Game in vector structure
fitness	Fitness values for this game
counter	Simple index counter for each game in a population
rd_choice_r	Randomly selected strategy for player row
rd_choice_c	Randomly selected strategy for player column
prsp_prob_dist	Strategy probabilities for the row player after Hurwicz rule
pcsp_prob_dist	Strategy probabilities for the column player after Hurwicz rule
r_utility	Payoffs yielded for the row player after equilibrium selection
c_utility	Payoffs yielded for the column player after equilibrium selection
poolname	Name of the population used in this simulation