The Evolution of Conventions: A Computational Adaptation*

Ege Can Doğaroğlu

March 21, 2024

Abstract

This project is a computational adaptation of the equilibrium selection model in Young (1993). The paper introduces the concept of "adaptive play", which is similar to fictitous play, although players are allowed to make mistakes and have limited information. Theorem 1 of the paper shows that under certain conditions adaptive play converges to pure strategy Nash equilibria, which are the absorbing states of the underlying stochastic process. The paper characterizes two distinct processes as Markov chains, one where mistakes are allowed and another, where they are not. Theorem 2 is technically interesting in that it documents an equivalence relation between certain properties of these two process, namely, stochastic stability and minimum stochastic potential by application of graph theory. This explicit characterization, together with the availability of efficient algorithms makes this paper an interesting candidate for a computational adaptation. The project mainly focuses on computing the states with those properties to portray the equivalence relation.

*Ege Can Doğaroğlu, University of Bonnn. Email: ecdogaroglu@uni-bonn.de.

1 Introduction

This project is a computational adaptation of Young (1993). The paper introduces a less restricting version of fictitious play called *adaptive play*, where in an n person repeated game, each player samples a random portion of *recent history* and plays optimally against this sample of play, as long as there are no mistakes. The author shows that, for a large class of *weakly acyclic games*, if the samples are sufficiently incomplete and if there are no mistakes, the adaptive play converges almost surely to a pure strategy Nash Equilibrium. Once this equilibrium is played for as long as anyone can remember, it's said to have become a *convention,* which are the *absorbing states* of the underlying process.

When mistakes are present there are no absorbing states but if the probability of making a mistake is sufficiently small, then *the stationary distribution* of the play is concentrated around a particular subset of pure strategy Nash equilibria, which are called *stochastically stable.*

Theorem 2 of the paper tells us that stochastically stable states of P^ϵ are the states contained in the recurrent communication classes of P^0 with minimum stochastic potential.

2 Model

Γ is an n-person game in strategic form where S_i denotes the strategies available to player i. N is a finite population of individuals partitioned into n nonempty classes. It's assumed that individuals in each classes have the same utility function $u_i(s)$ for strategy-tuples $s = (s_1, s_2, ..., s_n) \in \Pi S_i$. In each period t, one individual is drawn from each class to play their appropriate role in the game. While player identities for each role may change, their type and available strategies remain constant. There is no learning from past on individual level, so it can be thought that each individual plays the game only once and then *dies.* The strategies taken $s(t)$ are then stored, where sequences of strategy-tuples are referred to as *histories.* Before playing the game players "ask around" and individually sample play histories of size k from the most recent history of size m, where m can be thought as society's cumulative memory size. k/m measures the *completeness* of the agents' information. The probability distribution by which the sampling is made is not important as long as it has full support.

For a given memory size m, all possible such histories of play make up the state space H for the finite Markov chain we will consider.

Code Implementation

The function `define_state_space` in `game.py`is used to initiate the state space for given numbers of actions, players and history size m. First all possible plays that can occur in a period are calculated by taking the cartesion product of the set of actions $num\_players$ times. Then over the set of possible plays, the cartesion product function is again implemented m times, to find all possible histories. The result is a numpy array of size

$$(num\_act^{num\_players})^m \times m \times num\_players$$

3 Parametrization

The project currently only allows for two players. Number of actions can be easily increased. (default=3). As the state space grows exponentially with the memory size, unfortunately it's not feasible to increase m beyond 4 or 5 without long computation times. A similar issue concerns the sample size k (default=1) due to permutation calculations. These two limitations are the main short comings of the project as stochastic variation through large samples and insufficient information through large memory is desirable for the paper outcomes. It's also explicitly stated in the theorems that k must be much smaller than m for some results to apply. ϵ is set to 0.01 by default which is generally close enough to zero for asymptotic results. The payoff matrix is the replication of the 3x3 game in the original paper.

4 Markov Chains

We start from an arbitrary initial state $h(m)$. A history (state) h' is *succesor* of state h if and only if h' is obtained by deleting the left-most element of h and adjoining a new right-most element.

4.1 Unperturbed Process

For the adaptive play without mistakes, transition probability that the process will move from state h to a succesor state h' is

$$P^0_{hh'} = \Pi_{i=1,\ldots,n} p_i(s_i \mid h)$$

where $p_i(\cdot)$ is a *best reply distribution*, whose elements are only positive if there exists a sample of size k to which the strategy is a best-response, for the given state. For any h' that's not a successor state, $P^0_{hh'} = 0$. $P^0_{hh'}$. The process $P^0_{hh'} = 0$ is referred to as the *unperturebed process with no mistakes.* Through this function, the transition matrix of the markov chain for the unperturbed process is characterized.

Code Implementation

The function `compute_trans_matrix_unperturbed` in `markov_chain.py` creates the corresponding transition matrix and saves the resulting numpy array in a `.npy` file to be used later. It uses the `_trans_prob_unperturbed` and `_best_response_prob` helper helper functions to perform above mentioned calculations, together with the `quantecon` library. See the related `docstrings` for additional information on the methodology.

4.2 Perturbed Process

Suppose a subset J of players experiment (or make mistakes) and don't optimize. Conditional on this event, the transition probability that the process will move from state h to a succesor state h' is

$$Q^J_{hh'} = \Pi_{j \in J} q_j(s_j \mid h) \Pi_{j \notin J} p_j(s_j \mid h)$$

where $q_j(\cdot)$ is the conditional probability of choosing the strategy by experimenting in the given state. For any h' that's not a successor state, $Q^0_{hh'} = 0$. The perturbed process then has the transition function

$$P^\epsilon_{hh'} = \left(\Pi_{i,\ldots,n}(1 - \epsilon\lambda_i)\right) P^0_{hh'} + \sum_{J \subseteq N} \epsilon^{|J|} \left(\Pi_{j \in J} \lambda_j\right) \left(\Pi_{j \notin J}(1 - \epsilon\lambda_j)\right) Q^J_{hh'}$$

where $\epsilon\lambda_i > 0$ is the probability that player i experiments, with individual (λ_i) and general (ϵ) components. The process $P^{\epsilon}_{hh'}$ is referred to as the *perturbed process with mistakes.* Through this function, the transition matrix of the markov chain for the perturbed process is characterized.

Code Implementation

The function `compute_trans_matrix_perturbed` in `markov_chain.py` creates the corresponding transition matrix and saves the resulting numpy array in a `.npy` file to be used later. It uses the `_trans_prob_perturbed` and `_best_response_prob` helper helper functions to perform above mentioned calculations, together with the `quantecon` library. See the related `docstrings` for additional information on the methodology. As it's shown in the paper that the individual component of experimentation probability doesn't play a role in the analysis, I also drop this variable in the calculations.

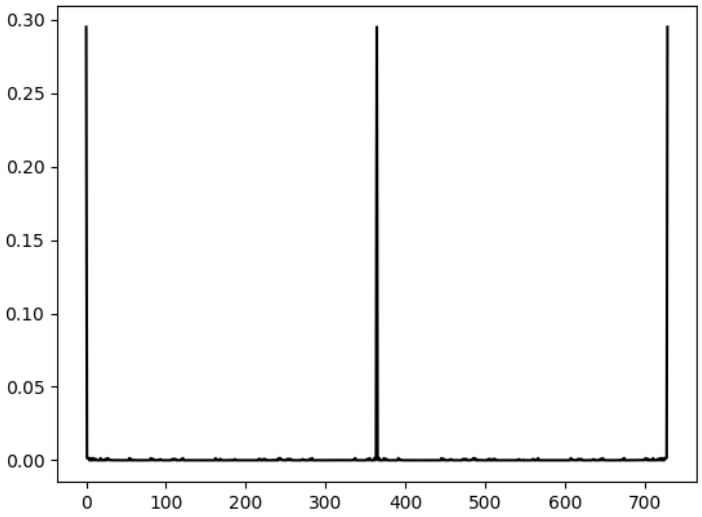


Figure 1: Stationary distribution of the perturbed process ($\epsilon = 0.01$)

state index	probability
0	0.295156
364	0.295156
728	0.295156

Table 1: Stationary distribution probabilities of stochastically stable states. ($\epsilon = 0.01$)

5 Stochastic Stability

Stationary distribution μ^ϵ of the perturbed process with ϵ close to zero, has by definition the property that

$$\mu^\epsilon P^\epsilon = \mu^\epsilon$$

Each element μ^ϵ_h of the distribution can then be seen as the cumulative relative frequency with which state h will be observed at any given time t, if t is sufficiently large. We say that a state $h \in H$ is *stochastically stable* relative to the process P^ϵ if at the limit it will be observed with positive probability i.e.

$$lim_{\epsilon \to 0} \mu^\epsilon_h > 0$$

For $\epsilon = 0.01$ and the 3x3 game example given in Young (1993) all three pure strategy Nash equilibria seem to have positive probability of occuring in the limit. These states are the conventions where both players play the same action (action 1: state id = 0, action 2: state id = 364, action 3: state id = 728) for three consecutive periods.

Code Implementation

The function `compute_stoch_stab_states` in `markov_chain.py` creates the corresponding stationary distribution for the markov chain that is defined by the related transition matrix and returns the states that have probabilities of occuring above the threshold of 0.01. It uses the `quantecon` library.

6 Stochastic Potential

Now we think of the state space H as the vertices of a directed graph, where each node corresponds to a state and for every pair of states there is a directed edge with weights corresponding to *resistances* along the path from one state to another. Resistances are simply the number of mistakes that have to be made to transition from one state to another. If this transition can be achieved through optimal behavior the resistance is zero. If on the other hand the target state is not a successor of the current state, the resistance is equal to infinity as the transition is impossible. Then we create another directed graph where vertices correspond to *recurrent communication classes* of the unperturbed process. For this graph, weights are sum of the resistances along the *shortest path* from one class to another. As by definition, the resistance within the RCCs are equal to zero, we can just pick representative states for each class to calculate the shortest paths. After creating this graph we look for possible *arboresences*, where for each state, the minimum arboresence weight where this state is the root, is the *stochastic potential* of that state.

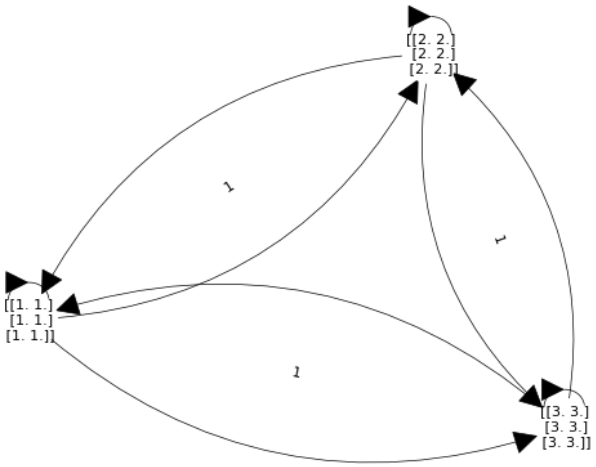
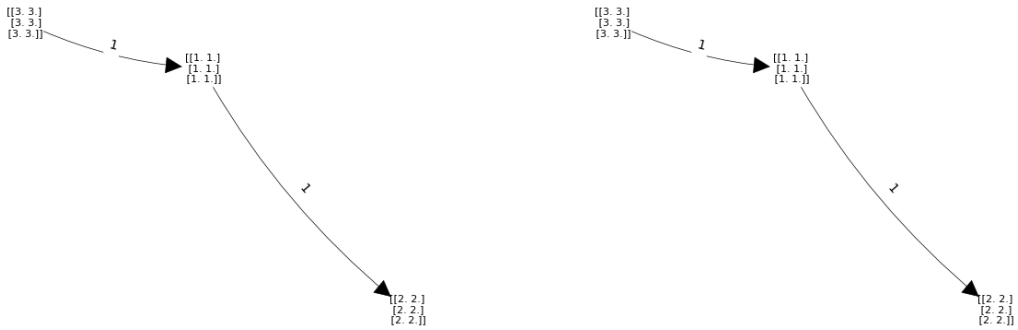


Figure 2: Directed graph of the recurrent communication classes.

Each node corresponds to a state, where each row of the state correspons to a stage game play. 1: "Action 1", 2: "Action 2", 3: "Action 3"



Direct Implementation Iterator 1

Figure 3: Minimum Arboresences

Theorem 2 (Young, (1993) - Shortened)**.** *Let* Γ *be an n-person game on a finite strategy space. The stochastically stable states of adaptive play* P^ϵ *are the states contained in the recurrent communication classes of* P^0 *with minimum stochastic potential.*

The same conventions that were stochastically stable also make up the recurrent communication classes of the unperturbed process and share the same minimum level for stochastic potential. The equivalence relation between the stochastic stability and stochastic potential is inline with the statement of the theorem.

Code Implementation

The mistakes and resistances are computed through best response probabilities over all possible samples, similar to the approach in computing transition probabilities.`quantecon` library allows for simple computation of recurrent communication classes of a given markov chain. `networkx` library is then used to create the directed graphs with corresponding weights and calculating the shortest paths. `networkx` library also offers a direct implementation of Edmonds et al. (1967)'s algorithm to find optimum arboresences, which is also mentioned by Young (1993). The function `find_edmonds_arboresence` in `graph.py` uses this direct implementation. However this method only returns a single such arboresence even if multiple of them have the same minimum weight. In the library also an arboresence iterator is available, which returns all of the arboresences of a given graph in increasing order of

weights by implementing an algorithm due to Sörensen and Janssens (2005). The function `find_states_with_min_stoch_pot` in `graph.py` utilizes this iterator and returns all arborescences with minimum weights and their corresponding roots, which are the states with minimum stochastic potential.

This project uses the reproducible project template by Von Gaudecker (2023).

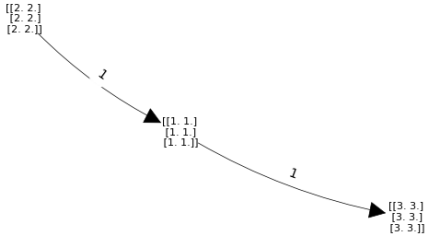
References

Edmonds, Jack et al. (1967). "Optimum branchings". In: *Journal of Research of the national Bureau of Standards B* 71.4, pp. 233–240.

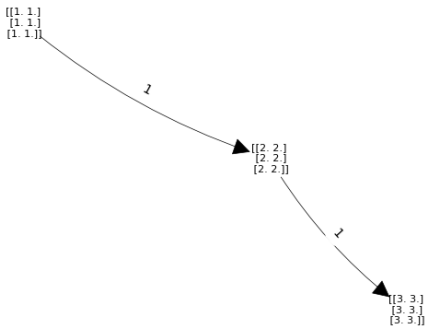
Gaudecker, Hans-Martin von (2023). "Templates for Reproducible Research Projects in Economics". https://doi.org/10.5281/zenodo.7780520.

Sörensen, Kenneth and Gerrit K Janssens (2005). "An algorithm to generate all spanning trees of a graph in order of increasing cost". In: *Pesquisa Operacional* 25, pp. 219–229.

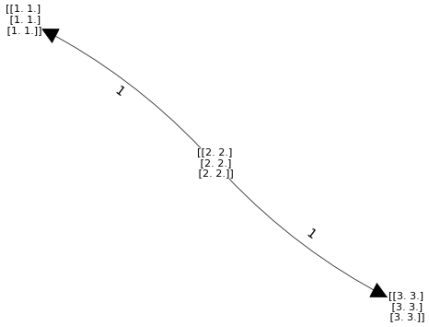
Young, H Peyton (1993). "The evolution of conventions". In: *Econometrica: Journal of the Econometric Society*, pp. 57–84.



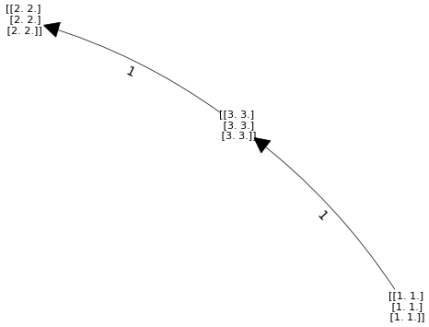
Iterator 2



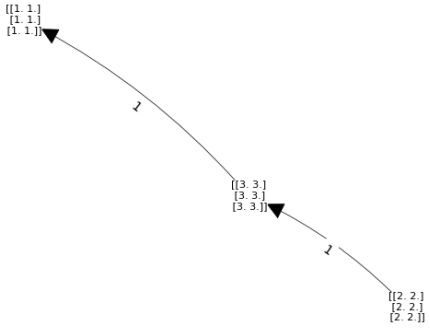
Iterator 3



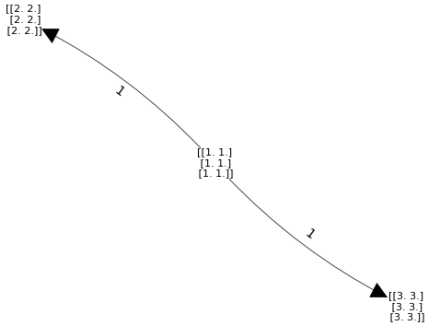
Iterator 4



Iterator 5



Iterator 6



Iterator 7

Figure 4: Minimum Arboresences

Each node corresponds to a state, where each row of the state correspons to a stage game playy. 1: "Action 1", 2: "Action 2", 3: "Action 3"