NSF-REU 2023 - README

Will Fowlkes and Thomas Narramore

Summer 2023

1 Introduction

This README file provides a description of the code used in the 2023 NSF-REU project on Evolutionary Dynamics: Kin Competition and Dispersal Timing: A game-theoretic approach.

The purpose of the code used in the project is to find Evolutionary Stable solutions to the game described in the paper. To this end, we programmed two different algorithms in Python. Thomas Narramore (thomas.narramore@western.edu) implemented the backwards induction algorithms, and Will Fowlkes (whitman.w.fowlkes@vanderbilt.edu) implemented the Fast Nash algorithm (see the paper for a description and proof of this algorithm).

2 Parameters of the model

In general, the parameters used in the program are stored in a Python dictionary. The parameters used and their default values are as follows

Parameters List		
Parameter	Value	Meaning
N	1	Number of breeding cycles
n	4	Litter size
T_{max}	120	Length of breeding cycle
R_{min}	40	Minimum resources required for dispersal
R_{max}	136	Resources required for reproduction
r	$\frac{12}{5}$	daily resources obtained by parent
c	2	daily resources obtained by dispersed child
k	0.5	constant representing death during dispersal
b	0.8	fecundity boost given to dispersers
f	4	base fecundity of an individual

In general, the values of n and T_{max} must be positive integers, and all other parameters must be positve real numbers.

3 Fast Nash Algorithm: Will Fowlkes

The file fastfinder.py contains the code which impelements the Fast Nash Algorithm. To use this program, simply call the main() function. The main() function has three steps: getting a dictionary of valid parameters, computing results, and writing results to an outfile or the console.

3.1 Getting Parameters

The parameters used in the model are stored in a Python dictionary. The key is the name of the parameter. For example, the parameter T_{max} stored in the dictionary `dict` can be read from or written to through `dict["Tmax"]`. Once a dictionary of parameters has been provided, the dictionary is used as the `d` argument to all functions that must access its data.

3.2 Calculating the ESS

After a dictionary of parameters has been provided, three main function calculate the evolutionary stable dispersal timings and payoffs. The first of these is `a_finder(d)`, which takes a dictionary as a parameter and returns `a_vector`, which is a list of length T_{max} giving the remaining number of philopatric individuals at the end of each day. For example, if `a_vector = [3, 2, 1]` and $n = 3$, we know that no individuals dispersed on day 0 (since 3 remain), 1 dispersed on day 1 (since 2 remain), and another dispersed on day 2 (since 1 remains). `a_finder(d)` calculates `a_vector` by looping through the players and calculating the payoff for each dispersal date. To know this payoff, the program must know the amount of resources gathered at each day. This is handled by the function `calc_resource_vector(d, a_vector)`, which uses the fact that later individuals disperse earlier to calculate and return `resource_vector` - the total resources an individual would have if they dispersed on any given day. `calc_resource_vector` returns `resource_vector`to `a_finder(d)`, and then `find_dispersal_date()` is called. This calculates the payoff given dispersal at any given date and the payoff with no dispersal. The date of dispersal yeilding the maximum payoff is found by simply looping through all possible days and checking the EFRS, which can be calculated since we have the `resource_vector` for that player. As `a_finder(d)` loops through players, it fills out the `timing_matrix`, which is a 2D visualization of the departure dates of individuals. After finishing the loop, the finished `a_vector` can be used to determine the departure date and payoff of any player. To get the departure vector (the list of all players' departure times), simply call `get_departure_vector(d, a_vector)` with the dictionary `d` of parameters and `a_vector` as arguments. To get the payoffs of all players, simply call `get_payoffs(d, get_departure_vector(d, a_vector), a_vector)` using the dictionary `d`, the departure vector, and the `a_vector` as arguments. Both the departure vector and payoff vector are Python lists are ordered by player number.

3.3 Time Complexity

the main computation of the algorithm involves looping through n player and checking the payoff obtained at T_{max} possible departure dates. The computations involved in payoff calculation are simple arithmetic operations an thus performed in $O(1)$ time (provided that the values are not excessively large). As such, the overall time complexity of the Fast Nash Algorithm is $O(T\_max \times n)$

3.4 Fast Nash Algorithm Visualization

This algorithm can be visualized in the form of a departure matrix. Here 0 represents that an individual has not dispersed on a certain date, and 1 represents dispersal. The departure times are given in the leftmost column, and the player numbers (in order of choice) are on the bottom row. In this example $T_{max} = 5$ and $n = 3$.

Departure			
4	0	0	0
3	0	0	0
2	0	0	0
1	0	0	0
0	0	0	0
	1	2	3

Since we know that no other players will disperse before player 3, we know that she will always obtain exacly $\frac{r}{3}$ resources per day. This means that player 3's decision need not depend on the other players, she can be guaranteed to maximize her EFRS. Suppose that she disperses on day 1. The departure matrix thus becomes

Departure			
4	0	0	1
3	0	0	1
2	0	0	1
1	0	0	1
0	0	0	0
	1	2	3

Note that since player 3 dispersed on day 1, the column corresponding to player 3 will be 1 for all $d_3 \geq 1$ The process is repeated for player 2, who now can expect to receive $\frac{r}{3}$ resources on day 0 and, since player 3 will have dispersed on day 1, $\frac{r}{2}$ resources per day thereafter. Suppose that it is then optimal for player 2 to disperse on day 3. The departure matrix thus becomes

Departure			
4	0	1	1
3	0	1	1
2	0	0	1
1	0	0	1
0	0	0	0
	1	2	3

This process is repeated for player 1. Supposing that player 1's best option is to never disperse, the final departure matrix is

Departure			
4	0	1	1
3	0	1	1
2	0	0	1
1	0	0	1
0	0	0	0
	1	2	3

By looking at the completed departure matrix, it is easy to check the daily resources and departure times of each player. es not necessarily return a strict Nash equilibrium. If, for a deciding player, both child nodes return the same efrs, then the node corresponding to the strategy of staying in the natal area is chosen. However, we found that Nash equilbria tended to be strict in our model, with the exception of edge cases that were contrived to have multiple Nash equilibria. More details can be found in the "Uniqueness of Nash equilibria" section of our paper.