Genetic Algorithms [MT21104] Term Paper

A study on the use of genetic algorithms for socio-economic simulations and predictive modelling.

Maitreyi Swaroop
19MA20065

Term paper presented for course evaluation
Spring Semester, 2021

Department of Mathematics & Computing
Indian Institute of Technology, Kharagpur

Date of Submission: 15th April, 2021.

Simulating Agricultural Economy Using Genetic Algorithms

Maitreyi Swaroop, 19MA20065

Department of Mathematics & Computing, Indian Institute of Technology, Kharagpur

April 15, 2021

Contents

Abstract

Genetic algorithms (GAs) are most commonly used for optimisation and search problems. However they also harbour great potential for socio-economic modelling and running simulations. Current modelling methods applied in Economics have the disadvantage of being unrealistic in their approach due to the many simplifying assumptions made. The heuristic and evolutionary nature of GAs allows them to account for some degree of randomness and non-ideality in the behaviour of the subjects in the simulation and can be used to mimic the actions of economic agents, both as individuals and as a collective. The aim of this paper is to test the use of genetic algorithms in the modelling of economic markets with the different market players. In particular, an example of the agriculture market of a hypothetical nation will be shown using a modified form of the simple genetic algorithm (SGA). The model in this paper, being an introductory one, will be quite simple. The discussion at the end explores how further levels of detail can be added and how genetic algorithms can be harnessed for running simulations resembling real-life more than existing classical methods.

1 Introduction

1.1 Genetic Algorithms

Genetic algorithms attempt to mimic natural systems and entities to benefit from the ingenuity and efficiency of natural design in solving problems. However, genetic algorithms can also help models incorporate the *inefficiency* and *irrationality* of human behaviour.
While GAs has been applied to Economics before, the use in this paper does not seem to have direct precedent (to the best of my knowledge). To illustrate, three papers on the use of Genetic Algorithms can be referred to (bibliography items [1],[2],[3]). The first one deals with application of auctioning mechanism and GAs to the land rental markets. Here the producer is assumed to act in the way that yields them the best possible payoff (in terms of profits), however the basis for profits i.e. agricultural production is not within the purview of the paper. The second paper mentioned above investigates use of binary encoded genetic algorithms for assessing economic strategies and concludes that such algorithms are difficult and unwieldy for large amounts of complex economic data. The third paper uses GAs for modelling the behaviour of adaptive and rational agents in an economic system.

Here we are using genetic algorithms to model and predict the future state of a set of economic players in the agriculture sector without any external influences on the sector itself. The model would predict changes over time in size of landholdings,number of landholders, crop production mix, profitability, productivity etc. In real life these changes do not happen without interference from external forces. Decline in profitability of agriculture, or indebtedness of cultivators could lead to government intervention by way of direct financial support or assured selling prices. Similarly acquisition of farmland by prosperous large farmers could be opposed by social or political groups. If the model is developed further it may be possible to factor some of these externalities in the model to give a future state closer to reality. While in this term paper we have investigated in a limited way, the potential changes in the composition of the agricultural economy, a more elaborate model can also predict the employment potential of the agricultural sector and in the absence of non-agricultural employment in villages, the extent of rural to urban migration along with corresponding need for urban employment and urban infrastructure (as discussed in conclusion).

1.2 Agricultural and Food Markets

Agricultural markets are peculiar in certain respects which differentiate them from markets for other products like manufactured consumer goods or industrial goods. The first is that usually, agricultural produce can be considered to be largely homogeneous- eg. Wheat, lentils and rice grown by different farmers are all identical (we shall ignore special cases such as the 'basmati' rice variant which is considered a distinct product, different from other kinds of rice). Introductory microeconomics courses taught at high school and college levels always classify agricultural markets as 'perfect competition', where each producer is indistinguishable from each other in terms of access to resources, information about the market, productivity and the good produced. They also classify each producer as a *price-taker*. Since no individual is large enough to influence the price of the commodity as they please, every individual is forced to sell at a prevailing market price. Here, we are doing away with all the above assumptions except those of product homogeneity and of producers being price takers. Thus the producers are different from one another but the goods they sell are not and ultimately they have to sell their produce at the determined market price. This is done for the sake of simplicity of modelling.

We are confining our study to only food crops and amongst food crops to foodgrains. Here we will focus on the market for 3 hypothetical crops ('A', 'B', 'C'). An analogy can be drawn to Rice, Wheat and Pulses.

In India the foodgrains market is highly regulated with sale through government mandated marketplaces and guarantee of Minimum Support Prices by the government. The de-regulation of the agricultural markets and the introduction of a 'free market' in the Indian Agriculture Sector is an ongoing subject of debate in recent Indian public life.

However, for the purpose of our simulation, we are assuming a free-market wherein there are no middle-men and no government intervention (i.e. no subsidies or benefits to producers). We also assume that most farmers sell their produce in their own state in neighbouring states (because of assumed resource constraint of transportation and storage). In later versions of this simulation we can take into account a more complex picture where depending on their own size, producers have differential access to resources, information and external markets. We can also tweak our model in later versions to accommodate regulated markets, middlemen and multiple crops.

1.3 Using Genetic Algorithms to run market simulations

The first question that comes to mind is - 'Why apply genetic algorithms to economics market simulations in the first place when fairly complex models do exist for similar purposes?'

The reason is, the study of theoretical economics is quite detached from what is seen in reality due to the assumptions needed to build the theory. The assumptions of rationality of consumers and producers, relying on them to take the *optimal* decisions in extremely controlled environments, assuming perfect knowledge of the market among others make the resulting model

Genetic algorithms, being randomised to an extent can account for randomness in the model, resembling the sometimes erratic and poor decision making of various players. However the fitness function and over-arching 'survival of the fittest' principle in determining who survives rewards good decision making. By assigning traits to individuals in varying degrees of intensity (eg. the propensity of a producer to maximise profit over sales, or to choose between alternate lines of production), allows the population in the simulation to mimic human behaviour more

realistically and also show us the possible outcomes for the individuals and the economy as a whole.

As a clarification, I am in no way implying that the models evolved over the iterations of the genetic algorithm can replace the standard models. However they can be used to augment the standard models to help arrive at a more holistic view future scenarios. This holistic view can lead to better decision making and planning.

1.4 In summary

What we assume

- The total supply needs to be created by domestic producers. There are no exports.
- The demand side of the market will remain unchanged over the course of the simulation.
- We will only limit ourselves to the production and sale of 3 crops *'A', 'B' and 'C'*.
- Homogeneity of producer - the produce of a given producer is indistinguishable from that of the other. There is no reason for a producer to demand a higher price other than a profit maximising incentive.

What we wish to achieve

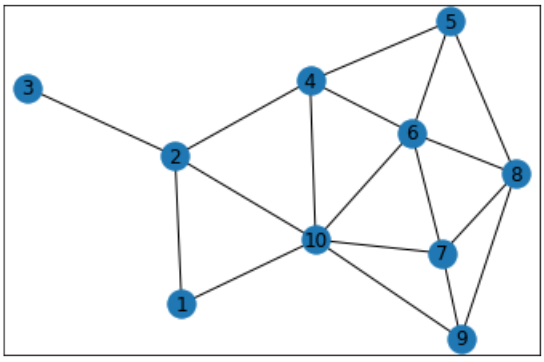
To run a market simulation. By the end of the simulation, we wish to get an idea of what the supply side of the market would look like. This can help in getting an idea of the impact of policy decisions and what letting the free market with zero government interference (a *'laissez faire* policy') would lead to.

2 Modelling the problem

This simulation makes use of the **Simple Genetic Algorithm (SGA)**. Each member of the population is a python list. The items in this list can be integers, floating point numbers and lists themselves. These features are elaborated on later in the paper in the table

We have a (hypothetical) nation comprising 10 states (numbered 1 to 10). The network graph of the states of our hypothetical nation is given below-

Figure 1: Each node represents a state and each edge a state border connected two states.



Each state will be referred to by its respective number (i.e. State 1, State 2,..., State 10). With each state there are its associated features:

Table 1: Features of the state. Each state is an object with the following attributes.

State Feature	**Description**
Name	The state number (as seen in the image above)
Available Area	Total land available for use in agriculture (in particular for growing the concerned crops).This is a fixed feature and will not change during the course of the simulation.
Effective Area	Effective land area is the amount of land that is actually being put to use at a particular point of time/ generation. This is a *variable feature* and can decrease/ increase. The upper limit for this is restricted by the available area.
Neighbours	Given as in image above. Only immediate neighbours are considered. A producer in one state can only sell to consumers in their state or in the neighbouring states.
Crops	The set of crops which can be grown in the state. The kind of crops to be grown are predetermined for every state and do not change with time (due to assumed limitations of geographical factors such as soil type etc). Here we are only considering the production and sale of 3 crops `A`,`B` and `C`.
Number of Consumers	Total population = number of consumers. For the purpose of this paper this will be assumed to be fixed. [1]
Number of Producers	These are the number of people in the state who are engaged in agriculture as *owners of the land*. We are not considering landless labourers engaged in agriculture for this paper, though they constitute a significant portion of the Indian agricultural workforce. [2]
Land Holdings	This is a list of the different land holding sizes. It is variable and will change from generation to generation. One of the aims of this paper is to see how the distribution of farm sizes would change over time in a largely free market. To generate the intial values for `holdings` the $lognormal(\mu = -0.17, \sigma = 0.9)$ distribution was used since the existing farm sizes in India fit this distribution best.

2.1 Demand Side

The consumers are considered to be a static population, with a constant size, tastes and preferences etc. The focus on agricultural markets makes such assumptions convenient as for a fixed population size, the dietary requirements of the population on generally do not change much. Food products, being essential commodities means that demand for food crops will be inelastic and the producers will ultimately have to buy the produce at the price set by the sellers.

The demand for each crop of the three crops under consideration here is specified for each state.

2.2 Supply Side

The producers are the main focus in this simulation. Thus whenever the word 'population' is used it is referring to the population of producers.

First, an ideal price is set by each producer: this is specific to each producer and indicates the price

at which they are willing to sell. Ultimately, due to the large number of producers in agricultural markets, each producer is a price taker (i.e. they are forced to sell at the market price, determined by all the producers at large). So the average of all individual prices is taken as the market price for the commodity in the given state. In later explorations of this topic an auction-mechanism based sale procedure can also be employed for sale of produce.

The simulation is run for a user specified maximum number of generations. Usually this will be in the range of 2-5, where each generation is equivalent to a crop/ market cycle ranging from a quarter of a year to a year. The basic outline of the simulation is given in the following flowchart-

Figure 2: Flowchart of simulation.

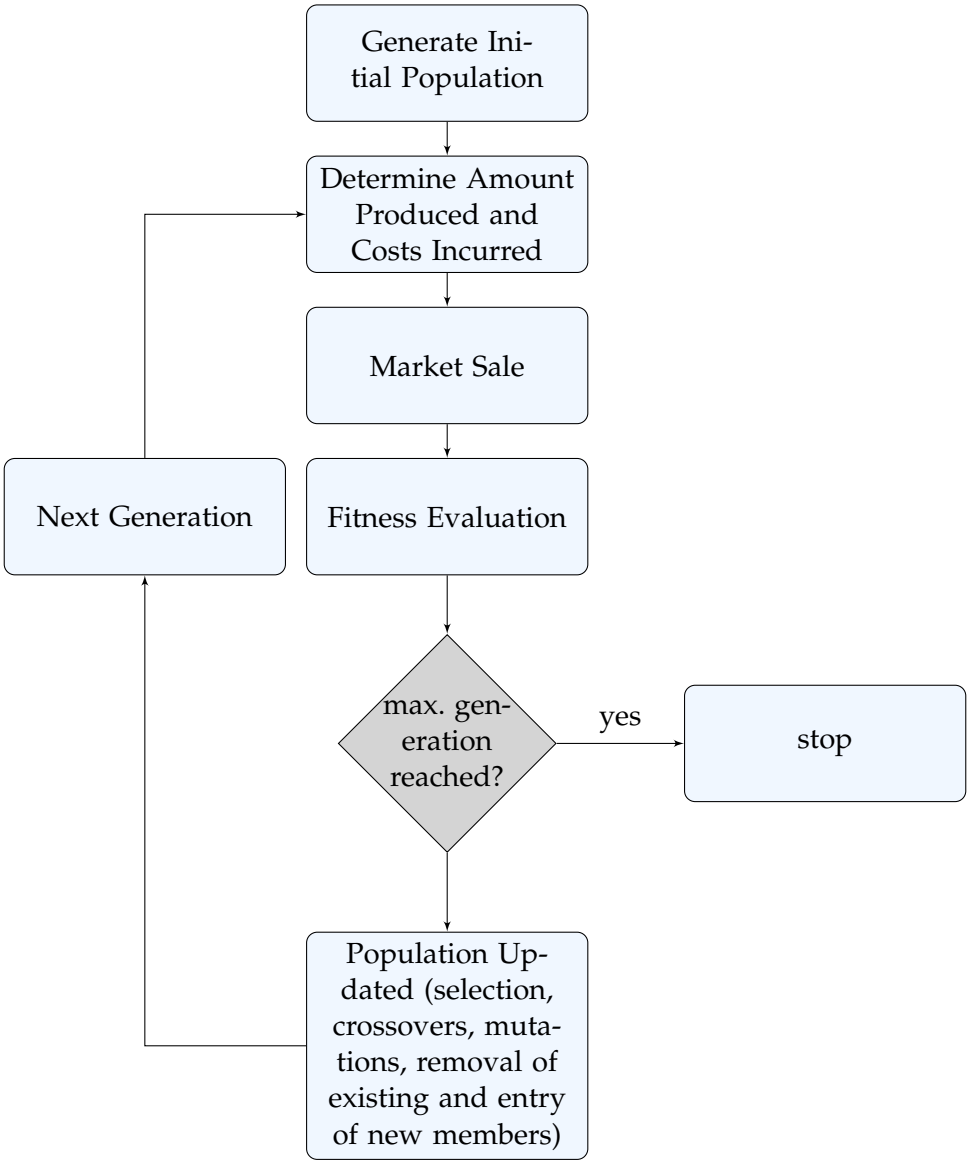


Table 2: Features of a producer. These are the items in each individual producer 'list'.

Producer Feature	**Description**
State	The state to which the producer belongs, i.e. where their farm land is located and where they produce and (primarily) sell.
Holding Size	The size of their land holding. For initial population members this is allotted from a lognormal distribution (as mentioned in the state description). Later on holding sizes may increase/decrease due to crossovers and mutations (to be explained below.)
Crops Grown	The crops which a producer can grow are determined by the state they live in. The proportion in which they are grown , i.e. the proportion in which they divide their land for growing the crops is at first randomly distributed and then varies based on which crops give better returns etc. In order to account for non-ideal behaviour an element of randomness is also introduced.
Productivity	This parameter determines how effectively a producer utilises their resources. A maximum amount of output per unit land for each crop is given. The productivity of a producer is the fraction of this maximum yield that the producer manages to produce. Productivity also determines the order in which sellers sell their produce in the market. More productive sellers get a higher preference. For the initial population this is be a random number in the interval $[0, 1]$ assigned from a normal distribution ($Normal(\mu = 0.5, \sigma = 1)$).
Money	This is the amount of money a producer has available for initial investment. For populations the distribution of income and wealth is seen to follow the lognormal distribution as used for the distribution of landholdings too, so for the initial populations the money is a function of their holding size with a scaling factor determined by both the crops they grow and a random integer.
Profit Emphasis	This is a measure of the importance the producer places on profit maximisation as opposed to sales maximisation. For the initial population this too is a random number in the interval $[0, 1]$ assigned from a normal distribution ($Normal(\mu = 0.5, \sigma = 1)$).
Amount Produced	This is determined by the producers productivity, the size of their landholding and the crops they are producing. It is given by the formula $Crop_i = r_i \times h \times \eta \times Crop_{i,max}$, where $Crop_i$ is the amount of crop i grown, r_i is the proportion of the land allotted by the farmer to crop i, h is the landholding size, η is the producer productivity and $Crop_{i,max}$ is the maximum possible yield per unit land for crop i (user specified).
Production Costs	The cost of production for each crop is (here) assumed to be fixed and is user specified for each unit of each crop. Production cost for each crop is thus simply the product of the amount produced for each crop with their specified production cost per unit. This is maintained as a list with costs for every crop grown.
Money Earned	Money earned is the product of the prevailing market price with the amount produced, should the seller get a chance to sell their produce at all (procedure for sale explained below in Market Sale-Sub procedure).

2.3 The evolution

2.3.1 Generating Initial Population

Calling the function *generate_population* generates the initial population. The procedure by which the features of the initial population members are allocated has been elaborated on before (in both the state and the producer descriptions) and hence will not be repeated here.

2.3.2 Market Sale

The *marketSale* function simulates the act of sale of the produce. It can be broken down into 3 stages:

(i) **Price Determination**- The price of each crop is determined for each state separately. It is taken as the average of the prices being offered by all the producers of that crop within that state. This is the state-market price for the crop at which each producer of that state will have to sell their produce.

(ii) **1st Stage Sale**- First the producers of every state try to meet the demand of the consumers within their state. The producers with the highest productivities are the first to sell their produce. Low productivity producers are late to act and thus are forced to sell later (if at all) in their state or to other states (if the demand exists). However they may have a chance of benefiting from this if the prevailing price in the neighbouring state is higher than in their state.

(iii) **2nd Stage Sale**- Each state is then checked for instances of excess demand (i.e. demand which could not be met by producers from their own state). If a state is found to have excess demand producers from the neighbouring states with excess produce sell the required crops in order of decreasing productivity at the prices prevailing *in the state of sale.*
Simultaneously the amount of money earned by each producer is being calculated which is a determinant of producer fitness.

2.3.3 Fitness function

The fitness function, a measure of the competence and the financial strength of a producer, here is taken to be the weighted sum of the accumulated money with the producer available for future investment (A) at the end of the most recent market cycle and the profits accrued in the same cycle ($P = R - C$, where R is the revenue, i.e. amount of money earned, and C is the cost). This is then divided by 1+ the size of their holding (H):

$$\frac{0.75A + 0.25P}{1 + H}$$

which can be written as

$$\frac{0.75(I + R - C) + 0.25(R - C)}{1 + H} = \frac{0.75I + R - C}{1 + H}$$

where I is the initial amount they had for investment. The reason for dividing by (1+ holding size) is that if a smaller landholder earns the same amount as a larger one, the one with less land is the fitter of the two.

2.3.4 Updating the population

Crossover We have 2 kinds of crossovers in this simulation. Both of them are replacement crossovers, i.e. the offspring generated replace the original parents. The types of crossovers are:

1. **2-1 crossover**: This is equivalent to a takeover of a landholding of a small producer by a bigger/fitter producer, or a merger of 2 landholdings. Two producers (p_1, p_2) combine their resources to produce one new production entity. The only criteria to be met is that the two producers belong to the same state. Here the net effect on the producer list is:

- Location will remain the same (they must belong to the same state).
- The holding size will be the sum of the individual holding sizes.
- Proportion in which crops are grown will be determined by a random number in the range of the production of crops for the two individual producers.
- Productivity will be a random number generated in the range of the respective individual productivity of each producer.
- Money will be added.
- Propensity to emphasise profit will be a random number generated in the range of the individual emphases on profit.
  Here number of landholdings and producers decrease.

2. **2-2 crossover**

- Locations of each individual will remain the same as before.
- The holding size for each will remain the same as before.
- Proportion in which crops are grown will change for the less fit producer and will remain the same for the fitter one. If the fitter producer belongs to the same state as or a neighbouring state as the weaker producer, the new proportion in which the weaker one produces is scaled to become closer to the proportion used by the fitter producer.
- Propensity to emphasise profit for both will be a random number generated in the range of the individual emphases on profit.

Mutation The mutation, though probability driven, will not be entirely random. The scope for change in the long run is not too high, though still much higher than that in the short run. The probability for every kind of change made to the individual will be quite low. It will be made lower by the multiplicative effect of probabilities as the decision for whether an individual should be considered for mutation at all is also probabilistically determined.

The following things can be altered and achieved during mutation of a producer p:

- **Holding size** can change: If the total amount of land available in the state for use in farming is greater than the sum of all holding sizes then the producer can buy more land to increase their land holding size. *This will be determined by whether the producer has enough money to buy more land or not. It will result in an increase in holding size and a decrease in amount of money available.* The holding size of a producer may decrease too, but this occurs only if their holding size is >2 units and even then with a very low probability.

- The **Proportion in which they grow their crops** can change. This new proportion will be determined by the following method: Two producers (p_i, p_j) belonging to the same state as the will be chosen from the current producer will be chosen from the population (with a weighted probability, the chances of being selected being proportional to the fitness of

the producers). The new proportions will be a scaled version of the proportions used by these two producers. In real markets too the less successful sellers often try to imitate the production patterns of the more successful ones.

- **Productivity** of the producer will be once again randomly generated, thus it may increase or decrease.

Selection for Staying in the Market and Production Cycle The range of fitnesses arrived at in this simulation is quite high. We can divide the course of action to be taken regarding a producer into categories depending on which interval of the fitness range they lie in. After sorting the population in descending order of fitness we consider the different sections:

1. `producer in first half of population members with fitness(producer)>0`:
   Probability of 2-1 crossover with any randomly chosen, weaker member of the population in their state = 0.1.
   Probability of a 2-2 crossover = 0.05×(1-0.1) = 0.045, since the 2-2 is only considered if the 2-1 crossover does not happen.
   Every time a producer is crossed over with another, the other producers are removed from the population under consideration for selection. This prevents double emphasis on an individual and their traits.
   Probability of mutation = 0.01×(1-0.1)×(1-0.05) = 0.00855, since a fit producer is not incentivised to undergo too much change within a given generation, we only consider mutation if no crossover occurs.
   If none of the above occur then the producer is passed into the next generation population unchanged.

2. `producer in bottom half of population members with fitness(producer)>0 :`
   Here instead of considering each member individually, we shuffle this portion of the population and take population members in pairs. These two individuals will be considered for crossovers with each other.
   Probability of 2-2 crossover = 0.2
   Probability of a 2-1 crossover < 0.15×(1-0.2) = 0.12, here the 2-1 is only considered if the 2-2 crossover does not happen and if the two producers belong to the same state (hence the inequality).
   Probability of mutation = 0.3×(1-0.2)×(1-0.15) = 0.204.

   If the number of producers in this category is odd the remaining individual is arbitrarily chosen to be part of the new population.

3. `producer for whom fitness(producer) < 0`:
   Here too we shuffle this portion of the population and take population members in pairs. These two individuals will be considered for crossover with each other.
   Probability of 2-2 crossover = 0.5
   Probability for a 2-1 crossover < 0.5×(1-0.5) = 0.25, here the 2-1 is only considered if the 2-2 crossover does not happen and if the two producers belong to the same state (hence the inequality).
   Probability of mutation = 0.5×(1-0.5)×(1-0.5) = 0.125. The ability for the weaker individuals to undergo change without external assistance (as it may be in crossovers) is quite low, hence their probability for mutations is kept lower than for members of the other sections of the population.

If a producer in this category does not undergo any form of change they are made to exit the market. If the number of producers in this category is odd the remaining individual is arbitrarily chosen to be part of the new population.

Entering and Leaving the Population In any real, there exist incentives for entry and exit of new and old sellers. New producers are likely to enter a market if

1. the market is not currently at optimal/required size from a supply perspective, i.e. if there is excess demand which is not being met by the current producers there is room for new producers to enter and tap into this consumer base.
2. there is a chance of making profits. Producers will only want to produce a commodity if there is a scope of earning money by making profits. If they see that entering a market is financially beneficial to the current market players they are likely to enter that market.

Inversely, lack excess demand in the market and current players in the market being loss making act as deterrents to new potential producers. In case of the agriculture markets, the physical limitation of resources (namely land available) also acts as a barrier or atleast an upper limit for entry of new individuals.
Here after the existing population is dealt with, each state is iterated over. If the conditions of excess demand and availability of land are satisfied, entry of new producers in the market of that state is considered. The new entrants are created by mutating producers of that state (as they were in the previous cycle). Only those producers with fitness > 0 will be considered. The production pattern of the new producer will be a scaled version of the producer chosen for mutation. The scaling of production (i.e. the proportion in which the crops are grown) will be biased towards that crop which has greater demand. However, since it is known that people do not necessarily make such systematically thought out decisions, and also may not have sufficient market knowledge, a different random number is additionally used as a scaling factor for each crop.

The new producer's landholding is of the same size as that of the producer chosen for mutation (for simplicity) and their productivity and money for initial invest are drawn from a normal distribution, as they were for the initial population.

The number of new entrants is a fraction of the population size of the state having fitness greater than zero. A random number generator also allows variation in this fraction by randomly rejecting certain individuals.
Finally the new population is arrived at and the procedures of production, amount produced and cost determination and market sale are repeated for the specified number of generations.

3 Simulation Results

Here are the results of the simulation run for 5 generations (assumed equivalent to 5 years). Below is the tabulated form of the data generated for the number of producers, the effective area and the mean landholdings size. As trends in the number of producers and mean landholding size also affect the trends in effective area, looking at the former two should give us a sufficient understanding of the latter as well. Although details of crop production, profits etc. have not been added here they can easily be generated and analysed.

```
GENERATION 00
=============
Total Number of Producers:  642
```

	State	Number of Producers	Effective Area	Mean Landholding Size
0	1	110	130	1.18
1	2	54	70	1.30
2	3	20	30	1.50
3	4	33	40	1.21
4	5	79	80	1.01
5	6	111	150	1.35
6	7	102	130	1.27
7	8	21	20	0.95
8	9	30	40	1.33
9	10	82	100	1.22

```
GENERATION 01
=============
Total Number of Producers:  664
```

	State	Number of Producers	Effective Area	Mean Landholding Size
0	1	103	123.706	1.26
1	2	54	79.260	1.47
2	3	19	30.000	1.67
3	4	40	48.670	1.22
4	5	75	79.610	1.07
5	6	106	146.006	1.46
6	7	117	168.520	1.50
7	8	26	26.734	1.04
8	9	28	39.570	1.41
9	10	96	119.460	1.26

```
GENERATION 02
=============
Total Number of Producers:  638
```

	State	Number of Producers	Effective Area	Mean Landholding Size
0	1	89	118.062	1.42
1	2	51	79.247	1.55
2	3	18	30.000	1.77

3	4	40	48.278	1.27
4	5	75	78.421	1.11
5	6	99	134.500	1.57
6	7	121	179.944	1.66
7	8	26	29.404	1.15
8	9	26	38.440	1.48
9	10	93	119.969	1.35

GENERATION 03
==========
Total Number of Producers: 603

	State	Number of Producers	Effective Area	Mean Landholding Size
0	1	81	102.790	1.63
1	2	48	79.247	1.65
2	3	16	29.260	1.94
3	4	37	47.839	1.38
4	5	73	77.567	1.25
5	6	96	123.714	1.52
6	7	112	179.934	1.81
7	8	25	28.725	1.22
8	9	23	37.980	1.66
9	10	92	118.737	1.48

GENERATION 04
==========
Total Number of Producers: 584

	State	Number of Producers	Effective Area	Mean Landholding Size
0	1	70	94.680	1.87
1	2	46	74.947	1.70
2	3	16	29.260	1.99
3	4	37	46.159	1.43
4	5	69	74.564	1.40
5	6	93	120.682	1.56
6	7	115	179.784	1.90
7	8	25	29.325	1.24
8	9	23	36.794	1.71
9	10	90	118.067	1.58

Observations

All the states show an increase in average landholding size, and *most* states show slight a downward trend in the number of producers.

Let us consider the data for states 1, 7 and 10.

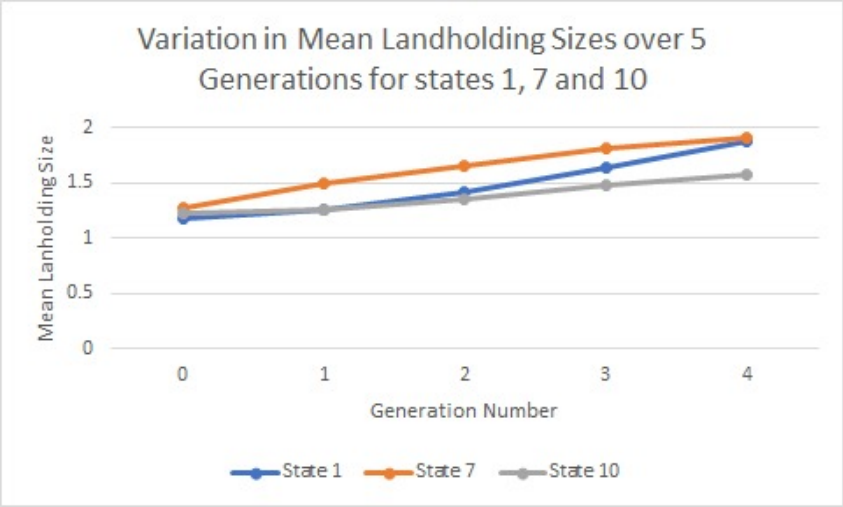


Figure 3: Trends in Mean Landholdings over 5 Generations

All states show a slight variation in the number of producers over the generations, with a mixed trend (here states 1 and 3 on a downward trend and states 7 and 10 fluctuating). State 3 already starting out with few producers sees very little change in their number.

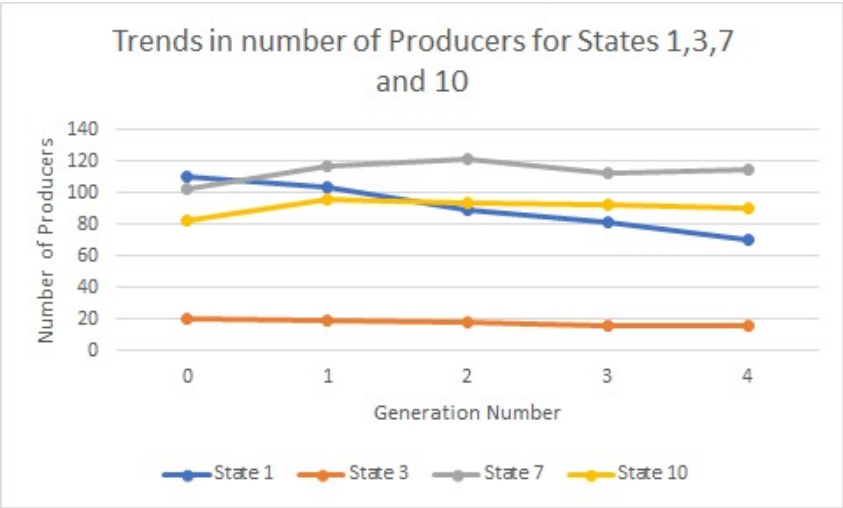


Figure 4: Trends in Number of Producers over 5 Generations

Similar data can be analysed for trends in crop production, prices and farmer incomes. Through our simulation of a hypothetical state we have seen how genetic algorithms can be adopted for running economic simulations.

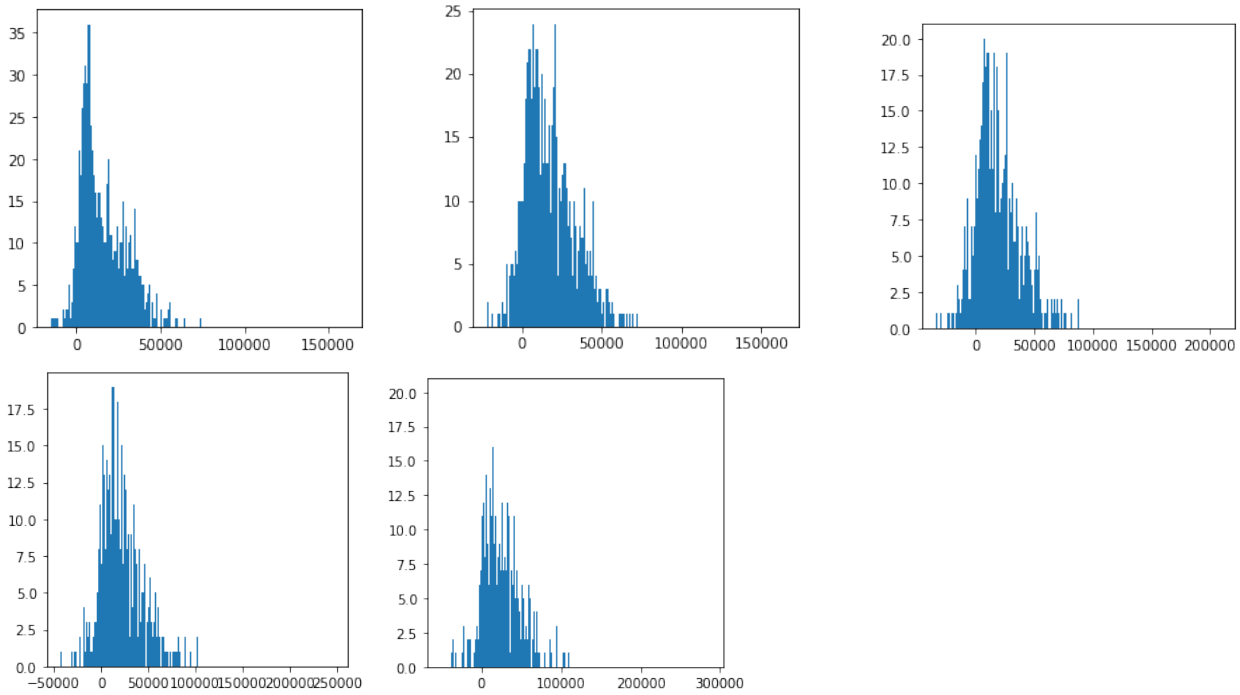


Figure 5: Trends in fitness distribution of entire population over 5 generations

4 Code

4.1 Libraries and Data

[641]:
```
from random import choices, randint, randrange, shuffle
from typing import List, Optional, Callable, Tuple
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import lognorm
import math
```

[642]:
```
import random
def shuffleFunc():  return 0.5
```

[643]:
```
# STATES DATA
state_num = 10 # We are working with 10 states here
# Hypothetical data for the states for the purpose of this term paper:
sizes = [150, 80, 30, 50, 120, 200, 180, 30, 40, 120]
effective_sizes = [130, 70, 30, 40, 80, 150, 130, 20, 40, 100]
neighbours = [[2, 10],[1, 3, 4, 10],[2],[2, 6, 5, 10],[4, 6, 8],[4, 5, 8, 10,␣
 ↪7],\
              [6, 8, 9, 10],[5, 6, 7, 9],[7, 8, 10],[1, 2, 4, 6, 7, 9]]
crops = [[1,1,1], [1,1,1], [0, 1, 0], [1, 1, 1], [1, 0, 1],\
         [1, 1, 1], [1, 1, 1], [1, 0, 1], [1, 0, 1], [1, 1, 1]]
consumer_num = [800, 680, 300, 330, 680, 1200, 2200, 280, 300, 1000]
producer_num = [154, 119, 60, 66, 102, 198, 396, 56, 36, 150]
holdings = []

CropDict = {
    # (max yield per hectare, avg cost per unit, avg units per hectare, units␣
 ↪per consumer)
    'A': (5200, 22500/4500,4500, 3*48),
    'B': (12000, 31000/5000,5000, 3*60),
    'C': (2000,11000/1500,1500, 3*20)
}
# PRODUCERS DATA
# for individual producers (initially) the productivity is assumed to be␣
 ↪normally distributed
mu_prod, sigma_prod = 0.5, 1
```

[644]:
```
consumerDemand = [[600,500,550], [280,400,500], [0, 400, 0], [300, 400, 330],␣
 ↪[640, 0, 600],\
        [1000, 800, 800], [1600, 1800, 1500], [280, 0, 280], [300, 0, 300],␣
 ↪[400, 1000, 800]]
consumerDemand = [[a*CropDict['A'][3],b*CropDict['B'][3],c*CropDict['C'][3]] for␣
 ↪[a,b,c] in consumerDemand]
```

[645]:
```
# Generating holding sizes based on lognormal(mu,sigma) distribution
# parameters of lognormal distribution
mu = -0.17#-0.17489777690658245
sigma = 0.9
for j in range(len(sizes)):
    total = effective_sizes[j]
    tot = 0
    data = [0]*(total+10)
    i = 0
    while tot < total:
        num = round(lognorm.rvs(sigma, size=1)[0] * np.exp(mu),2)
        if num < 0.1: continue
        data[i] = num
        tot += data[i]
        i+=1
    data = data[0:i]
    data[data.index(max(data))] = round(data[data.index(max(data))] -(tot -␣
 ↪total),2)
    producer_num[j] = i
    holdings.append(data)
```

[646]:
```
# State Features
class State:
    def __init__(self, name, available_area, effective_area, neighbours, crops,␣
 ↪consumer_number, producer_number, holdings):
        self.name = name
        self.available_area = available_area # fixed: Geographical
        self.effective_area = effective_area # variable to an extent:␣
 ↪geographical + fffficiency and technology dependent
        self.neighbours = neighbours # fixed: Geographical
        self.crops = crops # fixed: Geographical
        self.consumer_number = consumer_number # fixed: for current purposes
        self.producer_number = producer_number # variable: can vary every␣
 ↪generation
        self.holdings = holdings
```

[647]:
```
S = []
dfList = []
for i in range(state_num):
    state = State(str(i+1), sizes[i], effective_sizes[i],neighbours[i]␣
 ↪,crops[i],consumer_num[i],producer_num[i],holdings[i])
    S.append(state)
    dfList.append([str(i+1), sizes[i], effective_sizes[i],neighbours[i]␣
 ↪,crops[i],\
```

```
                ␣
 ↪consumer_num[i],producer_num[i],min(holdings[i]),max(holdings[i]),np.
 ↪mean(holdings[i])])
```

[648]:
```
df = pd.DataFrame(dfList)
df.columns = ['State','Available Area','Effective Area',␣
 ↪'Neighbours','Crops','Number of Consumers','Number of Producers','Min. Land␣
 ↪Holding','Max. Land Holding','Mean. Land Holding']
total_df = ['Total', df['Available Area'].sum(), df['Effective Area'].
 ↪sum(),'-','-',df['Number of Consumers'].sum(),df['Number of Producers'].
 ↪sum(),'-','-','-']
dfList.append(total_df)
df = pd.DataFrame(dfList)
df.columns = ['State','Available Area','Effective Area',␣
 ↪'Neighbours','Crops','Number of Consumers','Number of Producers','Min. Land␣
 ↪Holding','Max. Land Holding','Mean. Land Holding']

display(df)
```

	State	Available Area	...	Max. Land Holding	Mean. Land Holding
0	1	150	...	7.05	1.18182
1	2	80	...	8.99	1.2963
2	3	30	...	3.78	1.5
3	4	50	...	6.12	1.21212
4	5	120	...	4.29	1.01266
5	6	200	...	7.12	1.35135
6	7	180	...	5.88	1.27451
7	8	30	...	2.7	0.952381
8	9	40	...	4.36	1.33333
9	10	120	...	5.29	1.21951
10	Total	1000	...	-	-

[11 rows x 10 columns]

(*the data here is a hypothetical, scaled down version of what is seen in the Indian Agricultural Market. To arrive at values near the actual values both the land (in ha) and the population need to be scaled up by 10^5.)

4.2 GA

[649]:
```
def ratios(myList):
    if sum(myList)!=0:
        n = sum(myList)
        return [int(10*a/n) for a in myList]
    else:   return [0]*len(myList)
```

[650]:
```
def CostCurve(x,crop):
    minmum = 6
    if crop == 'B': minmum = 8
    if crop == 'C': minmum = 4
    if x <= minmum:
        return 0.025*(x - minmum)**2 + CropDict[crop][1]
    else:
        return 0.75*(x - minmum)**2 + CropDict[crop][1]
```

[651]:
```
def amtProduced(producer, crop):
    # Amount of the crop produced by the farmer
    n = 0
    if crop == 'B': n = 1
    elif crop == 'C': n = 2
    if sum(producer[cropsIdx]) == 0:
        # print("0 here in amt produced")
        return [0]*len(CropDict)
    return round(CropDict[crop][0]*producer[cropsIdx][n]/(np.
 ↪sum(producer[cropsIdx]))*producer[holdingIdx]*producer[productivityIdx],1)
```

[652]:
```
def prodCost(producer, crop):
    # Cost of producing a crop for a producer
    if crop == 'A': n = 0
    elif crop == 'B': n = 1
    else: n = 2
    if sum(producer[amtProdIdx]) == 0:  return [0]*len(CropDict)
    return round(producer[amtProdIdx][n]*CropDict[crop][1])
```

[653]:
```
def moneyFunc(holdingSize):
    x = (randint(25,100)/100*(CropDict['A'][1]*CropDict['A'][2])+randint(25,100)/
 ↪100*(CropDict['B'][1]*CropDict['B'][2])+randint(25,100)/
 ↪100*(CropDict['C'][1]*CropDict['C'][2]))/3
    if holdingSize < 1:
        return holdingSize*x*randint(1,3)
    if holdingSize < 8:
        return holdingSize*x*randint(3,5)
    return holdingSize*x*randint(3,10)
```

[654]:
```
stateIdx = 0
holdingIdx = 1
cropsIdx = 2
productivityIdx = 3
moneyIdx = 4
profitEmphIdx = 5
amtProdIdx = 6
costsIdx = 7
moneyEarnedIdx = 8
```

[655]:
```
# this function generates the initial population
def generateProducers(state_num: int):
    producers = []
    for i in range(S[state_num].producer_number):
        # producer is identified by [location, holding size, proportions of␣
↪crops grown (division of land holding among crops),
        #                            productivity, money available for␣
↪investment,
        #                            emphasis on profit,  amt produced, total␣
↪cost]
        producers.append([state_num,
                          S[state_num].holdings[i],
                          [randint(0,10)*crop for crop in S[state_num].crops],
                          round(1 - abs(np.random.normal(mu_prod, sigma_prod,␣
↪None))%1,4),
                           round(moneyFunc(S[state_num].holdings[i])),
                          round(1 - abs(np.random.normal(mu_prod, sigma_prod,␣
↪None))%1,4)])
        if np.sum(producers[i][cropsIdx]) == 0:
            producers[i][cropsIdx] = [randint(1,10)*crop for crop in␣
↪S[state_num].crops]
        producers[i]+=[[amtProduced(producers[i], crop) for crop in␣
↪['A','B','C']]]
        producers[i]+=[[prodCost(producers[i], crop) for crop in ['A','B','C']]]
        producers[i]+=[[0]*len(CropDict)]
    return producers
```

4.2.1 Generating Initial Population

[656]:
```
# This calls the generate_genome function as many times as required
def generate_population():
    total_pop = []
    for j in range(state_num):
        total_pop+=generateProducers(j)
    return total_pop
```

[657]:
```
def getStatePrices(population, stateNum):
    priceList = np.array([0]*len(CropDict))
    totalUnits = np.array([0]*len(CropDict))
    semiPop = list(filter(lambda member: member[stateIdx] == stateNum ,␣
↪population))
    for member in semiPop:
        priceList = priceList + (1+member[profitEmphIdx])*np.
↪array(member[costsIdx])
        totalUnits =  totalUnits + np.array(member[amtProdIdx])
    for i in range(len(CropDict)):
```

```
        if totalUnits[i] == 0:  continue
        priceList[i] = priceList[i]/totalUnits[i]
    return np.round(priceList,2)
```

[658]:
```
def marketSale(population):
    # STEP 1: PRICE DETERMINATION: STATE-WISE
    prices = []
    for i in range(state_num):
        prices.append(getStatePrices(population,i))
    # STEP 2: SELL
    # 1st LEVEL SALE: producers of a state sell ONLY to consumers within their␣
↪state
    population = sorted(population, key=lambda producer:␣
↪producer[productivityIdx], reverse=True)
    # The most efficient producers will sell first
    currDemand = np.array(consumerDemand[:])
    for producer in population:
        amtSold = np.array(producer[amtProdIdx])
        currDemand[producer[stateIdx]] = currDemand[producer[stateIdx]] - amtSold
        for i in range(len(currDemand[producer[stateIdx]])):
            if currDemand[producer[stateIdx]][i] < 0:
                amtSold[i] = amtSold[i] -currDemand[producer[stateIdx]][i]
                currDemand[producer[stateIdx]][i] = 0
        amtSold = amtSold - np.array(producer[amtProdIdx])
        tempAmtRemaining = amtSold # amount of the crops left with the consumer
        amtSold = np.array(producer[amtProdIdx]) - amtSold # the actual amt sold
        amtEarned = np.array(prices[producer[stateIdx]])*amtSold
        producer[amtProdIdx] = tempAmtRemaining
        producer[moneyEarnedIdx] = amtEarned

    # 2nd LEVEL SALE: producers left with produce want to get rid of their stock
    # Can only sell to consumers in their neighbouring states (in this model)
    stateCount = 0
    for stateDemand in currDemand:
        for i in range(len(stateDemand)):
            if round(stateDemand[i]) > 0:
                candidates = list(filter(lambda neighbourProd:␣
↪((neighbourProd[stateIdx]+1 in S[stateIdx].neighbours) and␣
↪(neighbourProd[amtProdIdx][i]> 0.)), population))
                # print(count)
                # print(candidates)
                candidates = sorted(candidates, key=lambda producer:␣
↪producer[productivityIdx], reverse=True)
                for producer in candidates:
                    if producer[amtProdIdx][i] <= stateDemand[i]:
                        stateDemand[i]-=producer[amtProdIdx][i]
```

```
                        producer[moneyEarnedIdx][i] +=␣
↪prices[stateCount][i]*producer[amtProdIdx][i]
                        producer[amtProdIdx][i] = 0
                    else:
                        producer[moneyEarnedIdx][i] +=␣
↪prices[stateCount][i]*stateDemand[i]
                        producer[amtProdIdx][i] -= stateDemand[i]
                        stateDemand[i] = 0
        stateCount+=1
    return population, currDemand, prices
```

4.2.2 Fitness Calculation

[659]:
```
def fitness(producer):
    # (0.75*accumulated_growth +0.25*profit)/(1+holding_size)
    #  = (initial capital + Revenue - Cost)/(1+holding_size)
    fitnessAmt = (0.75*producer[moneyIdx] + sum(producer[moneyEarnedIdx]) -␣
↪sum(producer[costsIdx]))/(1+producer[holdingIdx])
    return fitnessAmt
```

4.2.3 Crossovers

[660]:
```
def TwoOneCrossOver(p1,p2):
    if len(p1) != len(p2):
        raise ValueError("Genomes p1 and p2 must be of same length")
    if p1[stateIdx]!=p2[stateIdx]:
        raise ValueError("p1, p2 must belong to the same state")
    newP = [p1[stateIdx],\
            p1[holdingIdx]+p2[holdingIdx],\
           ␣
↪[randint(min(p1[cropsIdx][i],p2[cropsIdx][i]),max(p1[cropsIdx][i],p2[cropsIdx][i]))␣
↪for i in range(len(p1[cropsIdx]))],\
             min(p1[productivityIdx],p2[productivityIdx])+random.
↪random()*(max(p1[productivityIdx],p2[productivityIdx])-min(p1[productivityIdx],p2[productivit
             p1[moneyIdx]+p2[moneyIdx],\
             min(p1[profitEmphIdx],p2[profitEmphIdx])+random.
↪random()*(max(p1[profitEmphIdx],p2[profitEmphIdx])-min(p1[profitEmphIdx],p2[profitEmphIdx])),
             p1[amtProdIdx]+p2[amtProdIdx],\
             p1[costsIdx]+p2[costsIdx],\
             p1[moneyEarnedIdx]+p2[moneyEarnedIdx]]
    return newP
```

[661]:
```
def TwoTwoCrossOver(p1,p2):
    if len(p1) != len(p2):
        raise ValueError("Genomes p1 and p2 must be of same length")
    newP1, newP2 = p1[:], p2[:]
```

```
    if fitness(p1) < fitness(p2):
        newP1[cropsIdx]  =␣
↪[randint(min(p1[cropsIdx][i],p2[cropsIdx][i]),max(p1[cropsIdx][i],p2[cropsIdx][i]))␣
↪for i in range(len(p1[cropsIdx]))]
    else:
        newP2[cropsIdx]  =␣
↪[randint(min(p1[cropsIdx][i],p2[cropsIdx][i]),max(p1[cropsIdx][i],p2[cropsIdx][i]))␣
↪for i in range(len(p1[cropsIdx]))]
    newP1[profitEmphIdx] = min(p1[profitEmphIdx],p2[profitEmphIdx])+random.
↪random()*(max(p1[profitEmphIdx],p2[profitEmphIdx])-min(p1[profitEmphIdx],p2[profitEmphIdx]))
    newP2[profitEmphIdx] = min(p1[profitEmphIdx],p2[profitEmphIdx])+random.
↪random()*(max(p1[profitEmphIdx],p2[profitEmphIdx])-min(p1[profitEmphIdx],p2[profitEmphIdx]))

    return newP1, newP2
```

4.2.4 Mutation

[662]:
```
def selection_pair(population):
    return choices(
        population=population,
        weights=[fitness(gene) for gene in population],
        k=2)
```

[663]:
```
def mutation(producer, population):
    # holding size change
    if random.random()<0.25:
        # holding size may decrease
        if random.random()<0.1 and producer[holdingIdx] > 2:
            temp = producer[holdingIdx]
            producer[holdingIdx] = randint(1,10)/10*producer[holdingIdx]
            S[producer[stateIdx]].effective_area+= temp-producer[holdingIdx]
        # holding size may decrease
        elif S[producer[stateIdx]].available_area > S[producer[stateIdx]].
↪effective_area:
            prop = round(randint(1,10)/10,1)
            temp = prop*producer[holdingIdx]
            if temp < S[producer[stateIdx]].available_area -␣
↪S[producer[stateIdx]].effective_area:
                S[producer[stateIdx]].effective_area -= temp
                producer[holdingIdx] += temp
                producer[moneyIdx] = producer[moneyIdx] - prop*producer[moneyIdx]
    # change in proportion in which crops are grown
    if random.random()<0.3:
        [p1,p2] = selection_pair(list(filter(lambda member: member[stateIdx] ==␣
↪producer[stateIdx], population)))
```

```
        producer[cropsIdx] = [int(0.5*(p1[cropsIdx][i]+p2[cropsIdx][i])) for i␣
↪in range(len(producer[cropsIdx]))]
    if random.random()<0.5:
        producer[productivityIdx] = random.random()
    return producer
```

[664]:
```
def population_fitness(population):
    return [max([fitness(producer) for producer in population]),\
            min([fitness(producer) for producer in population]),\
            np.mean([fitness(producer) for producer in population]),\
            np.mode([fitness(producer) for producer in population]),\
            np.median([fitness(producer) for producer in population])]
```

[665]:
```
def GetStateData(population):
    StatesData = []
    for i in range(state_num):
        tempList = list(filter(lambda member: member[stateIdx] == i, population))
        StatesData+=[[len(tempList),
                 [sum([producer[holdingIdx] for producer in tempList]),
                  round(np.mean([producer[holdingIdx] for producer in␣
↪tempList]),2),
                  round(np.median([producer[holdingIdx] for producer in␣
↪tempList]),2),
                  round(np.min([producer[holdingIdx] for producer in␣
↪tempList]),3),
                  round(np.max([producer[holdingIdx] for producer in␣
↪tempList]),2)],

                  [round(np.mean([producer[productivityIdx] for producer in␣
↪tempList]),4),
                  round(np.median([producer[productivityIdx] for producer in␣
↪tempList]),4),
                  np.min([producer[productivityIdx] for producer in tempList]),
                  round(np.max([producer[productivityIdx] for producer in␣
↪tempList]),2)],

                  [round(sum([producer[moneyIdx] for producer in tempList]),2),
                  round(np.mean([producer[moneyIdx] for producer in␣
↪tempList]),2),
                  round(np.median([producer[moneyIdx] for producer in␣
↪tempList]),2),
                  np.min([producer[moneyIdx] for producer in tempList]),
                  round(np.max([producer[moneyIdx] for producer in␣
↪tempList]),2)],
```

```
                    [round(np.mean([producer[profitEmphIdx] for producer in␣
↪tempList]),2),
                    round(np.median([producer[profitEmphIdx] for producer in␣
↪tempList]),2),
                    np.min([producer[profitEmphIdx] for producer in tempList]),
                    round(np.max([producer[profitEmphIdx] for producer in␣
↪tempList]),2)],

                    [[sum([producer[amtProdIdx][i] for producer in tempList])for i␣
↪in range(len(CropDict))],
                    np.round(np.array([sum([producer[amtProdIdx][i] for producer␣
↪in tempList])for i in range(len(CropDict))])/len(tempList),2),
                    [np.median([producer[amtProdIdx][i] for producer in tempList])␣
↪for i in range(len(CropDict))],
                    [min([producer[amtProdIdx][i] for producer in tempList]) for i␣
↪in range(len(CropDict))],
                    [max([producer[amtProdIdx][i] for producer in tempList]) for i␣
↪in range(len(CropDict))]],

                    [[sum([producer[costsIdx][i] for producer in tempList])for i␣
↪in range(len(CropDict))],
                    np.round(np.array([sum([producer[costsIdx][i] for producer in␣
↪tempList])for i in range(len(CropDict))])/len(tempList),2),
                    [np.median([producer[costsIdx][i] for producer in tempList])␣
↪for i in range(len(CropDict))],
                    [min([producer[costsIdx][i] for producer in tempList]) for i␣
↪in range(len(CropDict))],
                    [max([producer[costsIdx][i] for producer in tempList]) for i␣
↪in range(len(CropDict))]],

                    [[sum([producer[moneyEarnedIdx][i] for producer in␣
↪tempList])for i in range(len(CropDict))],
                    np.round(np.array([sum([producer[moneyEarnedIdx][i] for␣
↪producer in tempList])for i in range(len(CropDict))])/len(tempList),2),
                    [np.median([producer[moneyEarnedIdx][i] for producer in␣
↪tempList])for i in range(len(CropDict))],
                    [np.min([producer[moneyEarnedIdx][i] for producer in␣
↪tempList])for i in range(len(CropDict))],
                    [np.max([producer[moneyEarnedIdx][i] for producer in␣
↪tempList])for i in range(len(CropDict))]]
                ]]
    return StatesData
```

[666]:
```
def print_stats(population, currDemand, generation_id):
    print("GENERATION %02d" % generation_id,'\n=============')
    print("Total Number of Producers: ", len(population))
```

```
    # STATE WISE DATA
    stateData = GetStateData(population)
    # print(stateData)
    dfList = []
    for i in range(state_num):
        dfList+=[[i+1, stateData[i][0],S[i].effective_area,stateData[i][1][1]]]
        # print("STATE:",i+1)
        # print("Number of producers:", stateData[i][0])
        # print("Mean Landholding Size:", stateData[i][1][1])
        # print("Mean Costs (for each crop):", stateData[i][6][1])
        # print("Mean Amount earned (for each crop):", stateData[i][7][1])
        # print("Mean profit over costs incurred:", np.
↪array(stateData[i][7][1])-np.array(stateData[i][6][1]))
        # print("Percentage of demand met in state", [],"(for crops A, B , C␣
↪respectively)")
        # print('_____________')
    df = pd.DataFrame(dfList)
    df.columns = ['State','Number of Producers','Effective Area','Mean␣
↪Landholding Size']
    display(df)
    print("Overall Population Fitness Distribution:")
    # FITNESS PLOT
    fitList = [fitness(producer) for producer in population]
    fitList = np.array(fitList)
    # Creating histogram
    fig, ax = plt.subplots(figsize =(4, 4))
    ax.hist(fitList, bins = [math.floor(min(fitList))+1000*i for i in␣
↪range(int(math.ceil((max(fitList)-min(fitList))/500)))])
    # Show plot
    plt.show()
    return stateData
```

4.2.5 Evolution

[667]:
```
def flatten(myList):
    flatList = []
    for x in myList:
        if type(x) == list or type(x) == np.ndarray or type(x) == np.array:
            for y in x:
                flatList.append(y)
        else:
            flatList.append(x)
    # print(flatList)
    return flatList
```

[668]:
```
def compareLists(l1,l2):
    if len(l1)!=len(l2):    return 0
    templ1 = flatten(l1)
    templ2 = flatten(l2)
    for i in range(len(templ1)):
        if templ1[i] != templ2[i]:  return 0
    return 1
```

[669]:
```
def run_evolution(generation_limit: int = 5): # number of generations to the␣
↪simulation for
        currGen = 0
        # GENERATING THE INITIAL POPULATION
        population = generate_population()

        while currGen < generation_limit:
            # MARKET SALE
            population = sorted(population, key=lambda producer:␣
↪producer[stateIdx], reverse=False)
            population, currDemand, prices = marketSale(population)
            print_stats(population, currDemand, currGen)

            # We now sort the population in decreasing order of fitness
            # population = sorted(population, key=lambda producer:␣
↪fitness(producer), reverse=True)
            partHighFitPop = list(filter(lambda producer: fitness(producer) >=0␣
↪, population))
            partHighFitPop = sorted(partHighFitPop, key=lambda producer:␣
↪fitness(producer), reverse=True)
            partLowFitPop = list(filter(lambda producer: fitness(producer) < 0 ,␣
↪population))

            newPopulation = []
            # Dealing with first half members first:
            topHalf = partHighFitPop[:int(len(partHighFitPop)/2)]
            bottomHalf = partHighFitPop[int(len(partHighFitPop)/2):]

            for producer in topHalf:
                producer2 = []
                if random.random()<0.1:
                    # 2-1 crossover
                    eligibleProds = list(filter(lambda member: member[stateIdx]␣
↪== producer[stateIdx], bottomHalf+partLowFitPop))
                    producer2 = choices(eligibleProds,
                                        weights=[abs(1/(1+fitness(p2))) for p2␣
↪in eligibleProds],
                                        k=1)[0]
```

```
                    # print(producer2)##########
                    if any(compareLists(currList,producer2) for currList in␣
↪partLowFitPop):
                        try: partLowFitPop.remove(producer2)
                        except ValueError: pass  # do nothing!
                    elif any(compareLists(currList,producer2) for currList in␣
↪bottomHalf):
                        try: bottomHalf.remove(producer2)
                        except ValueError: pass  # do nothing!
                    producer = TwoOneCrossOver(producer, producer2)
                    S[producer2[stateIdx]].producer_number -=1
                    producer2 = []
                elif random.random()<0.05:
                    # 2-2 cross over
                    producer2 = choices(population,
                                        weights=[fitness(p2) for p2 in␣
↪population],k=1)[0]
                    if producer2 != producer:
                        # print(producer2)##########
                        if any(compareLists(currList,producer2) for currList in␣
↪partLowFitPop):
                            try: partLowFitPop.remove(producer2)
                            except ValueError: pass  # do nothing!
                        elif any(compareLists(currList,producer2) for currList␣
↪in bottomHalf):
                            try: bottomHalf.remove(producer2)
                            except ValueError: pass  # do nothing!
                        producer, producer2 = TwoTwoCrossOver(producer,producer2)
                elif random.random()<0.01:
                    producer = mutation(producer, population)
                producer[moneyIdx] = producer[moneyIdx] +␣
↪sum(producer[moneyEarnedIdx]) - sum(producer[costsIdx])
                producer[costsIdx] = [0]*len(CropDict)
                producer[moneyEarnedIdx] = [0]*len(CropDict)
                newPopulation += [producer]
                if producer2 != []:
                    producer2[moneyIdx] = producer2[moneyIdx] +␣
↪sum(producer2[moneyEarnedIdx]) - sum(producer2[costsIdx])
                    producer2[costsIdx] = [0]*len(CropDict)
                    producer2[moneyEarnedIdx] = [0]*len(CropDict)
                    newPopulation += [producer2]

            random.shuffle(bottomHalf)
            for j in range(int(len(bottomHalf)/2)):
                producer, producer2 = bottomHalf[j],␣
↪bottomHalf[len(bottomHalf)-j-1]
```

```
                # Crossovers
                if random.random()<0.2:
                    producer, producer2 = TwoTwoCrossOver(producer, producer2)
                elif random.random()<0.15 and␣
↪producer[stateIdx]==producer2[stateIdx]:
                    producer = TwoOneCrossOver(producer, producer2)
                    S[producer2[stateIdx]].producer_number -=1
                    producer2 = []
                elif random.random()<0.3:
                    producer, producer2 = mutation(producer,population),␣
↪mutation(producer2,population)
                producer[moneyIdx] = producer[moneyIdx] +␣
↪sum(producer[moneyEarnedIdx]) - sum(producer[costsIdx])
                producer[costsIdx] = [0]*len(CropDict)
                producer[moneyEarnedIdx] = [0]*len(CropDict)
                newPopulation += [producer]
                if producer2 != []:
                    producer2[moneyIdx] = producer2[moneyIdx] +␣
↪sum(producer2[moneyEarnedIdx]) - sum(producer2[costsIdx])
                    producer2[costsIdx] = [0]*len(CropDict)
                    producer2[moneyEarnedIdx] = [0]*len(CropDict)
                    newPopulation += [producer2]
            if len(bottomHalf)%2!=0:
                # border case, randomly letting it enter new population without␣
↪any altering
                newPopulation += [bottomHalf[int(len(bottomHalf)/2)]]

            random.shuffle(partLowFitPop)
            for j in range(int(len(partLowFitPop)/2)):
                producer, producer2 = partLowFitPop[j],␣
↪partLowFitPop[len(partLowFitPop)-j-1]
                # Crossovers
                if random.random()<0.5:
                    producer, producer2 = TwoTwoCrossOver(producer, producer2)
                elif random.random()<0.5 and␣
↪producer[stateIdx]==producer2[stateIdx]:
                    producer = TwoOneCrossOver(producer, producer2)
                    S[producer2[stateIdx]].producer_number -=1
                    producer2 = []
                elif random.random()<0.5:
                    producer, producer2 = mutation(producer,population),␣
↪mutation(producer2,population)
                else:
                    # If the weak members of the population do not undergo any␣
↪change
```

```
                    # they are removed from the population and their␣
↪landholdings are removed from the effective area
                    # This increases the amount of land avaiable for sale␣
↪(available_area-effective_area increases)
                    S[producer[stateIdx]].effective_area -= producer[holdingIdx]
                    S[producer2[stateIdx]].effective_area -=␣
↪producer2[holdingIdx]
                    producer, producer2 = [],[]
                if producer != []:
                    producer[moneyIdx] = producer[moneyIdx] +␣
↪sum(producer[moneyEarnedIdx]) - sum(producer[costsIdx])
                    producer[costsIdx] = [0]*len(CropDict)
                    producer[moneyEarnedIdx] = [0]*len(CropDict)
                    newPopulation += [producer]
                if producer2 != []:
                    producer2[moneyIdx] = producer2[moneyIdx] +␣
↪sum(producer2[moneyEarnedIdx]) - sum(producer2[costsIdx])
                    producer2[costsIdx] = [0]*len(CropDict)
                    producer2[moneyEarnedIdx] = [0]*len(CropDict)
                    newPopulation += [producer2]
            if len(partLowFitPop)%2!=0:
                # border case, randomly letting it enter new population without␣
↪any altering
                newPopulation += [partLowFitPop[int(len(partLowFitPop)/2)]]

            # New entrants to population
            # Will be dealt with State-Wise
            # Depends on: 1. Is there excess demand to be met? 2. Is it␣
↪profitable to enter? 3. Is there land left?
            count = 0
            for i in range(state_num):
                if sum(currDemand[i]) > 0 and S[i].available_area-S[i].
↪effective_area:
                    tempList = list(filter(lambda producer: producer[stateIdx]␣
↪== i and fitness(producer)>0 , population))
                    for j in range(int(len(tempList)*0.25)):
                        if random.random() < 0.1: continue
                        count+=1
                        producer = choices(tempList, weights=[fitness(p2) for p2␣
↪in tempList],k=1)[0]
                        if S[i].available_area-(S[i].effective_area +␣
↪producer[holdingIdx]) <=0:  continue
                        S[i].effective_area += producer[holdingIdx]
                        if random.random() < 0.75:
                            producer[cropsIdx] = np.
↪array(ratios(currDemand[i]))*np.array(producer[cropsIdx])
```

```
                        if sum(producer[cropsIdx])==0:
                            producer[cropsIdx] = np.
↪array(ratios(currDemand[i]))*10
                        producer[productivityIdx] = round(1 - abs(np.random.
↪normal(mu_prod, sigma_prod, None))%1,4)
                        producer[moneyIdx] =␣
↪round(moneyFunc(producer[holdingIdx]))
                        producer[profitEmphIdx] = round(1 - abs(np.random.
↪normal(mu_prod, sigma_prod, None))%1,4)
                        producer[amtProdIdx] = [0]*len(CropDict)
                        producer[costsIdx] = [0]*len(CropDict)
                        producer[moneyEarnedIdx] = [0]*len(CropDict)
                        newPopulation+=[producer]
            population = newPopulation
            # NEW PRODUCTION CYCLE
            for producer in population:
                producer[amtProdIdx] = [amtProduced(producer, crop) for crop in␣
↪['A','B','C']]
                producer[costsIdx] = [prodCost(producer, crop) for crop in␣
↪['A','B','C']]
                producer[moneyEarnedIdx] = [0]*len(CropDict)
            currGen+=1
```

[670]:
```
run_evolution(5)
```

5 Conclusion: Next Steps- Scaling Up to Real Life Data data

This paper is my first attempt at market modelling and simulating behaviour of economic entities. Although the consumer population was an 'imaginary' and static population, later versions can also incorporate a population of consumers, as an 'invisible population', i.e. a number with a given growth factor and distribution of traits which evolves in a predetermined way. This 'invisible population' will not take part in any evolutionary procedures but will interact with the evolving population of the producers and determine their behaviour. Different genetic algorithms can also be used in parallel with one another to model different kinds of individuals in a population. For example, *Differential Evolution*(DE) or *Particle Swarm Optimisation*(PSO) techniques can be used to model the populations of producers while an SGA can be used to model the consumers. In this simulation too, though the underlying algorithmic technique was that of the simple genetic algorithm, the mutation and crossovers used here did include methods resembling those used in DE and PSO.

Another promising aspect to consider is that of modelling agricultural labour. Though land owners are bound by the location of their land holdings, the same does not apply to the labour hired by them to work the farms. For modelling agricultural labourers, a PSO/DE derived technique can be used. The migration of labour can be modelled through this. The producers in a particular state need to balance the objectives of having cheap labour to cultivate their land and to maximise their outputs and profits. The labour would migrate to those states where the pay is highest, but

their state preferences may also be determined by regional, linguistic and other factors which can be included as state features and can act as determinants of individual behaviour. The interaction of these two sets of populations with conflicting motives play a crucial role in real life scenarios and an realistic model can help predict the impact of different decision making strategies. Incorporating game-theory derived concepts of trade-offs can help aid the model too.

The time taken to run the current model was around half a minute for 5 generations was just under a minute for 10. If we wish to scale up the population sizes to make them comparable to actual numbers the time complexity of the above proposed models would may be quite a lot. However, the impact of this can be allayed by the few number of generations that the model is run for since each generation is meant to model a single production cycle and to attempt to model the very long run in economics is futile. The technique would however be quite space intensive.

The potential of genetic algorithms to model socio-economic phenomenon deserves a deeper exploration so that it can be deployed to predict and solve scaled up real-life situations.

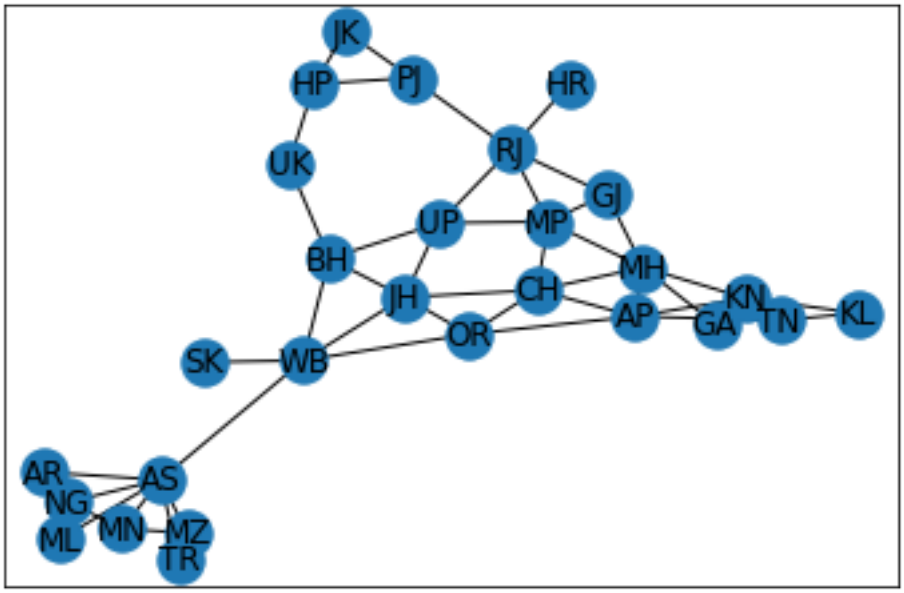


Figure 6: The network graph of political map of India.

Appendix

A Microeconomics

A.1 Microeconomics

A.1.1 Demand, Supply and Elasticity

The concepts of demand and supply form the basis of modern economics.

Demand for a commodity refers to the quantities of a commodity that consumers are willing and able to purchase at various possible prices during a particular period of time.

The market demand for a commodity is determined by the price of the commodity, the demo-

graphic profile (income distribution of consumers, tastes and preferences of consumers etc), prices of related goods (substitute goods and complementary goods), government policy etc.

> **Law of Demand:** The law of demand states that other things remaining constant, the quantity demanded of a commodity increases when its price falls and decreases when its price rises. The assumption that all factors other than price remain constant is called *'ceteris paribus'*.

Supply of a commodity refers to the quantities of a commodity which producers/sellers are willing to produce and offer for sale at various prices during a particular period of time.

The supply for a commodity is determined by the price of the commodity, the goals of the producers, prices of inputs, the4 techniques of production, future price expectations, natural factors, nature of the industry, agreements between producers, government policy etc.

> **Law of Supply:** The law of supply states that other things remaining constant (*ceteris paribus*), the quantity of any commodity that producers are willing to produce and offer for sale rises with a rise in its price and falls with a fall in its price.

A.1.2 Time Frame

In Economics time is divided into three stages:

1. Short Run: Here price acts as the sole determinant of demand and supply. It is too short a period of time for any other factor to vary. This may range from a few days to a few months.
2. Long Run: Here most factors are variable (the amount of resources available, number of producers etc). This period can last from a few months to a few years.
3. Very Long Run

The very long run is quite unpredictable so we shall restrict our model to the long run. Considering each generation of the GA to be equivalent to one crop cycle, we can run the code for upto 10 generations and expect useful output.

Elasticity of demand/ supply is the degree of responsiveness of the quantity demanded/supplied of a commodity to a change in any of its determinants.

Price elasticity refers to the degree of responsiveness of the quantity demanded/supplied of a commodity to a change in price of the commodity. The broad categories of elasticity 'e' are

1. Perfectly elastic (quantity demanded/ supplied responds by an infinite amount to a very small change in price, $e = \infty$)
2. Relatively elastic (the percentage change in quantity demanded/supplied > percentage change in price, $e > 1$)
3. Relatively inelastic (the percentage change in quantity demanded/supplied < percentage change in price, $e < 1$)
4. Perfectly inelastic (quantity demanded/ supplied does not respond to any change in price, $e = 0$)

References

Theory

Genetic Algorithms

[1] Alfons Balmann & Kathrin Happe, *Applying Parallel Genetic Algorithms to Economic Problems: The Case of Agricultural Land Markets.*

[2] Ludo Waltman, Nees Jan van Eck, Rommert Dekker & Uzay Kaymak, *Economic modeling using evolutionary algorithms: the effect of a binary encoding of strategies.*

[3] Herbert Dawid & Michael Kopel , *On economic applications of the genetic algorithm: a model of the cobweb type.*

[4] John H. Holland, *Computer programs that "evolve" in ways that resemble natural selection can solve complex problems even their creators do not fully understand:* https://www2.econ.iastate.edu/tesfatsi/holland.gaintro.htm .

[5] Edmund Chattoe, *Just How (Un)realistic are Evolutionary Algorithms as Representations of Social Processes?* Journal of Artificial Societies and Social Simulation vol. 1, no. 3, 1998. http://jasss.soc.surrey.ac.uk/1/3/2.html.

[6] Franz Rothlauf, Daniel Schunk,Jella Pfeiffer, *Classification of Human Decision Behavior: Finding Modular Decision Rules with Genetic Algorithms* January 23, 2005. https://www.researchgate.net/publication/23752988_Classification_of_Human_Decision_Behavior_Finding_Modular_Decision_Rules_with_Genetic_Algorithms

Misc.

[7] MarketSim: Computers and Typesetting, https://serc.carleton.edu/sp/library/simulations/examples/example2.html

[8] Census of India 2011, https://censusindia.gov.in/2011census/population_enumeration.html

[9] Jonathan Levin and Paul Milgrom, *Producer Theory,October 2004.* https://web.stanford.edu/~jdlevin/Econ%20202/Producer%20Theory.pdf

[10] Rachel Purdy and Michael Langemeier, *International Benchmarks for Wheat Production.* Center for Commercial Agriculture, Purdue University. https://farmdocdaily.illinois.edu/2018/07/international-benchmarks-for-wheat-production.html.

Code

[11] The code used in this project was written using Daniel Kiedrowski's (Kie Codes) code for solving the knapsack problem as a template due to its simplistic and easy to understand code writing.
Github link: https://github.com/kiecodes/genetic-algorithms/blob/master/algorithms/genetic.py