Sicoe Horia-Alexandru

Bucharest Academy of Economic Studies

Faculty of Economic Cybernetics, Statistics, and Informatics

Evolutionary Programming and Genetic Algorithms

-Project-

Profit Optimisation using a Genetic Algorithm

Bucharest 2018

Contents

I. Statement

Problem #8

Elaborate a project to solve the following problem using a genetic algorithm:

In the process of manufacturing wood logs, a timber factory supplies two types of planks: finished, denoted Prod1, and for use in constructions denoted Prod2.

In order to obtain 1000 units of Prod1, the cutting process lasts 2 hours and the planing process 5 hours. To obtain 1000 units of Prod2, the cutting process lasts 2 hours and the planing process 3 hours.

The industrial saw blade can be used up to 8 hours/day, and the planer is available 15 hours/day.

If the profit obtained from the production 1000 units of Prod1 is 120 lei, and from 1000 units of Prod2 is 100 lei, what is the quantity of each type of plank, expressed in thousands of units, that must be produced daily in order to maximise the factory's profit?

II. Algorithm description

Genetic Algorithm schema

- Initialisation: Generate an initial population *pop*
- Evaluation: Compute fitness for each individual x ∈ *pop*
- Evolution:

  t ← 1

  Repeat

  - *Mating selection:* compute the selected parents *selPop* ⊂ *pop*
  - *Crossover:* Let *O* be the offspring population. Compute *O* by applying crossover to pairs of individuals from *selPop*, with a probability of crossover *pc*.
  - *Mutation:* Let *MO* be the mutated population. Compute *MO* by mutating each individual from *O*, with a probability of mutation *pm*.
  - *Evaluation:* Evaluate the fitness of each individual mx ∈ *MO*
  - *Survival selection:* select the next generation *pop* using the current *pop* and *MO*.
  - t ← t+1

  Until a stop condition is fulfilled

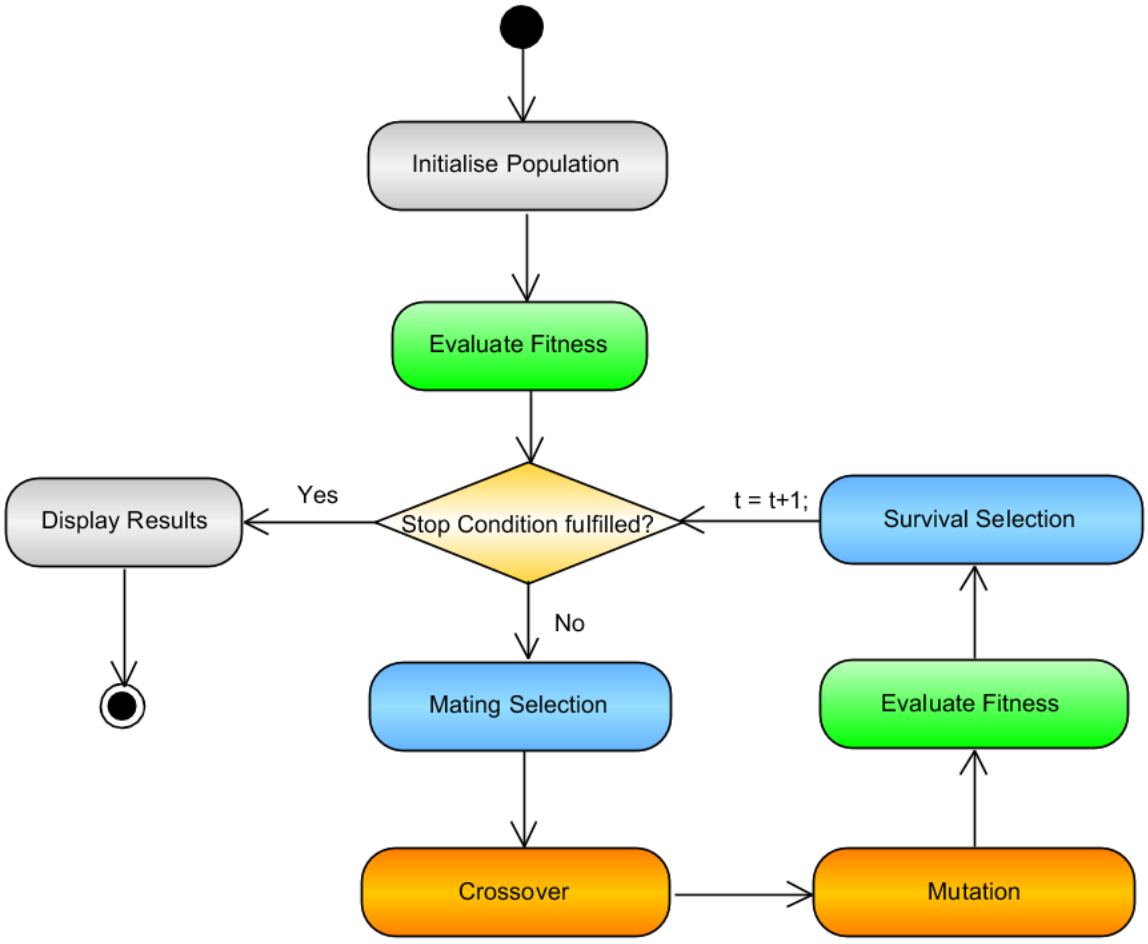

Figure II.1

Stop condition examples

- ➔ t reaches an inputted value for the number of generations to be computed.
- ➔ Convergence is achieved. The algorithm is stuck at a a certain fitness value and can't find higher fitness values for an inputted number of generations.
- ➔ An inputted fitness value is achieved.

III. Statement analysis

Fitness Function

$$f(x_1, x_2) = 120 * x_1 + 100 * x_2$$

where x_1 and x_2 are the quantities of product 1 and product 2, respectively.

In this case, the fitness value represents the total profit in Romanian lei.

The goal is to find a combination of feasible quantities which can maximise the profit.

Phenotype Representation

The 1st and 2nd genes represent the product quantities in thousands for Prod1 and Prod2 respectively.

The representation of choice are real positive numbers, because using integers gives a smaller possible profit.

Also, each gene should not be bigger than a certain value that depletes all resources of an operation, even if the product were to be produced alone. The function findMaxGeneValues() takes care of this aspect.

Constraints

Each tool can be used daily for a limited amount of time; 8 or 15 hours/day, respectively. Each operation requires both tools, in different amounts of time. Therefore a feasible solution must not exceed these constraints.

The maximisation problem can be further formulated as:

$$Maximise\ f = 120x_1 + 100x_2$$

Subject to:

$$\begin{cases} 2x_1 + 2x_2 \leq 8 \\ 5x_1 + 3x_2 \leq 15 \end{cases}$$

With the help of the linear optimization method, I have determined the maximum value of **f = 430 lei**, represented by the combination:

x_1 = 1.5k

x_2 = 2.5k

While linear optimization is a *direct approach* which gives a concrete answer, the genetic algorithm *handles the problem differently*. Genetic algorithms are not necessarily giving the best solution, but they do aim to give a practical answer within a reasonable time frame.

IV. Implementation

Data Files

The program reads the data from a text file. The data is stored into a global variable in order not to read multiple times from the same file, nor to expand function signatures.

Also, I have generalized the solution, to give the possibility of maximising the profit of any number of products and operations.

File format

- Operations are linked to rows whereas product types are linked to columns.
- Each cell from (1,1) to (m-1, n-1) represents how much hours/day the product needs from a certain operation.
- The m row are profits. The n column represents the time constraints for each operation.
- A 0 is needed in the bottom-right corner for matlab to read the matrix correctly.

File example

```
2   2   8
5   3   15
120 100 0
```

Fitness Function

Call example: *f = fitness(individual);*

Input – qty (an individual, a vector of quantities of each product)

Output – f (fitness in lei)

In order to generalize for any number of products, the fitness function returns the sum of profit(i) * quantity(i).

Feasability Function

fez = checkFez(individual);

This function will be called to check each individual when generating the population, and after executing mutation or crossover. It receives an *individual* and returns a boolean value *fez* which indicates if that individual respects the problem constraints or not.

■ Auxiliary Feasability Function

chosenIndividual = checkFez_decision(newIndividual, oldIndividual);

This function is an auxiliary for the previous function. It's called inside the mutation and crossover functions. Instead of writing the same instruction in both of these variation operators, I chose to respect clean code principle and write only once.

It decides if the new individual is feasible. If it is, he will be chosen. If not, the old individual (which was already proved to be feasible) will be chosen. Obviously this method impacts the actual percentages of the mutation and crossover (in practice, the percentages are almost always lower than the inputted ones), but it ensures only feasible solutions. An alternative would be to surround checkFez's call with a while loop (i.e. Compute every mutation until feasible), thus maintaining the mutation/crossover percentages, with the downside of a longer execution time.

Function that finds maximum gene values

Call example: *maxValues = findMaxGeneValues();*

Input - *Needs no input*, as it reads from the *data* global variable.

Output – maxVal (a vector of maximum values for each gene)

In our particular case, there is no point to produce any more than min(8/2, 15/5) = 3 thousand pieces of Prod1. Any more than that would deplete resources even if Prod1 would be produced alone. A higher value would be filtered anyway by the checkFez function.

This function has an optimization role. Without it, we still have the possibility of surrounding checkFez() with a while loop, until the algorithm gives feasible solutions, but that would translate into more instructions to execute. Setting the maximum limits from the start improves the whole process.

Function that Generates Initial Population

Call example: *population = genPop(nInd);*

Input – nInd (number of individuals)

Output – a randomly generated population of feasible solutions.

As stated in the phenotype representation, each allele should not exceed a certain maxValue.

Variation Operators

The variation operators are used to increase the genetic variation of the solutions and bring new blood to the population. Because the representation is on real numbers I have chosen to implement *arithmetic crossover* and *creep mutation*.

■ Crossover

Call example: *O = crossoverPop(P, pc, alfa);*
Input – P = population, pc = probability of crossover, alfa = the weight of parent x with respect to y.
Output – O = offspring population

The function selects parents from the population 2 by 2. If the number of individuals is odd, it will ignore the last individual. If a random number $r \in [0, 1]$ is smaller or equal to the probability of crossover, *crossoverReal()* function is called twice, switching parents. *crossoverReal()* takes care of recombination at the individual's level.

To shuffle the selection, the function generates a permutation based on the number of individuals, and the selected parents have the indices pointed by the permutation. This implies that the parents are no longer selected in the same consecutive manner and in the case of odd number of individuals, the individual left without a pair is not necessarily the last one, but a random one from the population (only the index of its permutation is the last). This also guarantees that for each generation, the pairs are different, leading to a higher variation in the population.

In *crossoverReal() the parents x, y create a new child z based on the formula:*

$$z_i = \text{alfa} * x_i + (1\text{-alfa}) * y_i$$

where x,y are the parents and z the child.
Alfa can either be chosen randomly between 0 and 1 at every call, or it can be fixed to a certain value. A value of alfa = 0.5 obtains uniform arithmetic crossover.

Lastly we check if the child is a feasible solution. If not, the x parent will take its place.

- Mutation

Call example: *MO = mutatePop(O, pm, sigma);*

Input – O = Offspring, pm = probability of mutation, sigma = standard deviation

Output – MO = Mutated Offspring

If a random number r ∈ [0, 1] is smaller or equal with the probability of mutation, then creep mutation at individual level is executed.

In mutationCreep() for each gene, a random normal number R is generated, centered on 0 and with standard deviation sigma. R is added to the gene. There is the possibility that mutated gene exceeds the representation's lower or upper limit. To handle this problem, the gene will be given the value of the exceeded limit.

After mutating the individual's genes we check if the new individual is feasible. If not, the old individual will take its place.

Selection Mechanisms

- Tournament Selection

Call example: *selPop = selectTournament(pop, nParticipants);*

Input – pop = population, k = number of participants

Ouput – selPop = selected population

Tournament is the considered function for mating selection. The function selects k individuals to take part in a tournament. The most fit individual from the tournament is selected. If k is a large value, lower individuals have no chance of being selected.

■ Elitist Selection

Call example: *nextGeneration = selectElitist(pop, MO);*

Input – parents (initial population), children (mutated population)

Output – next generation

Elitism has been chosen for the survival selection. The elitist selection searches for the elite (most fit individual) of the initial population, and compares it with elite of the mutated population. If the first elite is more fit then it replaces the weakest individual from the mutated population. The obtained population becomes the next generation.

V. Results

For a population of 50 individuals, 200 generations,

pc = 0.8,

alfa = 0.5,

pm = 0.2,

sigma = 0.07,

tournamentParticipants = 3,

we get the following results:

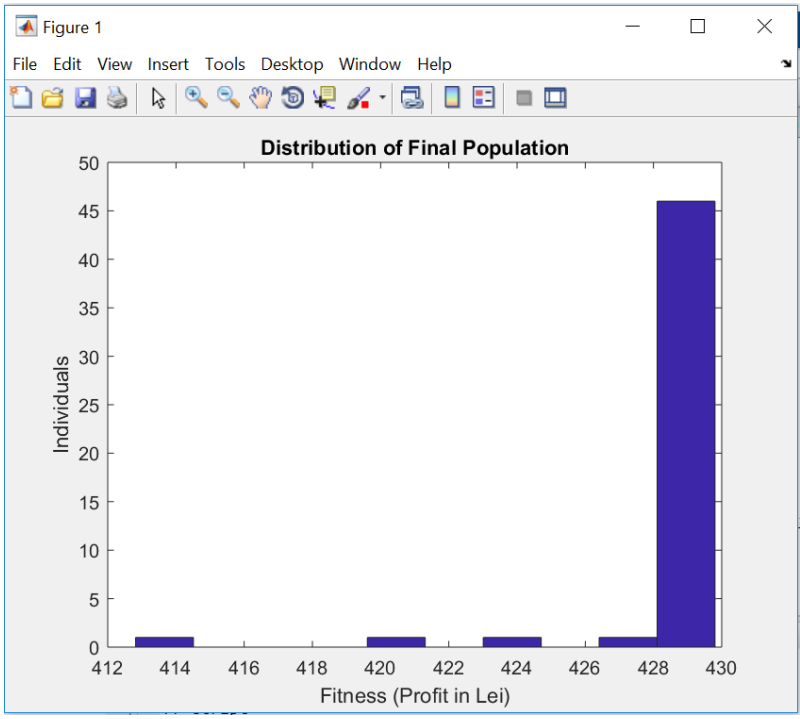

Figure V.1

```
Best candidate from initial population:
    1.8980    1.7915

Fitness: 406.9067
Best candidate from final population:
    1.4927    2.5068

Fitness: 429.804
```

Figure V.2

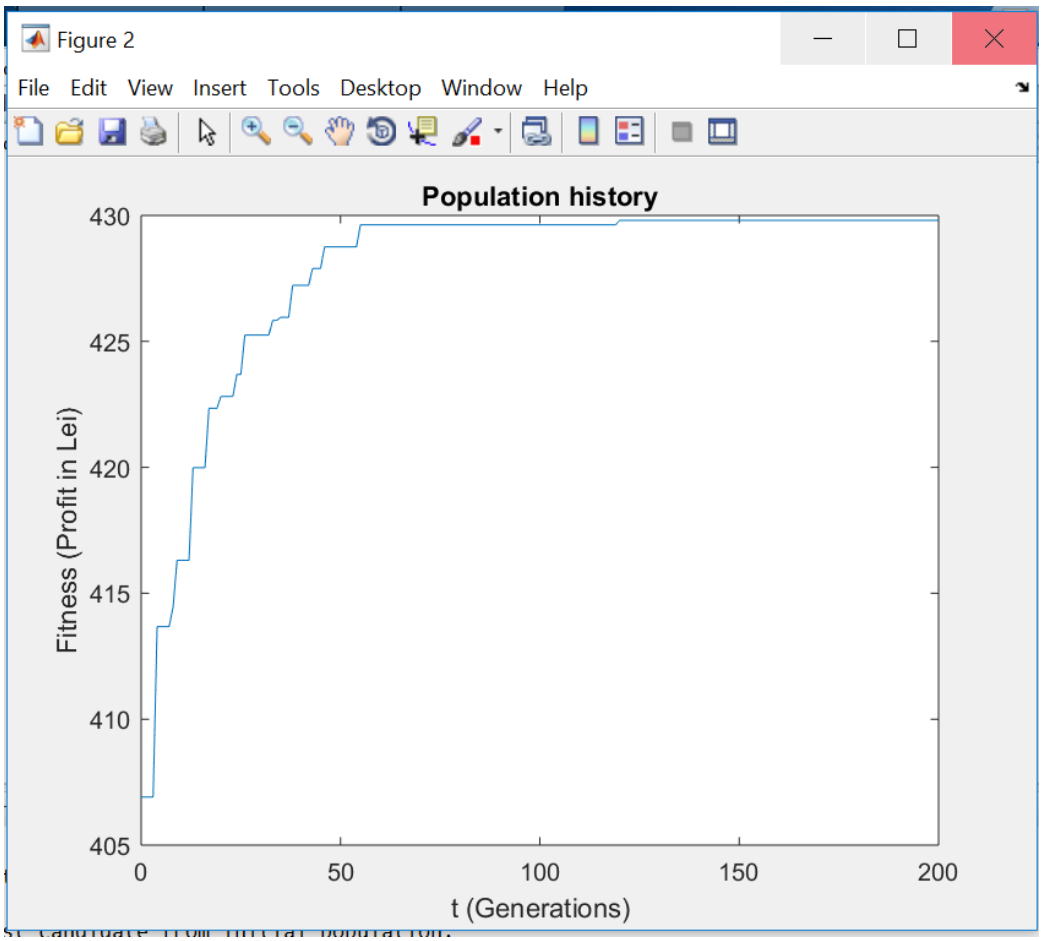

Figure ***Error! No text of specified style in document.****3*

The solution indicates 1.492k pieces of Prod1 and 2.506k pieces of Prod2. Fitness indicates a profit of 429 lei from this combination. For each run, the results might differ.

We can observe that the population distribution has been pushed to higher fitness values (negatively skewed) thanks to the selection mechanisms.