*Sicoe Horia-Alexandru*

*Bucharest Academy of Economic Studies*

*Faculty of Economic Cybernetics, Statistics, and Informatics*

Evolutionary Programming and

Genetic Algorithms

-Project-

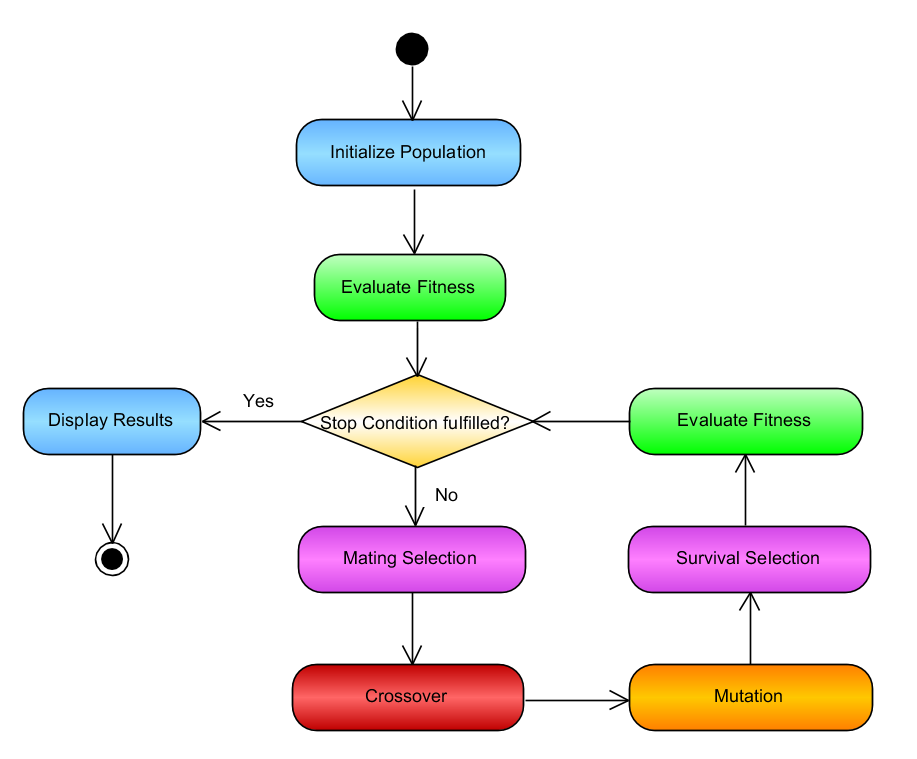
*Profit Optimisation using a Genetic Algorithm*

Bucharest 2018

1. Statement

Problem #8. Elaborați un proiect pentru rezolvarea genetică a următoarei probleme. În procesul de prelucrare a butucilor, o fabrică de cherestea furnizează scândură de două tipuri: finisată, notată cu *Prod1*, şi pentru construcţii, notată cu *Prod2*. Pentru obţinerea a 1000 de unităţi de scândură finisată procedura de tăiere durează 2 ore şi procedura de rindeluire durează 5 ore. Pentru furnizarea a 1000 unităţi de scândură de construcţii, procesul de tăiere durează 2 ore şi cel de rindeluire necesită 3 ore. Fierăstrăul industrial cu care este realizată tăierea poate fi folosit 8 ore pe zi şi rindeaua este disponibilă 15 ore pe zi. Dacă profitul obţinut din producerea a 1000 unităţi de produs este de 120 lei în cazul scândurii finisate, respectiv 100 de lei în cazul scândurii de construcţii, se cere să se determine cantitatea de scândură din fiecare tip, în mii de unităţi, care trebuie produsă zilnic pentru maximizarea profitului fabricii.

Utilised algorithm: Genetic Algorithm



1. Result of statement analysis

Fitness Function: ;

where x1 and x2 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:

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

**x1 = 1.5k**

**x2 = 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.

1. Implementation

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).

* Function to find 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 to Generate 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.*

The variation operators are used to increase the genetic variation of the solutions and bring new blood to the population. Because the genes are represented real numbers, the operators must be choosen to help increase this variation.

* 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 ∈ [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:*

zi = alfa \* xi + (1-alfa) \* yi

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

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.

* Check 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.

* Check Feasability Decision 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 perfd]-‘centages are almost always lower than the inputted ones), but it ensures only feasible solutions. An alternative would be to surround this function’s call with a while loop in mutation/crossover (i.e. Compute every mutation until feasible), thus maintaining the mutation/crossover percentages, with the downside of a longer execution time.

* Tournament Selection Function:
* Elitist Selection Function: