Instituto Politécnico Nacional
Escuela Superior de Cómputo
Evolutionary Computing Genetic Algorithm
Student: Rodríguez Coronado Ricardo Professor: Jorge Luis Rosas Trigueros
Realization: 02/mar/20 Delivery date: 08/jul/20

Theoretical framework

Genetic Algorithm (GA)

Metaheuristic inspired by the process of biological evolution in a genetic algorithm, a population of candidate solutions (called individuals) to an optimization problem is evolved toward better solutions.

Each candidate solution has a set of properties (it's chromosomes or genotype) which can be mutated and altered.

The evolution usually starts from a population of randomly generated individuals, and is an iterative process, with the population in each interaction called generation.

In each generation, the fitness of every individual in the population is evaluated, the fitness is usually the value of the objective function in the optimization problem being solved. The move fit individuals (best phenotypes) are stochastically select from the current population, and each individual's gnome is modified (recombined and randomly mutated) to from a new generation.

Commonly the algorithm terminates when either a maximum number of generations has been produced, or satisfactory fitness level has been reached.

A GA requires:

- A representation of the solution.
- A fitness function.

Initialization

Typically contains as many individuals as possible. Often, the initial population is generated randomly allowing the entire range of the possible solutions. The selections may be “selected” in areas of the search species where optimal solutions are likely to be found

Selection

A portion of the existing population is selected to bring a new generation.

Fitter solutions are typically more likely to be selected. An example could be an weighted roulette wheel.

Genetic Algorithm's for combinatorial optimization.

A GA is designed as a combination of patterns in encoding generation, selection, joining, replacement and stopping criteria.

In a typical optimization problem, there is a range of points named search the range and the goal is to find a point leads us to the optimal solution.

An essential characteristic of GA is the encoding of the variables that describe the problem. It is often possible to encode the problem using a set of ones and zeros, but other ideas have been explored.

One of the most important components of a GA is the crossover operator. Two main options remain popular:

Point crossover: The string of the parent’s chromosomes is randomly cut on one or more points of the parent’s chromosomes a new child is created.

Uniform crossover: Any genes of both parent’s chromosomes have a chance to become a part of the child’s chromosomes

Mutation: Acts like an insurance policy against premature less if important information.

A mutation probability can be assigned to each individual and each gene.

The children of the old generation can be replaced with the newly generated children. One possible scheme us to have M existing chromosomes and generate L new children.

The chose M individual from the argumented population of (M+L) chromosomes.

Jugs Problem

Water pouring puzzles (also called water jug problems) are a class of puzzle involving a finite collection of water jugs of known integer capacities. Initially each jug contains a known integer volume of liquid, not necessarily equal to its capacity. Puzzles of this type ask how many steps of pouring water from one jug to another (until either one jug becomes empty or the other becomes full) are needed to reach a goal state, specified in terms of the volume of liquid that must be present in some jug or jugs.

By Bézout's identity, such puzzles have solution if and only if the desired volume is a multiple of the greatest common divisor of all the integer volume capacities of jugs. [1]

Material and equipment:

Python 3.0+
A text editor
PC or Laptop with: Windows 7+/Linux/MacOS X

Practice development
The objective of the practice is to solve the jugs problem, using a genetic algorithm. The problem consists of 2 jugs, the first with a capacity of 5 liters, while the second with 3 liters, it is sought that any of these two contain exactly 4 liters without any other measuring instrument than the jugs themselves, this is achieved with actions about the jugs; fill them, empty them or pour the content from one into another. The first step in solving this problem is to define the chromosome to be used in the algorithm. In this case it is necessary to represent a series of steps, so all possible actions that can be performed with the jugs are listed.

1. Fill first jug
2. Fill second jug
3. Empty second jug
4. Empty first jug
5. Move content from G2 to G1
6. Move content from G1 to G2

thereby, the genes on the chromosomes will contain a number; limited from one to six, representing an action.

The next step is to define the size of the chromosomes, since there are different ways of reaching the same result; not all optimally, it is necessary that the length is variable, in this way the algorithm is generalized and different input data can be used, such as the capacity of the jugs or the objective to reach, since the number of steps necessary to Solving these problems varies based on these.

Focusing more on the code, the following function is used to generate random chromosomes that work as a starting point for the program.

```python
def random_chromosome():
    """Genera los cromosomas de forma aleatoria"""
    global L_INI
    chromosome=[]
    for _ in range(0, randrange(1,L_INI)):
        chromosome.append(randrange(6))
    return chromosome
```

Where the genes contain a random number from 1 to 6.

To achieve mentioned coding, a function called "hacerAccion" is used, which, based on an instruction number and a pair of jugs, returns a duple of jugs affected by the instruction, as follows.

```
def hacerAccion(instruccion: int, garrafones: Tuple[int, int], mostrarPasos:bool = Fa
lse) -> Tuple[int, int]:
   cadaux = ''
   if(instruccion == 0): #Llenar el primer garrafón
      garrafones = (G1, garrafones[1])
      cadaux = '-LLenar G1 ' + str(garrafones)
   elif(instruccion == 1): #LLenar el garrafón 2
      garrafones = (garrafones[0], G2)
      cadaux = '-Llenar G2 ' + str(garrafones)
   elif(instruccion == 2): #Vaciar el garrafón 2
      garrafones = (garrafones[0], 0)
      cadaux = '-Vaciar G2 ' + str(garrafones)
   elif(instruccion == 3): #Vaciar el garrafón 1
      garrafones = (0, garrafones[1])
      cadaux = '-Vaciar G1 ' + str(garrafones)
   elif(instruccion == 4): #Pasar contenido de G2 a G1
      aux = garrafones[0] + garrafones[1]
      if aux>G1:
         garrafones = (G1, aux - G1)
      else:
         garrafones = (aux, 0)
      cadaux = '-G2 -> G1 ' + str(garrafones)
   elif(instruccion == 5): #Pasar contenido de G1 a G2
      aux = garrafones[0] + garrafones[1]
      if aux>G2:
         garrafones = (aux - G2, G2)
      else:
         garrafones = (0, aux)
      cadaux = '-G1 -> G2 '+ str(garrafones)
   if mostrarPasos:
      print(cadaux)
   return garrafones
```

It is worth mentioning that "cadaux" and "show steps" are variables that help to show the steps that were carried out, but do not affect the purpose of the function in any way.

Since this function only performs one action at a time, it is necessary to use it within a cycle that goes through a chromosome, but that will be seen later.

As already mentioned, chromosomes will have a variable length, which is why the following lines of code are used when generating a new iteration to the best two in the population, so in case that the probability is met; represented by the variable "prob_i", a new gene is added to the chromosome, with a random instruction; once again, within the range of 1 to 6.

```
if random.random() < prob_i:
        o1.append(randrange(6))
if random.random() < prob_i:
        o2.append(randrange(6))
```

With all the above, a function is now required that tells the algorithm a value that represents how close each of the chromosomes is to “containing” the result, the decoding function.

```
def decodificarCromosoma(chromosome: List[int]) -> float:
   """A partir de un cromosoma, genera todas las instrucciones que tiene
   y regresa el valor final de primer garrafón"""
   g = (0,0)
   for i in chromosome:
      g = hacerAccion(i, g, False)
   return g[0]
```

The third parameter of the “hacerAccion” function is False, since it is a flag that, if it is true, shows the action that was taken, once again, this does not affect the algorithm at all.

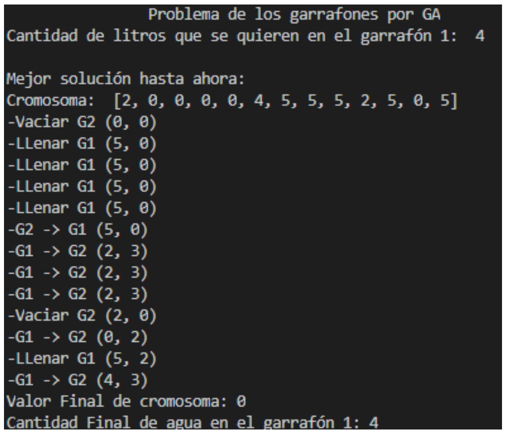
As mentioned before, it was necessary to use the “hacerAccion” function within a cycle that would go through each of the genes on a chromosome, here it makes its triumphal entry. This function receives a chromosome and returns a value that indicates how close it is to the expected result. The variable "i" is the representation of each gene, while "g" is a pair of integers that represents the amount of water that the jugs contain, which conserves its value from the interaction n, to be used in n+1.

At this point, what the algorithm lacks is knowing how to interpret the decoded value, how close it is to the solution, to solve this, it is used the adjustment function.

```
def f(c: List[int]) -> int:
   """Función de ajuste, determinar qué tan apto es un cromosoma"""
   global T
   return abs(decodificarCromosoma(c) - T)
```

T represents the value that is wanted in a jug, in this case 4.
As simple as obtaining the absolute value of the difference of T and the value returned by the decoding function, in this way the chromosome whose adjustment is closest to zero is the closest to the result.

Running the algorithm whit 2000 iterations, an insertion probability of 0.85 and a mutation probability of 0.85 the output is shown in Image 1 - Output of algorithm.



```
                Problema de los garrafones por GA
Cantidad de litros que se quieren en el garrafón 1:  4

Mejor solución hasta ahora:
Cromosoma:  [2, 0, 0, 0, 0, 4, 5, 5, 5, 2, 5, 0, 5]
-Vaciar G2 (0, 0)
-LLenar G1 (5, 0)
-LLenar G1 (5, 0)
-LLenar G1 (5, 0)
-LLenar G1 (5, 0)
-G2 -> G1 (5, 0)
-G1 -> G2 (2, 3)
-G1 -> G2 (2, 3)
-G1 -> G2 (2, 3)
-Vaciar G2 (2, 0)
-G1 -> G2 (0, 2)
-LLenar G1 (5, 2)
-G1 -> G2 (4, 3)
Valor Final de cromosoma: 0
Cantidad Final de agua en el garrafón 1: 4
```

Image 1 - Output of algorithm

As can be seen, the result is not optimum, but the goal is reached.

Conclusions

The operation of genetic algorithms is not only limited to finding minimum values, coding is a key part and what is due to these is possible to solve different problems, in this case, a series of actions.
The different combinations of chromosome growth and mutation difficulties, as well as the number of iterations, will be seen in the final result, making it necessary to find a good balance between these for optimal performance in terms of speed and reliability.

Bibliography

[1] «Wikipedia,» 29 May 2020. [En línea]. Available: https://en.wikipedia.org/wiki/Water_pouring_puzzle#cite_note-3. [Last access: 02 Mar 2020].