Instituto Politécnico Nacional

Escuela Superior de Cómputo

Evolutionary Computing

Session 4 "Genetic Algorithms. Design"

Student: Naranjo Ferrara Guillermo

Teacher: Rosas Trigueros Jorge Luis

Lab session date: 25 February 2020

Delivery date: 03 March 2020

1 Theoretical Framework

1.1 Genetic Algorithms

Genetic algorithms are probabilistic search procedures designed to work on large spaces involving states that can be represented by strings[1] of bits, numbers or symbols.[2]

The term genetic algorithm, almost universally abbreviated nowadays to GA, was first used by John Holland, whose book Adaptation in Natural and Artificial Systems published in 1975 was s significant contribution on the flourishing field of Evolutionary Computing.[3] Holland's major interest was in the phenomenon of adaptation, how living systems evolve or otherwise change in response to other organisms or to a changing environment, and how computer systems might use similar principles to be adaptive as well. In his book he laid out a set of general principles for adaptation.[2]

1.1.1 Some preliminaries

In a GA, a population of candidate solutions to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties which can be mutated or altered. Th evolution is an iterative process which usually starts from a random generated population. Every iteration is called a generation.[4]

In each generation, the fitness (the value of the optimization function in the optimization problem being solved) of every individual is evaluated. The more fit individuals are stochastic-ally selected to form a new generation by recombining or mutating its genome.[4]

A GA is designed as a combination of patterns in encoding, generation, selection, replacing and stopping criteria. In a typical optimization problem there is a range of points named as search range, and the goal is to find a point that leads us to the optimal solution.[4]

1.1.2 A recipe to construct a GA

The input to the GA has two parts: a population of candidate programs, and a fitness function that takes a candidate program and assigns to it a fitness value that measures how well that program works on the desired task.[2]

Now, we can repeat the following steps for some number of generations:

1. Generate an initial population of candidate solutions. The simplest way to create the initial population is just to generate a bunch of random programs (strings), called "individuals."

2. Calculate the fitness of each individual in the current population.

3. Select some number of the individuals with highest fitness to be the parents of the next generation.

4. Pair up the selected parents. Each pair produces offspring by recombining parts of the parents, with some chance of random mutations, and the offspring enter the new population. The selected parents continue creating offspring until the new population is full (i.e., has the same number of individuals as the initial population). The new population now becomes the current population.

5. Go to step 2.

2 Material and Equipment

In order to this practice we need:

- A working computer.
- Any version (preferentially the last) of python installed in the computer.

3 Development

3.1 0/1 Knapsack problem

Write an algorithm for the 0/1 Knapsack problem. Prove it works correctly with the next parameters:
$C = 11$ $\quad$ $W = \{3, 4, 5, 9, 4\}$ $\quad$ $V = \{3, 4, 4, 10, 4\}$

To solve this problem with a GA, it's necessary to first think about the modeling of the problem, i.e., the encoding for the chromosomes, the fitness function, the methods of selection and the genetic operators.

3.1.1 Encoding

In class was proposed a very simple and useful encoding for the possible solution. Given the number of items that can be chosen to put inside the knapsack, we use list of 1's and 0's of size equals to the number of items available. Each item in the list corresponds to one item to put in the knapsack, so to represent that an item has been chosen we use 1, in contrary case we use 0.

3.1.2 Fitness function

For this we're considering two aspects, the weight and the value of the item. In first instance I decided to use the function proposed in class:

$$f(x) = totValue(x) - \begin{cases} totWeight(x) - C & if \quad totWeight(x) > C \\ 0 & if \quad totWeight(x) \leq C \end{cases} \tag{1}$$

where:

$$C = \text{capacity of the knapsack}$$

$$totValue(x) = \sum^{x_i} value(x_i) \tag{2}$$

$$totWeight(x) = \sum^{x_i} weight(x_i) \tag{3}$$

In the implementation I had problems two with this function

1. In cases where the weight was bigger than the capacity, but the value were so high that even when subtracting the difference between the capacity and the weight to it, it continue being the fittest chromosome.

2. To solve the previous problem I decided to simply return 0 when the weight surpassed the capacity, but doing this, in cases where all the chromosomes surpass the weight, I had problems calculating the probabilities for each chromosome in the roulette because there was a division by 0.

To solve this I decided change a little bit the equation by multiplying by 0.1 the value of the subtraction, and removing the subtract of $totValue(x)$, resulting in the next equation:

$$f(x) = \begin{cases} (totWeight(x) - C) \times 0.1 & if \quad totWeight(x) > C \\ totValue(x) & if \quad totWeight(x) \leq C \end{cases} \tag{4}$$

3.1.3 Methods of selection and Genetic operators

I used two methods:

1. Elitism. The two fittest individuals pass immediately to the next generation.

2. Roulette wheel. Depending on the fitness value for each individual, is calculated the probability to be chosen to be the parent of an individual of the next generation.

For the operators I decided to use the same two that were in the example provided, being: crossover between two parents and mutation over the resulting child of the crossover.

3.1.4 Implementation

I decided to implement it in a class in order to have a more organized an legible code. In the constructor if the class I ask for all the parameters needed for the problem and initialize the variables that I'll require all over the process.

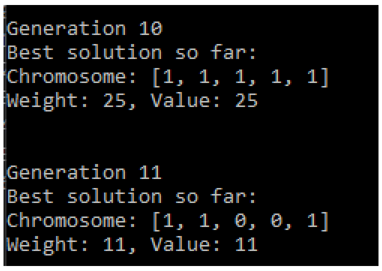
After creating the object I only need to generate the chromosomes randomly by calling the *generateChromosome* method, and then, for each generation, call the method *nextGeneration*, which uses the other methods to perform the algorithm

The *nextGeneration* function first orders chromosomes from the fittest to unfittest individual using the function *compareChromosome*, after that, the list that contains the fittest values for each chromosome is filled by calling *evaluateChromosome*. Now the best two individuals pass to the next generation, and to decide the other individuals, I create a roulette wheel by calling the method *createWheel*.

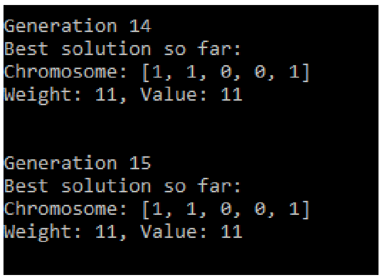
The method *createWheel* assigns a n-number of places to each chromosome in the roulette, based in its fitness value, assigning more probabilities to be mutated to the chromosomes with the unfittest values.

After the wheel is created, four times the program will chose randomly from the roulette two parents which will be crossing to generate two child, each one of which will be mutated according to a mutation probability. Finally this two children evolved from the last generation are added to the list representing the new generation and the process is repeated till reach the generation specified.

In the figures showing the test of the algorithm we can see that, fifteen generations were more than enough, in two runs realized, to reach the optimal solution

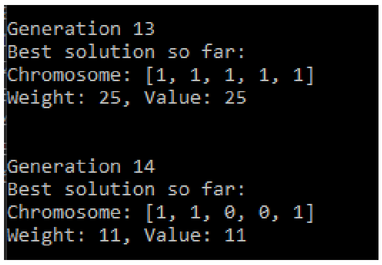


```
Generation 10
Best solution so far:
Chromosome: [1, 1, 1, 1, 1]
Weight: 25, Value: 25


Generation 11
Best solution so far:
Chromosome: [1, 1, 0, 0, 1]
Weight: 11, Value: 11
```



```
Generation 14
Best solution so far:
Chromosome: [1, 1, 0, 0, 1]
Weight: 11, Value: 11


Generation 15
Best solution so far:
Chromosome: [1, 1, 0, 0, 1]
Weight: 11, Value: 11
```

Figure 1,2: Optimal solution found in the 11^{th} generation of the first run.



```
Generation 13
Best solution so far:
Chromosome: [1, 1, 1, 1, 1]
Weight: 25, Value: 25


Generation 14
Best solution so far:
Chromosome: [1, 1, 0, 0, 1]
Weight: 11, Value: 11
```

Figure 3: Optimal solution found in the 14^{th} generation of the second run.

3.2 Minimum number of coins (CMP) problem

As well as in the last problem and every problem involving GA, we need to think of the abstraction of the problem and represent it, as presented below.

3.2.1 Encoding

I decided to use the solution proposed in class which was to use a list of elements representing the coins following the next form

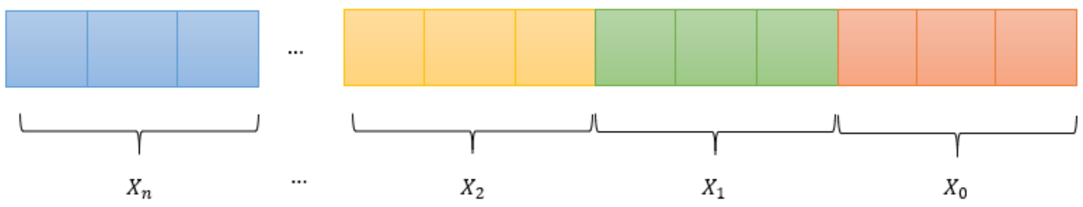


Figure 4: Encoding for the GA of the CMP

where n is the number of denominations that we can use. By first see is notorious that this encoding is limiting because we can only use as maximum 7 coins of each denomination to reach the change required, because we're representing each value with three positions in the array so the number maximum of coins is the maximum binary value we can represent with three bits 111 which is equals to 7,but for the session purposes it is more than enough.

3.2.2 Fitness function

For the fitness function I decided to use the proposal made in class which was the following one

$$f(x) = \alpha(abs(T - T(x))) + \beta(x_n + x_{n-1} + ... + x_1) \tag{5}$$

where:
n = Number of denominations available
x_i = Number of coins chosen of the denomination denoted by i
T = Total change required
$T(x)$ = Change of the proposed solution
$\alpha = 0.9$
$\beta = 0.1$

As it can seen, for the function relies on part on an α-β factor used in areas like Neural Networks and Economics. For this case, we use it more like in Economics. This factor will adjust the fitness of the individual in order to give priority to the fact of having the exact change required, and then obtain the optimal number of coins to give the change.

3.2.3 Methods of selection and Genetic operators

I'm using the same two methods for the knapsack problem and are implemented the same but with an exception in the case of the roulette that will be mentioned later.

For the operators, because of the encoding of the chromosomes I decided to only use mutation.

3.2.4 Implementation

This algorithm also was implemented with POO, creating only one class encapsulating the behaviour of the algorithm. As well as in the knapsack, the constructor asks for the parameters required for the problem and initialize some variables.

After that the method *creationChromosome* is called and creates the chromosomes the same way as in the last problem.

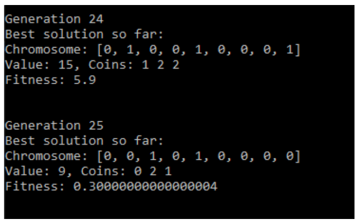
I only need to call the method *nextGeneration* to obtain the next generation of the individuals. This method first orders the chromosomes from the fittest to the unfittest using the function *compareChromosome*. After this, the list containing the fitness value for each chromosome is filled calling the method *evaluateChromosome*, which calls the method *fitnessFunction* in which is implemented the fitness function selected, this method calls *decodeChromosome*.

How is decoded the chromosome? Well, in the last mentioned method is calculated the value in decimal denoted for each one of the "sub-lists" of 3 elements, this value represents the number of coins for the denomination specified. At the same time an accumulator is storing the value of each sub-list multiplied for the its corresponding denomination. After this, the total value is append to a list, as well as the number of coins for each denomination, returning this list, in this particular case of size 4.

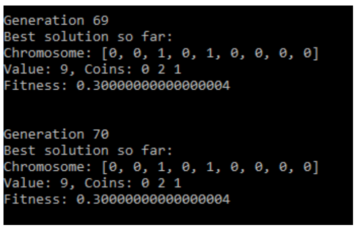
Continuing with the process of *nextGeneration*, the two fittest individuals pass directly to the next generation, then a roulette is created with the method *createWheel*. Earlier I talked about a little change in the creation of the wheel, it is that, contrary to the knapsack problem, where we use the maximum value from the fitness values to calculate the number of places that each chromosome will take in the wheel, here, because of the nature of the fitness function, where the individual with the minimum value of fitness is the fittest, instead of using the maximum, we're using the minimum. All other things referring to the wheel is the same.

Being created the wheel, a random chromosome is chosen from it, and an random gene in the chromosome is mutated, finally adding it to the next generation.

In the images we can see some runs of the algorithm. In the first, run was set the maximum number of generations at 70, coincidentally, the algorithm converged to the optimal solution at the 25th generation. Because of that I ran the algorithm some more times, decreasing the maximum of generations to 30, but the algorithm never converged.

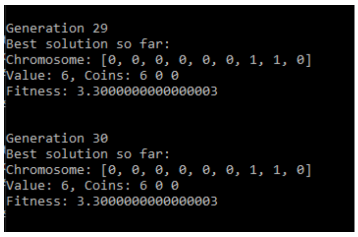


```
Generation 24
Best solution so far:
Chromosome: [0, 1, 0, 0, 1, 0, 0, 0, 1]
Value: 15, Coins: 1 2 2
Fitness: 5.9

Generation 25
Best solution so far:
Chromosome: [0, 0, 1, 0, 1, 0, 0, 0, 0]
Value: 9, Coins: 0 2 1
Fitness: 0.30000000000000004
```

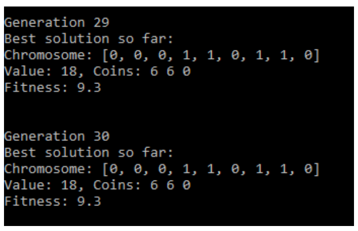


```
Generation 69
Best solution so far:
Chromosome: [0, 0, 1, 0, 1, 0, 0, 0, 0]
Value: 9, Coins: 0 2 1
Fitness: 0.30000000000000004

Generation 70
Best solution so far:
Chromosome: [0, 0, 1, 0, 1, 0, 0, 0, 0]
Value: 9, Coins: 0 2 1
Fitness: 0.30000000000000004
```

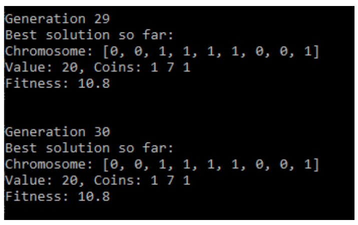
Figure 5,6: Optimal solution found in the 25th generation of the first run.



```
Generation 29
Best solution so far:
Chromosome: [0, 0, 0, 0, 0, 0, 1, 1, 0]
Value: 6, Coins: 6 0 0
Fitness: 3.3000000000000003

Generation 30
Best solution so far:
Chromosome: [0, 0, 0, 0, 0, 0, 1, 1, 0]
Value: 6, Coins: 6 0 0
Fitness: 3.3000000000000003
```



```
Generation 29
Best solution so far:
Chromosome: [0, 0, 0, 1, 1, 0, 1, 1, 0]
Value: 18, Coins: 6 6 0
Fitness: 9.3

Generation 30
Best solution so far:
Chromosome: [0, 0, 0, 1, 1, 0, 1, 1, 0]
Value: 18, Coins: 6 6 0
Fitness: 9.3
```



```
Generation 29
Best solution so far:
Chromosome: [0, 0, 1, 1, 1, 1, 0, 0, 1]
Value: 20, Coins: 1 7 1
Fitness: 10.8

Generation 30
Best solution so far:
Chromosome: [0, 0, 1, 1, 1, 1, 0, 0, 1]
Value: 20, Coins: 1 7 1
Fitness: 10.8
```

Figure 7,8,9: Multiple runs in which any chromosome converged to the optimal solution

4 Conclusion

Having the labor to implement the algorithms designed in class and even modified them a little bit in order to function correctly made me realize of the importance of having not only a good designed and specified solution, but also the importance of the probabilities, that play an important roll in every step of the algorithm, since the randomly creation of the chromosomes, to the probabilities of mutation and even the ones used to construct the roulette. Although in the first case there isn't in fact, but a random event, I see the effect that, being able to "modified" that randomness, this would have in the time or generations required to get to the solution. Effect that also is seeing in the probability selected for a genetic operation to perform.

This practice also made me realize of the powerfull of a GA well designed, because, for problems that would take a very long time to solve or that are difficult to design a solution with common algorithms, instead, a GA with well defined rules can help me to find a solution based on previous calculated possible solutions that eventually will reach the correct or optimal solution.

5 Bibliography

[1]D. E. Goldberg, J. H. Holland, "Genetic Algorithms and Machine Learning", *Machine Learning*, no. 3, pp. 95-99, 1998.

[2]C. Reeves, "Genetic Algorithms", *Handbook of metaheuristics*, Chapter 3, pp. 109-139, September 2010.

[3]M. Mitchel. *Complexity. A guided tour.* New York: Oxford University Press, 2009.

[4]"Evolutionary Computing", class notes, Academy of Computer Science, ESCOM-IPN, Primavera 2020.