Instituto Politécnico Nacional
Escuela Superior de Cómputo
Evolutionary Computing Dynamic Programming
Student: Rodríguez Coronado Ricardo
Professor: Jorge Luis Rosas Trigueros
Realization: 18/feb/20 Delivery date: 25/feb/20

Theoretical framework

Genetic Algorithm (GA)

Meta heuristic 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.

Material and equipment:

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

Practice development

In the lab session we used of a python script to exemplify the process of genetic algorithms, however, it was written in version 2.7, for that, some lines of code were incompatible with the interpreter of version 3.0 or higher . To solve this problem, some lines were modified, such as:

Original	Modification	Reason
`from Tkinter import *`	`from tkinter import *`	The name of the library used for python3 changes.
`crossover_point=L_chromosome/2`	`crossover_point=int(L_chromosome/2)`	The division between two integers using the '/' symbol resulted in an integer. This changed in Python 3, now results in a floating number.[1]
`for i in range(0, (N_chromosomes-2)/2):`	`for i in range(0,int((N_chromosomes-2)/2)):`	
`F0.sort(cmp=compare_chromosomes)`	`import functools` `F0.sort(key=functools.-cmp_to_key(compare_chromosomes))`	CMP is no longer a function built into .sort () in python version 3, but it can be emulated. [2]

Once the changes were made, the program was executed, which showed a window, Figure 1.

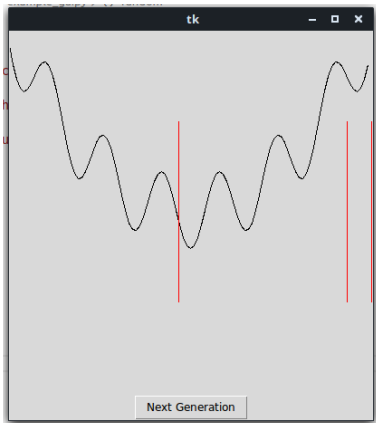

Figure 1.

By pressing the ‘Next Generation’ button, the program proceeds to do an iteration; or generation, mutating the chromosomes and displaying the best current solution on the console, as shown in Figure 2-1 and Figure 2-2.

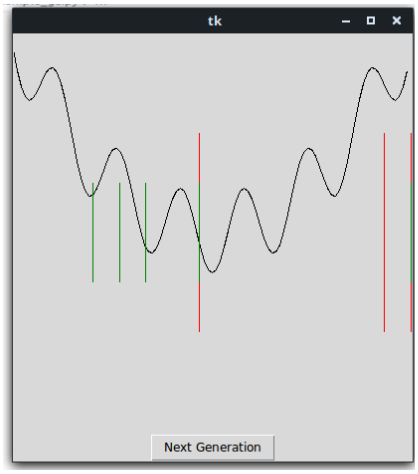

Figure 2-1.

```
ample_ga.py"
Best solution so far:
f(-1.3333333333333321)= -0.8520614043230732
```

Figure 2-2.

When making some interactions; Usually 2, it can be seen how the solution becomes optimal with the value:
f(-6.666666666666666)= -1.4872485899816805.

The next step was the modification of some script’s variables.
First the length of the chromosomes

```
L_chromosome=4
```

The new value of the variable is 16, when executing the program again and performing a couple of generations, it can be seen that the optimal solution is a smaller number, with the chromosomes converging towards the minimum of the graph, as shown in Figure 3-1 and Figure 3-2.

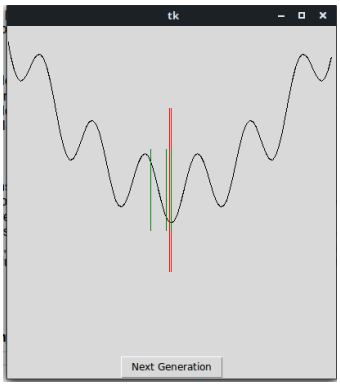

Figure 3-1

```
Best solution so far:
f(-0.0003051804379339273 5)= -3.999999809073047
```

Figure 3-2

The next variable that changed was the number of chromosomes

```
N_chromosomes=10
```

The number of chromosomes influences in such a way that, the more there are; for example 20, you are more likely to find the optimal value and faster; usually to the first generation, while a smaller number; for example, 2 will create the opposite effect, including a situation where the true optimal solution was not reached.
However, there are cases in which a large number of chromosomes does not result in the expected optimal solution.

The last variable modified was the probability of mutation of the chromosomes

```
prob_m=0.5
```

Initially the probability that a chromosome mutes is 50%. The changes that can be observed by decreasing this number to 25% is that the optimal solution takes longer generations; due to the lack of diversity in the chromosomes, or even not expect the optimal solution. Whereas, when the probability increases, the stagnation of the chromosomes in the non-local minimums is avoided.

The next step was the modification of the adjustment function, which is:

```
def f(x):
return 0.05*x*x-4*math.cos(x)
```

The previous function was changed by the Rastrigin function, in 1 Dimension, causing the graph and the optimal solution data to be different, as shown in Figure 4-1 and Figure 4-2.

```
def f(x):

   d = 1 #dimension
   return 10*d + (x**2 - 10 * math.cos( 2 * math.pi * x ))
```

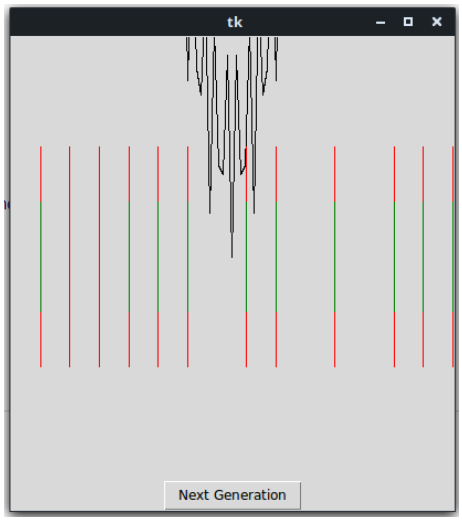

Figure 4-1

```
Best solution so far:
f(-4.0)= 16.0
```

Figure 4-2

Conclusions

The genetic algorithms generate solutions in a very peculiar way, since they approach optimal values, like randomly, however the interesting point is to penalize or reward some chromosomes for their characteristics, in this way it is possible to make the attributes of a population converge to an optimal solution.

Diversity is also an important topic in the combination of chromosomes, since the more diversity, there will be a bigger range of possible solutions, increasing the possibility that one of these is optimal. To get a varied diversity, it is important to talk about the importance of the probability of mutation and the amount of chromosomes, because, from the combination of these, different attributes can be obtained in the child chromosomes.

Bibliography

- [1] Python Documentation contents (2020). In The Python Language Documentation. *Retrieved* 24 February 2020 https://docs.python.org/3/reference/expressions.html
- [2] Python Documentation contents (2020). In The Python Language Documentation. *Retrieved* 24 February 2020 https://docs.python.org/3/library/functools.html#functools.cmp_to_key