

30/04/2019

# Trabajo Práctico 1: Algoritmo Genético Canónico

ALGORITMOS GENÉTICOS 2019

ANTONELLI (44852) – RECALDE (44704) – ROHN (41355)

# Índice

---

<b>ÍNDICE .....</b>	<b>1</b>
<b>INTRODUCCIÓN .....</b>	<b>2</b>
<b>CÓDIGO .....</b>	<b>3</b>
<b>EXPLICACIÓN DE FUNCIONAMIENTO.....</b>	<b>11</b>
<b>GRÁFICAS Y TABLAS .....</b>	<b>12</b>
ALGORITMO EN 20 GENERACIONES .....	12
ALGORITMO EN 100 GENERACIONES .....	13
ALGORITMO EN 200 GENERACIONES .....	16
<b>CONCLUSIÓN .....</b>	<b>22</b>

---

# Introducción

---

Implementamos algoritmos genéticos como una simulación computacional en la cual una población de las representaciones abstractas de las soluciones candidata (individuos) de un problema de optimización, evoluciona hacia mejores soluciones.

En este caso utilizamos *Python* y *C++* para la creación de un programa, mediante el uso de un **algoritmo genético canónico**, que nos permite buscar el máximo de la función objetivo:

$$f(x) = \left(\frac{x}{coef}\right)^2 \text{ en el dominio } [0, 2^{30} - 1]$$

Donde:  $coef = 2^{30} - 1$

Para lograr la implementación del algoritmo genético se tuvieron en cuenta los siguientes datos:

- Probabilidad de Crossover = 0,75
- Probabilidad de Mutación = 0,05
- Población inicial: 10 individuos
- Ciclos del programa: 20
- Método de Selección: Ruleta
- Método de Crossover: 1 punto
- Método de Mutación: invertida

# Código

## Python

- `__init__`

```
1. from GeneticAlgorithms.Population import Population
2. from GeneticAlgorithms.Graphs import Graphic
3. # from GeneticAlgorithms.Chromosome import Chromosome
4. # import random
5.
6.
7. if __name__ == '__main__':
8.     # ImportantValues
9.     iterationLimit = 50 # 20,100,200 # Population Iterations
10.    initPopulationNum = 10 # Initial Population Size
11.    chromosomeSize = 30 # Chromosome Size
12.    crossoverProb = 0.75 # Probability of CrossOver
13.    mutationProb = 0.05 # Probability of Mutation
14.
15.    # Initialize
16.    class Main(object):
17.        # First Population
18.        pob = Population(initPopulationNum, chromosomeSize, crossoverProb, mutationProb)
19.        graphicsData = {'averageOPs': [], 'minOPs': [], 'maxOPs': []} # Dictionary for Graphics
20.
21.        # Iterations
22.        for iterationCount in range(iterationLimit):
23.            print()
24.            averageOP, minOP, maxOP = pob.showPopulation(iterationCount) # Show Actual Population and Return Data
25.
26.            # Update Dictionary with important values
27.            graphicsData['averageOPs'].append(averageOP)
28.            graphicsData['minOPs'].append(minOP)
29.            graphicsData['maxOPs'].append(maxOP)
30.
31.            # In the last iteration, the chromosomes population mustn't reproduce
32.            if iterationCount < iterationLimit - 1:
33.                pob.reproduce() # Reproduction of Actual Generation
34.
35.            # Graph Population's Evolution
36.            graph = Graphic(graphicsData, iterationLimit)
37.            graph.showPlots()
38.
39.            # Last Reproduction Message
40.            print("Last Generation Reached Correctly")
41.            print("-----")
42.            print()
43.            print()
44.
45.            # Final Tables
46.            print("TABLAS FINALES")
47.            print()
48.            print("Población ----- Mínimo ----- Máximo ----- Promedio")
49.            for value in range(iterationLimit):
50.                print("Generación ", value, ":", graphicsData['minOPs'][value], "---",
51.                    graphicsData['maxOPs'][value], "----",
52.                    graphicsData['averageOPs'][value])
```

- Chromosome

```
• #!/usr/bin/env python
• # -*- coding: utf-8 -*-
• import random
```

```

•
•
• class Chromosome(object):
•     # Class Attribute (Constant)
•     # coef = random.randint(1, 230 - 1)
•     coef = (2**30)-1 # (2^30)-1
•
•     # Constructor / Instance Attributes
•     def __init__(self, large, newBody):
•
•         # Chromosome's Genes
•         if newBody is None:
•             self.body = []
•             for _ in range(large):
•                 self.body.append(str((random.randint(0, 1))))
•         else:
•             self.body = newBody
•             self.large = large
•             # Initialize Objective Function Punctuation and Fitness
•             self.setObjectivePunctuation()
•             self.fitness = 0
•
•     # Show All Genes of the Chromosome
•     def getBody(self):
•         return self.body
•
•     def toBinInteger(self):
•         str_bin_num = ''.join(str(i) for i in self.body)
•
•         return int(str_bin_num)
•
•     # Real Number to pass on Objective Function
•     def getRealValue(self):
•         num = ''.join(str(i) for i in self.body)
•         return int(num, 2) # Convert body to String and then to Binary Int
•
•     def calcObjPunc(self):
•         return (self.getRealValue() / self.getCoef()) ** 2
•
•     def calcFitness(self, totalObj):
•         # if totalObj == 0: totalObj = 1 # Prevent Division by Zero Error
•         self.fitness = (self.getObjectivePunctuation() / totalObj) # Update Fitness
•         return self.fitness
•
•     # def copy(self, Chrom, num1, num2): pass
•     # def mutate(self): pass
•
•     # Class Methods
•     @classmethod
•     def getCoef(cls):
•         return cls.coef
•
•     @classmethod
•     def setCoef(cls, coefficient):
•         cls.coef = coefficient
•
•     # Getters and Setters
•     def getLarge(self):
•         return self.large
•
•     def setLarge(self, large):
•         self.large = large
•
•     def getObjectivePunctuation(self):
•         return self.objectivePunctuation
•
•     def setObjectivePunctuation(self):

```

```

•         self.objectivePunctuation = self.calcObjPunc()
•
•     def getFitness(self):
•         return self.fitness
•
•     def setFintess(self, fitness):
•         self.fitness = fitness
•
•     def copy(self, another_crom, start, end):
•         for i in range(start, end):
•             self.body[i] = (another_crom.body[i])

```

## • Population

```

• #!/usr/bin/env python
• # -*- coding: utf-8 -*-
• from GeneticAlgorithms.Chromosome import Chromosome
• import random
•
•
• class Population(object):
•     # Class Attributes
•     population = [] # Initial Population (Array of Chromosomes)
•     totalObjPunc = 0 # The Sum of All Objective Functions Punctuation
•     totalFitness = 0 # The Sum of All Objective Values
•
•     # Constructor / Instance Attributes
•     def __init__(self, numChroms, chromSize, crossProb, mutProb):
•         self.numChroms = numChroms
•         self.chromSize = chromSize
•         self.crossProb = crossProb
•         self.mutProb = mutProb
•         # print("Objective Function Coeficient:", Chromosome.getCoef())
•         print("Start Algorithm")
•         for _ in range(numChroms):
•             oneChrom = Chromosome(chromSize, None) # Initialization of Chromosomes
•             self.addChrom(oneChrom) # Add to Population
•
•     # Show Actual Population and Stats
•     def showPopulation(self, numIter):
•         self.setTotalFitness(0)
•         self.setTotalObjPunc(self.calcTotalObjPunc())
•         large = self.getChromSize()
•         averageObjPunc = self.getTotalObjPunc() / len(self.population)
•         fitness = 0
•         maxVal = 0
•         minVal = 0
•         maxChrom = 0
•         minChrom = 0
•         print("Population ", (numIter + 1), ":")
•         for i in range(len(self.population)):
•             fitness = self.population[i].calcFitness(self.getTotalObjPunc())
•             self.updateTotalFitness(fitness)
•             if i == 0:
•                 maxVal = fitness
•                 minVal = fitness
•             elif fitness > maxVal:
•                 maxVal = fitness
•                 maxChrom = i
•             elif fitness < minVal:
•                 minVal = fitness
•                 minChrom = i
•         for j in range(large):

```

```

    print(self.population[i].getBody()[j], end='')
    print()
    fitness = self.getTotalFitnessAverage()
    print()
    print("Chromosome --- Value --- Objective Punctuation --- Fitness")
    print("Max Values: Chrom Nº", maxChrom, "with:", self.population[maxChrom].getRealValue(), "Val,",
          self.population[maxChrom].getObjectivePunctuation(), "OP,", round(maxVal, 4), "Fit")
    print("Min Values: Chrom Nº", minChrom, "with:", self.population[minChrom].getRealValue(), "Val,",
          self.population[minChrom].getObjectivePunctuation(), "OP,", round(minVal, 4), "Fit")
    print("Average OP:", averageObjPunc, "--- Average Fitness:", fitness) # round(fitness,6)
    print()
    # Return Important Data to use on Graphics
    return (averageObjPunc, self.population[minChrom].getObjectivePunctuation(),
            self.population[maxChrom].getObjectivePunctuation())

# Calculate Total of Objective Functions Punctuation in the actual Generation
def calcTotalObjPunc(self):
    acumObjPunc = 0
    for chromosome in self.population:
        acumObjPunc += chromosome.getObjectivePunctuation() # Add Every Objective Function Punctuation
    # self.setTotalObjPunc(accumulator)
    return acumObjPunc

# Update Total Fitness
@classmethod
def updateTotalFitness(cls, fitness):
    cls.totalFitness += fitness

# Add to Population
def addChrom(self, Chrom):
    self.population.append(Chrom)

# Reproduction
def reproduce(self):
    parents = [] # List of Potential Parents
    newGeneration = [] # List of Children
    print("Roulette Results: ", end='')
    # lastParent = None # Check if a Chromosome tries to reproduce with himself
    for _ in range(0, len(self.population), 2):
        lastParent = None
        for i in range(2):
            lastParent = self.roulette(lastParent) # Parents Selected by Roulette
            parents.append(self.population[lastParent])
    print()
    for i in range(0, len(parents), 2):
        father1 = parents[i]
        father2 = parents[i + 1]
        if self.crossPosibility(): # CrossOver Probability Evaluation
            son1, son2 = self.cross(father1, father2) # CrossOver
            print("Successful CrossOver in reproduction:", (i + 2) / 2) # Only Print
        else:
            son1, son2 = self.copy(father1, father2) # Direct Assignment (Without CrossOver)
            print("CrossOver didn't happen in reproduction:", (i + 2) / 2) # Only Print
        # Individual Mutation Probability Evaluation
        if self.mutationPosibility():
            son1 = self.mutation(son1)
        if self.mutationPosibility():
            son2 = self.mutation(son2)
        self.addChildren(son1, son2, newGeneration)
    self.replacePopulation(newGeneration)
    self.setTotalFitness(0)

# Genetic Operator (Roulette Method)
def roulette(self, lastParent):
    # Generator of a Bidimensional List (Fitness Range of Chromosomes)
    newRoulette = [[0] * 2 for _ in range(len(self.population))]
    acum = 0 # Acumulator of Relative Fitness from 0 to 1 (Fills Roulette)

```

```

•     for i in range(len(self.population)):
•         newRoulette[i][0] = acum # Range Min: Last Acum Value
•         acum += round(self.population[i].getFitness(), 6) # Acum's Value From Zero
•         newRoulette[i][1] = acum # Range Max: New Acum Value
•         ranNum = round(random.uniform(0, 1), 6) # Random Number from 0.000000 to 0.999999
•         # print("Random: ", ranNum) # Only Print
•         count = 0
•         while count <= 100: # If the same parent is selected more than 100 times... select the next or last one
•             for i in range(len(newRoulette)):
•                 if newRoulette[i][0] < ranNum < newRoulette[i][1]:
•                     # Return Selected Chromosome if the Random Number Exists in its Range
•                     if lastParent != i:
•                         print(i, end=', ')
•                         return i
•                     else:
•                         print("REP", end=', ')
•                         ranNum = round(random.uniform(0, 1), 6)
•                         break
•                 count += 1
•             if count == 100: # Reach Only if the same parent goes selected 100 times
•                 if lastParent < len(newRoulette)-1:
•                     print(lastParent, end=', ')
•                     return lastParent+1
•                 else:
•                     print(lastParent, end=', ')
•                     return lastParent-1
•             print("Error")
•             return "Error" # Error Exit
•
•     def crossPosibility(self): # CrossOver possibility evaluation
•         if self.getCrossProb()*100 >= random.randint(1, 100):
•             return True
•         else:
•             return False
•
•     def cross(self, parent1, parent2):
•         crom_size = parent1.getLarge()
•         son1 = Chromosome(crom_size, None)
•         son2 = Chromosome(crom_size, None)
•         cut = random.randint(1, crom_size - 2) # Random Cut Point (Except by zero or all genes)
•
•         son1.copy(parent1, 0, cut)
•         son1.copy(parent2, cut, crom_size)
•
•         son2.copy(parent2, 0, cut)
•         son2.copy(parent1, cut, crom_size)
•
•         son1.setObjectivePunctuation()
•         son2.setObjectivePunctuation()
•
•         print()
•         print("Son 1:", son1.toBinInteger()) # Only Print
•         print("Son 2:", son2.toBinInteger()) # Only Print
•         print("Cut Point on:", cut) # Only Print
•         return son1, son2
•
•     def copy(self, chrom1, chrom2):
•         # newGeneration.append(chrom1)
•         # newGeneration.append(chrom2)
•         son1 = chrom1
•         son2 = chrom2
•         print()
•         print("Son 1 (Identical):", self.listToInt(chrom1.getBody())) # Only Print
•         print("Son 2 (Identical):", self.listToInt(chrom2.getBody())) # Only Print
•         return son1, son2
•
•     def mutationPosibility(self): # Mutation possibility evaluation

```



```

•         if self.getMutProb() * 100 >= random.randint(1, 100):
•             return True
•         else:
•             return False
•
•     def mutation(self, chrom): # Select one random Gen and Switch its Value
•         mutPos = random.randint(1, self.chromSize)
•         newBody = []
•         for i in range(len(chrom.getBody())):
•             if i != mutPos:
•                 newBody.append(chrom.getBody()[i])
•             else:
•                 if chrom.getBody()[mutPos] == 1: # If is a '0' then change to '1', and vice-versa
•                     newBody.append(0)
•                 else:
•                     newBody.append(1)
•         son = Chromosome(self.chromSize, newBody)
•         print("Mutated Chrom in position:", mutPos, ":", self.listToInt(chrom.getBody())) # Only Print
•         return son
•
•     def addChildren(self, son1, son2, newGeneration):
•         newGeneration.append(son1)
•         newGeneration.append(son2)
•
•     def replacePopulation(self, newGeneration): # Replace All Population in every Iteration
•         self.population = []
•         for i in range(len(newGeneration)):
•             self.population.append(newGeneration[i])
•
•     def listToInt(self, arr):
•         num = ''.join(str(i) for i in arr)
•         return int(num)
•
•     # Class Getters and Setters
•     @classmethod
•     def getTotalObjPunc(cls):
•         return cls.totalObjPunc
•
•     @classmethod
•     def setTotalObjPunc(cls, total):
•         cls.totalObjPunc = total
•
•     @classmethod
•     def getTotalFitness(cls):
•         return cls.totalFitness
•
•     @classmethod
•     def setTotalFitness(cls, total):
•         cls.totalFitness = total
•
•     @classmethod
•     def getTotalFitnessAverage(cls):
•         return cls.totalFitness / len(cls.population)
•
•     # Getters and Setters
•     def getNumChroms(self):
•         return self.numChroms
•
•     def setNumChroms(self, numChroms):
•         self.numChroms = numChroms
•
•     def getChromSize(self):
•         return self.chromSize
•
•     def setChromSize(self, chromSize):
•         self.chromSize = chromSize
•

```

```

•     def getCrossProb(self):
•         return self.crossProb
•
•     def setCrossProb(self, crossProb):
•         self.crossProb = crossProb
•
•     def getMutProb(self):
•         return self.mutProb
•
•     def setMutProb(self, mutProb):
•         self.mutProb = mutProb

```

## • Graphs

```

1.  #!/usr/bin/env python
2.  # -*- coding: utf-8 -*-
3.
4.  from matplotlib import pyplot
5.  # import math
6.  # import numpy as np
7.
8.  # pyplot.plot(x1,y1,'b-',x2,y2,'r-',x3,y3,'g-')
9.  # pyplot.legend('value 1', 'value 2', 'value 3')
10.
11.
12. class Graphic(object):
13.     def __init__(self, graphicsData, iterationLimit):
14.         self.averageOPs = graphicsData['averageOPs']
15.         self.minOPs = graphicsData['minOPs']
16.         self.maxOPs = graphicsData['maxOPs']
17.         self.generations = []
18.         for i in range(iterationLimit):
19.             self.generations.append(i)
20.         # pyplot.ion()
21.
22.     # Show one Plot with all Graphs
23.     def showPlots(self):
24.         # with pyplot.style.context(('dark_background')):
25.             self.drawAll(self.minOPs, self.generations, "Puntuación Objetiva", "Generación", "PUNTUACIÓN OBJETIVA MÍNIMA", 221)
26.             self.drawAll(self.maxOPs, self.generations, "Puntuación Objetiva", "Generación", "PUNTUACIÓN OBJETIVA MÁXIMA", 222)
27.             self.drawAll(self.averageOPs, self.generations, "Puntuación Objetiva", "Generación", "PUNTUACIÓN OBJETIVA PROMEDIO", 212)
28.             pyplot.show() # Draw all the "Subplots"
29.
30.     # Show one Graph in each Plot
31.     def showPlotsApart(self):
32.         self.draw(self.minOPs, self.generations, "Puntuación Objetiva", "Generación", "PUNTUACIÓN OBJETIVA MÍNIMA"
33.         )
34.         self.draw(self.maxOPs, self.generations, "Puntuación Objetiva", "Generación", "PUNTUACIÓN OBJETIVA MÁXIMA"
35.         )
36.         self.draw(self.averageOPs, self.generations, "Puntuación Objetiva", "Generación", "PUNTUACIÓN OBJETIVA PRO
37.         MEDIO")
38.
39.     # Generate one Subplot for every Graph (and Show all Graphs)
40.     def drawAll(self, axisY, axisX, labY, labX, title, region):
41.         pyplot.subplot(region)
42.         pyplot.axis([-1, len(self.generations), 0, 1])
43.         pyplot.grid(True)

```

```
41.     pyplot.plot(axisX, axisY)
42.     pyplot.ylabel(labY)
43.     pyplot.xlabel(labX)
44.     pyplot.title(title)
45.
46.     # Generate one plot for every Graph
47.     def draw(self, axisY, axisX, labY, labX, title):
48.         pyplot.axis([-1, len(self.generations), 0, 1])
49.         pyplot.grid(True)
50.         pyplot.plot(axisX, axisY)
51.         pyplot.ylabel(labY)
52.         pyplot.xlabel(labX)
53.         pyplot.title(title)
54.         pyplot.show()
```

Repositorio GitHub:

<https://github.com/NicoCaptain/Genetic-Algorithms>

### ***Código también en C++***

Repositorio GitHub:

<https://github.com/alereca/Genetic-Algorithms-Cpp>

# Explicación de Funcionamiento

---

## Funcionamiento de Cromosoma:

- Mostrar sus genes en diferentes formatos
- Setear sus genes de manera random al inicializarse con el numero indicado de estos
- Calcular función objetivo a partir del cuadrado de la valuación a número real del conjunto de genes dividido el coeficiente
- Calcular su fitness a partir de dividir la valuación de la función objetivo por la valuación de la función objetivo acumulada por todos los cromosomas
- Mutar un gen en una posición random
- Copiar en sí mismo la parte indicada de otro cromosoma

## Funcionamiento de Población:

- Mostrar los cromosomas que la conforman
- Inicializarse con el numero de cromosomas, la probabilidad de crossover y la probabilidad de mutación indicados
- Reproducirse, lo que incluiría la selección y crossover
- Seleccionar a los cromosomas dentro de una ruleta de manera probabilística según su fitness
- Crear una ruleta que permita que los cromosomas más aptos tengan mayores posibilidades
- Calcular la puntuación objetivo acumulada por todos los cromosomas que la conforman
- Calcular el mínimo, máximo y promedio en cada generación (al inicializarse y después de cada reproducción)

## Funcionamiento de la Grafica:

- Mostrar la grafica de líneas de los valores máximo, mínimo y promedio de cada generación
- Setearse a través de una tabla enviada desde población que contenga todos estos valores

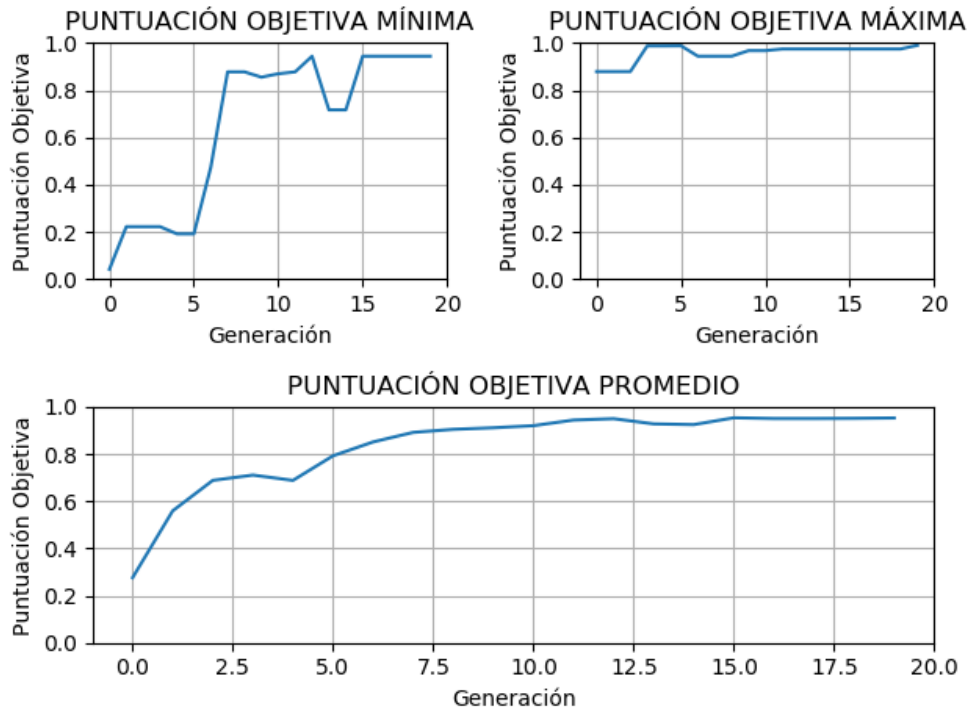
## Funcionamiento Conjunto:

- Se inicializa la población con el numero de cromosomas, el número de genes, la probabilidad de crossover indicados por el usuario *[entrada]*
- Se muestra la generación inicial *[proceso]*
- Se reproducen los cromosomas de la población y se obtiene una nueva generación generalmente mas optima que remplaza a la anterior
- Se agregan los valores máximo, mínimo y promedio de esta generación a una tabla de valores históricos
- Se repite hasta llegar al número de iteraciones indicado por el usuario
- Se generan los gráficos de línea a partir de los valores de cada generación obtenidos de la tabla de valores históricos *[salida]*
- Se analiza si la población convergió a la solución optima o se estanco en un máximo local

# Gráficas y Tablas

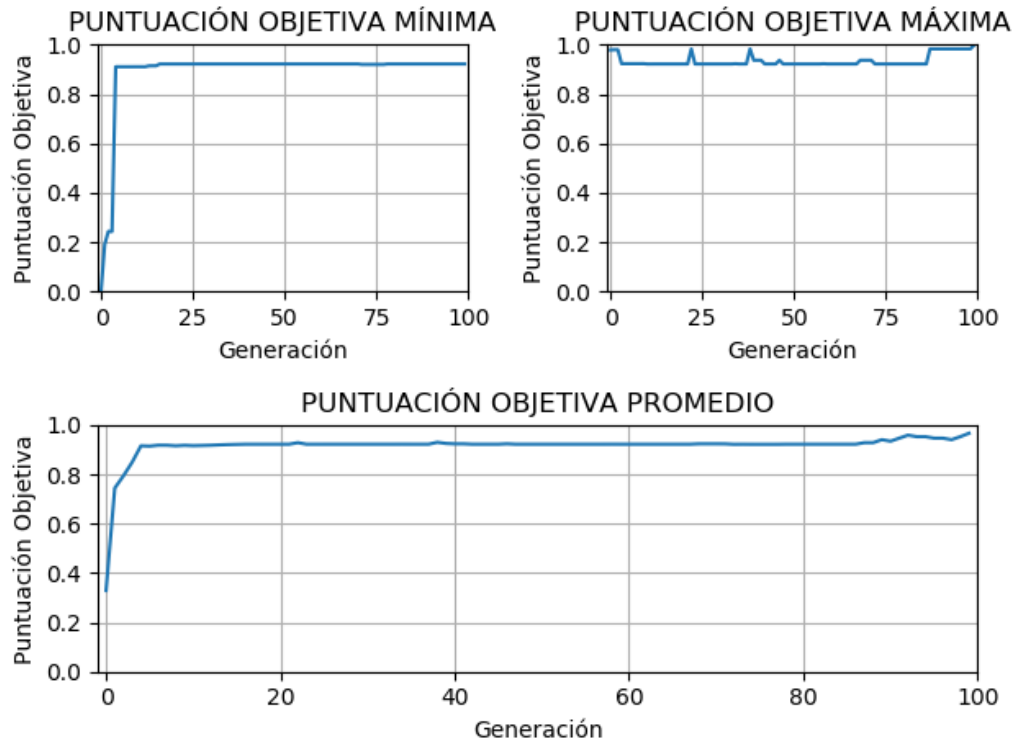
## Python: Matplotlib

### Algoritmo en 20 Generaciones



Número de Generación	Mínimo	Máximo	Promedio
Generación 0	0.04136325621646656	0.8787327844870441	0.27614223876953925
Generación 1	0.2218140906579105	0.8787327844870441	0.5590941751644709
Generación 2	0.2218140906579105	0.8788508184451957	0.6880873504730107
Generación 3	0.2218140906579105	0.9876432034378259	0.7105151774541307
Generación 4	0.19138038199981344	0.9876432034378259	0.6880394742908388
Generación 5	0.19132530358517782	0.9881316361420756	0.7922285982682917
Generación 6	0.4725282449527984	0.9439619643546071	0.8508834954163321
Generación 7	0.8775770511910796	0.9439637523422957	0.8913506418735189
Generación 8	0.877571628022631	0.9439693769137989	0.904259116291855
Generación 9	0.8550738848932281	0.9675984568151124	0.9108839858184912
Generación 10	0.8695833896248739	0.9675984568151124	0.9198054779913676
Generación 11	0.8775699040588221	0.9745772988626644	0.9435825658991359
Generación 12	0.9439564013349167	0.974582951391214	0.9493878972582686
Generación 13	0.7166927016230302	0.974582951391214	0.9273575665835887
Generación 14	0.7166927016230302	0.9745678878251469	0.9242958153882732
Generación 15	0.9439564049543125	0.974582951391214	0.9531476451666109
Generación 16	0.9439564049543125	0.9745678289833203	0.9500855405787674
Generación 17	0.9439619643546071	0.9745678289833203	0.950087584824596
Generación 18	0.9439619643546071	0.9745678289833203	0.950843707730386
Generación 19	0.943961906444103	0.99005389572202	0.9523923144973881

## Algoritmo en 100 Generaciones



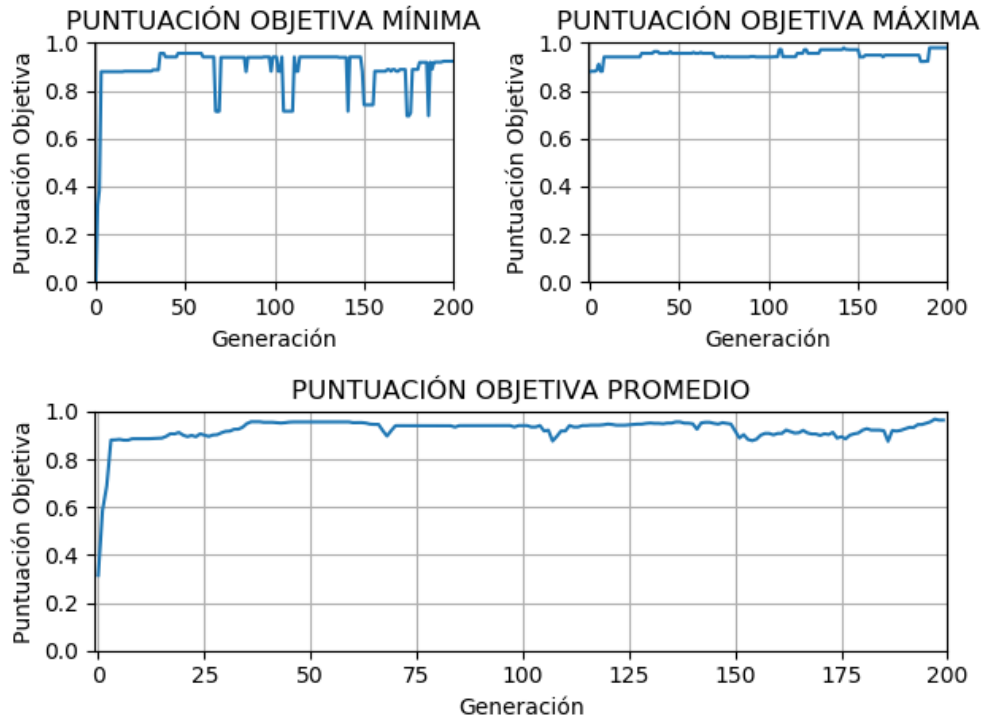
Número de Generación	Mínimo	Máximo	Promedio
Generación 0	0.0005246260358118554	0.9786779705077044	0.329109243093235
Generación 1	0.19035855844277258	0.9796443041836185	0.74396055562241
Generación 2	0.24323216996398675	0.9796442949656687	0.7939948021062727
Generación 3	0.24323219476699787	0.92311823636844	0.8492369519164177
Generación 4	0.910596893578787	0.92311823636844	0.9152952934098071
Generación 5	0.910596893578787	0.9231181898385384	0.9140433548636093
Generación 6	0.910596893578787	0.9231181898385384	0.9177996567112894
Generación 7	0.910596893578787	0.9231181898385384	0.9176446916600411
Generación 8	0.910596893578787	0.92311823636844	0.9154139765126088
Generación 9	0.9107024619315998	0.9232208430226426	0.9173516430753317
Generación 10	0.9107279910811472	0.9218138665760793	0.9159327584292034
Generación 11	0.9107134311386875	0.9218139148614438	0.9167025501845796
Generación 12	0.9105969415694752	0.9219311196863949	0.9176735768949378
Generación 13	0.9140839281785269	0.9219311196863949	0.9191431185339434
Generación 14	0.9144450312600445	0.9219311196863949	0.9202992315209924
Generación 15	0.9144450312600445	0.9219311196863949	0.9210975258262082
Generación 16	0.9217992415338697	0.9219311196863949	0.9218446719856213
Generación 17	0.9218138916129349	0.9219311196863949	0.9218695777722719
Generación 18	0.9218138916129349	0.9219311196863949	0.9218475972584894
Generación 19	0.9218138916129349	0.9219311196863949	0.9218695707972943
Generación 20	0.9218138916129349	0.9219310964364079	0.921868108021088
Generación 21	0.9218138916129349	0.9219310964364079	0.9218563900425556
Generación 22	0.9218138916129349	0.9827974163843114	0.9279547382276723
Generación 23	0.9218138916129349	0.9219164686758223	0.9218754337370916
Generación 24	0.9217992647821939	0.9219310964364079	0.9218856893903586

Generación 25	0.9217992647821939	0.9219310964364079	0.9218959492427927
Generación 26	0.9219164454260201	0.9219310982248684	0.9219179228673399
Generación 27	0.9219164454260201	0.9219164686758223	0.9219164574086106
Generación 28	0.9219164454260201	0.9219164686758223	0.9219164550836304
Generación 29	0.9219164454260201	0.9219164686758223	0.9219164581239891
Generación 30	0.9219164543682518	0.9219164686758223	0.9219164672450653
Generación 31	0.9219164543682518	0.9219164686758223	0.9219164672450653
Generación 32	0.9219164543682518	0.9219164686758223	0.9219164658143081
Generación 33	0.9219164543682518	0.9219164686758223	0.9219164672450653
Generación 34	0.9219164543682518	0.9228543680457514	0.922010255751301
Generación 35	0.9219164543682518	0.9219164686758223	0.9219164629527942
Generación 36	0.9219164543682518	0.9219164686758223	0.9219164658143082
Generación 37	0.9219164543682518	0.9219164686758223	0.9219164658143081
Generación 38	0.9219164543682518	0.9829033320640121	0.9295215131909227
Generación 39	0.9219164543682518	0.9369800790537787	0.9249291878898994
Generación 40	0.9219164543682518	0.9369800790537787	0.9234228254213466
Generación 41	0.9219164543682518	0.9369800790537787	0.9234228282828608
Generación 42	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 43	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 44	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 45	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 46	0.9219164686758223	0.9369800790537787	0.9234228297136179
Generación 47	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 48	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 49	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 50	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 51	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 52	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 53	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 54	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 55	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 56	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 57	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 58	0.9219164686758223	0.9228543680457514	0.9220102586128153
Generación 59	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 60	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 61	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 62	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 63	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 64	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 65	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 66	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 67	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 68	0.9219164686758223	0.9369800790537787	0.923422829713618
Generación 69	0.9219164686758223	0.9369800790537787	0.923422829713618
Generación 70	0.9219164686758223	0.9369800790537787	0.923422829713618
Generación 71	0.9200421004474414	0.9369800790537787	0.9232353928907798
Generación 72	0.9200421004474414	0.9219164686758223	0.9215415950301461

Generación 73	0.9200421004474414	0.9219164686758223	0.9217290318529843
Generación 74	0.9200421004474414	0.9219164686758223	0.9215415950301461
Generación 75	0.9200421004474414	0.9219164686758223	0.9215415950301461
Generación 76	0.9200421004474414	0.9219164686758223	0.9215415950301461
Generación 77	0.9200421004474414	0.9219164686758223	0.9215415950301461
Generación 78	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 79	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 80	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 81	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 82	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 83	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 84	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 85	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 86	0.9219164686758223	0.9219164686758223	0.9219164686758223
Generación 87	0.9219164686758223	0.9829033320640121	0.9280151550146412
Generación 88	0.9219164686758223	0.9829033320640121	0.9280151550146414
Generación 89	0.9219164686758223	0.9829033320640121	0.9402125276922793
Generación 90	0.9219164686758223	0.9829033320640121	0.9341138413534603
Generación 91	0.9219164686758223	0.9829033320640121	0.9463112140310983
Generación 92	0.9219164686758223	0.9829033320640121	0.9585085867087362
Generación 93	0.9219164686758223	0.9829033320640121	0.9524099003699172
Generación 94	0.9219164686758223	0.9829033320640121	0.9524099003699172
Generación 95	0.9219164686758223	0.9829033320640121	0.9463112140310983
Generación 96	0.9219164686758223	0.9829033320640121	0.9463112140310983
Generación 97	0.9219164686758223	0.9829033320640121	0.9402125276922793
Generación 98	0.9219164686758223	0.9829033320640121	0.9524099003699172
<b>Generación 99</b>	<b>0.9219164686758223</b>	<b>0.998455223692878</b>	<b>0.9661624622104418</b>



## Algoritmo en 200 Generaciones



Número de Generación	Mínimo	Máximo	Promedio
Generación 0	0.002277189352044981	0.879173799643822	0.3146832063053033
Generación 1	0.31959097399855274	0.8810205015964007	0.5856649153341211
Generación 2	0.381358639544012	0.8810205015964007	0.6884245744599695
Generación 3	0.8791738083763	0.8810205068413873	0.8802115823925238
Generación 4	0.879173799643822	0.8865143918950918	0.8809441996042452
Generación 5	0.879173799643822	0.910582078736037	0.8829214757202848
Generación 6	0.879173799643822	0.881006086683685	0.8796374840872184
Generación 7	0.879173799643822	0.881006086683685	0.8795172891446628
Generación 8	0.8791165801396651	0.9406463520392986	0.8855844159123955
Generación 9	0.879173799643822	0.9406463520392986	0.8855500728819591
Generación 10	0.879173799643822	0.9406463520392986	0.8857333015859454
Generación 11	0.879173799643822	0.9406463520392986	0.8857333015859453
Generación 12	0.879173799643822	0.9406463520392986	0.8858707409953286
Generación 13	0.879173799643822	0.9406463520392986	0.8867880289244419
Generación 14	0.879173799643822	0.9406463520392986	0.8879812804768583
Generación 15	0.879173799643822	0.9406463520392986	0.8884399482832727
Generación 16	0.881006086683685	0.9406463520392986	0.8955044914519339
Generación 17	0.881006086683685	0.9406463520392986	0.9065146843893525
Generación 18	0.881006086683685	0.9406463520392986	0.9059638538682118
Generación 19	0.881006086683685	0.9406463520392986	0.9119278804037732
Generación 20	0.881006086683685	0.9406463520392986	0.9005453172776093
Generación 21	0.881006086683685	0.9406463520392986	0.8943972990986072
Generación 22	0.881006086683685	0.9406463520392986	0.899621162893405
Generación 23	0.881006086683685	0.9406463520392986	0.8938464685774665
Generación 24	0.881006086683685	0.9406463520392986	0.9061360199501072

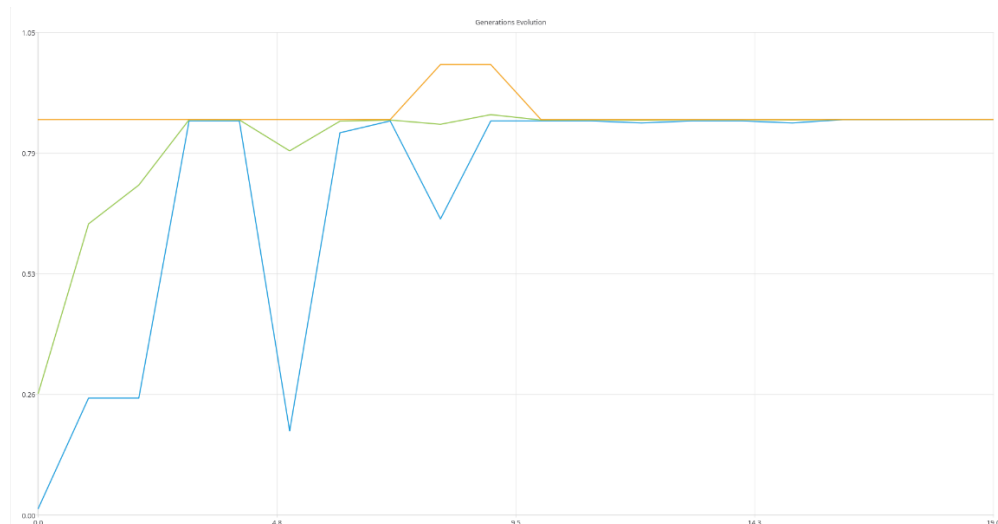
Generación 25	0.881006086683685	0.9406463520392986	0.9003613256341685
Generación 26	0.881006086683685	0.9406463520392986	0.8952291477169375
Generación 27	0.881006086683685	0.9406463520392986	0.9011931742524988
Generación 28	0.881006086683685	0.9406463520392986	0.9025787620986403
Generación 29	0.881006086683685	0.9558615941600204	0.9104429772855192
Generación 30	0.881006086683685	0.9558615941600204	0.9164924530399897
Generación 31	0.881006086683685	0.9558615941600204	0.9176860860621335
Generación 32	0.8865143918950918	0.9558615941600204	0.9251662962335852
Generación 33	0.8865143918950918	0.9558615941600204	0.9254476481167864
Generación 34	0.8865143918950918	0.9558615941600204	0.9337199009119104
Generación 35	0.8865143918950918	0.9558615941600204	0.9489268739335275
Generación 36	0.9558615941600204	0.9635149915876541	0.9566269339027837
Generación 37	0.9558615941600204	0.9635149915876541	0.9566269339027837
Generación 38	0.9558615941600204	0.9635149915876541	0.9566269339027837
Generación 39	0.9406463520392986	0.9558615941600204	0.9543400699479481
Generación 40	0.9406463520392986	0.9558615941600204	0.9543400699479481
Generación 41	0.9406463520392986	0.9558615941600204	0.9543400699479481
Generación 42	0.9406463520392986	0.9558615941600204	0.952818545735876
Generación 43	0.9406463520392986	0.9558615941600204	0.9512970215238038
Generación 44	0.9406463520392986	0.9558615941600204	0.952818545735876
Generación 45	0.9406463520392986	0.9635149915876541	0.9551054096907116
Generación 46	0.9558615941600204	0.9558615941600204	0.9558615941600201
Generación 47	0.9558615941600204	0.9558615941600204	0.9558615941600201
Generación 48	0.9558615941600204	0.9558615941600204	0.9558615941600201
Generación 49	0.9558615941600204	0.9558615941600204	0.9558615941600201
Generación 50	0.9558615941600204	0.9558615941600204	0.9558615941600201
Generación 51	0.9558615941600204	0.9558615941600204	0.9558615941600201
Generación 52	0.9558615941600204	0.9558615941600204	0.9558615941600201
Generación 53	0.9558615941600204	0.9558615941600204	0.9558615941600201
Generación 54	0.9558615941600204	0.9558615941600204	0.9558615941600201
Generación 55	0.9558615941600204	0.9558615941600204	0.9558615941600201
Generación 56	0.9558615941600204	0.9558615941600204	0.9558615941600201
Generación 57	0.9558615941600204	0.9558615941600204	0.9558615941600201
Generación 58	0.9558615941600204	0.9596844781765644	0.9562438825616747
Generación 59	0.9558615941600204	0.9558615941600204	0.9558615941600201
Generación 60	0.9406463520392986	0.9558615941600204	0.952818545735876
Generación 61	0.9406463520392986	0.9558615941600204	0.952818545735876
Generación 62	0.9406463520392986	0.9596844781765644	0.9532008341375302
Generación 63	0.9406463520392986	0.9558615941600204	0.9512970215238038
Generación 64	0.9406463520392986	0.9558615941600204	0.9467324488875872
Generación 65	0.9406463520392986	0.9558615941600204	0.9455901613193513
Generación 66	0.9406463520392986	0.9558615941600204	0.9455901613193511
Generación 67	0.7138040406386721	0.9558615941600204	0.9196735495355217
Generación 68	0.7121548590679002	0.9558615941600204	0.8968244002383818
Generación 69	0.7138040406386721	0.9558615941600204	0.9196735495355215
Generación 70	0.938753029843072	0.9406463520392986	0.9402676876000532
Generación 71	0.938753029843072	0.9406463520392986	0.9402676876000531
Generación 72	0.938753029843072	0.9406463520392986	0.9400783553804306

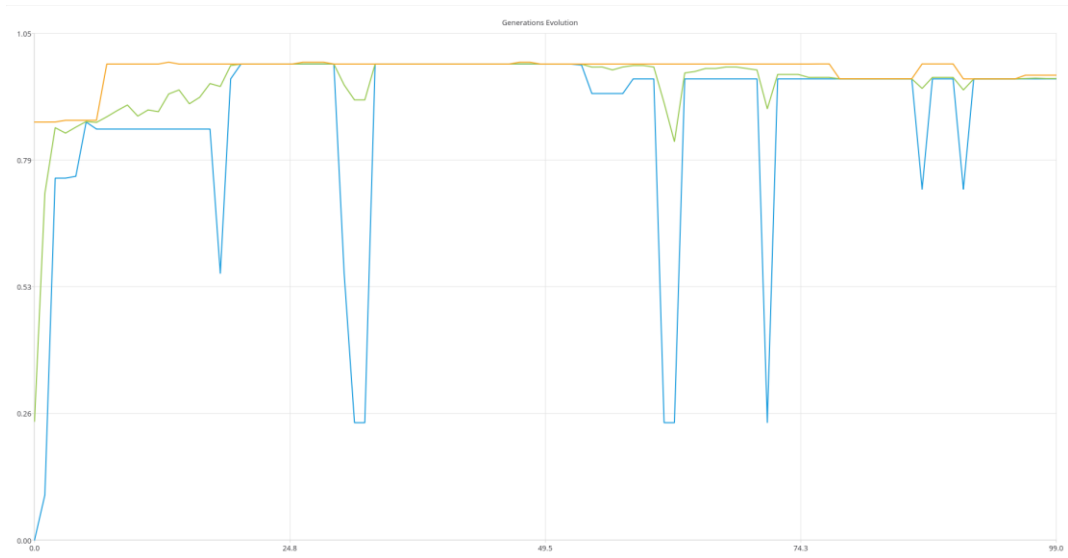
Generación 73	0.938753029843072	0.9406463520392986	0.939889023160808
Generación 74	0.938753029843072	0.9425415815841615	0.9402678783349169
Generación 75	0.938753029843072	0.9406463520392986	0.9398890231608078
Generación 76	0.938753029843072	0.9406463520392986	0.9402676876000531
Generación 77	0.938753029843072	0.9425415815841615	0.9400785461152943
Generación 78	0.938753029843072	0.9406463520392986	0.9398890231608078
Generación 79	0.938753029843072	0.9406463520392986	0.9400783553804304
Generación 80	0.938753029843072	0.9406463520392986	0.9400783553804306
Generación 81	0.938753029843072	0.9406463520392986	0.9398890231608078
Generación 82	0.938753029843072	0.9406463520392986	0.9400783553804304
Generación 83	0.938753029843072	0.9406463520392986	0.9398890231608078
Generación 84	0.879173799643822	0.9406463520392986	0.9343097645801283
Generación 85	0.938753029843072	0.9406463520392986	0.9404570198196758
Generación 86	0.938753029843072	0.9406463520392986	0.9404570198196758
Generación 87	0.938753029843072	0.9406463520392986	0.9404570198196758
Generación 88	0.938753029843072	0.9406463520392986	0.9404570198196758
Generación 89	0.938753029843072	0.9406463520392986	0.9404570198196758
Generación 90	0.938753029843072	0.9425415815841615	0.940646542774162
Generación 91	0.938753029843072	0.9425415815841615	0.9404572105545395
Generación 92	0.938753029843072	0.9425415815841615	0.9404572105545395
Generación 93	0.938753029843072	0.9406463520392986	0.9404570198196758
Generación 94	0.9406463520392986	0.9406463520392986	0.9406463520392985
Generación 95	0.9406463520392986	0.9406463520392986	0.9406463520392985
Generación 96	0.9406463520392986	0.9406463520392986	0.9406463520392985
Generación 97	0.9406463520392986	0.9406463520392986	0.9406463520392985
Generación 98	0.881006086683685	0.9406463520392986	0.9346823255037371
Generación 99	0.9406463520392986	0.9406463520392986	0.9406463520392985
Generación 100	0.9406463520392986	0.9406463520392986	0.9406463520392985
Generación 101	0.9406463520392986	0.9406463520392986	0.9406463520392985
Generación 102	0.881006086683685	0.9406463520392986	0.9346823255037371
Generación 103	0.881006086683685	0.9415937283931505	0.9347770631391222
Generación 104	0.9406463520392986	0.9415937283931505	0.9408358273100689
Generación 105	0.7138040406386721	0.9415937283931505	0.9181515961700063
Generación 106	0.7138040406386721	0.9721615417364124	0.921397852775103
Generación 107	0.7138040406386721	0.9721615417364124	0.8760293904949774
Generación 108	0.7138040406386721	0.9415937283931505	0.8956568403007141
Generación 109	0.7138040406386721	0.9415937283931505	0.9182463338053914
Generación 110	0.7138040406386721	0.9415937283931505	0.9180568585346209
Generación 111	0.9406463520392986	0.9415937283931505	0.9408358273100689
Generación 112	0.881006086683685	0.9415937283931505	0.9347770631391222
Generación 113	0.881006086683685	0.9415937283931505	0.9347770631391222
Generación 114	0.9406463520392986	0.9415937283931505	0.9407410896746837
Generación 115	0.9406463520392986	0.9415937283931505	0.9407410896746835
Generación 116	0.9406463520392986	0.9558615941600204	0.9422626138867558
Generación 117	0.9406463520392986	0.9558615941600204	0.9422626138867558
Generación 118	0.9406463520392986	0.9558615941600204	0.942357351522141
Generación 119	0.9406463520392986	0.9558615941600204	0.9439736133695986
Generación 120	0.9406463520392986	0.9711989065934695	0.9469341311896304

Generación 121	0.9406463520392986	0.9711989065934695	0.9452231317067877
Generación 122	0.9406463520392986	0.9558615941600204	0.9421678762513708
Generación 123	0.9406463520392986	0.9558615941600204	0.9421678762513708
Generación 124	0.9406463520392986	0.9558615941600204	0.9421678762513708
Generación 125	0.9406463520392986	0.9558615941600204	0.9436894004634429
Generación 126	0.9406463520392986	0.9558615941600204	0.9452109246755149
Generación 127	0.9406463520392986	0.9558615941600204	0.9467324488875872
Generación 128	0.9406463520392986	0.9558615941600204	0.9467324488875871
Generación 129	0.9406463520392986	0.9711989065934695	0.9497877043430044
Generación 130	0.9406463520392986	0.9711989065934695	0.9513214355863491
Generación 131	0.9406463520392986	0.9711989065934695	0.9497877043430043
Generación 132	0.9406463520392986	0.9711989065934695	0.9497877043430044
Generación 133	0.9406463520392986	0.9711989065934695	0.9480753220324004
Generación 134	0.9406463520392986	0.9711989065934695	0.9511427845190902
Generación 135	0.9406463520392986	0.9711989065934695	0.9511280979345902
Generación 136	0.938753029843072	0.9711989065934695	0.9559094686107932
Generación 137	0.938753029843072	0.9711989065934695	0.9559094686107932
Generación 138	0.938753029843072	0.9711989065934695	0.9509372398360585
Generación 139	0.938753029843072	0.9711989065934695	0.9505580031922222
Generación 140	0.938753029843072	0.9711989065934695	0.9480770386461732
Generación 141	0.7138040406386721	0.9711989065934695	0.9254032979236104
Generación 142	0.9406463520392986	0.9789133391774669	0.9520104912917843
Generación 143	0.9406463520392986	0.9721615417364124	0.9543905670030959
Generación 144	0.9406463520392986	0.9721615417364124	0.9542943034888017
Generación 145	0.9406463520392986	0.9711989065934695	0.949906856040935
Generación 146	0.9406463520392986	0.9711989065934695	0.9467568629501327
Generación 147	0.9406463520392986	0.9711989065934695	0.9528673738609668
Generación 148	0.9406463520392986	0.9711989065934695	0.9498121184055499
Generación 149	0.881006086683685	0.9711989065934695	0.9469033473254056
Generación 150	0.7404503451855672	0.9711989065934695	0.9177179802737812
Generación 151	0.7404503451855672	0.9406463520392986	0.8886790975974297
Generación 152	0.7404503451855672	0.9406463520392986	0.9027346717472413
Generación 153	0.7404503451855672	0.9406463520392986	0.8827150710618683
Generación 154	0.7404503451855672	0.9482387143105685	0.8775102807534338
Generación 155	0.7404503451855672	0.9482387143105685	0.8834743072889951
Generación 156	0.881006086683685	0.9482387143105685	0.9004166387446231
Generación 157	0.881006086683685	0.9482387143105685	0.9078991377344385
Generación 158	0.881006086683685	0.9482387143105685	0.9078991377344385
Generación 159	0.881006086683685	0.9482387143105685	0.9019106971363318
Generación 160	0.881006086683685	0.9482387143105685	0.9078991377344385
Generación 161	0.881006086683685	0.9482387143105685	0.907874723671893
Generación 162	0.881006086683685	0.9482387143105685	0.9220316572993058
Generación 163	0.8883543083295002	0.9482387143105685	0.9160188026386535
Generación 164	0.8883543083295002	0.9406463520392986	0.9092711258134196
Generación 165	0.881006086683685	0.9482387143105685	0.9107743394785827
Generación 166	0.8883543083295002	0.9482387143105685	0.9204887707825062
Generación 167	0.8883543083295002	0.9482387143105685	0.9100303620405465
Generación 168	0.881006086683685	0.9482387143105685	0.9055451351076028

Generación 169	0.881006086683685	0.9482387143105685	0.9055451351076028
Generación 170	0.8883543083295002	0.9482387143105685	0.899571953298587
Generación 171	0.8883543083295002	0.9482387143105685	0.9063196301238208
Generación 172	0.8883543083295002	0.9482387143105685	0.9033009957622218
Generación 173	0.8883543083295002	0.9482387143105685	0.9122592425968366
Generación 174	0.6941397467994102	0.9482387143105685	0.8883281454483383
Generación 175	0.6941397467994102	0.9482387143105685	0.8943165860464453
Generación 176	0.7072187584474029	0.9482387143105685	0.8852795081213063
Generación 177	0.8883543083295002	0.9482387143105685	0.9004664960952116
Generación 178	0.8883543083295002	0.9482387143105685	0.9063140412209331
Generación 179	0.8883543083295002	0.9482387143105685	0.9093774406836446
Generación 180	0.8883543083295002	0.9482387143105685	0.9217289808869669
Generación 181	0.9180523706945796	0.9482387143105685	0.927670119238892
Generación 182	0.9180523706945796	0.9482387143105685	0.922007509522803
Generación 183	0.9180523706945796	0.9482387143105685	0.9220075095228031
Generación 184	0.9180523706945796	0.9482387143105685	0.9217264437418976
Generación 185	0.9180523706945796	0.9217989607656691	0.9193632480660178
Generación 186	0.6949536087487165	0.9217989607656691	0.8747434956768452
Generación 187	0.9189883029566134	0.9217989607656691	0.9201125660802356
Generación 188	0.889274981802443	0.9227368003763392	0.9177035562614936
Generación 189	0.9180523706945796	0.9227368003763392	0.9207686723379774
Generación 190	0.9189883029566134	0.9789133391774669	0.9270423371084002
Generación 191	0.9189883029566134	0.9789133391774669	0.9330348407304856
Generación 192	0.9189883029566134	0.9789133391774669	0.93275377494958
Generación 193	0.9189883029566134	0.9789133391774669	0.9447387821937507
Generación 194	0.9217989607656691	0.9789133391774669	0.9451136319357234
Generación 195	0.9227368003763392	0.9789133391774669	0.9508250697769031
Generación 196	0.9227368003763392	0.9789133391774669	0.9564427236570159
Generación 197	0.9227368003763392	0.9789133391774669	0.9676780314172413
Generación 198	0.9227368003763392	0.9789133391774669	0.9635674058995644
<b>Generación 199</b>	<b>0.9227368003763392</b>	<b>0.9789133391774669</b>	<b>0.9635674058995644</b>

## C++: Qt Charts





# Conclusión

---

Al ser el espacio solución tan grande y nuestra cantidad de cromosomas por población tan chico, notamos que para llegar a la mejor solución se requiere de una mayor cantidad de iteraciones o corridas del programa. Ya que mientras más sean las iteraciones, mas son las posibilidades de que se salga de un máximo local a partir de una oportuna mutación que agregue diversidad genética y permita mejores oportunidades de crossover.

Otro acercamiento al problema podría haber sido el utilizar una mayor cantidad de cromosomas por población, lo cual nos daría una gráfica parecida a una exponencial y nos provee la mejor solución con mayor certeza por cada ejecución del programa, debido a la mayor diversidad genética. Pero no necesariamente seria más eficiente ya que aumenta considerablemente el tiempo de corrida del mismo, además del consumo mucho mayor de otros recursos como memoria RAM (En este punto la implementación de Python y C++ difieren ya que en la primera se utiliza un método más eficiente en consumo de memoria inspirado en el uso del fitness relativo acumulado, en cambio la segunda se vale del tradicional método de crear una ruleta con casillas proporcionales al fitness de cada cromosoma). En la práctica al parecer el primer método más innovador tiene además mejores resultados en cuanto a la calidad de las corridas.

También se puede notar que optamos por mostrar la información de las corridas en dos formatos distintos: La primera forma, utilizada en Matplotlib (Python), fue mostrar por separado las graficas de valores de la función objetivo máximos, mínimos y promedios. A diferencia de la segunda, utilizada en Qt Charts (C++), que muestra todos los valores juntos en la misma grafica. Lo que tiene como ventaja una mejor vista de la convergencia de los valores. En cambio la primera permite ver en mucho más detalle los valores específicos y no tanto un panorama general.