INSTITUTO POLITÉCNICO NACIONAL

Centro de Investigación en Computación

ASIGNATURA:

Metaheurísticas

Actividad #26:

Guía Taller No.15

"Solución de problemas mediante

Algoritmos Genéticos"

PROFESORA:

Dra. Yenny Villuendas Rey

PRESENTA:

Juan René Hernández Sánchez

Adriana Montserrat García Carrillo

1. Introducción

Los algoritmos genéticos (AG) son algoritmos inspirados en la selección natural que han logrado semejar todo el proceso evolutivo de los seres vivos para resolver problemas de optimización, búsqueda y aprendizaje en las máquinas

La teoría de los algoritmos genéticos fue desarrollada por John H. Holland, sus colegas y sus estudiantes en la Universidad de Michigan, en un proyecto que buscaba explicar el proceso de adaptación natural de los sistemas y diseñar sistemas artificiales que conservaran los más importantes mecanismos de los sistemas naturales. Casi todos los conceptos de los algoritmos genéticos están basados en conceptos de biología y genética.

Las características de los seres vivos se encuentran registradas en los cromosomas y los genes, lo que hace que cada individuo sea diferente dentro de una población. En algoritmos genéticos los cromosomas se representan mediante cadenas (strings) y cada posición de la cadena representa un gen.

El primer paso desarrollar un algoritmo genético es codificar el parámetro deseado como una cadena y usar un alfabeto. La longitud de un cromosoma está determinada por el número de genes que lo conforman.

El conjunto ordenado de genes de un cromosoma se conoce como genotipo. De acuerdo con los valores que tengan los genes de una persona podremos decir que es alta o baja, de ojos azules o verdes, de cabello liso o rizado, etc. Esto se denomina fenotipo, y en los algoritmos genéticos corresponde a la decodificación de un cromosoma.

Cada cromosoma es una solución, buena o mala, del problema. A medida que el algoritmo genético se va desarrollando las soluciones se acercan cada vez más a la óptima. Simulando los sistemas de selección natural a los que se encuentran sometidos los organismos vivos, los algoritmos genéticos utilizan tres operadores básicos para solucionar el problema: Selección, Cruzamiento y Mutación.

2. Desarrollo

Asignatura: Metaheurísticas
Actividad No.26
Guía Taller No.15
Título: Solución de problemas mediante Algoritmos Genéticos

Contenido:

- Métodos heurísticos de solución de problemas.
- Algoritmos Genéticos

Objetivo: Implementar algoritmos genéticos, en lenguajes de alto nivel, para la solución de problemas de competencia.

1. Analice detalladamente las seis funciones definidas en el documento "Funciones de prueba.pdf".

Nombre de la función	Ref.	Fórmula	Punto mínimo	Valor mínimo
Alpine 1 Function	[1]	$f_1(x) = \sum_{i=1}^{D} \lvert x_i \sin(x_i) + 0.1x_i \rvert$	x* = f(0,0)	f(x*)=0
Dixon & Price Function	[2]	$f_2(x) = (x_1 - 1)^2 + \sum_{i=2}^{D} i(2\sin(x_i) - x_{i-1})^2$	x* = f(2($\frac{2^i-2}{2^i}$))	f(x*)=0
Quintic Function	[3]	$f_3(x) = \sum_{i=1}^{D} \lvert x_i^5 - 3x_i^4 + 4x_i^3 - 2x_i^2 - 10x_1 - 4 \rvert$	x* = f(-1 or 2)	f(x*)=0
Schwefel 2.23 Function	[4]	$f_4(x) = \sum_{i=1}^{D} x_i^{10}$	x* = f(0,0)	f(x*)=0
Streched V Sine Wave Function	[5]	$f_5(x) = \sum_{i=1}^{D-1} (x_{i+1}^2 + x_i^2)^{0.25} \left[sin^2 \left\{ 50(x_{i+1}^2 + x_i^2)^{0.1} \right\} + 0.1 \right]$	x* = f(0,0)	f(x*)=0
Sum Squares Function	[6]	$f_6(x) = \sum_{i=1}^{D} i x_i^2$	x* = f(0,0)	f(x*)=0

Figura 1. Descripción de las 6 funciones

2. Implemente dichas funciones.

```
#Alpine 1 Function
    def calcular_aptitud1(self):
        valmax = 0.0
        for i in self.cromosoma:
            valmax += abs(i*(math.sin(i))+0.1*(i))
        self.aptitud = valmax

#Dixon & Price Function
    def calcular_aptitud2(self):
        valmax = (self.cromosoma[0] - 1)**2
        for i in range(1, len(self.cromosoma)):
            valmax += i*(2*math.sin(self.cromosoma[i])-self.cromosoma[i-
1])**2
        self.aptitud = valmax

#Quintic Function
    def calcular_aptitud3(self):
        valmax = 0.0
        for i in self.cromosoma:
            valmax += abs((i**5)-(3*(i**4))+(4*(i**3))-(2*(i**2))-(10*i)-4)
        self.aptitud = valmax

#Schwefel 2.23 Function
    def calcular_aptitud4(self):
        valmax = 0.0
        for i in self.cromosoma:
```

```
            valmax += i**10
        self.aptitud = valmax

#Streched V Sine Wave Function
    def calcular_aptitud5(self):
        valmax = 0.0
        for i in range(len(self.cromosoma)-1):
            valmax += ((self.cromosoma[i+1]**2 +
self.cromosoma[i]**2)**0.25)*((math.sin(50*((self.cromosoma[i+1]**2 +
self.cromosoma[i]**2)**0.1))**2)+0.1)
        self.aptitud = valmax

#Sum Squares Function
    def calcular_aptitud6(self):
        valmax = 0.0
        for j, i in enumerate(self.cromosoma):
            valmax += j * (i**2)
        self.aptitud = valmax
```

3. Implemente al menos 6 versiones de Algoritmos Genéticos (3 de modelo estacionario, 3 de modelo generacional) para la solución de los problemas de minimización de las funciones anteriores. Considere y posteriormente dimensiones.

- **Modelo Estacionario**

- **Versión 1:**

```
import math
import random
from time import time
import statistics

tam_cromosoma = 10
tam_poblacion = 10
k_torneo = 2  # tam de la poblacion del torneo
humbral = 0.1
prob_cruzamiento = 0.5

class Gen:
    cromosoma = []
    aptitud = float
    generacion = int

    def __init__(self, generacion, cromosoma):
        self.cromosoma = cromosoma
        self.fitness6()
```

```
        self.generacion = generacion

#*****************************************************************************

    # Alpine 1 Function
    def fitness1(self):
        valmax = 0.0
        for i in self.cromosoma:
            valmax += abs(i*(math.sin(i))+0.1*(i))
        self.aptitud = valmax


    # Dixon & Price Function
    def fitness2(self):
        valmax = (self.cromosoma[0] - 1)**2
        for i in range(1, tam_cromosoma):
            valmax += i*(2*math.sin(self.cromosoma[i])-self.cromosoma[i-
1])**2
        self.aptitud = valmax

    # Quintic Function
    def fitness3(self):
        valmax = 0
        for i in self.cromosoma:
            valmax += abs((i**5)-(3*(i**4))+(4*(i**3))-(2*(i**2))-(10*i)-4)
        self.aptitud = valmax

    # Schwefel 2.23 Function
    def fitness4(self):
        valmax = 0
        for i in self.cromosoma:
            valmax += i**10
        self.aptitud = valmax

    # Streched V Sine Wave Function
    def fitness5(self):
        valmax = 0
        for i in range(tam_cromosoma - 1):
            valmax += ((self.cromosoma[i+1]**2 +
self.cromosoma[i]**2)**0.25) * \
                ((math.sin(50*((self.cromosoma[i+1]**2 +
self.cromosoma[i]**2)**0.1))**2)+0.1)
        self.aptitud = valmax
```

```
    # Sum Squares Function
    def fitness6(self):
        valmax = 0
        for j, xi in enumerate(self.cromosoma):
            valmax += j * (xi**2)
        self.aptitud = valmax

#*****************************************************************************

    def mutacion(self):
        for i in range(tam_cromosoma):
            if random.random() < humbral:
                self.cromosoma[i] = random.uniform(-10, 10)

    def __str__(self):
        cromosoma_str = [round(x, 2) for x in self.cromosoma]
        return "< Apt: " + str(round(self.aptitud,2)) + " Gen: " +
str(self.generacion) + " >"


# Con Valores de Representaciones de Orden
def generar_cromosoma():
    cromosoma = [random.uniform(-10, 10) for _ in range(tam_cromosoma)]
    random.shuffle(cromosoma)
    return cromosoma

def seleccion_torneo(poblacion):
    torneo = []
    for i in range(k_torneo):
        seleccion = random.randint(0, tam_poblacion - 1)
        gen = poblacion[seleccion]
        torneo.append(gen)
    return min(torneo, key=lambda x: x.aptitud)

def uniform_crossover(gen1, gen2):
    descendienteA = []
    descendienteB = []

    for j in range(tam_cromosoma):
        if random.random() < 0.5:
            descendienteA.append(gen1.cromosoma[j])
            descendienteB.append(gen2.cromosoma[j])
        else:
            descendienteA.append(gen2.cromosoma[j])
            descendienteB.append(gen1.cromosoma[j])
```

```
    return descendienteA, descendienteB

def elitism_replacement(poblacion, descendiente):
    # Seleccion de Peor (Minimizar)
    reemplazo = max(poblacion, key=lambda x: x.aptitud)

    if descendiente.aptitud < reemplazo.aptitud:
        poblacion.pop(poblacion.index(reemplazo))
        poblacion.append(descendiente)

#*****************************************************************************

def GA():
    poblacion = []
    generacion_global = 1
    evaluaciones = 0

    # Generacion Inicial aleatoria
    for _ in range(tam_poblacion):
        poblacion.append(Gen(generacion_global, generar_cromosoma()))
        evaluaciones += 1

    # Iteracion para 500 evaluaciones
    while evaluaciones < 500:
        # Generando nueva poblacion
        generacion_global += 1
        gen1 = seleccion_torneo(poblacion)
        gen2 = seleccion_torneo(poblacion)
        # Probabilidad de Cruzamiento.
        if random.random() > prob_cruzamiento:
            # Cruzamiento de los padres
            crom_des1, crom_des2 = uniform_crossover(gen1, gen2)
            descendiente1 = Gen(generacion_global, crom_des1)
            descendiente2 = Gen(generacion_global, crom_des2)
            # Mutacion del descendiente
            descendiente1.mutacion()
            descendiente2.mutacion()

            descendiente1.fitness6()
            descendiente2.fitness6()
            evaluaciones += 2
            # Remplazo Elitita
            elitism_replacement(poblacion, descendiente1)
            elitism_replacement(poblacion, descendiente2)
```

```
    return min(poblacion, key=lambda x: x.aptitud)

aptitudes = []
tiempo = []

for i in range(20):
    start_time = time()
    aptitudes.append(GA().aptitud)
    tiempo.append(time() - start_time)

mejorA = min(aptitudes)
peorA = max(aptitudes)
meanA = statistics.mean(aptitudes)
medianA = statistics.median(aptitudes)
sigmaA = statistics.pstdev(aptitudes)

mejorT = min(tiempo)
peorT = max(tiempo)
meanT = statistics.mean(tiempo)
medianT = statistics.median(tiempo)
sigmaT = statistics.pstdev(tiempo)

print("aptitud: ")
print(str(round(mejorA,2)) + " " + str(round(peorA,2)) + " " +
str(round(meanA,2)) + " " + str(round(medianA,2)) + " " +
str(round(sigmaA,2)))
print("Tiempo: ")
print(str(round(mejorT,2)) + " " + str(round(peorT,2)) + " " +
str(round(meanT,2)) + " " + str(round(medianT,2)) + " " +
str(round(sigmaT,2)))
```

- **Versión 2:**

```
import math
import random
from time import time
import statistics

tam_cromosoma = 10
tam_poblacion = 10
humbral = 0.1
prob_cruzamiento = 0.5

class Gen:
    cromosoma = []
```

```
    aptitud = float
    generacion = int

    def __init__(self, generacion, cromosoma):
        self.cromosoma = cromosoma
        self.fitness4()
        self.generacion = generacion

#*********************************************************************

    # Alpine 1 Function
    def fitness1(self):
        valmax = 0.0
        for i in self.cromosoma:
            valmax += abs(i*(math.sin(i))+0.1*(i))
        self.aptitud = valmax

    # Dixon & Price Function
    def fitness2(self):
        valmax = (self.cromosoma[0] - 1)**2
        for i in range(1, tam_cromosoma):
            valmax += i*(2*math.sin(self.cromosoma[i])-self.cromosoma[i-
1])**2
        self.aptitud = valmax

    # Quintic Function
    def fitness3(self):
        valmax = 0
        for i in self.cromosoma:
            valmax += abs((i**5)-(3*(i**4))+(4*(i**3))-(2*(i**2))-(10*i)-4)
        self.aptitud = valmax

    # Schwefel 2.23 Function
    def fitness4(self):
        valmax = 0
        for i in self.cromosoma:
            valmax += i**10
        self.aptitud = valmax

    # Streched V Sine Wave Function
    def fitness5(self):
        valmax = 0
        for i in range(tam_cromosoma - 1):
            valmax += ((self.cromosoma[i+1]**2 +
self.cromosoma[i]**2)**0.25) * \
```

```
                ((math.sin(50*((self.cromosoma[i+1]**2 +
self.cromosoma[i]**2)**0.1))**2)+0.1)
        self.aptitud = valmax

    # Sum Squares Function
    def fitness6(self):
        valmax = 0
        for j, xi in enumerate(self.cromosoma):
            valmax += j * (xi**2)
        self.aptitud = valmax

#*****************************************************************

    def mutacion(self):
        for i in range(tam_cromosoma):
            if random.random() < humbral:
                self.cromosoma[i] = random.uniform(-10, 10)

    def __str__(self):
        cromosoma_str = [round(x, 2) for x in self.cromosoma]
        return "< Crom: " + str(cromosoma_str) + " Apt: " +
str(round(self.aptitud,2)) + " Gen: " + str(self.generacion) + " >"

# Con Valores de Representaciones de Orden
def generar_cromosoma():
    cromosoma = [random.uniform(-10, 10) for _ in range(tam_cromosoma)]
    random.shuffle(cromosoma)
    return cromosoma

def seleccion_proporcional(poblacion):

    probabilidad = []
    probTotal = 0
    flecha = random.random()
    ruleta = [0]

    for i in poblacion:
        probTotal += i.aptitud

    for i in poblacion:
        probabilidad.append(i.aptitud / probTotal)

    for i in range(tam_poblacion):
        ruleta.append(probabilidad[i]+ruleta[i])
```

```
    ruleta.append(flecha)
    ruleta.sort()
    posicion = ruleta.index(flecha)
    apareamiento = poblacion[posicion-1]

    return apareamiento

def one_point_crossover(gen1, gen2):
    descendienteA = []
    descendienteB = []

    punto_corte = random.randint(1, tam_cromosoma-1)

    descendienteA = gen1.cromosoma[:punto_corte] +
gen2.cromosoma[punto_corte:]
    descendienteB = gen2.cromosoma[:punto_corte] +
gen1.cromosoma[punto_corte:]

    return descendienteA, descendienteB

def random_replacement(poblacion, descendiente):
    reemplazo = random.randint(0, tam_poblacion-1)
    poblacion.pop(reemplazo)
    poblacion.append(descendiente)

#*****************************************************************************

def GA():
    poblacion = []
    generacion_global = 1
    evaluaciones = 0

    # Generacion Inicial aleatoria
    for _ in range(tam_poblacion):
        poblacion.append(Gen(generacion_global, generar_cromosoma()))
        evaluaciones += 1

    # Iteracion para 500 generaciones
    while evaluaciones < 500:
        # Generando nueva poblacion
        generacion_global += 1
        gen1 = seleccion_proporcional(poblacion)
        gen2 = seleccion_proporcional(poblacion)
        # Probabilidad de Cruzamiento.
        if random.random() > prob_cruzamiento:
```

```
            # Cruzamiento de los padres
            crom_des1, crom_des2 = one_point_crossover(gen1, gen2)
            descendiente1 = Gen(generacion_global, crom_des1)
            descendiente2 = Gen(generacion_global, crom_des2)
            # Mutacion del descendiente
            descendiente1.mutacion()
            descendiente2.mutacion()

            descendiente1.fitness4()
            descendiente2.fitness4()
            evaluaciones += 2
            # Remplazo Elitita
            random_replacement(poblacion, descendiente1)
            random_replacement(poblacion, descendiente2)

    return min(poblacion, key=lambda x: x.aptitud)

aptitudes = []
tiempo = []

for i in range(20):
    start_time = time()
    aptitudes.append(GA().aptitud)
    tiempo.append(time() - start_time)

mejorA = min(aptitudes)
peorA = max(aptitudes)
meanA = statistics.mean(aptitudes)
medianA = statistics.median(aptitudes)
sigmaA = statistics.pstdev(aptitudes)

mejorT = min(tiempo)
peorT = max(tiempo)
meanT = statistics.mean(tiempo)
medianT = statistics.median(tiempo)
sigmaT = statistics.pstdev(tiempo)

print("aptitud: ")
print(str(round(mejorA,2)) + " " + str(round(peorA,2)) + " " +
str(round(meanA,2)) + " " + str(round(medianA,2)) + " " +
str(round(sigmaA,2)))
print("Tiempo: ")
print(str(round(mejorT,2)) + " " + str(round(peorT,2)) + " " +
str(round(meanT,2)) + " " + str(round(medianT,2)) + " " +
str(round(sigmaT,2)))
```

➢ **Versión 3:**

```
import math
import random
from time import time
import statistics


tam_cromosoma = 10
tam_poblacion = 10
humbral = 0.1
prob_cruzamiento = 0.5

class Gen:
    cromosoma = []
    aptitud = float
    generacion = int

    def __init__(self, generacion, cromosoma):
        self.cromosoma = cromosoma
        self.fitness1()
        self.generacion = generacion

#*************************************************************************
    # Alpine 1 Function
    def fitness1(self):
        valmax = 0.0
        for i in self.cromosoma:
            valmax += abs(i*(math.sin(i))+0.1*(i))
        self.aptitud = valmax


    # Dixon & Price Function
    def fitness2(self):
        valmax = (self.cromosoma[0] - 1)**2
        for i in range(1, tam_cromosoma):
            valmax += i*(2*math.sin(self.cromosoma[i])-self.cromosoma[i-
1])**2
        self.aptitud = valmax

    # Quintic Function
    def fitness3(self):
        valmax = 0
        for i in self.cromosoma:
            valmax += abs((i**5)-(3*(i**4))+(4*(i**3))-(2*(i**2))-(10*i)-4)
        self.aptitud = valmax
```

```
    # Schwefel 2.23 Function
    def fitness4(self):
        valmax = 0
        for i in self.cromosoma:
            valmax += i**10
        self.aptitud = valmax

    # Streched V Sine Wave Function
    def fitness5(self):
        valmax = 0
        for i in range(tam_cromosoma - 1):
            valmax += ((self.cromosoma[i+1]**2 +
self.cromosoma[i]**2)**0.25) * \
                ((math.sin(50*((self.cromosoma[i+1]**2 +
self.cromosoma[i]**2)**0.1))**2)+0.1)
        self.aptitud = valmax

    # Sum Squares Function
    def fitness6(self):
        valmax = 0
        for j, xi in enumerate(self.cromosoma):
            valmax += j * (xi**2)
        self.aptitud = valmax

#*****************************************************************************

    def mutacion(self):
        for i in range(tam_cromosoma):
            if random.random() < humbral:
                self.cromosoma[i] = random.uniform(-10, 10)

    def __str__(self):
        cromosoma_str = [round(x, 2) for x in self.cromosoma]
        return "< Crom: " + str(cromosoma_str) + " Apt: " +
str(round(self.aptitud,2)) + " Gen: " + str(self.generacion) + " >"


# Con Valores de Representaciones de Orden
def generar_cromosoma():
    cromosoma = [random.uniform(-10, 10) for _ in range(tam_cromosoma)]
    random.shuffle(cromosoma)
    return cromosoma
```

```python
def seleccion_ruleta(poblacion):
    flecha_ruleta = random.randint(0, len(poblacion) - 1)
    seleccion = poblacion.pop(flecha_ruleta)
    return seleccion


def uniform_order_based_crossover(cromosomaA, cromosomaB):
    # Inicializacion Plantilla binaria aleatoria
    template_binario = [0 for _ in range(tam_cromosoma)]

    # Rellenamos Plantilla binaria aleatoria
    for i in range(tam_cromosoma):
        if random.random() < 0.5:
            template_binario[i] = 1

    # Inicializacion de cromosomas Hijos
    descendienteA = [0 for _ in range(tam_cromosoma)]
    descendienteB = [0 for _ in range(tam_cromosoma)]

    # Rellenamos Hijos con Plantilla binaria aleatoria
    for i in range(tam_cromosoma):
        if template_binario[i] == 1:
            descendienteA[i] = cromosomaA[i]
            descendienteB[i] = cromosomaB[i]

    buffer_hijo_A = []
    buffer_hijo_B = []

    # creando Buffer de genes restantes del Hijo A
    for i in range(tam_cromosoma):
        if not (cromosomaB[i] in descendienteA):
            buffer_hijo_A.append(cromosomaB[i])

    # creando Buffer de genes restantes del Hijo B
    for i in range(tam_cromosoma):
        if not (cromosomaA[i] in descendienteB):
            buffer_hijo_B.append(cromosomaA[i])

    # Rellenar los espacios restantes del Hijo A con los valores en orden
del Padre B
    for i in range(tam_cromosoma):
        if template_binario[i] == 0:
            descendienteA[i] = buffer_hijo_A.pop(0)
            descendienteB[i] = buffer_hijo_B.pop(0)
```

```
    return descendienteA, descendienteB

def euclidean_distance(gen1, gen2):
    distancia = 0
    for i in range(tam_cromosoma):
        distancia += (gen1.cromosoma[i] - gen2.cromosoma[i])**2
    return math.sqrt(distancia)

def crowding_replacement(poblacion, descendiente1, descendiente2, padre1,
padre2):

    dis_11 = euclidean_distance(padre1, descendiente1)
    dis_12 = euclidean_distance(padre1, descendiente2)

    dis_21 = euclidean_distance(padre2, descendiente1)
    dis_22 = euclidean_distance(padre2, descendiente2)

    if (dis_11 + dis_22) <= (dis_12 + dis_21):
        gan1 = min(padre1, descendiente1, key=lambda x: x.aptitud)
        poblacion.append(gan1)
        gan2 = min(padre2, descendiente2, key=lambda x: x.aptitud)
        poblacion.append(gan2)
    else:
        gan1 = min(padre1, descendiente2, key=lambda x: x.aptitud)
        poblacion.append(gan1)
        gan2 = min(padre2, descendiente1, key=lambda x: x.aptitud)
        poblacion.append(gan2)

#*****************************************************************************

def GA():
    poblacion = []
    generacion_global = 1
    evaluaciones = 0

    # Generacion Inicial aleatoria
    for _ in range(tam_poblacion):
        poblacion.append(Gen(generacion_global, generar_cromosoma()))
        evaluaciones += 1

    # Iteracion para 500 generaciones
    while evaluaciones < 500:
        # Generando nueva poblacion
        generacion_global += 1
        gen1 = seleccion_ruleta(poblacion)
```

```
        gen2 = seleccion_ruleta(poblacion)
        # Probabilidad de Cruzamiento.
        if random.random() > prob_cruzamiento:
            # Cruzamiento de los padres
            crom_des1, crom_des2 = uniform_order_based_crossover(
                gen1.cromosoma, gen2.cromosoma)
            descendiente1 = Gen(generacion_global, crom_des1)
            descendiente2 = Gen(generacion_global, crom_des2)
            # Mutacion del descendiente
            descendiente1.mutacion()
            descendiente2.mutacion()

            descendiente1.fitness1()
            descendiente2.fitness1()
            evaluaciones += 2
            # Remplazo Elitita
            crowding_replacement(poblacion, descendiente1,
                                 descendiente2, gen1, gen2)
        else:
            poblacion.append(gen1)
            poblacion.append(gen2)

    return min(poblacion, key=lambda x: x.aptitud)

aptitudes = []
tiempo = []

for i in range(20):
    start_time = time()
    aptitudes.append(GA().aptitud)
    tiempo.append(time() - start_time)

mejorA = min(aptitudes)
peorA = max(aptitudes)
meanA = statistics.mean(aptitudes)
medianA = statistics.median(aptitudes)
sigmaA = statistics.pstdev(aptitudes)

mejorT = min(tiempo)
peorT = max(tiempo)
meanT = statistics.mean(tiempo)
medianT = statistics.median(tiempo)
sigmaT = statistics.pstdev(tiempo)
```

```
print("aptitud: ")
print(str(round(mejorA,2)) + " " + str(round(peorA,2)) + " " +
str(round(meanA,2)) + " " + str(round(medianA,2)) + " " +
str(round(sigmaA,2)))
print("Tiempo: ")
print(str(round(mejorT,2)) + " " + str(round(peorT,2)) + " " +
str(round(meanT,2)) + " " + str(round(medianT,2)) + " " +
str(round(sigmaT,2)))
```

- **Modelo Generacional**

- **Versión 4:**

```
import random
import math
from time import time
import statistics

tam_cromosoma = 10
tam_poblacion = 10
humbral = 0.1
prob_cruzamiento = 0.5

class Gen:
    cromosoma = []
    aptitud = float
    generacion = int

    def __init__(self, generacion, cromosoma):
        self.cromosoma = cromosoma
        self.calcular_aptitud6()
        self.generacion = generacion

#*****************************************************************************

#Alpine 1 Function
    def calcular_aptitud1(self):
        valmax = 0.0
        for i in self.cromosoma:
            valmax += abs(i*(math.sin(i))+0.1*(i))
        self.aptitud = valmax

#Dixon & Price Function
    def calcular_aptitud2(self):
        valmax = (self.cromosoma[0] - 1)**2
```

```
        for i in range(1, len(self.cromosoma)):
            valmax += i*(2*math.sin(self.cromosoma[i])-self.cromosoma[i-
1])**2
        self.aptitud = valmax

#Quintic Function
    def calcular_aptitud3(self):
        valmax = 0.0
        for i in self.cromosoma:
            valmax += abs((i**5)-(3*(i**4))+(4*(i**3))-(2*(i**2))-(10*i)-4)
        self.aptitud = valmax

#Schwefel 2.23 Function
    def calcular_aptitud4(self):
        valmax = 0.0
        for i in self.cromosoma:
            valmax += i**10
        self.aptitud = valmax

#Streched V Sine Wave Function
    def calcular_aptitud5(self):
        valmax = 0.0
        for i in range(len(self.cromosoma)-1):
            valmax += ((self.cromosoma[i+1]**2 +
self.cromosoma[i]**2)**0.25)*((math.sin(50*((self.cromosoma[i+1]**2 +
self.cromosoma[i]**2)**0.1))**2)+0.1)
        self.aptitud = valmax

#Sum Squares Function
    def calcular_aptitud6(self):
        valmax = 0.0
        for j, i in enumerate(self.cromosoma):
            valmax += j * (i**2)
        self.aptitud = valmax

#*****************************************************************************

    def mutacion(self):
        for i in range(tam_cromosoma):
            if random.random() < humbral:
                self.cromosoma[i] = random.uniform(-10, 10)

    def __str__(self):
        cromosoma_str = [round(x, 2) for x in self.cromosoma]
```

```
        return "< Crom: " + str(cromosoma_str) + " Apt: " +
str(self.aptitud) + " Gen: " + str(self.generacion) + " >"


# Con Valores de Representaciones de Orden
def generar_cromosoma():
    cromosoma = [random.uniform(-10, 10) for _ in range(tam_cromosoma)]
    random.shuffle(cromosoma)
    return cromosoma

def seleccion_ruleta(poblacion):

    flecha_ruleta = random.randint(0, len(poblacion) - 1)
    seleccion = poblacion.pop(flecha_ruleta)
    return seleccion

def one_point_crossover(gen1, gen2):
    descendienteA = []
    descendienteB = []

    punto_corte = random.randint(1, tam_cromosoma-1)

    descendienteA = gen1.cromosoma[:punto_corte] +
gen2.cromosoma[punto_corte:]
    descendienteB = gen2.cromosoma[:punto_corte] +
gen1.cromosoma[punto_corte:]

    return descendienteA, descendienteB

def replacement(poblacion, nueva_poblacion):
    poblacion.clear()
    poblacion.extend(nueva_poblacion)
    nueva_poblacion.clear()

#*********************************************************************

def GA():
    poblacion = []
    nueva_poblacion = []
    generacion_global = 1
    evaluaciones = 0

    # Generacion Inicial aleatoria
    for _ in range(tam_poblacion):
        poblacion.append(Gen(generacion_global, generar_cromosoma()))
```

```
        evaluaciones += 1

    # Iteracion para 500 generaciones
    while generacion_global < 500:
        # Generando nueva poblacion
        generacion_global += 1
        while len(nueva_poblacion) != tam_poblacion:

            if random.random() > prob_cruzamiento:
                continue

            gen1 = seleccion_ruleta(poblacion)
            gen2 = seleccion_ruleta(poblacion)
            # Cruzamiento de los padres
            crom_des1, crom_des2 = one_point_crossover(gen1, gen2)
            descendiente1 = Gen(generacion_global, crom_des1)
            descendiente2 = Gen(generacion_global, crom_des2)
            # Mutacion del descendiente
            descendiente1.mutacion()
            descendiente2.mutacion()

            descendiente1.calcular_aptitud6()
            descendiente2.calcular_aptitud6()
            evaluaciones += 2

            # Insertando el descendiente en la nueva poblacion
            nueva_poblacion.append(descendiente1)
            nueva_poblacion.append(descendiente2)

        replacement(poblacion, nueva_poblacion)

    return min(poblacion, key=lambda x: x.aptitud)

aptitudes = []
tiempo = []

for i in range(20):
    start_time = time()
    aptitudes.append(GA().aptitud)
    tiempo.append(time() - start_time)

mejorA = min(aptitudes)
peorA = max(aptitudes)
meanA = statistics.mean(aptitudes)
medianA = statistics.median(aptitudes)
```

```
sigmaA = statistics.pstdev(aptitudes)
mejorT = min(tiempo)
peorT = max(tiempo)
meanT = statistics.mean(tiempo)
medianT = statistics.median(tiempo)
sigmaT = statistics.pstdev(tiempo)

print("aptitud: ")
print(str(round(mejorA,2)) + " " + str(round(peorA,2)) + " " +
str(round(meanA,2)) + " " + str(round(medianA,2)) + " " +
str(round(sigmaA,2)))
print("Tiempo: ")
print(str(round(mejorT,2)) + " " + str(round(peorT,2)) + " " +
str(round(meanT,2)) + " " + str(round(medianT,2)) + " " +
str(round(sigmaT,2)))
```

- **Versión 5:**

```
import random
import math
from time import time
import statistics

tam_cromosoma = 10
tam_poblacion = 10
k_torneo = 2  # tam de la poblacion del torneo
humbral = 0.1
prob_cruzamiento = 0.5

class Gen:
    cromosoma = []
    aptitud = float
    generacion = int

    def __init__(self, generacion, cromosoma):
        self.cromosoma = cromosoma
        self.calcular_aptitud6()
        self.generacion = generacion

#*****************************************************************************

#Alpine 1 Function
    def calcular_aptitud1(self):
```

```
        valmax = 0.0
        for i in self.cromosoma:
            valmax += abs(i*(math.sin(i))+0.1*(i))
        self.aptitud = valmax

#Dixon & Price Function
    def calcular_aptitud2(self):
        valmax = (self.cromosoma[0] - 1)**2
        for i in range(1, len(self.cromosoma)):
            valmax += i*(2*math.sin(self.cromosoma[i])-self.cromosoma[i-
1])**2
        self.aptitud = valmax

#Quintic Function
    def calcular_aptitud3(self):
        valmax = 0.0
        for i in self.cromosoma:
            valmax += abs((i**5)-(3*(i**4))+(4*(i**3))-(2*(i**2))-(10*i)-4)
        self.aptitud = valmax

#Schwefel 2.23 Function
    def calcular_aptitud4(self):
        valmax = 0.0
        for i in self.cromosoma:
            valmax += i**10
        self.aptitud = valmax

#Streched V Sine Wave Function
    def calcular_aptitud5(self):
        valmax = 0.0
        for i in range(len(self.cromosoma)-1):
            valmax += ((self.cromosoma[i+1]**2 +
self.cromosoma[i]**2)**0.25)*((math.sin(50*((self.cromosoma[i+1]**2 +
self.cromosoma[i]**2)**0.1))**2)+0.1)
        self.aptitud = valmax

#Sum Squares Function
    def calcular_aptitud6(self):
        valmax = 0.0
        for j, i in enumerate(self.cromosoma):
            valmax += j * (i**2)
        self.aptitud = valmax

#*****************************************************************************
```

```
    def mutacion(self):
        for i in range(tam_cromosoma):
            if random.random() < humbral:
                self.cromosoma[i] = random.uniform(-10, 10)

    def __str__(self):
        cromosoma_str = [round(x, 2) for x in self.cromosoma]
        return "< Crom: " + str(cromosoma_str) + " Apt: " +
str(self.aptitud) + " Gen: " + str(self.generacion) + " >"


# Con Valores de Representaciones de Orden
def generar_cromosoma():
    cromosoma = [random.uniform(-10, 10) for _ in range(tam_cromosoma)]
    random.shuffle(cromosoma)
    return cromosoma

def seleccion_torneo(poblacion):
    torneo = []
    for i in range(k_torneo):
        seleccion = random.randint(0, tam_poblacion - 1)
        gen = poblacion[seleccion]
        torneo.append(gen)
    return min(torneo, key=lambda x: x.aptitud)

def uniform_order_based_crossover(cromosomaA, cromosomaB):
    # Inicializacion Plantilla binaria aleatoria
    template_binario = [0 for _ in range(tam_cromosoma)]

    # Rellenamos Plantilla binaria aleatoria
    for i in range(tam_cromosoma):
        if random.random() < 0.5:
            template_binario[i] = 1

    # Inicializacion de cromosomas Hijos
    descendienteA = [0 for _ in range(tam_cromosoma)]
    descendienteB = [0 for _ in range(tam_cromosoma)]

    # Rellenamos Hijos con Plantilla binaria aleatoria
    for i in range(tam_cromosoma):
        if template_binario[i] == 1:
            descendienteA[i] = cromosomaA[i]
            descendienteB[i] = cromosomaB[i]

    buffer_hijo_A = []
```

```
    buffer_hijo_B = []

    # creando Buffer de genes restantes del Hijo A
    for i in range(tam_cromosoma):
        if not (cromosomaB[i] in descendienteA):
            buffer_hijo_A.append(cromosomaB[i])

    # creando Buffer de genes restantes del Hijo B
    for i in range(tam_cromosoma):
        if not (cromosomaA[i] in descendienteB):
            buffer_hijo_B.append(cromosomaA[i])

    # Rellenar los espacios restantes del Hijo A con los valores en orden
del Padre B
    for i in range(tam_cromosoma):
        if template_binario[i] == 0:
            descendienteA[i] = buffer_hijo_A.pop(0)
            descendienteB[i] = buffer_hijo_B.pop(0)

    return descendienteA, descendienteB

def replacement(poblacion, nueva_poblacion):
    poblacion.clear()
    poblacion.extend(nueva_poblacion)
    nueva_poblacion.clear()

#*************************************************************************

def GA():
    poblacion = []
    nueva_poblacion = []
    generacion_global = 1
    evaluaciones = 0

    # Generacion Inicial aleatoria
    for _ in range(tam_poblacion):
        poblacion.append(Gen(generacion_global, generar_cromosoma()))
        evaluaciones += 1

    # Iteracion para 500 generaciones
    while generacion_global < 500:
        # Generando nueva poblacion
        generacion_global += 1
        while len(nueva_poblacion) != tam_poblacion:
```

```
            if random.random() > prob_cruzamiento:
                continue

            gen1 = seleccion_torneo(poblacion)
            gen2 = seleccion_torneo(poblacion)
            # Cruzamiento de los padres
            crom_des1, crom_des2 = uniform_order_based_crossover(
                gen1.cromosoma, gen2.cromosoma)
            descendiente1 = Gen(generacion_global, crom_des1)
            descendiente2 = Gen(generacion_global, crom_des2)
            # Mutacion del descendiente
            descendiente1.mutacion()
            descendiente2.mutacion()

            descendiente1.calcular_aptitud6()
            descendiente2.calcular_aptitud6()
            evaluaciones += 2

            # Insertando el descendiente en la nueva poblacion
            nueva_poblacion.append(descendiente1)
            nueva_poblacion.append(descendiente2)

        replacement(poblacion, nueva_poblacion)

    return min(poblacion, key=lambda x: x.aptitud)

aptitudes = []
tiempo = []

for i in range(20):
    start_time = time()
    aptitudes.append(GA().aptitud)
    tiempo.append(time() - start_time)

mejorA = min(aptitudes)
peorA = max(aptitudes)
meanA = statistics.mean(aptitudes)
medianA = statistics.median(aptitudes)
sigmaA = statistics.pstdev(aptitudes)

mejorT = min(tiempo)
peorT = max(tiempo)
meanT = statistics.mean(tiempo)
medianT = statistics.median(tiempo)
sigmaT = statistics.pstdev(tiempo)
```

```
print("aptitud: ")
print(str(round(mejorA,2)) + " " + str(round(peorA,2)) + " " +
str(round(meanA,2)) + " " + str(round(medianA,2)) + " " +
str(round(sigmaA,2)))
print("Tiempo: ")
print(str(round(mejorT,2)) + " " + str(round(peorT,2)) + " " +
str(round(meanT,2)) + " " + str(round(medianT,2)) + " " +
str(round(sigmaT,2)))
```

- **Versión 6:**

```
import random
import math
from time import time
import statistics

tam_cromosoma = 10
tam_poblacion = 10
humbral = 0.1
prob_cruzamiento = 0.5

class Gen:
    cromosoma = []
    aptitud = float
    generacion = int

    def __init__(self, generacion, cromosoma):
        self.cromosoma = cromosoma
        self.calcular_aptitud6()
        self.generacion = generacion

#*********************************************************************

#Alpine 1 Function
    def calcular_aptitud1(self):
        valmax = 0
        for i in self.cromosoma:
            valmax += abs(i*(math.sin(i))+0.1*(i))
        self.aptitud = valmax

#Dixon & Price Function
    def calcular_aptitud2(self):
        valmax = (self.cromosoma[0] - 1)**2
        for i in range(1, tam_cromosoma):
```

```
            valmax += i*(2*math.sin(self.cromosoma[i])-self.cromosoma[i-
1])**2
        self.aptitud = valmax

#Quintic Function
    def calcular_aptitud3(self):
        valmax = 0
        for i in self.cromosoma:
            valmax += abs((i**5)-(3*(i**4))+(4*(i**3))-(2*(i**2))-(10*i)-4)
        self.aptitud = valmax

#Schwefel 2.23 Function
    def calcular_aptitud4(self):
        valmax = 0
        for i in self.cromosoma:
            valmax += i**10
        self.aptitud = valmax

#Streched V Sine Wave Function
    def calcular_aptitud5(self):
        valmax = 0
        for i in range(tam_cromosoma-1):
            valmax += ((self.cromosoma[i+1]**2 +
self.cromosoma[i]**2)**0.25)*((math.sin(50*((self.cromosoma[i+1]**2 +
self.cromosoma[i]**2)**0.1))**2)+0.1)
        self.aptitud = valmax

#Sum Squares Function
    def calcular_aptitud6(self):
        valmax = 0
        for j, xi in enumerate(self.cromosoma):
            valmax += j * (xi**2)
        self.aptitud = valmax

#*****************************************************************************

    def mutacion(self):
        for i in range(tam_cromosoma):
            if random.random() < humbral:
                self.cromosoma[i] = random.uniform(-10, 10)

    def __str__(self):
        cromosoma_str = [round(x, 2) for x in self.cromosoma]
        return "< Crom: " + str(cromosoma_str) + " Apt: " +
str(self.aptitud) + " Gen: " + str(self.generacion) + " >"
```

```python
# Con Valores de Representaciones de Orden
def generar_cromosoma():
    cromosoma = [random.uniform(-10, 10) for _ in range(tam_cromosoma)]
    random.shuffle(cromosoma)
    return cromosoma


def seleccion_proporcional(poblacion):

    probabilidad = []
    probTotal = 0
    flecha = random.random()
    ruleta = [0]

    for i in poblacion:
        probTotal += i.aptitud

    for i in poblacion:
        probabilidad.append(i.aptitud / probTotal)

    for i in range(tam_poblacion):
        ruleta.append(probabilidad[i]+ruleta[i])

    ruleta.append(flecha)
    ruleta.sort()
    posicion = ruleta.index(flecha)
    apareamiento = poblacion[posicion-1]

    return apareamiento


def uniform_crossover(gen1, gen2):
    descendienteA = []
    descendienteB = []

    for j in range(tam_cromosoma):
        if random.random() < 0.5:
            descendienteA.append(gen1.cromosoma[j])
            descendienteB.append(gen2.cromosoma[j])
        else:
            descendienteA.append(gen2.cromosoma[j])
            descendienteB.append(gen1.cromosoma[j])
    return descendienteA, descendienteB


def replacement(poblacion, nueva_poblacion):
```

```
    poblacion.clear()
    poblacion.extend(nueva_poblacion)
    nueva_poblacion.clear()

#**************************************************************************

def GA():
    poblacion = []
    nueva_poblacion = []
    generacion_global = 1
    evaluaciones = 0

    # Generacion Inicial aleatoria
    for _ in range(tam_poblacion):
        poblacion.append(Gen(generacion_global, generar_cromosoma()))
        evaluaciones += 1

    # Iteracion para 500 generaciones
    while generacion_global < 500:
        # Generando nueva poblacion
        generacion_global += 1
        while len(nueva_poblacion) != tam_poblacion:

            if random.random() > prob_cruzamiento:
                continue

            gen1 = seleccion_proporcional(poblacion)
            gen2 = seleccion_proporcional(poblacion)
            # Cruzamiento de los padres
            crom_des1, crom_des2 = uniform_crossover(gen1, gen2)
            descendiente1 = Gen(generacion_global, crom_des1)
            descendiente2 = Gen(generacion_global, crom_des2)
            # Mutacion del descendiente
            descendiente1.mutacion()
            descendiente2.mutacion()

            descendiente1.calcular_aptitud6()
            descendiente2.calcular_aptitud6()
            evaluaciones += 2

            # Insertando el descendiente en la nueva poblacion
            nueva_poblacion.append(descendiente1)
            nueva_poblacion.append(descendiente2)

        replacement(poblacion, nueva_poblacion)
```

```
    return min(poblacion, key=lambda x: x.aptitud)

aptitudes = []
tiempo = []

for i in range(20):
    start_time = time()
    aptitudes.append(GA().aptitud)
    tiempo.append(time() - start_time)

mejorA = min(aptitudes)
peorA = max(aptitudes)
meanA = statistics.mean(aptitudes)
medianA = statistics.median(aptitudes)
sigmaA = statistics.pstdev(aptitudes)

mejorT = min(tiempo)
peorT = max(tiempo)
meanT = statistics.mean(tiempo)
medianT = statistics.median(tiempo)
sigmaT = statistics.pstdev(tiempo)

print("aptitud: ")
print(str(round(mejorA,2)) + " " + str(round(peorA,2)) + " " +
str(round(meanA,2)) + " " + str(round(medianA,2)) + " " +
str(round(sigmaA,2)))
print("Tiempo: ")
print(str(round(mejorT,2)) + " " + str(round(peorT,2)) + " " +
str(round(meanT,2)) + " " + str(round(medianT,2)) + " " +
str(round(sigmaT,2)))
```

4. Reporte los resultados obtenidos. Para ello, realice 20 ejecuciones independientes, con la siguiente configuración:

a. Considere un total de 500 evaluaciones.

b. Muestre el mejor, peor, promedio, mediana y desviación estándar de los resultados en las 20 ejecuciones.

- **D = 10**

- **Tabla 1:**

Tabla de Aptitudes para D = 10 Estacionario Ver1					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	0.43	2.90	1.64	1.85	0.63
F2	7.40	63.07	32.29	31.15	16.17
F3	18.63	310.77	73.26	56.39	59.87
F4	0.28	45673.14	3615.43	894.46	9829.39
F5	1.43	3.42	2.52	2.55	0.46
F6	3.33	47.20	21.21	19.76	11.10
Promedio	**5.25**	**7683.42**	**624.39**	**167.69**	**1652.94**

- **Tabla 2:**

Tabla de Aptitudes para D = 10 Estacionario Ver2					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	12.96	37.25	29.83	30.88	5.55
F2	336.20	2172.93	1434.83	1445.81	416.28
F3	6950.95	335445.82	184503.56	184584.84	80823.18
F4	16255405.03	23113340520.40	13860747032.31	12862638184.57	5608022543.81
F5	7.92	15.73	11.65	11.95	2.26
F6	732.95	2317.61	1519.01	1484.06	300.75
Promedio	**2710574.34**	**3852280084.96**	**2310155755.20**	**2143804290.35**	**934684015.31**

- **Tabla 3:**

Tabla de Aptitudes para D = 10 Estacionario Ver3					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	0.71	3.85	2.50	2.60	0.77
F2	23.41	212.41	100.86	92.07	50.74
F3	35.07	1048.00	230.87	127.39	235.87
F4	4.35	112586.43	22721.84	3560.82	31368.97
F5	2.04	4.51	3.54	3.52	0.59
F6	28.54	181.87	87.29	87.50	38.39
Promedio	**15.69**	**19006.18**	**3857.82**	**645.65**	**5282.56**

- **Tabla 4:**

Tabla de Aptitudes para D = 10 Generacional Ver4					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	7.13	21.53	15.90	16.01	2.50
F2	202.56	931.17	601.85	629.67	141.96
F3	8638.27	61455.29	34860.21	27715.75	15149.63
F4	3796996.49	1342465658.05	543100808.03	365770549.47	369808348.14
F5	4.50	10.22	8.15	8.27	1.07
F6	112.00	825.90	585.13	607.13	141.55
Promedio	**634326.83**	**223754817.03**	**90522813.21**	**60966587.72**	**61637297.48**

- **Tabla 5:**

Tabla de Aptitudes para D = 10 Generacional Ver5					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	4.30	11.56	7.17	6.86	1.84
F2	94.95	714.90	359.90	336.77	151.83
F3	318.27	10167.04	3472.08	2934.95	2570.08
F4	40170.55	100566739.69	24635913.84	8492846.42	27769959.67
F5	2.57	8.49	6.42	6.43	1.13
F6	77.73	604.46	365.43	329.77	124.15
Promedio	**6778.06**	**16763041.02**	**4106687.47**	**1416076.87**	**4628801.45**

- **Tabla 6:**

Tabla de Aptitudes para D = 10 Generacional Ver6					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	18.01	36.18	25.45	26.21	4.79
F2	505.03	2108.32	1468.53	1466.65	399.68
F3	40285.45	340657.92	197692.88	211074.48	82121.11
F4	1211401176.65	24200596451.69	15978362337.29	17083445543.24	5429875022.75
F5	4.22	13.12	9.90	10.34	1.94
F6	464.42	1976.48	1309.90	1328.08	354.82
Promedio	**201907075.63**	**4033490207.29**	**2663093807.33**	**2847276574.83**	**904992984.18**

- **D = 30**

- **Tabla 1:**

Tabla de Aptitudes para D = 30 Estacionario Ver1					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	13.26	23.82	18.99	19.93	2.59
F2	1074.96	2405.01	1711.47	1743.63	333.00
F3	2852.15	16623.71	9061.32	8590.15	3540.83
F4	2135296.64	107590412.98	32794730.70	21521019.43	27957943.25
F5	11.18	18.04	15.66	16.09	1.87
F6	835.53	2260.43	1474.84	1495.36	318.48
Promedio	**356680.62**	**17935290.67**	**5467835.50**	**3588814.10**	**4660356.67**

- **Tabla 2:**

Tabla de Aptitudes para D = 30 Estacionario Ver2					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	58.01	112.36	85.16	84.49	12.05
F2	9314.82	19340.33	14592.79	14611.29	2061.45
F3	290696.22	846251.12	509847.89	497033.11	135600.68
F4	9862702494.43	55958784217.00	29475578035.07	26064243890.80	11190359609.35
F5	28.20	51.40	40.85	41.74	5.14
F6	9529.19	18459.75	13861.25	14213.76	2098.49
Promedio	**1643835353.48**	**9326611405.33**	**4912686077.17**	**4344128312.53**	**1865083231.19**

- **Tabla 3:**

Tabla de Aptitudes para D = 30 Estacionario Ver3					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	17.17	36.35	26.73	26.76	4.79
F2	2110.04	6402.97	4806.99	4732.26	1134.62
F3	5009.92	33641.83	19116.75	18995.06	7747.15
F4	8907752.98	219879138.5	94437816.78	82955625.68	58335137.59
F5	17.84	28.42	23.35	22.84	2.54
F6	2344.61	5938.85	3690.71	3629.70	815.41
Promedio	**1486208.76**	**36654197.83**	**15744246.89**	**13830505.38**	**9724140.35**

- **Tabla 4:**

Tabla de Aptitudes para D = 30 Generacional Ver4					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	53.16	81.93	66.96	66.82	8.17
F2	7608.51	12312.13	9868.27	9895.11	1183.74
F3	114354.18	459618.62	279650.34	272254.10	81388.19
F4	3122595960.76	13709818884.77	9080559577.98	8150344734.52	2873098696.19
F5	30.67	40.14	35.82	35.94	2.57
F6	6648.79	11266.96	9428.50	9464.79	1352.27
Promedio	**520454109.35**	**2285050367.43**	**1513476437.98**	**1358439408.55**	**478863771.86**

- **Tabla 5:**

Tabla de Aptitudes para D = 30 Generacional Ver5					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	37.83	56.33	47.26	46.53	5.66
F2	6218.92	10553.40	8560.96	8576.17	987.50
F3	54812.94	199923.69	107646.52	91521.61	37847.60
F4	435770656.24	7005715463.72	2962282526.19	2505174562.42	1735134168.86
F5	26.90	41.22	32.24	32.06	3.38
F6	5934.67	11687.60	8625.22	8840.73	1303.93
Promedio	**72639614.58**	**1167656287.66**	**493734573.07**	**417547263.25**	**289195719.49**

- **Tabla 6:**

Tabla de Aptitudes para D = 30 Generacional Ver6					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	61.86	93.26	81.03	82.66	7.77
F2	6298.86	10994.43	8506.18	8534.64	1074.65
F3	4825.38	9808.96	7933.67	8007.25	1308.05
F4	13467933709.69	39482266656.45	29231893352.23	30263322062.93	6353092104.80
F5	29.38	48.07	38.16	38.40	4.12
F6	9793.46	14617.41	12568.62	12752.45	1589.35
Promedio	**2244659119.77**	**6580383703.10**	**4871987079.98**	**5043891913.06**	**1058849348.12**

c. Muestre el mejor, peor, promedio, mediana y desviación estándar de los tiempos de ejecución (en segundos) en las 20 ejecuciones.

- **D = 10**

- **Tabla 1:**

Tabla de Tiempos (segundos) para D = 10 Estacionario Ver1					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	0.01	0.02	0.01	0.01	0.00
F2	0.02	0.02	0.02	0.02	0.00
F3	0.02	0.02	0.02	0.02	0.00
F4	0.01	0.02	0.01	0.01	0.00
F5	0.02	0.03	0.03	0.02	0.00
F6	0.01	0.03	0.02	0.01	0.00
Promedio	**0.02**	**0.02**	**0.02**	**0.02**	**0.00**

- **Tabla 2:**

Tabla de Tiempos (segundos) para D = 10 Estacionario Ver2					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	0.01	0.01	0.01	0.01	0.00
F2	0.02	0.02	0.02	0.02	0.00
F3	0.02	0.02	0.02	0.02	0.00
F4	0.01	0.02	0.01	0.01	0.00
F5	0.02	0.04	0.03	0.03	0.00
F6	0.01	0.02	0.01	0.01	0.00
Promedio	**0.02**	**0.02**	**0.02**	**0.02**	**0.00**

- **Tabla 3:**

Tabla de Tiempos (segundos) para D = 10 Estacionario Ver3					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	0.01	0.02	0.01	0.01	0.00
F2	0.01	0.02	0.02	0.02	0.00
F3	0.02	0.03	0.02	0.02	0.00
F4	0.01	0.02	0.01	0.01	0.00
F5	0.02	0.05	0.03	0.03	0.01
F6	0.01	0.03	0.02	0.01	0.01
Promedio	**0.01**	**0.03**	**0.02**	**0.02**	**0.00**

- **Tabla 4:**

Tabla de Tiempos (segundos) para D = 10 Generacional Ver4					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	0.11	0.17	0.13	0.12	0.02
F2	0.16	0.37	0.20	0.18	0.05
F3	0.26	0.41	0.31	0.28	0.03
F4	0.09	0.17	0.12	0.11	0.02
F5	0.37	0.59	0.45	0.42	0.05
F6	0.11	0.15	0.13	0.12	0.02
Promedio	**0.18**	**0.31**	**0.22**	**0.21**	**0.03**

- **Tabla 5:**

Tabla de Tiempos (segundos) para D = 10 Generacional Ver5					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	0.18	0.44	0.23	0.21	0.06
F2	0.24	0.44	0.29	0.28	0.04
F3	0.34	0.65	0.42	0.39	0.07
F4	0.17	0.28	0.21	0.19	0.03
F5	0.45	0.70	0.53	0.51	0.06
F6	0.18	0.27	0.22	0.21	0.03
Promedio	**0.26**	**0.46**	**0.32**	**0.30**	**0.05**

- **Tabla 6:**

Tabla de Tiempos (segundos) para D = 10 Generacional Ver6					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	0.18	0.47	0.24	0.23	0.06
F2	0.25	0.44	0.31	0.30	0.05
F3	0.35	0.60	0.41	0.41	0.05
F4	0.16	0.32	0.20	0.18	0.05
F5	0.44	0.72	0.53	0.51	0.07
F6	0.17	0.46	0.23	0.21	0.06
Promedio	**0.26**	**0.50**	**0.32**	**0.31**	**0.06**

- **D = 30**

- **Tabla 1:**

Tabla de Tiempos (segundos) para D = 30 Estacionario Ver1					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	0.02	0.03	0.02	0.02	0.00
F2	0.03	0.04	0.03	0.03	0.00
F3	0.04	0.11	0.05	0.05	0.02
F4	0.01	0.04	0.02	0.02	0.01
F5	0.06	0.10	0.07	0.07	0.01
F6	0.02	0.03	0.02	0.02	0.00
Promedio	**0.03**	**0.06**	**0.04**	**0.04**	**0.01**

- **Tabla 2:**

Tabla de Tiempos (segundos) para D = 30 Estacionario Ver2					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	0.01	0.03	0.02	0.02	0.00
F2	0.02	0.06	0.03	0.03	0.01
F3	0.04	0.08	0.05	0.05	0.01
F4	0.01	0.02	0.02	0.02	0.00
F5	0.06	0.10	0.08	0.08	0.01
F6	0.01	0.03	0.02	0.02	0.01
Promedio	**0.03**	**0.05**	**0.04**	**0.04**	**0.01**

- **Tabla 3:**

Tabla de Tiempos (segundos) para D = 30 Estacionario Ver3					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	0.03	0.06	0.04	0.04	0.01
F2	0.04	0.10	0.06	0.05	0.02
F3	0.06	0.12	0.07	0.07	0.02
F4	0.03	0.06	0.04	0.04	0.01
F5	0.08	0.13	0.09	0.09	0.01
F6	0.04	0.07	0.05	0.05	0.01
Promedio	**0.05**	**0.09**	**0.06**	**0.06**	**0.01**

- **Tabla 4:**

Tabla de Tiempos (segundos) para D = 30 Generacional Ver4					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	0.03	0.07	0.04	0.04	0.01
F2	0.05	0.08	0.06	0.06	0.01
F3	0.02	0.04	0.03	0.03	0.00
F4	0.02	0.05	0.03	0.03	0.01
F5	0.03	0.06	0.04	0.04	0.01
F6	0.03	0.06	0.04	0.05	0.01
Promedio	**0.03**	**0.06**	**0.04**	**0.04**	**0.01**

- **Tabla 5:**

Tabla de Tiempos (segundos) para D = 30 Generacional Ver5					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	0.15	0.16	0.16	0.16	0.00
F2	0.24	0.25	0.25	0.25	0.00
F3	0.20	0.23	0.22	0.22	0.01
F4	0.24	0.25	0.25	0.25	0.00
F5	0.57	0.60	0.59	0.59	0.01
F6	0.26	0.28	0.27	0.27	0.00
Promedio	**0.28**	**0.30**	**0.29**	**0.29**	**0.00**

- **Tabla 6:**

Tabla de Tiempos (segundos) para D = 30 Generacional Ver6					
Función	Mejor	Peor	Promedio	Mediana	Desviación Estándar
F1	0.20	0.22	0.22	0.22	0.00
F2	0.35	0.37	0.36	0.36	0.00
F3	0.26	0.28	0.27	0.27	0.00
F4	0.15	0.16	0.16	0.16	0.00
F5	0.45	0.48	0.47	0.47	0.01
F6	0.18	0.19	0.19	0.19	0.00
Promedio	**0.27**	**0.28**	**0.28**	**0.28**	**0.00**

3. Conclusiones

Los algoritmos genéticos presentan importantes ventajas sobre los algoritmos tradicionales, como son: su capacidad para trabajar con varios puntos simultáneamente, abarcando así un espacio mayor de búsqueda (paralelismo), además no requieren manipular directamente los parámetros del problema y poseen una gran facilidad para adaptarse a diferentes tipos de problemas.

A su vez, los algoritmos genéticos también presentan algunas desventajas como son: la dificultad para encontrar una representación apropiada de los parámetros del problema o para hallar una función objetivo adecuada.

Los AG algoritmos no aseguran encontrar la solución óptima, pero en muchos casos pueden proporcionar soluciones bastante adecuadas y que por otros métodos hubiera sido imposible encontrar. Dicho lo anterior, estos algoritmos representan, por todas sus características, una opción que debe ser tenida en cuenta cuando se busca resolver un problema con algún grado de complejidad.

4. Referencias

[1] Yu & Gen. Introduction to Evolutionary Algorithms. 2010. Capítulos 1 al 3.

[2] Burke & Kendall. Search Metodologies. 2005 Capítulo 4.

[3] Russell & Norving. Artificial Intelligence - A Modern Approach – 1995. Capítulo 4.