



Tecnológico nacional de México  
Instituto tecnológico de la laguna



**Inteligencia Artificial  
INGENIERÍA EN SISTEMAS COMPUTACIONALES**

**Documentación Proyecto #3. Algoritmos genéticos**

Presenta:

Omar Adrián Tapia Guzmán 20130022

Israel Castañeda Luna 20130026

Cesar Alexis Ochoa Tapia 19130952

Jorge Antonio Reyes Muñoz 19130965

# ÍNDICE

<b>Introducción</b>	<b>3</b>
¿Qué son los algoritmos genéticos?	3
¿Cómo funciona un AG?	4
Aplicaciones de un AG	5
<b>Metodología</b>	<b>6</b>
Planteamiento del problema	6
Lenguaje de programación	6
<b>Análisis del problema</b>	<b>7</b>
<b>Objetivo</b>	<b>7</b>
<b>Justificación</b>	<b>7</b>
<b>Especificación de requerimientos</b>	<b>8</b>
Algoritmo Genético.	8
Librerías y clases.	8
Generar aleatoriamente las tonalidades de verde.	8
Definir las funciones de probabilidad y de fitness.	8
Definir la función de selección de padres	9
Definir la función de mutación	9
Definir la función objetivo	9
<b>Diseño de diagramas</b>	<b>10</b>
Modelo de datos.	10
Diagrama de procesos	10
Diagrama de secuencias	11
<b>Prototipos</b>	<b>12</b>
<b>Implementación</b>	<b>14</b>
mainGraf.py	14
GeneticAlgorithm.py	17
<b>Pruebas</b>	<b>20</b>
Prueba de caja negra	20
Prueba de caja blanca	20
Prueba de rendimiento	20
Pruebas de funcionalidad	20
Prueba de Ejecución	21
<b>Funcionamiento</b>	<b>21</b>
<b>Observaciones</b>	<b>23</b>
<b>Conclusiones</b>	<b>24</b>
<b>Bibliografía</b>	<b>25</b>

## Introducción

### ¿Qué son los algoritmos genéticos?

Un algoritmo es una serie de pasos que describen el proceso de búsqueda de una solución a un problema concreto. Y un algoritmo genético es cuando se usan mecanismos que simulan los de la evolución de las especies de la biología para formular estos pasos. Es una técnica de inteligencia artificial inspirada en la idea de que el que sobrevive es el que está mejor adaptado al medio, es decir la misma que subyace a la teoría de la evolución que formuló Charles Darwin y que combina esa idea de la evolución con la genética.

Los AGs fueron delineados por un par de científicos norteamericanos, John Holland [1929–] en los años 1970 y presentados en 1989 por David Goldberg [1953–] como un método de optimización de búsqueda global, debido a que este tipo de métodos explora todo el espacio de soluciones del problema permitiendo salir de posibles óptimos locales e ir en busca de óptimos globales. Para entender lo que es la optimización hay que considerar que la programación matemática intenta resolver procesos que tienen diferentes posibles soluciones, pero sólo una de ellas corresponde al óptimo global, es decir la solución que se ajusta mejor a las condiciones del mismo problema. Cualquier otra solución que se parezca al óptimo global es un óptimo local.

Los AGs hacen evolucionar una población de individuos, o conjunto de soluciones posibles del problema, sometiéndose a acciones aleatorias semejantes a las que actúan en la evolución biológica tales como mutaciones y recombinaciones genéticas; así como también a una selección de acuerdo con algún criterio, en función del cual se decide cuáles son los individuos más adaptados, que sobreviven, y cuáles los menos aptos, que son descartados.

## ¿Cómo funciona un AG?

Un AG funciona de la siguiente manera. Dado un problema específico de optimización a resolver, el AG requiere de un conjunto inicial de soluciones potenciales a ese problema, codificadas de alguna manera y de una función de aptitud que permite evaluar cuantitativamente a cada candidata a solución. Estas candidatas se suelen generar aleatoriamente, o bien pueden ser soluciones que ya se sabe que funcionan, con el objetivo de que el AG depure las opciones válidas hasta escoger la mejor.

Cada una de las soluciones potenciales es evaluada por la función de aptitud, una ecuación matemática, que le da una calificación para saber qué tan “buena” es con respecto a las demás soluciones. Por supuesto, la mayoría de estas soluciones no funcionarán en absoluto, y serán eliminadas. Sin embargo, por puro azar, unas pocas pueden ser prometedoras, es decir, pueden mostrar parte de la solución, aunque ésta sea débil e imperfecta, hacia la solución final del problema. Por lo tanto un AG es un método de búsqueda dirigida basado en probabilidades.

Las candidatas prometedoras se conservan y se les permite reproducirse. Se realizan múltiples copias de ellas, que como no son perfectas a algunas de ellas se les introduce, con cierta probabilidad, cambios aleatorios o mutaciones durante el proceso de copia para garantizar que ninguna solución tenga una probabilidad nula de ser examinada. Luego, esta descendencia digital prosigue con la siguiente generación, formando un nuevo acervo de soluciones candidatas que son sometidas a una ronda de evaluación de aptitud. Las candidatas que han empeorado, o no han mejorado, con los cambios en su código son eliminadas de nuevo; pero, por puro azar, las variaciones aleatorias introducidas en la población pueden haber mejorado a algunos individuos, convirtiéndolos en mejores soluciones del problema, más completas o más eficientes. De nuevo, se seleccionan y copian estos individuos vencedores hacia la siguiente generación con cambios aleatorios, y el proceso se repite. Las expectativas son que la aptitud media de la población se incrementará en cada ronda y, por tanto, repitiendo este proceso cientos o miles de rondas, pueden descubrirse soluciones muy buenas del problema.

## Aplicaciones de un AG

Hay una gran cantidad de problemas que pueden ser resueltos mediante los algoritmos genéticos. Algunos de estos problemas son:

**Caso 1: Optimización de rutas.** Aplicado en la optimización de rutas, permite encontrar la ruta más corta o más rápida entre ciudades o zonas en muy poco tiempo. También resulta muy útil en Smart cities, para tratar de reducir las emisiones de CO<sub>2</sub>.

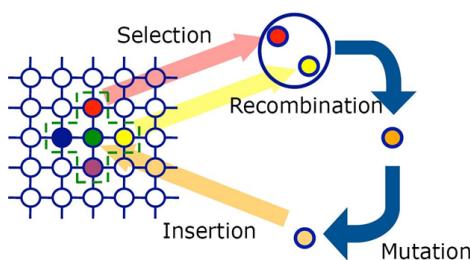
**Caso 2: Optimización de tareas.** Cuando se trata de realizar tareas en el tiempo más breve posible, optimizarlas es fundamental. Con los algoritmos genéticos, además el cálculo se hace rápido y eficaz.

**Caso 3: Gestión automatizada de equipamiento industrial.** Es posible hacer un cálculo en tiempo real para optimizar un proceso automatizado de equipamiento industrial.

**Caso 4: Aprendizaje de comportamiento de robots.** El aprendizaje de robots es posible respecto a una función de coste. Con estos algoritmos el aprendizaje se realiza de un modo más rápido y eficaz.

**Caso 5: Sistemas del sector financiero.** Una aplicación interesante en finanzas es que estos algoritmos permiten descubrir reglas de inversión que indican cuándo entrar y salir de un mercado para obtener los máximos beneficios. Por otro lado, en métodos de Splitwise, se puede optimizar el método de compartir gastos entre distintos usuarios.

**Caso 6: Encontrar errores en programas.** Permite detectar errores en los programas de los desarrolladores, lo que ayuda a ahorrar tiempo y dinero en su implementación.



## **Metodología**

### **Planteamiento del problema**

El objetivo principal de este algoritmo genético es simular y generar un camuflaje militar con diferentes tonalidades de color verde. En dicho algoritmo se generarán primeramente de forma aleatoria los tonos de verde a usar y una vez generados se tomarán, y se realizarán los cruces y mutaciones necesarias a través de las diferentes generaciones (tomando 50 como nuestro límite) para al final obtener un camuflaje mejor definido y dando un efecto parecido al de un camuflaje bordado en tela. Esto también podría ser utilizado como un generador de texturas coloridas para videojuegos o algo por el estilo.

### **Lenguaje de programación**

Para programar este algoritmo genético utilizamos el lenguaje python, ya que brinda una sintaxis sencilla y relativamente fácil de aprender, además de otras ventajas muy útiles.

Optamos por programar con python desde visual studio code, ya que es un editor de código muy completo en realidad.

## **Análisis del problema**

Se debe desarrollar un programa que implemente un algoritmo genético para generar camuflajes militares de tonalidad verde.

Para ello debemos tener una cuadrícula la cual será nuestra población inicial ( $5 \times 5$ ). Dicha cuadrícula será dividida de cierta forma, para que cada cuadro de la población(que será de  $4 \times 4$ ) sean los individuos de nuestra población. Y finalmente los elementos del arreglo tendrán 4 individuos y 4 cromosomas por individuo, en total 16 cromosomas por población.

Una vez que entendemos este panorama y tenemos detectada, la población, sus individuos y sus cromosomas podemos comenzar la codificación de las funciones necesarias, como lo son: La función objetivo, las probabilidades, la función de fitness, selección de padres, crossover y mutación.

## **Objetivo**

Aprender a hacer uso de los algoritmos genéticos e identificar las partes que lo componen, para lograr crear un sistema que genera patrones de camuflaje para soldados o algún otro sistema que lo requiera.

## **Justificación**

Un sistema que emplea algoritmos genéticos para generar patrones de camuflaje se justifica por su capacidad de adaptación y optimización en entornos variables. Esto resulta valioso en aplicaciones militares, de conservación de la fauna y caza, ya que permite a las unidades y animales camuflarse de manera más eficaz, mejorando su capacidad de supervivencia y éxito en diversas situaciones.

# Especificación de requerimientos

## **Algoritmo Genético.**

Necesitaremos hacer uso de un algoritmo genético para lograr que se genere el camuflaje, empezaremos con una población inicial de colores que irá cambiando con el paso de las generaciones, para darnos nuestro camuflaje final. Todo esto se hará mediante código y el usuario solo verá los resultados de los 2 camuflajes el inicial y el final.

## **Librerías y clases.**

NumPy es una biblioteca fundamental en Python para computación numérica y científica. Se utiliza para trabajar con arreglos multidimensionales (arrays) y ofrece una amplia gama de funciones y herramientas para realizar operaciones matemáticas y científicas de manera eficiente.

matplotlib.pyplot es un módulo de la biblioteca Matplotlib, que se utiliza en Python para crear visualizaciones y gráficos de datos de manera eficiente.

GeneticAlgorithm es la clase que creamos que tiene codificado el algoritmo genético que usaremos en nuestro proyecto, así como sus métodos.

## **Generar aleatoriamente las tonalidades de verde.**

Se van a llenar los arreglos de la población y sus individuos con los valores aleatorios de tonalidades de verde definidas en forma hexadecimal.

## **Definir las funciones de probabilidad y de fitness.**

Con dichas funciones se determina que tan aptos son los individuos y sus cromosomas para sobrevivir y pertenecer a las siguientes generaciones de individuos.

## **Definir la función de cruce**

Con esta función se combinan aquellos cromosomas de cada individuo que resultaron ser más aptos, para formar nuevos individuos a partir de esas combinaciones generadas.

### **Definir la función de selección de padres**

Con esta función se determinan quiénes serán los padres para formar las nuevas generaciones de individuos a partir de los cruces realizados.

### **Definir la función de mutación**

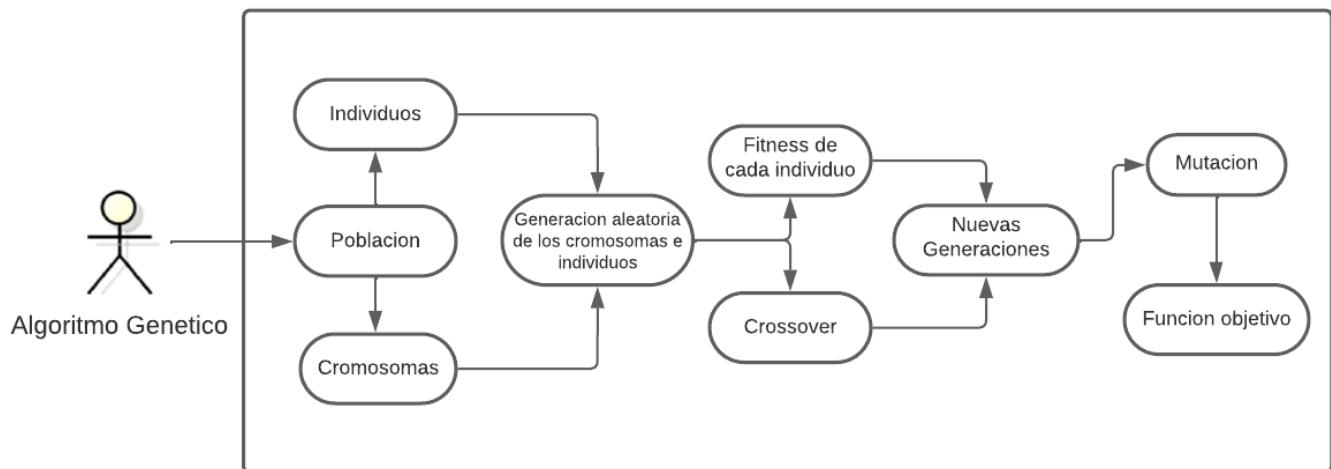
Con esta función, los individuos podrán tener mutaciones, para que generación tras generación, se adapten de una mejor manera al entorno y así obtengamos un mejor resultado. Para ello se seleccionan los valores de cromosomas de manera aleatoria para realizar la mutación respectiva.

### **Definir la función objetivo**

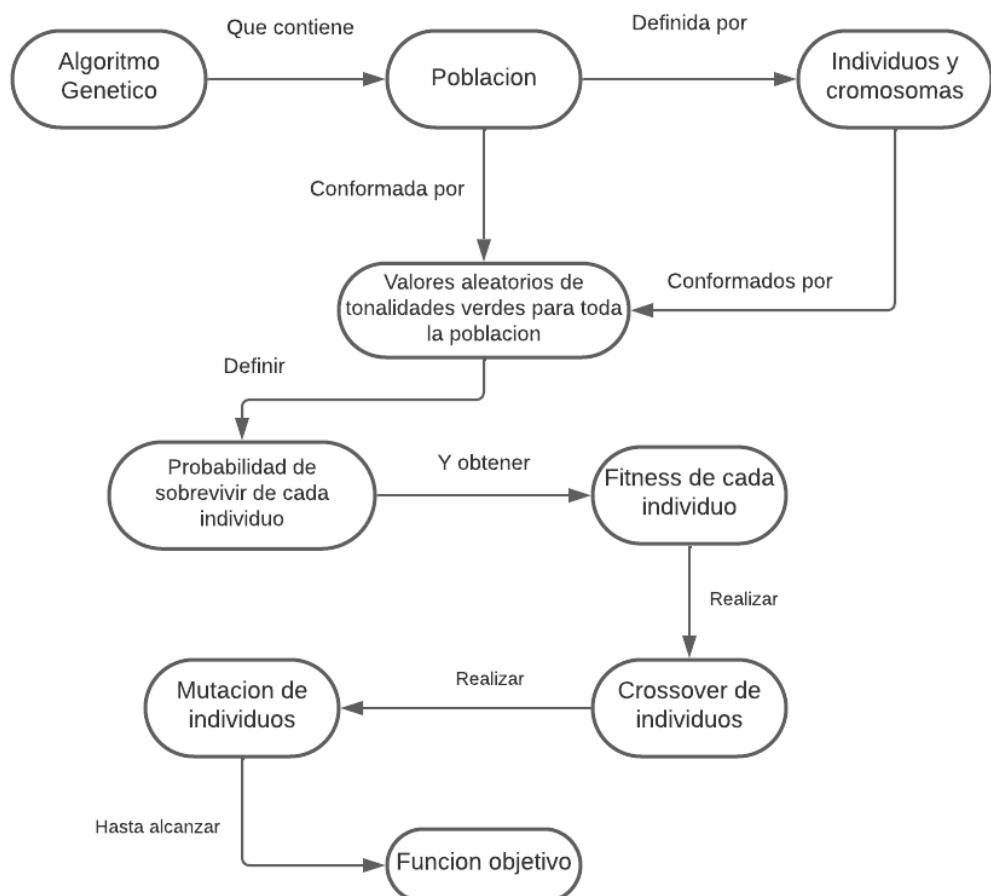
Se define una función que el algoritmo tiene que alcanzar para detenerse en ese punto y no realizar más generaciones. Es su límite de generaciones a realizar.

## Diseño de diagramas

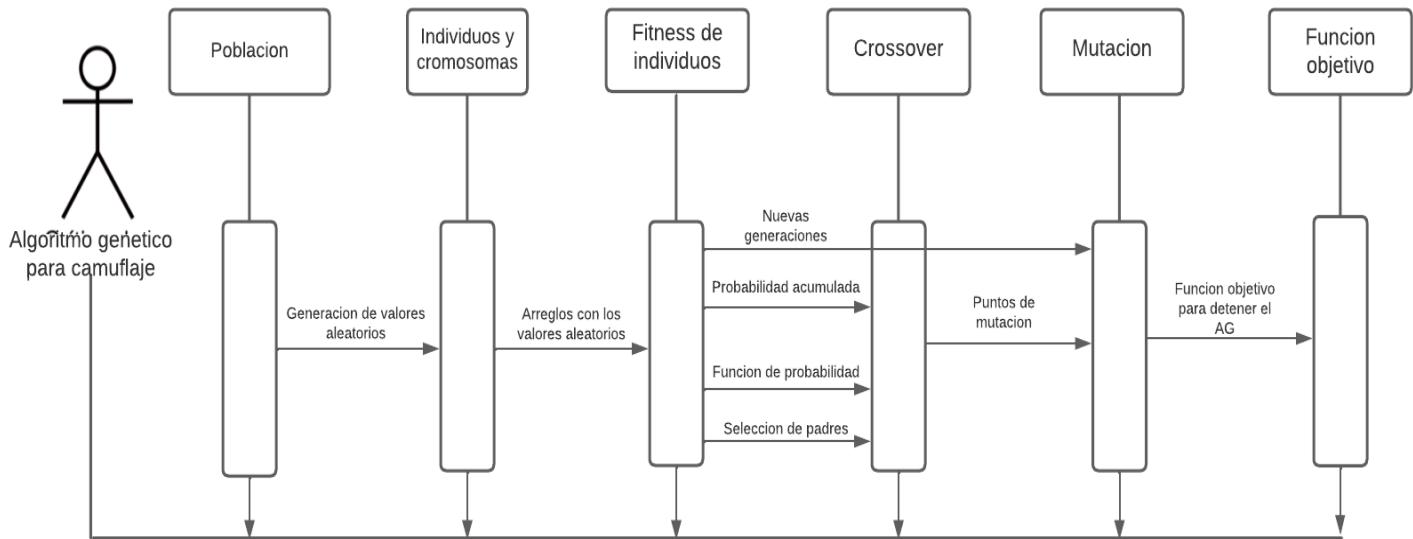
### Modelo de datos.



### Diagrama de procesos

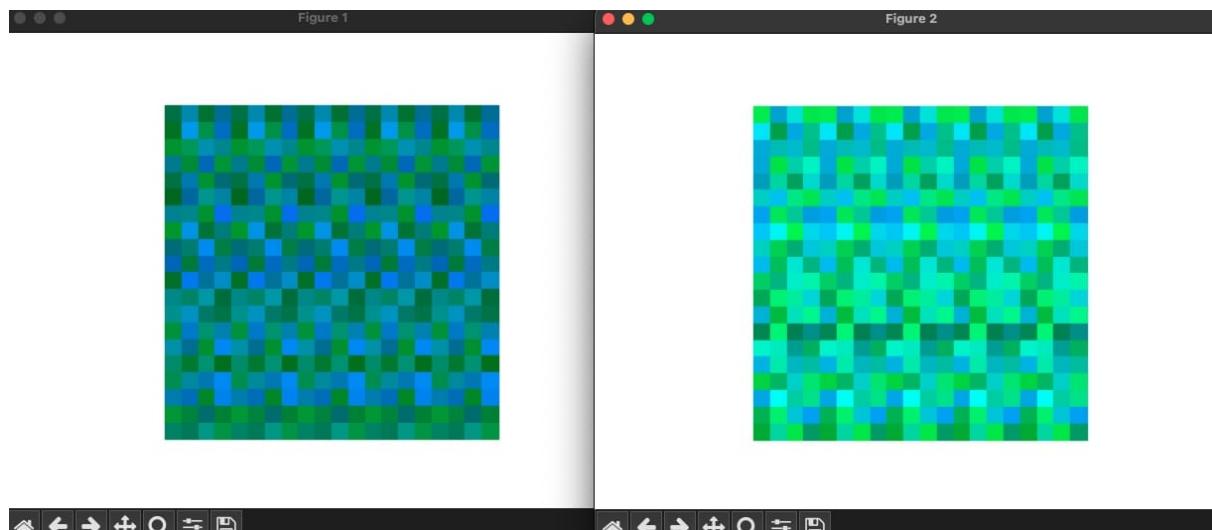


## Diagrama de secuencias

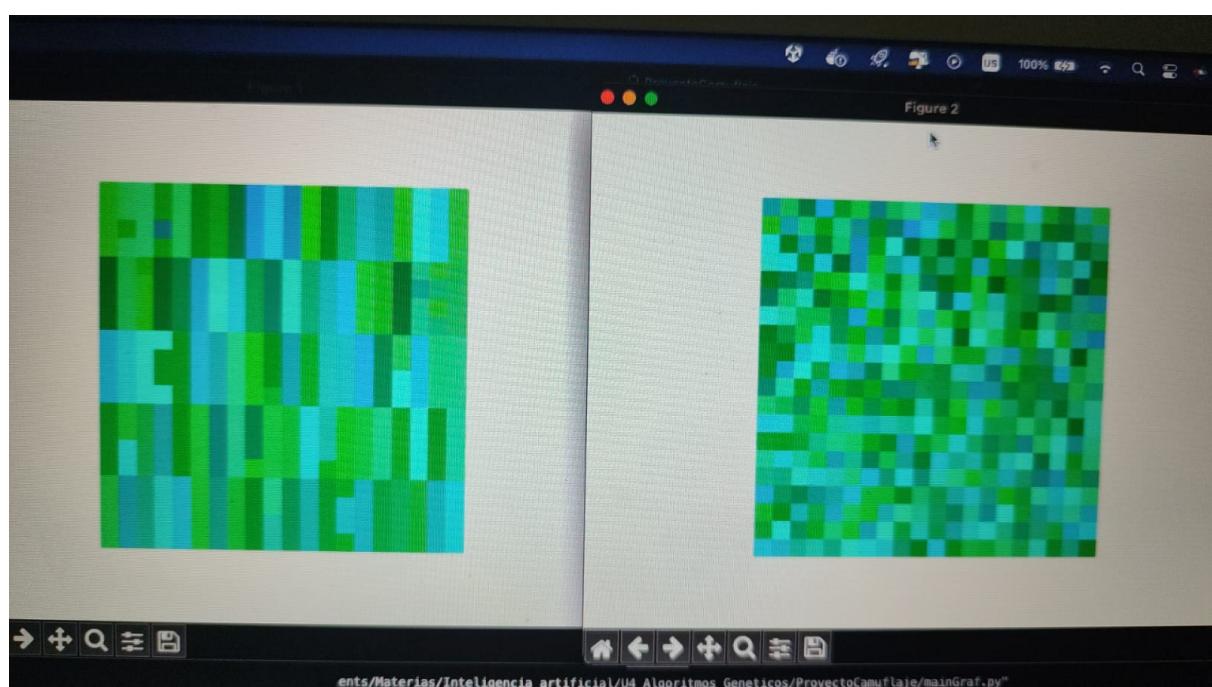


## Prototipos

prototipo no funcional



prototipo que converge



prototipo en consola

```
f(x) = [500546. 468296. 559040. 537543.]
fitness = [500546. 468296. 559040. 537543.]
total = 2065425.0
-----
P = [0.24234528 0.22673106 0.27066584 0.26025782]
C = [0.24234528 0.46907634 0.73974218 1.]
Comparacion del if lo puse antes porque el print lo puse aqui en vez de haya
None
R = [0.0124139 0.97809531 0.67252745 0.05359489]
R2 = [0.37518892 0.09890149 0.0946935 0.94925693]
```

---

```
Matriz con cruces
[[38271 37726 61635 50487]
 [53877 41881 60689 64564]
 [34985 41881 60689 56631]
 [38271 37726 61635 50487]]
Posiciones cromosomas menor a 25% = [1. 2.]
Los puntos de cruce son [1. 3.]
```

---

```
NUMERO DE MUTACIONES 1
posicion seleccionada [15.]
numero de reemplazo[55945.]
CROMOSOMAS MUTADOS
[[38271 37726 61635 50487]
 [53877 41881 60689 64564]
 [34985 41881 60689 56631]
 [38271 37726 55945 50487]]
f(x) = [500546. 577932. 527308. 483476.]
fitness = [500546. 577932. 527308. 483476.]
total = 2089262.0
```

---

```
P = [0.23958029 0.27662017 0.2523896 0.23140994]
C = [0.23958029 0.51620046 0.76859006 1.]
Comparacion del if lo puse antes porque el print lo puse aqui en vez de haya
None
R = [0.65237052 0.64254294 0.32603333 0.48681968]
R2 = [0.06529889 0.86347435 0.48057723 0.87482448]
```

---

```
Matriz con cruces
None
Posiciones cromosomas menor a 25% = [0.]
Los puntos de cruce son [1.]
```

---

```
NUMERO DE MUTACIONES 1
posicion seleccionada [7.]
numero de reemplazo[49246.]
CROMOSOMAS MUTADOS
[[34985 41881 60689 56631]
 [34985 41881 49246 56631]
 [53877 41881 60689 64564]
 [53877 41881 60689 64564]]
f(x) = [500546. 468296. 559040. 537543.]
```

# Implementación

## mainGraf.py

```
⌚ mainGraf.py U ×
⌚ mainGraf.py > ...
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from GeneticAlgorithm import AlgoritmoGenetico
4
5
6 # declaramos la matriz interna
7 matriz_interna = np.empty((4, 4), dtype=object)
8
9
10 |
11 num_chromosomes = 4#numero de cromosomas de la matriz
12 max_value_verde = 0x00FF00
13 min_value_verde = 0x009900
14
15 array = np.array([], dtype=int)
16 cromosomaMutado = np.array([])
17 matrizGrandeFinal = [[[] [0] for k in range(1)]for j in range(5)]for i in range (5)]
18 matrizGrandeInicial = [[[] [0] for k in range(1)]for j in range(5)]for i in range (5)]
19 arrayFin = np.array([])
20 arrayIni = np.array([])
21 populationPrueba = np.array([])
22
```

```
⌚ mainGraf.py U ×
⌚ mainGraf.py > ...
23
24 col = 0
25 ren = 0
26 for i in range(1,26):#SE REPITE 25 QUE ES EL TOTAL DE LA MATRIZ DE 3 DIMENSIONES 5X5
27     population = np.random.randint(min_value_verde, max_value_verde + 1, size=(num_chromosomes, 4))#crea la matriz 4x4 random de 4 individuos
28     populationPrueba = np.array(population)
29
30 ag = AlgoritmoGenetico(population)#llamo al algoritmo que se repita 25 veces con una nueva poblacion por cuadrito osea 25 veces
31
32 generations = 0#el numero de generaciones que han pasado
33 while True:
34     #IF PARA QUE SALGA DEL WHILE DESPUES DE 50 GENERACIONES
35     if generations >=50:
36         # print("No se encontró una solución satisfactoria después de " + str(i))
37         break
38
39     ag.run()
40
41     generations += 1
42
43     array = np.array(ag.cromosomas)
44
45
46     #convierto la poblacion inicial en hexadecimal
47     #_____
48     array_hex1 = np.empty(populationPrueba.shape, dtype="<U7")
49     # Recorrer el array de números enteros y convertirlos a cadenas hexadecimales
50     for i in range(populationPrueba.shape[0]):
51         for j in range(populationPrueba.shape[1]):
52             # Convertir el número entero a una cadena hexadecimal con el prefijo "0x"
53             hexa = hex(populationPrueba[i,j])
54             # Eliminar el prefijo "0x" y añadir el símbolo "#" al principio
55             #hexa = "#" + hexa[2:]
56             hexa = hexa[2:]
57             # Rellenar con ceros a la izquierda si es necesario
58             hexa = hexa.zfill(6)
59             # Guardar la cadena hexadecimal en el array_hex1
60             hexa = "#" + hexa[0:]
61
62             array_hex1[i,j] = hexa
63             cromosomaInicial = np.array(array_hex1)
64             #cromosomaMutado ES EL CROMOSOMA MUTADO FINAAAAL
```

```

⌚ mainGraf.py U ×
⌚ mainGraf.py > ...

#EN ESTA PARTE CONVIERTO EL ARRAY DE ENTEROS A LOS CROMOSOMAS
#_____
array_hex2 = np.empty(array.shape, dtype="<U7")
# Recorrer el array de números enteros y convertirlos a cadenas hexadecimales
for i in range(array.shape[0]):
    for j in range(array.shape[1]):
        # Convertir el número entero a una cadena hexadecimal con el prefijo "0x"
        hexa = hex(array[i,j])
        # Eliminar el prefijo "0x" y añadir el símbolo "#" al principio
        hexa = "#" + hexa[2:]
        hexa = hexa[2:]
        # Rellenar con ceros a la izquierda si es necesario
        hexa = hexa.zfill(6)
        # Guardar la cadena hexadecimal en el array_hex
        hexa = "#" + hexa[0:]

        array_hex2[i,j] = hexa
cromosomaMutado = np.array(array_hex2)

if col < 5:
    matrizGrandeFinal[ren][col] = cromosomaMutado
    matrizGrandeInicial[ren][col] = cromosomaInicial
    col += 1
else:
    ren += 1
    col = 0

    matrizGrandeFinal[ren][0] = cromosomaMutado
    matrizGrandeInicial[ren][col] = cromosomaInicial
    col += 1

arrayFin = np.array(matrizGrandeFinal)
arrayFin = np.reshape(arrayFin, (5,5,16))

arrayIni = np.array(matrizGrandeInicial)
arrayIni = np.reshape(arrayIni, (5,5,16))

#_____
matriz_principal = np.array(arrayFin)
matriz_inicial = np.array(arrayIni)

```

```

⌚ mainGraf.py U ×
⌚ mainGraf.py > ...

#IMPLEMENTACION VISUAL DE LA GRAFICA EN MATLIBPLOP
112 # Función para dibujar una celda
113 def dibujar_celda(ax, x, y, color, celda):
114     ax.add_patch(plt.Rectangle((x, y), celda, celda, color=color))
115
116
117 # Configuración de la cuadrícula
118 fila_principal = matriz_principal.shape[0]
119 columna_principal = matriz_principal.shape[1]
120 filas_internas = 4*matriz_interna.shape[0]
121 columnas_internas = 4*matriz_interna.shape[1]
122 tamaño_celdaPrincipal = 70
123 tamaño_celdaInterna = tamaño_celdaPrincipal / max(filas_internas, columnas_internas)
124
125 # Crear una figura y ejes de Matplotlib
126
127 fig, ax, = plt.subplots()
128
129 # dibujar cuadrícula principal
130 for fila in range(fila_principal):
131     for columna in range(columna_principal):
132         x_inicio = -tamaño_celdaPrincipal * columna_principal / 2 + columna * tamaño_celdaPrincipal
133         y_inicio = tamaño_celdaPrincipal * fila_principal / 2 - fila * tamaño_celdaPrincipal
134         # dibujar_celda(ax, x_inicio, y_inicio, "white", tamaño_celdaPrincipal) # Colorear la celda
135         #colorearMatriz()
136         # # Dibujar la cuadrícula interna
137         posHex = 0
138         for fila_inter in range(filas_internas):
139             for columna_inter in range(columnas_internas):
140                 x_inicio_interno = x_inicio - tamaño_celdaInterna / 20 + columna_inter * tamaño_celdaInterna
141                 y_inicio_interno = y_inicio - tamaño_celdaInterna / 20 - fila_inter * tamaño_celdaInterna
142
143                 # dibujar_celda(ax, x_inicio_interno, y_inicio_interno, matriz_principal[fila][fila_inter][columna_inter],
144                 # | tamaño_celdaInterna) # Colorear la celda
145                 dibujar_celda(ax, x_inicio_interno, y_inicio_interno, matriz_principal[fila][columna][posHex],
146                 | tamaño_celdaInterna) # Colorear la celda
147                 posHex += 1
148
149
150

```

```
⌚ mainGraf.py U ×
⌚ mainGraf.py > ...

152     # Configurar límites y ejes
153     ax.set_xlim(0, tamaño_celdaPrincipal * columna_principal)
154     ax.set_ylim(0, tamaño_celdaPrincipal * fila_principal * filas_internas)
155
156
157     # Mostrar la figura
158
159     plt.axis('equal')
160     plt.axis('off') # Ocultar los ejes
161     plt.title('Generacion ' + str(generations ))
162     #plt.show()
163
164     # Configuración de la cuadricula PARA LA POBLACION INICIAL
165     fila_principal = matriz_inicial.shape[0]
166     columna_principal = matriz_inicial.shape[1]
167     filas_internas = 4*matriz_interna.shape[0]
168     columnas_internas = 4*matriz_interna.shape[1]
169     tamaño_celdaPrincipal = 70
170     tamaño_celdaInterna = tamaño_celdaPrincipal / max(filas_internas, columnas_internas)
171
172     # Crear una figura y ejes de Matplotlib
173
174     fig2, ax2 = plt.subplots()
175
176     # dibujar cuadricula principal
177     for fila in range(fila_principal):
178         for columna in range(columna_principal):
179             x_inicio = -tamaño_celdaPrincipal * columna_principal / 2 + columna * tamaño_celdaPrincipal
180             y_inicio = tamaño_celdaPrincipal * fila_principal / 2 - fila * tamaño_celdaPrincipal
181
182             # # Dibujar la cuadricula interna
183             posHex = 0
184             for fila_inter in range(filas_internas):
185                 for columna_inter in range(columnas_internas):
186                     x_inicio_interno = x_inicio - tamaño_celdaInterna / 20 + columna_inter * tamaño_celdaInterna
187                     y_inicio_interno = y_inicio + tamaño_celdaInterna / 20 - fila_inter * tamaño_celdaInterna
188
189                     dibujar_celda(ax2, x_inicio_interno, y_inicio_interno, matriz_inicial[fila][columna][posHex],
190                                   | tamaño_celdaInterna) # Colorear la celda
191             posHex += 1
192
```

```
⌚ mainGraf.py U ×
⌚ mainGraf.py > ...
194
195     ax2.set_xlim(0, tamaño_celdaPrincipal * columna_principal)
196     ax2.set_ylim(0, tamaño_celdaPrincipal * fila_principal * filas_internas)
197
198     # Mostrar la figura
199     plt.axis('equal')
200     plt.axis('off') # Ocultar los ejes
201     plt.title(' Poblacion inicial')
202     plt.show()
203
```

## GeneticAlgorithm.py

```
遗传Algorithm.py M ×
遗传Algorithm.py > AlgoritmoGenetico > crossoverPoints
1 #ALGORITMO GENETICO EN PYTHON
2 import numpy as np
3 class AlgoritmoGenetico(crossoverPoints):
4     def __init__(self, cromosomas: Any):
5         self.cromosomas = poblacion_inicial
6         self.parentPos = np.array([])
7         #el crosspoint debe de ser de los genes de cromosoma -1 osea abcd 4 - 3, lo dejare estatico pero puedo cambiarlo luego para el proyecto
8
9         self.nRandom = np.array([])
10
11
12     #f(x) = ((a + 2b + 3c + 4d) - 30)
13     def objectiveFunction(self):
14         F_obj = np.zeros(self.cromosomas.shape[0])
15         for i in range(self.cromosomas.shape[0]):
16             #F_obj[i] = abs((self.cromosomas[i,0] + (self.cromosomas[i,1] * 2) + (self.cromosomas[i,2] * 3) + (self.cromosomas[i,3] * 4)) - 30)#MINIMIZAR
17             F_obj[i] = (self.cromosomas[i,0] + (self.cromosomas[i,1] * 2) + (self.cromosomas[i,2] * 3) + (self.cromosomas[i,3] * 4)) - 30
18         return F_obj
19
20     #PASO 3: FUNCION FITNESS
21     def fitnessFunction(self):
22         #Fitness[i] = 1 / (1+F_obj[i]) EJEMPLO
23         functionResult = self.objectiveFunction()
24         Fitness = np.zeros(functionResult.shape)
25         total = 0
26
27         for i in range(self.cromosomas.shape[0]):
28             #Fitness[i] = 1 / (1+functionResult[i])#MINIMIZAR
29             Fitness[i] = functionResult[i]
30             total += Fitness[i]
31
32         return Fitness, total
33
34
35     def probabilityFunction(self):
36         #P[i] = Fitness[i] / Total EJEMPLO
37         #fitness = self.fitnessFunction#minimizar creo
38         fitness = self.fitnessFunction()#0
39         total = self.fitnessFunction()#1
40         P = np.zeros(fitness.shape)
41         for i in range(self.cromosomas.shape[0]):
42             P[i] = fitness[i] / total
43         return P
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
```

```
遗传Algorithm.py M ×
遗传Algorithm.py > AlgoritmoGenetico > crossoverPoints
44
45     def probabilityComulative(self):
46         p = self.probabilityFunction()
47         C[i] = 0.1254 + n
48         C = np.zeros(p.shape)
49         C[0] = p[0]
50         for i in range(self.cromosomas.shape[0]):
51             C[i] = p[i]
52             C[i] = C[i-1] + p[i]
53
54         return C
55
56
57     def comparacionesRandom(self):
58         NewChromosome = np.zeros(self.cromosomas.shape)
59         self.nRandom = np.random.rand(self.cromosomas.shape[0])
60         #for hasta el tamaño de los cromosomas
61         for i in range(len(self.cromosomas)):
62
63             #En esta parte se hace los cambios con los individuos mas fuertes
64             for j in range(len(self.probabilityComulative()) - 1):
65                 if self.nRandom[i] > self.probabilityComulative()[j] and self.nRandom[i] < self.probabilityComulative()[j+1]:
66                     NewChromosome[i] = self.cromosomas[j+1]
67
68
69                 if self.nRandom[i] < self.probabilityComulative()[0]:
70                     NewChromosome[i] = self.cromosomas[0]
71
72
73             #Convertir NewChromosome al tipo de datos de self.cromosomas
74             NewChromosome = NewChromosome.astype(self.cromosomas.dtype)
75
76
77             #guardar los nuevos cambios de los nuevos cromosomas a la poblacion inicial
78             np.copyto(self.cromosomas, NewChromosome)
79
```

```

⌚ GeneticAlgorithm.py M ✘
⌚ GeneticAlgorithm.py > ⌚ AlgoritmoGenetico > ⌚ crossoverPoints
59
60     #EN ESTE PUNTO AHORA SIGUE LO DE SELECCIONAR NUEVOS RANDOMS PARA SACAR LA NUEVA GENERACION
61     def seleccionar_padres(self):
62
63         pc = .30
64         k = 0
65         i = 0
66
67         self.nRandom = np.random.rand(len(self.cromosomas))
68
69         tam = 0
70         for j in range(len(self.nRandom)):
71             if self.nRandom[j] <= pc:
72                 tam += 1
73
74         self.parentPos = np.empty(tam)
75
76
77         while k < len(self.cromosomas):
78             #este lo activo cuando ya los reemplaza con random
79             #self.nRandom[k] = np.random.rand
80             if self.nRandom[k] <= pc:
81                 self.parentPos[i] = k
82             #AQUI GUARDO EN PARENT POSITION K QUE ES LA POSICION DEL CROMOSOMA QUE FUE MENOR DE .25(PC)
83             # parent[k] = k
84             # i = i + 1# LE VOY AUMENTANDO AL INDICE SI EL IF HIZO MATCH, PARA SEGUIR GUARDANDO EN PARENT
85             i = i + 1# LE VOY AUMENTANDO AL INDICE SI EL IF HIZO MATCH, PARA SEGUIR GUARDANDO EN PARENT
86
87             k = k + 1
88
89
90         #APARTIR DE AQUI SELECCIONAREMOS CUAL ES EL PUNTO DE CRUZ EN LAS POSICIONES
91
92         self.crossPoint = np.empty(len(self.parentPos))
93         for i in range(len(self.parentPos)):
94             # se genera de 1 a cromosomas -1 que seria abcd son 4 -1 son 3, NO PONGO -1 PORQUE COMO EMPIEZA EN 1 Y LA POSICION CUENTA COMO 0 SE EQUILIBRA
95             primer_arreglo = self.cromosomas[0]
96             self.crossPoint[i] = np.random.randint(1, primer_arreglo.shape[0])
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117

```

```

⌚ GeneticAlgorithm.py M ●
⌚ GeneticAlgorithm.py > ⌚ AlgoritmoGenetico > ⌚ crossoverPoints
120     #recibe dos parametros la posicion de los cromosomas seleccionados y los puntos de cruce
121     #tambien se requeriran los cromosomas
122     def crossoverPoints(self):
123         pos = self.parentPos
124         point = self.crossPoint
125
126         if len(pos) <= 1:
127             return
128
129         self.cromosomas = self.cromosomas.astype(int)
130         pos = pos.astype(int)
131         # Crear una copia temporal del arreglo porque si guardo en el principal los siguientes cruces tomaran
132         # los nuevos datos envez de los anteriores
133         temp = np.copy(self.cromosomas)
134         #lo qu hago es que para cada self.cromosomas que tomo de point(posiciones de self.cromosomas) le extraigo cada gen que seria abcd, son diferentes ifs,
135         # en caso de que el punto de cruce sea 1 pues toma 1 y los otros 3 del otro array, si son 2 pues toma 2 del primero y 2 del otro, y si sonn 3
136         # pues toma 3 del primero y 1 del otro [primer cruce Chromosome[1] >< Chromosome[4]
137
138         for i in range(len(pos)):
139             gen = np.array([])
140             gen = np.append(gen, self.cromosomas.item(pos[i],0))
141             punto = point
142             punto = punto.astype(int)
143             lenGen = self.cromosomas[0]
144
145             source = 1
146             for j in range(1,lenGen.shape[0]):# -1 porque en la ultima posicion se va a regresar, CREO QUE AQUI LO
147                 #SEVOY A CAMBIAR HASTA EL TAMAÑO DE LOS GENES OSEA ABCD
148                 if i == len(punto) - 1:#si es igual a la ultima posicion regresa a comparar la primera
149                     if j != punto[i]:
150                         gen = np.append(gen, self.cromosomas.item(pos[i],j))
151                     else:
152                         gen = np.append(gen, self.cromosomas.item(pos[0],j))
153
154                 elif source < point[i]:#AQUI LE TENGO QUE CAMBIAR ENTRADA NUEVA
155                     gen = np.append(gen, self.cromosomas.item(pos[i],j))
156                 else:
157                     gen = np.append(gen, self.cromosomas.item(pos[i+1],j))
158
159                 source += 1
160
161             #Entro las veces necesarias
162             temp[pos[i]] = gen;

```

```

➊ GeneticAlgorithm.py M •
➋ GeneticAlgorithm.py > ⚭ AlgoritmoGenetico > ⚭ crossoverPoints
165     #MUEVO MI ARREGLO COPIA AL PRINCIPAL PARA REFLEJAR LOS CAMBIOS
166     self.cromosomas = temp
167     return self.cromosomas
168
169
170 def mutationPoints(self):
171     #VALIDACION PARA EL ALGORITMO GENETICO
172     #le pondre 20%, genera un numero aleatorio si es mayor o igual a 2 no lo ejecuta
173     #porque hay un 20% de que salga 1 o 2, porque son numeros del 1 al 10
174     number = np.random.randint(1, 11)
175     if number < 1:
176         return
177
178     #AQUI COMIENZA LO DE LA MUTACION
179     #MAS ADELANTE PUEDO SACAR ESTO PARA HACER LOS PUNTOS DEPENDIENDO DE OTRA ESTRUCTURA
180     #total_gen = number_of_gen_in_Chromosome * number_of_population
181     #total_gen = 4 * 6
182
183     lenGen = self.cromosomas[0]
184
185     total_gen = lenGen.shape[0] * self.cromosomas.shape[0]
186
187
188     number_of_mutations = 0.1 * total_gen
189
190
191     #para poder iterar en number_of_lo convertimos a int para que redonde el numero de mutaciones
192     number_of_mutations = int(number_of_mutations)
193     arrPosition = np.array([])
194     arrCambios = np.array([])
195
196     for i in range(number_of_mutations):
197         #Este random es la posicion de la poblacion que se reemplazara
198         Rposition = np.random.randint(1,total_gen + 1)
199         arrPosition = np.append(arrPosition, Rposition)
200
201         #Este random es el numero por el cual se reemplazara
202         #RANDOMS DE COLORES VERDES HEXAS
203         Rcambio = np.random.randint(0x009900,0xFFFF00)
204         arrCambios = np.append(arrCambios, Rcambio)
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
```

```

➊ GeneticAlgorithm.py M •
➋ GeneticAlgorithm.py > ⚭ AlgoritmoGenetico > ⚭ crossoverPoints
205
206     #AQUÍ HAGO EL INTERCAMBIO EN LA POSICION SELECCIONADA
207     np.put(self.cromosomas,Rposition-1, Rcambio)
208     return self.cromosomas, arrPosition, arrCambios, number_of_mutations
209
210
211 def run(self):
212     self.objectiveFunction()
213     self.fitnessFunction()
214     self.probabilityFunction()
215     self.probabilityCumulative()
216     # self.comparacionesRandom()# DESACTIVE ESTO PORQUE AL HACER LA COMPARACION CONVERGIA DEMASIADO RAPIDO EL ALGORITMO
217     self.seleccionar_padres()
218     self.crossoverPoints()
219     self.mutationPoints()
```

## **Pruebas**

### **Prueba de caja negra**

En las pruebas de caja negra, nos enfocamos solamente en cómo el usuario interactúa con el sistema, sin conocer nada del código interno. Para esto se realizó la ejecución del programa varias veces, para comprobar que el sistema de generación de patrones funcione correctamente, al ser un programa que no requiere datos de entrada por parte del usuario la prueba se realizó de manera exitosa.

### **Prueba de caja blanca**

Las pruebas de caja blanca (también conocidas como pruebas de caja de cristal o pruebas estructurales) se centran en los detalles procedimentales del software, por lo que su diseño está fuertemente ligado al código fuente.

### **Prueba de rendimiento**

El objetivo de las pruebas de rendimiento es determinar el rendimiento del sistema bajo una carga de trabajo definida utilizando diferentes tipos de pruebas de rendimiento tales como pruebas de carga, de estrés y de estabilidad. Al ser un programa sencillo en el que se ejecuta una vez, no hubo fallas en el rendimiento de nuestro sistema.

### **Pruebas de funcionalidad**

Dentro de la presente prueba se realiza la integridad del proyecto, basándose en su correcto funcionamiento y que todo se realice según lo establecido sin ninguna falla en su proceso. Los patrones se dibujaron y se comprobó que los patrones se realizaron correctamente.

## Prueba de Ejecución

Despues de ejecutar el mainGraf.py obtenemos la salida deseada



## Funcionamiento

Para el funcionamiento del proyecto se creó una clase llamada Genetic algorithm, esta clase es donde se almacenan todos los métodos y procesos que realizará nuestro algoritmo. Esta clase recibe como parámetro la población inicial, para generar nuestra población, dentro de nuestro main definimos la cantidad de individuos que esté pulsera además de los cromosomas que contendrá cada individuo, en este caso se cuenta con una matriz de 4x4 interna, por lo cual se cuenta con 4 individuos y 4 cromosomas individuales, 16 cromosomas en total de la población.

Cuando se genera la población se generan números aleatorios en un rango de colores verdes hexadecimales para los valores de nuestros cromosomas.

Ahora se crea el objeto de la clase del algoritmo pasandole nuestra población inicial para posteriormente ejecutarla con el método run que sólo contiene el llamado a los métodos de nuestro algoritmo, nuestra clase se ejecuta dentro de un bucle for de tamaño 25 debido a que nuestra matriz principal es de 5x5, esto quiere decir que se creara una nueva población 25 veces, dentro de cada población se decidió ejecutar el algoritmo un total de 50 generaciones para encontrar los mejores resultados.

Dentro de cada llamada a nuestro algoritmo se ejecutan un total de 7 metodos de 8 con los que se cuentan, el primero se encarga de encontrar la funcion objetivo de cada individuo mediante la funcion que se cuenta, en este caso para el algoritmo como dato se utilizo una funcion de maximizar, Cuando se calcula la funcion se ejecuta nuestra funcion fitness o de aptitud la cual esta es la que nos permite evaluar que tan bueno es el individuo segun lo que estamos buscando en nuestra funcion, de esta manera los individuos mas altos tienen mas posibilidades de ser seleccionados en el cruce, cuando se realiza esto se pasa a calcular la probabilidad conmutativa que no es mas que la suma de la probabilidad de nuestro fitness y la suma de esto nos debe dar 1, con lo cual se pasa al siguiente paso que es la comparacion de numeros randoms comparando si este numero esta en el rango de la probabilidad comulativa entre dos individuos en caso de que no recorre la tabla, y al final selecciona el individuo del rango mas alto, una vez generado esto se procede a reemplazar en la poblacion a los individuos mas debiles, con los que resultaron seleccionados en las comparaciones, en este caso dentro de este programa en especifico se omite este paso que se explicara mas adelante en las observaciones.

En este punto se continua con el cruce de los cromosomas, para esto en este metodo llamado seleccionar\_padres se generan nuevos numeros aleatorios de 0 a 1 por cada individuo de la generacion, para posteriormente comparar si este numero es menor a 30% que fue la probabilidad que se eligio, esto se hace de esta manera debido a que es con una ruleta y esto ayuda a que nuestro algoritmo no converga demasiado rapido y exista una mayor probabilidad de exito, una vez que han sido seleccionados los padres se pasa al siguiente metodo el cual es el crossover, dentro de este lo que se realiza es obtener los puntos de cruce para los individuos seleccionados, generando un numero aleatorio de el total de cromosomas de cada individuo - 1 en este caso serian numeros del 1 al 3, esto nos sirve para poder cruzar y reemplazar los cromosomas de los individuos que han sido seleccionados y reemplazarlos segun los puntos de cruce, una vez que se tienen los nuevos cromosomas, se pasa al siguiente y ultimo metodo el cual es el de mutaciones, dentro de este primero se genera un nuevo numero aleatorio que esta dentro del rango de colores verdes hexadecimales, este sera el nuevo valor de la mutacion con el cual sera cambiado, acto seguido generamos un valor aleatorio desde 0 hasta el numero de cromosomas totales de nuestra poblacion que en este caso seria 16 que

es nuestro total, conociendo ya estos datos procedemos a realizar la mutacion de los cromosomas con el nuevo numero y la posicion resultante, esta mutacion no siempre se ejecutara originalmente se tenia planeado que se tuviera un 10% de probabilidades de mutacion pero esta tasa se aumento por diversas razones.

Finalmente fuera de nuestro algoritmo y después de ejecutarlo 50 generaciones obtenemos nuestra población final la cual guardaremos en un array de numpy para después poder mostrarla junto con nuestras otras 24 poblaciones mas de nuestra tabla, estas poblaciones en hexadecimal fueron mostradas gráficamente con la librería matplotlib que nos permite manejar diversos tipos de gráficas en python.

## **Observaciones**

Como primera observación se tiene que se tuvo que desactivar el reemplazo de los individuos por medio de las comparaciones, debido a que este reemplazo provocaba que el algoritmo genético convergieron demasiado rápido, incluso en las primeras generaciones, precisamente para este proyecto no era óptimo debido a que nuestro salida es mostrar una interfaz gráfica de nuestros cromosomas del algoritmo, y con la convergencia demasiado rápido no se lograba apreciar correctamente el funcionamiento del algoritmo.

La segunda observación se tiene al momento de realizar la mutación, se decidió aumentar exponencialmente la probabilidad de que esto suceda debido a que con una tasa alta de mutación podemos observar claramente el funcionamiento de todas las partes y correcto funcionamiento del algoritmo, además ayudaba a que el algoritmo no converja rápidamente, y nos ayuda a tener una variedad mayor de mutaciones para mostrar en nuestra gráfica de colores verdes hexadecimales.

## **Conclusiones**

La generación de camuflaje utilizando algoritmos genéticos representa una aplicación innovadora de la inteligencia artificial que puede tener un impacto significativo en campos como la milicia. Estos algoritmos permiten la creación de patrones de camuflaje adaptables y eficaces, que pueden ajustarse automáticamente a entornos cambiantes. El desarrollar el proyecto, amplio nuestros conocimientos sobre la inteligencia artificial y en específico sobre algoritmos genéticos, ya que uno podría pensar desde el punto de vista biológico que no se puede replicar en una computadora, pero como observamos si se puede conseguir de cierta forma, y más aún se puede aplicar a muchos sistemas que lo necesitan, ya que los algoritmos genéticos son una poderosa técnica de optimización y búsqueda inspirada en la evolución natural que ha demostrado su eficacia en una amplia gama de aplicaciones, como quedó demostrado en el proyecto estos son capaces de encontrar soluciones aproximadas a problemas complejos y pueden adaptarse a entornos cambiantes.

## Bibliografía

- [1] “Algoritmos genéticos”. Conogasi. Accedido el 11 de noviembre de 2023. [En línea]. Disponible: <https://conogasi.org/articulos/algoritmos-geneticos/>
- [2] “¿Qué son los algoritmos genéticos? | fundación dr. antoni esteve”. Fundación Dr. Antoni Esteve. Accedido el 11 de noviembre de 2023. [En línea]. Disponible: <https://www.esteve.org/publicaciones/nosotras-respondemos-algoritmos/>
- [3] Enzyme. “Algoritmos genéticos y su gran cantidad de aplicaciones”. Enzyme, consultoría tecnológica | Beyond the unexpected. Accedido el 11 de noviembre de 2023. [En línea]. Disponible: <https://enzyme.biz/blog/algoritmos-geneticos-y-sus-aplicaciones-para-soluciones>