Practica 2. Algoritmo genético para codificación de permutaciones.

Algoritmos Bioinspirados

Diego Castillo Reyes
Marthon Leobardo Yañez Martinez
Aldo Escamilla Resendiz

28 de abril de 2024

Índice general

1. Introducción

En esta práctica se implementó un algoritmo genético para resolver el problema de codificación de permutaciones. En especifico encontrar las combinaciones para las soluciones de un cuadrado mágico. El cuadrado mágico se refiere a una matriz cuadrada de números enteros en la que la suma de los números en cada fila, columna y diagonal es la misma.

Por ejemplo:

8	1	6
3	5	7
4	9	2

Cuadro 1: Ejemplo de un cuadrado mágico

En este ejemplo la suma de los números en cada fila, columna y diagonal es 15.
La idea del algoritmo genético es encontrar la permutación de los números del 1 al 9 que formen un cuadrado mágico de tamaño nxn.

2. Desarrollo

Para la implementación del algoritmo genético se utilizó el lenguaje de programación Python.

```python
import math
import random
import copy
import sys
import time
import matplotlib.pyplot as plt
import statistics

random.seed(23)

n = 3
nn = n * n
n2 = nn // 2
magicNumber = n * (n ** 2 + 1) / 2
poblationSize = 200
elites = 150
alfaMutaciones = 0.9
probMutacion = 1
epocas = 1000000
intentos_cruce = 20
tenencia_tabu = 10

poblacion = []
aptitud = []
tabu = []

def generarIndividuos():
    populationGen = []
    for _ in range(poblationSize):
        cuadrado = list(range(1, nn + 1))
        random.shuffle(cuadrado)
        populationGen.append(cuadrado)
    return populationGen

def aptitudPoblacional(poblacionActual):
    aptitudes = []
    for individuo in poblacionActual:
        aptitudes.append(calcularAptitudCuadrado(individuo))
    return aptitudes
```

```python

def calcularAptitudCuadrado(cuadrado):
    aptitud = 0
    suma = 0
    for i in range(nn):
        if (i % n == 0 and i != 0):
            aptitud += abs(magicNumber - suma)
            suma = 0
        suma += cuadrado[i]
    aptitud += abs(magicNumber - suma)

    for j in range(n):
        suma = 0
        for i in range(j, nn, n):
            suma += cuadrado[i]
        aptitud += abs(magicNumber - suma)

    suma = 0
    for i in range(0, nn, n + 1):
        suma += cuadrado[i]
    aptitud += abs(magicNumber - suma)

    suma = 0
    for i in range(n - 1, nn - 1, n - 1):
        suma += cuadrado[i]
    aptitud += abs(magicNumber - suma)

    return aptitud

def verificar_exito(aptitudes):
    if 0.0 in aptitudes:
        return aptitudes.index(0.0), 'exito'

    error = 5
    for indice, aptitud in enumerate(aptitudes):
        if aptitud <= error:
            return indice, 'error_minimo'

    return -1, 'no_encontrado'

def es_cuadrado_valido(cuadrado):
    numeros = set(range(1, nn + 1))
    cuadradoPosible = set(cuadrado)

    if numeros == cuadradoPosible:
        return True
    return False

def fronteras(poblacionActual, aptitudes):
    combinados = list(zip(poblacionActual, aptitudes))
    combinados = sorted(combinados, key=lambda x: x[1])
    elites, _ = zip(*combinados)
    mejorAptitud = aptitudes[0]
    peorAptitud = aptitudes[-1]
    mejor = sum(aptitudes) / len(aptitudes)
    dessviacionSTD = statistics.stdev(aptitudes)
    print(f"Mejor Aptitud: {mejorAptitud}")
    print(f'Peor Aptitud: {peorAptitud}')
    print(f'Aptitud Promedio: {mejor}')
    print(f'Desviacion Estandar: {dessviacionSTD}')
    print(f'Élite Top: {elites[0]}')
    return list(elites)

def mutar(cuadrado):
    numGenes = len(cuadrado)
    numMutaciones = random.randint(1, numGenes)

    for _ in range(numMutaciones):
        i = random.randint(0, numGenes - 1)
        j = random.randint(0, numGenes - 1)
        cuadrado[i], cuadrado[j] = cuadrado[j], cuadrado[i]
```

```
111
112     return cuadrado
113
114 def reproducir(poblacionActual, aptitudes):
115     poblacion2 = copy.deepcopy(poblacionActual)
116     apareamiento = []
117     generacion2 = []
118
119     individuosRestantes = poblationSize - elites
120     elites = fronteras(poblacionActual, aptitudes)
121     unicos_elites = []
122     for elite in elites:
123         if elite not in unicos_elites:
124             unicos_elites.append(elite)
125
126     generacion2 = unicos_elites[0:elites]
127
128     sumaAptitudes = 0
129     inversoAptitudes = []
130     for aptitud in aptitudes:
131         inversoAptitudes.append(10000 - aptitud)
132     for aptitud in inversoAptitudes:
133         sumaAptitudes += aptitud
134
135     apareamiento.append(inversoAptitudes[0] / sumaAptitudes)
136     for i in range(1, len(inversoAptitudes)):
137         probabilidad = inversoAptitudes[i] / sumaAptitudes
138         apareamiento.append(probabilidad + apareamiento[i - 1])
139
140     while individuosRestantes > 0:
141         eleccion1 = random.random()
142         eleccion2 = random.random()
143         indice1 = 0
144         indice2 = 0
145         for i in range(len(apareamiento)):
146             if eleccion1 <= apareamiento[i]:
147                 indice1 = i
148                 break
149         for i in range(len(apareamiento)):
150             if eleccion2 <= apareamiento[i]:
151                 indice2 = i
152                 break
153
154         candidato1 = copy.copy(poblacion2[indice1])
155         candidato2 = copy.copy(poblacion2[indice2])
156
157         hijo1, hijo2 = crossover(candidato1, candidato2)
158
159         if (hijo1 not in generacion2) and (hijo1 not in tabu):
160             generacion2.append(hijo1)
161             individuosRestantes -= 1
162             if len(tabu) >= tenencia_tabu:
163                 tabu.pop(0)
164             tabu.append(hijo1)
165
166         if ((hijo2 not in generacion2) and (hijo2 not in tabu) and individuosRestantes > 0):
167             generacion2.append(hijo2)
168             individuosRestantes -= 1
169             if len(tabu) >= tenencia_tabu:
170                 tabu.pop(0)
171             tabu.append(hijo2)
172
173     print()
174     return generacion2
175
176 def crossover(padre1, padre2):
177     index1 = random.randint(0, nn - 1)
178     index2 = random.randint(0, nn - 1)
179     if index1 > index2:
180         index1, index2 = index2, index1
181
```

```
    hijo1 = [None] * nn
    hijo2 = [None] * nn

    hijo1[index1:index2 + 1] = padre2[index1:index2 + 1]
    hijo2[index1:index2 + 1] = padre1[index1:index2 + 1]

    def completarHijo(hijo, padre):
        posicion = (index2 + 1) % nn
        for gen in padre:
            if gen not in hijo:
                hijo[posicion] = gen
                posicion = (posicion + 1) % nn

    completarHijo(hijo1, padre1)
    completarHijo(hijo2, padre2)

    return hijo1, hijo2

poblacion = generarIndividuos()
historial = {'mejor': [], 'peor': [], 'promedio': [], 'desviacion': []}

for i in range(epocas):
    print(f"Generación actual: {i}")
    aptitud = aptitudPoblacional(poblacion)

    historial['mejor'].append(min(aptitud))
    historial['peor'].append(max(aptitud))
    historial['promedio'].append(sum(aptitud) / len(aptitud))

    if i % 500 == 0 and i != 0:
        for j in range(len(poblacion)):
            print(f'Cuadrado: {poblacion[j]}  Aptitud: {aptitud[j]}')
        time.sleep(3)

    indice, estado = verificar_exito(aptitud)
    if estado == 'exito':
        print(f"Cuadrado Mágico Encontrado: {poblacion[indice]}")
        break
    elif estado == 'error_minimo':
        print(f"Cuadrado Mágico Encontrado: {poblacion[indice]} con Aptitud {aptitud[indice]}")
        break

    poblacion = reproducir(poblacion, aptitud)

generaciones = list(range(len(historial['mejor'])))

plt.scatter(generaciones, historial['mejor'], color='green', label='Mejor Aptitud')
plt.plot(generaciones, historial['mejor'], color='green')

plt.scatter(generaciones, historial['peor'], color='red', label='Peor Aptitud')
plt.plot(generaciones, historial['peor'], color='red')

plt.scatter(generaciones, historial['promedio'], color='blue', label='Aptitud Promedio')
plt.plot(generaciones, historial['promedio'], color='blue')

plt.legend()
plt.xlabel('Épocas')
plt.ylabel('Aptitud')
plt.title("Gráfico de Convergencia")
plt.show()



random.seed(23)
n = 4
nn = n * n
n2 = nn // 2
magicNumber = n * (n ** 2 + 1) / 2
tamanioPoblacion = 200
elites = 150
num_mututacion = 0.9
```

```
mutationChance = 1
epoch = 1000000
intentoCruza = 20
tabu2 = 10

poblacion = []
aptitud = []
listaTabu = []

def generarIndividuos():
    pop = []
    for _ in range(tamanioPoblacion):
        c = list(range(1, nn + 1))
        random.shuffle(c)
        pop.append(c)
    return pop

def obtenerAptitudPoblacional(pop):
    fit = []
    for c in pop:
        fit.append(encontrarAptitud2(c))
    return fit

def encontrarAptitud2(hijo):
    fit = 0
    suma = 0
    for i in range(nn):
        if (i % n == 0 and i != 0):
            fit += abs(magicNumber - suma)
            suma = 0
        suma += hijo[i]
    fit += abs(magicNumber - suma)

    for j in range(n):
        suma = 0
        for i in range(j, nn, n):
            suma += hijo[i]
        fit += abs(magicNumber - suma)

    suma = 0
    for i in range(0, nn, n + 1):
        suma += hijo[i]
    fit += abs(magicNumber - suma)

    suma = 0
    for i in range(n - 1, nn - 1, n - 1):
        suma += hijo[i]
    fit += abs(magicNumber - suma)

    return fit

def wins(apt):
    if 0.0 in apt:
        return apt.index(0.0), 'exito'

    umbral_error_minimo = 5
    for indice, valor_aptitud in enumerate(apt):
        if valor_aptitud <= umbral_error_minimo:
            return indice, 'error_minimo'

    return -1, 'no_encontrado'


def esCuadradoValido(criatura):
    numeros = set(range(1, nn + 1))
    cuadrado_posible = set(criatura)

    if numeros == cuadrado_posible:
        return True
    return False

```

```
324
325 def obtenerElites(pop, fit):
326     combinados = list(zip(pop, fit))
327     combinados = sorted(combinados, key=lambda x: x[1])
328     elites, trash = zip(*combinados)
329     betterApt = trash[0]
330     worstApt = trash[-1]
331     elBueno = sum(trash) / len(trash)
332     desviacion = statistics.stdev(aptitud)
333     print(f"Mejor Aptitud: {betterApt}")
334     print(f'Peor Aptitud: {worstApt}')
335     print(f'Aptitud Promedio: {elBueno}')
336     print(f'Desviacion Estandar: {desviacion}')
337     print(f'Élite Top: {elites[0]}')
338     return list(elites)
339
340 def mutar(hijo):
341     nn = len(hijo)
342     num_mut = random.randint(1, nn)
343
344     for _ in range(num_mut):
345         i = random.randint(0, nn - 1)
346         j = random.randint(0, nn - 1)
347         hijo[i], hijo[j] = hijo[j], hijo[i]
348
349     return hijo
350
351 def reproducir(pop, fit):
352     popc = copy.deepcopy(pop)
353     apareamiento = []
354     siguienteGen = []
355
356     numeroRestante = tamanioPoblacion - elites
357     elites = fronteras(pop, fit)
358     elites = []
359     for e in elites:
360         if not e in elites:
361             elites.append(e)
362
363     siguienteGen = elites[0:elites]
364
365     fitsum = 0
366     bigfit = []
367     for i in aptitud:
368         bigfit.append(10000 - i)
369
370     for i in bigfit:
371         fitsum += i
372
373     apareamiento.append(bigfit[0] / fitsum)
374     for i in range(1, len(bigfit)):
375         prob = bigfit[i] / fitsum
376         apareamiento.append(prob + apareamiento[i - 1])
377
378     while numeroRestante > 0:
379         eleccion1 = random.random()
380         eleccion2 = random.random()
381         indice1 = 0
382         indice2 = 0
383         for i in range(len(apareamiento)):
384             if eleccion1 <= apareamiento[i]:
385                 indice1 = i
386                 break
387         for i in range(len(apareamiento)):
388             if eleccion2 <= apareamiento[i]:
389                 indice2 = i
390                 break
391
392         candidato1 = copy.copy(popc[indice1])
393         candidato2 = copy.copy(popc[indice2])
394
```

```
        hijo1, hijo2 = crossover(candidato1, candidato2)

        if (not hijo1 in siguienteGen) and (not hijo1 in listaTabu):
            siguienteGen.append(hijo1)
            numeroRestante -= 1
            if len(listaTabu) >= tabu2:
                listaTabu.pop(0)
            listaTabu.append(hijo1)

        if ((not hijo2 in siguienteGen) and (not hijo2 in listaTabu) and numeroRestante > 0):
            siguienteGen.append(hijo2)
            numeroRestante -= 1
            if len(listaTabu) >= tabu2:
                listaTabu.pop(0)
            listaTabu.append(hijo2)

    print()
    return siguienteGen

def crossover(padre1, padre2):
    nSquared = len(padre1)
    index1 = random.randint(0, nSquared - 1)
    index2 = random.randint(0, nSquared - 1)
    if index1 > index2:
        index1, index2 = index2, index1

    child1 = [None] * nSquared
    child2 = [None] * nSquared

    child1[index1:index2 + 1] = padre2[index1:index2 + 1]
    child2[index1:index2 + 1] = padre1[index1:index2 + 1]

    def completeChild(child, parent):
        current_pos = (index2 + 1) % nSquared
        for gene in parent:
            if gene not in child:
                child[current_pos] = gene
                current_pos = (current_pos + 1) % nSquared

    completeChild(child1, padre1)
    completeChild(child2, padre2)

    return child1, child2

poblacion = generarIndividuos()
historialAptitud = {'mejor': [], 'peor': [], 'promedio': [], 'desviacion': []}

for i in range(epoch):
    print(f"Generación actual: {i}")
    aptitud = obtenerAptitudPoblacional(poblacion)

    historialAptitud['mejor'].append(min(aptitud))
    historialAptitud['peor'].append(max(aptitud))
    historialAptitud['promedio'].append(sum(aptitud) / len(aptitud))


    if i % 500 == 0 and i != 0:
        for i in range(len(poblacion)):
            print(f'Cuadrado: {poblacion[i]}  Aptitud: {aptitud[i]}')
        time.sleep(3)

    indice, estado = wins(aptitud)
    if estado == 'exito':
        print(f"Cuadrado Mágico Encontrado: {poblacion[indice]}")
        break
    elif estado == 'error_minimo':
        print(f"Cuadrado Mágico Encontrado: {poblacion[indice]} con Aptitud {aptitud[indice]}")
        break

    poblacion = reproducir(poblacion, aptitud)
```

```
generaciones = list(range(len(historialAptitud['mejor'])))

plt.scatter(generaciones, historialAptitud['mejor'], color='green', label='Mejor Aptitud')
plt.plot(generaciones, historialAptitud['mejor'], color='green')

plt.scatter(generaciones, historialAptitud['peor'], color='red', label='Peor Aptitud')
plt.plot(generaciones, historialAptitud['peor'], color='red')

plt.scatter(generaciones, historialAptitud['promedio'], color='blue', label='Aptitud Promedio')
plt.plot(generaciones, historialAptitud['promedio'], color='blue')

plt.legend()
plt.xlabel('Generaciones')
plt.ylabel('Aptitud')
plt.title("Gráfica de convergencia")
plt.show()
```

3. Resultados

3.1 Cuadrados mágicos de 3x3

En la primera ejecución usando la semilla 1, tardando 6 generaciones, dio como resultado lo siguiente:

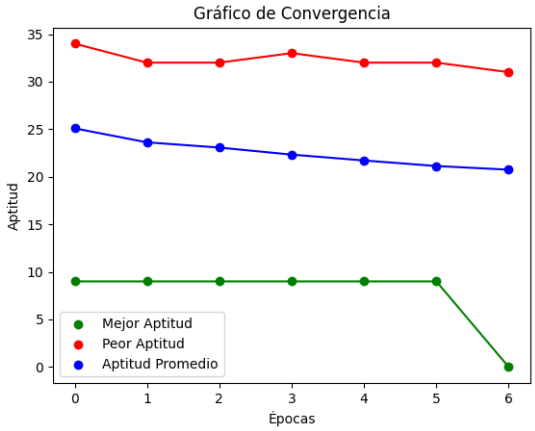

Figura 1: Gráfica de la aptitud promedio por generación, Semilla 1

2	7	6
9	5	1
4	3	8

Cuadro 2: Cuadrado mágico de 3x3, Semilla 1

En la segunda ejecución usando la semilla 3, tardando 37 generaciones, dio como resultado lo siguiente:

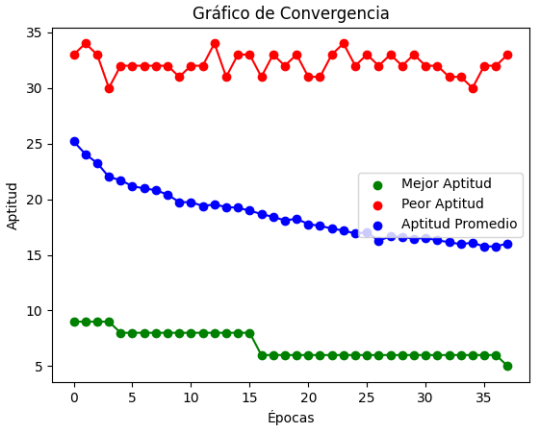

Figura 2: Gráfica de la aptitud promedio por generación, Semilla 3

8	3	4
2	5	9
6	7	1

Cuadro 3: Cuadrado mágico de 3x3, Semilla 3

En la tercera ejecución usando la semilla 5, tardando 73 generaciones, dio como resultado lo siguiente:

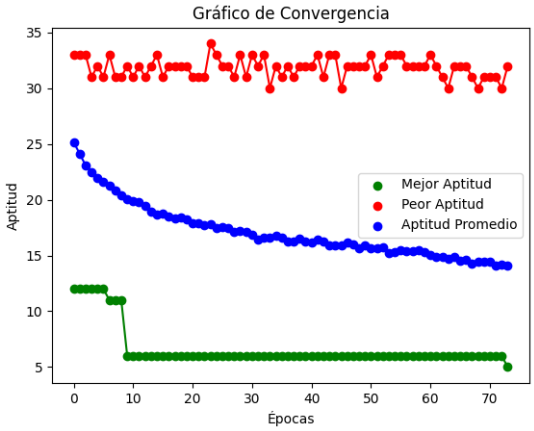

Figura 3: Gráfica de la aptitud promedio por generación, Semilla 5

4	3	9
8	5	1
2	7	6

Cuadro 4: Cuadrado mágico de 3x3, Semilla 5

En la cuarta ejecución usando la semilla 7, tardando 3 generaciones, dio como resultado lo siguiente:

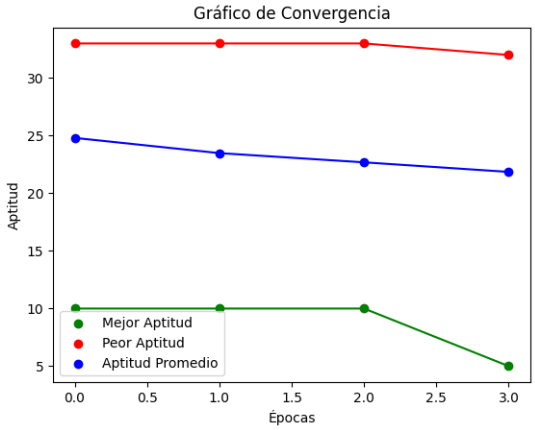

Figura 4: Gráfica de la aptitud promedio por generación, Semilla 7

6	8	2
1	5	9
7	3	4

Cuadro 5: Cuadrado mágico de 3x3, Semilla 7

En la quinta ejecución usando la semilla 11, tardando 21 generaciones, dio como resultado lo siguiente:

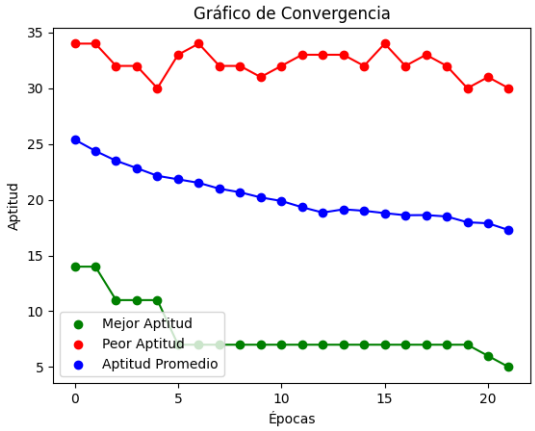

Figura 5: Gráfica de la aptitud promedio por generación, Semilla 11

4	9	3
2	5	7
8	1	6

Cuadro 6: Cuadrado mágico de 3x3, Semilla 11

En la sexta ejecución usando la semilla 13, tardando 23 generaciones, dio como resultado lo siguiente:

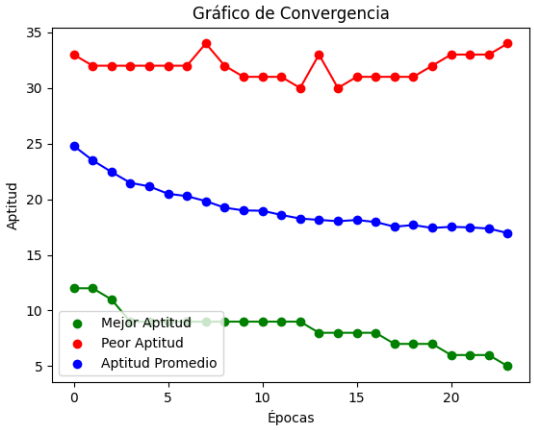

Figura 6: Gráfica de la aptitud promedio por generación, Semilla 13

4	3	7
9	5	1
2	8	6

Cuadro 7: Cuadrado mágico de 3x3, Semilla 13

En la séptima ejecución usando la semilla 19, tardando 48 generaciones, dio como resultado lo siguiente:

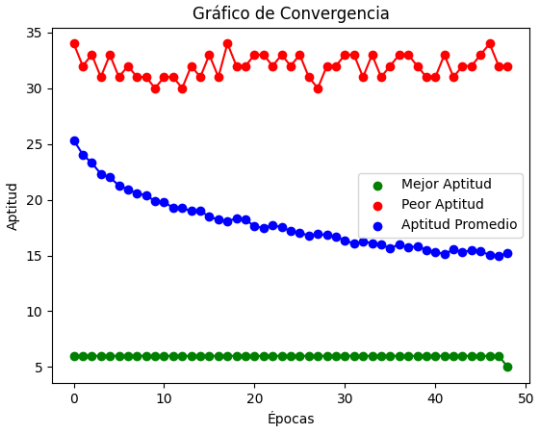

Figura 7: Gráfica de la aptitud promedio por generación, Semilla 19

8	3	5
1	4	9
6	7	2

Cuadro 8: Cuadrado mágico de 3x3, Semilla 19

En la octava ejecución usando la semilla 23, tardando 58 generaciones, dio como resultado lo siguiente:

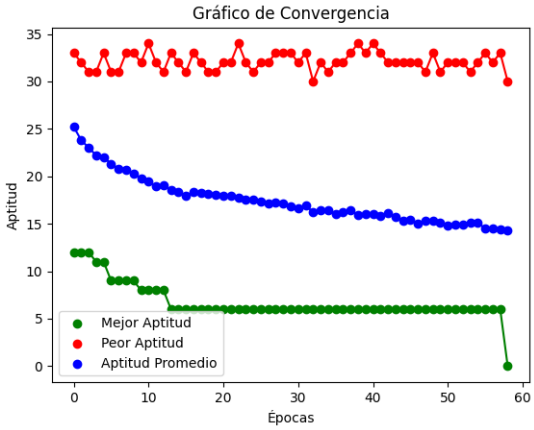

Figura 8: Gráfica de la aptitud promedio por generación, Semilla 23

4	9	2
3	5	7
8	1	6

Cuadro 9: Cuadrado mágico de 3x3, Semilla 23

3.2 Cuadrados mágicos de 4x4

En la primera ejecución usando la semilla 1, tardando 11,002 generaciones, dio como resultado lo siguiente:

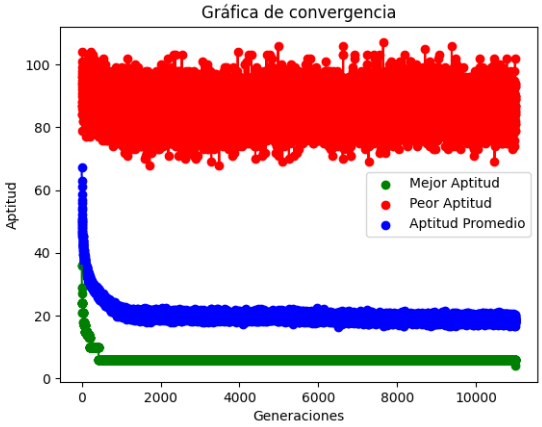

Figura 9: Gráfica de la aptitud promedio por generación, Semilla 1

10	1	14	9
11	4	3	16
5	15	12	2
7	13	6	8

Cuadro 10: Cuadrado mágico de 4x4, Semilla 1

En la segunda ejecución usando la semilla 3, tardando 563 generaciones, dio como resultado lo siguiente:

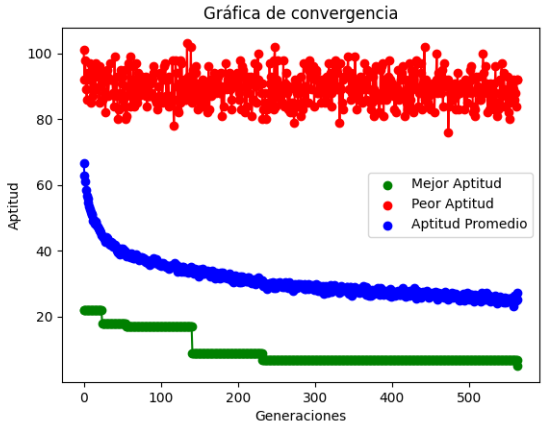

Figura 10: Gráfica de la aptitud promedio por generación, Semilla 3

2	3	15	14
12	16	4	1
9	5	7	13
11	10	8	6

Cuadro 11: Cuadrado mágico de 4x4, Semilla 3

En la tercera ejecución usando la semilla 5, tardando 805 generaciones, dio como resultado lo siguiente:

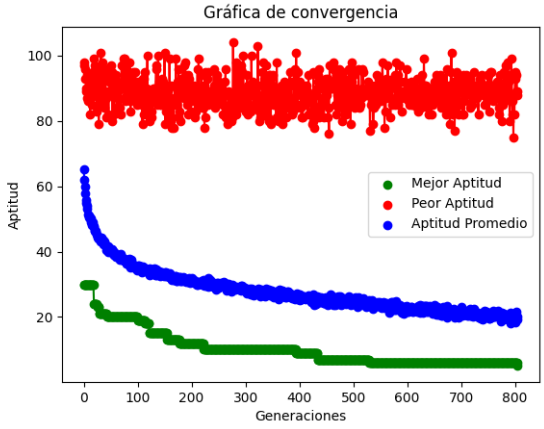

Figura 11: Gráfica de la aptitud promedio por generación, Semilla 5

2	3	15	14
12	16	4	1
9	5	7	13
11	10	8	6

Cuadro 12: Cuadrado mágico de 4x4, Semilla 5

En la cuarta ejecución usando la semilla 7, tardando 823 generaciones, dio como resultado lo siguiente:

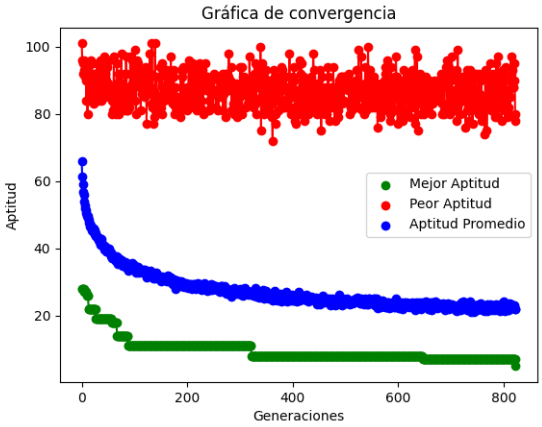

Figura 12: Gráfica de la aptitud promedio por generación, Semilla 7

5	8	10	11
13	4	14	2
12	6	9	7
3	16	1	15

Cuadro 13: Cuadrado mágico de 4x4, Semilla 7

En la quinta ejecución usando la semilla 11, tardando 342 generaciones, dio como resultado lo siguiente:

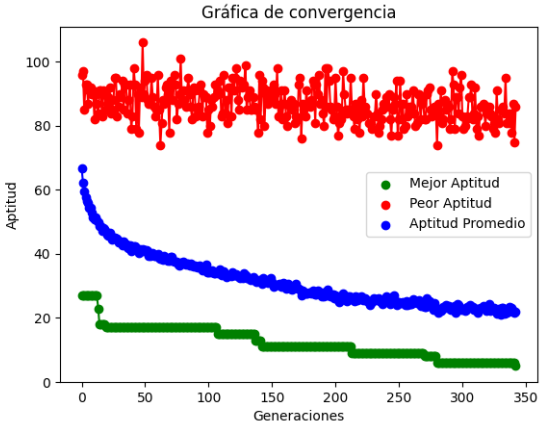

Figura 13: Gráfica de la aptitud promedio por generación, Semilla 11

8	9	4	13
15	14	4	1
16	1	5	12
6	11	10	7

Cuadro 14: Cuadrado mágico de 4x4, Semilla 11

En la sexta ejecución usando la semilla 13, tardando 520 generaciones, dio como resultado lo siguiente:

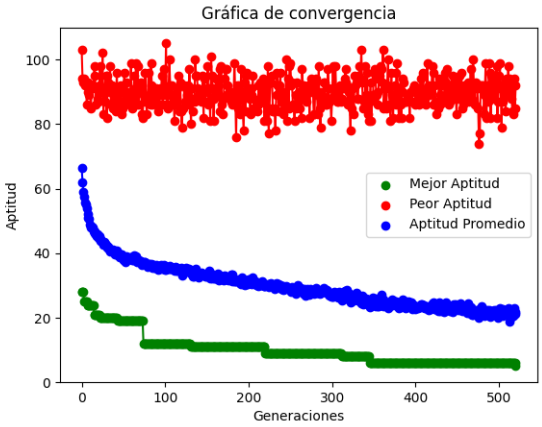

Figura 14: Gráfica de la aptitud promedio por generación, Semilla 13

9	12	5	8
11	7	10	6
2	1	15	16
14	13	4	3

Cuadro 15: Cuadrado mágico de 4x4, Semilla 13

En la séptima ejecución usando la semilla 19, tardando 1609 generaciones, dio como resultado lo siguiente:

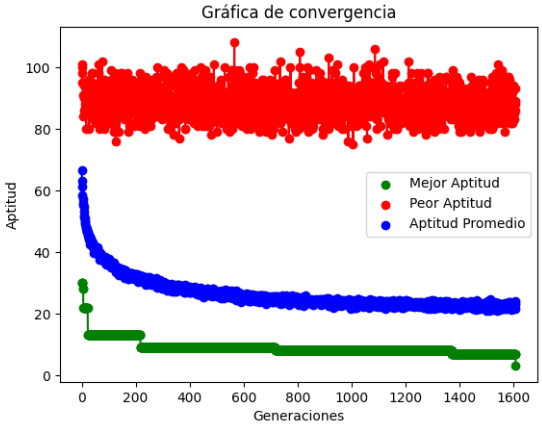

Figura 15: Gráfica de la aptitud promedio por generación, Semilla 19

16	14	1	2
4	3	13	15
5	11	8	10
9	6	12	7

Cuadro 16: Cuadrado mágico de 4x4, Semilla 19

En la octava ejecución usando la semilla 23, tardando 2505 generaciones, dio como resultado lo siguiente:

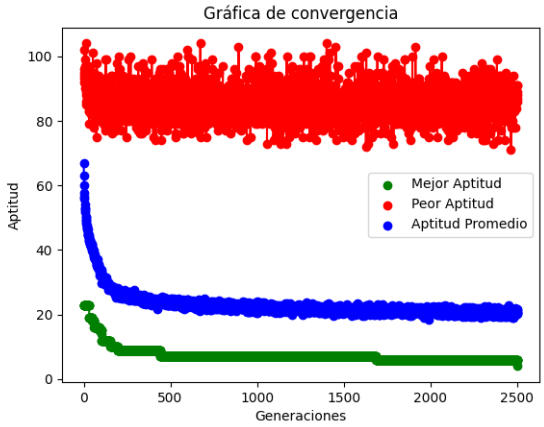

Figura 16: Gráfica de la aptitud promedio por generación, Semilla 23

4	13	9	8
2	14	15	3
16	1	5	12
10	7	6	11

Cuadro 17: Cuadrado mágico de 4x4, Semilla 23

4. Discusión de Resultados

• ¿Cuál fue la función objetivo que encontró más rápido el cuadrado mágico? La función que efectivamente puede encontrar más rápido un cuadrado mágico es la parte de la función *verificarexito* que busca una aptitud de 0.0 Esto se debe a que esta condición identifica directamente un cuadrado mágico perfecto sin errores, y la detección es inmediata tan pronto como se encuentre un individuo con esta aptitud en la población. La evaluación se realiza en cada generación, y tan pronto como se encuentra una aptitud de 0.0, el algoritmo puede terminar, lo que es la manera más rápida de concluir la búsqueda bajo las condiciones ideales.

La búsqueda de un error mínimo no garantiza un cuadrado perfecto y, además, puede continuar evaluando más individuos incluso después de encontrar cuadrados casi perfectos, lo cual es menos eficiente si el objetivo es encontrar cuadrados mágicos absolutamente precisos.

• ¿Cómo modificaría el algoritmo para encontrar todos los posibles cuadrados mágicos? (recuerde que existe más de una solución) Diseñar operaciones de cruce que preserven características clave de los cuadrados mágicos, como las sumas de filas, columnas y diagonales. Esto puede ayudar a mantener la viabilidad de las soluciones a través de generaciones. Ademar de conservar siempre una copia de los mejores individuos encontrados hasta ahora para asegurarte de que las buenas soluciones no se pierdan a lo largo de las generaciones.

• ¿Modificó los parámetros de su algoritmo para los diferentes tamaños del cuadrado mágico(n= 4yn= 5)? Si su respuesta es afirmativa ¿cuáles fueron los parámetros? Si, se utilizó parámetros para n = 3 y 4, ya que el tiempo de ejecución con otros parámetros era bastante e incluso llegaba a romperse el programa.

Aldo Escamilla: El algoritmo para esta práctica fue diseñado primeramente en inicializar una población, esto quiere decir que, el algoritmo inicia con posibles cuadrados mágicos, donde cada individuo en la población es una permutación aleatoria, después se diseño la funcion de aptitud que para cada cuadrado en la población, se calcula una función de aptitud que mide qué tan cerca está el cuadrado de ser un cuadrado mágico. Esta aptitud se calcula como la suma de las diferencias absolutas entre el número mágico y las sumas de filas, columnas y diagonales del cuadrado, después se hace la elección y la reproducción que la cruce, para después la mutación, con esto, el algoritmo revisa si algún individuo tiene una aptitud de 0.0 o con la otra función se ejecuta hasta que encuentra una solución perfecta.

Leobardo Yañez: En esta practica pude aprender a utilizar algoritmos genéticos para resolver problemas de optimización, en este caso el problema de encontrar cuadrados mágicos. Fue todo un reto ya que tuvimos que im-

plementar funciones para calcular la aptitud de cada individuo, funciones para seleccionar y cruzar a los individuos, y funciones para mutar a los individuos. Fue gratificante al final ya que pude ver como el algoritmo encontraba cuadrados mágicos.