

Practica 2. Algoritmo genético para codificación de permutaciones.

Algoritmos Bioinspirados

Diego Castillo Reyes
Marthon Leobardo Yañez Martinez
Aldo Escamilla Resendiz

12 de abril de 2024

Índice general

1.	Introducción	2
2.	Desarrollo	2
3.	Resultados	9
	3.1. Cuadrados mágicos de 3x3	9
	3.2. Cuadrados mágicos de 4x4	13
4.	Discusión de Resultados	17

1. Introducción

En esta práctica se implementó un algoritmo genético para resolver el problema de codificación de permutaciones. En específico 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 $n \times n$.

2. Desarrollo

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

```
1 import math
2 import random
3 import copy
4 import sys
5 import time
6 import matplotlib.pyplot as plt
7 import statistics
8
9 random.seed(23)
10
11 n = 3
12 nn = n * n
13 n2 = nn // 2
14 magic_number = n * (n ** 2 + 1) / 2
15 tamaño_poblacion = 200
16 num_elites = 150
17 num_mutaciones = 0.9
18 probabilidad_mutacion = 1
19 epocas = 1000000
20 intentos_cruce = 20
21 tenencia_tabu = 10
22
23 poblacion = []
24 aptitud = []
25 lista_tabu = []
26
27 def generar_individuos():
28     poblacion_generada = []
29     for _ in range(tamaño_poblacion):
30         cuadrado = list(range(1, nn + 1))
31         random.shuffle(cuadrado)
32         poblacion_generada.append(cuadrado)
33     return poblacion_generada
34
35 def calcular_aptitud_poblacional(poblacion_actual):
36     aptitudes = []
37     for individuo in poblacion_actual:
38         aptitudes.append(calcular_aptitud_cuadrado(individuo))
39     return aptitudes
```

```

40
41 def calcular_apptitud_cuadrado(cuadrado):
42     aptitud = 0
43     suma = 0
44     for i in range(nn):
45         if (i % n == 0 and i != 0):
46             aptitud += abs(magic_number - suma)
47             suma = 0
48             suma += cuadrado[i]
49         aptitud += abs(magic_number - suma)
50
51     for j in range(n):
52         suma = 0
53         for i in range(j, nn, n):
54             suma += cuadrado[i]
55         aptitud += abs(magic_number - suma)
56
57     suma = 0
58     for i in range(0, nn, n + 1):
59         suma += cuadrado[i]
60     aptitud += abs(magic_number - suma)
61
62     suma = 0
63     for i in range(n - 1, nn - 1, n - 1):
64         suma += cuadrado[i]
65     aptitud += abs(magic_number - suma)
66
67     return aptitud
68
69 def verificar_exito(apptitudes):
70     if 0.0 in apptitudes:
71         return apptitudes.index(0.0), 'exito'
72
73     umbral_error_minimo = 5
74     for indice, valor_apptitud in enumerate(apptitudes):
75         if valor_apptitud <= umbral_error_minimo:
76             return indice, 'error_minimo'
77
78     return -1, 'no_encontrado'
79
80 def es_cuadrado_valido(cuadrado):
81     numeros = set(range(1, nn + 1))
82     cuadrado_posible = set(cuadrado)
83
84     if numeros == cuadrado_posible:
85         return True
86     return False
87
88 def obtener_elites(poblacion_actual, apptitudes):
89     combinados = list(zip(poblacion_actual, apptitudes))
90     combinados = sorted(combinados, key=lambda x: x[1])
91     elites, _ = zip(*combinados)
92     mejor_apptitud = apptitudes[0]
93     peor_apptitud = apptitudes[-1]
94     favorito = sum(apptitudes) / len(apptitudes)
95     desviacion_estandar = statistics.stdev(apptitudes)
96     print(f"Mejor Aptitud: {mejor_apptitud}")
97     print(f'Peor Aptitud: {peor_apptitud}')
98     print(f'Aptitud Promedio: {favorito}')
99     print(f'Desviacion Estandar: {desviacion_estandar}')
100    print(f'Élite Top: {elites[0]}')
101    return list(elites)
102
103 def mutar(cuadrado):
104     numero_genes = len(cuadrado)
105     numero_mutaciones = random.randint(1, numero_genes)
106
107     for _ in range(numero_mutaciones):
108         i = random.randint(0, numero_genes - 1)
109         j = random.randint(0, numero_genes - 1)
110         cuadrado[i], cuadrado[j] = cuadrado[j], cuadrado[i]

```

```

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

```

```

181
182     hijo1 = [None] * nn
183     hijo2 = [None] * nn
184
185     hijo1[index1:index2 + 1] = padre2[index1:index2 + 1]
186     hijo2[index1:index2 + 1] = padre1[index1:index2 + 1]
187
188     def completar_hijo(hijo, padre):
189         pos_actual = (index2 + 1) % nn
190         for gen in padre:
191             if gen not in hijo:
192                 hijo[pos_actual] = gen
193                 pos_actual = (pos_actual + 1) % nn
194
195     completar_hijo(hijo1, padre1)
196     completar_hijo(hijo2, padre2)
197
198     return hijo1, hijo2
199
200 poblacion = generar_individuos()
201 historial = {'mejor': [], 'peor': [], 'promedio': [], 'desviacion': []}
202
203 for i in range(epocas):
204     print(f"Generación actual: {i}")
205     aptitud = calcular_aptitud_poblacional(poblacion)
206
207     historial['mejor'].append(min(aptitud))
208     historial['peor'].append(max(aptitud))
209     historial['promedio'].append(sum(aptitud) / len(aptitud))
210
211     if i % 500 == 0 and i != 0:
212         for j in range(len(poblacion)):
213             print(f'C cuadrado: {poblacion[j]} Aptitud: {aptitud[j]}')
214             time.sleep(3)
215
216     indice, estado = verificar_exito(aptitud)
217     if estado == 'exito':
218         print(f"C cuadrado Mágico Encontrado: {poblacion[indice]}")
219         break
220     elif estado == 'error_minimo':
221         print(f"C cuadrado Mágico Encontrado: {poblacion[indice]} con Aptitud {aptitud[indice]}")
222         break
223
224     poblacion = reproducir(poblacion, aptitud)
225
226 generaciones = list(range(len(historial['mejor'])))
227
228 plt.scatter(generaciones, historial['mejor'], color='green', label='Mejor Aptitud')
229 plt.plot(generaciones, historial['mejor'], color='green')
230
231 plt.scatter(generaciones, historial['peor'], color='red', label='Peor Aptitud')
232 plt.plot(generaciones, historial['peor'], color='red')
233
234 plt.scatter(generaciones, historial['promedio'], color='blue', label='Aptitud Promedio')
235 plt.plot(generaciones, historial['promedio'], color='blue')
236
237 plt.legend()
238 plt.xlabel('Épocas')
239 plt.ylabel('Aptitud')
240 plt.title("Gráfico de Convergencia")
241 plt.show()
242
243
244
245 random.seed(23)
246 n = 4
247 nn = n * n
248 n2 = nn // 2
249 magic_number = n * (n ** 2 + 1) / 2
250 population_size = 200
251 num_elites = 150

```

```

252 num_mutations = 0.9
253 mutation_chance = 1
254 epoch = 1000000
255 crossover_attempts = 20
256 tabu_tenure = 10
257
258 poblacion = []
259 aptitud = []
260 tabu_list = []
261
262 def generar_individuos():
263     pop = []
264     for i in range(population_size):
265         c = list(range(1, nn + 1))
266         random.shuffle(c)
267         pop.append(c)
268     return pop
269
270 def obtener_aptitud_poblacional(pop):
271     fit = []
272     for c in pop:
273         fit.append(encontrar_aptitud2(c))
274     return fit
275
276 def encontrar_aptitud2(criatura):
277     fit = 0
278     suma = 0
279     for i in range(nn):
280         if (i % n == 0 and i != 0):
281             fit += abs(magic_number - suma)
282             suma = 0
283             suma += criatura[i]
284         fit += abs(magic_number - suma)
285
286     for j in range(n):
287         suma = 0
288         for i in range(j, nn, n):
289             suma += criatura[i]
290         fit += abs(magic_number - suma)
291
292     suma = 0
293     for i in range(0, nn, n + 1):
294         suma += criatura[i]
295     fit += abs(magic_number - suma)
296
297     suma = 0
298     for i in range(n - 1, nn - 1, n - 1):
299         suma += criatura[i]
300     fit += abs(magic_number - suma)
301
302     return fit
303
304 def ganamos(apt):
305     if 0.0 in apt:
306         return apt.index(0.0), 'exito'
307
308     umbral_error_minimo = 5
309     for indice, valor_aptitud in enumerate(apt):
310         if valor_aptitud <= umbral_error_minimo:
311             return indice, 'error_minimo'
312
313     return -1, 'no_encontrado'
314
315
316 def es_cuadrado_valido(criatura):
317     numeros = set(range(1, nn + 1))
318     cuadrado_posible = set(criatura)
319
320     if numeros == cuadrado_posible:
321         return True
322     return False

```

```

323
324
325 def obtener_elites(pop, fit):
326     combinados = list(zip(pop, fit))
327     combinados = sorted(combinados, key=lambda x: x[1])
328     elites, trash = zip(*combinados)
329     mejor_apitud = trash[0]
330     peor_apitud = trash[-1]
331     favorito = sum(trash) / len(trash)
332     desviacion = statistics.stdev(apitud)
333     print(f"Mejor Aptitud: {mejor_apitud}")
334     print(f'Peor Aptitud: {peor_apitud}')
335     print(f'Aptitud Promedio: {favorito}')
336     print(f'Desviacion Estandar: {desviacion}')
337     print(f'Elite Top: {elites[0]}')
338     return list(elites)
339
340 def mutar(criatura):
341     nn = len(criatura)
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         criatura[i], criatura[j] = criatura[j], criatura[i]
348
349     return criatura
350
351 def reproducir(pop, fit):
352     popc = copy.deepcopy(pop)
353     pool_apareamiento = []
354     siguiente_gen = []
355
356     numero_restante = population_size - num_elites
357     elites = obtener_elites(pop, fit)
358     unicos_elites = []
359     for e in elites:
360         if not e in unicos_elites:
361             unicos_elites.append(e)
362
363     siguiente_gen = unicos_elites[0:num_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     pool_apareamiento.append(bigfit[0] / fitsum)
374     for i in range(1, len(bigfit)):
375         prob = bigfit[i] / fitsum
376         pool_apareamiento.append(prob + pool_apareamiento[i - 1])
377
378     while numero_restante > 0:
379         eleccion1 = random.random()
380         eleccion2 = random.random()
381         indice1 = 0
382         indice2 = 0
383         for i in range(len(pool_apareamiento)):
384             if eleccion1 <= pool_apareamiento[i]:
385                 indice1 = i
386                 break
387         for i in range(len(pool_apareamiento)):
388             if eleccion2 <= pool_apareamiento[i]:
389                 indice2 = i
390                 break
391
392     candidato1 = copy.copy(popc[indice1])
393     candidato2 = copy.copy(popc[indice2])

```



```

394     bebe1, bebe2 = cruce(candidato1, candidato2)
395
396     if (not bebe1 in siguiente_gen) and (not bebe1 in tabu_list):
397         siguiente_gen.append(bebe1)
398         numero_restante -= 1
399         if len(tabu_list) >= tabu_tenure:
400             tabu_list.pop(0)
401             tabu_list.append(bebe1)
402
403     if ((not bebe2 in siguiente_gen) and (not bebe2 in tabu_list) and numero_restante > 0):
404         siguiente_gen.append(bebe2)
405         numero_restante -= 1
406         if len(tabu_list) >= tabu_tenure:
407             tabu_list.pop(0)
408             tabu_list.append(bebe2)
409
410     print()
411     return siguiente_gen
412
413 def cruce(p1, p2):
414     nn = len(p1)
415     index1 = random.randint(0, nn - 1)
416     index2 = random.randint(0, nn - 1)
417     if index1 > index2:
418         index1, index2 = index2, index1
419
420     c1 = [None] * nn
421     c2 = [None] * nn
422
423     c1[index1:index2 + 1] = p2[index1:index2 + 1]
424     c2[index1:index2 + 1] = p1[index1:index2 + 1]
425
426     def complete_child(child, parent):
427         current_pos = (index2 + 1) % nn
428         for gene in parent:
429             if gene not in child:
430                 child[current_pos] = gene
431                 current_pos = (current_pos + 1) % nn
432
433     complete_child(c1, p1)
434     complete_child(c2, p2)
435
436     return c1, c2
437
438 poblacion = generar_individuos()
439 historial_aptitud = {'mejor': [], 'peor': [], 'promedio': [], 'desviacion': []}
440
441 for i in range(epoch):
442     print(f"Generación actual: {i}")
443     aptitud = obtener_aptitud_poblacional(poblacion)
444
445     historial_aptitud['mejor'].append(min(aptitud))
446     historial_aptitud['peor'].append(max(aptitud))
447     historial_aptitud['promedio'].append(sum(aptitud) / len(aptitud))
448
449
450     if i % 500 == 0 and i != 0:
451         for i in range(len(poblacion)):
452             print(f'Cuadrado: {poblacion[i]} Aptitud: {aptitud[i]}')
453             time.sleep(3)
454
455     indice, estado = ganamos(aptitud)
456     if estado == 'exito':
457         print(f"Cuadrado Mágico Encontrado: {poblacion[indice]}")
458         break
459     elif estado == 'error_minimo':
460         print(f"Cuadrado Mágico Encontrado: {poblacion[indice]} con Aptitud {aptitud[indice]}")
461         break
462
463     poblacion = reproducir(poblacion, aptitud)
464

```

```

465 generaciones = list(range(len(historial_aptitud['mejor'])))
466
467 plt.scatter(generaciones, historial_aptitud['mejor'], color='green', label='Mejor Aptitud')
468 plt.plot(generaciones, historial_aptitud['mejor'], color='green')
469
470 plt.scatter(generaciones, historial_aptitud['peor'], color='red', label='Peor Aptitud')
471 plt.plot(generaciones, historial_aptitud['peor'], color='red')
472
473 plt.scatter(generaciones, historial_aptitud['promedio'], color='blue', label='Aptitud Promedio')
474 plt.plot(generaciones, historial_aptitud['promedio'], color='blue')
475
476 plt.legend()
477 plt.xlabel('Generaciones')
478 plt.ylabel('Aptitud')
479 plt.title("Gráfica de convergencia")
480 plt.show()
481

```

3. Resultados

3.1 Cuadrados mágicos de 3x3

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

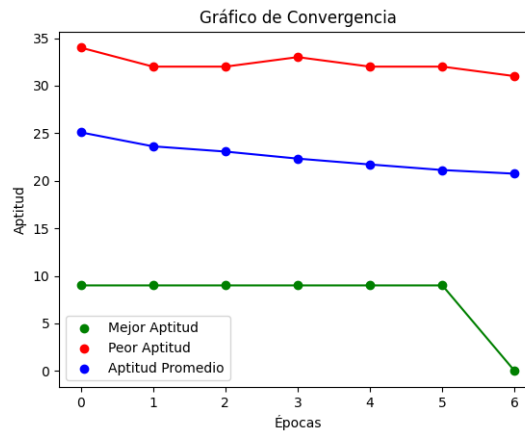


Figura 1: Gráfica de la aptitud promedio por generación, Semilla 1

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:

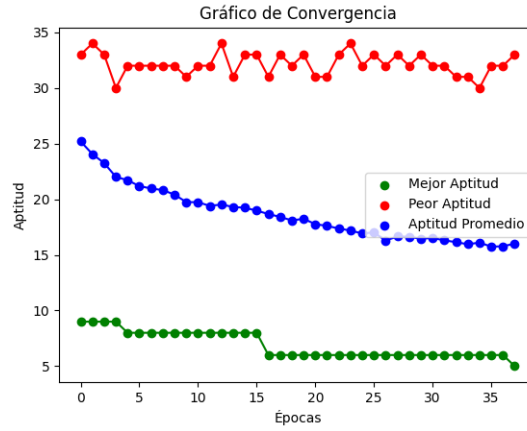


Figura 2: Gráfica de la aptitud promedio por generación, Semilla 3

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:

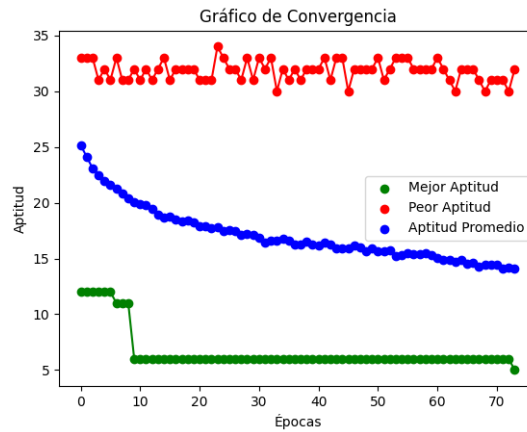


Figura 3: Gráfica de la aptitud promedio por generación, Semilla 5

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:

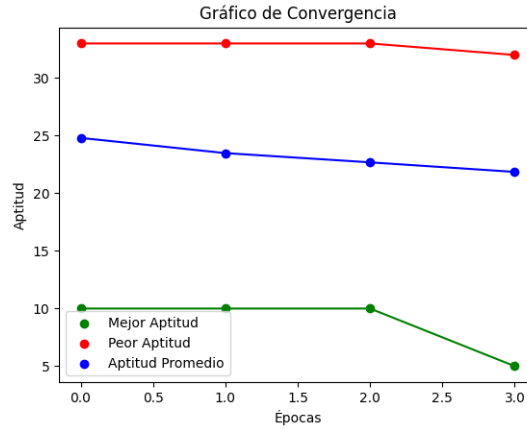


Figura 4: Gráfica de la aptitud promedio por generación, Semilla 7

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:

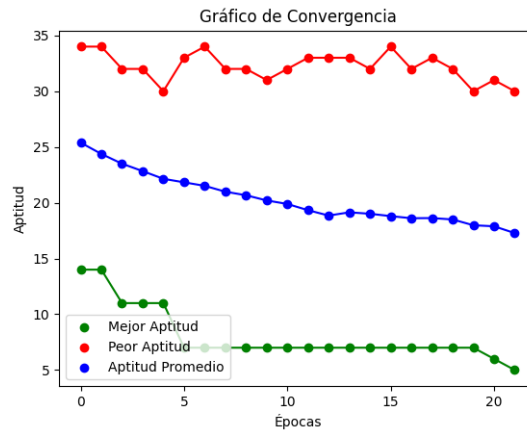


Figura 5: Gráfica de la aptitud promedio por generación, Semilla 11

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:

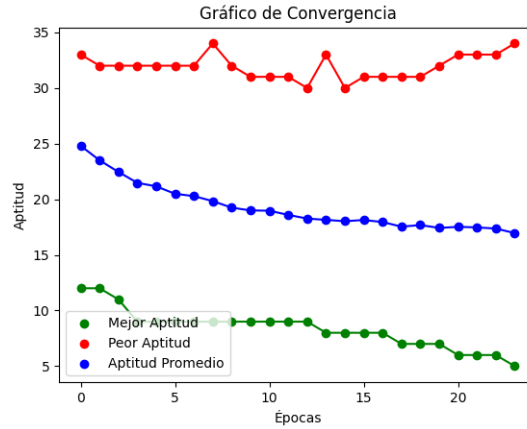


Figura 6: Gráfica de la aptitud promedio por generación, Semilla 13

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:

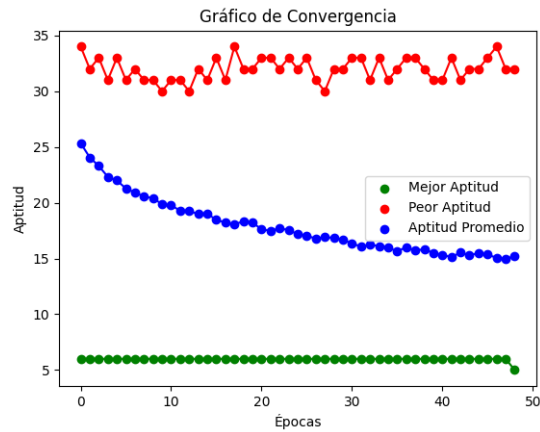


Figura 7: Gráfica de la aptitud promedio por generación, Semilla 19

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:

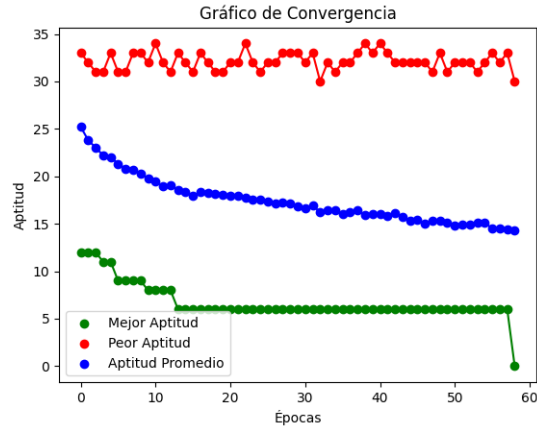


Figura 8: Gráfica de la aptitud promedio por generación, Semilla 23

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:

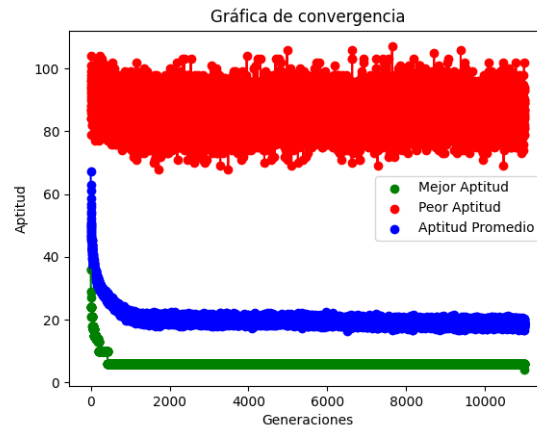


Figura 9: Gráfica de la aptitud promedio por generación, Semilla 1

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:

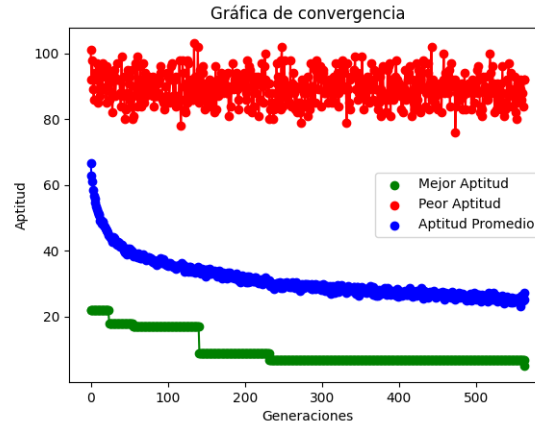


Figura 10: Gráfica de la aptitud promedio por generación, Semilla 3

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:

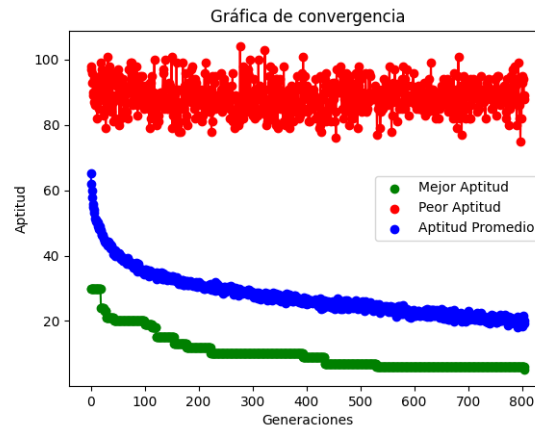


Figura 11: Gráfica de la aptitud promedio por generación, Semilla 5

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:

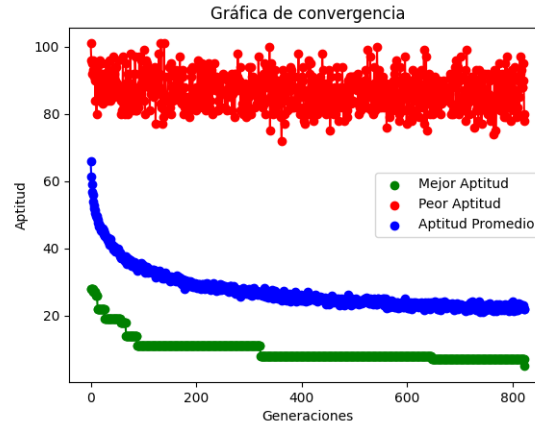


Figura 12: Gráfica de la aptitud promedio por generación, Semilla 7

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:

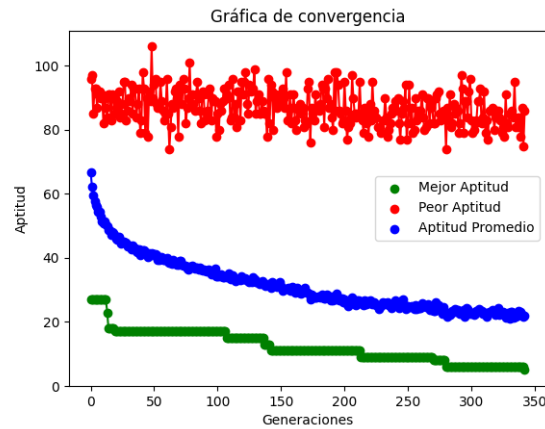


Figura 13: Gráfica de la aptitud promedio por generación, Semilla 11

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:

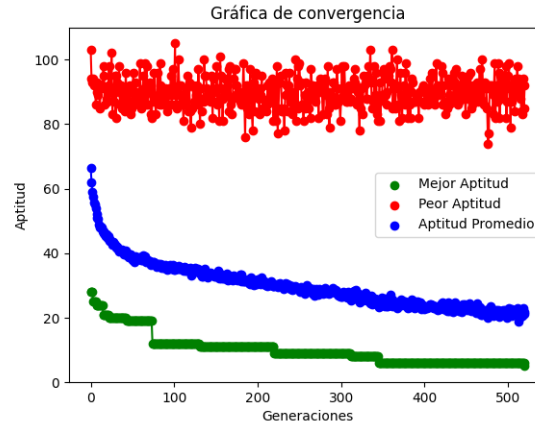


Figura 14: Gráfica de la aptitud promedio por generación, Semilla 13

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:

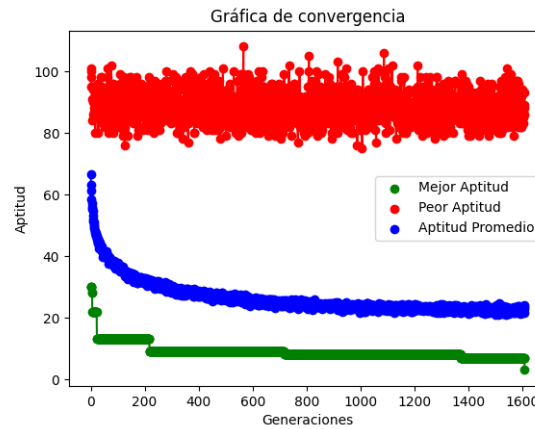


Figura 15: Gráfica de la aptitud promedio por generación, Semilla 19

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:

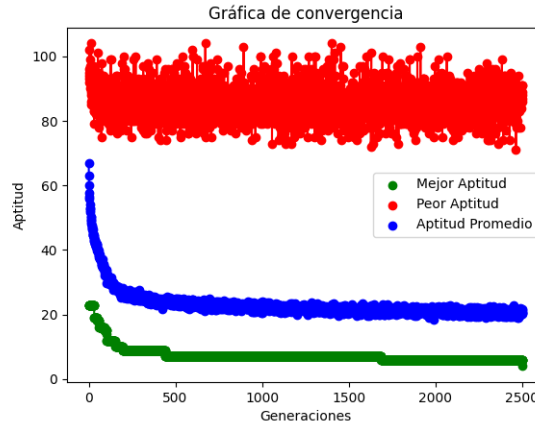


Figura 16: Gráfica de la aptitud promedio por generación, Semilla 23

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. Además 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 ($n=4$ y $n=5$)? 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 diseña la función 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 práctica 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.