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

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
magic_number = n * (n ** 2 + 1) / 2
tamano_poblacion = 200
num_elites = 150
num_mutaciones = 0.9
probabilidad_mutacion = 1
epocas = 1000000
intentos_cruce = 20
tenencia_tabu = 10

poblacion = []
aptitud = []
lista_tabu = []

def generar_individuos():
    poblacion_generada = []
    for _ in range(tamano_poblacion):
        cuadrado = list(range(1, nn + 1))
        random.shuffle(cuadrado)
        poblacion_generada.append(cuadrado)
    return poblacion_generada

def calcular_aptitud_poblacional(poblacion_actual):
    aptitudes = []
    for individuo in poblacion_actual:
        aptitudes.append(calcular_aptitud_cuadrado(individuo))
    return aptitudes
```

```
40
41  def calcular_aptitud_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(aptitudes):
70      if 0.0 in aptitudes:
71          return aptitudes.index(0.0), 'exito'
72
73      umbral_error_minimo = 5
74      for indice, valor_aptitud in enumerate(aptitudes):
75          if valor_aptitud <= 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, aptitudes):
89      combinados = list(zip(poblacion_actual, aptitudes))
90      combinados = sorted(combinados, key=lambda x: x[1])
91      elites, _ = zip(*combinados)
92      mejor_aptitud = aptitudes[0]
93      peor_aptitud = aptitudes[-1]
94      favorito = sum(aptitudes) / len(aptitudes)
95      desviacion_estandar = statistics.stdev(aptitudes)
96      print(f"Mejor Aptitud: {mejor_aptitud}")
97      print(f'Peor Aptitud: {peor_aptitud}')
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
    individuos_restantes > 0):
167             siguiente_generacion.append(hijo2)
168             individuos_restantes -= 1
169             if len(lista_tabu) >= tenencia_tabu:
170                 lista_tabu.pop(0)
171             lista_tabu.append(hijo2)
172
173     print()
174     return siguiente_generacion
175
176 def cruce(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
```

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

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

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

    completar_hijo(hijo1, padre1)
    completar_hijo(hijo2, padre2)

    return hijo1, hijo2

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

for i in range(epocas):
    print(f"Generación actual: {i}")
    aptitud = calcular_aptitud_poblacional(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
magic_number = n * (n ** 2 + 1) / 2
population_size = 200
num_elites = 150
```

```
num_mutations = 0.9
mutation_chance = 1
epoch = 1000000
crossover_attempts = 20
tabu_tenure = 10

poblacion = []
aptitud = []
tabu_list = []

def generar_individuos():
    pop = []
    for i in range(population_size):
        c = list(range(1, nn + 1))
        random.shuffle(c)
        pop.append(c)
    return pop

def obtener_aptitud_poblacional(pop):
    fit = []
    for c in pop:
        fit.append(encontrar_aptitud2(c))
    return fit

def encontrar_aptitud2(criatura):
    fit = 0
    suma = 0
    for i in range(nn):
        if (i % n == 0 and i != 0):
            fit += abs(magic_number - suma)
            suma = 0
        suma += criatura[i]
    fit += abs(magic_number - suma)

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

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

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

    return fit

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

    if numeros == cuadrado_posible:
        return True
    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_aptitud = trash[0]
330     peor_aptitud = trash[-1]
331     favorito = sum(trash) / len(trash)
332     desviacion = statistics.stdev(aptitud)
333     print(f"Mejor Aptitud: {mejor_aptitud}")
334     print(f'Peor Aptitud: {peor_aptitud}')
335     print(f'Aptitud Promedio: {favorito}')
336     print(f'Desviacion Estandar: {desviacion}')
337     print(f'Élite 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])
```

```
        bebe1, bebe2 = cruce(candidato1, candidato2)

        if (not bebe1 in siguiente_gen) and (not bebe1 in tabu_list):
            siguiente_gen.append(bebe1)
            numero_restante -= 1
            if len(tabu_list) >= tabu_tenure:
                tabu_list.pop(0)
            tabu_list.append(bebe1)

        if ((not bebe2 in siguiente_gen) and (not bebe2 in tabu_list) and numero_restante > 0):
            siguiente_gen.append(bebe2)
            numero_restante -= 1
            if len(tabu_list) >= tabu_tenure:
                tabu_list.pop(0)
            tabu_list.append(bebe2)

    print()
    return siguiente_gen

def cruce(p1, p2):
    nn = len(p1)
    index1 = random.randint(0, nn - 1)
    index2 = random.randint(0, nn - 1)
    if index1 > index2:
        index1, index2 = index2, index1

    c1 = [None] * nn
    c2 = [None] * nn

    c1[index1:index2 + 1] = p2[index1:index2 + 1]
    c2[index1:index2 + 1] = p1[index1:index2 + 1]

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

    complete_child(c1, p1)
    complete_child(c2, p2)

    return c1, c2

poblacion = generar_individuos()
historial_aptitud = {'mejor': [], 'peor': [], 'promedio': [], 'desviacion': []}

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

    historial_aptitud['mejor'].append(min(aptitud))
    historial_aptitud['peor'].append(max(aptitud))
    historial_aptitud['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 = ganamos(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)
```

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

3. Conclusión

• ¿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 parametros para n = 3 y 4, ya que el tiempo de ejecucion con otros parametros era bastante e incluso llegaba a romperse el programa.