Inteligencia Artificial 1

Universidad Nacional de Cuyo - Facultad de Ingeniería

Trabajo Práctico N°3

Trabajo Práctico N°3: Algoritmos Genéticos

Grupo 2: Avila J., Barrios F., Patricelli N.

Septiembre 2025

1 Temas Tratados en el Trabajo Práctico 3

- Estrategias de búsqueda local.
- Algoritmos Evolutivos.
- Problemas de Satisfacción de Restricciones.

2 Ejercicios Teóricos

2.1 ¿Qué mecanismo de detención presenta el algoritmo de Ascensión de Colinas? Describa el problema que puede presentar este mecanismo y cómo se llaman las áreas donde ocurren estos problemas.

El mecanismo "Ascensión de Colinas" o "Climbing Hills" utiliza una meta-heurística la cual consiste en avanzar continuamente en dirección del valor creciente en el espacio de estados. Es decir, el mecanismo se desplaza únicamente en dirección de "un mejor estado", siempre toma el camino que genere un mejor resultado instantáneo. Además, no mantiene un árbol de búsqueda, por lo que solo monitorea el estado actual y su valor de función objetivo.

El hecho de que solo avance hacia un mejor estado del problema tiene sus desventajas:

- Como solo "avanza hacia adelante" (solo busca estados mejores que el actual) podría culminar en un máximo local y quedarse allí, y no encontrar nunca el máximo global.
- Tiene dificultad para tratar a las crestas: puede pasar que la pendiente se aproxime demasiado a un pico y la búsqueda oscilará de un lado al otro, obteniendo un avance muy bajo o nulo.
- En las mesetas del problema el algoritmo podría quedarse dando vueltas indefinidamente sin saber hacia dónde avanzar.
- Incluso cuando encuentra un máximo, no hay forma de saber si existe uno mejor en otra parte del espacio.

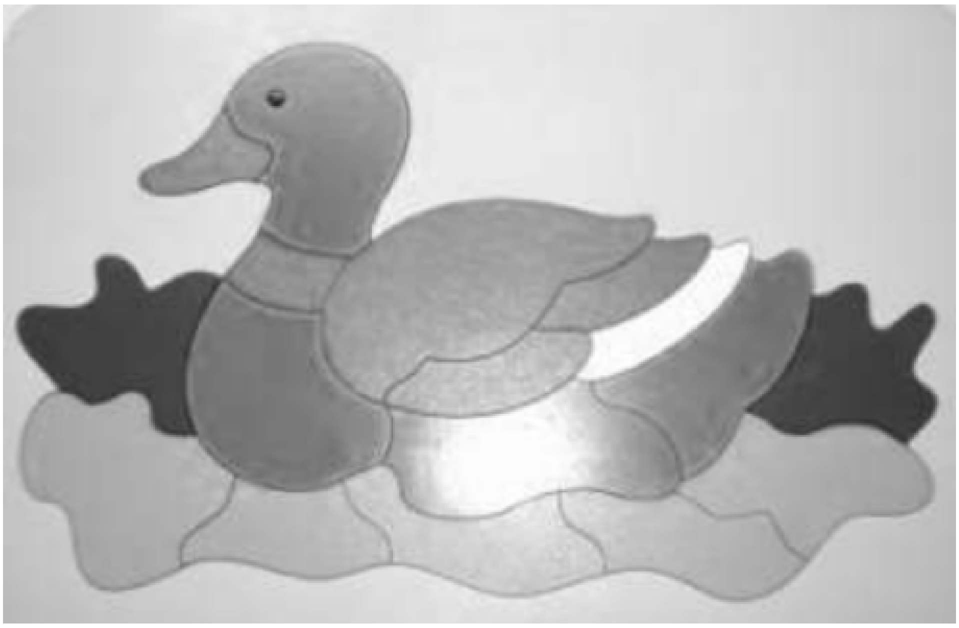
Para remarcarlo, el mecanismo presenta problemas en: máximos locales, mesetas (tanto terrazas como mesetas de máximos locales) y crestas muy empinadas.

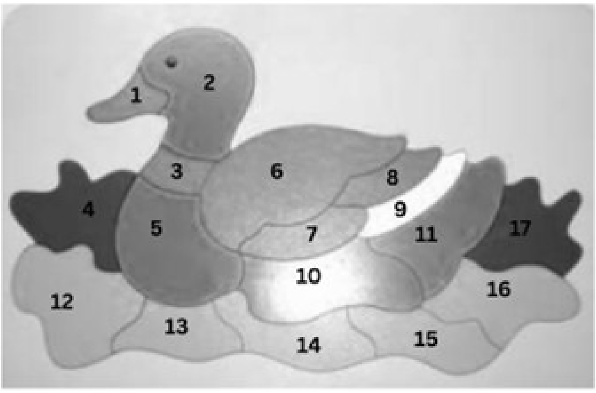
2.2 Describa las distintas heurísticas que se emplean en un problema de Satisfacción de Restricciones.

Las heurísticas utilizadas en un problema de satisfacción de Restricciones son eficaces y genéricas, ya que no requieren información específica adicional del dominio. las distintas heurísticas que se pueden utilizr son: - **Chequeo hacia adelante:** cada vez que se asigna un valor a una variable, se reduce el dominio de las variables vecinas. Pero si algún vecino se queda sin valores posibles, se debe retroceder. - **Heurística de grado máximo:** se toma como estado inicial aquel que tiene mayor n° de restricciones para evitar futuros conflictos (que se quede sin valores posibes). - **Heurística de mínimos valores restantes:** En un cierto estado, siempre elige la variable con menos valores legales, ya que si se cubre esa primero, el resto siempre tendrá algún valor legal restante para asignarle. - **Heurística del valor menos restringida:** en cada estado, siempre que se pueda elegir un valor que está repetido se lo elige, es decir, aquel que elimine menos posibilidades para las demás variables, de forma que queden valores restantes sin usar para las variables con mayores restricciones.

2.3 Se desea colorear el rompecabezas mostrado en la imagen con 7 colores distintos de manera que ninguna pieza tenga el mismo color que sus vecinas. Realice en una tabla el proceso de una búsqueda con Comprobación hacia Adelante empleando una heurística del Valor más Restringido.

```python
import requests
from PIL import Image
from io import BytesIO
import matplotlib.pyplot as plt

# URL directa de Google Drive
url = "https://drive.google.com/uc?export=view&id=1j94jFVxVG9y_ZnrMWOscQGb2MZOCdb3R"

# Descargar la imagen
response = requests.get(url)
img = Image.open(BytesIO(response.content))

# Mostrar la imagen
plt.imshow(img)
plt.axis('off')  # Ocultar ejes
plt.show()
```





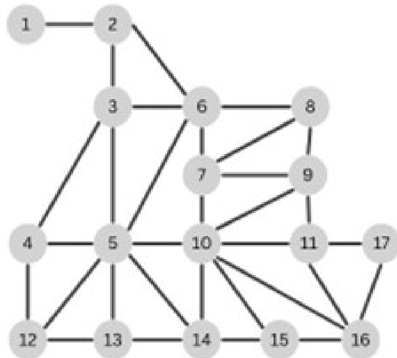


Tabla del proceso de búsqueda con comprobación hacia adelante, usando heurística del valor más restringido.

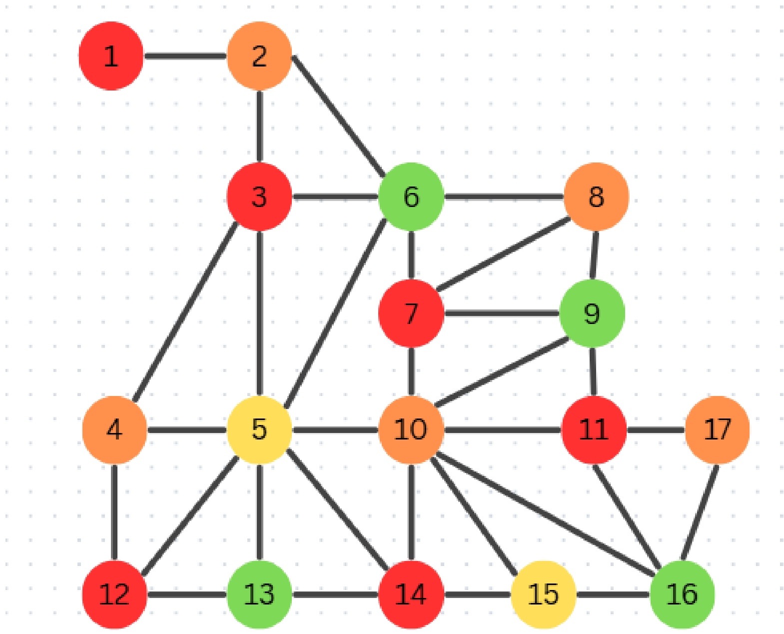
iteracion	1	2	3	4	5	6	7	8	9
0	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
1	R	N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
2		N	R-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
3			R	N-Am-Ve-Az-Vi-Ma	N-Am-Ve-Az-Vi-Ma	Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
4				N	Am-Ve-Az-Vi-Ma	Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
5					Am	Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
6						Ve	R-N-Am-Az-Vi-Ma	R-N-Am-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
7							R	N-Am-Az-Vi-Ma	N-Am-Ve-Az-Vi-Ma
8								N	Am-Ve-Az-Vi-Ma
9									Ve
10									
11									
12									
13									
14									
15									
16									
17									

10	11	12	13	14	15	16	17
R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
R-N-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-Ve-Az-Vi-Ma	R-N-Ve-Az-Vi-Ma	R-N-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
R-N-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-Ve-Az-Vi-Ma	R-N-Ve-Az-Vi-Ma	R-N-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
N-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-Ve-Az-Vi-Ma	R-N-Ve-Az-Vi-Ma	R-N-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
N-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-Ve-Az-Vi-Ma	R-N-Ve-Az-Vi-Ma	R-N-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
N-Az-Vi-Ma	R-N-Am-Az-Vi-Ma	R-Ve-Az-Vi-Ma	R-N-Ve-Az-Vi-Ma	R-N-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
N	R-Am-Az-Vi-Ma	R-Ve-Az-Vi-Ma	R-N-Ve-Az-Vi-Ma	R-Ve-Az-Vi-Ma	R-Am-Ve-Az-Vi-Ma	R-Am-Ve-Az-Vi-Ma	R-N-Am-Ve-Az-Vi-Ma
	R	R-Ve-Az-Vi-Ma	R-N-Ve-Az-Vi-Ma	R-Ve-Az-Vi-Ma	R-Am-Ve-Az-Vi-Ma	Am-Ve-Az-Vi-Ma	N-Am-Ve-Az-Vi-Ma
		R	N-Ve-Az-Vi-Ma	R-Ve-Az-Vi-Ma	R-Am-Ve-Az-Vi-Ma	Am-Ve-Az-Vi-Ma	N-Am-Ve-Az-Vi-Ma
			Ve	R-Az-Vi-Ma	R-Am-Ve-Az-Vi-Ma	Am-Ve-Az-Vi-Ma	N-Am-Ve-Az-Vi-Ma
				R	Am-Ve-Az-Vi-Ma	Am-Ve-Az-Vi-Ma	N-Am-Ve-Az-Vi-Ma
					Am	Ve-Az-Vi-Ma	N-Am-Ve-Az-Vi-Ma
						Ve	N-Am-Az-Vi-Ma
							N

Verificación mediante código

```
Pieza 1 -> R
Pieza 2 -> N
Pieza 3 -> R
Pieza 4 -> N
Pieza 5 -> Am
Pieza 6 -> Ve
Pieza 7 -> R
Pieza 8 -> N
Pieza 9 -> Ve
Pieza 10 -> N
Pieza 11 -> R
Pieza 12 -> R
Pieza 13 -> Ve
Pieza 14 -> R
Pieza 15 -> Am
Pieza 16 -> Ve
Pieza 17 -> N
```

Se observa que solo son necesarios 4 colores para pintar esta imagen.



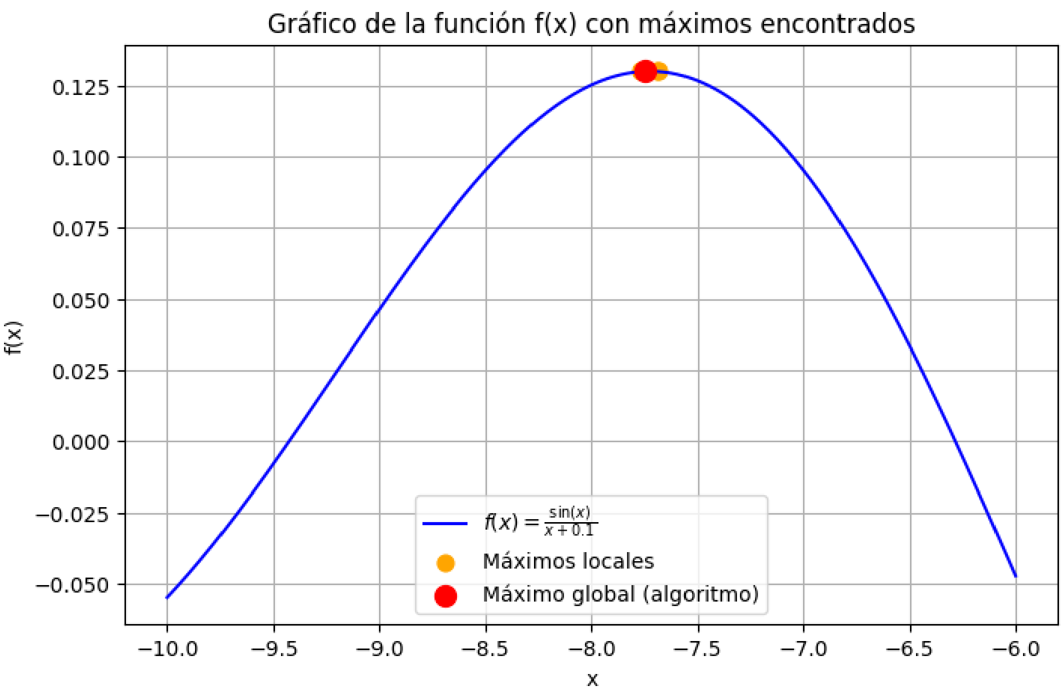
3 Ejercicios de Implementación

3.1 Encuentre el máximo de la función $f(x) = \frac{\sin(x)}{x+0.1}$ en $x \in [-10; -6]$ con un error menor a 0.1 utilizando el algoritmo *hill climbing*.

```
# Importo numpy para la función sin(x)
import numpy as np
import random
import pandas as pd
import matplotlib.pyplot as plt

def f(x):
    if x < -10 or x > -6:
        raise ValueError("x debe estar en el intervalo [-10, -6]")
    else:
        return (np.sin(x))/(x+0.1)

def ClimbingHill(f, x0, error, max_iter=100000):
    step = 0.05
    x = x0
    for i in range(max_iter):
        vecinos = [x - step, x + step]
        next_x = max(vecinos, key=f)

        if f(next_x) - f(x) < error:
            return next_x
        x = next_x

resultados = []

for i in range(8):
    semilla = -10 + (random.random() * (-6 - (-10)))
    error = 1e-9

    maximo = ClimbingHill(f, semilla, error)

    resultados.append({
        "x_inicial": semilla,
        "f(x_inicial)": f(semilla),
        "x_max_encontrado": maximo,
        "f(x_max_encontrado)": f(maximo)
    })

# Convertir a tabla con pandas
df = pd.DataFrame(resultados)

# Mostrar tabla
print(df.to_string(index=False))

# Generar puntos en el intervalo válido
x_vals = np.linspace(-10, -6, 400)
y_vals = [f(x) for x in x_vals]

# Buscar el máximo global dentro de los encontrados por el algoritmo
idx_max = df["f(x_max_encontrado)"].idxmax()   # índice del máximo
x_max_global = df.loc[idx_max, "x_max_encontrado"]
y_max_global = df.loc[idx_max, "f(x_max_encontrado)"]

print("Máximo global encontrado por el algoritmo:")
```

```python
print("x =", x_max_global, " ; f(x) =", y_max_global)

# Graficar la función
plt.figure(figsize=(8,5))
plt.plot(x_vals, y_vals, label=r"$f(x) = \frac{\sin(x)}{x+0.1}$", color="blue")

# Marcar todos los máximos locales encontrados
plt.scatter(df["x_max_encontrado"], df["f(x_max_encontrado)"],
            color="orange", s=60, label="Máximos locales")

# Marcar el máximo maximorum
plt.scatter(x_max_global, y_max_global,
            color="red", s=100, zorder=5, label="Máximo global (algoritmo)")

# Configuración
plt.xlabel("x")
plt.ylabel("f(x)")
plt.title("Gráfico de la función f(x) con máximos encontrados")
plt.legend()
plt.grid(True)
plt.show()

```

```
 x_inicial  f(x_inicial)  x_max_encontrado  f(x_max_encontrado)
 -8.818769      0.065329         -7.768769             0.129926
 -7.600496      0.129064         -7.750496             0.130011
 -6.350011      0.010684         -7.750011             0.130013
 -9.485597     -0.006476         -7.685597             0.129964
 -7.541716      0.127879         -7.691716             0.129992
 -9.754369     -0.033524         -7.754369             0.129997
 -7.053900      0.100181         -7.753900             0.129999
 -8.520915      0.093306         -7.770915             0.129913
Máximo global encontrado por el algoritmo:
x = -7.750010920807812  ; f(x) = 0.1300128748659443
```



Notas

- Solución

En este caso, buscar una solución analítica es difícil. Se conoce la expresión para la función y su derivada, por lo que usando métodos numéricos se encuentra el máximo local en [-10; -6]. La solución de acá se saca con MATLAB.

$$f(x) = \frac{\sin(x)}{x + 0.1}$$

$$f'(x) = \frac{\cos(x)(x + 0.1) - \sin(x)}{(x + 0.1)^2}$$

$$f_{max} \to x \approx 0.130015$$

- `semilla = -10 + (random.random() * (-6 - (-10)))`

Tomamos una semilla aleatorea para mostrar que existen máximos locales y que el método Climbing Hills podría quedarse atorado en un máximo local en vez del máximo global.

- `next_x = max(vecinos, key=f)`

Se utiliza “key=f” para que la función `max()` compare los valores `f(vecinos[i])` en vez de los valores de la lista `vecinos[i]`.

Anexo 4.1 En el intervalo [-10, -6] existe un único máximo, que podríamos decir que es el “máximo global” de nuestro “paisaje del espacio de estado”. Para ello, si disminuimos la frecuencia de la expreción sinusoidal, podemos “apretar” la función y observaremos que aparecen máximos locales y un único “máximo global” en nuestro “paisaje del espacio de estado” en [-10, -6].

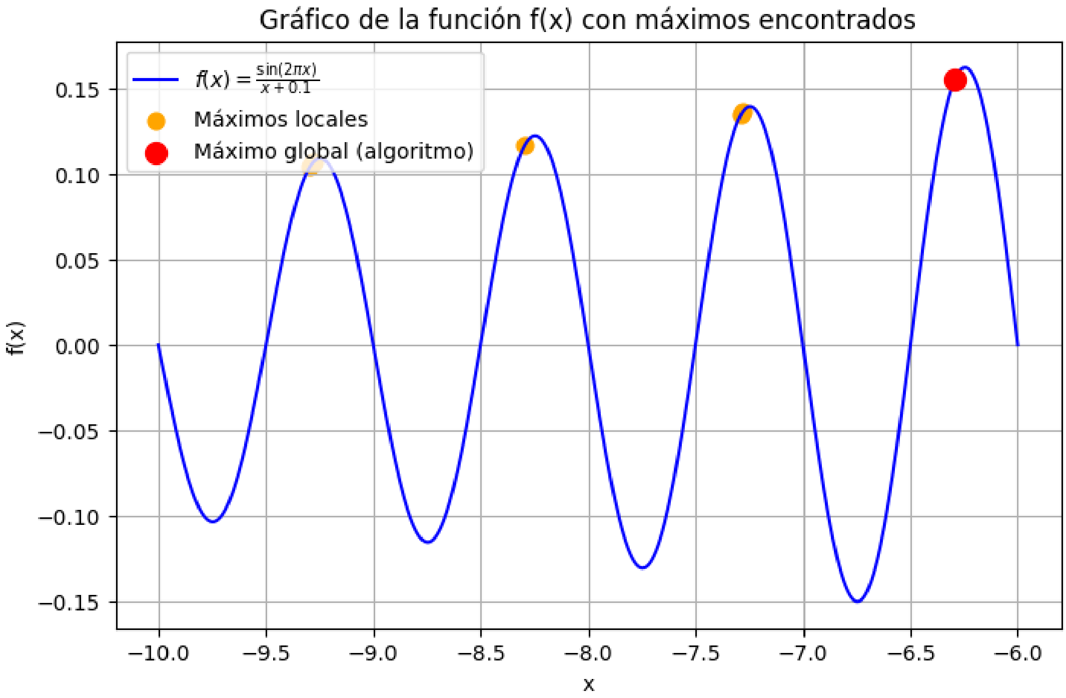
Nueva expresión:

$$f_{\text{nueva}} = \frac{\sin(2\pi x)}{x + 0.1}$$

```python
# Importo numpy para la función sin(x)
import numpy as np
import random
import pandas as pd
import matplotlib.pyplot as plt

def f(x):
    if x < -10 or x > -6:
        raise ValueError("x debe estar en el intervalo [-10, -6]")
    else:
        return (np.sin(2*np.pi*x))/(x+0.1)

def ClimbingHill(f, x0, error, max_iter=100000):
    step = 0.05
    x = x0
    for i in range(max_iter):
        vecinos = [x - step, x + step]
        next_x = max(vecinos, key=f)

        if f(next_x) - f(x) < error:
            return next_x
        x = next_x

resultados = []

for i in range(8):
    semilla = -10 + (random.random() * (-6 - (-10)))
    error = 1e-9

    maximo = ClimbingHill(f, semilla, error)

    resultados.append({
        "x_inicial": semilla,
        "f(x_inicial)": f(semilla),
        "x_max_encontrado": maximo,
        "f(x_max_encontrado)": f(maximo)
    })

# Convertir a tabla con pandas
df = pd.DataFrame(resultados)

# Mostrar tabla
print(df.to_string(index=False))

# Generar puntos en el intervalo válido
x_vals = np.linspace(-10, -6, 400)
y_vals = [f(x) for x in x_vals]

# Buscar el máximo global dentro de los encontrados por el algoritmo
idx_max = df["f(x_max_encontrado)"].idxmax()   # índice del máximo
x_max_global = df.loc[idx_max, "x_max_encontrado"]
y_max_global = df.loc[idx_max, "f(x_max_encontrado)"]

print("Máximo global encontrado por el algoritmo:")
```

```python
print("x =", x_max_global, " ; f(x) =", y_max_global)

# Graficar la función
plt.figure(figsize=(8,5))
plt.plot(x_vals, y_vals, label=r"$f(x) = \frac{\sin(2 \pi x)}{x+0.1}$", color="blue")

# Marcar todos los máximos locales encontrados
plt.scatter(df["x_max_encontrado"], df["f(x_max_encontrado)"],
            color="orange", s=60, label="Máximos locales")

# Marcar el máximo maximorum
plt.scatter(x_max_global, y_max_global,
            color="red", s=100, zorder=5, label="Máximo global (algoritmo)")

# Configuración
plt.xlabel("x")
plt.ylabel("f(x)")
plt.title("Gráfico de la función f(x) con máximos encontrados")
plt.legend()
plt.grid(True)
plt.show()
```

```
 x_inicial  f(x_inicial)  x_max_encontrado  f(x_max_encontrado)
 -9.177386      0.098896         -9.277386             0.107354
 -6.095235      0.093959         -6.295235             0.154938
 -6.988760     -0.010244         -7.288760             0.135001
 -8.443475      0.041678         -8.293475             0.117523
 -7.731912     -0.130183         -7.281912             0.136449
 -6.644429     -0.120400         -6.294429             0.155186
 -8.994513     -0.003876         -9.294513             0.104534
 -9.578870     -0.050166         -9.278870             0.107158
Máximo global encontrado por el algoritmo:
x = -6.294428553951991  ; f(x) = 0.15518610041423628
```



Acá se nota mejor el problema de este mecanismo: puede quedar atrapado en máximos locales, y no salir de ellos, fallando al objetivo de encontrar el máximo global.

3.2 Diseñe e implemente un algoritmo de Recocido Simulado para que juegue contra usted al Ta-te-ti. Varíe los valores de temperatura inicial entre partidas, ¿qué diferencia observa cuando la temperatura es más alta con respecto a cuando la temperatura es más baja?

Cuando jugás con Recocido Simulado, la diferencia clave entre una temperatura inicial alta y una baja está en la probabilidad de aceptar jugadas "malas" al principio:

- Con temperatura alta el algoritmo es más permisivo: acepta con mayor probabilidad movimientos peores (jugadas que a corto plazo parecen malas). Esto permite explorar más el espacio de soluciones, escapar de óptimos locales y probar estrategias poco convencionales. En el Ta-te-ti, la IA puede sorprender con movimientos menos obvios, incluso arriesgados, antes de estabilizarse.
- Con temperatura baja la aceptación de jugadas malas casi desaparece: la búsqueda se vuelve codiciosa, solo toma jugadas que parecen mejorar inmediatamente. Esto hace que la IA juegue más rígida y predecible, pero también corre el riesgo de quedarse "atascada" en decisiones mediocres sin explorar alternativas mejores a largo plazo

Se pone el script del código, pero para su correcto funcionamiento debe ejecutarse el código TaTeTi.py en la terminal, caso contrario no funcionará de la manera deseada.

3.2.1 1. Tablero y reglas

- def new_board(): return [' '] * 9

Crea un tablero vacío de 9 casillas.

- def print_board(b):

  Dibuja el tablero en consola, reemplazando espacios vacíos por números de casilla (1–9).

  def available_moves(b): ...

  def place(b, i, mark): ...

  def winner(b): ...

  def is_draw(b): ...

  def copy_board(b): ...

Funciones utilitarias:

- available_moves: devuelve lista de posiciones libres.
- place: coloca X u O.
- winner: chequea las combinaciones ganadoras.
- is_draw: empate (tablero lleno sin ganador).
- copy_board: copia para simular sin modificar el original.

3.2.2 2. Rollouts (simulaciones de jugadas)

Estas funciones permiten evaluar una jugada mirando al futuro:

- random_policy_move: juega greedy (si puede ganar, gana; si debe bloquear, bloquea; si no, juega al azar).
- simulate_from_move: ejecuta una partida entera desde una jugada, devolviendo +1 si gana la IA, 0 empate, -1 si pierde.
- estimated_value: repite la simulación N veces (rollouts) y calcula el valor esperado de esa jugada.

  Esto te da una función heurística estocástica: mide la "calidad" de cada casilla.

3.2.3 3. Recocido Simulado (SA)

def Recocido(...):

  Acá está la esencia:

  Arranca con un movimiento candidato (current).

  Evalúa su valor (cur_val).

  Repite mientras la temperatura T > Tf:

  Genera un vecino (otro movimiento posible).

  Calcula la diferencia de energía dE.

  Acepta o no con la probabilidad:

$$P = e^{(-\Delta E/T)}$$

Si mejora, guarda el mejor movimiento (best).

Al final devuelve el best.

Traducción a IA: cuanto más alta es la T, más probable es que el agente acepte jugadas malas (explora). Con T baja, se vuelve codicioso (explotación).

3.2.4 4. Interfaz de juego

- def ask_move(b, mark): …

Pregunta al humano por consola (usa input()). Valida la jugada.

def play_human_vs_sa(…):

Loop principal: Elige al azar quién es humano y quién IA.

Turnos alternados.

Humano → ask_move.

IA → Recocido.

Revisa ganador o empate.

3.2.5 5. Punto de entrada

def main(): … play_human_vs_sa(…)

Ejecuta el juego con parámetros por defecto (T0=5.0, alpha=0.95, etc.).

3.2.6 NOTA

Se deshabilito la opción de “bloquear jugada”. Es decir, la IA no tiene un comportamiento de “si no pongo mi jugada acá, perderé, entonces bloquearé al rival”, sino que se rige exclusivamente por el comportamiento del Recocido. Para habilitar esta opción se debe descomentar las lineas de 98 a 117.

```
1  # Ta-te-ti con IA por Recocido Simulado (Simulated Annealing)
2
3  import math
4  import random
5
6  # ----------------- Tablero y reglas -----------------
7
8  def new_board():
9      return [' '] * 9
10
11 def print_board(b):
12     print()
13     for r in range(3):
14         row = b[3*r:3*r+3]
15         print(' ' + ' | '.join(c if c != ' ' else str(3*r+i+1) for i, c in enumerate(row)))
16         if r < 2: print("---+---+---")
17     print()
18
19 def available_moves(b):
20     return [i for i, c in enumerate(b) if c == ' ']
```

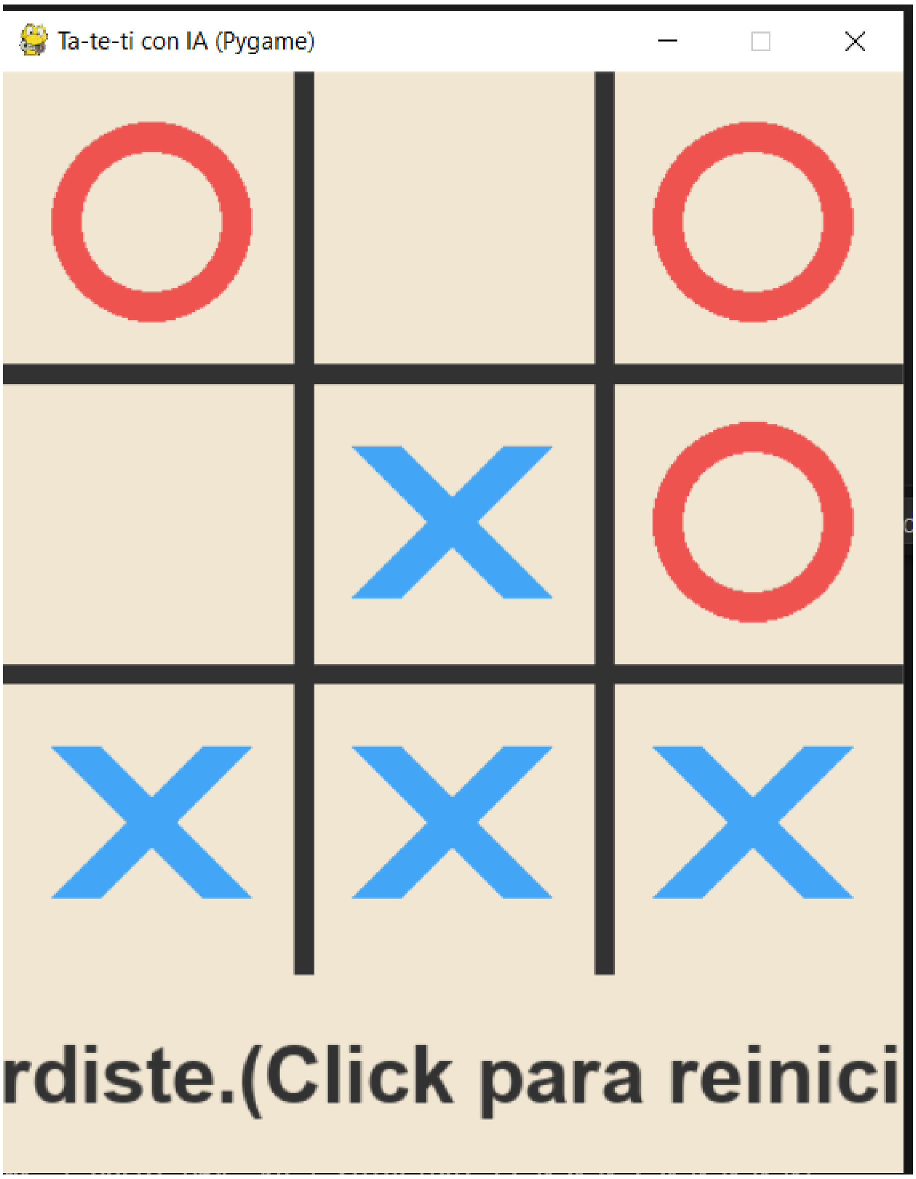
```python

def place(b, i, mark):
    b[i] = mark

def winner(b):
    lines = [(0,1,2),(3,4,5),(6,7,8),
             (0,3,6),(1,4,7),(2,5,8),
             (0,4,8),(2,4,6)]
    for i,j,k in lines:
        if b[i] != ' ' and b[i] == b[j] == b[k]:
            return b[i]
    return None

def is_draw(b):
    return winner(b) is None and all(c != ' ' for c in b)

def copy_board(b):
    return b[:]

# ----------------- Rollouts para evaluar una jugada -----------------

def random_policy_move(b, player):
    """Política simple para los rollouts: si hay jugada ganadora inmediata la toma,
    si puede bloquear pérdida inmediata bloquea; si no, juega al azar."""
    for i in available_moves(b):
        bb = copy_board(b)
        place(bb, i, player)
        if winner(bb) == player:
            return i
    opp = 'O' if player == 'X' else 'X'
    for i in available_moves(b):
        bb = copy_board(b)
        place(bb, i, opp)
        if winner(bb) == opp:
            return i
    return random.choice(available_moves(b))

def simulate_from_move(b, move, ai, hu):
    """Simula una partida completa desde la jugada 'move' del AI.
       Devuelve +1 si gana AI, 0 empate, -1 si pierde."""
    bb = copy_board(b)
    place(bb, move, ai)
    w = winner(bb)
    if w == ai: return 1
    if is_draw(bb): return 0

    turn = hu
    while True:
        m = random_policy_move(bb, turn)
        place(bb, m, turn)
        w = winner(bb)
        if w == ai: return 1
        if w == hu: return -1
        if is_draw(bb): return 0
        turn = ai if turn == hu else hu

def estimated_value(b, move, ai, hu, rollouts=40):
    """Promedia N simulaciones desde la jugada 'move'."""
    s = 0
    for _ in range(rollouts):
```

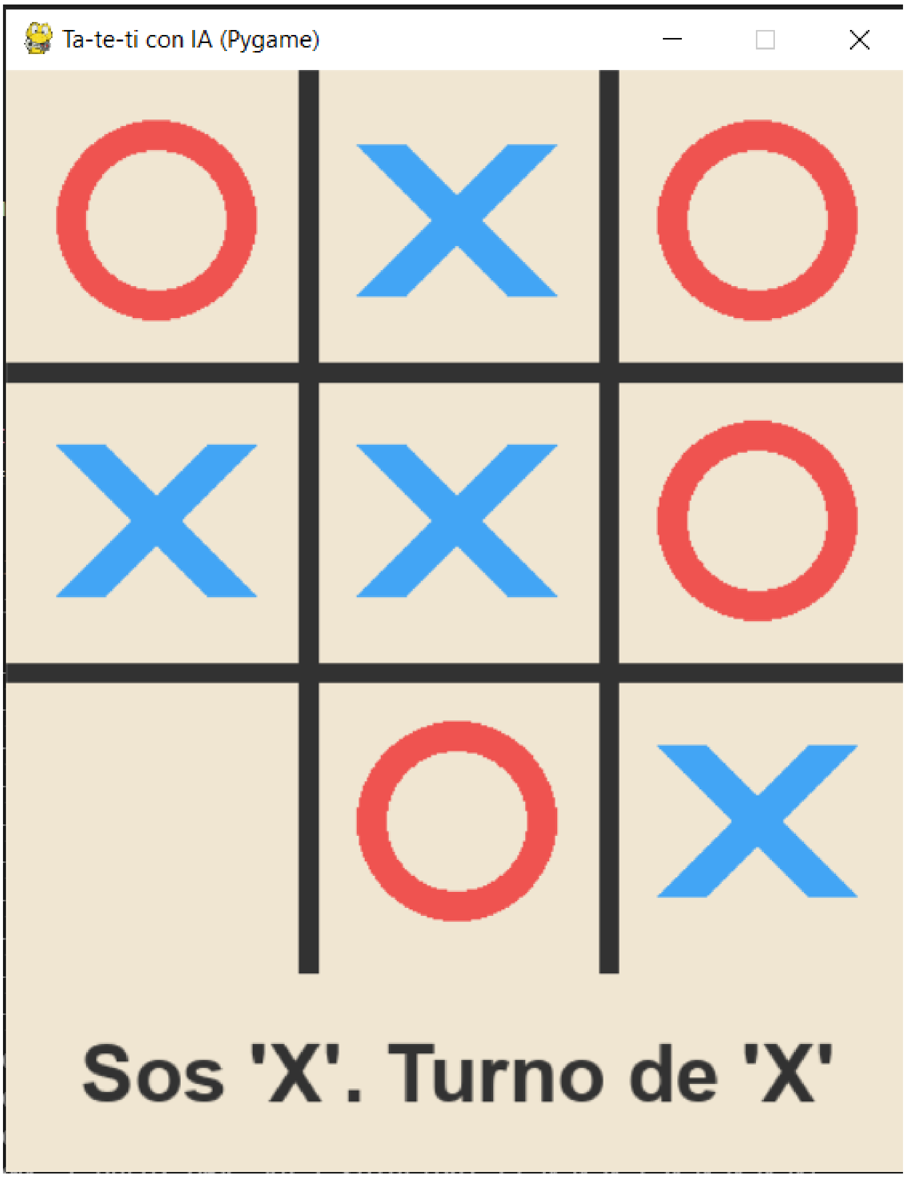
```python
            s += simulate_from_move(b, move, ai, hu)
        return s / rollouts

# ----------------- Recocido Simulado para elegir jugada -----------------

def Recocido(b, ai, hu, T0=5.0, Tf=0.1, alpha=0.95, L=20, rollouts=40):
    """Devuelve una casilla usando Simulated Annealing.
       - T0: temperatura inicial
       - Tf: temperatura final
       - alpha: factor de enfriamiento (geométrico)
       - L: iteraciones por temperatura
       - rollouts: simulaciones por evaluación"""

    empties = available_moves(b)

    # Si hay jugada ganadora inmediata o bloqueo, sé pragmático:

    #for i in empties:
    #    bb = copy_board(b)
    #    place(bb, i, ai)
    #    if winner(bb) == ai:
    #        return i
    #for i in empties:
    #    bb = copy_board(b)
    #    place(bb, i, hu)
    #    if winner(bb) == hu:
    #        return i

    # Candidato inicial: cualquiera libre
    current = random.choice(empties)
    best = current
    cur_val = estimated_value(b, current, ai, hu, rollouts=rollouts)
    best_val = cur_val

    T = T0
    while T > Tf and len(empties) > 1:
        for _ in range(L):
            # Vecino: otra casilla libre distinta
            neighbor = current
            while neighbor == current:
                neighbor = random.choice(empties)
            neigh_val = estimated_value(b, neighbor, ai, hu, rollouts=rollouts)
            dE = -(neigh_val - cur_val)  # energía = -valor
            # Aceptación de Metrópolis
            if dE < 0 or random.random() < math.exp(-dE / T):
                current, cur_val = neighbor, neigh_val
                if cur_val > best_val:
                    best, best_val = current, cur_val
        T *= alpha
    return best

# ----------------- Interfaz de juego -----------------

def ask_move(b, mark):
    while True:
        s = input(f"Turno de {mark}. Casillero (1-9): ").strip()
        try:
            i = int(s) - 1
            if i not in range(9): print("Rango 1-9, maestro."); continue
            if b[i] != ' ': print("Ocupado. Probá otro."); continue
```

```python
            return i
        except:
            print("Número válido, por favor.")

def play_human_vs_sa(T0=5.0, Tf=0.1, alpha=0.95, L=20, rollouts=40):
    b = new_board()
    human = random.choice(['X', 'O'])
    ai = 'O' if human == 'X' else 'X'
    print(f"Vos sos {human}. La IA es {ai}.")
    print_board(b)
    turn = 'X'
    while True:
        if turn == human:
            i = ask_move(b, human)
            place(b, i, human)
        else:
            print(f"IA pensando con SA (T0={T0})...")
            i = Recocido(b, ai, human, T0=T0, Tf=Tf, alpha=alpha, L=L, rollouts=rollouts)
            place(b, i, ai)
            print(f"IA juega en {i+1}.")
        print_board(b)
        w = winner(b)
        if w: print(f"Gana {w}."); return
        if is_draw(b): print("Empate."); return
        turn = 'O' if turn == 'X' else 'X'

def main():
    print("=== TA-TE-TI con Recocido Simulado ===")
    try:
        T0 = float(input("T0 (ej. 0.2, 1, 5, 10): ").strip() or "5")
    except:
        T0 = 5.0
    Tf = 0.1
    alpha = 0.95
    L = 20
    rollouts = 40
    play_human_vs_sa(T0=T0, Tf=Tf, alpha=alpha, L=L, rollouts=rollouts)

if __name__ == "__main__":
    main()
```

Resultados de nuestro TaTeTi con interfaz grafica

Aca la IA nos gano:


Ta-te-ti con IA (Pygame)
rdiste.(Click para reinici



3.3 Diseñe e implemente un algoritmo genético para cargar una grúa con $n = 10\ cajas$ que puede soportar un peso máximo $C = 1000\ kg$. Cada caja j tiene asociado un precio p_j y un peso w_j como se indica en la tabla de abajo, de manera que el algoritmo debe ser capaz de maximizar el precio sin superar el límite de carga.

Elemento (j)

1

2

3

4

5

6

7

8

9

10

Precio (p_j)

100

50

115

25

200

30

40

100

100

100

Peso (w_j)

300

200

450

145

664

90

150

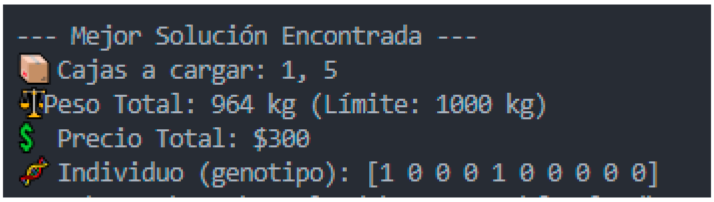
355

401

395

```
6.1 En primer lugar, es necesario representar qué cajas estarán cargadas en la grúa y cu

6.2 A continuación, genere una Población que contenga un número $N$ de individuos (se re

6.3 Cree ahora una función que permita evaluar la Idoneidad de cada individuo y seleccio

6.4 Por último, Cruce las parejas elegidas, aplique un mecanismo de Mutación y verifique

6.5 Realice este proceso iterativamente hasta que se cumpla el mecanismo de detención de
```

Resultados de nuestro codigo:




```python
import numpy as np
import sys
sys.stdout.reconfigure(encoding='utf-8')

# --- Datos del Problema ---
#              Caja: 1    2    3    4    5    6    7    8    9    10
precios = np.array([100, 50, 115, 25, 200, 30, 40, 100, 100, 100])
pesos   = np.array([300, 200, 450, 145, 664, 90, 150, 355, 401, 395])
capacidad_maxima = 1000
n_cajas = 10

# --- Parámetros del Algoritmo Genético ---
tamano_poblacion = 20       # Un número par
tasa_mutacion = 0.05        # Probabilidad de que un gen (caja) mute
num_generaciones = 200      # Mecanismo de detención

# --- 6.2 Generación de la Población Inicial ---
def crear_poblacion_inicial(tamano, n_items, pesos_items, capacidad):
    poblacion = []
    while len(poblacion) < tamano:
        individuo = np.random.randint(2, size=n_items)
        peso_actual = np.sum(individuo * pesos_items)
        if peso_actual <= capacidad:
            poblacion.append(individuo)
    return np.array(poblacion)

# --- 6.3 Función de Idoneidad y Selección por Ruleta ---
def calcular_idoneidad(poblacion, precios_items):
    return np.dot(poblacion, precios_items)

def seleccion_ruleta(poblacion, idoneidad):
    suma_idoneidad = np.sum(idoneidad)
    if suma_idoneidad == 0:
        probabilidades = np.ones(len(poblacion)) / len(poblacion)
    else:
        probabilidades = idoneidad / suma_idoneidad
    indices_elegidos = np.random.choice(len(poblacion), size=len(poblacion), p=probabilidades)
    return poblacion[indices_elegidos]

# --- 6.4 Cruce, Mutación y Verificación ---
def cruce_y_mutacion(padres, pesos_items, capacidad, tasa_mut):
    nueva_generacion = []
    np.random.shuffle(padres)
    for i in range(0, len(padres), 2):
        padre1 = padres[i]
        padre2 = padres[i+1] if i + 1 < len(padres) else padres[i]

        punto_cruce = np.random.randint(1, len(padre1))
        hijo1 = np.concatenate([padre1[:punto_cruce], padre2[punto_cruce:]])
        hijo2 = np.concatenate([padre2[:punto_cruce], padre1[punto_cruce:]])

```

```python
        for j in range(len(hijo1)):
            if np.random.rand() < tasa_mut:
                hijo1[j] = 1 - hijo1[j]
            if np.random.rand() < tasa_mut:
                hijo2[j] = 1 - hijo2[j]

        if np.sum(hijo1 * pesos_items) <= capacidad:
            nueva_generacion.append(hijo1)
        else:
            nueva_generacion.append(padre1)
        if np.sum(hijo2 * pesos_items) <= capacidad:
            nueva_generacion.append(hijo2)
        else:
            nueva_generacion.append(padre2)

    return np.array(nueva_generacion)

# --- 6.5 Proceso Iterativo y Resultado Final ---

print("--- Iniciando Evolución del Algoritmo Genético ---\n")

mejor_individuo_global = None
mejor_idoneidad_global = -1

poblacion = crear_poblacion_inicial(tamano_poblacion, n_cajas, pesos, capacidad_maxima)

for generacion in range(num_generaciones):
    idoneidad = calcular_idoneidad(poblacion, precios)

    # Encontrar el mejor individuo de la generación actual
    indice_mejor_gen = np.argmax(idoneidad)
    mejor_individuo_gen = poblacion[indice_mejor_gen]
    mejor_idoneidad_gen = idoneidad[indice_mejor_gen]

    # Actualizar el mejor individuo global si es necesario
    if mejor_idoneidad_gen > mejor_idoneidad_global:
        mejor_idoneidad_global = mejor_idoneidad_gen
        mejor_individuo_global = mejor_individuo_gen

    # *** NUEVA LÍNEA PARA MOSTRAR EL PROGRESO ***
    # Muestra el mejor resultado encontrado HASTA AHORA en cada generación
    peso_actual = np.sum(mejor_individuo_global * pesos)
    print(f"Generación {generacion+1:03d} | Mejor Precio: ${mejor_idoneidad_global:<4} | Peso: {peso_actual:<4} kg | Solución: {mejor_individuo_global}")

    padres = seleccion_ruleta(poblacion, idoneidad)
    poblacion = cruce_y_mutacion(padres, pesos, capacidad_maxima, tasa_mutacion)

# --- Mostrar Resultados Finales ---
precio_final = np.sum(mejor_individuo_global * precios)
peso_final = np.sum(mejor_individuo_global * pesos)
cajas_seleccionadas = np.where(mejor_individuo_global == 1)[0] + 1

print("\n--- Mejor Solución Encontrada ---")
print(f"  Cajas a cargar: {list(cajas_seleccionadas)}")
print(f"  Peso Total: {peso_final} kg (Límite: {capacidad_maxima} kg)")
print(f"  Precio Total: ${precio_final}")
print(f"  Individuo (genotipo): {mejor_individuo_global}")
```

4 Bibliografía

Russell, S. & Norvig, P. (2004) *Inteligencia Artificial: Un Enfoque Moderno.* Pearson Educación S.A. (2a Ed.) Madrid, España

Poole, D. & Mackworth, A. (2023) *Artificial Intelligence: Foundations of Computational Agents.* Cambridge University Press (3a Ed.) Vancouver, Canada