

# Temas Tratados en el Trabajo Práctico 3

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

## Ejercicios Teóricos

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 de detención del algoritmo de ascenso de colinas es la ausencia de mejora en los vecinos, lo que hace que se detenga generalmente en un máximo local o en una meseta, no en el óptimo global. En cada paso selecciona el mejor vecino según la función objetivo, avanzando hacia soluciones con mejor valor. Su principal limitación es que puede quedar atrapado en un máximo local (que no es la mejor solución global) o estancado en zonas de mesetas o terrazas, donde los vecinos tienen el mismo valor y no hay una dirección clara de mejora.

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

### 1. Ordenación de variables

MRV (Minimum Remaining Values / elección por dominio más pequeño) Elegir la variable que tiene menos valores legales en su dominio. Por qué ayuda: detecta pronto variables "más restringidas" y evita explorar ramas que van a fallar tarde. Ejemplo: en 8-reinas, elegir la fila que sólo tiene 2 columnas posibles en lugar de una con 6.

Degree heuristic (heurística del grado) Como desempate del MRV, elegir la variable que está involucrada en más restricciones con variables no asignadas. Por qué ayuda: reduce el impacto en muchas otras variables (maximiza la propagación).

### 2. Ordenación de valores

LCV (Least Constraining Value / valor menos restrictivo) Para la variable elegida, probar primero los valores que dejan más opciones a las variables no asignadas. Por qué ayuda: facilita encontrar continuaciones válidas, reduce retrocesos. Ejemplo: al colocar una reina en una columna, elegir la columna que elimina menos columnas válidas para las demás filas.

### 3. Comprobación e inferencia durante la búsqueda

Forward Checking (comprobación hacia adelante) Cuando asignás una variable, se eliminan de inmediato de los dominios de las variables vecinas los valores que violan la restricción. Si algún dominio queda vacío, hay retroceso inmediato. Beneficio: detecta fallos más temprano que la búsqueda pura.

Arc-consistency (p. ej. AC-3) Algoritmo que mantiene consistencia binaria entre pares de variables (elimina valores que no tienen soporte en vecinos). Beneficio: más fuerte que forward-checking; reduce dominios antes y durante la búsqueda.

MAC (Maintaining Arc Consistency) Mantener AC dinámicamente después de cada asignación (combina backtracking con AC). Es costoso por paso, pero suele ahorrar muchísimos retrocesos.

Propagación por restricciones globales Usar propagadores eficientes para restricciones especiales (p. ej. AllDifferent, cumulative para scheduling). Estos propagadores quitan muchos valores inválidos rápidamente.

#### 4. Mejoras en el backtracking

Backjumping / Conflict-directed backjumping En vez de retroceder sólo a la última variable, saltar directamente a la variable que causó el conflicto. Beneficio: evita explorar ramas inútiles.

Learning / Nogoods Guardar combinaciones parciales que ya fallaron (nogoods) para no volver a probarlas.

#### 5. Heurísticas y estrategias para búsqueda local

Min-conflicts Para grandes CSPs, iniciar con una asignación completa (posible en conflicto) y en cada paso cambiar la variable que reduce más los conflictos. Muy eficaz en problemas como 8-reinas o grandes grafos de coloreo.

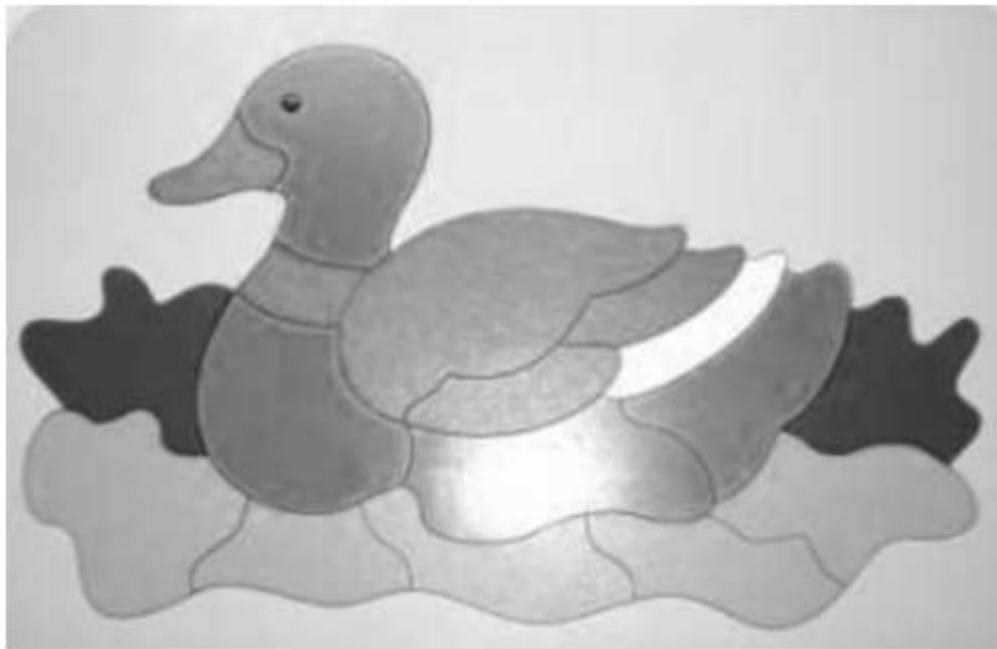
Random restarts / random walk Reiniciar la búsqueda local varias veces o permitir movimientos aleatorios para escapar de óptimos locales.

#### 6. Preprocesamiento y descomposición

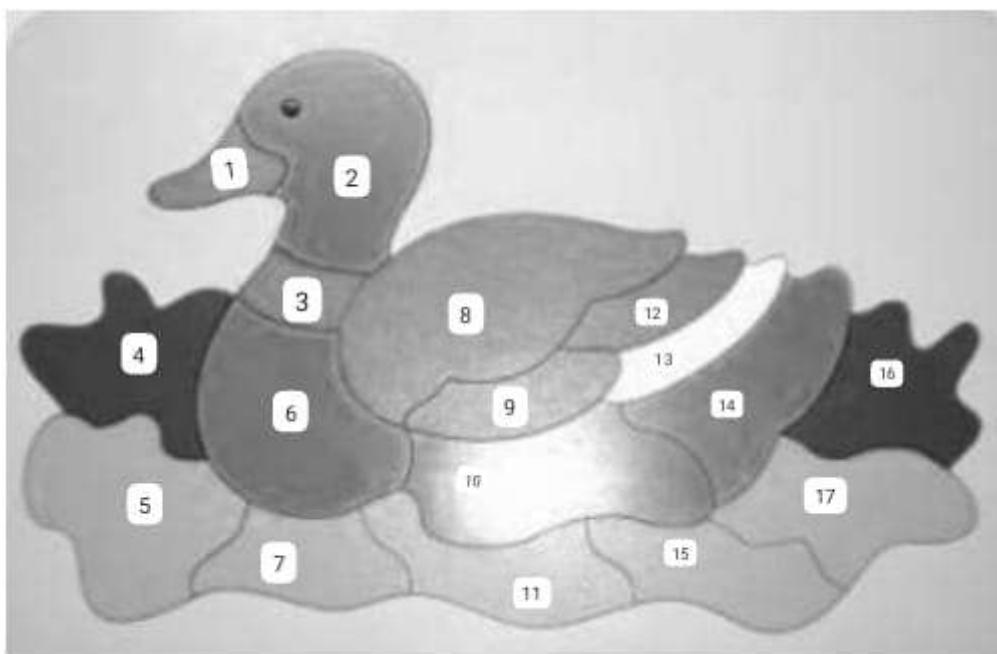
Descomponer el grafo de restricciones en componentes conectados y resolver cada componente por separado.

Ordenar/renombrar variables con algún criterio global (p. ej. heurística estática: ordenar por grado antes de comenzar) para acelerar la búsqueda.

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.



Primero vamos a dividir el pato en numeros para poder identificar las partes. Luego haremos una lista de los vecinos de cada cuadro



```
In [1]: graph = {
    "1": ["2"],
    "2": ["1", "3", "8"],
    "3": ["2", "6", "8"],
    "4": ["5", "6"],
    "5": ["4", "6", "7"],
    "6": ["3", "4", "5", "7", "8", "9", "10", "11"],
    "7": ["5", "6", "11"],
    "8": ["2", "3", "6", "9", "12"],
    "9": ["6", "8", "10", "12", "13"],
    "10": ["6", "9", "11", "13", "14", "15"],
    "11": ["6", "7", "10", "15"],
    "12": ["8", "9", "13"],
    "13": ["9", "10", "12", "14"],
    "14": ["10", "13", "16", "17"],
    "15": ["10", "11", "17"]}
```

```

        "16": ["14", "17"],
        "17": ["14", "15", "16"]
    }
colors = {"Rojo", "Azul", "Verde", "Amarillo", "Naranja", "Violeta", "Marrón"}

```

Paso	Nodo elegido (MRV)	Dominio antes	Color asignado	Vecinos afectados	Cambios en dominios
1	6	{Amarillo, Azul, Marrón, Naranja, Rojo, Verde, Violeta}	Amarillo	3, 4, 5, 7, 8, 9, 10, 11	3→Amarillo; 4→Amarillo; 5→Amarillo; 7→Amarillo; 8→Amarillo; 9→Amarillo; 10→Amarillo; 11→Amarillo
2	10	{Azul, Marrón, Naranja, Rojo, Verde, Violeta}	Azul	9, 11, 13, 14, 15	9→Azul; 11→Azul; 13→Azul; 14→Azul; 15→Azul
3	9	{Marrón, Naranja, Rojo, Verde, Violeta}	Marrón	8, 12, 13	8→Marrón; 12→Marrón; 13→Marrón
4	8	{Azul, Naranja, Rojo, Verde, Violeta}	Azul	2, 3, 12	2→Azul; 3→Azul; 12→Azul
5	11	{Marrón, Naranja, Rojo, Verde, Violeta}	Marrón	7, 15	7→Marrón; 15→Marrón
6	5	{Azul, Marrón, Naranja, Rojo, Verde, Violeta}	Azul	4, 7	4→Azul; 7→Azul
7	3	{Marrón, Naranja, Rojo, Verde, Violeta}	Marron	2	2→Azul
8	7	{Naranja, Rojo, Verde, Violeta}	Naranja		
9	4	{Marrón, Naranja, Rojo, Verde, Violeta}	Marron		
10	14	{Amarillo, Marrón, Naranja, Rojo, Verde, Violeta}	Amarillo	13, 16, 17	13→Amarillo; 16→Amarillo; 17→Amarillo
11	13	{Naranja, Rojo, Verde, Violeta}	Naranja	12	12→Naranja
12	15	{Amarillo, Naranja, Rojo, Verde, Violeta}	Amarillo	17	17→Amarillo
13	16	{Azul, Marrón, Naranja, Rojo, Verde, Violeta}	Azul	17	17→Azul
14	17	{Marrón, Naranja, Rojo, Verde, Violeta}	Marron		
15	2	{Amarillo, Marrón, Rojo, Verde, Violeta}	Amarillo	1	1→Amarillo
16	1	{Azul, Marrón, Naranja, Rojo, Verde, Violeta}	Azul		
17	12	{Amarillo, Rojo, Verde, Violeta}	Amarillo		

# Ejercicios de Implementación

4. 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*.

In [168...]

```
import math
import random

# Definición de la función objetivo
def f(x):
    """
    Función objetivo: f(x) = sin(x) / (x + 0.1)
    Desde el punto de vista de búsqueda local:
    - Esta función tiene varios máximos y mínimos locales.
    - Es un buen ejemplo para mostrar cómo el algoritmo Hill Climbing
      puede encontrar un óptimo local sin garantizar el global.
    """
    return math.sin(x) / (x + 0.1)

# Implementación del algoritmo Hill Climbing
def hill_climb(x0, step=0.5, max_iter=200000):
    """
    Algoritmo de Hill Climbing (ascenso de colinas).

    Parámetros:
    - x0: estado inicial (semilla aleatoria dentro del dominio).
    - step: tamaño de paso para explorar los vecinos.
      Se elige 0.5 como valor inicial porque es suficientemente grande
      para moverse rápido en las primeras fases, pero luego se ajusta dinámicamente.
    - max_iter: número máximo de iteraciones para evitar bucles infinitos.

    Estrategia:
    1. Evaluar la función en la posición actual.
    2. Explorar vecinos a izquierda y derecha (x + step, x - step).
    3. Si algún vecino mejora el valor de f(x), se mueve allí (voraz).
    4. Si no se mejora, se reduce el tamaño de paso (step /= 2).
      Esto implementa una especie de "enfriamiento local",
      permitiendo ajustar la precisión hasta detenerse.
    5. Se detiene si el paso es menor a 0.01 o si se alcanza max_iter.
    """
    x = x0
    fx = f(x)

    for _ in range(max_iter):
        improved = False
        # Explora vecinos: derecha (step) e izquierda (-step)
        for dx in (step, -step):
            xn = x + dx
            # Restricción: el nuevo candidato debe estar en el intervalo [-10, -6]
            if xn < -10 or xn > -6:
                continue
            fn = f(xn)
            # Se acepta el vecino si mejora significativamente la solución actual
            if fn > fx + 1e-7: # umbral evita oscilaciones por redondeo
                x, fx = xn, fn
                improved = True
                break # movimiento voraz: toma la primera mejora
        # Si no hay mejora, reducimos el paso
        if not improved:
            step /= 2.0
```

```

        if step < 1e-3:
            break # criterio de parada por precisión
    return x, fx

# Elección del punto inicial
start = random.uniform(-10, -6) # se elige aleatoriamente dentro del dominio
x, fx = hill_climb(start, step=0.5)

# Resultado final
#print("Mejor encontrado: x =", x, "f(x) =", fx)
print(f"Mejor encontrado: x = {x:.4g}, f(x) = {fx:.4g}")

```

Mejor encontrado: x = -7.723, f(x) = 0.1301

5. 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?

```

In [ ]: import pygame
import sys
import random
import math
import time

# ===== Configuración inicial =====
WIDTH, HEIGHT = 300, 300
LINE_WIDTH = 5
CELL_SIZE = WIDTH // 3

WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
RED = (255, 0, 0)
BLUE = (0, 0, 255)

pygame.init()
FONT = pygame.font.SysFont(None, 40)
SMALL_FONT = pygame.font.SysFont(None, 30)

# ===== Funciones básicas del Ta-te-ti =====
def draw_board(screen, board):
    screen.fill(WHITE)
    for i in range(1, 3):
        pygame.draw.line(screen, BLACK, (0, CELL_SIZE*i), (WIDTH, CELL_SIZE*i), LINE_WIDTH)
        pygame.draw.line(screen, BLACK, (CELL_SIZE*i, 0), (CELL_SIZE*i, HEIGHT), LINE_WIDTH)

    for i in range(9):
        row, col = divmod(i, 3)
        x = col * CELL_SIZE + CELL_SIZE // 2
        y = row * CELL_SIZE + CELL_SIZE // 2
        if board[i] == "X":
            text = FONT.render("X", True, BLUE)
            screen.blit(text, text.get_rect(center=(x, y)))
        elif board[i] == "O":
            text = FONT.render("O", True, RED)
            screen.blit(text, text.get_rect(center=(x, y)))

    pygame.display.flip()

def check_winner(board, player):
    win_states = [
        [0,1,2],[3,4,5],[6,7,8],
        [0,3,6],[1,4,7],[2,5,8],

```

```

        [0,4,8],[2,4,6]
    ]
    return any(all(board[pos] == player for pos in line) for line in win_states)

def is_full(board):
    return all(cell != " " for cell in board)

def enhanced_evaluate(board):
    # Evaluación mejorada
    if check_winner(board, "O"):
        return 1000 # victoria IA
    if check_winner(board, "X"):
        return -1000 # derrota IA
    if is_full(board):
        return 0 # empate

    score = 0
    lines = [
        [0,1,2],[3,4,5],[6,7,8], # filas
        [0,3,6],[1,4,7],[2,5,8], # columnas
        [0,4,8],[2,4,6]          # diagonales
    ]

    for line in lines:
        values = [board[i] for i in line]

        # Jugada ganadora inminente
        if values.count("O") == 2 and values.count(" ") == 1:
            score += 100 # Mayor prioridad a movimientos ganadores
        if values.count("X") == 2 and values.count(" ") == 1:
            score -= 90 # Bloquear al oponente es crucial

        # Posiciones intermedias
        if values.count("O") == 1 and values.count(" ") == 2:
            score += 5
        if values.count("X") == 1 and values.count(" ") == 2:
            score -= 4

    # Bonificación por centro (la posición más fuerte)
    if board[4] == "O":
        score += 50
    elif board[4] == "X":
        score -= 75

    # Bonificación por esquinas
    for i in [0, 2, 6, 8]:
        if board[i] == "O":
            score += 7
        elif board[i] == "X":
            score -= 5

    return score

def neighbors(board, player):
    neigh = []
    for i in range(9):
        if board[i] == " ":
            new_board = board[:]
            new_board[i] = player
            neigh.append((new_board, i))
    return neigh

# ===== Recocido Simulado Mejorado =====

```

```

def simulated_annealing(board, player, T_init=10, alpha=0.95, steps=500):
    current_board = board[:]
    current_eval = enhanced_evaluate(current_board)

    # Mejor estado global encontrado
    best_board = current_board[:]
    best_eval = current_eval
    best_move = None

    T = T_init

    for step in range(steps):
        # Generar vecinos (movimientos posibles)
        neigs = neighbors(current_board, player)
        if not neigs:
            break

        # Seleccionar un vecino aleatorio
        new_board, move_idx = random.choice(neigs)
        new_eval = enhanced_evaluate(new_board)

        # Calcular la diferencia de evaluación
        delta = new_eval - current_eval

        # Criterio de aceptación
        if delta > 0:
            # Siempre aceptar mejoras
            current_board, current_eval = new_board, new_eval
            # Actualizar el mejor global si es necesario
            if new_eval > best_eval:
                best_board, best_eval = new_board[:], new_eval
                best_move = move_idx
        else:
            # Aceptar empeoras con probabilidad basada en temperatura
            probability = math.exp(delta / T) if T > 0 else 0
            if random.random() < probability:
                current_board, current_eval = new_board, new_eval

        # Enfriar la temperatura
        T = max(T * alpha, 0.01)

    # Si encontramos un movimiento bueno, usarlo
    if best_move is not None:
        return best_move

    # Si no se encontró un movimiento, elegir aleatoriamente
    empty_cells = [i for i in range(9) if board[i] == " "]
    if empty_cells:
        return random.choice(empty_cells)

    return -1 # No hay movimientos posibles

# ===== Juego principal =====
def play_game(T_init, alpha=0.95, steps=500): # Corregido: usar steps=500 consistentemente
    screen = pygame.display.set_mode((WIDTH, HEIGHT))
    pygame.display.set_caption(f"Ta-te-ti - Recocido Simulado (T={T_init}, Steps={steps})")

    board = [" "] * 9
    turn = "X" # Jugador empieza
    running = True
    game_over = False
    winner = None

```

```

while running:
    # Dibujar información de temperatura
    screen.fill(WHITE)
    info_text = SMALL_FONT.render(f"Temperatura: {T_init}", True, BLACK)
    screen.blit(info_text, (10, 10))
    steps_text = SMALL_FONT.render(f"Steps: {steps}", True, BLACK)
    screen.blit(steps_text, (10, 40))

    draw_board(screen, board)

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

        if not game_over and turn == "X" and event.type == pygame.MOUSEBUTTONDOWN:
            x, y = event.pos
            col = x // CELL_SIZE
            row = y // CELL_SIZE
            move = row * 3 + col

            if 0 <= move < 9 and board[move] == " ":
                board[move] = "X"

                if check_winner(board, "X"):
                    game_over = True
                    winner = "Humano (X)"
                elif is_full(board):
                    game_over = True
                    winner = "Empate"
                else:
                    turn = "O"

    # Turno de la IA
    if not game_over and turn == "O":
        pygame.time.delay(500) # Pausa para ver el turno

        move = simulated_annealing(board, "O", T_init, alpha, steps)

        if move != -1: # Si hay movimientos válidos
            board[move] = "O"

            if check_winner(board, "O"):
                game_over = True
                winner = "IA (O)"
            elif is_full(board):
                game_over = True
                winner = "Empate"
            else:
                turn = "X"
        else:
            # No hay movimientos posibles
            game_over = True
            winner = "Empate"

    # Mostrar resultado final
    if game_over:
        draw_board(screen, board)
        pygame.time.delay(1000)

    # Pantalla de resultado
    overlay = pygame.Surface((WIDTH, HEIGHT), pygame.SRCALPHA)
    overlay.fill((255, 255, 255, 200))
    screen.blit(overlay, (0, 0))

```

```

        if winner == "Empate":
            msg = FONT.render("¡Empate!", True, BLACK)
        else:
            msg = FONT.render(f"¡{winner} gana!", True, BLACK)

        screen.blit(msg, msg.get_rect(center=(WIDTH//2, HEIGHT//2 - 30)))

    restart_msg = SMALL_FONT.render("Clic para jugar otra vez", True, BLACK)
    screen.blit(restart_msg, restart_msg.get_rect(center=(WIDTH//2, HEIGHT//2 + 30)))

    pygame.display.flip()

    # Esperar clic para reiniciar
    waiting_for_click = True
    while waiting_for_click and running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            waiting_for_click = False
            if event.type == pygame.MOUSEBUTTONDOWN:
                waiting_for_click = False
        return True # Indicar que se debe reiniciar

    pygame.quit()
    return False

# ===== Función principal =====
def main():
    # Configuración de parámetros
    temperatures = [1, 10, 100] # Diferentes temperaturas a probar
    current_temp_index = 0
    alpha = 0.95 # Factor de enfriamiento
    steps = 500 # Número de iteraciones (ahora consistentemente 500)

    restart = True
    while restart:
        try:
            # Solicitar temperatura al usuario
            T_init = float(input(f"Temperatura inicial ({temperatures[current_temp_index]})"))
            if T_init <= 0:
                print("La temperatura debe ser positiva. Usando valor por defecto.")
                T_init = temperatures[current_temp_index]
        except ValueError:
            print("Entrada no válida. Usando valor por defecto.")
            T_init = temperatures[current_temp_index]

        # Jugar con la temperatura seleccionada
        restart = play_game(T_init, alpha, steps)

    # Cambiar a la siguiente temperatura para la próxima partida
    if restart:
        current_temp_index = (current_temp_index + 1) % len(temperatures)
        print(f"\nPróxima partida con temperatura: {temperatures[current_temp_index]}")
        print("Observaciones:")
        if temperatures[current_temp_index] < 10:
            print("- Temperatura baja: la IA será más conservadora y predecible")
        else:
            print("- Temperatura alta: la IA explorará más movimientos, pudiendo ser más ")

if __name__ == "__main__":
    main()

```

Para una temperatura alta hay una mayor exploración de movimientos, por lo que la IA puede tomar decisiones más arriesgadas, teniendo una mayor variabilidad pero cometer más errores. Con una menor temperatura la IA es más conservadora y predecible, pero tiene una menor exploración, con lo que puede quedarse con óptimos locales. Con una temperatura media se obtiene un balance entre exploración y explotación. Combina movimientos seguros con ocasionalmente obtener

6. Diseñe e implemente un algoritmo genético para cargar una grúa con  $n = 10$  cajas que puede soportar un peso máximo  $C = 1000 \text{ kg}$ . 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 cuáles no.

Esta representación corresponde a un Individuo con el que trabajará el algoritmo.

6.2 A continuación, genere una Población que contenga un número  $N$  de individuos (se recomienda elegir un número par).

Es necesario crear un control que verifique que ninguno de los individuos supere el peso límite.

6.3 Cree ahora una función que permita evaluar la Idoneidad de cada individuo y seleccione  $N/2$  parejas usando el método de la ruleta.

6.4 Por último, Cruce las parejas elegidas, aplique un mecanismo de Mutación y verifique que los individuos de la nueva población no superen el límite de peso.

6.5 Realice este proceso iterativamente hasta que se cumpla el mecanismo de detención de su elección y muestre el mejor individuo obtenido junto con el peso y el precio que alcanza.

In [ ]:

```
import random
import numpy as np
import matplotlib.pyplot as plt

# ===== DATOS DEL PROBLEMA =====
# Lista de pesos de las 10 cajas (kg)
weights = [300, 200, 450, 145, 664, 90, 150, 355, 401, 395]
# Lista de precios de las 10 cajas ($)
prices = [100, 50, 115, 25, 200, 30, 40, 100, 100, 100]
# Capacidad máxima que soporta la grúa (kg)
MAX_CAPACITY = 1000
# Número total de cajas
N_BOXES = len(weights)

# ===== PARÁMETROS DEL ALGORITMO GENÉTICO =====
```

```

POPULATION_SIZE = 20    # Tamaño de la población (número de individuos, debe ser par)
MUTATION_RATE = 0.1     # Probabilidad de mutación de cada gen
MAX_GENERATIONS = 100   # Número máximo de generaciones
ELITISM_COUNT = 2        # Número de mejores individuos que pasan directamente (elitismo)

# ===== REPRESENTACIÓN DE UN INDIVIDUO =====
def create_individual():
    """
    Crea un individuo aleatorio representado como una lista de 0s y 1s:
    - 1: caja incluida en la grúa
    - 0: caja no incluida
    Asegura que el individuo no supere el límite de peso.
    """
    individual = [random.randint(0, 1) for _ in range(N_BOXES)]

    # Si el individuo supera la capacidad, se "repara" quitando cajas
    while calculate_weight(individual) > MAX_CAPACITY:
        for i in range(N_BOXES):
            if individual[i] == 1 and random.random() < 0.5:
                individual[i] = 0
            if calculate_weight(individual) <= MAX_CAPACITY:
                break
    return individual

def calculate_weight(individual):                                         #Devuelve
    return sum(weights[i] for i in range(N_BOXES) if individual[i] == 1)

def calculate_price(individual):
    """Devuelve el precio total de las cajas seleccionadas."""
    return sum(prices[i] for i in range(N_BOXES) if individual[i] == 1)

# ===== POBLACIÓN INICIAL =====
def create_population():
    """Genera una población inicial de individuos válidos."""
    return [create_individual() for _ in range(POPULATION_SIZE)]

# ===== FUNCIÓN DE FITNESS =====
def evaluate_fitness(individual):
    """
    Calcula la idoneidad (fitness) del individuo:
    - Si el peso excede el límite → fitness = 0
    - Si no excede → fitness = precio total
    """
    total_weight = calculate_weight(individual)
    if total_weight > MAX_CAPACITY:
        return 0
    return calculate_price(individual)

# ===== SELECCIÓN POR RULETA =====
def select_parents(population, fitnesses):
    """
    Selección por ruleta:
    - La probabilidad de cada individuo es proporcional a su fitness.
    - Si todos tienen fitness 0, se eligen al azar.
    """
    total_fitness = sum(fitnesses)
    if total_fitness == 0:
        return random.sample(population, 2) # todos igual de malos

    # Probabilidades proporcionales al fitness
    probabilities = [f/total_fitness for f in fitnesses]

    # Elegir dos padres

```

```

parents = []
for _ in range(2):
    spin = random.random()
    cumulative_prob = 0
    for i, prob in enumerate(probabilities):
        cumulative_prob += prob
        if spin <= cumulative_prob:
            parents.append(population[i])
            break
return parents

# ===== OPERADOR DE CRUCE =====
def crossover(parent1, parent2):
    """
    Realiza cruce de un punto:
    - Se elige un punto aleatorio.
    - Se combinan las partes de los dos padres para formar dos hijos.
    """
    point = random.randint(1, N_BOXES - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

# ===== OPERADOR DE MUTACIÓN =====
def mutate(individual):
    """
    Aplica mutación: con probabilidad MUTATION_RATE,
    invierte el valor de un gen (0→1 o 1→0).
    Después, repara al individuo si supera la capacidad.
    """
    for i in range(N_BOXES):
        if random.random() < MUTATION_RATE:
            individual[i] = 1 - individual[i] # flip bit

    # Reparación: si excede peso, eliminar cajas hasta que sea válido
    total_weight = calculate_weight(individual)
    while total_weight > MAX_CAPACITY:
        loaded_boxes = [i for i in range(N_BOXES) if individual[i] == 1]
        if not loaded_boxes:
            break
        box_to_remove = random.choice(loaded_boxes)
        individual[box_to_remove] = 0
        total_weight = calculate_weight(individual)

    return individual

# ===== ALGORITMO GENÉTICO PRINCIPAL =====
def genetic_algorithm():
    """Ejecuta el algoritmo genético completo."""
    # 1. Crear población inicial
    population = create_population()
    best_individual = None
    best_fitness = 0
    fitness_history = []

    # 2. Iterar sobre generaciones
    for generation in range(MAX_GENERATIONS):
        # Evaluar fitness
        fitnesses = [evaluate_fitness(ind) for ind in population]

        # Guardar mejor individuo de la generación
        current_best_fitness = max(fitnesses)
        current_best_index = fitnesses.index(current_best_fitness)

```

```

        if current_best_fitness > best_fitness:
            best_fitness = current_best_fitness
            best_individual = population[current_best_index].copy()

    fitness_history.append(best_fitness)

    if generation % 10 == 0:
        print(f"Generación {generation}: Mejor fitness = {best_fitness}")

# 3. Crear nueva población
new_population = []

# Elitismo: mantener los mejores sin cambios
sorted_indices = np.argsort(fitnesses)[::-1]
for i in range(ELITISM_COUNT):
    new_population.append(population[sorted_indices[i]].copy())

# Rellenar el resto con cruce y mutación
while len(new_population) < POPULATION_SIZE:
    parents = select_parents(population, fitnesses)
    child1, child2 = crossover(parents[0], parents[1])
    new_population.append(mutate(child1))
    if len(new_population) < POPULATION_SIZE:
        new_population.append(mutate(child2))

population = new_population

return best_individual, best_fitness, fitness_history

# ===== EJECUCIÓN Y RESULTADOS =====
if __name__ == "__main__":
    print("Ejecutando algoritmo genético para el problema de la grúa...")
    print(f"Capacidad máxima: {MAX_CAPACITY} kg")
    print(f"Número de cajas: {N_BOXES}")
    print(f"Pesos: {weights}")
    print(f"Precios: {prices}\n")

# Ejecutar el algoritmo
best_solution, best_value, history = genetic_algorithm()

# Mostrar mejor solución
print("\n" + "="*50)
print("MEJOR SOLUCIÓN ENCONTRADA:")
print("="*50)

total_weight = calculate_weight(best_solution)
total_price = calculate_price(best_solution)

print(f"Cajas seleccionadas: {[i+1 for i in range(N_BOXES) if best_solution[i] == 1]}")
print(f"Peso total: {total_weight} kg")
print(f"Precio total: ${total_price}\n")

print("\nDetalle de cajas seleccionadas:")
for i in range(N_BOXES):
    if best_solution[i] == 1:
        print(f"Caja {i+1}: Peso = {weights[i]} kg, Precio = ${prices[i]}")

# Graficar evolución del fitness
plt.figure(figsize=(10, 6))
plt.plot(history, linewidth=2)
plt.title('Evolución del Mejor Fitness por Generación')
plt.xlabel('Generación')

```

```
plt.ylabel('Fitness (Precio Total $)')
plt.grid(True, alpha=0.3)
plt.show()
```

## 6.1 Representación del Individuo

Cada individuo se representa como una lista de 0s y 1s, donde 1 indica que la caja correspondiente está cargada en la grúa.

## 6.2 Población Inicial

Se genera una población inicial de individuos aleatorios, asegurando que ninguno supere el peso límite de 1000 kg.

## 6.3 Función de Evaluación y Selección

La función de fitness calcula el precio total de las cajas seleccionadas.

La selección de padres se realiza mediante el método de la ruleta, donde individuos con mayor fitness tienen mayor probabilidad de ser seleccionados.

## 6.4 Operadores Genéticos

Cruce: Se realiza un cruce de un punto entre dos padres para producir dos hijos.

Mutación: Cada gen (caja) tiene una probabilidad de mutar (cambiar de 0 a 1 o viceversa).

Reparación: Si después de la mutación un individuo excede la capacidad, se reparan quitando cajas aleatoriamente.

## 6.5 Mecanismo de Detención

El algoritmo se detiene después de un número máximo de generaciones (100), mostrando el mejor individuo encontrado junto con su peso y precio total.

El algoritmo incluye elitismo para preservar los mejores individuos entre generaciones y un gráfico que muestra la evolución del fitness a lo largo de las generaciones.

# 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