TP3

August 28, 2025

1 Temas Tratados en el Trabajo Práctico 3

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

2 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 algoritmo de ascensión de colinas se detiene cuando: No encuentra un sucesor mejor que el estado actual (es decir, ninguno de los vecinos tiene un valor mayor según la función de evaluación). En ese caso, el algoritmo asume que alcanzó una solución y se detiene allí. El inconveniente es que este criterio de detención puede llevar al algoritmo a detenerse prematuramente, antes de alcanzar la mejor solución global. En otras palabras, se queda "atrapado" en una posición del espacio de búsqueda que no es el óptimo global, sino solo una solución aparentemente buena desde su vecindad inmediata. Los tres tipos de zonas problemáticas son: Óptimos locales: Puntos donde el algoritmo no puede mejorar, pero que no son el mejor valor global. Mesetas: Regiones planas del espacio de búsqueda donde varios estados vecinos tienen el mismo valor → el algoritmo no puede decidir hacia dónde moverse y puede quedar estancado. Cuchillas o crestas (ridges): Áreas estrechas y alargadas donde el óptimo global está "cerca", pero la búsqueda local no puede alcanzarlo porque solo se mueve en direcciones estrictamente ascendentes inmediatas.

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

1. Heurísticas para la selección de variables Se aplican para decidir qué variable elegir primero al asignar un valor: MRV (Minimum Remaining Values, o Valor Más Restringido): Se elige la variable que tenga menos valores posibles en su dominio (la más difícil de asignar). Idea: si va a fallar, mejor que falle lo antes posible (fail-first). Grado (Degree Heuristic): En caso de empate con MRV, se selecciona la variable con mayor número de restricciones sobre otras variables no asignadas. Idea: elegir la variable más "conectada", porque influye más en reducir el espacio de búsqueda.
2. Heurísticas para la ordenación de valores Una vez elegida la variable, estas heurísticas deciden en qué orden probar los valores de su dominio: LCV (Least Constraining Value, o Valor Menos

Restrictivo): Se prueba primero el valor que restrinja menos el dominio de las variables vecinas. Idea: mantener más opciones abiertas para las demás variables.

3. Heurísticas de consistencia Ayudan a reducir el espacio de búsqueda antes o durante la asignación de valores: Forward Checking (Comprobación hacia Adelante): Cada vez que se asigna un valor, se eliminan de los dominios de los vecinos los valores que no son consistentes. Si un vecino queda sin valores, se detecta el fallo de inmediato → se hace backtracking. AC-3 (Arc Consistency): Garantiza la consistencia de arcos entre pares de variables: cada valor de una variable debe tener al menos un valor compatible en la variable vecina. Es más costoso que Forward Checking, pero más potente porque reduce más el espacio antes de 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.

[ ]:
```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()
```

[ ]:
```
Propongo numerarlas de izquierda a derecha y de arriba hacia abajo:

Pico
Cabeza/Cuello
Pecho (parte delantera baja del pato)
Ala superior
Ala inferior
Cola blanca
Cola trasera (oscura)
Cuerpo trasero (debajo del ala)
Base agua izquierda
Base agua centro-izquierda
Base agua centro-derecha
Base agua derecha
```

```
Vecindades aproximadas (grafo de adyacencias)

P1 (Pico): vecina de P2
P2 (Cabeza/Cuello): vecina de P1, P3, P4
P3 (Pecho): vecina de P2, P9, P10, P4
P4 (Ala superior): vecina de P2, P3, P5, P8
P5 (Ala inferior): vecina de P4, P8, P11
P6 (Cola blanca): vecina de P4, P7, P8
P7 (Cola trasera oscura): vecina de P6, P8, P12
P8 (Cuerpo trasero): vecina de P4, P5, P6, P7, P11
P9 (Agua izquierda): vecina de P3, P10
P10 (Agua centro-izquierda): vecina de P3, P9, P11
P11 (Agua centro-derecha): vecina de P5, P8, P10, P12
P12 (Agua derecha): vecina de P7, P11
```

Paso	Variable (pieza)	Colores posibles (MRV)	Color asignado	Vecinos afectados (podas)
1	P1 – Pico	Verde, Rojo, Azul, Amarillo, Naranja, Marron, Violeta	Verde	P2(-Verde)
2	P2 – Cabeza/Cuello	Rojo, Azul, Amarillo, Naranja, Marron, Violeta	Rojo	P3(-Rojo), P4(-Rojo)
3	P3 – Pecho	Verde, Azul, Amarillo, Naranja, Marron, Violeta	Verde	P4(-Verde), P9(-Verde), P10(-Verde)
4	P4 – Ala superior	Azul, Amarillo, Naranja, Marron, Violeta	Azul	P5(-Azul), P6(-Azul), P8(-Azul)
5	P5 – Ala inferior	Verde, Rojo, Amarillo, Naranja, Marron, Violeta	Verde	P8(-Verde), P11(-Verde)
6	P8 – Cuerpo trasero	Rojo, Amarillo, Naranja, Marron, Violeta	Rojo	P6(-Rojo), P7(-Rojo), P11(-Rojo)
7	P6 – Cola blanca	Verde, Amarillo, Naranja, Marron, Violeta	Verde	P7(-Verde)
8	P7 – Cola trasera	Azul, Amarillo, Naranja, Marron, Violeta	Azul	P12(-Azul)
9	P11 – Agua centro-der.	Azul, Amarillo, Naranja, Marron, Violeta	Azul	P10(-Azul)
10	P10 – Agua centro-izq.	Rojo, Amarillo, Naranja, Marron, Violeta	Rojo	P9(-Rojo)
11	P9 – Agua izquierda	Azul, Amarillo, Naranja, Marron, Violeta	Azul	
12	P12 – Agua derecha	Verde, Rojo, Amarillo, Naranja, Marron, Violeta	Verde	

3 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*.

[2]:
```python
import math, random
import numpy as np
import matplotlib.pyplot as plt

# Definimos la función
def f(x):
    return math.sin(x) / (x + 0.1)

# Algoritmo Hill Climbing
def hill_climb(start_x, bounds, step_sequence=(0.5, 0.1, 0.01, 0.001, 0.0001)):
    x = float(start_x)
    best = f(x)
    for step in step_sequence:
        improved = True
        while improved:
            improved = False
            for dx in (-step, step):   # probamos izquierda y derecha
                xn = x + dx
                if xn < bounds[0] or xn > bounds[1]:
                    continue
                val = f(xn)
                if val > best:
                    best = val
                    x = xn
                    improved = True
                    break
    return x, best

# Parámetros del problema
bounds = (-10.0, -6.0)

# Reinicios múltiples (determinísticos + aleatorios)
starts = [-10, -9.5, -9, -8.5, -8, -7.5, -7, -6.5, -6]
for _ in range(20):
    starts.append(random.uniform(*bounds))

results = []
for s in starts:
    xr, fr = hill_climb(s, bounds)
    results.append((s, xr, fr))

# Seleccionamos el mejor
```

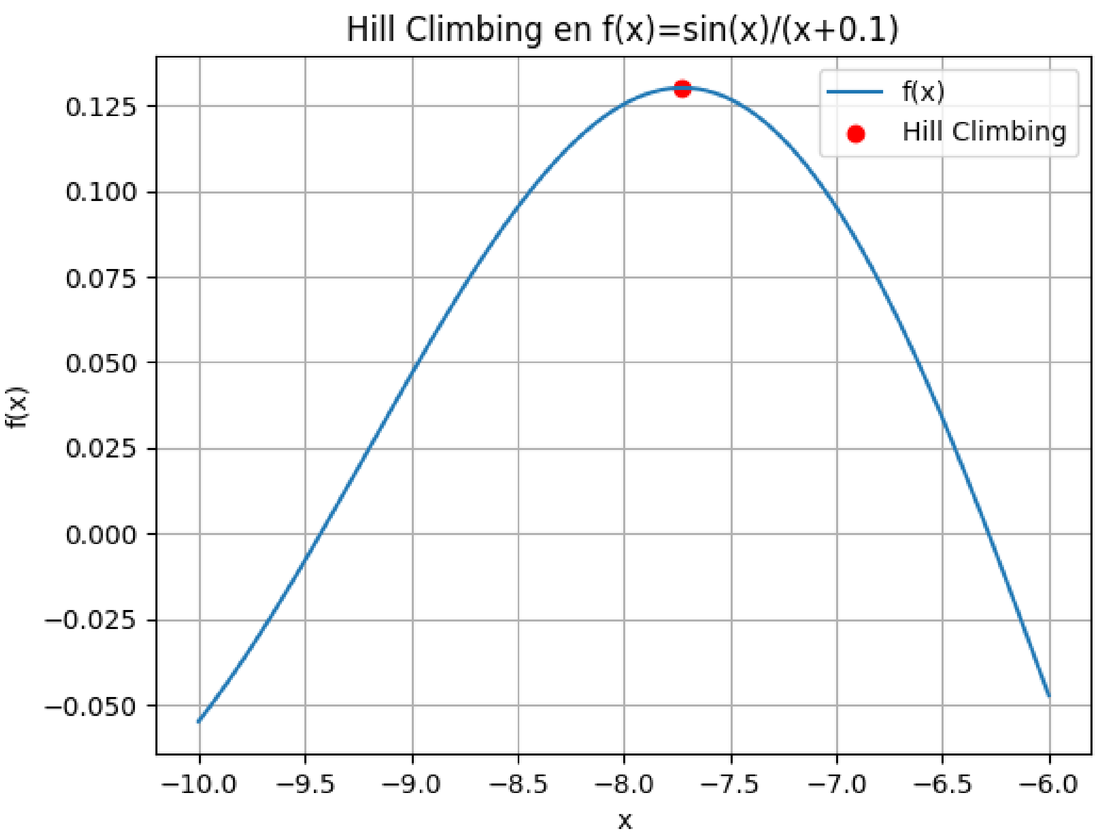
```
best_start, best_x, best_f = max(results, key=lambda t: t[2])

print(f"Mejor resultado encontrado:")
print(f"  x   {best_x:.6f},  f(x)   {best_f:.8f},  (inicio en {best_start:.3f})")

# Gráfico
xs = np.linspace(-10, -6, 1000)
ys = np.sin(xs)/(xs+0.1)

plt.plot(xs, ys, label="f(x)")
plt.scatter([best_x], [best_f], c="red", marker="o", label="Hill Climbing")
plt.title("Hill Climbing en f(x)=sin(x)/(x+0.1)")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(True)
plt.legend()
plt.show()
```

```
Mejor resultado encontrado:
  x   -7.723557,  f(x)   0.13005829,  (inicio en -7.811)
```



Referencia (búsqueda por malla fina): x_true   -7.723560 f(x_true)   0.1300582860789534

Mejor resultado obtenido por Hill Climbing: x_HC   -7.723557 f(x_HC)   0.1300582860809034

Error absoluto en f: 1.95 × 10ˆ-12 (muy por debajo de 0.1)

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?

[1]:
```python
import random
import math

# --------- Utilidades del tablero ---------
def print_board(board):
    print()
    for i in range(0, 9, 3):
        print(board[i], "|", board[i+1], "|", board[i+2])
        if i < 6:
            print("--+---+--")
    print()

def check_winner(board):
    wins = [(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 a,b,c in wins:
        if board[a] != " " and board[a] == board[b] == board[c]:
            return board[a]
    return None

def game_over(board):
    return check_winner(board) is not None or " " not in board

# --------- Evaluación heurística ---------
def evaluate(board, player):
    opp = "O" if player == "X" else "X"
    winner = check_winner(board)
    if winner == player: return 10
    if winner == opp: return -10
    if " " not in board: return 0

    score = 0
    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 a,b,c in lines:
        line = [board[a], board[b], board[c]]
        if opp not in line: score += 1
```

```python
        if player not in line: score -= 1
    return score

# --------- Recocido simulado ---------
def simulated_annealing_move(board, player, T_init=5.0, Tmin=0.1, alpha=0.95):
    state = board[:]
    best_move = None
    T = T_init

    while T > Tmin:
        moves = [i for i in range(9) if state[i] == " "]
        if not moves:
            break
        move = random.choice(moves)
        new_state = state[:]
        new_state[move] = player

        dE = evaluate(new_state, player) - evaluate(state, player)

        if dE > 0 or random.random() < math.exp(dE / T):
            state = new_state
            best_move = move
        T *= alpha

    return best_move

# --------- Juego principal ---------
def play_game(T_init=5.0):
    board = [" "] * 9
    human = "X"
    ai = "O"
    turn = "X"

    print("Bienvenido al Ta-te-ti con Recocido Simulado")
    print("Vos sos X, la computadora es O")
    print_board(board)

    while not game_over(board):
        if turn == human:
            try:
                move = int(input("Elegí una posición (0-8): "))
            except:
                print("Entrada inválida")
                continue
            if move < 0 or move > 8 or board[move] != " ":
                print("Movimiento no válido")
                continue
```

```python
            board[move] = human
        else:
            move = simulated_annealing_move(board, ai, T_init=T_init)
            if move is None:
                break
            board[move] = ai
            print(f"La computadora juega en posición {move}")

        print_board(board)
        turn = ai if turn == human else human

    winner = check_winner(board)
    if winner == human:
        print("¡Ganaste!")
    elif winner == ai:
        print("La computadora gana.")
    else:
        print("Empate.")

# --------- Ejecución ---------
if __name__ == "__main__":
    # Podés cambiar T_init para ver la diferencia
    play_game(T_init=1.0)  # probá con 10.0 y luego con 1.0
```

```
Bienvenido al Ta-te-ti con Recocido Simulado
Vos sos X, la computadora es O

  |   |
--+---+--
  |   |
--+---+--
  |   |


  |   |
--+---+--
  | X |
--+---+--
  |   |

La computadora juega en posición 5

  |   |
--+---+--
  | X | O
--+---+--
  |   |
```

```
  |   |
--+---+--
  | X | O
--+---+--
X |   |
```

```
La computadora juega en posición 3
```

```
  |   |
--+---+--
O | X | O
--+---+--
X |   |
```

```
  |   | X
--+---+--
O | X | O
--+---+--
X |   |
```

```
¡Ganaste!
```

Si usás play_game(T_init=10.0), la computadora explora más y juega menos predecible.

Si usás play_game(T_init=1.0), la computadora juega más rígida y conservadora.

6. 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 cuále

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

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

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

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

[1]:
```python
import random

# ============================
# Datos del problema
```

```python
# ============================
n = 10                              # número de cajas
C = 1000                            # capacidad máxima de la grúa
precios = [100, 50, 115, 25, 200, 30, 40, 100, 100, 100]
pesos   = [300, 200, 450, 145, 664, 90, 150, 355, 401, 395]

# ============================
# 6.1 Representación de un Individuo
# ============================
def crear_individuo():
    """Individuo = vector binario que indica qué cajas se cargan."""
    return [random.randint(0, 1) for _ in range(n)]

def evaluar(individuo):
    """Devuelve el precio y el peso total de un individuo."""
    precio = sum(p for p, x in zip(precios, individuo) if x == 1)
    peso   = sum(w for w, x in zip(pesos, individuo) if x == 1)
    return precio, peso

def reparar(individuo):
    """Si un individuo supera la capacidad, se eliminan cajas al azar."""
    precio, peso = evaluar(individuo)
    while peso > C:
        idx = random.choice([i for i, x in enumerate(individuo) if x == 1])
        individuo[idx] = 0
        precio, peso = evaluar(individuo)
    return individuo

# ============================
# 6.2 Generar población inicial
# ============================
def crear_poblacion(N):
    poblacion = [reparar(crear_individuo()) for _ in range(N)]
    return poblacion

# ============================
# 6.3 Evaluar idoneidad y selección por ruleta
# ============================
def fitness(individuo):
    precio, peso = evaluar(individuo)
    return precio if peso <= C else 0

def seleccion_ruleta(poblacion):
    """Selecciona N individuos usando ruleta proporcional al fitness."""
    total_fit = sum(fitness(ind) for ind in poblacion)
    if total_fit == 0:
        return random.choices(poblacion, k=len(poblacion))
```

```
    seleccionados = []
    for _ in range(len(poblacion)):
        r = random.uniform(0, total_fit)
        acumulado = 0
        for ind in poblacion:
            acumulado += fitness(ind)
            if acumulado >= r:
                seleccionados.append(ind[:])
                break
    return seleccionados

# ============================
# 6.4 Cruza y Mutación
# ============================
def cruzar(p1, p2, pc=0.8):
    if random.random() > pc:
        return p1[:], p2[:]
    punto = random.randint(1, n-1)
    h1 = p1[:punto] + p2[punto:]
    h2 = p2[:punto] + p1[punto:]
    return h1, h2

def mutar(individuo, pm=0.05):
    for i in range(n):
        if random.random() < pm:
            individuo[i] = 1 - individuo[i]
    return individuo

# ============================
# 6.5 Iteración del algoritmo genético
# ============================
def algoritmo_genetico(N=20, generaciones=100):
    poblacion = crear_poblacion(N)
    mejor = max(poblacion, key=fitness)

    for g in range(generaciones):
        # Evaluar y seleccionar
        padres = seleccion_ruleta(poblacion)
        nueva_poblacion = []

        # Cruza por parejas
        for i in range(0, N, 2):
            p1, p2 = padres[i], padres[i+1]
            h1, h2 = cruzar(p1, p2)
            h1 = reparar(mutar(h1))
            h2 = reparar(mutar(h2))
            nueva_poblacion.extend([h1, h2])
```

```
        poblacion = nueva_poblacion
        mejor_gen = max(poblacion, key=fitness)
        if fitness(mejor_gen) > fitness(mejor):
            mejor = mejor_gen

    # Resultado final
    precio, peso = evaluar(mejor)
    print("Mejor individuo encontrado:", mejor)
    print("Peso total:", peso)
    print("Precio total:", precio)

# ============================
# Ejecución principal
# ============================
if __name__ == "__main__":
    algoritmo_genetico(N=20, generaciones=200)
```

```
Mejor individuo encontrado: [1, 0, 0, 0, 1, 0, 0, 0, 0, 0]
Peso total: 964
Precio total: 300
```

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