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 algoritmo de Ascensión de Colinas se detiene cuando no encuentra un vecino mejor que el estado actual. El problema es que puede quedar atrapado en regiones del espacio de estados que no llevan a la mejor solución global. Estas áreas problemáticas se llaman óptimos locales, mesetas y terrazas.

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

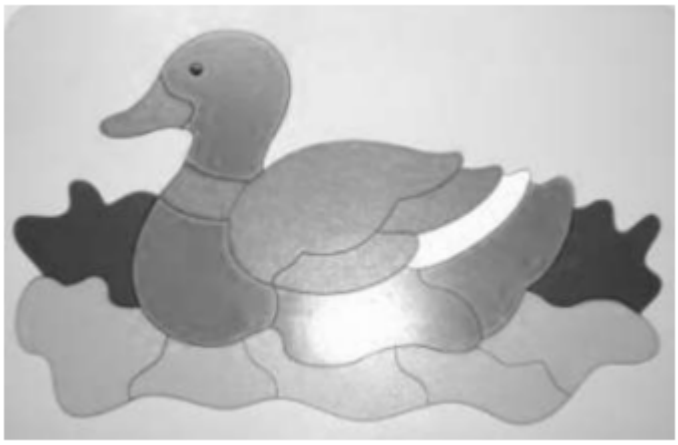
Las heurísticas más comunes en PSR son:

- Variable más restringida (MRV): elegir la variable con menos valores posibles en su dominio.
- Grado: elegir la variable que participa en más restricciones con otras no asignadas.
- Valor menos restrictivo (LRV): asignar el valor que deje más opciones abiertas para las demás variables.
- Además, se usan técnicas de consistencia (nodo, arco, camino) para reducir el espacio de búsqueda antes de asignar.

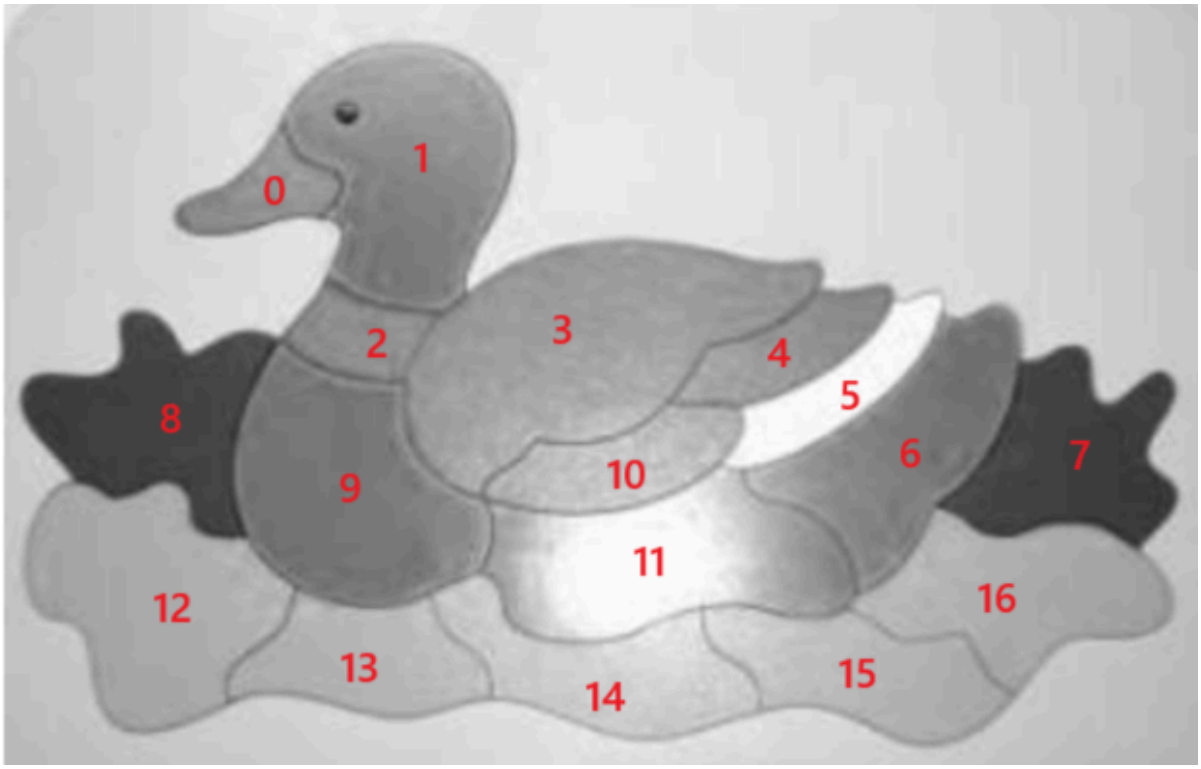
1. Se desea colorear el rompecabezas mostrado en la imagen con 7 colores distintos de manera que ninguna pieza tenga el mismo color que sus vecinas. Realice en una tabla el proceso de una búsqueda con Comprobación hacia Adelante empleando una heurística del Valor más Restringido.

In [92]:
```
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_ZnrMWOscQGb2MZ0Cdb3R"

# 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()
```



Enumeramos las piezas



Hemos hecho dos tablas, pero que expresan el mismo procedimiento

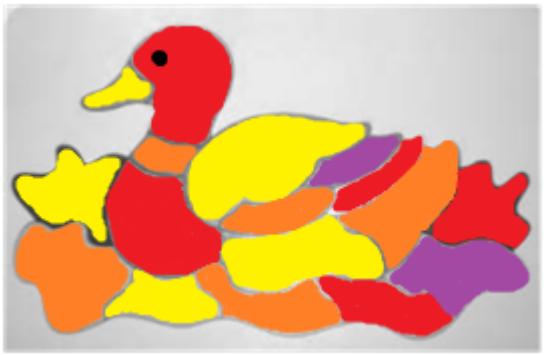
- R: Rojo
- A: Amarillo
- N: Naranja
- V: Violeta
- G: Gris
- M: Marrón
- C: Celeste

paso	celda	posibles	color asignado	dominios modificados
1	9	R, A, N, V, G, M, C	**R**	2:-R, 3:-R, 8:-R, 11:-R, 12:-R, 13:-R, 14:-R
2	11	A, N, V, G, M, C	**A**	5:-A, 6:-A, 10:-A, 14:-A, 15:-A
3	14	N, V, G, M, C	**N**	13:-N, 15:-N
4	13	A, V, G, M, C	**A**	12:-A
5	12	N, V, G, M, C	**N**	8:-N
6	15	R, V, G, M, C	**R**	16:-R
7	8	A, V, G, M, C	**A**	
8	3	A, N, V, G, M, C	**A**	2:-A, 4:-A
9	2	N, V, G, M, C	**N**	1:-N
10	5	R, N, V, G, M, C	**R**	4:-R, 6:-R, 10:-R
11	6	N, V, G, M, C	**N**	7:-N, 16:-N
12	10	N, V, G, M, C	**N**	4:-N
13	4	V, G, M, C	**V**	
14	16	A, V, G, M, C	**V**	7:-V
15	7	R, A, G, M, C	**R**	
16	1	R, A, V, G, M, C	**R**	0:-R
17	0	A, N, V, G, M, C	**A**	

En esta tabla las columnas son las celdas del rompecabezas. A medida que se avanza, se van eliminando posibilidades y asignando colores.

Paso	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0 (inicio)	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C
1 (9=R)	R,A,N,V,G,M,C	R,A,N,V,G,M,C	A,N,V,G,M,C	A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	R,A,N,V,G,M,C	A,N,V,G,M,C	R	R,A,N,V,G,M,C	A,N,V,G,M,C	A,N,V,G,M,C	A,N,V,G,M,C	A,N,V,G,M,C	A,N,V,G,M,C	R,A,N,V,G,M,C
2 (11=A)	R,A,N,V,G,M,C	R,A,N,V,G,M,C	A,N,V,G,M,C	A,N,V,G,M,C	R,A,N,V,G,M,C	R,N,V,G,M,C	R,N,V,G,M,C	R,A,N,V,G,M,C	A,N,V,G,M,C	R	R,N,V,G,M,C	A	A,N,V,G,M,C	A,N,V,G,M,C	N,V,G,M,C	R,N,V,G,M,C	R,A,N,V,G,M,C
3 (14=N)	R,A,N,V,G,M,C	R,A,N,V,G,M,C	A,N,V,G,M,C	A,N,V,G,M,C	R,A,N,V,G,M,C	R,N,V,G,M,C	R,N,V,G,M,C	R,A,N,V,G,M,C	A,N,V,G,M,C	R	R,N,V,G,M,C	A	A,N,V,G,M,C	A,V,G,M,C	N	R,V,G,M,C	R,A,N,V,G,M,C
4 (13=A)	R,A,N,V,G,M,C	R,A,N,V,G,M,C	A,N,V,G,M,C	A,N,V,G,M,C	R,A,N,V,G,M,C	R,N,V,G,M,C	R,N,V,G,M,C	R,A,N,V,G,M,C	N,V,G,M,C	R	R,N,V,G,M,C	A	N,V,G,M,C	A	N	R,V,G,M,C	R,A,N,V,G,M,C
5 (12=N)	R,A,N,V,G,M,C	R,A,N,V,G,M,C	A,N,V,G,M,C	A,N,V,G,M,C	R,A,N,V,G,M,C	R,N,V,G,M,C	R,N,V,G,M,C	R,A,N,V,G,M,C	A,V,G,M,C	R	R,N,V,G,M,C	A	N	A	N	R,V,G,M,C	R,A,N,V,G,M,C
6 (15=R)	R,A,N,V,G,M,C	R,A,N,V,G,M,C	A,N,V,G,M,C	A,N,V,G,M,C	R,A,N,V,G,M,C	R,N,V,G,M,C	R,N,V,G,M,C	R,A,N,V,G,M,C	A,V,G,M,C	R	R,N,V,G,M,C	A	N	A	N	R	A,N,V,G,M,C
7 (8=A)	R,A,N,V,G,M,C	R,A,N,V,G,M,C	A,N,V,G,M,C	A,N,V,G,M,C	R,A,N,V,G,M,C	R,N,V,G,M,C	R,N,V,G,M,C	R,A,N,V,G,M,C	A	R	R,N,V,G,M,C	A	N	A	N	R	A,N,V,G,M,C
8 (3=A)	R,A,N,V,G,M,C	R,A,N,V,G,M,C	N,V,G,M,C	A	R,N,V,G,M,C	R,N,V,G,M,C	R,N,V,G,M,C	R,A,N,V,G,M,C	A	R	R,N,V,G,M,C	A	N	A	N	R	A,N,V,G,M,C
9 (2=N)	R,A,N,V,G,M,C	R,A,V,G,M,C	N	A	R,N,V,G,M,C	R,N,V,G,M,C	R,N,V,G,M,C	R,A,N,V,G,M,C	A	R	R,N,V,G,M,C	A	N	A	N	R	A,N,V,G,M,C
10 (5=R)	R,A,N,V,G,M,C	R,A,V,G,M,C	N	A	N,V,G,M,C	R	N,V,G,M,C	R,A,N,V,G,M,C	A	R	N,V,G,M,C	A	N	A	N	R	A,N,V,G,M,C
11 (6=N)	R,A,N,V,G,M,C	R,A,V,G,M,C	N	A	N,V,G,M,C	R	N	R,A,G,M,C	A	R	N,V,G,M,C	A	N	A	N	R	A,V,G,M,C
12 (10=N)	R,A,N,V,G,M,C	R,A,V,G,M,C	N	A	V,G,M,C	R	N	R,A,G,M,C	A	R	N	A	N	A	N	R	A,V,G,M,C
13 (4=V)	R,A,N,V,G,M,C	R,A,V,G,M,C	N	A	V	R	N	R,A,G,M,C	A	R	N	A	N	A	N	R	A,V,G,M,C
14 (16=V)	R,A,N,V,G,M,C	R,A,V,G,M,C	N	A	V	R	N	R,A,G,M,C	A	R	N	A	N	A	N	R	V
15 (7=R)	R,A,N,V,G,M,C	R,A,V,G,M,C	N	A	V	R	N	R	A	R	N	A	N	A	N	R	V
16 (1=R)	R,A,N,V,G,M,C	R	N	A	V	R	N	R	A	R	N	A	N	A	N	R	V
17 (0=A)	A	R	N	A	V	R	N	R	A	R	N	A	N	A	N	R	V

Pato coloreado



Ejercicios de Implementación

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*.

In [ ]:

```
import numpy as np

f = lambda x: np.sin(x) / (x+0.1)
intervalo = (-10,-6)

df = lambda x, dx=1e-5: (f(x+dx) - f(x-dx)) / (2*dx)
```

```python
alpha = 0.05

# Arranco en un punto aleatorio del intervalo
inicio = np.random.uniform(-10, -6)

x_actual = inicio
df_actual = df(inicio)

print("Iniciando en x=", inicio)

while abs(df_actual) >= 1e-5:
    x_actual += alpha * df_actual
    df_actual = df(x_actual)

    if x_actual > intervalo[1]:
        print(f"Máximo en x={intervalo[1]} -- f({intervalo[1]})={f(intervalo[1])}")
        break

    if x_actual < intervalo[0]:
        print(f"Máximo en x={intervalo[0]} -- f({intervalo[0]})={f(intervalo[0])}")
        break

else:
    print(f"Máximo en x={x_actual} -- f({x_actual})={f(x_actual)}")
```

```
Iniciando en x= -6.461013844727939
Máximo en x=-7.723477232173654 -- f(-7.723477232173654)=0.13005828569946465
```

1. 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?

Realizado en archivo "ej5.py"

1. 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.

</table>

```
    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 [3]:

```
import numpy as np

# Datos del problema

precios_x_elemento=np.array([100,20,115,25,200,30,40,100,100,100])
pesos_x_elemento=np.array([300,200,450,145,664,90,150,355,401,395])
npoblacion = 1000
peso_maximo=1000

#Variables auxiliares

poblacion = list()
padres = list()
poblacion_nueva = list()
precios_poblacion = list()
pesos_poblacion = list()
i=0

#Incializacion de la poblacion

while i<npoblacion:
    individuo = np.random.randint(0, 2, size=10)
    peso = np.sum(individuo*pesos_x_elemento)
    if peso>peso_maximo:
        continue
    else:

precios_poblacion.append(np.sum(individuo*precios_x_elemento))
        pesos_poblacion.append(peso)
        poblacion.append(individuo)
        i+=1

# Algoritmo genetico, 1000 iteraciones con poblacion de 100
individuos da como resultado Precio total 300 . Aumentar la cantidad
de inddividuos
# mejoró mucho el resultado.

#variables del algoritmo genetico
p_mutacion=0.3
generaciones=100
i=0

while i<generaciones:
    poblacion_nueva = list()
    padres = list()
    #calculo de ruleta de probabilidades y seleccion de poblacion
nueva
    numeros = np.arange(npoblacion)
    probabilidad = precios_poblacion/np.sum(precios_poblacion)
    idxs = np.random.choice(numeros, size=npoblacion, replace=True,
p=probabilidad)
    padres = [poblacion[i] for i in idxs]

    #seleccion de padres y cruza
    for j in range(0, npoblacion, 2):
        idxs2 = np.random.choice(numeros, size=2)
```

```
        padre1 = padres[idxs2[0]]
        padre2 = padres[idxs2[1]]
        punto_cruce = np.random.randint(1, 9)
        hijo1 = np.concatenate((padre1[:punto_cruce],
padre2[punto_cruce:]))
        hijo2 = np.concatenate((padre2[:punto_cruce],
padre1[punto_cruce:]))

        peso_hijo1 = np.sum(hijo1*pesos_x_elemento)
        peso_hijo2 = np.sum(hijo2*pesos_x_elemento)
        #mutacion generica y elitismo
        if np.random.rand() < p_mutacion:
            punto_mutacion = np.random.randint(0, 10)
            if hijo1[punto_mutacion] == 1:
                hijo1[punto_mutacion] = 0
            peso_hijo1 = np.sum(hijo1*pesos_x_elemento)
        if np.random.rand() < p_mutacion:
            punto_mutacion = np.random.randint(0, 10)
            if hijo2[punto_mutacion] == 1:
                hijo2[punto_mutacion] = 0
            peso_hijo2 = np.sum(hijo2*pesos_x_elemento)
        #mutacion solo si el peso del hijo supera el maximo
        while peso_hijo1 > peso_maximo:
            punto_mutacion = np.random.randint(0, 10)
            hijo1[punto_mutacion] = 1 - hijo1[punto_mutacion]
            peso_hijo1 = np.sum(hijo1*pesos_x_elemento)

        while peso_hijo2 > peso_maximo:
            punto_mutacion = np.random.randint(0, 10)
            hijo2[punto_mutacion] = 1 - hijo2[punto_mutacion]
            peso_hijo2 = np.sum(hijo2*pesos_x_elemento)
        poblacion_nueva.append(hijo1)
        poblacion_nueva.append(hijo2)
    poblacion=poblacion_nueva
    precios_poblacion = [np.sum(individuo*precios_x_elemento) for
individuo in poblacion]
    pesos_poblacion = [np.sum(individuo*pesos_x_elemento) for
individuo in poblacion]


    i+=1
#reordenamiento final de la poblacion
if len(poblacion_nueva) == npoblacion:
        sorted_indices = np.argsort(precios_poblacion)[-npoblacion:]
        poblacion = [poblacion[i] for i in sorted_indices]
        precios_poblacion = [precios_poblacion[i] for i in
sorted_indices]
        pesos_poblacion = [pesos_poblacion[i] for i in
sorted_indices]
print("Mejor individuo:", poblacion[npoblacion-1])
print("Precio total:", precios_poblacion[npoblacion-1])
print("Peso total:", pesos_poblacion[npoblacion-1])
```

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

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

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