

✓ AG3 - Actividad Guiada 3

Nombre: Kenny Pizarro

Link:

https://colab.research.google.com/drive/1_pgnhZ3UfNIzfZleG5h_hnkNYHQobHFH#scrollTo=ta1tvzVvskPC

Github: <https://github.com/xxxxx/AlgoritmosOptimizacion>

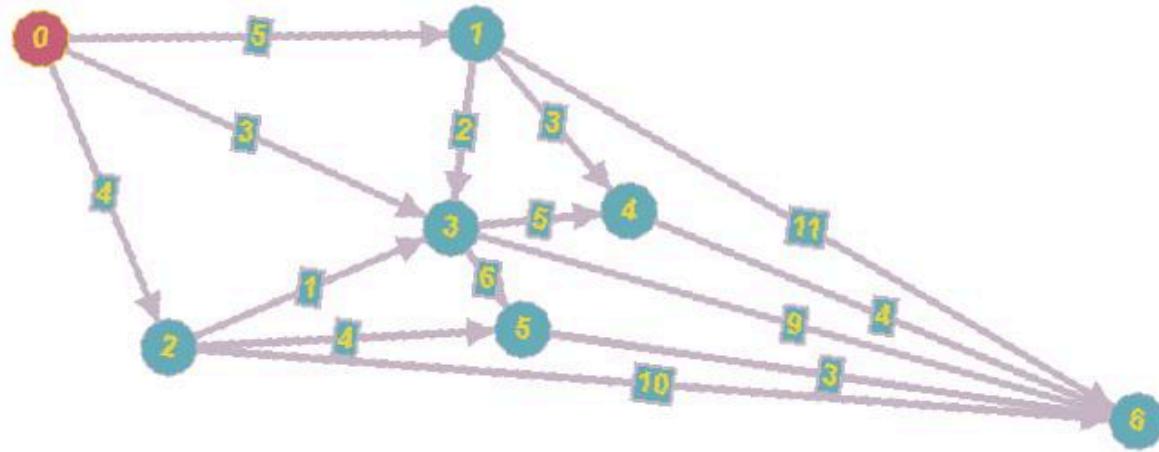
```
import math
```

✓ Programación Dinámica. Viaje por el río

- **Definición:** Es posible dividir el problema en subproblemas más pequeños, guardando las soluciones para ser utilizadas más adelante.
- **Características** que permiten identificar problemas aplicables:
 - Es posible almacenar soluciones de los subproblemas para ser utilizados más adelante
 - Debe verificar el principio de optimalidad de Bellman: "en una secuencia óptima de decisiones, toda sub-secuencia también es óptima" (*)
 - La necesidad de guardar la información acerca de las soluciones parciales unido a la recursividad provoca la necesidad de preocuparnos por la complejidad espacial (cuantos recursos de espacio usaremos)

Problema

En un río hay **n** embarcaderos y debemos desplazarnos río abajo desde un embarcadero a otro. Cada embarcadero tiene precios diferentes para ir de un embarcadero a otro situado más abajo. Para ir del embarcadero i al j , puede ocurrir que sea más barato hacer un trasbordo por un embarcadero intermedio k . El problema consiste en determinar la combinación más barata.



*Consideramos una tabla TARIFAS(i,j) para almacenar todos los precios que nos ofrecen los embarcaderos.

*Si no es posible ir desde i a j daremos un valor alto para garantizar que ese trayecto no se va a elegir en la ruta óptima(modelado habitual para restricciones)

```
#Viaje por el río - Programación dinámica
```

```
#####
#####
```

```
TARIFAS = [
[0,5,4,3,float("inf"),999,999], #desde nodo 0
[999,0,999,2,3,999,11], #desde nodo 1
[999,999, 0,1,999,4,10], #desde nodo 2
[999,999,999, 0,5,6,9],
[999,999, 999,999,0,999,4],
[999,999, 999,999,999,0,3],
[999,999,999,999,999,999,0]
]
```

```
#999 se puede sustituir por float("inf") del modulo math
TARIFAS
```

```
[[0, 5, 4, 3, inf, 999, 999],
[999, 0, 999, 2, 3, 999, 11],
[999, 999, 0, 1, 999, 4, 10],
[999, 999, 999, 0, 5, 6, 9],
[999, 999, 999, 999, 0, 999, 4],
[999, 999, 999, 999, 999, 0, 3],
[999, 999, 999, 999, 999, 999, 0]]
```

```
#Calculo de la matriz de PRECIOS y RUTAS
```

```
# PRECIOS - contiene la matriz del mejor precio para ir de un nodo a otro
```

```
# RUTAS - contiene los nodos intermedios para ir de un nodo a otro
```

```
#####
#####
```

```
def Precios(TARIFAS):
```

```
#####
#####
```

```
#Total de Nodos
N = len(TARIFAS[0])

#Inicialización de la tabla de precios
PRECIOS = [ [9999]*N for i in [9999]*N] #n x n
RUTA = [ [""]*N for i in [""]*N]

#Se recorren todos los nodos con dos bucles(origen - destino)
# para ir construyendo la matriz de PRECIOS
for i in range(N-1):
    for j in range(i+1, N):
        MIN = TARIFAS[i][j]
        RUTA[i][j] = i

        for k in range(i, j):
            if PRECIOS[i][k] + TARIFAS[k][j] < MIN:
                MIN = min(MIN, PRECIOS[i][k] + TARIFAS[k][j] )
                RUTA[i][j] = k
                PRECIOS[i][j] = MIN

return PRECIOS,RUTA
```

```
PRECIOS,RUTA = Precios(TARIFAS)
#print(PRECIOS[0][6])

print("PRECIOS")
for i in range(len(TARIFAS)):
    print(PRECIOS[i])

print("\nRUTA")
for i in range(len(TARIFAS)):
    print(RUTA[i])
```

PRECIOS

```
[9999, 5, 4, 3, 8, 8, 11]
[9999, 9999, 999, 2, 3, 8, 7]
[9999, 9999, 9999, 1, 6, 4, 7]
[9999, 9999, 9999, 9999, 5, 6, 9]
[9999, 9999, 9999, 9999, 9999, 999, 4]
[9999, 9999, 9999, 9999, 9999, 9999, 3]
[9999, 9999, 9999, 9999, 9999, 9999, 9999]
```

RUTA

```
['', 0, 0, 0, 1, 2, 5]
['', 1, 1, 1, 3, 4]
['', 2, 3, 2, 5]
[', , , , 3, 3, 3]
[', , , , , 4, 4]
[', , , , , , 5]
[', , , , , , ]
```

```
#Calculo de la ruta usando la matriz RUTA
def calcular_ruta(RUTA, desde, hasta):
    if desde == RUTA[desde][hasta]:
        #if desde == hasta:
        #print("Ir a :" + str(desde))
        return desde
```

```

else:
    return str(calcular_ruta(RUTA, desde, RUTA[desde][hasta])) + ',' + str(RUTA[desde][hasta])
print("\nLa ruta es:")
calcular_ruta(RUTA, 0,6)

```

La ruta es:
'0,2,5'

Haz doble clic (o ingresa) para editar

▼ Problema de Asignacion de tarea

```

#Asignacion de tareas - Ramificación y Poda
#####
# T A R E A
# A
# G
# E
# N
# T
# E

COSTES=[[11,12,18,40],
         [14,15,13,22],
         [11,17,19,23],
         [17,14,20,28]]

```

```

#Calculo del valor de una solucion parcial
def valor(S,COSTES):
    VALOR = 0
    for i in range(len(S)):
        VALOR += COSTES[S[i]][i]
    return VALOR

```

valor((3,2,),COSTES)

34

```

#Coste inferior para soluciones parciales
# (1,3,) Se asigna la tarea 1 al agente 0 y la tarea 3 al agente 1

def CI(S,COSTES):
    VALOR = 0
    #Valores establecidos
    for i in range(len(S)):
        VALOR += COSTES[i][S[i]]

```

```
#Estimacion
for i in range( len(S), len(COSTES) ):
    VALOR += min( [ COSTES[j][i] for j in range(len(S), len(COSTES)) ] )
return VALOR

def CS(S,COSTES):
    VALOR = 0
    #Valores establecidos
    for i in range(len(S)):
        VALOR += COSTES[i][S[i]]

    #Estimacion
    for i in range( len(S), len(COSTES) ):
        VALOR += max( [ COSTES[j][i] for j in range(len(S), len(COSTES)) ] )
    return VALOR

CI((0,1),COSTES)
```

68

```
#Genera tantos hijos como posibilidades haya para la siguiente elemento de la tupla
#(0,) -> (0,1), (0,2), (0,3)
def crear_hijos(NODO, N):
    HIJOS = []
    for i in range(N):
        if i not in NODO:
            HIJOS.append({'s':NODO +(i,)})
    return HIJOS
```

```
crear_hijos((0,) , 4)
[{'s': (0, 1)}, {'s': (0, 2)}, {'s': (0, 3)}]
```

```
def ramificacion_y_poda(COSTES):
    #Construccion iterativa de soluciones(arbol). En cada etapa asignamos un agente(ramas).
    #Nodos del grafo { s:(1,2),CI:3,CS:5 }
    #print(COSTES)
    DIMENSION = len(COSTES)
    MEJOR_SOLUCION=tuple( i for i in range(len(COSTES)) )
    CotaSup = valor(MEJOR_SOLUCION,COSTES)
    #print("Cota Superior:", CotaSup)

    NODOS=[]
    NODOS.append({'s':(), 'ci':CI(),COSTES} )

    iteracion = 0

    while( len(NODOS) > 0):
        iteracion +=1

        nodo_prometedor = [ min(NODOS, key=lambda x:x['ci']) ][0]['s']
        #print("Nodo prometedor:", nodo_prometedor)

        #Ramificacion
        #Se generan los hijos
```

```

HIJOS =[ {'s':x['s'], 'ci':CI(x['s'], COSTES) } for x in crear_hijos(nodo_prometido)

#Revisamos la cota superior y nos quedamos con la mejor solucion si llegamos a una so
NODO_FINAL = [x for x in HIJOS if len(x['s']) == DIMENSION ]
if len(NODO_FINAL ) >0:
    #print("\n*****Soluciones:", [x for x in HIJOS if len(x['s']) == DIMENSION ] )
    if NODO_FINAL[0]['ci'] < CotaSup:
        CotaSup = NODO_FINAL[0]['ci']
        MEJOR_SOLUCION = NODO_FINAL

#Poda
HIJOS = [x for x in HIJOS if x['ci'] < CotaSup ]

#Añadimos los hijos
NODOS.extend(HIJOS)

#Eliminamos el nodo ramificado
NODOS = [ x for x in NODOS if x['s'] != nodo_prometedor ]

print("La solucion final es:" ,MEJOR_SOLUCION , " en " , iteracion , " iteraciones" , "


ramificacion_y_poda(COSTES)

La solucion final es: [{s': (1, 2, 0, 3), 'ci': 64}] en 10 iteraciones para dimension

```

▼ Descenso del gradiente

```

import math                      #Funciones matematicas
import matplotlib.pyplot as plt   #Generacion de graficos (otra opcion seaborn)
import numpy as np                #Tratamiento matriz N-dimensionales y otras (fundamental
# import scipy as sc

import random

```

Vamos a buscar el minimo de la funcion paraboloida :

$$f(x) = x^2 + y^2$$

Obviamente se encuentra en $(x,y)=(0,0)$ pero probaremos como llegamos a él a través del descenso del gradiante.

```

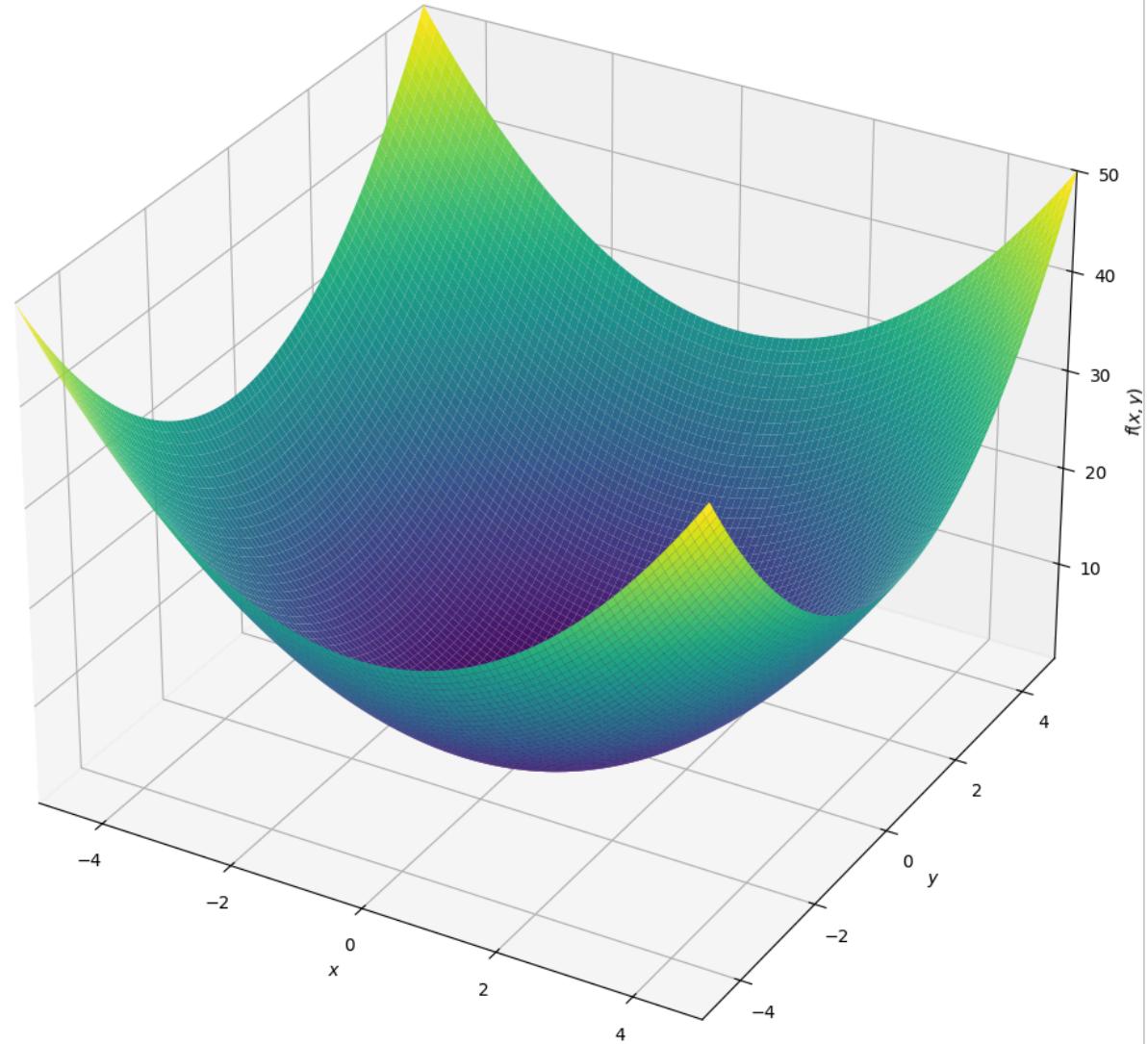
#Definimos la funcion
#Paraboloida
f = lambda X:      X[0]**2 + X[1]**2      #Funcion
df = lambda X: [2*X[0] , 2*X[1]]           #Gradiente

df([1,2])
[2, 4]

```

```
from sympy import symbols
from sympy.plotting import plot
from sympy.plotting import plot3d
x,y = symbols('x y')
plot3d(x**2 + y**2,
       (x,-5,5),(y,-5,5),
       title='x**2 + y**2',
       size=(10,10))
```

x**2 + y**2



<sympy.plotting.backends.matplotlibbackend.matplotlib.MatplotlibBackend at 0x7d9d39c99ee0>

```
#Prepara los datos para dibujar mapa de niveles de Z
resolucion = 100
rango=5.5

X=np.linspace(-rango,rango,resolucion)
```

```

Y=np.linspace(-rango,rango,resolucion)
Z=np.zeros((resolucion,resolucion))
for ix,x in enumerate(X):
    for iy,y in enumerate(Y):
        Z[iy,ix] = f([x,y])

#Pinta el mapa de niveles de Z
plt.contourf(X,Y,Z,resolucion)
plt.colorbar()

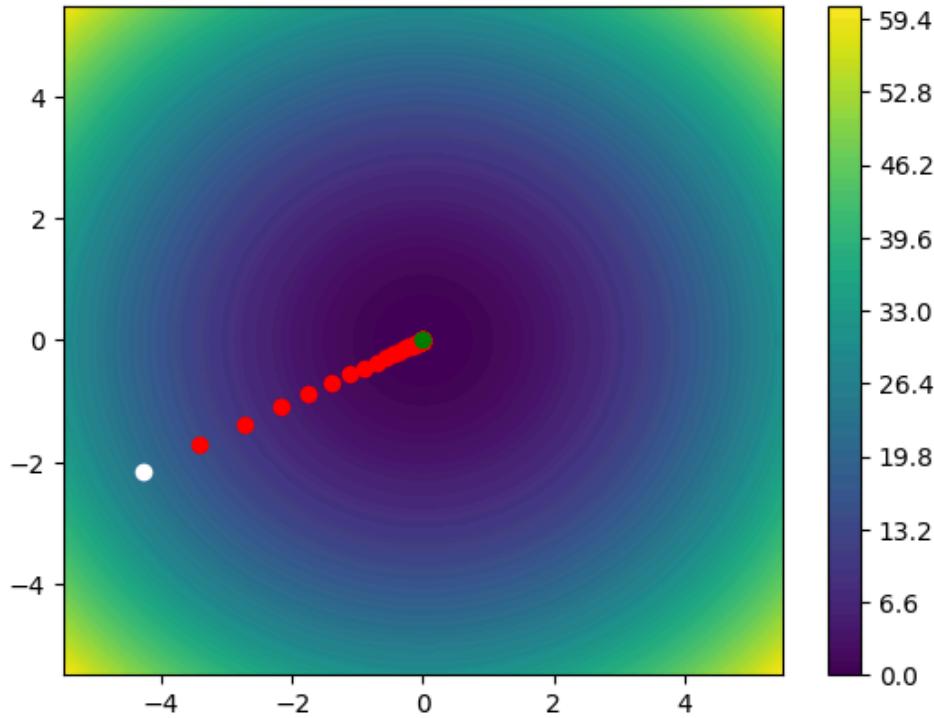
#Generamos un punto aleatorio inicial y pintamos de blanco
P=[random.uniform(-5,5),random.uniform(-5,5) ]
plt.plot(P[0],P[1],"o",c="white")

#Tasa de aprendizaje. Fija. Sería más efectivo reducirlo a medida que nos acercamos.
TA=.1

#Iteraciones:50
for _ in range(50):
    grad = df(P)
    #print(P,grad)
    P[0],P[1] = P[0] - TA*grad[0] , P[1] - TA*grad[1]
    plt.plot(P[0],P[1],"o",c="red")

#Dibujamos el punto final y pintamos de verde
plt.plot(P[0],P[1],"o",c="green")
plt.show()
print("Solucion:" , P , f(P))

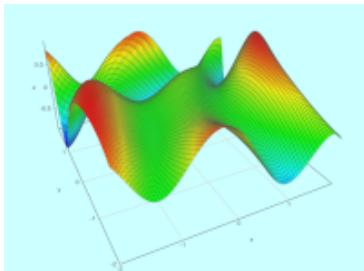
```



Solución: [-6.1016090938825914e-05, -3.060498942089196e-05] 4.659628730907982e-09

¿Te atreves a optimizar la función?:

$$f(x) = \sin(1/2 * x^2 - 1/4 * y^2 + 3) * \cos(2 * x + 1 - e^y)$$



```
#Definimos la función
f= lambda X: math.sin(1/2 * X[0]**2 - 1/4 * X[1]**2 + 3) *math.cos(2*X[0] + 1 - math.exp(
```

```
#Gradiente de la función
df = lambda X: [
    math.cos(0.5*X[0]**2 - 0.25*X[1]**2 + 3)
    * math.cos(2*X[0] + 1 - math.exp(X[1])) * X[0]
    -
    2 * math.sin(0.5*X[0]**2 - 0.25*X[1]**2 + 3)
    * math.sin(2*X[0] + 1 - math.exp(X[1])),

    -0.5 * X[1]
    * math.cos(0.5*X[0]**2 - 0.25*X[1]**2 + 3)
    * math.cos(2*X[0] + 1 - math.exp(X[1]))
    +
    math.exp(X[1])
    * math.sin(0.5*X[0]**2 - 0.25*X[1]**2 + 3)
    * math.sin(2*X[0] + 1 - math.exp(X[1]))
]
```

```
#Descenso de Gradiente
resolucion = 100
rango = 5.5

X = np.linspace(-rango, rango, resolucion)
Y = np.linspace(-rango, rango, resolucion)
Z = np.zeros((resolucion, resolucion))

for ix, x in enumerate(X):
    for iy, y in enumerate(Y):
        Z[iy, ix] = f([x, y])

plt.contourf(X, Y, Z, resolucion)
plt.colorbar()

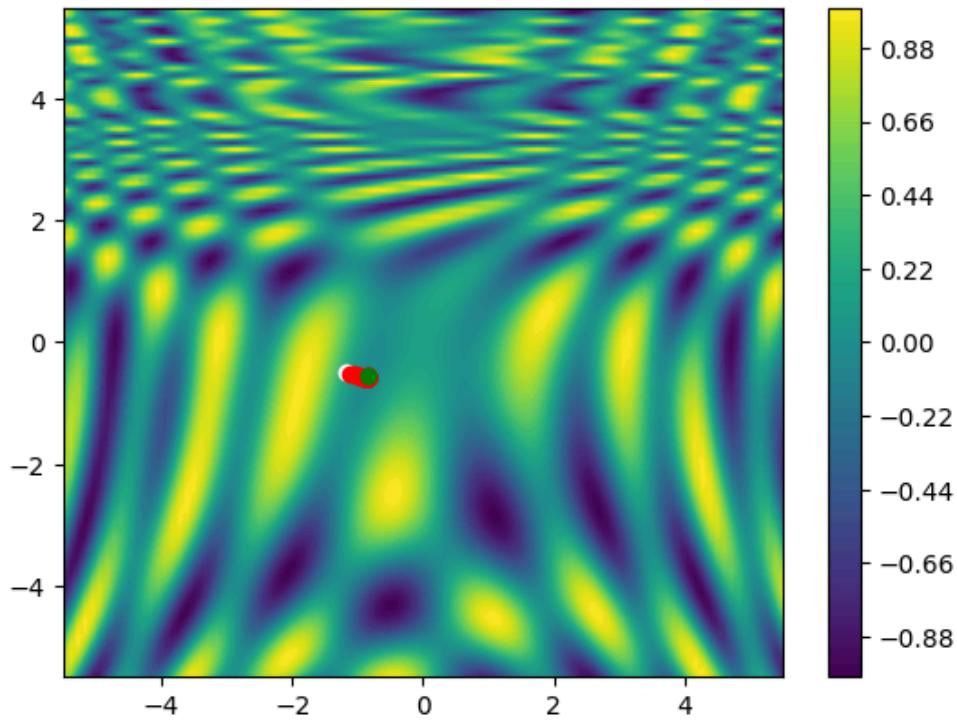
# Punto inicial aleatorio
P = [random.uniform(-2, 2), random.uniform(-2, 2)]
plt.plot(P[0], P[1], "o", c="white")

# Tasa de aprendizaje
TA = 0.05
```

```
# Iteraciones
for _ in range(50):
    grad = df(P)
    P[0] = P[0] - TA * grad[0]
    P[1] = P[1] - TA * grad[1]
    plt.plot(P[0], P[1], "o", c="red")

# Punto final
plt.plot(P[0], P[1], "o", c="green")
plt.show()

print("Solucion:", P, f(P))
```



Solucion: [-0.8462630971228065, -0.564705440886932] -0.04155794323993174