AG3 - Actividad Guiada 3

Nombre: Briam Sebastian Ramos Guevara
Link: https://colab.research.google.com/drive/1raIRTpMU2PRT8IrXZpXrfV_H8DgNhIE2#scrollTo=ta1tvzVvsKPC
Github: https://github.com/bramosguevara/AlgoritmosdeOptimizacion

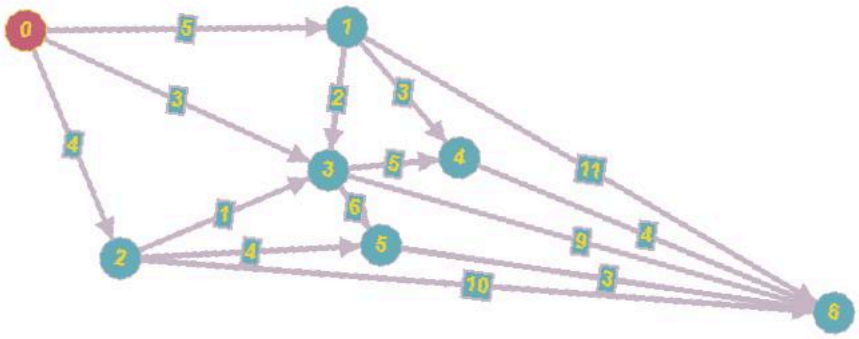
```
import math
```

Programación Dinámica. Viaje por el rio

- **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 optima 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 rio - 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'
```

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 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_prometedor, DIMENSION) ]

    #Revisamos la cota superior y nos quedamos con la mejor solucion si llegamos a una solucion final
    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" , " para dimension: " ,DIMENSION  )



ramificacion_y_poda(COSTES)
```

```
La solucion final es: [{'s': (1, 2, 0, 3), 'ci': 64}]  en  10  iteraciones  para dimension:  4
```

Descenso del gradiente

```
import math                      #Funciones matematicas
import matplotlib.pyplot as plt  #Generacion de gráficos (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 paraboloide :

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

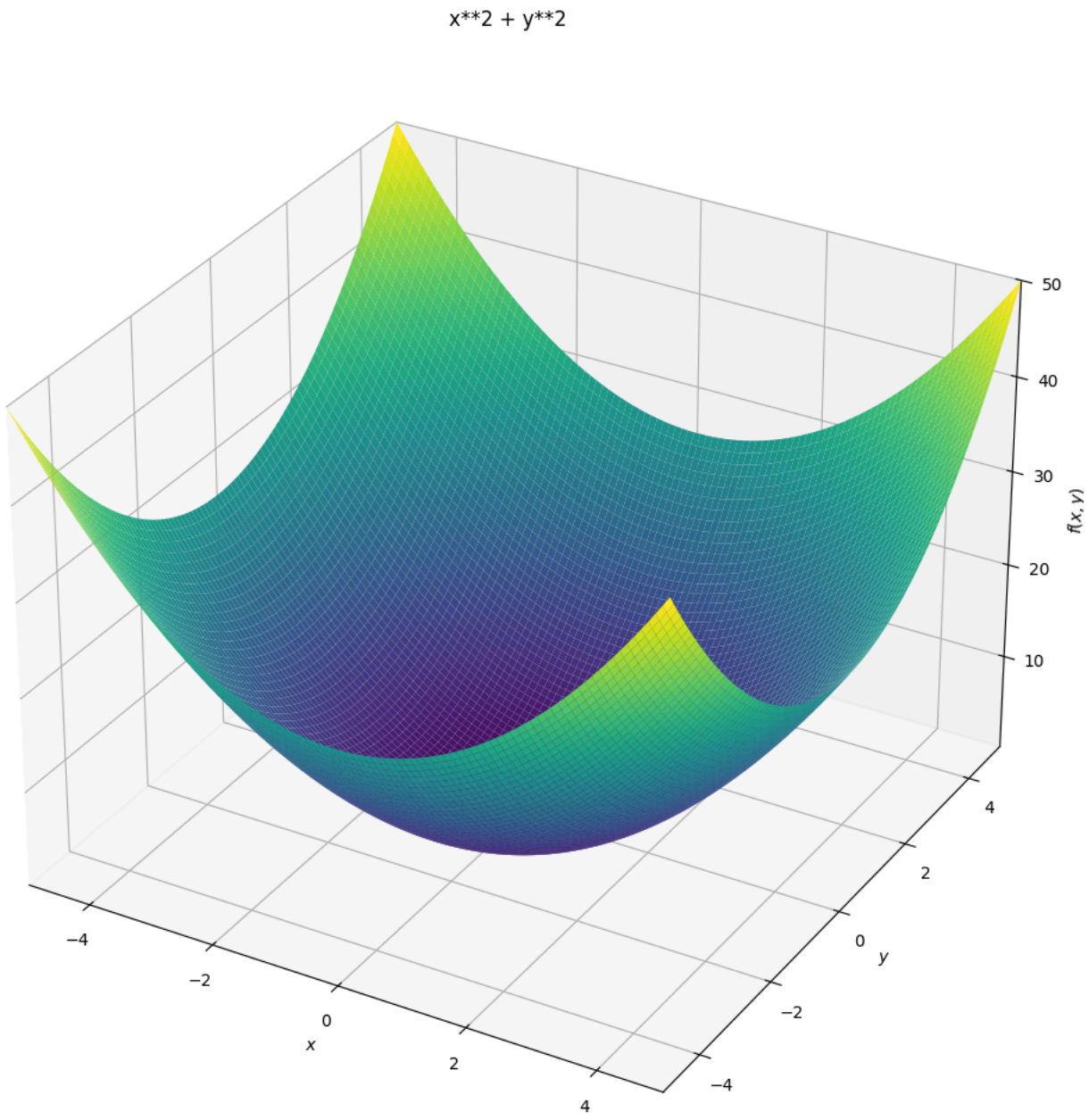
Obviamente se encuentra en (x,y)=(0,0) pero probaremos como llegamos a él a través del descenso del gradiante.

```
#Definimos la funcion
#Paraboloide
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]
```

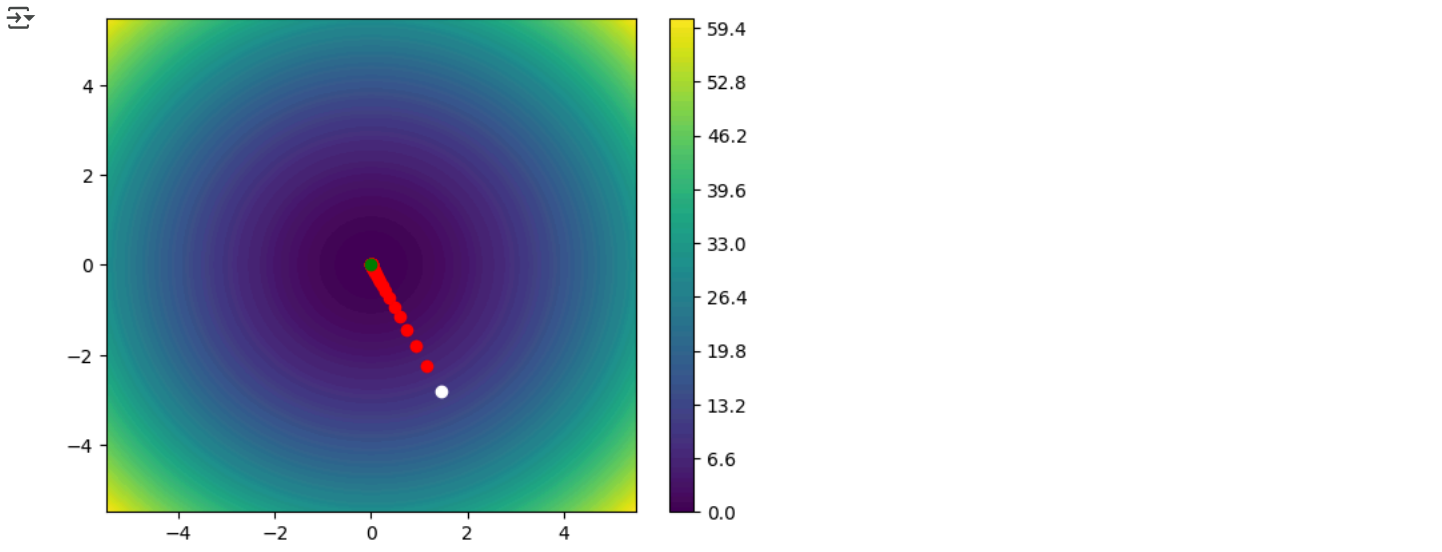
```
from sympy import symbols
from sympy.plotting import plot
from sympy.plotting import plot3d
x,y = symbols('x y')
plot3d(x**2 + y**2,
       (x,-5,5),(y,-5,5),
       title='x**2 + y**2',
       size=(10,10))
```




```
<sympy.plotting.backends.matplotlibbackend.matplotlib.MatplotlibBackend at 0x7d19fb3d3a90>
```

```
#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):
```

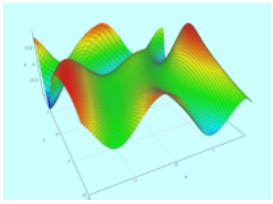
```
  for iy,y in enumerate(Y):
    Z[iy,ix] = f([x,y])

#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))
```




```
Solucion: [2.062193958821061 5e-05, -4.029841146339029 6e-05] 2.049226358852514 7e-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 funcion
f= lambda X: math.sin(1/2 * X[0]**2 - 1/4 * X[1]**2 + 3) *math.cos(2*X[0] + 1 - math.exp(X[1]) )
```

Practica Individual

Ramificación y Poda

- Generar matrices con valores aleatorios de mayores dimensiones (5,6,7,...) y ejecutar ambos algoritmos.

- ¿A partir de que dimensión el algoritmo por fuerza bruta deja de ser una opción?
- ¿Hay algún valor de la dimensión a partir de la cual el algoritmo de ramificación y poda también deja de ser una opción válida?

Algoritmo por fuerza bruta con generacion de matrices de mayores dimensiones:

```
import random
from itertools import permutations

def matriz_aleatoria(dimension, max_valor=100):

    return [[random.randint(1, max_valor) for _ in range(dimension)] for _ in range(dimension)]

def calcular_costo(solucion, costos):

    return sum(costos[i][solucion[i]] for i in range(len(solucion)))

def fuerza_bruta(COSTES):

    dimension = len(COSTES)
    todas_permutaciones = permutations(range(dimension))

    mejor_costo = float('inf')
    mejor_solucion = None

    for perm in todas_permutaciones:
        costo_actual = calcular_costo(perm, COSTES)
        if costo_actual < mejor_costo:
            mejor_costo = costo_actual
            mejor_solucion = perm

    print(f"Mejor solución: {mejor_solucion} con costo: {mejor_costo}")
    return mejor_solucion, mejor_costo

def probar_fuerza_bruta():
    dimensiones = [5, 6, 7, 8]  # Dimensiones para probar

    for dim in dimensiones:
        print(f"\nDimensión: {dim}x{dim}")
        matriz_costos = matriz_aleatoria(dim)

        print("Matriz de costos:")
        for fila in matriz_costos:
            print(fila)

        try:
            fuerza_bruta(matriz_costos)
        except MemoryError:
            print(f"Dimensión {dim} es demasiado grande para fuerza bruta.")
        except Exception as e:
            print(f"Error: {e}")

# Ejecutamos las pruebas
if __name__ == "__main__":
    probar_fuerza_bruta()
```

```
Dimensión: 5x5
Matriz de costos:
[36, 74, 43, 72, 58]
[4, 93, 33, 84, 3]
[1, 28, 76, 17, 89]
[21, 65, 74, 51, 70]
[29, 29, 39, 13, 81]
Mejor solución: (2, 4, 1, 0, 3) con costo: 108

Dimensión: 6x6
Matriz de costos:
[58, 40, 86, 22, 84, 87]
[30, 26, 80, 50, 34, 43]
[77, 20, 56, 33, 97, 68]
[54, 17, 14, 36, 13, 62]
[43, 43, 38, 98, 6, 45]
[55, 100, 87, 11, 59, 36]
Mejor solución: (3, 0, 1, 2, 4, 5) con costo: 128

Dimensión: 7x7
Matriz de costos:
```

```
[24, 91, 3, 29, 2, 8, 68]
[97, 12, 63, 19, 77, 99, 81]
[65, 41, 100, 48, 94, 9, 13]
[34, 15, 24, 7, 89, 38, 46]
[57, 80, 53, 86, 29, 62, 36]
[23, 65, 81, 53, 32, 15, 29]
[78, 49, 97, 76, 4, 33, 18]
Mejor solución: (2, 1, 5, 3, 6, 0, 4) con costo: 94

Dimensión: 8x8
Matriz de costos:
[85, 78, 35, 10, 50, 9, 4, 73]
[27, 94, 15, 25, 49, 12, 29, 98]
[78, 57, 75, 34, 34, 87, 42, 42]
[41, 51, 40, 85, 56, 13, 77, 32]
[74, 30, 51, 64, 85, 76, 56, 41]
[87, 76, 77, 58, 23, 23, 72, 77]
[82, 41, 13, 92, 99, 87, 98, 82]
[18, 1, 35, 69, 55, 14, 81, 82]
Mejor solución: (6, 0, 3, 5, 7, 4, 2, 1) con costo: 156
```

```
import random

def matriz_aleatoria(dimension, max_valor=100):

    return [[random.randint(1, max_valor) for _ in range(dimension)] for _ in range(dimension)]

# Función de prueba que ejecuta el algoritmo con diferentes dimensiones.
def dim_ramificacion_y_poda():
    dimensiones = [5, 6, 7, 8]

    for dim in dimensiones:
        print(f"\nDimensión: {dim}x{dim}")
        matriz_costos = matriz_aleatoria(dim)
        for fila in matriz_costos:
            print(fila)  # Muestra la matriz para verificar

        try:
            ramificacion_y_poda(matriz_costos)
        except Exception as e:
            print(f"Error al ejecutar con dimensión {dim}: {e}")

# Si tus funciones valor, CI, y crear_hijos aún no están definidas, añade sus definiciones aquí.
def valor(solucion, costos):

    return sum(costos[i][solucion[i]] for i in range(len(solucion)))

def CI(parcial, costos):

    asignados = set(parcial)
    return sum(min(costos[i]) for i in range(len(costos)) if i not in asignados)

def crear_hijos(nodo, dimension):

    hijos = []
    utilizados = set(nodo)
    for i in range(dimension):
        if i not in utilizados:
            hijos.append({'s': nodo + (i,)})
    return hijos

def ramificacion_y_poda(COSTES):

    DIMENSION = len(COSTES)
    MEJOR_SOLUCION = tuple(i for i in range(len(COSTES)))
    CotaSup = valor(MEJOR_SOLUCION, COSTES)

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

    iteracion = 0

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

        nodo_prometedor = min(NODOS, key=lambda x: x['ci'])['s']

        # Ramificacion
```

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

        # Cota Superior
        NODO_FINAL = [x for x in HIJOS if len(x['s']) == DIMENSION]
        if len(NODO_FINAL) > 0:
            if NODO_FINAL[0]['ci'] < CotaSup:
                CotaSup = NODO_FINAL[0]['ci']
                MEJOR_SOLUCION = NODO_FINAL[0]['s']

        # 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 solución final es:", MEJOR_SOLUCION, "en", iteracion, "iteraciones", "para dimensión:", DIMENSION)

# Ejecutamos la prueba.
if __name__ == "__main__":
    dim_ramificacion_y_poda()
```

```
Dimensión: 5x5
[33, 9, 52, 84, 70]
[91, 45, 64, 42, 22]
[88, 40, 46, 16, 46]
[51, 7, 70, 85, 31]
[15, 69, 48, 48, 79]
La solución final es: (1, 2, 4, 0, 3) en 15 iteraciones para dimensión: 5

Dimensión: 6x6
[26, 36, 42, 35, 30, 71]
[87, 78, 38, 57, 34, 7]
[15, 53, 81, 55, 31, 13]
[13, 5, 36, 48, 97, 19]
[24, 93, 13, 8, 63, 69]
[89, 96, 33, 12, 14, 34]
La solución final es: (0, 2, 5, 4, 1, 3) en 21 iteraciones para dimensión: 6

Dimensión: 7x7
[92, 70, 62, 86, 58, 1, 63]
[43, 64, 56, 87, 62, 40, 24]
[100, 65, 36, 89, 97, 34, 35]
[55, 14, 24, 62, 20, 37, 52]
[42, 86, 54, 22, 67, 62, 77]
[27, 46, 20, 38, 12, 99, 78]
[51, 87, 71, 23, 84, 37, 22]
La solución final es: (2, 1, 4, 6, 3, 5, 0) en 28 iteraciones para dimensión: 7

Dimensión: 8x8
[61, 77, 32, 70, 76, 7, 56, 6]
[38, 63, 86, 40, 22, 94, 73, 72]
[91, 56, 97, 27, 29, 57, 8, 18]
[58, 13, 63, 38, 17, 54, 16, 72]
[51, 48, 87, 3, 48, 81, 71, 47]
[78, 95, 48, 63, 16, 77, 7, 45]
[4, 86, 61, 39, 29, 17, 53, 26]
[56, 80, 80, 66, 81, 52, 39, 39]
La solución final es: (7, 1, 3, 2, 5, 0, 6, 4) en 36 iteraciones para dimensión: 8
```

De esto se puede deducir que el algoritmo por fuerza bruta deja de ser viable cuando las dimensiones aumentan en mayor cantidad, por lo que significa que cuando se tiene una matriz de $n * n$ costos, el numero de permutaciones posibles es $n!$, cuando se tiene este factorial como resultado se tendria que evaluar una permutacion de asignaciones por cada posible solucion, lo que genera que cuando $n > 10$ se produce la inviabilidad de resolver este algoritmo por fuerza bruta.

Un algoritmo por Ramificación y Poda podria volverse ineficiente cuando alcanza una dimension mayor a 20, por lo que se produce un crecimiento exponencial del espacio de busqueda y a esto se le añade las limitaciones de la memoria y el tiempo. Es importante mencionar los factores principales que pueden causar que la viabilidad de la Ramificacion y Poda no sea el camino a tomar para la optimizacion de un algoritmo cuando la dimension es muy grande.

En estos casos donde la matriz de dimension n se presenta una cantidad maxima de nodos en el arbol de busqueda correspondiente a $n!$ donde por mas que la poda sea efectiva el numero de nodos crece rapidamente con n. Asi mismo, cuando las cotas superiores e inferiores no

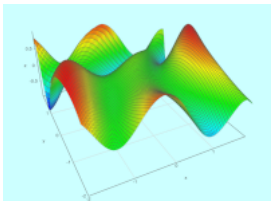
son lo suficientemente restrictivas, la mayoria de las ramas del arbol podrian permanecer abiertas, lo cual genera que se reduzca la eficiencia del algoritmo, en casos extremos se puede llegar a presentar que la cantidad de ramas presentes sea similar al de fuerza bruta con $n!$.

Como conclusion se recomienda usar la tecnica de Ramificacion y Poda cuando la dimension de n es mayor a 20 en la mayoria de casos practicos. Si dada la situacion se tiene un mayor numero de dimensiones claramente es necesario recurrir a tecnicas mas avanzadas en pro de optimizar estos casos muy grandes como heuristicas, optimizaciones avanzadas, o la descomposicion del problema.

Descenso del gradiente

Optimizar la siguiente función:

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



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

# Definir la función objetivo
f = lambda X: math.sin(0.5 * X[0]**2 - 0.25 * X[1]**2 + 3) * math.cos(2 * X[0] + 1 - math.exp(X[1]))

# Gradiente de la función (derivado manualmente)
def grad_f(X):

    x, y = X[0], X[1]
    # Derivada parcial con respecto a x
    df_dx = (
        math.cos(0.5 * x**2 - 0.25 * y**2 + 3) * x * math.cos(2 * x + 1 - math.exp(y)) -
        math.sin(0.5 * x**2 - 0.25 * y**2 + 3) * 2 * math.sin(2 * x + 1 - math.exp(y))
    )
    # Derivada parcial con respecto a y
    df_dy = (
        -math.cos(0.5 * x**2 - 0.25 * y**2 + 3) * 0.25 * y * math.cos(2 * x + 1 - math.exp(y)) +
        math.sin(0.5 * x**2 - 0.25 * y**2 + 3) * math.exp(y) * math.sin(2 * x + 1 - math.exp(y))
    )
    return np.array([df_dx, df_dy])

# Descenso del gradiente
def gradient_descent(f, grad_f, learning_rate=0.01, max_iter=1000, tolerance=1e-6):

    # Inicialización de X = [x, y] aleatorios
    X = np.random.uniform(-1, 1, size=2)
    trajectory = [X]  # Para guardar el recorrido del algoritmo
    for i in range(max_iter):
        grad = grad_f(X)  # Calcula el gradiente en X
        X_new = X - learning_rate * grad  # Actualiza X

        # Guardar la trayectoria
        trajectory.append(X_new)

        # Verifica convergencia (si el cambio en X es pequeño)
        if np.linalg.norm(X_new - X) < tolerance:
            print(f"Convergencia alcanzada en {i} iteraciones")
            break

        X = X_new  # Actualiza el punto actual

    return X, f(X), np.array(trajectory)

# Visualización del paisaje de la función y el recorrido
def plot_descent(f, trajectory, xlim=(-2, 2), ylim=(-2, 2), resolution=100):

    x = np.linspace(xlim[0], xlim[1], resolution)
    y = np.linspace(ylim[0], ylim[1], resolution)
    X, Y = np.meshgrid(x, y)

    # Evaluar la función en la rejilla
```

```
    Z = np.array([[f([x1, y1]) for x1, y1 in zip(row_x, row_y)] for row_x, row_y in zip(X, Y)])

    # Gráfico de la superficie 3D
    fig, ax = plt.subplots(figsize=(10, 6), subplot_kw={"projection": "3d"})
    ax.plot_surface(X, Y, Z, cmap="viridis", alpha=0.8)
    ax.set_title("Función con descenso de gradiente")
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("f(x, y)")

    # Agregar la trayectoria en 3D
    trajectory = np.array(trajectory)
    ax.plot(trajectory[:, 0], trajectory[:, 1], [f(xy) for xy in trajectory], color="r", marker="o", label="Recorrido")
    ax.legend()
    plt.show()

# Ejecutar y graficar
if __name__ == "__main__":
    optimal_X, optimal_value, trajectory = gradient_descent(f, grad_f)
    print(f"Óptimo en X: {optimal_X}, Valor óptimo: {optimal_value}")
    plot_descent(f, trajectory)
```

```
Óptimo en X: [-0.91888812 -0.78075718], Valor óptimo: -0.03470944080454091
```