Actividad Guiada 2 del curso: Algoritmos de optimización

Nombre: Carlos Esteban Posada

URL: https://colab.research.google.com/drive/1hESA5X-TeFVtTEO9NyGfugsqYg_Ujs-I?usp=sharing

URL GIT: https://github.com/cposada8/03MAIR-Algoritmos-de-Optimizacion-CEPM/blob/main/carlos_esteban_posada_AG2.ipynb

```
import math
import numpy as np
import itertools
from time import time
import matplotlib.pyplot as plt
```

Viaje por el río

```
tarifas = [
[0,5,4,3,999,999,999],   #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]
]
```

```
hijos = [
[1, 2, 3], # hijos del nodo 0
[3, 4, 6], # hijos del nodo 1
[3, 5, 6], # hijos del nodo 2
[4, 5, 6],
[6],
[6],
[]
]
```

```
# función para generar el grafo inverso.
# se recorre el grafo original y se buscan los padres
def padres_de(grafo):
  # se inicializa el grafo de padres (el inverso de hijos) todo con listas vacías
  # es decir por ahora nadie tiene padre
  padres = [[] for i in range(len(hijos))]
```

```
  for padre in range(len(grafo)): # cada línea es un nodo
    for hijo in grafo[padre]:
      padres[hijo].append(padre)
  return padres


# generación del grafo de padres (grafo inverso)
padres = padres_de(hijos)
padres
```

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

```
origen = 0
objetivo = 6


# se inicializan los vectores que se usarán durantes las iteraciones del algoritmo
distancias = [np.inf for x in range(len(hijos))] # vector de distancias desde cada nodo hasta
visitados = set() # todos nodos que están perfectamente visitados
rutas = [[] for x in range(len(hijos))] # representa el nodo óptimo al que se puede viajar de
print(distancias)
print(visitados)
print(rutas)
```

```
[inf, inf, inf, inf, inf, inf, inf]
set()
[[], [], [], [], [], [], []]
```

```
# inicio visitando al nodo objetivo
distancias[objetivo] = 0
visitados.add(objetivo)


# algoritmo de atrás hacia adelante
queue = [] # cola para ir recorriendo los nodos desde el final

# agregar los padres del objetivo
for padre in padres[objetivo]:
  queue.append(padre)

# inicio del llenado de las distancias:
while (queue) and (origen not in visitados):
  # print("cola actual:",queue)
  elemento = queue.pop(0)
  # print("elemento", elemento)

  # primero se verifica que el elemento no haya sido visitado ya:
  if elemento in visitados:
    continue

  # verificar que el elemento no tenga hijos sin visitar
```

```
      # si los tiene, se manda el elemento para el final de la lista
      hijos_visitados = set(hijos[elemento]).intersection(visitados)
      # print("hijos visitados", hijos_visitados)
      if len(hijos[elemento]) != len(hijos_visitados):
        # print(elemento, hijos[elemento], "tiene sin visitar")
        queue.append(elemento)
        pass
      else: # todos sus hijos están visitados

        # se encuentra la distancia del elemento al objetivo pasando por cada hijo
        distancias_nodo = {}
        for hijo in hijos[elemento]:
          distancias_nodo[hijo] = distancias[hijo] + tarifas[elemento][hijo]
        min_distancia = min(distancias_nodo.values())
        # print("mindistancia", min_distancia)

        # se agregan la distancia del elemento al nodo objetivo, los visitados a la
        # lista de visitados y se agregan las rutas óptimas desde el elemento
        # hasta el objetivo
        distancias[elemento] = min_distancia
        visitados.add(elemento)
        # enlistar todos los elementos de distancia minima
        posibles_rutas = [key for key, val in distancias_nodo.items() if val == min_distancia]
        # print("posibles_rutas", posibles_rutas)
        rutas[elemento] = posibles_rutas

        # finalmente se agregan los padres del elemento a la cola
        for padre in padres[elemento]:
          queue.append(padre)

      # print()


    print("posibles rutas óptimas:", posibles_rutas)
```

```
posibles rutas óptimas: [2]
```

```
    # imprimir una ruta óptima
    ruta_optima = [origen]
    visitado = origen
    while (visitado != objetivo):
      siguiente = rutas[visitado][0]
      ruta_optima.append(siguiente)
      visitado = siguiente
    print("Una ruta óptima es:", ruta_optima)
    print("con un costo total de:", distancias[origen])
```

```
Una ruta óptima es: [0, 2, 5, 6]
con un costo total de: 11
```

Problema de Asignacion de tarea

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

```
def valor(S,COSTES):
  VALOR = 0
  for i in range(len(S)):
    VALOR += COSTES[S[i]][i]
  return VALOR
valor((1, 2, 0, 3),COSTES)
```

```
77
```

```
#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
```

```
#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


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, D

    #Revisamos la cota superior y nos quedamos con la mejor solucion si llegamos a una soluci
    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" , " par
```

```
ramificacion_y_poda(COSTES)
```

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

Análisis adicional (Mejorar nota)

- ¿Qué complejidad tiene el algoritmo por fuerza bruta?
- Generar matrices con valores aleatorios de mayores dimensiones (5, 6, 7, ...) y ejecutar ambos algoritmos.
  - ¿A partir de qué 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 deja de ser una opción válida?

El algoritmo por fuerza bruta tendrá un orden de complejidad n! pues deberá comprobar todas las posibles permutaciones de los índices de un array de tamaño n.

Implementación del método por fuerza bruta

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

def asignacion_tareas_fuerza_bruta(costes):
  n = len(costes)
  min_cost = np.inf
  sol_opt = None
  # obtengo todas las permutaciones posibles
  permutations = itertools.permutations(range(n))
  for option in permutations:
    costo_actual = costo_solucion(option, costes)
    if costo_actual < min_cost:
      min_cost = costo_actual
      sol_opt = option
  return sol_opt


print(asignacion_tareas_fuerza_bruta(COSTES))
```

```
costo_solucion(asignacion_tareas_fuerza_bruta(COSTES), COSTES)
```

```
(0, 3, 1, 2)
61
```

Comparación ramificación y poda vs fuerza bruta

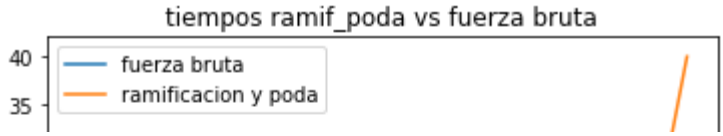
```
max_coste =50
dimensiones = [2, 3, 4, 5, 6, 7, 8, 9, 10]
matrices_aleatorias = {dimension:np.random.randint(max_coste, size=(dimension, dimension)) fo


# ejecutando método de ramificación y poda
tiempos_ramif_poda = []
for dimension in dimensiones:
  print("dimension:", dimension)
  # ejecutar el algoritmo de ramificación y poda
  t_start = time()
  ramificacion_y_poda(matrices_aleatorias[dimension])
  t_end = time()-t_start
  print(t_end)
  tiempos_ramif_poda.append(t_end)
```

```
dimension: 2
La solucion final es: (0, 1)  en  1  iteraciones  para dimension:  2
0.001697540283203125
dimension: 3
La solucion final es: [{'s': (0, 2, 1), 'ci': 15}]  en  4  iteraciones  para dimension:
0.0014483928680419922
dimension: 4
La solucion final es: (0, 1, 2, 3)  en  5  iteraciones  para dimension:  4
0.00037169456481933594
dimension: 5
La solucion final es: [{'s': (2, 4, 3, 0, 1), 'ci': 84}]  en  37  iteraciones  para dim
0.0016393661499023438
dimension: 6
La solucion final es: [{'s': (3, 0, 2, 5, 4, 1), 'ci': 31}]  en  114  iteraciones  para
0.0046536922454833984
dimension: 7
La solucion final es: [{'s': (2, 3, 1, 5, 0, 4, 6), 'ci': 74}]  en  145  iteraciones  p
0.009642601013183594
dimension: 8
La solucion final es: [{'s': (7, 3, 1, 0, 2, 6, 4, 5), 'ci': 42}]  en  5404  iteracione
2.681424140930176
dimension: 9
La solucion final es: [{'s': (4, 5, 3, 8, 6, 0, 7, 1, 2), 'ci': 64}]  en  1836  iteraci
0.3778538703918457
dimension: 10
La solucion final es: [{'s': (3, 4, 5, 9, 6, 8, 2, 7, 1, 0), 'ci': 77}]  en  18693  ite
39.892725706100464
```

```
# ejecutando método de fuerza bruta
tiempos_fuerza_bruta = []
for dimension in dimensiones:
  print("dimension:", dimension)
  # ejecutar el algoritmo de ramificación y poda
  t_start = time()
  asignacion_tareas_fuerza_bruta(matrices_aleatorias[dimension])
  t_end = time()-t_start
  print(t_end)
  tiempos_fuerza_bruta.append(t_end)
```

```
dimension: 2
8.749961853027344e-05
dimension: 3
6.747245788574219e-05
dimension: 4
0.00013399124145507812
dimension: 5
0.0006535053253173828
dimension: 6
0.002458333969116211
dimension: 7
0.02022695541381836
dimension: 8
0.18480420112609863
dimension: 9
1.7972385883331299
dimension: 10
19.158466815948486
```

```
# comparación tiempos
plt.plot(dimensiones, tiempos_fuerza_bruta, label="fuerza bruta")
plt.plot(dimensiones, tiempos_ramif_poda, label="ramificacion y poda")
plt.legend()
plt.title("tiempos ramif_poda vs fuerza bruta")
plt.xlabel("dimensiones (entero)")
plt.ylabel("tiempo (segundos)")
plt.show()
```



Como se puede observar, el método de ramificación y poda tiene una eficiencia muy superior al de fuerza bruta. Esto debido a los órdenes de complejidadad de ambos algoritmos pues el de fuerza bruta es **O(n^2)** mientras que el de ramificación y poda es exponencial. Por este motivo, tampoco el algoritmo de ramificación y poda será útil en casos de n grande como comprobaremos en la siguiente sección

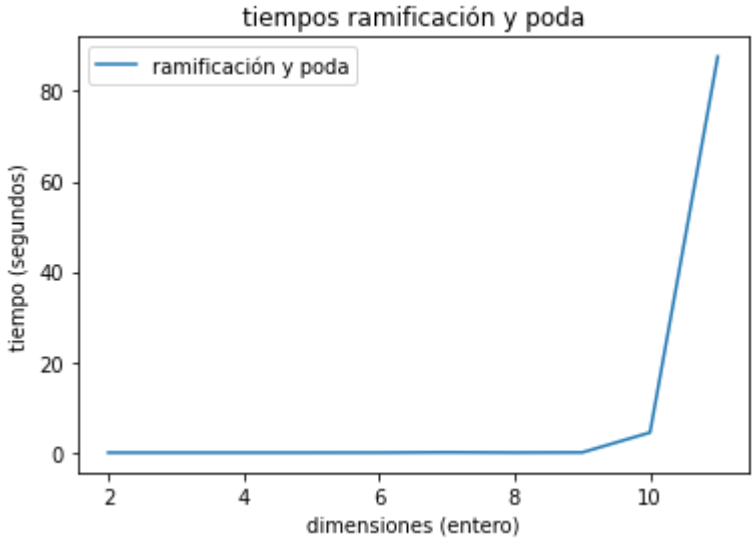
¿Hay alguna dimensión para la cual el algoritmo de ramificación y poda deja de ser opción?

```
max_coste =50
dimensiones = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
matrices_aleatorias = {dimension:np.random.randint(max_coste, size=(dimension, dimension)) fo
# ejecutando método de ramificación y poda
tiempos_ramif_poda = []
for dimension in dimensiones:
  print("dimension:", dimension)
  # ejecutar el algoritmo de ramificación y poda
  t_start = time()
  ramificacion_y_poda(matrices_aleatorias[dimension])
  t_end = time()-t_start
  print(t_end)
  tiempos_ramif_poda.append(t_end)
```

```
dimension: 2
La solucion final es: (0, 1)  en  1  iteraciones  para dimension:  2
0.00028133392333984375
dimension: 3
La solucion final es: (0, 1, 2)  en  1  iteraciones  para dimension:  3
0.0002751350402832031
dimension: 4
La solucion final es: [{'s': (0, 1, 3, 2), 'ci': 87}]  en  17  iteraciones  para dimens
0.0009584426879882812
dimension: 5
La solucion final es: [{'s': (0, 2, 3, 1, 4), 'ci': 68}]  en  43  iteraciones  para dim
0.0013470649719238281
dimension: 6
La solucion final es: [{'s': (3, 2, 4, 5, 0, 1), 'ci': 29}]  en  151  iteraciones  para
0.01044464111328125
dimension: 7
La solucion final es: [{'s': (3, 4, 6, 0, 5, 2, 1), 'ci': 78}]  en  720  iteraciones  p
0.07729005813598633
```

```
dimension: 8
La solucion final es: [{'s': (3, 5, 4, 7, 0, 1, 2, 6), 'ci': 69}]  en  174  iteraciones
0.0184938907623291
dimension: 9
La solucion final es: [{'s': (1, 5, 6, 2, 4, 0, 7, 3, 8), 'ci': 57}]  en  367  iteracio
0.050852060317993164
dimension: 10
La solucion final es: [{'s': (6, 2, 4, 9, 0, 5, 8, 3, 1, 7), 'ci': 62}]  en  6390  iter
4.448698282241821
dimension: 11
La solucion final es: [{'s': (6, 5, 2, 7, 3, 4, 0, 10, 1, 9, 8), 'ci': 51}]  en  24863
87.59654641151428
```

```
# gráfica tiempos
plt.plot(dimensiones, tiempos_ramif_poda, label="ramificación y poda")
plt.legend()
plt.title("tiempos ramificación y poda")
plt.xlabel("dimensiones (entero)")
plt.ylabel("tiempo (segundos)")
plt.show()
```



Como se puede observar, desde la dimensión 11 en adelante el algoritmo empezará a ser inviable. Su naturaleza al ser exponencial dicta que habrá puntos donde ya no será una opción revisar este tipo de algoritmos.

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


def funobj_1(X):
  # x^2 + y^2
  return X[0]**2 + X[1]**2

def d_funobj_1(X):
  return np.array([
                   2*X[0],
                   2*X[1]
  ])



w = np.array([random.uniform(-2,2), random.uniform(-2,2)]) # vector solución inicial


def step_gradiente_descendiente(df, w, alpha = 0.05):
    w = w - alpha*df(w)
    return w

def gradiente_descendiente(f, df, w, max_iter = 100, alpha = 0.05):
  # esta función ejecuta varios pasos del algoritmo de gradiente descendiente
  # y retornará:
  # w: el vector de solución luego de ejecutar los steps de gradiente descendiente
  # ws: un vector con todas las w que encontró en cada step (incluida la solución original)
  ws = [w]
  for i in range(max_iter):
    w = step_gradiente_descendiente(df, w, alpha)
    ws.append(w)
  return w, ws


w_opt, ws = gradiente_descendiente(d_funobj_1, d_funobj_1, w ,500, 0.08)
w_opt
```

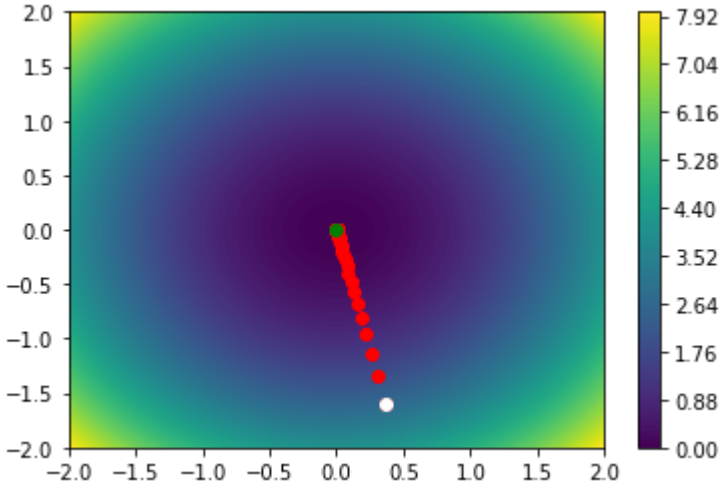
```
array([ 4.96715392e-39, -2.21531246e-38])
```

```
#Prepara los datos para dibujar mapa de niveles de Z
resolucion = 100
rango=2
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[ix,iy] = f([x,y])
```

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

```
for sol in ws:
  grad = df(P)
  #print(P,grad)
  # P[0],P[1] = P[0] - TA*grad[0] , P[1] - TA*grad[1]
  plt.plot(sol[0],sol[1],"o",c="red")
# pintar el punto de partida y el punto de llegada
plt.plot(w[0],w[1],"o",c="white")
plt.plot(w_opt[0],w_opt[1],"o",c="green")
```

```
[<matplotlib.lines.Line2D at 0x7f8bb0e6a208>]
```



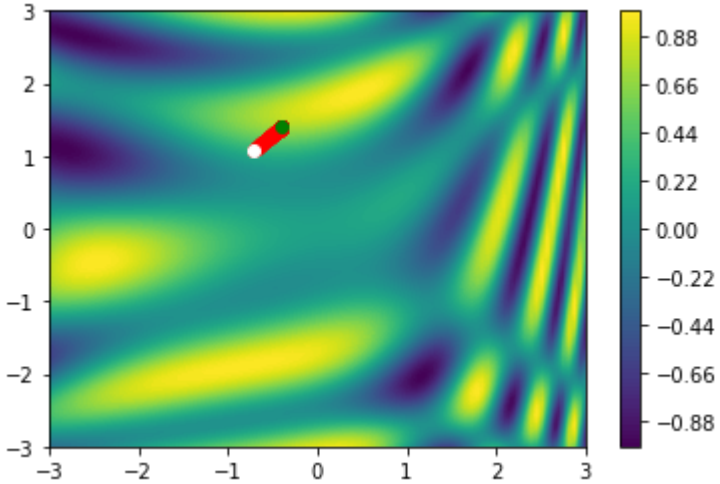
Funcón con muchos mínimos y máximos locales

```
def f2(X):
  # función con muchos mínimos
  return np.sin(1/2 * X[0]**2 - 1/4 * X[1]**2 + 3) * np.cos(2 * X[0] + 1 - np.exp(X[1]))

def numeric_derivative(f, X, h=0.001):
  # función para aproximar la derivada de una función dada evaluada en un punto
  X = np.array(X)
  grad = np.zeros(len(X))
  for i in range(len(X)):
    grad[i] = (f(X+h) - f(X)) / h
  return grad

def d_f2(X):
  return numeric_derivative(f2, X)
```

```
w = np.array([random.uniform(-2,2),random.uniform(-2,2)])
```

```
w_opt, ws = gradiente_descendiente(f2, d_f2, w ,100, 0.01)
# w_opt

 #Prepara los datos para dibujar mapa de niveles de Z
rango=3
resolucion = 100
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[ix,iy] = f2([x,y])

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


# pintar todos los puntos por los que se pasó
for sol in ws:
  grad = df(P)
  plt.plot(sol[0],sol[1],"o",c="red")
# pintar el punto de partida y el punto de llegada
plt.plot(w[0],w[1],"o",c="white")
plt.plot(w_opt[0],w_opt[1],"o",c="green")
```

```
[<matplotlib.lines.Line2D at 0x7f8bb06e6a20>]
```



Se puede observar que el contorno de la gráfica anterior es demasiado complejo, presentando muchos máximos y mínimos locales. Esto implica que el algoritmo del descenso del gradiente no garantizará encontrar el mínimo global, pues lo único que hará será encontrar el máximo más cercano al punto de partida (w0 en este caso)