Algoritmos - Actividad Guiada 2

Nombre : Juan Carlos Marin Mejia

URL: https://colab.research.google.com/drive/13n86aVcqwJ5gMTvFYXxqcg2DNZSWYr1X?usp=sharing

GitHub: https://github.com/jcmm518/03MAIR-Algoritmos-de-optimizacion.git

Problema: Viaje por el río - Técnica: Programacion dinamica

```
# Funcion para calcular el tiempo de ejecucuion
from time import time

def calcular_tiempo(f):

    def wrapper(*args, **kwargs):
        inicio = time()
        resultado = f(*args, **kwargs)
        tiempo = time() - inicio
        print("Tiempo de ejecución para algoritmo: "+str(tiempo))
        return resultado

    return wrapper
from time import time


import math



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]
]


def Precios(TARIFAS):

  N = len(TARIFAS[0])  # N = la longitud de la matriz de tarifas
  PRECIOS = [ [9999]*N for i in [9999]*N]  # inicializamos la matriz de precios con el valor
  RUTA = [ [""]*N for i in [""]*N]  # inicializamos la matriz de rutas con valores en blanco
```

```
    RUTA = [ [  ]*N for i in [  ]*N]  # Inicializamos la matriz de rutas con valores en blanco

    for i in range(N-1):
      for j in range(i+1, N):
        MIN = TARIFAS[i][j]
        RUTA[i][j] = i
        # Recorremos los nodos intermedios para encontrar el minimo
        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]
['', '', '', '', '', '', '']
```

```
def calcular_ruta(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)
```

```
calcular_ruta(RUTA, 0,0)
```

```
La ruta es:
'0,0,2,5'
```

Problema: Asignación de tareas - Técnica: Ramificación y poda

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

```
73
```

Ramificacion y poda por Fuerza Bruta

```
import itertools
def fuerza_bruta (COSTES) :
  mejor_valor = 10e10
  mejor_solucion = ()

  for s in list(itertools.permutations(range(len(COSTES)))):
    valor_tmp = valor(s,COSTES)
    if valor_tmp < mejor_valor :
      mejor_valor = valor_tmp
      mejor_solucion = s

  print('La mejor solucion es : ', mejor_solucion, " con valor: ", mejor_valor)

start_time = time() # inicio el cronometro
fuerza_bruta(COSTES)
run_time = time() - start_time # calculo el tiempo despues de terminado el proceso
print("Calculo valor: %.10f Segundos." % run_time)
```

```
La mejor solucion es :  (0, 3, 1, 2)  con valor:  61
Calculo valor: 0.0015864372 Segundos.
```

Función para estimar una cota inferior para una solución parcial:

```
#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

CS((0,1),COSTES)
```

```
74
```

```
#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)
```

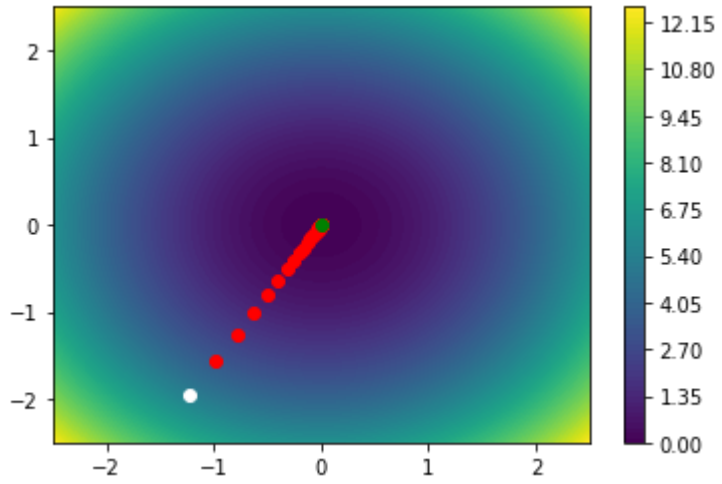
```
  NODOS=[]
  NODOS.append({'s':(), 'ci':CI((),COSTES)    } )

  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 dimensi
```

Algoritmos para 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


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


#Prepara los datos para dibujar mapa de niveles de Z
resolucion = 100
rango=2.5
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
P=[random.uniform(-2,2  ),random.uniform(-2,2 ) ]
plt.plot(P[0],P[1],"o",c="white")

#Tasa de aprendizaje
TA=.1

#Iteraciones
for _ in range(500):
  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")

plt.plot(P[0],P[1],"o",c="green")
plt.show()
print("Solucion:" , P , f(P))
```




```
Solucion: [-4.304414775100683e-49, -6.885835324891302e-49] 6.594271467762597e-97
```