

AG2 – Actividad Guiada 2

03MAIR – Algoritmos de Optimización


Agenda

- Nuevo tema en el Foro
- Teoría: Ramificación y Poda
- Práctica: Programación dinámica(Viaje por el rio).
- Práctica: Búsqueda en grafos, ramificación y poda(asignación de tareas).
- Practica: Descenso del gradiente.

Foro. Nuevo tema

Problemas Tipo

Ahora mismo

Raul Reyero Diez 

Problemas Tipo

Quisiera proponer para esta semana la siguiente contribución al foro.

Se trata de recopilar problemas tipo de optimización matemática.

Hay muchos pero quisiera que investigaras un poco sobre uno de ellos. El que tu elijas. No se trata de resolver ningún caso sino de conocer algo de información.

Inicio yo a modo de ejemplo con el problema del agente viajero:

Nombre: Problema del agente viajero (travelling salesman problem - TSP)

Descripción: Búsqueda óptima del camino más corto que debe recorrer un agente por una serie de ciudades pasando por todas, una y sólo una vez y volviendo a la ciudad de partida.

Modelización: A través de grafos, con programación lineal entera.

Resolución: Exactos (Ramificación y poda, Programación dinámica) y Heurísticos (colonia de hormigas, genéticos)

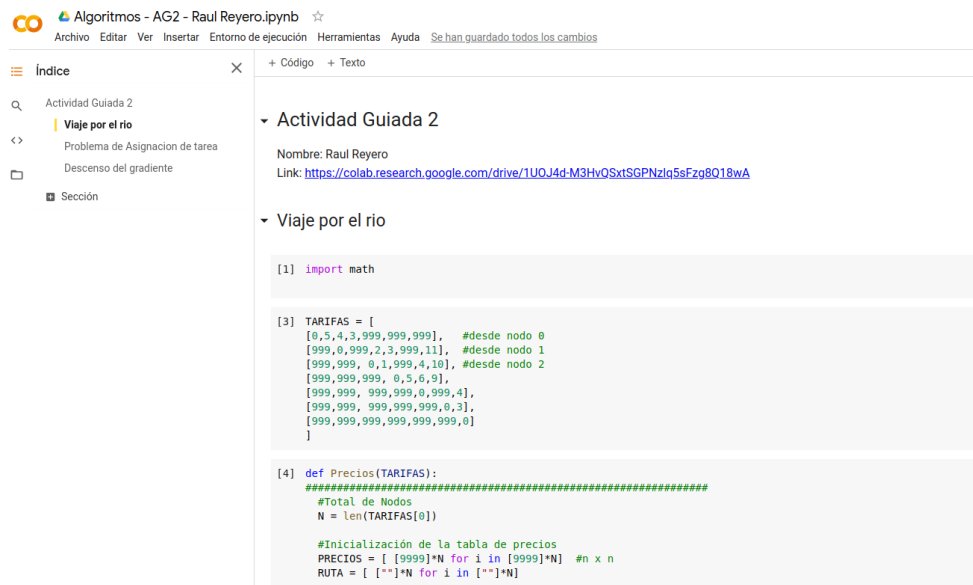
Recursos:

- [Wikipedia \(para empezar\)](#)
- [Técnicas heurísticas aplicadas al problema del cartero viajante \(tsp\)](#) - Ricardo Alberto Hincapié
- [Lin-Kernighan heuristic](#)
- [TSPLIB](#) (recurso con diferentes datos de prueba para probar nuestros diseños)

Responder

Google Colab

<https://colab.research.google.com/drive/1UOJ4d-M3HvQSxtSGPNzIq5sFzg8Q18wA?usp=sharing>



The screenshot shows a Google Colab notebook interface. The title bar reads "Algoritmos - AG2 - Raul Reyero.ipynb" with a star icon. Below the title bar are tabs for "Archivo", "Editar", "Ver", "Insertar", "Entorno de ejecución", "Herramientas", "Ayuda", and a link "Se han guardado todos los cambios". On the left, a sidebar shows a file explorer with "Actividad Guiada 2" containing "Viaje por el rio", "Problema de Asignación de tarea", and "Descenso del gradiente". Below this is a "Sección" icon. The main area has a tab for "Actividad Guiada 2" with a sub-tab for "Viaje por el rio". The code cell contains the following Python code:

```
[1] import math

[3] 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]
]

[4] 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]
```

Programación dinámica (I)

- **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 reutilizadas.
 - ✓ 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)



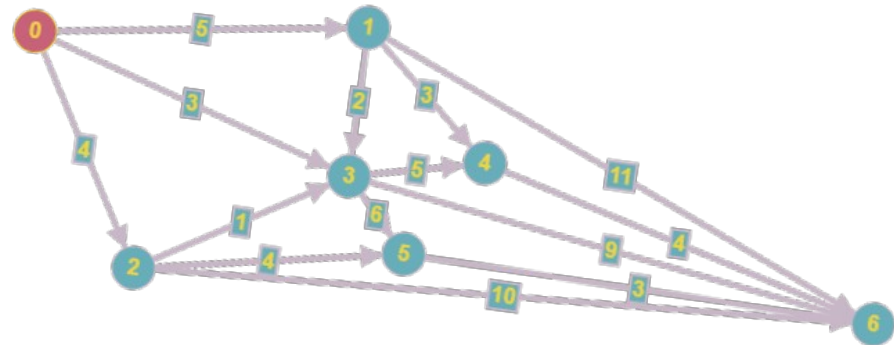
importante

Programación dinámica (I)

Problema: Viaje por el río

- Consideramos una tabla $T(i,j)$ 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)
- Establecer una tabla intermedia($P(i,j)$) para guardar soluciones óptimas parciales para ir desde i a j .

$$P(i,j) = \min \{ T(i,j) , P(i,k) + T(k,j) \text{ para todo } i < k \leq j \}$$



Programación dinámica (II)

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

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

    for i in range(N-1):
        for j in range(i+1, N):
            MIN = TARIFAS[i][j]
            RUTA[i][j] = i

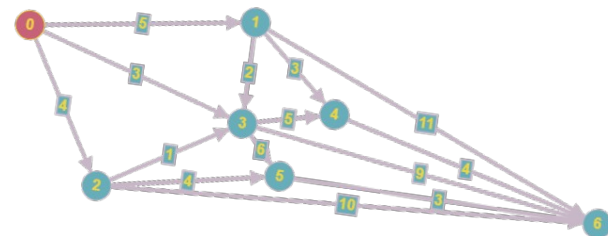
            for k in range(i, j):
                if PRECIOS[i][k] + TARIFAS[k][j] < MIN:
                    MIN = min(MIN, PRECIOS[i][k] + TARIFAS[k][j])
                    RUTA[i][j] = k
                    PRECIOS[i][j] = MIN

    return PRECIOS, RUTA
#####
```

Operaciones

n

n

 $3 \cdot n^3$ 

```
[ ] TARIFAS = [
[0,5,4,3,999,999,999],
[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]
]
```

(*) En lugar de 999

```
import math
math.inf
```

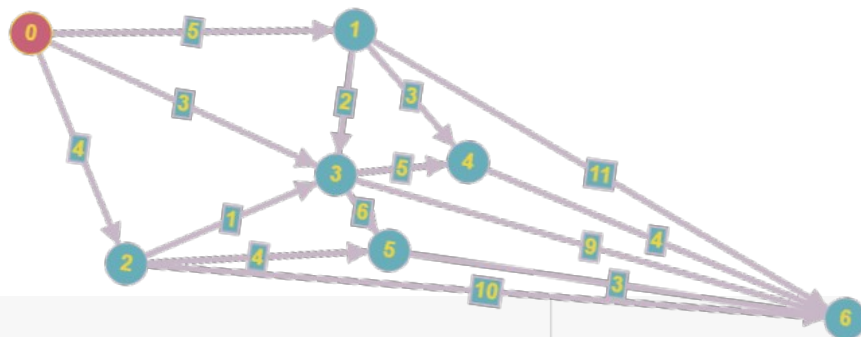
$$P(i,j) = \min \{ T(i,j) , P(i,k)+T(k,j) \text{ para todo } i < k \leq j \}$$

Programación dinámica (III)

- RUTA contiene la mejor opción intermedia para ir de un nodo a otro

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)
```

Recursividad

Ramificación y Poda.

Problema: Asignación de tareas

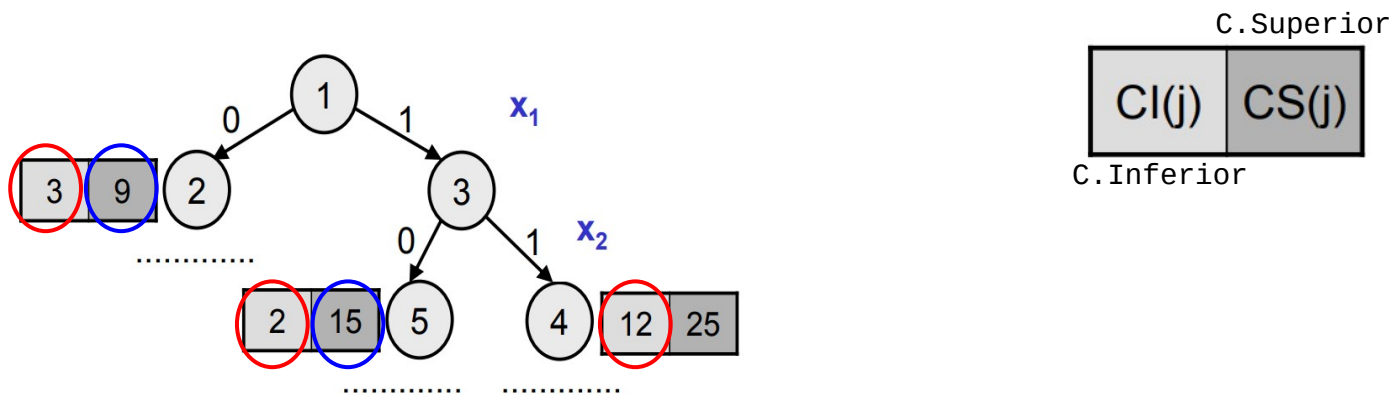
- El problema consiste en **maximizar** el rendimiento (o **minimizar** los costes) en cuanto a la asignación de N tareas a N agentes. Cada tarea solo puede ser asignado a un agente.
- Los beneficios que se obtienen al realizar la **tarea 1 por el agente A es 9**
- La matriz de beneficios es la que se muestra en la figura
- Aplicando Ramificación y Poda, obtener la asignación que maximice los beneficios.



	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Algoritmos de Búsqueda. Técnicas de Ramificación y Poda(V)

- Estrategia de poda: Poda i si $CS(i) \leq CI(j)$ Para algún j ya generado



$C=12$ (max. de las cotas inferiores)

El nodo 2 no puede mejorar(valor 9) las peores predicciones de 4(valor 12 = max{todo CI() })

El nodo 5 quizá pueda mejorar al nodo 4

Ramificación y Poda.

Problema: Asignación de tareas

- P.ej. Si asignamos el agente A a la tarea1
 - Cota Inferior para la tarea2 es 4
 - Cota Superior para la tarea2 es 8

	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Ramificación y Poda.

Problema: Asignación de tareas

- ¿Cómo diseñamos un estado?
- ¿Como expandimos nodos (ramificación)?
- ¿Como podemos?
- ¿Como recorreremos el árbol completo de estados?
- ¿Qué complejidad tiene? (*)

(*) es difícil calcular el número de operaciones exacto. Calcular el **mejor** caso y el **peor** caso.

Peor caso = tengo que expandir todos los niveles.

Mejor caso = puedo podar todos los nodos de un nivel menos uno.

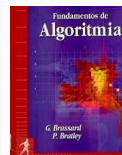
En general podemos asumir una complejidad exponencial

Ramificación y Poda. Practica

#Del libro Fundamentos de algoritmia - G. Brassard & P. Bratley pg 349, sec. 9.7.1

'''Hay que asignar n tareas a n agentes, de forma que a cada agente le corresponda una tarea.
Se dispone de una tabla de costes: el coste para el agente i al realizar la tarea j es $c_{ij} \geq 0$.
Se busca la asignación que reduzca el coste total.
Diferentes aplicaciones,

Ejemplo de tabla de costes para $n=4$. Agentes: a, b, c y d . Tareas: 1,2,3 y 4
'''



TAREAS

A

G

E

N

T

E

S

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

La tupla: (1, 0, 3, 2) representa la solución :

Agente 0 a la tarea 1

"	1	"	0
"	2	"	3
"	3	"	2

Ramificación y Poda. Practica

Función objetivo

- S es una solución parcial P.ej (1,0,)
- C es la matriz de costes(o beneficios)

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

```
# TAREAS  
# A  
# G  
# E  
# N  
# T  
# E  
# S  
  
COSTES=[[11,12,18,40],  
        [14,15,13,22],  
        [11,17,19,23],  
        [17,14,20,28]]
```

Ramificación y Poda. Practica

Fuerza Bruta:

```
@calcular_tiempo
def fuerza_bruta(COSTES):
    #Representacion de la solucion será una tupla donde cada valor en la cordenada i-sima es la tarea asignado al agente i
    # Ejemplo (1,2,3,4) Tiene valor 11+15+19+28=73
    #
    #¿Cuantas posibilidades hay? n! -> Complejidad factorial(exponencial)
    #Con dimension 11 se va a 1 minuto de ejecucion

    mejor_valor = 10e10
    mejor_solucion = ()

    for s in list(itertools.permutations(range(len(COSTES))));
        #print(s,valor(s,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 )
```

```
# TAREAS
# A
# G
# E
# N
# T
# E
# S
```

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

Ramificación y Poda. Practica

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
```

Ej. La tupla (2,0,)

Representa un nodo
intermedio con :
Agente 0 asignado a 2
Agente 1 asignado a 0

Ramificación y Poda. Practica

Función para ramificar:

```
#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
```

Ramificación y Poda. Practica

Proceso principal (I):

@calcular_tiempo

```
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
```

Inicializamos con una
solución
cualquiera(opcional)

NODOS guarda los nodos
que debemos
explorar(ramificación)

Ramificación y Poda. Practica

Proceso principal (II):

```
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 )
```

1

De entre los nodos pendientes a explorar, elegimos el mas prometedor

2

3

4

5

6

Ramificación y Poda. Practica

Análisis para **mejorar nota**:

- ¿Que 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 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 deja de ser una opción válida?



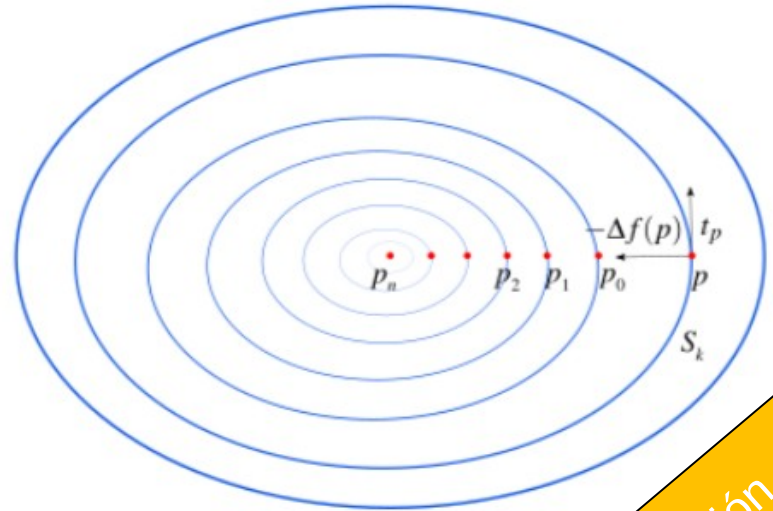
10/10

Descenso del Gradiente – AGD

- El procedimiento parte de un punto **p** como **solución aproximada** y da pasos en el sentido opuesto al gradiente (si minimizamos) de la función en dicho punto.

$$p_{t+1} = p_t - \alpha_t \nabla f(p_t)$$

Donde el parámetro α_t se selecciona para que p_{t+1} sea solución
 $\gamma \quad f(p_t) \geq f(p_{t+1})$

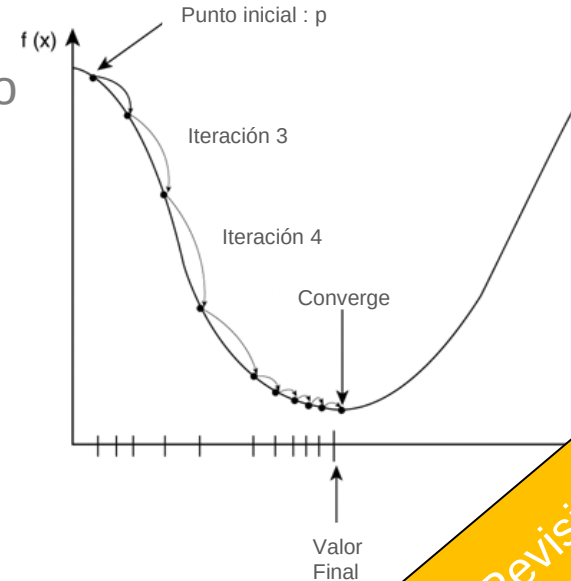


Revisión

Descenso del Gradiente – AGD

- En general se va reduciendo el valor de α_t dinámicamente a medida que nos aproximamos a la solución que podemos deducir por:
 - la magnitud del gradiente
 - cantidad de iteraciones que hemos realizado

$$p_{t+1} = p_t - \alpha_t \nabla f(p_t)$$



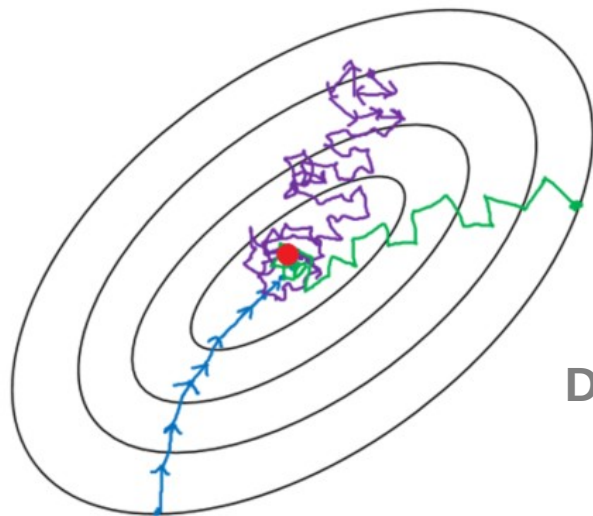
Revisión

Descenso del Gradiente – AGD

- **Dependiendo del Volumen de Datos**
 - ✓ Descenso del gradiente por lotes (**batch gradient descent**)
Calcula la desviación para todos los puntos en cada iteración!!!
 - ✓ Descenso del gradiente estocástico(**stochastic gradient descent**)
Calcula la desviación para un punto en cada iteración!!!
 - ✓ Descenso del gradiente por lotes reducido(**mini-batch gradient descent**)
Mezcla de ambos conceptos

Descenso del Gradiente – AGD

- Dependiendo del Volumen de Datos



- Batch gradient descent (directo pero muy lento)
- Stochastic gradient descent(rápido pero muy disperso)
- Mini-batch gradient descent(equilibrio)

Decisión: Elección de mini-batch size

Descenso del gradiente. Práctica

Preparar entorno

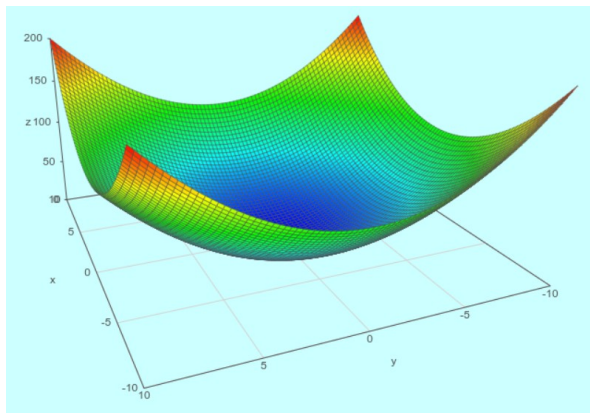
```
import math                                #Funciones matematicas
import matplotlib.pyplot as plt           #Generacion de gráficos (otra opcion seaborn)
import numpy as np                         #Tratamiento matriz N-dimensionales y otras (fundamental!)
#import scipy as sc

import random
```

Descenso del gradiente. Práctica

La función a minimizar. Paraboloide

```
f = lambda X: X[0]**2+X[1]**2      #Funcion  
df = lambda X: [2*X[0] , 2*X[1]]  #Gradiente
```



<http://al-roomi.org/3DPlot/index.html>

Descenso del gradiente. Práctica

Prepara los datos para el gráfico

```
#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()
```

Descenso del gradiente. Práctica

Generamos un Punto aleatorio

```
#Generamos un punto aleatorio  
P=[random.uniform(-2,2 ),random.uniform(-2,2 ) ]  
plt.plot(P[0],P[1],"o",c="white")
```

Descenso del gradiente. Práctica

Iteramos el algoritmo

```
#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")
```

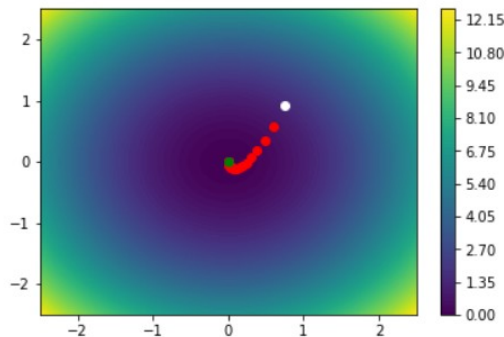
$$p_{t+1} = p_t - \alpha_t \nabla f(p_t)$$

Descenso del gradiente. Práctica

Pintamos el gráfico con las iteraciones

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

☞ Punto Inicial: [0.7569403161678125, 0.9223403659681089]

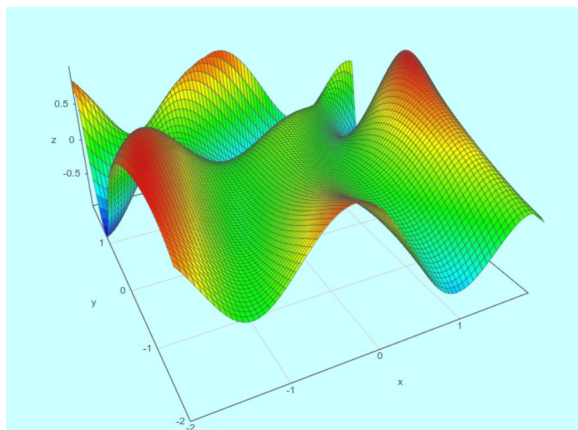


Solucion: [-0.004999999999999998, -0.005000000000000004] 5.000000000000002e-05

Descenso del gradiente. Práctica

Otra función a minimizar.

```
#Definimos la funcion  
#sin(1/2 * x^2 - 1/4 * y^2 + 3) * cos(2*x + 1 - E^y)  
f = lambda X: np.sin(1/2 * X[0]**2 - 1/4 * X[1]**2 + 3) * np.cos(2 * X[0] + 1 - np.e**X[1])
```



$$\sin\left(\frac{1}{2} * x^2 - \frac{1}{4} * y^2 + 3\right) * \cos(2*x + 1 - E^y)$$

Descenso del gradiente. Práctica

¿Y el gradiente? !!



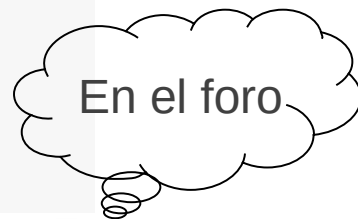
```
#Definimos la funcion
#sin(1/2 * x^2 - 1/4 * y^2 + 3) * cos(2*x + 1 - E^y)
f = lambda X: np.sin(1/2 * X[0]**2 - 1/4 * X[1]**2 + 3) * np.cos(2 * X[0] + 1 - np.e**X[1])
```

#Aproximamos el valor del gradiente en un punto por su definición

```
def df(PUNTO):
    h = 0.01
    T = np.copy(PUNTO)
    grad = np.zeros(2)
    for it, th in enumerate(PUNTO):
        T[it] = T[it] + h
        grad[it] = (f(T) - f(PUNTO)) / h
    return grad
```

$$f'(a) = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h}$$

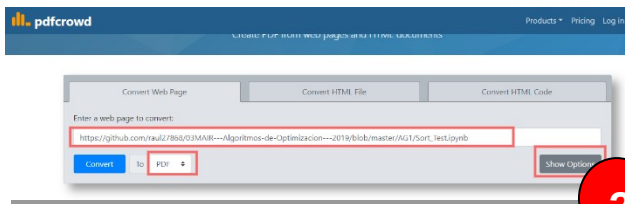
Aproximamos la derivada



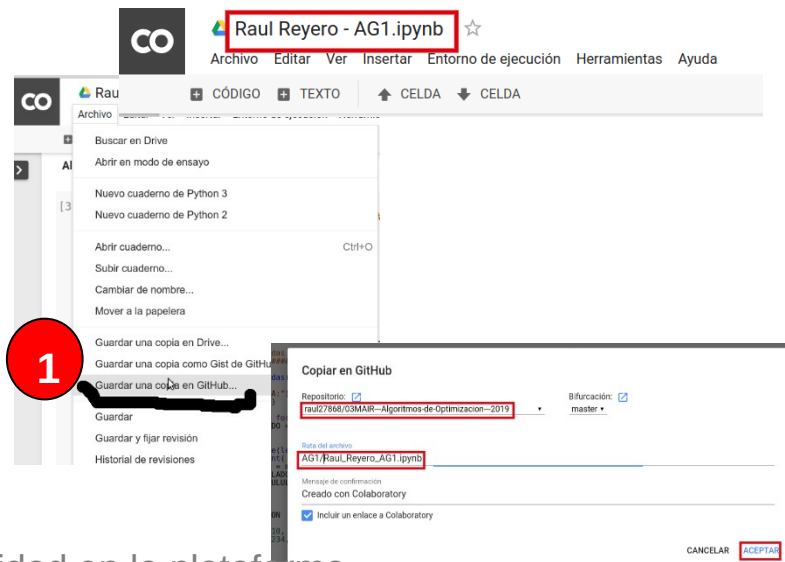
Finalizar la actividad. Grabar, subir a GitHub, Generar pdf (I)

- Guardar en GitHub
Repositorio: 03MAIR ---Algoritmos de Optimizacion
Ruta de Archivo con **AG2**

- Generar pdf (con <https://pdfcrowd.com>)



- Descargar pdf y adjuntar el documento generado a la actividad en la plataforma
 - Adjuntar .pdf en la actividad
 - URL GitHub en el texto del mensaje de la actividad



Actividades Guiadas.

- Si hay aportaciones, **añadir explicación**
- Si es la 2ª entrega, **añadir explicación**
- Si tienes dudas de que la 2ª entrega sea valida, **enviame un correo.**

The screenshot displays a user interface for a guided activity. On the left, under 'Instrucciones de la actividad', there is a download button for a file named 'github_com_pachon00_03MAIR_Algoritmos_de_optimizacion_blob_m(1).pdf'. On the right, under 'Detalles de la actividad', the 'CALIFICAR' section shows the current score as 8/10. Below this, a dropdown menu is open, highlighting 'Intento 2' with the date '16/02/19 12:08'. The 'ENVÍO' section shows the file name 'github_com_pachon00_03MAIR_Algoritmos_de_optimizacion_blob_m(1).pdf' and a large red box containing a thinking face emoji, indicating a pending submission or a placeholder for a message.

Ramificación y Poda. Practica

Análisis para **mejorar nota**:

- ¿Que 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 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 deja de ser una opción válida?



10/10

Vista de cuadrícula

Vista de lista

	Principiante	Competente	Muy competente
CE1. Dominio de técnicas avanzadas en machine learning y optimización.	0 (0,00 %) No ha logrado presentar el trabajo de la actividad.	6,4 (64,00 %) Ha logrado presentar algunas de las tareas del trabajo de la actividad.	8 (80,00 %) Ha logrado presentar todas las tareas del trabajo de la actividad.
CE6. Comprensión de los complejos fundamentos matemáticos en optimización computacional.	0 (0,00 %) No ha logrado presentar y resolver la tarea de aplicación de la actividad sobre la asignación de tareas usando técnicas de ramificación y poda	1 (10,00 %) Ha logrado presentar y resolver la tarea de aplicación de la actividad sobre la asignación de tareas usando técnicas de ramificación y poda en el apartado de complejidad	2 (20,00 %) No ha logrado presentar y resolver la tarea de aplicación de la actividad sobre la asignación de tareas usando técnicas de ramificación y poda en todos los apartados (complejidad y estudio de dimensiones mayores)

Próxima clase. VC4

Introducción a las metaheurísticas

1. Búsqueda aleatoria
2. Búsqueda basada en trayectorias
3. Métodos basados en trayectorias. Búsqueda Tabú
4. Métodos basados en trayectorias. Recocido Simulado
5. Métodos constructivos. Multiarranque
6. Métodos constructivos. GRASP
7. Métodos constructivos. Colonia de hormigas

Ampliación de conocimientos y habilidades

■ Bibliografía

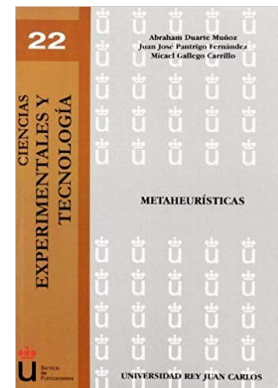
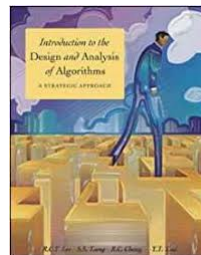
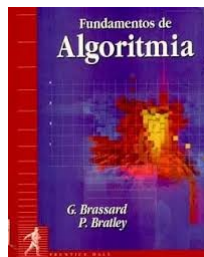
-Brassard, G., y Bratley, P. (1997). Fundamentos de algoritmia. ISBN 13: 9788489660007

-Guerequeta, R., y Vallecillo, A. (2000). Técnicas de diseño de algoritmos.(
<http://www.lcc.uma.es/~av/Libro/indice.html>)

-Lee, R. C. T., Tseng, S. S., Chang, R. C., y Tsai, Y. T. (2005). Introducción al diseño y análisis de algoritmos. ISBN 13: 9789701061244

-**Abraham Duarte,.. Metaheurísticas. ISBN 13: 9788498490169**

• Practicar



Gracias

raul.reyero@campusviu.es