Algoritmos_R2_Augusto_IbanezGarcia

February 27, 2024

1 Algoritmos de optimización - Reto 2

Nombre: Augusto Javier Ibañez Garcia
Github: https://github.com/cibergus/VIU-AlgOptimizacion

[1]:
```
# INSTALLS & IMPORTS
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 random
```

1.1 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)

1.1.1 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.

- Resuelve el problema del descenso por el rio utilizando la técnica de optimización que consideres más adecuada.

[2]:
```
# Viaje por el rio -> Versión Programación Dinámica

# Embarcaderos y costos conocidos (datos extraídos de la imagen)
n_embarcaderos = 7
```

```python
caminos_costos = [(0, 1, 5), (0, 2, 4), (0, 3, 11),
                  (1, 3, 2), (1, 4, 3),
                  (2, 3, 1), (2, 4, 5), (2, 5, 6),
                  (3, 4, 3), (3, 5, 4), (3, 6, 11),
                  (4, 5, 4), (4, 6, 11),
                  (5, 6, 3)
                 ]

# Inicialización costos y rutas
costos = np.full((n_embarcaderos, n_embarcaderos), np.inf)
rutas = np.full((n_embarcaderos, n_embarcaderos), None)

# Establecemos costos directos y cero para la misma posición
for i, j, costo in caminos_costos:
    costos[i][j] = costo
for i in range(n_embarcaderos):
    costos[i][i] = 0

# Relación de recurrencia y relleno de la tabla con ruta
for k in range(n_embarcaderos):
    for i in range(n_embarcaderos):
        for j in range(n_embarcaderos):
            if costos[i][j] > costos[i][k] + costos[k][j]:
                costos[i][j] = costos[i][k] + costos[k][j]
                rutas[i][j] = k

# Recuperamos y reconstruimos la ruta óptima utilizando la matriz de rutas
def reconstruir_ruta(rutas, origen, destino):
    # Si no hay ruta intermedia
    if rutas[origen][destino] is None:
        return []
    k = rutas[origen][destino]
    return reconstruir_ruta(rutas, origen, k) + [k] + reconstruir_ruta(rutas,␣
 ↪k, destino)

# Usamos la matriz de rutas para encontrar el camino óptimo del embarcadero 0␣
 ↪al 6
embarcadero_inicio = 0
embarcadero_final = 6
camino_optimo = [embarcadero_inicio] + reconstruir_ruta(rutas, 0, 6) +␣
 ↪[embarcadero_final]
costo_minimo = int(costos[embarcadero_inicio][embarcadero_final])
print(f"El camino óptimo del embarcadero {embarcadero_inicio} al␣
 ↪{embarcadero_final} con un costo de {costo_minimo} es: {camino_optimo}")
```

```
El camino óptimo del embarcadero 0 al 6 con un costo de 12 es: [0, 2, 3, 5, 6]
```

1.2 Descenso del gradiente

[3]:
```
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 :

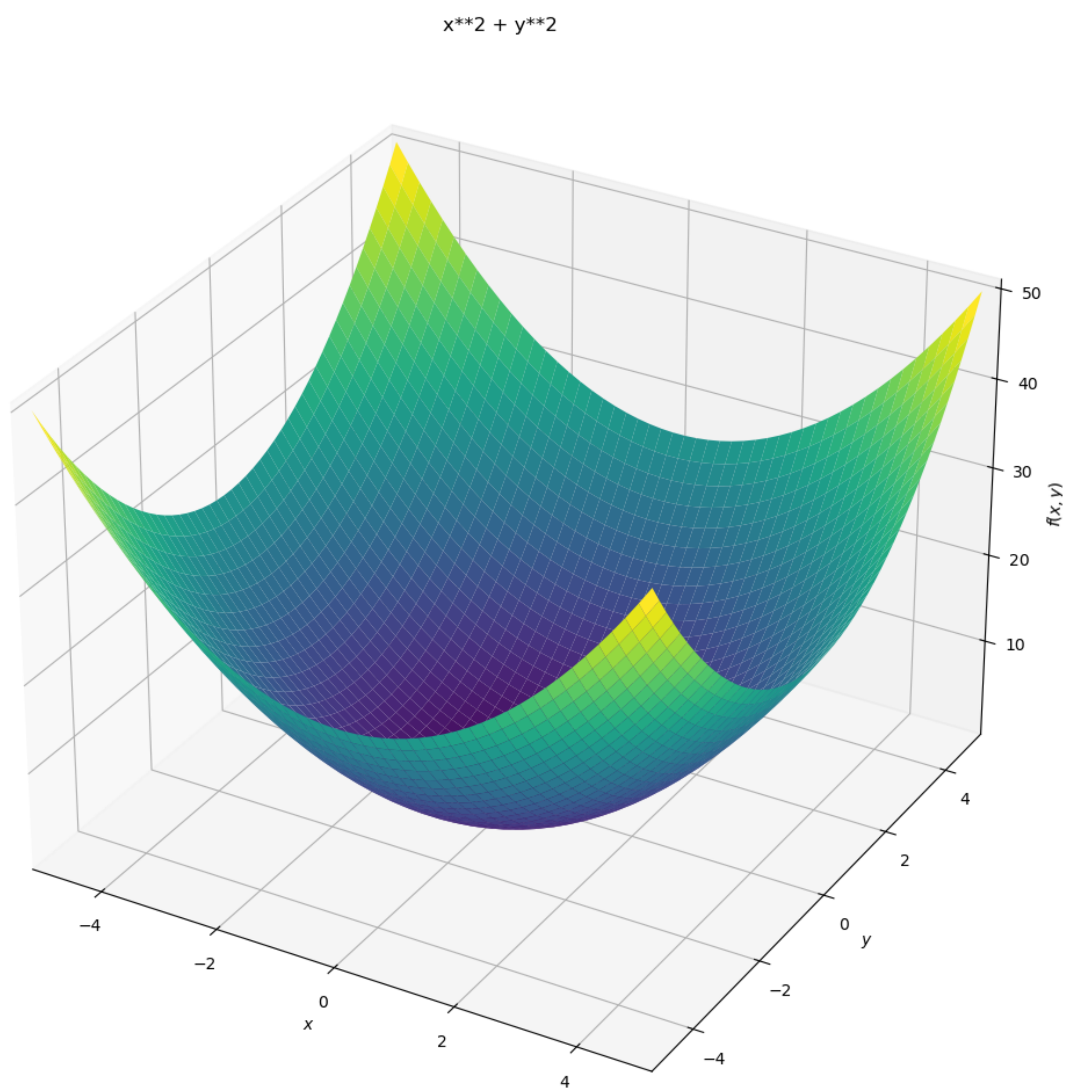
$$f(x) = x^2 + y^2$$

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

[4]:
```
#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])
```

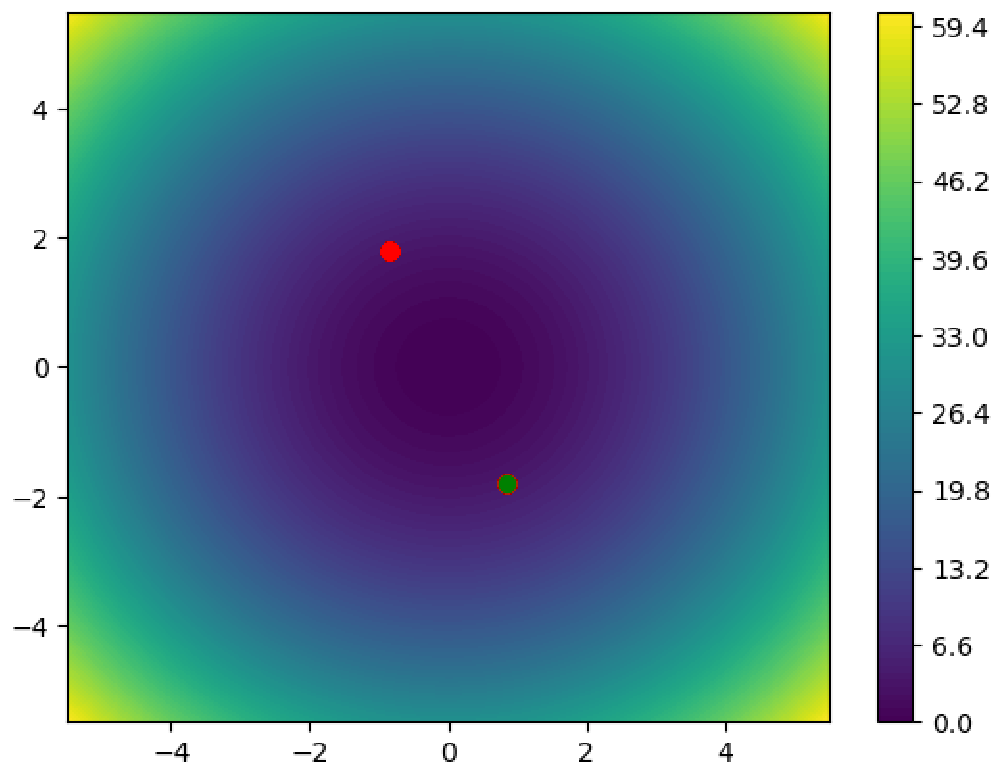
[4]: [2, 4]

[5]:
```
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));
```



[6]:
```
#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(1000):
  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: [0.846680214309151, -1.809588349157746] 3.9914773787100466
```

1.3 Reto

Optimizar la función siguiente mediante el algoritmo por descenso del gradiente.

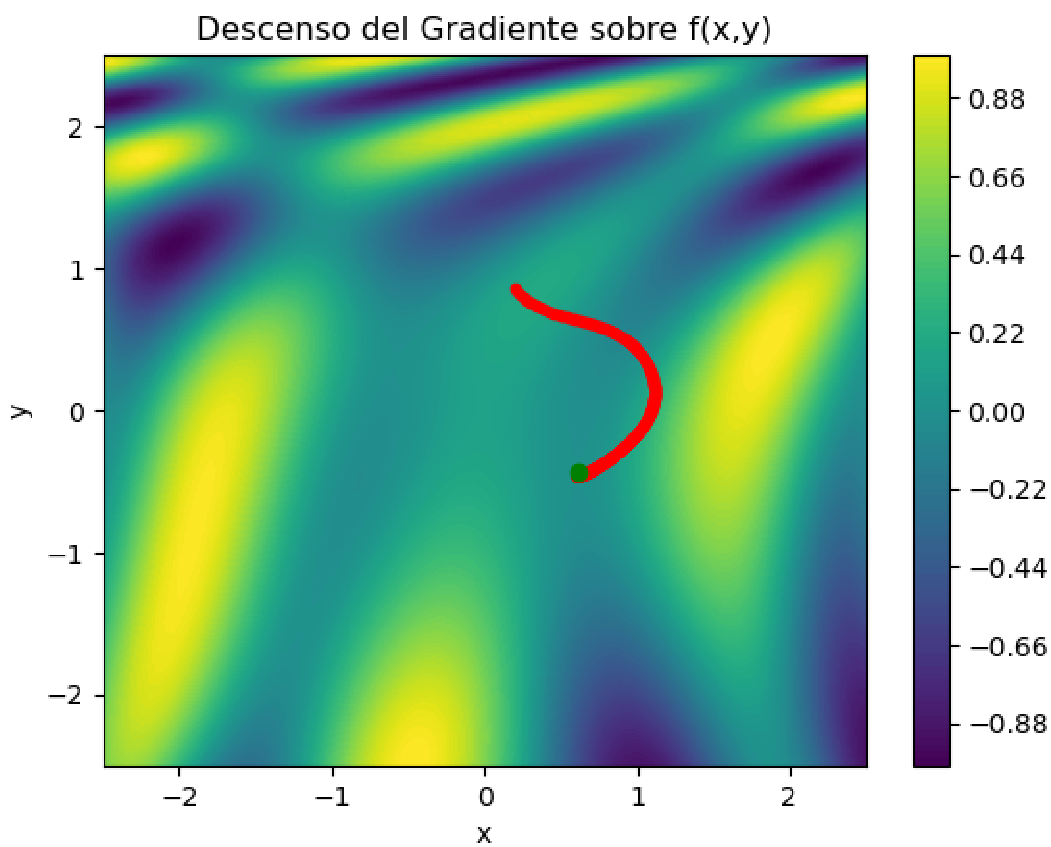
$$f(x) = sin(1/2 * x^2 - 1/4 * y^2 + 3) * cos(2 * x + 1 - e^y)$$

[7]:
```
#Definimos la funcion
import math
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]) )
```

[8]:
```
# Definimos el gradiente de la función
def df(X):
    x, y = X
    dfdx = np.cos(1/2 * x**2 - 1/4 * y**2 + 3) * np.cos(2*x + 1 - np.exp(y)) + \
           (x * np.sin(1/2 * x**2 - 1/4 * y**2 + 3) * np.sin(2*x + 1 - np.
↪exp(y)))
    dfdy = (-1/2 * y) * np.cos(1/2 * x**2 - 1/4 * y**2 + 3) * np.cos(2*x + 1 -␣
↪np.exp(y)) - \
           np.exp(y) * np.sin(1/2 * x**2 - 1/4 * y**2 + 3) * np.sin(2*x + 1 -␣
↪np.exp(y))
    return np.array([dfdx, dfdy])

# Fijamos la semilla + punto inicial aleatorio + tasa de aprendizaje
np.random.seed(0)
P = np.random.uniform(-2, 2, 2)
puntos = [P.copy()]
TA = 0.01

# Realizamos las iteraciones del descenso del gradiente
for _ in range(1000):
    grad = df(P)
    P -= TA * grad
    puntos.append(P.copy())
puntos = np.array(puntos)

# Preparamos los datos para el 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])
```

```python
# Solución y el mapa de niveles de Z
print(f"Solución aproximada: \nx = {P[0]}, y = {P[1]}, f(x,y) = {f(P)}\n")
plt.contourf(X, Y, Z, resolucion, cmap='viridis')
plt.colorbar()
plt.plot(puntos[:, 0], puntos[:, 1], 'r.-')  # Ruta en rojo
plt.plot(P[0], P[1], "o", c="green")  # Punto final en verde
plt.title('Descenso del Gradiente sobre f(x,y)')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

```
Solución aproximada:
x = 0.6091508268461311, y = -0.43017756771492505, f(x,y) = 6.710800650966057e-06
```



[ ]:

```

```