Material complementario

Prácticas didácticas para el estudio y comprensión de metaheurísticas utilizadas en la resolución de problemas de optimización difíciles

Investigadores participantes:

Adriana Menchaca Méndez [1]
Saúl Zapotecas Martínez [2]
Elizabeth Montero Ureta [3]
Carlos A. Coello Coello [4]
Katya Rodríguez Vázquez [5]
Sergio Rogelio Tinoco Martínez [1]

Alumnos participantes:

Jesús Armando Ortíz Peñafiel [1]
Angélica Nayeli Rivas Bedolla[1]

[1]Escuela Nacional de Estudios Superiores, Unidad Morelia, UNAM, México
[2]Universidad Autónoma Metropolitana, Unidad Cuajimalpa, México
[3]Facultad de Ingeniería, Universidad Andrés Bello, Chile
[4]Centro de Investigación y de Estudios Avanzados del IPN, Unidad Zacatenco, México
[5]Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas, UNAM, México

Trabajo realizado con el apoyo del Programa UNAM-DGAPA-PAPIME PE102320.

Agradecimientos

Agradecemos el apoyo de los alumnos y las alumnas de la Licenciantura en Tecnologías para la Información en Ciencias, de la ENES Unidad Morelia, por sus comentarios y ayuda en el mejoramiento de este material didáctico.

Índice general

Capítulo 1

Búsqueda Tabú (BT)

1.1. Problema de la mochila binario

El problema de la mochila binario (0/1 Knapsack problem) considera un conjunto finito de n objetos, donde cada objeto i tiene un valor p_i y un peso w_i, y una mochila que puede soportar hasta un peso determinado c. El objetivo es encontrar el subconjunto de objetos que puedan ser transportados en la mochila, maximizando el valor de la mochila. Formalmente se define como sigue:

$$\begin{array}{rll} \text{maximizar:} & f(\vec{x}) = \sum_{i=1}^{n} p_i \cdot x_i & \\ \text{tal que} & g_1(\vec{x}) = \sum_{i=1}^{n} w_i \cdot x_i \leq c & \\ & x_i \in \{0,1\} & i \in \{1,\ldots,n\} \end{array} \tag{1.1}$$

Para este ejemplo, consideremos que se tienen $n = 5$ objetos. Donde cada objeto tiene los siguientes valores $p = [5, 14, 7, 2, 23]$ y los siguientes pesos $w = [2, 3, 7, 5, 10]$. Además la mochila tiene una capacidad $c = 15$.

1.1.1. Principales componentes de BT

Librerías de **Python** que vamos a utilizar.

```
In [69]: import numpy
         import math
```

Datos de entrada.

```
In [70]: n=5
         p = [5,14,7,2,23]
         w = [2,3,7,5,10]
         c = 15
```

Representación de una solución

Cada solución es una cadena binaria de tamaño 5. Para facilitar la implementación en **Python**, lo tomaremos como una lista de 0's y 1's. Por ejemplo, la solución x='11001' es almacenada como:

```
In [71]: x = [1,1,0,0,1]
```

Solución inicial

Mientras no se exceda la capacidad de la mochila, se van introduciendo de manera aleatoria objetos a la mochila.

```
In [72]: def getInitialSolution(n, p, w, c):
             #Ningún objeto está en la mochila
             x = [0 for i in range(n)]
             weight_x = 0

             #Aleatoriamente elegimos el orden en el que intentaremos
             #introducir los objetos a la mochila
             objects = list(range(n))
             numpy.random.shuffle(objects)

             for o in objects:
                 #Intentamos introducir el objeto "o" a la mochila
                 if weight_x + w[o] <= c:
                     x[o] = 1
                     weight_x += w[o]

             return x

In [73]: x_0 = getInitialSolution(n, p, w, c)
         x_0

Out[73]: [1, 1, 0, 0, 1]
```

Función objetivo

```
In [74]: def f(p, x):
             profits = 0
             for i in range(len(x)):
                 profits += p[i]*x[i]

             return profits

In [75]: print ("f(x): \t", f(p, x))
         print ("f(x_0): ", f(p, x_0))
```

```
f(x): 42
f(x_0): 42
```

Restricción

```
In [76]: def g1(w, x):
             weight = 0
             for i in range(len(x)):
                 weight += w[i]*x[i]

             return weight
```

```
In [77]: print ("g1(w, x): \t", g1(w, x))
         print ("g1(w, x_o): \t", g1(w, x_0))
```

```
g1(w, x): 15
g1(w, x_o): 15
```

Movimiento

Altera el valor del elemento ubicado en la posición i de nuestra solución x.

```
In [78]: def bitflip(x, i):
             x_new = x.copy()
             if x[i] == 0:
                 x_new[i] = 1
             else:
                 x_new[i] = 0

             return x_new
```

```
In [79]: print("x: \t", x)
         x_new = bitflip(x, 2)
         print("x_new: \t", x_new)
```

```
x: [1, 1, 0, 0, 1]
x_new: [1, 1, 1, 0, 1]
```

Vecindario

El vecindario está definido como movimientos "bitflip" en cada una de las variables de la solución, teniendo en cuenta que solamente las soluciones factibles pueden ser parte del vecindario. En **Python** vamos a implementar el vecindario como una lista de listas, donde cada lista interna almacena una solución y su valor tanto de la función objetivo como de la restricción.

```
In [80]: def getNeighborhood(p, w, x, c):
             neighborhood = []
             for i in range(len(x)):
                 x_new = bitflip(x, i)
                 g1_x_new = g1(w, x_new)
                 #Si la solución creada es factible,
                 #la metemos al vecindario
                 if g1_x_new <= c:
                     neighborhood.append([x_new, f(p, x_new), g1_x_new, i])

             return neighborhood

In [81]: print("Vecindario de ", x)
         getNeighborhood(p, w, x, c)

Vecindario de  [1, 1, 0, 0, 1]


Out[81]: [[[0, 1, 0, 0, 1], 37, 13, 0],
          [[1, 0, 0, 0, 1], 28, 12, 1],
          [[1, 1, 0, 0, 0], 19, 5, 4]]
```

Es importante mencionar que para instancias grandes del problema (n's grandes) se debe cuidar la eficiencia. Por ejemplo, no es necesario recalcular completamente los valores de la función objetivo y la restricción de cada solución nueva se puede obtener a partir de la solución base y el cambio realizado.

Lista tabú

La lista tabú va a almacenar los valores que no son permitidos en ciertas posiciones de la solución y su "tiempo tabú". En **Python** utilizaremos un diccionario para guardar los elementos tabú. La llave del diccionario será la posición p y va estar asociada con el valor v que está prohibido en esa posición y con su tiempo tabú t. El siguiente ejemplo indica que en la posición 1 no puede haber un valor de 0 durante las próximas 2 iteraciones.

```
In [82]: tabu_list = {}
         tabu_list[1] = [0, 2]
         print (tabu_list)

{1: [0, 2]}
```

La lista tabú se va a modificar cada iteración actualizando el tiempo tabú de los elementos que ya se encontraban en ella y añadiendo un nuevo elemento.

```
In [83]: def updateTabuList(new_element, tabu_list):
             aux = []

             #Disminuimos en 1 el tiempo tabú de los elementos
             #que ya estaban en la lista tabú
             for key in tabu_list:
                 tabu_list[key][1] -= 1
                 if tabu_list[key][1] == 0:
                     aux.append(key)

             #Sacamos de la lista tabú los elementos con
             #tiempo tabú igual con 0
             for key in aux:
                 tabu_list.pop(key)

             #Agregamos el nuevo elemento a la lista tabú
             tabu_list[new_element[0]] = [new_element[1], new_element[2]]
```

```
In [84]: print(tabu_list)
         tabu_element = [3,1,2]
         updateTabuList(tabu_element, tabu_list)
         print(tabu_list)
         tabu_element = [2,0,2]
         updateTabuList(tabu_element, tabu_list)
         print(tabu_list)
```

```
{1: [0, 2]}
{1: [0, 1], 3: [1, 2]}
{3: [1, 1], 2: [0, 2]}
```

Vecindario reducido

El vecindario reducido se define como el conjunto de soluciones resultante de quitar del vecindario las soluciones generadas a partir de movimientos que se encuentran en la lista tabú.

```
In [85]: def getReducedNeighborhood(x, tabu_list, p, w, c):
             neighborhood = []
             for i in range(len(x)):
             #Si no está prohibido cambiar el valor de la posición i,
             #creamos una nueva solución haciendo bitflip en la posición i
                 if i not in tabu_list:
                     x_new = bitflip(x, i)
                     g1_x_new = g1(w, x_new)
                     #Si la solución creada es factible,
```

```
                #la metemos al vecindario
                if g1_x_new <= c:
                    neighborhood.append([x_new, f(p, x_new), g1_x_new, i])

        return neighborhood
```

1.1.2. Algoritmo completo de una BT simple

```
In [86]: #Obtenemos el índice de la mejor solución en el vecindario
         def getIndexBestNeighbor(neighborhood):
             best = 0
             for i in range(1, len(neighborhood)):
                 if neighborhood[i][1] >= neighborhood[best][1]:
                     best = i

             return best
```

```
In [87]: #Búsqueda tabú simple para el problema de la mochila binario
         def TabuSearch(num_ite, n, p, w, c):
             #Obtenemos la solución inicial
             x = getInitialSolution(n, p, w, c)
             f_x = f(p, x)
             g_x = g1(w, x)

             #Hasta ahora la solución inicial es la mejor
             #solución que se conoce
             x_best, f_best, g_best = x.copy(), f_x, g_x

             #Iniciamos con nuestra lista tabú vacía
             tabu_list = {}

             #Definimos el tiempo tabú
             tabu_time = n//2

             for k in range(num_ite):
                 #Obtenemos nuestro vecindario reducido
                 neighborhood =  getReducedNeighborhood(x, tabu_list, p, w, c)
                 #Si nuestro vecindario está vacío, ya no podemos movernos
                 #a otra solución y debemos terminar la búsqueda
                 if len(neighborhood) == 0:
                     break

                 #Nuestra siguiente solución es la mejor del vecindario
                 best_neighbor = getIndexBestNeighbor(neighborhood)
                 x = neighborhood[best_neighbor][0]
```

```
                f_x = neighborhood[best_neighbor][1]
                g_x = neighborhood[best_neighbor][2]

                #Verificamos si la nueva solución
                #es mejor que lo que conocemos
                if f_x >= f_best:
                    x_best, f_best, g_best = x.copy(), f_x, g_x

                #Actualizamos la lista tabú
                i = neighborhood[best_neighbor][3]
                tabu_element = [i, x[i], tabu_time]
                updateTabuList(tabu_element, tabu_list)

            return x_best, f_best, g_best


In [88]: def printSolution(x, f, g):
            print("f(x) = ", f, "\tg(x) = ", g)
            step = 25
            print("x = \n[", end="")
            for i in range(len(x)-1):
                if i != 0 and i % step == 0:
                    print()
                print(x[i], end=', ')
            print(x[-1], "]")


In [89]: n=5
         p = [5,14,7,2,23]
         w = [2,3,7,5,10]
         c = 15
         num_ite = 50

         x_best, f_best, g_best = TabuSearch(num_ite, n, p, w, c)
         printSolution(x_best, f_best, g_best)

f(x) =  42          g(x) =  15
x = [1, 1, 0, 0, 1 ]
```

Ejecutamos para una instancia aleatoria más grande del problema.

```
In [90]: n=50
         profits = list(range(30, 100))
         weights = list(range(20, 40))
```

```
        p = numpy.random.choice(profits, n)
        print("Arreglo con valores de cada objeto: \n", p)
        w = numpy.random.choice(weights, n)
        print("Arreglo con pesos de cada objeto: \n", w)
        c = numpy.random.randint((30*n)//2, 30*n)
        print("Capacidad de la mochila: ", c)
        num_ite = 1000

        print("Ejecución 1:")
        x_best, f_best, g_best = TabuSearch(num_ite, n, p, w, c)
        printSolution(x_best, f_best, g_best)
```

```
Arreglo con valores de cada objeto:
 [47 59 37 63 68 71 99 38 99 38 78 33 89 63 74 92 32 78 31 89 73 92 38 30
 38 30 66 68 45 30 72 73 41 51 88 65 85 76 65 40 67 62 96 68 30 81 67 42
 53 97]
Arreglo con pesos de cada objeto:
 [38 24 35 34 37 35 29 24 35 34 32 37 34 37 33 34 22 39 36 32 38 26 24 29
 20 32 26 39 29 22 22 33 22 24 21 23 22 32 39 38 38 37 34 27 34 22 29 37
 30 31]
Capacidad de la mochila:  1337
Ejecución 1:
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
```

Realizamos m ejecuciones de nuestro algoritmo y obtenemos la corrida que encontró la mejor solución, la corrida que encontró la peor solución, la media y la desviación estándar de las m corridas con respecto al valor de la función objetivo.

```
In [94]: m=21
         sol = []
         for i in range(m):
             print("Ejecución ", i, ": ")
             x_best, f_best, g_best = TabuSearch(num_ite, n, p, w, c)
             sol.append([f_best, g_best, x_best])
             printSolution(x_best, f_best, g_best)

         sol.sort()
         print("*************** Mejor solución ***************")
         printSolution(sol[-1][2], sol[-1][0], sol[-1][1])

         print("*************** Peor solución ****************")
```

```
        printSolution(sol[0][2], sol[0][0], sol[0][1])

        print("****************** Mediana ******************")
        med = m//2
        printSolution(sol[med][2], sol[med][0], sol[med][1])

        f_sol = [x[0] for x in sol]
        print("Media: ", numpy.mean(f_sol))
        print("Desviación estándar: ", numpy.std(f_sol))
```

```
Ejecución  0 :
f(x) =  2911         g(x) =  1336
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1 ]
Ejecución  1 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  2 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  3 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  4 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  5 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  6 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
```

```
Ejecución  7 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  8 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  9 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  10 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  11 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  12 :
f(x) =  2886         g(x) =  1316
x =
[1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  13 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  14 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  15 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
```

```
Ejecución  16 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  17 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  18 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  19 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Ejecución  20 :
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
*************** Mejor solución ***************
f(x) =  2911         g(x) =  1336
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1 ]
*************** Peor solución ****************
f(x) =  2886         g(x) =  1316
x =
[1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
****************** Mediana *******************
f(x) =  2891         g(x) =  1329
x =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
Media:  2891.714285714286
Desviación estándar:  4.4416090728994355
```

Capítulo 2

Recocido Simulado (RS)

2.1. Problema del agente viajero

El problema del agente viajero (Travelling salesman problem) considera un conjunto finito de n ciudades y la distancia entre cada par de ellas. El objetivo es encontrar el camino más corto que visite cada ciudad exactamente una vez y regrese a la ciudad de origen. Formalmente se define como sigue:

$$\begin{array}{rl} \text{minimizar:} & f(x) = d(x_n, x_1) + \sum_{i=1}^{n-1} d(x_i, x_{i+1}) \\ \text{tal que} & x_i \in \{1, 2, \cdots, n\} \end{array} \tag{2.1}$$

donde $d(x_i, x_j)$ es la distancia de ir de la ciudad x_i a la ciudad x_j, n es el número de ciudades y x es una permutación de las n ciudades.

2.1.1. Principales componentes de RS

Librerías de **Python** que vamos a utilizar.

```
In [81]: import numpy
         import math
```

Datos de entrada.

```
In [82]: n = 5
         dist_matrix = [
                         [0,49,30,53,72],
                         [49,0,19,38,32],
                         [30,19,0,41,98],
                         [53,38,41,0,52],
                         [72,32,98,52,0],
                        ]
```

Representación de una solución

Cada solución es una permutación de las n ciudades. En **Python** será una lista de n elementos. Nombraremos a las ciudades como 0, 1, $\cdots$, $n-1$ para que coincidan con los índices de nuestra lista. El primer elemento siempre será 0 porque es la ciudad de partida.

```
In [83]: x = [0,4,2,1,3]
```

Solución inicial

Utilizaremos una estrategia voraz: Empezamos en la ciudad 0, posteriormente revisamos las 4 ciudades restantes y elegimos la que tenga una distancia menor a la ciudad 0. Repitemos el proceso hasta tener nuestra permutación.

```
In [84]: def getNextCity(x, dist_matrix, n):
             current_city = x[-1]
             min_dist = max(dist_matrix[current_city])

             for i in range(1, n):
                 if (i not in x) and (dist_matrix[current_city][i] < min_dist):
                     min_city = i
                     min_dist = dist_matrix[current_city][i]

             return min_city

         def getInitialSolution(n, dist_matrix):
             x = [0]

             while len(x) != n:
                 x.append(getNextCity(x, dist_matrix, n))

             return x

In [85]: x_0 = getInitialSolution(n, dist_matrix)
         print ("x_0: ", x_0)

x_0:  [0, 2, 1, 4, 3]
```

Función objetivo

```
In [86]: def f(n, dist_matrix, x):
             cost = dist_matrix[x[-1]][0]

             for i in range(n-1):
```

```
                cost += dist_matrix[x[i]][x[i+1]]

            return cost
```

```
In [87]: print("f(x)\t= ", f(n, dist_matrix, x))
         print("f(x_0)\t= ", f(n, dist_matrix, x_0))
```

```
f(x)        =  280
f(x_0)        =  186
```

Vecindario

El vecindario se genera eligiendo aleatoriamente una posición i de la permutación. Posteriormente, se generan $N-2$ soluciones moviendo la ciudad que está en la posición p a cualquiera de las otras posiciones posibles. En **Python** utilizaremos una lista de listas para almacenar el vecindario. Cada lista va a tener una permutación y su valor en la función objetivo.

```
In [88]: def getNeighborhood(x, n, dist_matrix):
             aux_x = x.copy()
             #Elegimos una posición aleatoria
             i = numpy.random.randint(1, len(x))
             print ("Posición aleatoria: ", i)
             city = aux_x.pop(i)

             neighborhood = []
             for j in range(1, len(x)):
                 if j != i:
                     new_sol = aux_x.copy()
                     new_sol.insert(j, city)
                     neighborhood.append([new_sol, f(n, dist_matrix, new_sol)])

             return neighborhood
```

```
In [90]: print ("Solución actual: ", x_0)
         neighborhood = getNeighborhood(x_0, n, dist_matrix)
         print ("Vecindario:")
         for e in neighborhood:
             print(e)
```

```
Solución actual:  [0, 2, 1, 4, 3]
Posición aleatoria:  4
Vecindario:
[[0, 3, 2, 1, 4], 217]
[[0, 2, 3, 1, 4], 213]
```

```
[[0, 2, 1, 3, 4], 211]
```

Dado que elegiremos de manera aleatoria una solución del vecindario, no es necesario generar todo el vecindario. Por lo tanto, crearemos una función que nos permita obtener una única solución.

```
In [91]: def nextSolution(x, n, dist_matrix):
             aux_x = x.copy()
             #Elegimos una posición aleatoria
             i = numpy.random.randint(1, len(x))
             #print ("Posición aleatoria: ", i)
             city = aux_x.pop(i)

             #Elegimos una nueva posición
             j = numpy.random.randint(1, len(x))
             while i == j:
                 j = numpy.random.randint(1, len(x))

             aux_x.insert(j, city)

             return aux_x, f(n, dist_matrix, aux_x)
```

Temperatura

Para este ejemplo vamos a utilizar una función lineal para disminuir la temperatura.

```
In [92]: def updateTemperature(t):
             return 0.9*t
```

```
In [93]: t = 1000
         print ("Temperatura: ", t)
         new_t = updateTemperature(t)
         print ("Temperatura: ", new_t)
```

```
Temperatura:  1000
Temperatura:  900.0
```

2.1.2. Algoritmo completo de un RS simple

```
In [94]: #Recocido simulado simple para el problema del agente viajero
         def SimulatedAnnealing(t_0, t_f, n, dist_matrix):
             x_0 = getInitialSolution(n, dist_matrix)
             x = x_0
             fx = f(n, dist_matrix, x)
```

```
            t = t_0
            x_best = x.copy()
            f_best = fx

            print("Solución inicial: ", x, fx)
            while t >= t_f:
                new_x, new_f = nextSolution(x, n, dist_matrix)

                if new_f <= f_best:
                    x_best = new_x.copy()
                    f_best = new_f

                if new_f < fx or numpy.random.random() < math.exp(-1.0*(new_f-fx)/t):
                    x = new_x
                    fx = new_f

                t = updateTemperature(t)
                #print(x, fx)

            return x_best, f_best
```

Ejecutamos el algoritmo.

```
In [99]: n = 5
         dist_matrix = \
         [\
         [0,49,30,53,72],\
         [49,0,19,38,32],\
         [30,19,0,41,98],\
         [53,38,41,0,52],\
         [72,32,98,52,0],\
         ]

         x, fx = SimulatedAnnealing(10000, 0.1, n, dist_matrix)
         print("Solución encontrada: ", x, fx)
```

```
Solución inicial:  [0, 2, 1, 4, 3] 186
Solución encontrada:  [0, 2, 1, 4, 3] 186
```

Ejecutamos para una instancia más grande del problema.

```
In [102]: n = 10
          dist_matrix = \
          [\
          [0,49,30,53,72,19,76,87,45,48],\
```

```
            [49,0,19,38,32,31,75,69,61,25],\
            [30,19,0,41,98,56,6,6,45,53],\
            [53,38,41,0,52,29,46,90,23,98],\
            [72,32,98,52,0,63,90,69,50,82],\
            [19,31,56,29,63,0,60,88,41,95],\
            [76,75,6,46,90,60,0,61,92,10],\
            [87,69,6,90,69,88,61,0,82,73],\
            [45,61,45,23,50,41,92,82,0,5],\
            [48,25,53,98,82,95,10,73,5,0],\
            ]

            x, fx = SimulatedAnnealing(100000, 0.01, n, dist_matrix)
            print("Solución encontrada ", x, fx)
```

```
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Solución encontrada  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
```

Realizamos m ejecuciones de nuestro algoritmo y obtenemos la corrida que encontró la mejor solución, la corrida que encontró la peor solución, la media y la desviación estándar de las m corridas con respecto al valor de la función objetivo.

```
In [103]: m=21
          t_0 = 100000
          t_f = 0.01
          sol = []
          for i in range(m):
              print("Ejecución ", i, ": ")
              x_best, f_best, = SimulatedAnnealing(t_0, t_f, n, dist_matrix)
              sol.append([f_best, x_best])
              print(x_best, f_best)

          sol.sort()
          print("*************** Mejor solución ***************")
          print(sol[0][0], sol[0][1])

          print("*************** Peor solución ****************")
          print(sol[-1][0], sol[-1][1])

          print("****************** Mediana *******************")
          med = m//2
          print(sol[med][0], sol[med][1])

          f_sol = [x[0] for x in sol]
          print("Media: ", numpy.mean(f_sol))
          print("Desviación estándar: ", numpy.std(f_sol))
```

```
Ejecución  0 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  1 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  2 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  3 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  4 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  5 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  6 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  7 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  8 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 1, 4, 7, 2, 6, 9, 8, 3, 5] 248
Ejecución  9 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  10 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  11 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  12 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  13 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  14 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
```

```
Ejecución  15 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  16 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  17 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  18 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  19 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
Ejecución  20 :
Solución inicial:  [0, 5, 3, 8, 9, 6, 2, 7, 1, 4] 271
[0, 7, 2, 6, 9, 8, 3, 4, 1, 5] 271
*************** Mejor solución ***************
248 [0, 1, 4, 7, 2, 6, 9, 8, 3, 5]
*************** Peor solución ****************
271 [0, 7, 2, 6, 9, 8, 3, 4, 1, 5]
****************** Mediana *******************
271 [0, 5, 3, 8, 9, 6, 2, 7, 1, 4]
Media:  269.9047619047619
Desviación estándar:  4.89805366499954
```

Recordemos que esta es la versión más simple de un RS, se pueden hacer cambios en el diseño para mejorar la eficacia del algoritmo.

Capítulo 3

Programación Evolutiva (PE)

3.1. Función de Beale

La función de Beale es una función de dos variables y está definida como sigue:

$$\text{mín } f(x_1, x_2) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + (2.625 - x_1 + x_1 x_2^3)^2 \quad (3.1)$$

donde $-4.5 \leq x_1, x_2 \leq 4.5$. El mínimo global está en $x^* = (3, 0.5)$ y $f(x^*) = 0$.

3.1.1. Principales componentes de PE

Librería de **Python** que vamos a utilizar.

```
In [1]: import numpy as np
```

Datos de entrada.

```
In [2]: n = 2 #Número de variables de decisión
```

Representación de un individuo

Un arreglo de valores flotantes de tamaño 4. Las primeras dos componentes corresponden a las dos variables del problema. Las siguentes dos componentes corresponden a los tamaños de paso para la mutación de cada una de las variables del problema.

```
In [3]: i = np.array([-1.5, 2.8, 0.3, 0.15])
```

Aptitud

La aptitud de un individuo i la definimos como $F_a(i) = -f(x_1^{(i)}, x_2^{(i)})$, donde f es la función de Beale.

```
In [4]: def f(x):
            x1, x2 = x[0], x[1]
            term1 = (1.5 - x1 + x1*x2)**2
            term2 = (2.25 - x1 + x1*(x2**2))**2
            term3 = (2.625 - x1 + x1*(x2**3))**2

            return term1 + term2 + term3

        def fitness(x):
            return -f(x)

In [5]: print ("x = (%s, %s)\t-f(x) = %s\t"%(i[0], i[1], fitness(i[:2])))

x = (-1.5, 2.8)         -f(x) = -895.2129089999994
```

Población inicial

Por cada individuo requerido, se utiliza una distribución uniforme para generar números aleatorios. Primero se generan dos números en el intervalo $[-4.5, 4.5]$ para crear las variables de decisión. Posteriormente, se generan dos números en el intervalo $(0, 1)$ para crear los tamaños de paso.

```
In [6]: def getInitialPopulation(mu, n):
            parents = []
            for i in range(mu):
                #Generamos un individuo
                p = np.concatenate((np.random.uniform(-4.5, 4.5, n),
                np.random.uniform(0, 1, n)))
                #Calculamos la aptitud del individuo
                p = [p, fitness(p[:n])]
                #Agregamos al individuo a la población "parents"
                parents.append(p)

            #Regresamos la población generada
            return parents

In [7]: mu = 100 #Tamaño de la población
        population = getInitialPopulation(mu, n)
        print("Tamaño de la población:", len(population))
        print("Primer individuo generado: ", population[0])

Tamaño de la población: 100
Primer individuo generado:
[array([-3.57073909,  4.48033254,  0.04512549,  0.43092328]), -103643.736219691]
```

Mutación y autoadaptación

Para mutar al individuo, se necesita primero mutar los valores de los tamaño de paso de la siguiente forma:

$$\sigma_1' = \sigma_1 \cdot (1 + \alpha \cdot N(0,1))$$
$$\sigma_2' = \sigma_2 \cdot (1 + \alpha \cdot N(0,1))$$

verificamos que los valores de σ_1' y σ_2' no sean menores que ε_0, en caso de que sí, se dejarán en ε_0. Posteriormente, utilizamos los nuevos σ para mutar las variables de decisión como sigue:

$$x_1' = x_1 + \sigma_1' \cdot N(0,1)$$
$$x_2' = x_2 + \sigma_2' \cdot N(0,1)$$

Finalmente, construimos el individuo hijo:

$$i' = [x_1', x_2', \sigma_1', \sigma_2']$$

Cuando se mutan las variables de decisión, se debe revisar que estén dentro del rango que indica el problema. En caso de que no, se le asignará el valor del límite que rebasó.

```
In [10]: def mutation(i, n, alpha, epsilon):
             #Mutamos los últimos dos componentes de nuestro arreglo
             #(tamaños de paso)
             mutation_sigma = i[n:]*(1+(alpha*np.random.normal(0, 1, n)))
             #Verificamos que los nuevos valores no sean menores a épsilon
             mutation_sigma[mutation_sigma < epsilon] = epsilon

             #Mutamos las variables de decisión a partir de las mutaciones
             mutation_x = i[:n] + (mutation_sigma*np.random.normal(0, 1, n))

             #Revisamos que estén dentro de los límites
             mutation_x[mutation_x < -4.5] = -4.5
             mutation_x[mutation_x >  4.5] =  4.5

             #Creamos el nuevo Individuo y lo devolvemos
             return [np.concatenate((mutation_x, mutation_sigma)),
             fitness(mutation_x)]
```

```
In [11]: #Parámetros para la mutación
         alpha = 0.2
         epsilon = 0.01

         child = mutation(population[0][0], n, alpha, epsilon)
         print(child)
```

```
[array([-3.46492722,  3.83356143,  0.06230124,  0.43211314]), -37879.09011226479]
```

3.1.2. Algoritmo completo de PE simple

A partir de 100 padres, vamos a generar 100 hijos. Unimos ambas poblaciones y seleccionamos los 100 mejores individuos de acuerdo a su aptitud.

```
In [12]: def EvolutionaryProgramming(n, mu, G, alpha, epsilon):
             parents = getInitialPopulation(mu, n)

             for t in range(num_gen):
                 new_gen = parents.copy()

                 for parent in parents:
                     #Creamos un hijo
                     child = mutation(parent[0], n, alpha, epsilon)
                     #Agregamos al hijo a la nueva generación
                     new_gen.append(child)

                 #Ordenamos del peor individuo al mejor individuo
                 #(del menor valor de aptitud al mayor)
                 new_gen = sorted(new_gen, key=lambda individual: individual[-1])
                 #Nos quedamos con los mu mejores individuos
                 new_gen = new_gen[mu:]

                 parents = new_gen.copy()

             #devolvemos el mejor
             return parents[-1][0], -parents[-1][1]
```

```
In [13]: n = 2
         mu = 100
         G = 200
         alpha = 0.2
         epsilon = .01
```

```
In [14]: sol, fx = EvolutionaryProgramming(n, mu, G, alpha, epsilon)
         print(f"x: {sol[:n]}\nsigma: {sol[n:]}\nf(x): {fx}")
```

```
x: [2.99992434 0.50001577]
sigma: [0.01       0.01466159]
f(x): 2.8418807729833617e-08
```

Realizamos m ejecuciones de nuestro algoritmo y obtenemos la corrida que encontró la mejor solución, la corrida que encontró la peor solución, la corrida que se encuentra en la mediana de las m corridas, la media y la desviación estándar de las m corridas con respecto al valor de la función objetivo.

```
In [15]: m = 21
         sol =[]
         for m in range(m):
             x, fx = EvolutionaryProgramming(n, mu, num_gen, alpha, epsilon)
             sol.append([x, fx])

         sol = sorted(sol, key=lambda individuo: individuo[-1])
         print("*************** Mejor solución ***************")
         print(f"x: {sol[0][0][:n]}\nsigma: {sol[0][0][n:]}\n
         f(x): {sol[0][-1]}")

         print("\n*************** Peor solución ****************")
         print(f"x: {sol[-1][0][:n]}\nsigma: {sol[-1][0][n:]}\n
         f(x): {sol[-1][-1]}")

         print("\n****************** Mediana *******************")
         print(f"x: {sol[m//2][0][:n]}\nsigma: {sol[m//2][0][n:]}\n
         f(x): {sol[m//2][-1]}")

         f_sol = [x[-1] for x in sol]
         print("\nMedia: ", np.mean(f_sol))
         print("\nDesviación estándar: ", np.std(f_sol))
```

```
*************** Mejor solución ***************
x: [2.99971315 0.49991913]
sigma: [0.02772399 0.01081308]
f(x): 1.5359162146571287e-08

*************** Peor solución ****************
x: [2.999368   0.49990788]
sigma: [0.03042325 0.02163403]
f(x): 1.6027470532192182e-07

****************** Mediana *******************
x: [2.99950765 0.49988047]
sigma: [0.01059131 0.01625244]
f(x): 3.897568350940374e-08

Media:  5.586687828982277e-08

Desviación estándar:  3.49564794851159e-08
```

Recordemos que esta es la versión más simple de PE, se pueden hacer cambios en el diseño para mejorar la eficacia del algoritmo.

Capítulo 4

Estrategias Evolutivas (EE)

4.1. Función de Ackley

La función de Ackley es una función de n variables y está definida como sigue:

$$\text{mín } f(\vec{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n}x_i^2}\right) - exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + e \qquad (4.1)$$

donde $30 \leq x_i \leq 30$. El mínimo global está en $x_i = 0$ y $f(\vec{x}) = 0$.

4.1.1. Principales componentes de EE

Librerías de **Python** que vamos a utilizar.

```
In [3]: import numpy as np
        from copy import copy
        from math import e, exp, sqrt, pi, cos
        from random import choices
```

Datos de entrada.

```
In [4]: n = 2 #Número de variables de decisión
```

Representación de un individuo

Cada individuo será visto como un arreglo de valores flotantes de tamaño $n + 1$. Las primeras n componentes corresponden a las variables de decisión del problema y la siguiente componente corresponde al tamaño de paso para la mutación de las variables del problema.

```
In [5]: #Si n = 2
        i = np.array([-10.3, 7.84, 0.84])
        i

Out[5]: array([-10.3 ,   7.84,   0.84])
```

Aptitud

La aptitud de un individuo i la definimos como $F_a(i) = -f(x_1^{(i)}, x_2^{(i)})$, donde f es la función de Ackley.

```
In [7]: def f(x):
            part1 = 20*exp(-0.2*sqrt(np.average(x**2)))
            part2 = exp(np.average(np.array([cos(i) for i in  2*pi*x])))
            return -part1 - part2 + 20 + e

        def fitness(x):
            return -f(x)
```

```
In [8]: print ("x = (%s, %s)\t-f(x) = %s\t"%(i[0], i[1], fitness(i[:2])))
```

```
x = (-10.3, 7.84)        -f(x) = -18.39186319125107
```

Población inicial

Por cada individuo requerido, se utiliza una distribución uniforme para generar números aleatorios. Primero se generan n números en el intervalo $[-30, 30]$, cada uno corresponde a una variable de decisión. Posteriormente, un número aleatorio en el intervalo $(0, 1)$ que corresponde al tamaño de paso.

```
In [15]: def getInitialPopulation(mu, n):
             population = []
             for i in range(mu):
                 #Generamos un individuo
                 p = np.concatenate((np.random.uniform(-30, 30, n),
                 np.random.uniform(0, 1, 1)))
                 #Calculamos la aptitud del individuo
                 p = [p, fitness(p[:n])]
                 #Agregamos al individuo a la población "parents"
                 population.append(p)

             return population
```

```
In [16]: mu = 100 #Tamaño de la población
         population = getInitialPopulation(mu, n)
         population[0]
```

```
Out[16]: [array([-18.65372343,  -9.19726163,   0.87925505]),
-20.776871049837613]
```

Cruza

Recombinación discreta local. Utilizando una distribución uniforme, generamos un número aleatorio en el intervalo $(0, 1)$ por cada componente de nuestro individuo. Recordemos que las primeras n componentes del individuo corresponden a sus variables de decisión y la última componente a su tamaño de paso. Si el valor es menor a 0.5, el hijo se queda con el valor del primer padre. De lo contrario, el hijo se queda con el valor del segundo padre.

```
In [18]: def localDiscreteCrossover(parent_1, parent_2):
             N = len(parent_1)
             #Creamos un vector con N componentes en el intervalo
             # (0,1) con distribución uniforme
             z = np.random.uniform(0, 1, N)
             child = np.array([0.0]*N)
             #Asignamos el valor del padre correspondiente
             for i in range(N):
                 if z[i] < 0.5:
                     child[i] = parent_1[i]
                 else:
                     child[i] = parent_2[i]

             #Regresamos el individuo generado
             return child
```

```
In [21]: i1 = [-10.3, 7.84, 0.84]
         i2 = [2.4, -3.84, 0.98]
         new_i = localDiscreteCrossover(i1,i2)
         print(new_i)
```

```
[-10.3   -3.84   0.84]
```

Mutación

Para mutar un individuo, se necesita mutar primero el tamaño de paso:

$$\sigma' = \sigma \cdot e^{\tau \cdot N(0,1)}$$

Si el valor de σ' es menor a ε_0, hacemos que $\sigma' = \varepsilon_0$. Posteriormente, utilizamos σ' para mutar las variables de decisión. Si $n = 2$, nuestras variables de decisión mutadas quedan de la siguiente forma:

$$x_1' = x_1 + \sigma' \cdot N(0,1)$$
$$x_2' = x_2 + \sigma' \cdot N(0,1)$$

Revisamos que las variables de decisión generadas estén dentro del rango que indica el problema. En caso de que no, se les asigna el valor del límite que rebasó. Finalmente, construimos el individuo hijo:

$$i' = [x_1', x_2', \sigma']$$

```
In [23]: def mutation(i, n, epsilon, tau):
             #Mutamos el tamaño de paso
             mutation_sigma = i[-1]*(e**(tau*np.random.normal(0, 1, 1)))
             #Verificamos que el nuevo valor no sea menor a épsilon
             if mutation_sigma[0] < epsilon:
                 mutation_sigma[0] = epsilon

             #Mutamos las variables de decisión
             mutation_x = i[:n] + (mutation_sigma[0]*np.random.normal(0, 1, n))

             #Revisamos que estén dentro de los límites
             mutation_x[mutation_x < -30] = -30
             mutation_x[mutation_x >  30] =  30

             return np.concatenate((mutation_x, mutation_sigma))
```

```
In [24]: tau = 1/sqrt(2) #Tasa de aprendizaje
         epsilon = 0.01 #mínimo valor de sigma
         i = [-10.3, 7.84, 0.84]
         new_i = mutation(i, n, epsilon, tau)
         print(new_i)
```

```
[-10.15890813   7.69605627   0.4825125 ]
```

Crear hijo usando cruza y mutación

```
In [25]: def createChild(parent_1, parent_2, n, epsilon, tau):
             new_ind = localDiscreteCrossover(parent_1, parent_2)
             new_ind = mutation(new_ind, n, epsilon, tau)

             #Regresamos el nuevo individuo, junto con su valor de aptitud
             return [new_ind, fitness(new_ind[:n])]
```

```
In [26]: createChild(population[0][0], population[1][0], n, epsilon, tau)
```

```
Out[26]: [array([  9.79058706, -19.6921706 ,   1.24097226]), -20.876622230156865]
```

4.1.2. Algoritmo completo de (μ, λ) - EE

Se usa una selección (μ, λ) donde μ=100 y λ=700. Se elegirán dos padres al azar para generar un hijo nuevo y este proceso se llevará a cabo 700 veces. De los hijos resultantes, se elegirán a los 100 mejores que pasarán a la siguiente generación.

```
In [27]: def EvolutionStrategies(n, G, epsilon=0.01, tau=None, mu_=100,
lambda_=700):
             #Si el usuario no define tau,
             #se usa el valor recomendado en la literatura
             if tau == None:
                 tau = 1/sqrt(n)

             population = getInitialPopulation(mu_, n)
             #Generamos los índices de los individuos para poder elegirlos
             #aleatoriamente
             population_idx = range(mu_)

             #Repetimos el proceso por G generaciones
             for _ in range(G):
                 offspring = []
                 #Generamos lambda_ hijos
                 for i in range(lambda_):
                     #Seleccionamos a los padres
                     parents_idx = choices(population_idx, k=2)
                     parent_1 = population[population_idx[0]][0]
                     parent_2 = population[population_idx[1]][0]
                     #Creamos el hijo y lo añadimos a la población de hijos
                     offspring.append( createChild(parent_1, parent_2,
                     n, epsilon, tau) )

                 #Ordenamos del mejor individuo al peor individuo
                 offspring.sort(key=lambda x: x[-1], reverse=True)
                 #Seleccionamos a los mu_ mejores
                 population = offspring[:mu_].copy()


             #Regresamos el mejor individuo
             return population[0][0], -population[0][1]
```

```
In [28]: n = 2 #Número de variables/dimensiones
         mu_ = 100 #tamaño de la población
         lambda_ = 700 #Cantidad de hijos que se generan por generación
         G = 100 #número de generaciones
         tau = 1/sqrt(n) #tasa de aprendizaje
         epsilon = 0.01 #valor mínimo de sigma (tamaño de paso)
```

```
In [29]: sol, fx = EvolutionStrategies(n, G, epsilon, tau, mu_, lambda_)
         print(f"x: {sol[:n]}\nSigma: {sol[n:]}\nf(x): {fx}")
```

```
x: [-0.00065657  0.00037216]
Sigma: [0.01]
f(x): 0.002149800797706991
```

Realizamos m ejecuciones de nuestro algoritmo usando 5, 10 y 20 variables de decisión. Para cada instancia del problema, obtenemos la corrida que encontró la mejor solución, la corrida que encontró la peor solución, la corrida que se encuentra en la mediana de las m ejecuciones, la media y la desviación estándar de las m corridas con respecto al valor de la función objetivo.

```
In [30]: def makeMRuns(m, n, G, epsilon, tau):
             sol =[]
             for m in range(m):
                 x, fx = EvolutionStrategies(n, G, epsilon, tau)
                 sol.append([x, fx])

             print(f"Para {n} variables")

             sol.sort(key=lambda individuo: individuo[-1])
             print("*************** Mejor solución ***************")
             print(f"x: {sol[0][0][:n]}\nSigma: {sol[0][0][n:]}\n
             f(x): {sol[0][-1]}")

             print("\n*************** Peor solución ****************")
             print(f"x: {sol[-1][0][:n]}\nSigma: {sol[-1][0][n:]}\n
             f(x): {sol[-1][-1]}")

             print("\n****************** Mediana *******************")
             print(f"x: {sol[m//2][0][:n]}\nSigma: {sol[m//2][0][n:]}\n
             f(x): {sol[m//2][-1]}")

             f_sol = [x[-1] for x in sol]
             print("\nMedia: ", np.mean(f_sol))
             print("\nDesviación estándar: ", np.std(f_sol))
```

```
In [31]: #Instancia con 5 variables
         n = 5
         G = 100
         epsilon = 0.01
         tau = 1/sqrt(n)
         m = 21

         makeMRuns(m, n, G, epsilon, tau)
```

```
Para 5 variables
*************** Mejor solución ***************
x: [ 0.00064673 -0.00386858  0.00048901  0.00019717 -0.00088428]
Sigma: [0.01]
f(x): 0.007429178001150394

*************** Peor solución ****************
x: [ 29.99756535 -24.9972199  -14.9996601  -13.9957628   26.00348506]
Sigma: [0.01047811]
f(x): 19.79532217626284

****************** Mediana *******************
x: [ 1.29892061e+01  2.99840510e+00 -9.99321299e+00  3.37022526e-04
 -8.98909928e+00]
Sigma: [0.01]
f(x): 16.324552661876236

Media:  9.776078373439958

Desviación estándar:  9.336075319808344
```

```
In [32]: #Instancia con 10 variables
         n = 10
         G = 100
         epsilon = 0.01
         tau = 1/sqrt(n)
         m = 21

         makeMRuns(m, n, G, epsilon, tau)
```

```
Para 10 variables
*************** Mejor solución ***************
x: [-0.00355678 -0.00469178  0.00141009  0.00297665  0.00214689 -0.0053407
 -0.00626911 -0.0019492   0.00068978 -0.00434212]
Sigma: [0.01262651]
f(x): 0.015775159267399363

*************** Peor solución ****************
x: [ 28.99420216  28.99560935   8.99479738 -25.99707964 -19.00635308
  21.00779196  27.99865091   5.00792837  -5.99960455  -1.00384787]
Sigma: [0.01]
f(x): 19.650596767560053

****************** Mediana *******************
```

```
x: [ 3.00131095e+00 -1.29979440e+01 -9.99735252e+00 -1.50097207e-03
  1.09964389e+01  3.00359373e+00  3.99191258e+00  8.99595040e+00
  1.99896535e+01  1.29886174e+01]
Sigma: [0.01]
f(x): 17.482527440612373

Media:  12.858952654109073

Desviación estándar:  8.197688889476673
```

```
In [33]: #Instancia con 20 variables
         n = 20
         G = 100
         epsilon = 0.015
         tau = 1/sqrt(n)
         m = 21

         makeMRuns(m, n, G, epsilon, tau)
```

```
Para 20 variables
*************** Mejor solución ***************
x: [-0.00096883 -0.01002416 -0.01067195 -0.00834144  0.00048528  0.01650114
  0.00474224  0.0038431   0.00660964 -0.00306943  0.01213031 -0.0023433
  0.00077898 -0.01847668 -0.00778009 -0.00873538  0.00346705  0.00676743
 -0.01373322  0.01753902]
Sigma: [0.015]
f(x): 0.043170523843248265

*************** Peor solución ****************
x: [ 21.00506099  12.97651264  12.99219435  15.00800424 -10.00504334
 -15.99002891  -0.99609665  21.01679554   4.00619211   3.99273967
  21.00221368 -22.99073379  15.00043309 -22.99548788  20.98875876
 -12.00360914  18.00666284 -13.98518125   8.01997577 -16.98441983]
Sigma: [0.01602363]
f(x): 19.16371926140832

****************** Mediana *******************
x: [-1.49840187e+01  5.99552206e+00 -6.01179188e+00 -2.49974553e+01
 -1.00433095e+00  7.72362131e-03  4.00673058e+00 -1.50045372e+01
 -8.98240037e+00  9.77063376e-01  5.00800423e+00 -1.09996444e+01
  9.01293109e+00 -7.99212644e+00 -6.99015853e+00 -4.98356274e+00
 -1.79853169e+01 -8.99187456e+00  2.29892261e+01  1.39955873e+01]
Sigma: [0.015]
f(x): 18.08099069250812
```

```
Media:  11.55775897319607

Desviación estándar:  8.883558523605728
```

Capítulo 5

Algoritmos genéticos (AG)

5.1. Función de Beale

La función de Beale es una función de dos variables y está definida como sigue:

$$\text{mín } f(x_1, x_2) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + (2.625 - x_1 + x_1 x_2^3)^2 \quad (5.1)$$

donde $-4.5 \leq x_1, x_2 \leq 4.5$. El mínimo global está en $x^* = (3, 0.5)$ y $f(x^*) = 0$.

5.1.1. Principales componentes de AG

Librerías de **Python** que vamos a utilizar.

```
In [1]: from random import randint, random, shuffle
        from copy import copy, deepcopy
        from math import ceil, log, e, exp, sqrt, pi, cos
        import numpy as np
```

Datos de entrada.

```
In [2]: n = 2 #Número de variables de decisión
        precision = 3 #número de dígitos de precisión
        bounds = [-4.5, 4.5] #límites inferior y superior
```

Representación de la información genética

La fórmula vista para calcular el número de bits que se necesitan por variable es:

$$num\_bits = \lceil \log_2((l_{sup} - l_{inf}) * 10^p) \rceil$$

donde p es la precisión deseada y l_{inf} y l_{sup} son los límites inferior y superior de la variable.

```
In [3]: def getSizeVariable(bounds, p):
            return ceil(log((bounds[1] - bounds[0])*(10**p), 2))

In [4]: num_bits = getSizeVariable(bounds, precision)
        print(num_bits)

14
```

Dado que nuestro problema tiene dos variables y ambas tienen el mismo límite superior e inferior, el tamaño de la cadena binaria que representa el cromosoma de un individuo es de longitud 28. Por ejemplo:

```
In [5]: #Si n = 2
        i = '01110100001000110101011001'
        print(i, len(i))

01110100001000110101011001 28
```

Para poder calcular la aptitud de cada individuo es necesario obtener el valor real de cada una de sus variables.

```
In [6]: def getRealValues(chromosome, num_bits, num_variables, bounds):
            point = []
            for i in range(num_variables):
                b = chromosome[i*num_bits:(i+1)*num_bits]

                #binario a entero
                d = 0
                e = num_bits-1
                for j in b:
                    d += (int(j) * 2**e)
                    e -= 1

                #entero a decimal
                r = bounds[0] + (((bounds[1]-bounds[0])*d)/(2**num_bits-1))
                point.append(r)

            return np.array(point)

In [7]: i_real = getRealValues(i, 14, 2, bounds)
        i_real

Out[7]: array([-0.41723128,  3.00247207])
```

Aptitud

Función de Beale

```
In [8]: def f(x):
            x1, x2 = x[0], x[1]
            term1 = (1.5 - x1 + x1*x2)**2
            term2 = (2.25 - x1 + x1*(x2**2))**2
            term3 = (2.625 - x1 + x1*(x2**3))**2

            return term1 + term2 + term3
```

Dado que estamos resolviendo un problema de minimización, asignamos la aptitud de cada individuo i de la siguiente forma:

$$f_a(i) = \frac{1}{f(x_1^{(i)}, x_2^{(i)}) + 1}$$

```
In [9]: def getFitness(f_x):
            return 1/(f_x+1)
```

```
In [10]: fx = f(i_real)
         print (f"x = {i_real}\nf(x) = {fx}\nAptitud(x) = {getFitness(fx)}")
```

```
x = [-0.41723128  3.00247207]
f(x) = 69.71559241216701
Aptitud(x) = 0.014141152833331061
```

Representación de un individuo

Para almacenar la información de los individuos, vamos a crear una clase con los siguientes atributos:

1. **x**: Cadena binaria con el cromosoma del individuo.
2. **x_real**: Valores reales de cada variable.
3. **f**: Valor en la función objetivo.
4. **fitness**: Aptitud del individuo.

Y tendrá la función **__str__**.

```
In [11]: class Individual:
             def __init__(self, chromosome):
                 self.x = chromosome
                 self.x_real = None
                 self.f = None
                 self.fitness = None
```

```
            def __str__(self):
                #Imprime información del individuo.
                #Este método se manda a llamar a través print
                return "Cadena binaria: {}\nComponentes reales: {}\n
                f(x): {}\nAptitud: {}".format(self.x, self.x_real, self.f,
                self.fitness)
```

Población inicial

Para cada posición de la cadena binaria de un individuo, se utiliza una distribución uniforme para generar un número aleatorio en el intervalo $(0, 1)$. Si el valor resultante es menor a 0.5, toma el valor de cero. De lo contrario, toma el valor de 1.

```
In [12]: def getInitialPopulation(mu, num_bits, n):
             population = []
             for _ in range(mu):
                 chromosome = ""
                 for i in range(n*num_bits):
                     if random()<.5:
                         chromosome+="0"
                     else:
                         chromosome+="1"

                 population.append(Individual(chromosome))
             return population
```

```
In [13]: mu = 100
         population = getInitialPopulation(mu, num_bits, n)
```

Evaluamos cada uno de los individuos:

```
In [14]: for p in population:
             p.x_real = getRealValues(p.x, num_bits, n, bounds)
             p.f = f(p.x_real)
             p.fitness = getFitness(p.f)
```

```
In [15]: print(population[0])
```

```
Cadena binaria: 00110010101000011010100000101
Componentes reales: [-2.72010621 -0.77046329]
f(x): 94.56586667106828
Aptitud: 0.010463987141370645
```

Selección de padres

Universal estocástica

```
In [16]: def getPopulationMean(population, mu):
             mean = 0

             for p in range(mu):
                 mean += population[p].fitness

             return mean/mu
```

```
In [17]: #Regresa un arreglo que contiene los índices de los individuos
         #que serán padres
         def universalStochastic(population, mu):
             r = random()
             mean = getPopulationMean(population, mu)
             cont = 0
             parents = []
             for i in range(mu):
                 cont += population[i].fitness/mean
                 while cont > r and len(parents) < mu:
                     parents.append(i)
                     r += 1

             return parents
```

```
In [18]: parents = universalStochastic(population, mu)
         print(population[parents[0]])
         print(population[parents[10]])
```

```
Cadena binaria: 00110010101000011010100000101
Componentes reales: [-2.72010621 -0.77046329]
f(x): 94.56586667106828
Aptitud: 0.010463987141370645
Cadena binaria: 11100001010001100011111010010
Componentes reales: [3.41997803 0.53753891]
f(x): 0.1092864247827332
Aptitud: 0.9014804271095824
```

Recombinación

Cruza de dos puntos:

1. Generamos dos puntos de cruza aleatorios con distribución uniforme.

2. A partir de la posición 0 y hasta el primer punto de cruza, el primer hijo toma la información del primer padre y el segundo hijo toma la información del segundo padre.

3. A partir del primer punto de cruza y hasta el segundo punto de cruza, el primer hijo toma la información del segundo padre y el segundo hijo toma la información del primer padre.

4. A partir del segundo punto de cruza y hasta el final de la cadena, el primer hijo toma la información del primer padre y el segundo hijo toma la información del segundo padre.

```
In [19]: def twoPointsCrossover(p1, p2):
             index1 = randint(1, len(p2.x)-1)
             index2 = index1
             while index2 == index1:
                 index2 = randint(1,len(p2.x)-1)

             if index2 < index1:
                 index2, index1 = index1, index2

             offspring_1 = p1.x[:index1]+p2.x[index1:index2]+p1.x[index2:]
             offspring_2 = p2.x[:index1]+p1.x[index1:index2]+p2.x[index2:]

             return Individual(offspring_1), Individual(offspring_2)
```

```
In [20]: o1, o2 = twoPointsCrossover(population[parents[0]], population[parents[10]])
         o1.x_real = getRealValues(o1.x, num_bits, n, bounds)
         o2.x_real = getRealValues(o2.x, num_bits, n, bounds)
         o1.f = f(o1.x_real)
         o2.f = f(o2.x_real)
         o1.fitness = getFitness(o1.f)
         o2.fitness = getFitness(o2.f)
         print(o1)
         print(o2)
```

```
Cadena binaria: 00110010101000010010100000101
Componentes reales: [-2.72010621 -1.89553195]
f(x): 680.8091084998717
Aptitud: 0.0014666861846422343
Cadena binaria: 11100001010001101011110100 10
Componentes reales: [3.41997803 1.66260758]
f(x): 305.4954578474802
Aptitud: 0.0032626910918126064
```

Mutación

Por cada elemento en la cadena binaria, generamos un número aleatorio en el intervalo $(0,1)$, utilizando una distribución uniforme. Si el número generado es menor que la probabilidad de mutación, se mutará el elemento: si es 1 se hace 0 y si es 0 se hace 1.

```
In [21]: def mutation(individual, pm):
             for i in range(len(individual.x)):
                 if random() < pm:
                     if individual.x[i] == "1":
                         individual.x = individual.x[:i] + "0" + individual.x[i+1:]
                     else:
                         individual.x = individual.x[:i] + "1" + individual.x[i+1:]
```

```
In [22]: print(f"Antes de mutar\n{o1.x}")
         mutation(o1, 0.1)
         print(f"\nDespués de mutar\n{o1.x}")
```

```
Antes de mutar
00110010101000010010100000101

Después de mutar
00110010101000111010100000101
```

Selección de sobrevivientes

Selección basada en edad aplicando elitismo: Si el peor de los hijos es peor que la mejor solución de la población actual, se sustituye el peor de los hijos por la mejor solución actual. La siguiente generación estará conformada por los hijos generados.

```
In [23]: def getBestIndividual(population):
             best = 0
             for i in range(1, len(population)):
                 if population[i].fitness > population[best].fitness:
                     best = i
             return best

         def getWorstIndividual(population):
             worst = 0
             for i in range(1, len(population)):
                 if population[i].fitness < population[worst].fitness:
                     worst = i
             return worst
```

```
In [24]: def getSurvivors(parents, offspring):
             best_ind = getBestIndividual(parents)
             worst_ind = getWorstIndividual(offspring)

             if parents[best_ind].fitness > offspring[worst_ind].fitness:
                 offspring.pop(worst_ind)
                 offspring.append(copy(parents[best_ind]))

             return offspring
```

5.1.2. Algoritmo Genético

A continuación se muestra el algoritmo genético completo.

```
In [40]: def GeneticAlgorithm(G, mu, pc, pm, n, precision, bounds):
             num_bits = getSizeVariable(bounds, precision)

             population = getInitialPopulation(mu, num_bits, n)
             #Fitness
             for p in population:
                 p.x_real = getRealValues(p.x, num_bits, n, bounds)
                 p.f = f(p.x_real)
                 p.fitness = getFitness(p.f)

             for _ in range(G):
                 parents = universalStochastic(population, mu)
                 shuffle(parents)
                 offspring = []
                 for i in range(0,mu,2):
                     if random() < pc:
                         o1, o2 = twoPointsCrossover(population[parents[i]],
                         population[parents[i+1]])
                     else:
                         o1, o2 = deepcopy(population[parents[i]]),
                         deepcopy(population[parents[i+1]])

                     mutation(o1, pm)
                     o1.x_real = getRealValues(o1.x, num_bits, n, bounds)
                     o1.f = f(o1.x_real)
                     o1.fitness = getFitness(o1.f)

                     mutation(o2, pm)
                     o2.x_real = getRealValues(o2.x, num_bits, n, bounds)
                     o2.f = f(o2.x_real)
                     o2.fitness = getFitness(o2.f)
```

```
                    offspring.extend([o1, o2])

                population = getSurvivors(population, offspring)


            best = getBestIndividual(population)
            return population[best]
```

```
In [41]: G = 100 #Número de generaciones
         mu = 100 #tamaño de la población
         pc = 0.9 #probabilidad de cruza
         pm = 0.0001 #probabilidad de mutación
         n = 2 #número de variables de decision
         precision = 3 #número de dígitos de precisión
         bounds = [-4.5, 4.5] #límites inferior y superior
```

```
In [42]: print(GeneticAlgorithm(G, mu, pc, pm, n, precision, bounds))
```

```
Cadena binaria: 11011110000010100100000000110
Componentes reales: [3.30626259 0.56610511]
f(x): 0.010923175615842709
Aptitud: 0.9891948509250582
```

Realizamos m ejecuciones de nuestro algoritmo y obtenemos la corrida que encontró la mejor solución, la corrida que encontró la peor solución, la corrida que se encuentra en la mediana de las m ejecuciones, la media y la desviación estándar de las m corridas con respecto al valor de la función objetivo.

```
In [31]: def makeMRuns(m, G, mu, pc, pm, n, precision, bounds):
             sol =[]
             for m in range(m):
                 sol.append(GeneticAlgorithm(G, n, pc, pm, n, precision, bounds))


             sol.sort(key=lambda individuo: individuo.f)
             print("*************** Mejor solución ***************")
             print(sol[0])

             print("\n*************** Peor solución ****************")
             print(sol[-1])

             print("\n****************** Mediana *******************")
             print(sol[m//2])
```

```
            f_sol = [x.f for x in sol]
            print("\nMedia: ", np.mean(f_sol))
            print("Desviación estándar: ", np.std(f_sol))
```

```
In [34]: m = 21
         makeMRuns(m, G, mu, pc, pm, n, precision, bounds)
```

```
*************** Mejor solución ***************
Cadena binaria: 11011101100100100100000100001
Componentes reales: [3.28978209 0.58093756]
f(x): 0.020091304576727458
Aptitud: 0.9803044056090018

*************** Peor solución ****************
Cadena binaria: 11000010101010000000110001011
Componentes reales: [ 2.3438015  -4.28300678]
f(x): 35765.95829927722
Aptitud: 2.7958765507331612e-05

****************** Mediana *******************
Cadena binaria: 01010111111100100001011111110
Componentes reales: [-1.40825856  0.42107673]
f(x): 32.408936103276446
Aptitud: 0.029932111483847255

Media:  3412.875542684207
Desviación estándar:  8560.413403038703
```

Es importante mencionar que este es un diseño simple de un AG. Si se revisa el número de copias que se están realizando de los mejores individuos, se observará que es un valor muy alto y esto puede provocar convergencia a soluciones no óptimas.