AG3 - Actividad Guiada 3

Nombre: Guillermo Rios Gómez
Github: https://github.com/GuiRiGo88/03MIAR---Algoritmos-de-Optimizacion---2023

Carga de librerias

```
!pip install requests    #Hacer llamadas http a paginas de la red
!pip install tsplib95    #Modulo para las instancias del problema del TSP
```

```
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: tsplib95 in /usr/local/lib/python3.10/dist-packages (0
Requirement already satisfied: Click>=6.0 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: Deprecated~=1.2.9 in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: networkx~=2.1 in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: tabulate~=0.8.7 in /usr/local/lib/python3.10/dist-pacl
Requirement already satisfied: wrapt<2,>=1.10 in /usr/local/lib/python3.10/dist-packa
```

Carga de los datos del problema

```
import urllib.request #Hacer llamadas http a paginas de la red
import tsplib95       #Modulo para las instancias del problema del TSP
import math           #Modulo de funciones matematicas. Se usa para exp
import random         #Para generar valores aleatorios
```

```
#http://elib.zib.de/pub/mp-testdata/tsp/tsplib/
#Documentacion :
  # http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf
  # https://tsplib95.readthedocs.io/en/stable/pages/usage.html
  # https://tsplib95.readthedocs.io/en/v0.6.1/modules.html
  # https://pypi.org/project/tsplib95/

#Descargamos el fichero de datos(Matriz de distancias)
file = "swiss42.tsp" ;
urllib.request.urlretrieve("http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/swis
!gzip -d swiss42.tsp.gz     #Descomprimir el fichero de datos
```

```
#Coordendas 51-city problem (Christofides/Eilon)
#file = "eil51.tsp" ; urllib.request.urlretrieve("http://comopt.ifi.uni-heidelberg.de/soft

#Coordenadas - 48 capitals of the US (Padberg/Rinaldi)
#file = "att48.tsp" ; urllib.request.urlretrieve("http://comopt.ifi.uni-heidelberg.de/soft
```

```
#Carga de datos y generación de objeto problem
################################################################################
problem = tsplib95.load(file)

#Nodos
Nodos = list(problem.get_nodes())

#Aristas
Aristas = list(problem.get_edges())
```

```
Aristas
```

```
[(0, 0),
 (0, 1),
 (0, 2),
 (0, 3),
 (0, 4),
 (0, 5),
 (0, 6),
 (0, 7),
 (0, 8),
 (0, 9),
 (0, 10),
 (0, 11),
 (0, 12),
 (0, 13),
 (0, 14),
 (0, 15),
 (0, 16),
 (0, 17),
 (0, 18),
 (0, 19),
 (0, 20),
 (0, 21),
 (0, 22),
 (0, 23),
 (0, 24),
 (0, 25),
 (0, 26),
 (0, 27),
 (0, 28),
 (0, 29)
```

```
(0, 29),
(0, 30),
(0, 31),
(0, 32),
(0, 33),
(0, 34),
(0, 35),
(0, 36),
(0, 37),
(0, 38),
(0, 39),
(0, 40),
(0, 41),
(1, 0),
(1, 1),
(1, 2),
(1, 3),
(1, 4),
(1, 5),
(1, 6),
(1, 7),
(1, 8),
(1, 9),
(1, 10),
(1, 11),
(1, 12),
(1, 13),
(1, 14),
(1, 15),
```

```
NOMBRE: swiss42
TIPO: TSP
COMENTARIO: 42 Staedte Schweiz (Fricker)
DIMENSION: 42
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
0 15 30 23 32 55 33 37 92 114 92 110 96 90 74 76 82 72 78 82 159 122 131 206 112 57 28 43 70
  15 0 34 23 27 40 19 32 93 117 88 100 87 75 63 67 71 69 62 63 96 164 132 131 212 106 44 33 5
  30 34 0 11 18 57 36 65 62 84 64 89 76 93 95 100 104 98 57 88 99 130 100 101 179 86 51 4 18
  23 23 11 0 11 48 26 54 70 94 69 75 75 84 84 89 92 89 54 78 99 141 111 109 89 89 11 11 11 54
  32 27 18 11 0 40 20 58 67 92 61 78 65 76 83 89 91 95 43 72 110 141 116 105 190 81 34 19 35
  55 40 57 48 40 0 23 55 96 123 78 75 36 36 66 66 63 95 34 34 137 174 156 129 224 90 15 59 75
  33 19 36 26 20 23 0 45 85 111 75 82 69 60 63 70 71 85 44 52 115 161 136 122 210 91 25 37 54
  37 32 65 54 58 55 45 0 124 149 118 126 113 80 42 42 40 40 87 87 94 158 158 163 242 135 65 6
  92 93 62 70 67 96 85 124 0 28 29 68 63 122 148 155 156 159 67 129 148 78 80 39 129 46 82 65
 114 117 84 94 92 123 111 149 28 0 54 91 88 150 174 181 182 181 95 157 159 50 65 27 102 65 11
  92 88 64 69 61 78 75 118 29 54 0 39 34 99 134 142 141 157 44 110 161 103 109 52 154 22 63 6
 110 100 89 89 78 75 82 126 68 91 39 0 14 80 129 139 135 167 39 98 187 136 148 81 186 28 61 9
  96 87 76 75 65 62 69 113 63 88 34 14 0 72 117 128 124 153 26 88 174 136 142 82 187 32 48 79
  90 75 93 84 76 36 60 80 122 150 99 80 72 0 59 71 63 116 56 25 170 201 189 151 252 104 44 95
  74 63 95 84 83 56 63 42 148 174 134 129 117 59 0 11 8 63 93 35 135 223 195 184 273 146 71 9
```

```
#Probamos algunas funciones del objeto problem

#Distancia entre nodos
problem.get_weight(0, 1)
```

```
problem.get_weight(0, 1)

#Todas las funciones
#Documentación: https://tsplib95.readthedocs.io/en/v0.6.1/modules.html

#dir(problem)
```

```
15
```

Funcionas basicas

```
#Funcionas basicas
################################################################################

#Se genera una solucion aleatoria con comienzo en en el nodo 0
def crear_solucion(Nodos):
  solucion = [Nodos[0]]
  for n in Nodos[1:]:
    solucion = solucion + [random.choice(list(set(Nodos) - set({Nodos[0]}) - set(solucion
  return solucion

#Devuelve la distancia entre dos nodos
def distancia(a,b, problem):
  return problem.get_weight(a,b)

#Devuelve la distancia total de una trayectoria/solucion
def distancia_total(solucion, problem):
  distancia_total = 0
  for i in range(len(solucion)-1):
    distancia_total += distancia(solucion[i] ,solucion[i+1] ,  problem)
  return distancia_total + distancia(solucion[len(solucion)-1] ,solucion[0], problem)

sol_temporal = crear_solucion(Nodos)

distancia_total(sol_temporal, problem), sol_temporal
```

```
(4789,
 [0,
  38,
  4,
  18,
  17,
  6,
  5,
  2,
  14,
  41,
  12,
  20,
  16
```

```
       29,
       32,
       30,
       26,
       35,
       39,
       36,
       40,
       9,
       27,
       3,
       15,
       19,
       13,
       37,
       21,
       24,
       34,
       8,
       10,
       11,
       31,
       28,
       33,
       23,
       1,
       7,
       25,
       22])
```

BUSQUEDA ALEATORIA

```
################################################################################
# BUSQUEDA ALEATORIA
################################################################################

def busqueda_aleatoria(problem, N):
  #N es el numero de iteraciones
  Nodos = list(problem.get_nodes())

  mejor_solucion = []
  #mejor_distancia = 10e100                          #Inicializamos con un valor alto
  mejor_distancia = float('inf')                     #Inicializamos con un valor alto

  for i in range(N):                                 #Criterio de parada: repetir N veces
    solucion = crear_solucion(Nodos)                 #Genera una solucion aleatoria
    distancia = distancia_total(solucion, problem)   #Calcula el valor objetivo(distancia

    if distancia < mejor_distancia:                  #Compara con la mejor obtenida hasta
      mejor_solucion = solucion
```

```
      mejor_solucion = solucion
      mejor_distancia = distancia


  print("Mejor solución:" , mejor_solucion)
  print("Distancia     :" , mejor_distancia)
  return mejor_solucion


#Busqueda aleatoria con 5000 iteraciones
solucion = busqueda_aleatoria(problem, 10000)
```

```
     Mejor solución: [0, 7, 15, 37, 36, 5, 23, 39, 18, 21, 40, 41, 1, 3, 4, 14, 16, 27, 1
     Distancia     : 3637
```

BUSQUEDA LOCAL

```
# ################################################################################
# # BUSQUEDA LOCAL
# ################################################################################
# def genera_vecina(solucion):
#   #Generador de soluciones vecinas: 2-opt (intercambiar 2 nodos) Si hay N nodos se gene
#   #Se puede modificar para aplicar otros generadores distintos que 2-opt
#   #print(solucion)
#   mejor_solucion = []
#   mejor_distancia = 10e100
#   for i in range(1,len(solucion)-1):          #Recorremos todos los nodos en bucle dob
#     for j in range(i+1, len(solucion)):

#       #Se genera una nueva solución intercambiando los dos nodos i,j:
#       #  (usamos el operador + que para listas en python las concatena) : ej.: [1,2] +
#       vecina = solucion[:i] + [solucion[j]] + solucion[i+1:j] + [solucion[i]] + solucio

#       #Se evalua la nueva solución ...
#       distancia_vecina = distancia_total(vecina, problem)

#       #... para guardarla si mejora las anteriores
#       if distancia_vecina <= mejor_distancia:
#         mejor_distancia = distancia_vecina
#         mejor_solucion = vecina
#   return mejor_solucion


# #solucion = [1, 47, 13, 41, 40, 19, 42, 44, 37, 5, 22, 28, 3, 2, 29, 21, 50, 34, 30, 9
# print("Distancia Solucion Incial:" , distancia_total(solucion, problem))


# nueva_solucion = genera_vecina(solucion)
# print("Distancia Mejor Solucion Local:", distancia_total(nueva_solucion, problem))
```

```
# #Busqueda Local:
# #  - Sobre el operador de vecindad 2-opt(funcion genera_vecina)
# #  - Sin criterio de parada, se para cuando no es posible mejorar.
# def busqueda_local(problem):
#   mejor_solucion = []

#   #Generar una solucion inicial de referencia(aleatoria)
#   solucion_referencia = crear_solucion(Nodos)
#   mejor_distancia = distancia_total(solucion_referencia, problem)

#   iteracion=0             #Un contador para saber las iteraciones que hacemos
#   while(1):
#     iteracion +=1         #Incrementamos el contador
#     #print('#',iteracion)

#     #Obtenemos la mejor vecina ...
#     vecina = genera_vecina(solucion_referencia)

#     #... y la evaluamos para ver si mejoramos respecto a lo encontrado hasta el momento
#     distancia_vecina = distancia_total(vecina, problem)

#     #Si no mejoramos hay que terminar. Hemos llegado a un minimo local(según nuestro op
#     if distancia_vecina < mejor_distancia:
#       #mejor_solucion = copy.deepcopy(vecina)   #Con copia profunda. Las copias en pyth
#       mejor_solucion = vecina                   #Guarda la mejor solución encontrada
#       mejor_distancia = distancia_vecina

#     else:
#       print("En la iteracion ", iteracion, ", la mejor solución encontrada es:" , mejor
#       print("Distancia     :" , mejor_distancia)
#       return mejor_solucion

#     solucion_referencia = vecina


# sol = busqueda_local(problem )

####PROPUESTA DE MEJORA 1############
import random

# Función para intercambiar dos nodos en la solución
def swap(solucion, i, j):
    nueva_solucion = solucion.copy()
    nueva_solucion[i], nueva_solucion[j] = nueva_solucion[j], nueva_solucion[i]
    return nueva_solucion

# Función para invertir el orden de una subsecuencia de nodos
def reverse(solucion, i, j):
```

```
    nueva_solucion = solucion[:i] + solucion[i:j+1][::-1] + solucion[j+1:]
    return nueva_solucion

def genera_vecina(solucion, operador):
    mejor_solucion = solucion.copy()
    mejor_distancia = distancia_total(solucion, problem)

    n = len(solucion)

    for i in range(1, n-2):
        for j in range(i+1, n-1):
            if operador == 'swap':
                vecina = swap(solucion, i, j)
            elif operador == 'reverse':
                vecina = reverse(solucion, i, j)
            # Agregar otros operadores según sea necesario

            distancia_vecina = distancia_total(vecina, problem)

            if distancia_vecina < mejor_distancia:
                mejor_distancia = distancia_vecina
                mejor_solucion = vecina

    return mejor_solucion

# Solución inicial
# solucion = [1, 47, 13, 41, 40, 19, 42, 44, 37, 5, 22, 28, 3, 2, 29, 21, 50, 34, 30, 9,

# Mostrar la distancia inicial
print("Distancia Solución Inicial:", distancia_total(solucion, problem))

# Generar vecina usando el operador de swap
nueva_solucion_swap = genera_vecina(solucion, operador='swap')
print("Distancia Mejor Solución (Swap):", distancia_total(nueva_solucion_swap, problem))

# Generar vecina usando el operador de reverse
nueva_solucion_reverse = genera_vecina(solucion, operador='reverse')
print("Distancia Mejor Solución (Reverse):", distancia_total(nueva_solucion_reverse, prol
```

```
Distancia Solución Inicial: 3637
Distancia Mejor Solución (Swap): 3323
Distancia Mejor Solución (Reverse): 3432
```

SIMULATED ANNEALING

```
################################################################################
# SIMULATED ANNEALING
```

```
################################################################################

#Generador de 1 solucion vecina 2-opt 100% aleatoria (intercambiar 2 nodos)
#Mejorable eligiendo otra forma de elegir una vecina.
def genera_vecina_aleatorio(solucion):

  #Se eligen dos nodos aleatoriamente
  i,j = sorted(random.sample( range(1,len(solucion)) , 2))

  #Devuelve una nueva solución pero intercambiando los dos nodos elegidos al azar
  return solucion[:i] + [solucion[j]] + solucion[i+1:j] + [solucion[i]] + solucion[j+1:]


#Funcion de probabilidad para aceptar peores soluciones
def probabilidad(T,d):
  if random.random() <  math.exp( -1*d / T)  :
    return True
  else:
    return False

#Funcion de descenso de temperatura
def bajar_temperatura(T):
  return T*0.99

def recocido_simulado(problem, TEMPERATURA ):
  #problem = datos del problema
  #T = Temperatura

  solucion_referencia = crear_solucion(Nodos)
  distancia_referencia = distancia_total(solucion_referencia, problem)

  mejor_solucion = []             #x* del seudocodigo
  mejor_distancia = 10e100        #F* del seudocodigo


  N=0
  while TEMPERATURA > .0001:
    N+=1
    #Genera una solución vecina
    vecina =genera_vecina_aleatorio(solucion_referencia)

    #Calcula su valor(distancia)
    distancia_vecina = distancia_total(vecina, problem)

    #Si es la mejor solución de todas se guarda(siempre!!!)
    if distancia_vecina < mejor_distancia:
        mejor_solucion = vecina
        mejor_distancia = distancia_vecina

    #Si la nueva vecina es mejor se cambia
    #Si es peor se cambia según una probabilidad que depende de T y delta(distancia refe
```

```
    #Si es peor se cambia según una probabilidad que depende de T y delta(distancia_refer
    if distancia_vecina < distancia_referencia or probabilidad(TEMPERATURA, abs(distancia
      #solucion_referencia = copy.deepcopy(vecina)
      solucion_referencia = vecina
      distancia_referencia = distancia_vecina

    #Bajamos la temperatura
    TEMPERATURA = bajar_temperatura(TEMPERATURA)

  print("La mejor solución encontrada es " , end="")
  print(mejor_solucion)
  print("con una distancia total de " , end="")
  print(mejor_distancia)
  return mejor_solucion

sol  = recocido_simulado(problem, 10000000)
```

```
La mejor solución encontrada es [0, 7, 17, 31, 37, 15, 1, 28, 32, 34, 33, 20, 3, 2, 1
con una distancia total de 2131
```

```
####PROPUESTA DE MEJORA 2############
import random
import math

# Función para generar una solución vecina mejorada
def genera_vecina_mejorada(solucion):
    # Implementa una estrategia más sofisticada para generar soluciones vecinas

    # Se elige un índice aleatorio dentro del rango de la longitud de la solución
    i = random.randint(0, len(solucion) - 2)

    # Se intercambian dos nodos adyacentes
    vecina_mejorada = solucion[:i] + [solucion[i + 1], solucion[i]] + solucion[i + 2:]

    return vecina_mejorada

# Función de probabilidad para aceptar peores soluciones
def probabilidad(T, d):
    if random.random() < math.exp(-1 * d / T):
        return True
    else:
        return False

# Función de descenso de temperatura
def bajar_temperatura(T):
    return T * 0.99

def recocido_simulado_mejorado(problem, TEMPERATURA):
    # Inicialización de la solución de referencia y distancia de referencia
    solucion_referencia = crear_solucion(Nodos)  # Necesitas definir la función crear_solu
    distancia_referencia = distancia_total(solucion_referencia, problem)
```

```
    # Inicialización de la mejor solución encontrada
    mejor_solucion = solucion_referencia.copy()
    mejor_distancia = distancia_referencia

    N = 0
    while TEMPERATURA > 0.0001:
        N += 1

        # Genera una solución vecina mejorada
        vecina_mejorada = genera_vecina_mejorada(solucion_referencia)

        # Calcula la distancia de la solución vecina mejorada
        distancia_vecina_mejorada = distancia_total(vecina_mejorada, problem)

        # Actualiza la mejor solución si es necesario
        if distancia_vecina_mejorada < mejor_distancia:
            mejor_solucion = vecina_mejorada.copy()
            mejor_distancia = distancia_vecina_mejorada

        # Actualiza la solución de referencia si la vecina es mejor o se acepta probabilís
        if (
            distancia_vecina_mejorada < distancia_referencia
            or probabilidad(TEMPERATURA, abs(distancia_referencia - distancia_vecina_mejor
        ):
            solucion_referencia = vecina_mejorada.copy()
            distancia_referencia = distancia_vecina_mejorada

        # Reduce la temperatura
        TEMPERATURA = bajar_temperatura(TEMPERATURA)

    print("La mejor solución encontrada es ", end="")
    print(mejor_solucion)
    print("con una distancia total de ", end="")
    print(mejor_distancia)

    return mejor_solucion

# Ejemplo de uso
sol = recocido_simulado_mejorado(problem, 10000000)
```

```
La mejor solución encontrada es [4, 14, 13, 12, 0, 39, 24, 38, 34, 30, 26, 18, 41, 4(
con una distancia total de 3143
```