0.1- Carga de librerias

Nombre: Adrián García López **Github:**
https://github.com/adri14gl/Algoritmos_de_Optimizacion_VIU

```
#Hacer llamadas http a paginas de la red
!pip install requests

#Modulo para las instancias del problema del TSP
!pip install tsplib95
```

```
Requirement already satisfied: requests in c:\users\adri1\anaconda3\
lib\site-packages (2.32.2)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\
adri1\anaconda3\lib\site-packages (from requests) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in c:\users\adri1\
anaconda3\lib\site-packages (from requests) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\adri1\
anaconda3\lib\site-packages (from requests) (2.2.2)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\adri1\
anaconda3\lib\site-packages (from requests) (2026.1.4)
Requirement already satisfied: tsplib95 in c:\users\adri1\anaconda3\
lib\site-packages (0.7.1)
Requirement already satisfied: Click>=6.0 in c:\users\adri1\anaconda3\
lib\site-packages (from tsplib95) (8.1.7)
Requirement already satisfied: Deprecated~=1.2.9 in c:\users\adri1\
anaconda3\lib\site-packages (from tsplib95) (1.2.18)
Requirement already satisfied: networkx~=2.1 in c:\users\adri1\
anaconda3\lib\site-packages (from tsplib95) (2.8.8)
Requirement already satisfied: tabulate~=0.8.7 in c:\users\adri1\
anaconda3\lib\site-packages (from tsplib95) (0.8.10)
Requirement already satisfied: colorama in c:\users\adri1\anaconda3\
lib\site-packages (from Click>=6.0->tsplib95) (0.4.6)
Requirement already satisfied: wrapt<2,>=1.10 in c:\users\adri1\
anaconda3\lib\site-packages (from Deprecated~=1.2.9->tsplib95)
(1.14.1)
```

0.2.- 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("https://raw.githubusercontent.com/mastqe/
tsplib/refs/heads/master/swiss42.tsp", file)
#!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/softwa
re/TSPLIB95/tsp/eil51.tsp.gz", file)

#Coordenadas - 48 capitals of the US (Padberg/Rinaldi)
#file = "att48.tsp" ;
urllib.request.urlretrieve("http://comopt.ifi.uni-heidelberg.de/softwa
re/TSPLIB95/tsp/att48.tsp.gz", file)
```

```
('swiss42.tsp', <http.client.HTTPMessage at 0x1ef07711eb0>)
```

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

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

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

```
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)

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

#dir(problem)
```

```
15
```

0.3.- Funcionas basicas

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

#Se genera una solucion aleatoria con comienzo en en el nodo 0
def crear_solucion(Nodos):
  solucion = [Nodos[0]] #Inicializamos una lista y vamos añadiendo
ciudades sin repetir (sin bucles)
  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
```

```
(4582,
 [0,
  40,
  13,
  11,
  5,
  2,
  41,
  24,
  28,
  18,
  21,
  38,
  37,
  3,
  1,
  30,
  7,
  10,
  9,
  27,
  8,
  23,
  20,
  12,
  14,
  33,
  19,
  6,
  15,
  16,
  35,
  39,
  25,
```

```
  22,
  29,
  31,
  36,
  4,
  32,
  17,
  26,
  34])
```

1.- BUSQUEDA ALEATORIA

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

## En este caso simplemente calculamos N soluciones aleatorias y nos
quedamos con la mejor

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 pero podemos incluir otros
    solucion = crear_solucion(Nodos)                 #Genera una
solucion aleatoria
    distancia = distancia_total(solucion, problem)  #Calcula el valor
objetivo(distancia total)

    if distancia < mejor_distancia:                  #Compara con la
mejor obtenida hasta ahora
      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, 30, 18, 21, 26, 19, 37, 7, 23, 32, 31, 16, 4, 25,
34, 28, 41, 8, 9, 17, 36, 35, 14, 5, 29, 24, 38, 33, 6, 11, 13, 15,
20, 12, 10, 39, 22, 40, 27, 3, 1, 2]
Distancia     : 3765
```

2.- BUSQUEDA LOCAL

```
#############################################################################
#########
# BUSQUEDA LOCAL
#############################################################################
#########
def genera_vecina(solucion):
  #Generador de soluciones vecinas: 2-opt (intercambiar 2 nodos) Si
hay N nodos se generan (N-1)x(N-2)/2 soluciones
  #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 doble para evaluar todos los intercambios 2-opt
    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] + [3] = [1,2,3]
      vecina = solucion[:i] + [solucion[j]] + solucion[i+1:j] +
[solucion[i]] + solucion[j+1:]

      #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, 16, 11, 38, 49, 10, 39, 33, 45, 15, 24, 43, 26, 31,
36, 35, 20, 8, 7, 23, 48, 27, 12, 17, 4, 18, 25, 14, 6, 51, 46, 32]
print("Distancia Solucion Incial:" , distancia_total(solucion,
```

```
problem))


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

```
Distancia Solucion Incial: 3765
Distancia Mejor Solucion Local: 3510
```

```
#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 operador de vencindad 2-opt)
    if distancia_vecina < mejor_distancia:
      #mejor_solucion = copy.deepcopy(vecina)   #Con copia profunda.
Las copias en python son por referencia
      mejor_solucion = vecina                  #Guarda la mejor
solución encontrada
      mejor_distancia = distancia_vecina

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

    solucion_referencia = vecina
```

```
sol = busqueda_local(problem )
```

```
En la iteracion  40 , la mejor solución encontrada es: [0, 3, 27, 2,
29, 8, 41, 23, 40, 24, 21, 18, 14, 16, 15, 37, 6, 1, 7, 17, 36, 35,
31, 20, 33, 34, 32, 4, 26, 5, 19, 13, 12, 11, 25, 10, 9, 39, 22, 38,
30, 28]
Distancia: 1720
```

3.- 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_referencia - distancia_vecina)
    if distancia_vecina < distancia_referencia or
probabilidad(TEMPERATURA, abs(distancia_referencia - distancia_vecina)
) :
      #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, 31, 17, 7, 37, 32, 34, 20, 33, 38,
39, 22, 21, 24, 40, 23, 10, 12, 11, 25, 41, 18, 26, 14, 16, 35, 36,
15, 19, 13, 6, 4, 8, 9, 28, 30, 29, 5, 1, 3, 2, 27]
con una distancia total de 1986
```

4.- Propuesta de mejora: implementación de GRASP

Con el objetivo de tratar de mejorar la solución más eficiente encontrada hasta ahora **(1596 con algoritmos genéticos)** y acercarnos al resultado óptimo conocido para este problema **(1273)** se implementará una nueva solución haciendo uso de GRASP, adaptada a un tamaño del problema **N=42** para la que es viable.

```
# Parámetros del problema
alpha = 0.2
iteraciones = 100

mejor_solucion_grasp = []
mejor_distancia_grasp = 1596

for _ in range(iteraciones):
    # Inicialización del problema
    solucion = [0]
    Nodos = list(problem.get_nodes())
    ciudades_disponibles = Nodos
    ciudades_disponibles.remove(0)

    # Bucle de construcción inicial
    while ciudades_disponibles:
        ciudad_actual = solucion[-1]

        # Calculamos la distancia desde nuestra ciudad actual al resto
        distancias = {ciudad: distancia(ciudad_actual, ciudad,
problem) for ciudad in ciudades_disponibles}

        # Calculamos la cota y las ciudades candidatas
        cota = min(distancias.values()) +
alpha*(max(distancias.values())- min(distancias.values()))

        candidatos = [ciudad for ciudad, distancia in
distancias.items() if distancia <= cota]

        # Escogemos una ciudad aleatoriamente entre las candidatas
        nueva_ciudad = random.choice(candidatos)
        solucion.append(nueva_ciudad)
        ciudades_disponibles.remove(nueva_ciudad)

    # Bucle de búsqueda local
    mejora = True
    while mejora:
        mejora = False
        mejor_distancia = distancia_total(solucion, problem)
        #Doble bucle de comprobación de todas las posibles
```

```
inversiones. Si hay mejora, se realiza
        for i in range (1, len (solucion)):
            for j in range (i+1, len(solucion)):
                solucion_provisional = solucion [:i] + solucion
[i:j+1][::-1] + solucion [j+1:]
                distancia_provisional =
distancia_total(solucion_provisional, problem)
                if distancia_provisional < mejor_distancia:
                    mejor_distancia = distancia_provisional
                    mejor_solucion = solucion_provisional
                    mejora = True
        solucion = mejor_solucion
    if mejor_distancia < mejor_distancia_grasp:
        mejor_distancia_grasp = mejor_distancia
        mejor_solucion_grasp = mejor_solucion

print (mejor_solucion_grasp)
print ("Mejor distancia encontrada", mejor_distancia_grasp)
```

```
[0, 32, 34, 33, 20, 35, 36, 31, 17, 7, 37, 15, 16, 14, 19, 13, 5, 26,
18, 12, 11, 25, 10, 8, 41, 23, 9, 21, 40, 24, 39, 22, 38, 30, 29, 28,
27, 2, 3, 4, 6, 1]
Mejor distancia encontrada 1273
```

Podemos comprobar que en este caso la implementación ha sido exitosa ya que hemos conseguido encontrar la solución óptima del problema (**1273**) que conocíamos de antemano.

La mejor solución para esta variante del problema del agente viajero es:

[0, 32, 34, 33, 20, 35, 36, 31, 17, 7, 37, 15, 16, 14, 19, 13, 5, 26, 18, 12, 11, 25, 10, 8, 41, 23, 9, 21, 40, 24, 39, 22, 38, 30, 29, 28, 27, 2, 3, 4, 6, 1]
