Actividad Guiada 3

Javier Rodríguez Juárez

Link a Github

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 (2.27.1)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests) (1.26.16)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests) (2023.5.7)
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.10/dist-packages (from requests) (2.0.12)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests) (3.4)
Requirement already satisfied: tsplib95 in /usr/local/lib/python3.10/dist-packages (0.7.1)
Requirement already satisfied: Click>=6.0 in /usr/local/lib/python3.10/dist-packages (from tsplib95) (8.1.3)
Requirement already satisfied: Deprecated~=1.2.9 in /usr/local/lib/python3.10/dist-packages (from tsplib95) (1.2.14)
Requirement already satisfied: networkx~=2.1 in /usr/local/lib/python3.10/dist-packages (from tsplib95) (2.8.8)
Requirement already satisfied: tabulate~=0.8.7 in /usr/local/lib/python3.10/dist-packages (from tsplib95) (0.8.10)
Requirement already satisfied: wrapt<2,>=1.10 in /usr/local/lib/python3.10/dist-packages (from Deprecated~=1.2.9->tsplib95) (1.14.1
```

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

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

```
gzip: swiss42.tsp already exists; do you wish to overwrite (y or n)? y
```

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

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

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

```
#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
```

Funcionas basicas

BUSQUEDA ALEATORIA

```
#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(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)
```

```
################################################################################
# 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 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, 17, 10, 12, 7, 2, 22, 39, 30, 32, 3, 14, 6, 21, 11, 18, 34, 38, 40, 23, 1, 5, 26, 16, 19, 4, 27, 25, 29, 9, 33,
Distancia     : 3667
```

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,
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: 3667
Distancia Mejor Solucion Local: 3406
```

```
#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  35 , la mejor solución encontrada es: [0, 32, 20, 33, 34, 38, 22, 30, 29, 28, 27, 2, 3, 17, 31, 35, 36, 37, 5, 26,
Distancia     : 1599
```

```
#Busqueda Local con MULTI-START:
#  - 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 = []

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

  iteracion=0
  while(1):
    iteracion +=1
    vecina = genera_vecina(solucion_referencia)
    distancia_vecina = distancia_total(vecina, problem)

    if distancia_vecina < mejor_distancia:
      mejor_solucion = vecina
      mejor_distancia = distancia_vecina
    else:
      return mejor_distancia, mejor_solucion
```

```
    solucion_referencia = vecina


def busqueda_local_multiple(problem, iteraciones):
  mejor_solucion = []
  mejor_distancia = float('inf')
  it_num = 0

  for i in range(1, iteraciones + 1):
    dist, sol = busqueda_local(problem)

    if dist < mejor_distancia:
      mejor_solucion = sol
      mejor_distancia = dist
      it_num = i

    if i%10 == 0:
      print(f"Iteracion {i} de {iteraciones}")
      print(f"\tMejor distancia hasta el momento: {mejor_distancia} en la iteracion {it_num}")

  print("\n\nEn la solucion ", it_num, ", la mejor solución encontrada es:" , mejor_solucion)
  print("Distancia     :" , mejor_distancia)
  return mejor_solucion


sol = busqueda_local_multiple(problem, 50)
```

```
Iteracion 10 de 50
        Mejor distancia hasta el momento: 1716 en la iteracion 2
Iteracion 20 de 50
        Mejor distancia hasta el momento: 1591 en la iteracion 13
Iteracion 30 de 50
        Mejor distancia hasta el momento: 1534 en la iteracion 24
Iteracion 40 de 50
        Mejor distancia hasta el momento: 1454 en la iteracion 34
Iteracion 50 de 50
        Mejor distancia hasta el momento: 1454 en la iteracion 34


En la solucion  34 , la mejor solución encontrada es: [0, 27, 2, 3, 4, 6, 5, 26, 18, 13, 19, 14, 16, 15, 37, 17, 36, 35, 31, 20, 33
Distancia     : 1454
```