Actividad Guiada 3

Alejandro Vergara Richart

Github: https://github.com/alexveergara/algoritmos_optimizacion

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 (2.31.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests) (3.2.(
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests) (2.0.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests) (2023.7.22)
Collecting tsplib95
  Downloading tsplib95-0.7.1-py2.py3-none-any.whl (25 kB)
Requirement already satisfied: Click>=6.0 in /usr/local/lib/python3.10/dist-packages (from tsplib95) (8.1.7)
Collecting Deprecated~=1.2.9 (from tsplib95)
  Downloading Deprecated-1.2.14-py2.py3-none-any.whl (9.6 kB)
Collecting networkx~=2.1 (from tsplib95)
  Downloading networkx-2.8.8-py3-none-any.whl (2.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.0/2.0 MB 11.7 MB/s eta 0:00:00
Collecting tabulate~=0.8.7 (from tsplib95)
  Downloading tabulate-0.8.10-py3-none-any.whl (29 kB)
Requirement already satisfied: wrapt<2,>=1.10 in /usr/local/lib/python3.10/dist-packages (from Deprecated~=1.2.9->tsplib9
Installing collected packages: tabulate, networkx, Deprecated, tsplib95
  Attempting uninstall: tabulate
    Found existing installation: tabulate 0.9.0
    Uninstalling tabulate-0.9.0:
      Successfully uninstalled tabulate-0.9.0
  Attempting uninstall: networkx
    Found existing installation: networkx 3.1
    Uninstalling networkx-3.1:
      Successfully uninstalled networkx-3.1
Successfully installed Deprecated-1.2.14 networkx-2.8.8 tabulate-0.8.10 tsplib95-0.7.1
```

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.
!gzip -d swiss42.tsp.gz     #Descomprimir el fichero de datos



#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

```
#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)
```

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 pero pode
    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, 18, 4, 26, 5, 23, 40, 21, 12, 28, 33, 20, 1, 31, 35, 17, 39, 13, 11, 10, 24, 9, 19, 6, 32, 29, 8, 22,
Distancia     : 3681
```

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(
  #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 eval
    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,
      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
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: 3681
Distancia Mejor Solucion Local: 3367
```

```
#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 v
    if distancia_vecina < mejor_distancia:
      #mejor_solucion = copy.deepcopy(vecina)   #Con copia profunda. Las copias en python son por
      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  32 , la mejor solución encontrada es: [0, 3, 28, 29, 30, 22, 38, 31, 36, 35, 20, 33, 34, 32, 27, 2, 12,
Distancia     : 1649
```

Método multiarranque

```
import random

def busqueda_local(problem, num_arranques=10):
  mejor_solucion_global = []
  mejor_distancia_global = float('inf')

  for _ in range(num_arranques):
    # Generar una solución inicial aleatoria
    solucion_referencia = crear_solucion(Nodos)
    mejor_distancia_local = distancia_total(solucion_referencia, problem)

    iteracion = 0
    while True:
        iteracion += 1

        # Obtener la mejor vecina
        vecina = genera_vecina(solucion_referencia)

        # Evaluar la vecina
        distancia_vecina = distancia_total(vecina, problem)

        if distancia_vecina < mejor_distancia_local:
            mejor_solucion_local = vecina
            mejor_distancia_local = distancia_vecina
        else:
            break  # Terminar si no se mejora en esta iteración

        solucion_referencia = vecina

    # Actualizar la mejor solución global si es necesario
    if mejor_distancia_local < mejor_distancia_global:
        mejor_solucion_global = mejor_solucion_local
        mejor_distancia_global = mejor_distancia_local

  return mejor_solucion_global, mejor_distancia_global

# Ejemplo de uso
sol, dist = busqueda_local(problem, num_arranques=10)
print("La mejor solución encontrada es:" , sol)
print("Distancia     :" , dist)
```

```
La mejor solución encontrada es: [0, 32, 20, 31, 35, 36, 17, 37, 15, 16, 14, 19, 13, 12, 11, 25, 18, 26, 5, 6, 4, 2, 28,
Distancia     : 1597
```

```
import random
```

```
def busqueda_local(problem, num_arranques=10, max_iter=100, max_k=2):
    mejor_solucion_global = []
    mejor_distancia_global = float('inf')

    for _ in range(num_arranques):
        # Generar una solución inicial aleatoria
        solucion_referencia = crear_solucion(Nodos)
        mejor_distancia_local = distancia_total(solucion_referencia, problem)

        iteracion = 0
        while iteracion < max_iter:
            iteracion += 1

            # Aplicar el Variable Neighborhood Search (VNS)
            solucion_referencia, distancia_referencia = vns(solucion_referencia, problem, max_k)

            if distancia_referencia < mejor_distancia_local:
                mejor_solucion_local = solucion_referencia
                mejor_distancia_local = distancia_referencia

        # Actualizar la mejor solución global si es necesario
        if mejor_distancia_local < mejor_distancia_global:
            mejor_solucion_global = mejor_solucion_local
            mejor_distancia_global = mejor_distancia_local

    return mejor_solucion_global, mejor_distancia_global

# Variable Neighborhood Search (VNS)
def vns(solucion_referencia, problem, max_k):
    k = 1  # Inicializar el índice del vecindario
    distancia_referencia = distancia_total(solucion_referencia, problem)  # Inicializar distancia_

    while k <= max_k:
        vecindario = generar_vecindario(solucion_referencia, k)

        mejor_distancia_local = float('inf')  # Inicializar la mejor distancia local

        for vecina in vecindario:
            distancia_vecina = distancia_total(vecina, problem)
            if distancia_vecina < mejor_distancia_local:
                mejor_solucion_local = vecina
                mejor_distancia_local = distancia_vecina

        if mejor_distancia_local < distancia_referencia:
            solucion_referencia = mejor_solucion_local
            distancia_referencia = mejor_distancia_local
            k = 1  # Reiniciar el índice del vecindario si hay mejora
        else:
            k += 1  # Probar el siguiente vecindario si no hay mejora

    return solucion_referencia, distancia_referencia

# Función para generar un vecindario en el VNS
def generar_vecindario(solucion, k):
    vecindario = []

    if k == 1:
        # Vecindario 1: Intercambio de nodos
        for i in range(len(solucion)):
            for j in range(i + 1, len(solucion)):
                vecina = solucion[:]
                vecina[i], vecina[j] = vecina[j], vecina[i]
                vecindario.append(vecina)

    elif k == 2:
        # Vecindario 2: Inserción de un nodo en otra posición
        for i in range(len(solucion)):
            for j in range(len(solucion)):
                if i != j:
```

```
                    vecina = solucion[:]
                    nodo = vecina.pop(i)
                    vecina.insert(j, nodo)
                    vecindario.append(vecina)

        return vecindario

# Ejemplo de uso
sol, dist = busqueda_local(problem, num_arranques=10, max_iter=100, max_k=2)
print("Mejor solución encontrada:", sol)
print("Distancia :", dist)
```

```
Mejor solución encontrada: [29, 28, 2, 27, 3, 0, 1, 6, 4, 10, 25, 11, 12, 18, 26, 5, 13, 19, 14, 16, 15, 37, 7, 17, 31, 3
Distancia : 1345
```