Actividad Guiada 3

Autor: Roger Amorós Sirera

https://github.com/RogerAmoros13/algoritmos_de_optimizacion

In [1]:
```
# Instalación de librerias necesarias
!pip install requests
!pip install tsplib95

import gzip
```

```
Requirement already satisfied: requests in c:\users\roger\appdata\local\programs\python\python311\lib\site-packages (2.30.0)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\roger\appdata\local\programs\python\python311\lib\site-packages (from requests) (3.1.0)
Requirement already satisfied: idna<4,>=2.5 in c:\users\roger\appdata\local\programs\python\python311\lib\site-packages (from requests) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\roger\appdata\local\programs\python\python311\lib\site-packages (from requests) (1.26.16)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\roger\appdata\local\programs\python\python311\lib\site-packages (from requests) (2022.12.7)
[notice] A new release of pip available: 22.3.1 -> 23.1.2
[notice] To update, run: python.exe -m pip install --upgrade pip
Requirement already satisfied: tsplib95 in c:\users\roger\appdata\local\programs\python\python311\lib\site-packages (0.7.1)
Requirement already satisfied: Click>=6.0 in c:\users\roger\appdata\local\programs\python\python311\lib\site-packages (from tsplib95) (8.1.3)
Requirement already satisfied: Deprecated~=1.2.9 in c:\users\roger\appdata\local\programs\python\python311\lib\site-packages (from tsplib95) (1.2.14)
Requirement already satisfied: networkx~=2.1 in c:\users\roger\appdata\local\programs\python\python311\lib\site-packages (from tsplib95) (2.8.8)
Requirement already satisfied: tabulate~=0.8.7 in c:\users\roger\appdata\local\programs\python\python311\lib\site-packages (from tsplib95) (0.8.10)
Requirement already satisfied: colorama in c:\users\roger\appdata\local\programs\python\python311\lib\site-packages (from Click>=6.0->tsplib95) (0.4.6)
Requirement already satisfied: wrapt<2,>=1.10 in c:\users\roger\appdata\local\programs\python\python311\lib\site-packages (from Deprecated~=1.2.9->tsplib95) (1.14.1)
[notice] A new release of pip available: 22.3.1 -> 23.1.2
[notice] To update, run: python.exe -m pip install --upgrade pip
```

In [175…
```
def decompress(infile, tofile):
    with open(infile, 'rb') as inf, open(tofile, 'w', encoding='utf8') as tof:
        decom_str = gzip.decompress(inf.read()).decode('utf-8')
        tof.write(decom_str)
```

In [176…
```
import urllib
import tsplib95
import math
import random
import time

file = "swiss42.tsp"
urllib.request.urlretrieve("http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/swiss42.tsp.gz", file + ".gz")
decompress(file+".gz", file)
```

In [177…
```
problem = tsplib95.load(file)

Nodos = list(problem.get_nodes())

Aristas = list(problem.get_edges())

# Nodos
```

In [178…
```
# Al ser el nodo como tal un int, voy a cambiar la función de crear_nodos para utilizar la libreria
# random.shuffle para "barajar" la lista. De esta manera evitamos el bucle y hacer broadcast entre
# list y set, además de usar métodos que están de sobra implementados en python.
def crear_solucion(Nodos=Nodos):
    sol = Nodos.copy()[1:]
    random.shuffle(sol)
    sol = [0] + sol
    return sol

def distancia(a, b, problem=problem):
    return problem.get_weight(a, b)

def distancia_total(path, problem=problem):
    dist = 0
    n = len(path)
    for i in range(n - 1):
        dist += distancia(path[i], path[i+1], problem)
    return dist + distancia(path[n-1], path[0], problem)
```

In [179…
```
# Algoritmo que busca de manera aleatoria soluciones y se queda con la mejor
# que ha encontrado.

def busqueda_aleatoria(problem, N):
    # N es el numero de iteraciones
    best_sol = None
    best_dist = float("inf")
    for i in range(N):
        sol = crear_solucion()
        dist = distancia_total(sol)
        if dist < best_dist:
            best_sol = sol
            best_dist = dist
    print("Mejor solución: ", best_sol)
    print("Distancia     : ", best_dist)
    return best_sol

sol = busqueda_aleatoria(problem, 3000)
```

```
Mejor solución:  [0, 31, 28, 7, 2, 13, 15, 36, 33, 1, 12, 30, 22, 38, 34, 24, 39, 8, 16, 25, 21, 23, 3, 27, 35, 37, 20, 6, 32, 40, 9, 29, 4, 26, 14, 41, 10, 19, 5, 18, 11, 17]
Distancia     :  3790
```

In [180…
```
# Dada una solución comprueba cada uno de los swaps que puede realizar
# entre distintos nodos para ver si mejora la solución.

def genera_vecina(solucion, problem):
    best_sol = None
    best_dist = float("inf")
    len_sol = len(solucion)
    for i in range(1, len_sol-1):
        for j in range(i+1, len_sol):
            vecina = solucion[:i] + [solucion[j]] + solucion[i+1:j] + [solucion[i]] + solucion[j+1:]
            dist = distancia_total(vecina, problem)
            if dist < best_dist:
                best_dist = dist
                best_sol = vecina
#     print("Mejor solución: ", best_sol)
#     print("Distancia     : ", best_dist)
    return best_sol

# Dada una solución comprueba que pasaría si se inserta cada uno de los nodos
# en otra posición y si mejora lo guarda.

def genera_insert(solucion, problem):
    best_sol = None
    best_dist = float("inf")
    len_sol = len(solucion)
    for i in range(len_sol):
        for j in range(len_sol):
            if i == j:
                continue
            aux_list = solucion.copy()
            val = aux_list.pop(i)
            aux_list.insert(j, val)
            dist = distancia_total(aux_list, problem)
            if dist < best_dist:
                best_dist = dist
                best_sol = aux_list
#     print("Mejor solución: ", best_sol)
#     print("Distancia     : ", best_dist)
    return best_sol
```

In [181…
```
# Esta función realiza una busqueda local por vecindad o inserción reiterativa
# hasta que la solución ya no mejora.

# Mejores resultados obtenidos: 1430, 1440, 1535, 1616

def busqueda_local(ref_sol=False, mode="vecindad", problem=problem):
    # N es el numero de iteraciones
    if mode == "vecindad":
        func = genera_vecina
    elif mode == "insercion":
        func = genera_insert
    best_sol = None
    if not ref_sol:
        ref_sol = crear_solucion(Nodos)
    best_dist = distancia_total(ref_sol)
    iteration = 0
    while True:
        iteration += 1
        vecina = func(ref_sol, problem)
        dist = distancia_total(vecina, problem)
        if dist < best_dist:
            best_sol = vecina
            best_dist = dist

        else:
#             print("Mejor solución: ", best_sol)
#             print("Distancia     : ", best_dist)
            return best_sol, best_dist
        ref_sol = vecina

sol = busqueda_local(mode="insercion")
```

In [79]:
```
# Este algoritmo es una combinación de los tres vistos anteriormente. Por un lado,
# se genera una solución aleatoria a cada ciclo. A esta solución se le aplica una
# busqueda local por vecindad y si se encuentra una solución mejor que la actual,
# se le aplica a esta solución una busqueda local por inserción.

# Este orden se puede invertir mediante el parámetro first_func.

# La función se evalua por tiempo, siendo este aproximado, ya que puede sobrepasarse si
# entra a un ciclo "a última hora", pero esto no será más de 10 segundos.

# Resultados:
# Vecindad --> Inserción: 1449, 1469, 1515, 1519, 1652, 1728, 1755
# Inserción --> Vecindad: 1441, 1473, 1528, 1546, 1568, 1598, 1717

# A mi parecer, los resultados con la busqueda local por inserción aporta mejores resultados
# por lo que conviene realizarla en primera instancia y afinar con la busqueda local por vecindad.

def multi_step(problem, time_sec=100, first_func="vecindad"):
    st = time.time()
    et = time.time()
    best_sol = None
    best_dist = float("inf")
    if first_func == "vecindad":
        func1 = genera_vecina
        func2 = genera_insert
    elif first_func == "insercion":
        func1 = genera_insert
        func2 = genera_vecina

    while et-st < time_sec:
        ref_sol = crear_solucion(Nodos)
        while True:
            vecina = func1(ref_sol, problem)
            dist = distancia_total(vecina, problem)
            if dist < best_dist:
                best_sol = vecina
                best_dist = dist
                vecina = func2(vecina, problem)
                dist = distancia_total(vecina, problem)
                if dist < best_dist:
                    best_sol = vecina
                    best_dist = dist
                else:
                    break
                ref_sol = vecina
            else:
                break
            ref_sol = vecina
        et = time.time()

    print("Mejor solución: ", best_sol)
    print("Distancia     : ", best_dist)
    return best_sol

a = multi_step(problem, 60, "insercion")
```

```
Mejor solución:  [0, 32, 20, 33, 34, 30, 29, 39, 21, 40, 24, 38, 22, 9, 8, 28, 2, 27, 3, 4, 7, 31, 35, 36, 17, 37, 15, 16, 14, 19, 13, 18, 12, 11, 25, 41, 23, 10, 26, 5, 6, 1]
Distancia     :  1528
```

In [186…
```
# Algoritmo para obtener una solución voraz random. Se buscan los rand_set mejores
# nodos para realizar el siguiente paso y se escoje uno al azar de estos.

def greedy_randomized(Nodes=Nodos, problem=problem, rand_set=3):
    nodes = Nodos.copy()
    nodes.remove(0)
    sol = [0]
    while nodes:
        candidate_nodes = []
        best_dist = float("inf")
        worst_best_dist = float("inf")
        to_pop = []
        for i, node in enumerate(nodes):
            dist = distancia(node, sol[-1])
            if not candidate_nodes:
                candidate_nodes.append((i, node, dist))
                continue
            for j, cand in enumerate(candidate_nodes):
                if dist < cand[2]:
                    candidate_nodes.insert(j, (i, node, dist))
                    break
                if i > rand_set:
                    break
        res = random.choice(candidate_nodes)
        sol.append(res[1])
        nodes.pop(res[0])
    return sol, distancia_total(sol)
```

In [190…
```
# Método de GRASP (Greedy randomize adaptive search procedure)

# Con la función greedy_randomized encontramos soluciones distintas a cada iteración.
# Después se le aplican las busquedas locales que hemos visto.

# Ejecución con time_sec=60, max_iters=100

# Resultados: 1315, 1333, 1386, 1388

# Definitivamente esta es la mejor solución al problema. En 4 ejecuciones de 1~2 minutos se
# obtienen soluciones suficientemente buenas.

def grasp(problem, time_sec=100, first_func="insercion", rand_set=3, max_iters=40):
    st = time.time()
    et = time.time()
    best_sol = None
    best_dist = float("inf")
    second_func = "insercion" if first_func=="vecindad" else "vecindad"
    i = 0
    while et-st < time_sec and i < max_iters:
        i += 1
        et = time.time()
        cand_best_sol, cand_best_dist = greedy_randomized(Nodos, problem, rand_set)
        if cand_best_dist < best_dist:
            best_sol = cand_best_sol
            best_dist = cand_best_dist
        cand_best_sol, cand_best_dist = busqueda_local(cand_best_sol, first_func)
        if cand_best_dist < best_dist:
            best_sol = cand_best_sol
            best_dist = cand_best_dist
        cand_best_sol, cand_best_dist = busqueda_local(cand_best_sol, first_func)
        if cand_best_dist < best_dist:
            best_sol = cand_best_sol
            best_dist = cand_best_dist
    return best_sol, best_dist

sol, dist = grasp(problem, time_sec=60, max_iters=100)
```

```
print("Mejor solución: ", sol)
print("Distancia     : ", dist)
```

```
Mejor solución:  [17, 31, 32, 28, 27, 2, 3, 4, 5, 13, 19, 14, 16, 15, 37, 7, 0, 1, 6, 26, 18, 12, 11, 25, 10, 41, 23, 40, 24, 21, 39, 9, 8, 29, 30, 22, 38, 34, 33, 20, 35, 36]
Distancia     :  1388
```