

## AG3- Actividad Guiada 3

Nombre: Carlos Esteban Posada

**URL DRIVE:** [https://colab.research.google.com/drive/1Eapqfyk01k\\_nyCsLUMHFGPN2bDj8Sd98?usp=sharing](https://colab.research.google.com/drive/1Eapqfyk01k_nyCsLUMHFGPN2bDj8Sd98?usp=sharing)

**URL GITHUB:** [https://github.com/cposada8/03MAIR-Algoritmos-de-Optimizacion-CEPM/blob/main/carlos\\_esteban\\_posada\\_AG3.ipynb](https://github.com/cposada8/03MAIR-Algoritmos-de-Optimizacion-CEPM/blob/main/carlos_esteban_posada_AG3.ipynb)

### ▼ Instalación y carga de requisitos principales

En esta sección se instalan las bibliotecas necesarias y se cargan los datos para el ejercicio del TSP

```
!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/python3.6/dist-packages (2.23.0)
Requirement already satisfied: certifi<=2017.4.17 in /usr/local/lib/python3.6/dist-packages (2017.4.17)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.6/dist-packages (2.5)
Requirement already satisfied: urllib3!=1.25.0,!1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.6/dist-packages (1.25.0)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.6/dist-packages (3.0.2)
Requirement already satisfied: tsplib95 in /usr/local/lib/python3.6/dist-packages (0.7.0)
Requirement already satisfied: Click>=6.0 in /usr/local/lib/python3.6/dist-packages (6.7.0)
Requirement already satisfied: tabulate~0.8.7 in /usr/local/lib/python3.6/dist-packages (0.8.7)
Requirement already satisfied: networkx~2.1 in /usr/local/lib/python3.6/dist-packages (2.1)
Requirement already satisfied: Deprecated~1.2.9 in /usr/local/lib/python3.6/dist-packages (1.2.9)
Requirement already satisfied: decorator>=4.3.0 in /usr/local/lib/python3.6/dist-packages (4.3.0)
Requirement already satisfied: wrapt<2,>=1.10 in /usr/local/lib/python3.6/dist-packages (1.10)

import tsplib95          #Modulo para las instancias del problema del TSP
import random            #Modulo para generar números aleatorios
from math import e       #constante e
import copy              #Para copia profunda de estructuras de datos(en python la asignación es por referencia)
import math

import urllib.request    #Hacer llamadas http a paginas de la red

#http://elib.zib.de/pub/mp-testdata/tsp/tsplib/
#Documentacion :
# http://comopt.ifl.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://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/swiss42.tsp", f

#Coordendas 51-city problem (Christofides/Eilon)
#file = "eil51.tsp" ; urllib.request.urlretrieve("http://elib.zib.de/pub/mp-testdata/tsp/tspl

#Coordenadas - 48 capitals of the US (Padberg/Rinaldi)
#file = "att48.tsp" ; urllib.request.urlretrieve("http://elib.zib.de/pub/mp-testdata/tsp/tspl

('swiss42.tsp', <http.client.HTTPMessage at 0x7fa278ee5550>)

#Modulos extras, no esenciales
import numpy as np
import matplotlib.pyplot as plt
import imageio          #Para construir las imagenes con gif
from google.colab import files    #Para descargar ficheros generados con google colab

from tempfile import mkstemp      #Para genera carpetas y ficheros temporales
#import tempfile

#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

## ▼ Funciones generales

En esta sección se implementan algunas funciones generales que serán usadas a través de todo el project

```
def crear_solucion(n):
    # n es el tamaño de la lista solución.
    # esta función retorna una lista con todos los números del 0 al n-1
    # en un orden aleatorio
    solucion = np.random.choice(range(n), n, replace=False)
    return solucion

#Devuelve la distancia entre dos nodos
def distancia(a,b, problem):
    return problem.get_weight(int(a),int(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[-1],solucion[0], problem)
```

## ▼ Métodos de solución

En esta sección se implementarán 3 métodos para encontrar soluciones a este problema. Cabe destacar que por el gran tamaño de solución de este problema, no se garantizará encontrar una solución óptima, pero en los últimos dos métodos encontraremos soluciones buenas.

## ▼ Búsqueda aleatoria

```
def busqueda_aleatoria(problem, num_iter, verbose=True):
    # Esta función generará n soluciones aleatorias
    # y retornará la mejor de ellas y su correspondiente distancia
    # esta función retorna una tupla (mejor_sol, dist)
    n = len(list(problem.get_nodes()))
    mejor_solucion = None
    mejor_distancia = np.inf

    for i in range(num_iter):
        solucion_actual = crear_solucion(n)
        dist_actual = distancia_total(solucion_actual, problem)
```

```

dist_actual = distancia_total(solucion_actual, problem)

if dist_actual <= mejor_distancia:
    mejor_solucion = solucion_actual
    mejor_distancia = dist_actual
    if verbose:
        print(f"iteración {i}, distancia: {mejor_distancia}")
        print(mejor_solucion)
        print()
return mejor_solucion, mejor_distancia

solucion, dist = busqueda_aleatoria(problem, 10000, False)
print("Mejor solución:",solucion)
print("Distancia:", dist)

Mejor solución: [33 14 22 28 16 32 37 13 19  0 27 25 21  9  4 15 34  2  3 30 29 40 24 3
 23 41  7  5 10 17 36 31 20  8  6 12  1 26 18 11 35 38]
Distancia: 3706

```

## ▼ Búsqueda local

```

def swap_indices(lista, i, j):
    # esta función retorna una lista donde los elementos en las posiciones
    # i y j están intercambiados respecto a la lista original
    # ejemplo: [1, 2, 3, 4, 5], i=1, j=3
    # intercambiará los elementos en las posiciones i y j
    # retornará: [1, 4, 3, 2, 5] Nótese como se intercambiarón el 2 y el 4
    resp = lista.copy()
    resp[i], resp[j] = lista[j], lista[i]
    return resp

swap_indices([1, 2, 3, 4, 5], 1, 3)

[1, 4, 3, 2, 5]

def generar_vecinos_1(solucion):
    # genera soluciones vecinas cambiando todas las parejas posibles de nodos

    n = len(solucion)
    vecinos = [] # lista con todas las soluciones vecinas

    for i in range(n-1):
        for j in range(i+1, n):
            vecino = swap_indices(solucion, i, j)

```

```

        vecinos.append(vecino)
    return vecinos

def get_mejor_vecino(solucion, problem, generador_vecindad=generar_vecinos_1):
    # esta función retorna la solución vecina con mejor desempeño
    # y retorna también la mejor distancia
    # recibe la solución de referencia
    # y la función generadora de vecindad

    # 1. Generar todos los vecinos según la función generadora de vecindad
    vecinos = generador_vecindad(solucion)

    mejor_vecino = None
    mejor_distancia = np.inf
    for vecino in vecinos:
        dist_actual = distancia_total(vecino, problem)
        if dist_actual <= mejor_distancia:
            mejor_vecino = vecino
            mejor_distancia = dist_actual
    return mejor_vecino, mejor_distancia

def busqueda_local(problem, max_iter= 1000, solucion=None, verbose=False, generador_vecindad=
    # Esta función realizará una búsqueda local exhaustiva.
    # el punto de partida puede ser una solución pasada como argumento, en caso
    # de que este no sea entregado, el punto de partida será una solución aleatoria
    # se puede modificar el generador de vecindad

    n = len(list(problem.get_nodes()))

    # 1. Generar solución de referencia
    # será igual a la solución si esta fue pasada como argumento
    # de otro modo será una solución aleatoria
    solucion_referencia = solucion if solucion is not None else crear_solucion(n)

    # se inicializan la mejor solución y mejor distancia
    mejor_solucion = solucion_referencia
    mejor_distancia = distancia_total(mejor_solucion, problem)

    iteracion = 0 # contador para saber en qué iteración vamos
    hubo_mejora = True # marca para controlar si se mejoró o sino parar el algoritmo
    while iteracion < max_iter and hubo_mejora:
        iteracion += 1

        # se obtiene el mejor vecino con su respectiva distancia
        vecino, dist_vecino = get_mejor_vecino(solucion_referencia, problem, generador_vecindad=g

        if dist_vecino < mejor_distancia:
            mejor_solucion = vecino
            mejor_distancia = dist_vecino

```

```

    mejor_distancia = dist_vecino
else:
    hubo_mejora = False

if verbose:
    print(f"iteracion: {iteracion}, mejor_distancia: {mejor_distancia}")
    print(f"mejor_solucion {mejor_solucion}")
    print()
    solucion_referencia = vecino

return mejor_solucion, mejor_distancia, iteracion

# busqueda_local(problem, verbose=True)
solucion, dist, iteracion = busqueda_local(problem)
print(f"en la iteración {iteracion} la mejor solución encontrada es: {solucion}")
print("Distancia", dist)

    en la iteración 34 la mejor solución encontrada es: [ 2  3  0  7 37 15 16 14 19 13  5 2
    11 12 18  4 27 28 30 32 34 33 38 22 39 21 24 40 23  9]
    Distancia 1592

```

## ▼ Recocido simulado

```

def get_random_neighbor(solucion):
    # esta función retorna un vecino aleatorio para la solución dada
    n = len(solucion)

    # se obtienen los índices aleatorios a cambiar
    ind_a, ind_b = np.random.choice(range(n), 2, replace=False)

    return swap_indices(solucion, ind_a, ind_b)

def aceptar_por_probabilidad(temperatura, delta):
    epsilon = np.finfo(float).eps # epsilon para evitar divisiones por cero
    return random.random() < math.exp(-delta/(temperatura+epsilon))

def bajar_temperatura(temperatura, proporcion = 0.99):
    # esta función baja la temperatura por el método de descenso exponencial
    return temperatura*proporcion

def recocido_simulado(problem, temperatura, mejor_vecino=False, max_iter= 1000, solucion = Nc
    # esta función ejecuta una búsqueda por el método de recocido simulado
    # temperatura: es la temperatura inicial del método

```

```

# mejor_vecino: una variable booleana que indica si buscar el mejor de entre todos los veci
# cuando está en True, y sólo buscará un vecino aleatorio cuando está en Fals
# max_iter: máximo número de iteraciones, usado como criterio de parada
# solucion: solución de partida, si no se pasa una solución se generará una aleatoria
# min_tem: la temperatura mínima, usada como criterio de parada del algoritmo
# verbose: variable booleana para indicar si se requiere imprimir el avance por pantalla o

n = len(list(problem.get_nodes()))
solucion_referencia = solucion if solucion is not None else crear_solucion(n)
dist_referencia = distancia_total(solucion_referencia, problem)

mejor_solucion = None
mejor_distancia = np.inf

iteracion = 0
while temperatura > min_tem and iteracion < max_iter:
    iteracion += 1

    # forma de obtener el mejor vecino
    if mejor_vecino:
        vecino = get_mejor_vecino(solucion_referencia, problem)[0]
    else:
        vecino = get_random_neighbor(solucion_referencia)
    # cálculo de la distancia del vecino
    dist_vecino = distancia_total(vecino, problem)

    # Actualización de la mejor solución
    if dist_vecino < mejor_distancia:
        mejor_solucion = vecino
        mejor_distancia = dist_vecino

    # Actualización de la solución de referencia
    if dist_vecino < dist_referencia or aceptar_por_probabilidad(temperatura, abs(dist_vecino - dist_referencia)):
        solucion_referencia = vecino
        dist_referencia = dist_vecino

    # se baja la temperatura
    temperatura = bajar_temperatura(temperatura)

    if verbose:
        print("iteración", iteracion)
        print("temperatura restante:", temperatura)
        print("mejor distancia", mejor_distancia)
        print("mejor solucion", mejor_solucion)
        print()

return mejor_solucion, mejor_distancia, iteracion

```

```
# recocido simulado elección aleatoria
solucion, dist, iteracion = recocido_simulado(problem, 100000, max_iter=10000)
print(f"en la iteración {iteracion} la mejor solución encontrada es: {solucion}")
print("Distancia", dist)

en la iteración 2062 la mejor solución encontrada es: [33 32 28 10 41 9 23 40 24 21 39
13 19 14 16 1 0 34 20 17 37 15 11 25 4 7 31 36 35]
Distancia 2075
```



## ▼ Mejorar Nota:

Encontrar mejoras a los 2 métodos anteriores

## ▼ Mejora a la búsqueda local

Se implementará una métrica de vecindad donde no se cambian 2 nodos sino 3 nodos. Esto implica que cada solución tendrá muchos más vecinos, generando también que el tiempo computacional crezca

```
def generar_vecinos_2(solucion):
    # genera soluciones vecinas cambiando todos los tríos de nodos posibles

    n = len(solucion)
    vecinos = [] # lista con todas las soluciones vecinas

    for i in range(n-2):
        for j in range(i+1, n-1):
            for k in range(j+1, n):
                vec1 = swap_indices(solucion, i, j)
                vec2 = swap_indices(solucion, i, k)
                vec3 = swap_indices(solucion, k, k)
                vecinos += [vec1, vec2, vec3]
    return vecinos
```

Se creó otra función de vecindad y esta se pasa al método de búsqueda local, así que no se tiene que modificar nada del código usado en la sección correspondiente.

```
solucion, dist, iteracion = busqueda_local(problem, generador_vecindad=generar_vecinos_2, ver
print(f"en la iteración {iteracion} la mejor solución encontrada es: {solucion}")
print("Distancia", dist)
```



```
iteracion: 1, mejor_distancia: 4208
mejor_solucion [14 26 18 10  8  6 37  3 35 20 36  2 19  5 39 23 32 15 41 27 25 33 11
 9 34 30 29 31  1 22 38  7 21 24 40  0 13 17 12  4 16]

iteracion: 2, mejor_distancia: 3871
mejor_solucion [14 26 18 10  8  6 37  3 35 20 36  2 19  5 39 23 32 15 41 27 25 12 11
 9 34 30 29 31  1 22 38  7 21 24 40  0 13 17 33  4 16]

iteracion: 3, mejor_distancia: 3557
mejor_solucion [14 26 18 10  8  6 37  3 35 20 36  2 19  5 39 23 32 15  7 27 25 12 11
 9 34 30 29 31  1 22 38 41 21 24 40  0 13 17 33  4 16]

iteracion: 4, mejor_distancia: 3281
mejor_solucion [14 26 18 10  8  6 37 36 35 20  3  2 19  5 39 23 32 15  7 27 25 12 11
 9 34 30 29 31  1 22 38 41 21 24 40  0 13 17 33  4 16]

iteracion: 5, mejor_distancia: 3076
mejor_solucion [14 26 18 10  8  6 37 36 35 20  3  2 19  5 41 23 32 15  7 27 25 12 11
 9 34 30 29 31  1 22 38 39 21 24 40  0 13 17 33  4 16]

iteracion: 6, mejor_distancia: 2911
mejor_solucion [14 26 18 10  8  6 37 36 35 20  3  2 19  5 41 23 32 15  7 27 25 12 11
 9 34 30 29  4  1 22 38 39 21 24 40  0 13 17 33 31 16]

iteracion: 7, mejor_distancia: 2754
mejor_solucion [14 26 18 10  8  6 37 36 35 20  3  2 19  5 41 23  9 15  7 27 25 12 11
32 34 30 29  4  1 22 38 39 21 24 40  0 13 17 33 31 16]

iteracion: 8, mejor_distancia: 2685
mejor_solucion [14 26 18 10  8  6 37 36 35 20  3  2 19  5 41 23  9 13  7 27 25 12 11
32 34 30 29  4  1 22 38 39 21 24 40  0 15 17 33 31 16]

iteracion: 9, mejor_distancia: 2614
mejor_solucion [14 26 18 10  8  6 37 36 35 20  3 13 19  5 41 23  9  2  7 27 25 12 11
32 34 30 29  4  1 22 38 39 21 24 40  0 15 17 33 31 16]

iteracion: 10, mejor_distancia: 2499
mejor_solucion [14 26 18 10  8  6 37 36 35 20  7 13 19  5 41 23  9  2  3 27 25 12 11
32 34 30 29  4  1 22 38 39 21 24 40  0 15 17 33 31 16]

iteracion: 11, mejor_distancia: 2427
mejor_solucion [14 26 18 10  8  6 37 36 35 31  7 13 19  5 41 23  9  2  3 27 25 12 11
32 34 30 29  4  1 22 38 39 21 24 40  0 15 17 33 20 16]

iteracion: 12, mejor_distancia: 2389
mejor_solucion [14 26 18 10  8  6 37 36 35 31 15 13 19  5 41 23  9  2  3 27 25 12 11
32 34 30 29  4  1 22 38 39 21 24 40  0  7 17 33 20 16]

iteracion: 13, mejor_distancia: 2360
mejor_solucion [14 26 18 10  8  6 37 36 35 31 15 13 19  5 41 23  9  2  3 27 25 12 11
32 34 30 29  4 28 22 38 39 21 24 40  0  7 17 33 20 16]

iteracion: 14, mejor_distancia: 2322
mejor_solucion [14 26 18 10  8  6 37 36 35 31 15 13 19  5 41 23  9  2  3  4 25 12 11
32 34 30 29 27 28 22 38 39 21 24 40  0  7 17 33 20 16]
```

```
iteracion: 15, mejor_distancia: 2299
```

```
mejor solucion [14 26 18 10 8 6 37 36 35 31 15 13 10 5 11 22 9 7 3 4 25 12 11]
```

Este método no demuestra mejores resultados que la vecindad definida en la sección de búsqueda local.

No se puede concluir que el método de vecindad con el criterio 2-opt sea el mejor, con el experimento acá presentado sólo podemos concluir que la métrica presentada modificando 3 nodos **NO** es mejor que la 2-opt

## ▼ Mejora al recocido simulado

En esta sección experimentaremos con una variación al recocido simulado, donde a la hora de encontrar un vecino, no nos iremos por uno **aleatorio** sino que escogeremos al mejor vecino posible

```
# recocido simulado elección exhaustiva: cambiamos el parámetro "mejor_vecino" a True
solucion, dist, iteracion = recocido_simulado(problem, 1000, mejor_vecino=True, max_iter=50, \
print(f"en la iteración {iteracion} la mejor solución encontrada es: {solucion}")
print("Distancia", dist)
```

```
iteración 42
```

```
temperatura restante: 655.6592205741434
```

```
mejor distancia 1911
```

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

```
iteración 43
```

```
temperatura restante: 649.102628368402
```

```
mejor distancia 1911
```

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

```
iteración 44
```

```
temperatura restante: 642.6116020847179
```

```
mejor distancia 1899
```

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

```
iteración 45
```

```
temperatura restante: 636.1854860638707
```

```
mejor distancia 1895
```

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

```

iteración 46
temperatura restante: 629.823631203232
mejor distancia 1895
mejor solucion [12 10 29 30 32 34 38 22 28 6 19 14 16 15 37 36 35 20 33 39 24 40 21
26 5 13 11 25 41 23 8 2 27 0 31 17 7 1 3 4 18]

iteración 47
temperatura restante: 623.5253948911997
mejor distancia 1895
mejor solucion [12 10 29 30 32 34 38 22 28 6 19 14 16 15 37 36 35 20 33 39 24 40 21
26 5 13 11 25 41 23 8 2 27 0 31 17 7 1 3 4 18]

iteración 48
temperatura restante: 617.2901409422876
mejor distancia 1895
mejor solucion [12 10 29 30 32 34 38 22 28 6 19 14 16 15 37 36 35 20 33 39 24 40 21
26 5 13 11 25 41 23 8 2 27 0 31 17 7 1 3 4 18]

iteración 49
temperatura restante: 611.1172395328647
mejor distancia 1895
mejor solucion [12 10 29 30 32 34 38 22 28 6 19 14 16 15 37 36 35 20 33 39 24 40 21
26 5 13 11 25 41 23 8 2 27 0 31 17 7 1 3 4 18]

iteración 50
temperatura restante: 605.006067137536
mejor distancia 1895
mejor solucion [12 10 29 30 32 34 38 22 28 6 19 14 16 15 37 36 35 20 33 39 24 40 21
26 5 13 11 25 41 23 8 2 27 0 31 17 7 1 3 4 18]

en la iteración 50 la mejor solución encontrada es: [12 10 29 30 32 34 38 22 28 6 19
26 5 13 11 25 41 23 8 2 27 0 31 17 7 1 3 4 18]
Distancia 1895

```

```

# comparación de ambos métodos de recocido simulado: vecino aleatorio vs el mejor vecino
num_simulaciones = 10
distancias_rc_normal = []
distancias_rc_modif = []
for i in range(num_simulaciones):
    print(f"simulación # {i}")
    solucion = crear_solucion(42)
    _, dist_rc_normal, _ = recocido_simulado(problem, 100000, max_iter=10000)
    _, dist_rc_modif, _ = recocido_simulado(problem, 1000, mejor_vecino=True, max_iter=50)
    distancias_rc_normal.append(dist_rc_normal)
    distancias_rc_modif.append(dist_rc_modif)

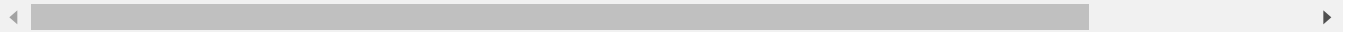
simulación # 0
simulación # 1
simulación # 2
simulación # 3
simulación # 4
simulación # 5
simulación # 6

```

```
simulación # 7  
simulación # 8  
simulación # 9
```

```
print("distancias del recocido simulado normal:", distancias_rc_normal)  
print("distancias del recocido simulado modificado:", distancias_rc_modif)
```

```
distancias del recocido simulado normal: [2083, 2158, 1925, 2021, 2029, 1989, 1895, 201  
distancias del recocido simulado modificado: [1680, 1761, 1742, 1518, 1714, 1785, 1660,
```



```
print("media de distancias del recocido simulado normal:", np.mean(distancias_rc_normal))  
print("media de distancias del recocido simulado modificado:", np.mean(distancias_rc_modif))
```

```
media de distancias del recocido simulado normal: 1997.0  
media de distancias del recocido simulado modificado: 1714.1
```

Se puede observar cómo en vez de trabajar con un vecino aleatorio, si buscamos todos los vecinos a la vez y trabajamos con el mejor, el algoritmo obtiene mejores resultados.

La comparación explícita es: en recocido simulado con vecino aleatorio se encontraban soluciones en una media de distancia superior a la media de distancia del método del mejor vecino

