Actividad Guiada 3 de Algoritmos de Optimización

- **Nombre y apellidos:** Gerardo Mauricio Gutierrez Quintana
- **URL Colab:** https://colab.research.google.com/drive/1oaqkKh7OmyfPJHgnMnS-E2GNZ2oqbZFn?usp=sharing
- **URL Github:** https://github.com/MauGutierrez/03MAIR---Algoritmos-de-Optimizacion---2021/tree/main/AG3

In [1]:
```
!pip install requests
!pip install tsplib95
```

```
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (2.23.0)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from reques
ts) (2.10)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.
7/dist-packages (from requests) (1.24.3)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from r
equests) (3.0.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from
requests) (2021.5.30)
Collecting tsplib95
  Downloading https://files.pythonhosted.org/packages/a0/2b/b1932d3674758ec5f49afa72d4519334a5ac2a
ac4d96cfd416eb872a1959/tsplib95-0.7.1-py2.py3-none-any.whl
Requirement already satisfied: Click>=6.0 in /usr/local/lib/python3.7/dist-packages (from tsplib9
5) (7.1.2)
Requirement already satisfied: tabulate~=0.8.7 in /usr/local/lib/python3.7/dist-packages (from tsp
lib95) (0.8.9)
Requirement already satisfied: networkx~=2.1 in /usr/local/lib/python3.7/dist-packages (from tspli
b95) (2.5.1)
Collecting Deprecated~=1.2.9
  Downloading https://files.pythonhosted.org/packages/fb/73/994edfcba74443146c84b91921fcc269374354
118d4f452fb0c54c1cbb12/Deprecated-1.2.12-py2.py3-none-any.whl
Requirement already satisfied: decorator<5,>=4.3 in /usr/local/lib/python3.7/dist-packages (from n
etworkx~=2.1->tsplib95) (4.4.2)
Requirement already satisfied: wrapt<2,>=1.10 in /usr/local/lib/python3.7/dist-packages (from Depr
ecated~=1.2.9->tsplib95) (1.12.1)
Installing collected packages: Deprecated, tsplib95
Successfully installed Deprecated-1.2.12 tsplib95-0.7.1
```

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt
# Libreria usada para construir las imagenes con gif
import imageio
# Libreria usada para descargar ficheros generados con google colab
from google.colab import files
# Libreria usada para generar carpetas y ficheros temporales
from tempfile import mkstemp
# Libreria usada para generar valores aleatorios
import random

import urllib.request
import tsplib95
import math

#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("http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/swiss42.tsp", file)

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

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

Out[3]:
```
('swiss42.tsp', <http.client.HTTPMessage at 0x7f1621a53750>)
```

In [36]:
```
# Carga de datos
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)
```

Out[36]: 15

In [37]:
```
# Funciones básicas

# Se genera una solucion aleatoria con comienzo 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)
```

Búsqueda Aleatoria

In [38]:
```
def busqueda_aleatoria(problem, N):
```

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

  mejor_solucion = []
  mejor_distancia = float('inf')

  for i in range(N):
    solucion = crear_solucion(Nodos)
    distancia = distancia_total(solucion, problem)

    if distancia < mejor_distancia:
      mejor_solucion = solucion
      mejor_distancia = distancia

  print(f'Mejor solución: {mejor_solucion}')
  print(f'Distancia     : {mejor_distancia}')
  return mejor_solucion

solucion = busqueda_aleatoria(problem, 5000)
```

```
Mejor solución: [0, 2, 28, 1, 18, 14, 39, 30, 9, 3, 5, 19, 23, 35, 40, 24, 22, 29, 41, 26, 13, 11,
16, 21, 32, 17, 38, 33, 34, 20, 36, 37, 6, 4, 7, 15, 31, 12, 25, 10, 8, 27]
Distancia     : 3706
```

Búsqueda Local

In [40]:
```
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
  mejor_solucion = []
  mejor_distancia = 10e100

  for i in range(1, len(solucion)-1):
    for j in range(i+1, len(solucion)):
      # Se genera una nueva solucion intercambiando los dos nodos i, j
      # (usamos el operador + para concatenar las listas): 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 solucion
```

```
      distancia_vecina = distancia_total(vecina, problem)

      # Guardamos la mejor solucion
      if distancia_vecina <= mejor_distancia:
        mejor_distancia = distancia_vecina
        mejor_solucion = vecina

  return mejor_solucion

print(f'Distancia Solucion Inicial: {distancia_total(solucion, problem)}')
nueva_solucion = genera_vecina(solucion)
print(f'Distnaica Mejor Solucion Local: {distancia_total(nueva_solucion, problem)}')
```

```
Distancia Solucion Inicial: 3706
Distnaica Mejor Solucion Local: 3120
```

In [41]:

```
# Sobre el operador de vecindad 2-opt (funcion genera_vecina)
# Sin criterio de parada. Nos detenemos cuando no es posible mjeorar

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)

  # Contador para saber las iteraciones que hemos hecho
  iteracion = 0
  while 1:
    # Incrementamos el contador
    iteracion += 1

    # Obtenemos la mejor vecina
    vecina = genera_vecina(solucion_referencia)

    # Evaluamos para ver si mejoramos con respecto a lo encontrado hasta el momento
    distancia_vecina = distancia_total(vecina, problem)

    # Si no mejoramos, hay que termminar ya que hemos llegado a un mínimo local (según nuestro op
erador de vecindad de 2-opt)
```

```
    if distancia_vecina < mejor_distancia:
      # Guardamos la mejor solucion encontrada
      mejor_solucion = vecina
      mejor_distancia = distancia_vecina

    else:
      print(f'En la iteracion {iteracion}, la mejor solucion encontrada es: {mejor_solucion}')
      print(f'Distancia: {mejor_distancia}')
      return mejor_solucion

    solucion_referencia = vecina

sol = busqueda_local(problem)
```

```
En la iteracion 44, la mejor solucion encontrada es: [0, 3, 4, 6, 37, 7, 1, 27, 2, 28, 32, 20, 33,
34, 30, 29, 8, 9, 23, 41, 25, 10, 26, 5, 19, 13, 11, 12, 18, 17, 31, 35, 36, 15, 16, 14, 22, 39, 2
1, 40, 24, 38]
Distancia: 1873
```

Simulated Annealing

In [42]:
```
# Generador de 1 solucion vecina 2-opt 100% aleatoria (intercambiar 2 nodos)
# Mejorable eligiendo otra forma de elegir una vecina

def genera_vecina_aleatoria(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:]
```

In [43]:
```
# Función de probabilidad para aceptar peores soluciones
def probabilidad(T, d):
  if random.random() < math.exp(-1*d / T):
    return True
  else:
    return False

# Función de descenso de temperatura
```

```
# Función de descenso de temperatura
def bajar_temperatura(T):
  return T * 0.99
```

In [44]:
```
def recocido_simulado(problem, temperatura):
  solucion_referencia = crear_solucion(Nodos)
  distancia_referencia = distancia_total(solucion_referencia, problem)

  mejor_solucion = []
  mejor_distancia = 10e100

  N = 0
  while temperatura > .0001:
    N += 1

    # Genera una solución vecina
    vecina = genera_vecina_aleatoria(solucion_referencia)

    # Calcula su valor (distancia)
    distancia_vecina = distancia_total(vecina, problem)

    # Si es la mejor solucion de todas se guarda siempre
    if distancia_vecina < mejor_distancia:
      mejor_solucion = vecina
      mejor_distancia = distancia_vecina

    # Si la nueva vecina es mejor, la cambiamos
    # Si es peor, la cambiamos según la probabilidad que depende de T y delta (distancia_referenc
ia - distancia_vecina)
    if distancia_vecina < distancia_referencia or probabilidad(temperatura, abs(distancia_referen
cia - distancia_vecina)):
      solucion_referencia = vecina
      distancia_referencia = distancia_vecina

    # Bajamos la temperatura
    temperatura = bajar_temperatura(temperatura)

  print(f'La mejor solucion encontrada es {mejor_solucion}')
  print(f'Distancia: {mejor_distancia}')
  return mejor_solucion
```

```
sol = recocido_simulado(problem, 10000000)
```

```
La mejor solucion encontrada es [0, 1, 5, 6, 16, 13, 19, 14, 15, 31, 33, 34, 3, 7, 37, 17, 36, 35,
20, 32, 38, 39, 22, 21, 24, 40, 23, 41, 10, 25, 11, 12, 18, 26, 4, 8, 9, 29, 30, 2, 27, 28]
Distancia: 1839
```

Colonia de Hormigas

In [46]:
```
def add_nodo(problem, H, T):
  # Mejora: establecer una función de probabilidad para añadir un nuevo nodo.
  # Añadiremos un nuevo nodo dependiendo de los nodos más cercanos y de las feromonas depositadas
  Nodos = list(problem.get_nodes())
  return random.choice(list(set(range(1, len(Nodos))) - set(H)))

def incrementa_feromona(problem, T, H):
  # Incrementa según la calidad de la solución. Añadir una cantidad inversamente proporcional a l
a distancia total
  for i in range(len(H)-1):
    T[H[i]][H[i+1]] += 1000/distancia_total(H, problem)

  return T

def evaporar_feromonas(T):
  # Evapora 0.3 el valor de la feromona sin que baje de 1
  # Mejora: Podemos elegir diferentes funciones de evaporación dependiendo de la cantidad actual
 y de la suma total de feromonas depositadas
  T = [[ max(T[i][j] - 0.3, 1) for i in range(len(Nodos))] for j in range(len(Nodos))]
  return T
```

In [48]:
```
def hormigas(problem, N):
  # Nodos
  Nodos = list(problem.get_nodes())

  # Aristas
  aristas = list(problem.get_nodes())

  # Inicializa las aristas con una cantidad inicial de feromonas = 1
  # Mejora: inicializar con valores diferentes dependiendo diferentes criterios
```

```
  # Mejora: Inicializar con valores diferentes dependiendo diferentes criterios
  T = [[ 1 for _ in range(len(Nodos)) ] for _ in range(len(Nodos))]

  # Se generan los agentes (hormigas) que serán estructuras de caminos desde 0
  hormiga = [[0] for _ in range(N)]

  # Recorre cada agente construyendo la solucion
  for h in range(N):
    # Para cada agente se contruye un camino
    for i in range(len(Nodos)-1):

      # Elige el siguiente nodo
      nuevo_nodo = add_nodo(problem, hormiga[h], T)
      hormiga[h].append(nuevo_nodo)

    # Incrementa feromonas en esa arista
    T = incrementa_feromona(problem, T, hormiga[h])

    # Evapora feromonas
    T = evaporar_feromonas(T)

  # Seleccionamos el mejor agente
  mejor_solucion = []
  mejor_distancia = 10e100
  for h in range(N):
    distancia_actual = distancia_total(hormiga[h], problem)
    if distancia_actual < mejor_distancia:
      mejor_solucion = hormiga[h]
      mejor_distancia = distancia_actual

  print(f'La mejor solucion es: {mejor_solucion}')
  print(f'Distancia: {mejor_distancia}')


hormigas(problem, 1000)
```

```
La mejor solucion es: [0, 7, 5, 20, 23, 16, 37, 10, 27, 1, 19, 25, 26, 31, 36, 17, 2, 28, 24, 14,
40, 22, 38, 39, 29, 9, 21, 35, 33, 32, 34, 4, 3, 6, 15, 12, 41, 8, 30, 18, 13, 11]
Distancia: 3880
```

Práctica Individual

Búsqueda local con Entornos Variables

Como bien sabemos, el objetivo principal de los algoritmos de optimización es el de encontrar una solución o un conjunto de soluciones factibles X con las que se pueda optimizar una función $f(x)$. Habrá dos tipos de problemas de optimización, aquellos donde se busque maximizar la función $f(x)$ y por otro lado, aquellos donde se busque minimizar $f(x)$. Un ejemplo muy claro de optimización sería el de la signación de n trabajos a n personas con el objetivo de minimizar el costo total.

Dentro de los problemas de optimización es muy común que existan soluciones globales y soluciones locales. Diremos que X es el espacio de soluciones factibles del problema y que x^* será una solución óptima (o mínimo global).

Podemos decir que una $x^* \in X$ es una solución o un mínimo global si no encontramos una $x \in X$ tal que $f(x)<f(x^*)$. Uno de los inconvenientes de utilizar algoritmos como búsuqueda local, es que podríamos quedar estancados en un mínimo local que no necesariamente sea el mínimo global. Sin embargo, existen alternativas que nos ayudan a continuar nuestra búsqueda después de haber encontrado el primer optimo local. Una de ellas es una metaheurística llamda VNS o Búsqueda por Entornos Variables.

Básicamente las metaheurísticas VNS se basan en 3 principios:

- Un mínimo local con una estructura de entornos no lo es necesariamente con otra.
- Un mínimo global es mínimo local con todas las posibles estructuras de entornos.
- Para muchos problemas, los mínimos locales están relativamente próximos entre sí.

Podemos decir que el punto 1 y 2 sugieren el uso de varias estructuras de entornos en las búsquedas locales para problemas de optimización, y el punto 3, indica que los óptimos locales proporcionan información acerca del óptimo global.

Referencias

Moreno Pérez, J. (2003). *Búsqueda por Entornos Variables para Planificación Logística*. 1-2. https://jamoreno.webs.ull.es/www/papers/VNS2PL.pdf

Recocido Simulado - ¿Se puede mejorar con otra elección no tan aleato

Se podría decir que todas las metaheurísticas podrían ser divididas en tres tipos:

- Solución Inicial: en este tipo, el resultado del algortimo y el performance se verá afectado por la calidad de una solución inicial.
- Selección de vecions: se podría decir que en este tipo lo que se busca es obtener los mejores vecinos que ya que esto podría obtener ben de soluciones optimas locales.
- Estrategia de Optimización: en este tipo lo que se busca es obtener el mejor parametro para decidir si aceptamos o rechazamos una soluci

De manera resumida, en el paper "Improving the Neighborhood Selection Strategy in Simulated Annealing Using the Optimal Stopping Proble una estrategia para elegir los mejores vecinos haciendo uso del Optimal Stopping Problem.

El Optimal Stopping Problem es un problema del tipo de Negative Dynamic Problem que se basa en lo siguiente:

- Suponiendo que tenemos un sistema con estados no negativos, en cada sistema tenemos la oportunidad de tomar la decisión de detenern actual y ganar la recompensa correspondiente $R(i)$, o pagar el costo $C(i)$ y continuar el proceso. Si decidimos continuar, el siguiente estac estaremos será el estado j con una probabilidad P_{ij}.

Ahora, aplicando este problema al Algoritmo de Recocido Simulado, la elección del vecino ya no sería de manera aleatoria, ya que, en alcancemos el criterio de aceptación, para cada estructura de vecinos encontrarmos un valor de Threshold. Una vez que hayamos obtenido estructura de vecinos con el mayor valor de Threshold para realizar el cálculo de la solución.

Referencias

Alizamir, S. Rebennack, S. Pardalos, P. (2008). *Improving the Neighborhood Selection Strategy in Simulated Annealing Using the Op* https://www.researchgate.net/publication/221787142_Improving_the_Neighborhood_Selection_Strategy_in_Simulated_Annealing_Using_the_O