03MIAR_10_A Algoritmos de Optimización

AG3: Actividad Guiada 3

Máster en Inteligencia Artificial

Curso 2023-2024. Edición Oct-2023

Profr.: **Raúl Reyero**

- raul.reyero@professor.universidadviu.com (raul.reyero@professor.universidadviu.com)

Alumno: **Victor David Betancourt Leal**

- victor.betancourt@alumnos.viu.es (victor.betancourt@alumnos.viu.es)
- vbleal@gmail.com (vbleal@gmail.com)

Repositorio

- 📒 Notebook Colab:
- https://drive.google.com/file/d/1K52i3YtufDhOhKN8sc5odvT3DXdeASm_/view?usp=sharing (https://drive.google.com/file/d/1K52i3YtufDhOhKN8sc5odvT3DXdeASm_/view?usp=sharing)
- 🚀 Repositorio GitHub:
- https://github.com/vbleal/03MIAR (https://github.com/vbleal/03MIAR)

Índice

1. Búsqueda Aleatoria
2. Búsqueda Local
3. Simulated Annealing (Recocido Simulado)
4. Búsqueda Local Mejorada con Entornos Variables
5. Búsqueda Local Mejorada con Simulated Annealing

.

##Carga de librerias

In [ ]:
```
!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-pac
kages (2.31.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/pyt
hon3.10/dist-packages (from requests) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist
-packages (from requests) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.1
0/dist-packages (from requests) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.1
0/dist-packages (from requests) (2023.11.17)
Requirement already satisfied: tsplib95 in /usr/local/lib/python3.10/dist-pac
kages (0.7.1)
Requirement already satisfied: Click>=6.0 in /usr/local/lib/python3.10/dist-p
ackages (from tsplib95) (8.1.7)
Requirement already satisfied: Deprecated~=1.2.9 in /usr/local/lib/python3.1
0/dist-packages (from tsplib95) (1.2.14)
Requirement already satisfied: networkx~=2.1 in /usr/local/lib/python3.10/dis
t-packages (from tsplib95) (2.8.8)
Requirement already satisfied: tabulate~=0.8.7 in /usr/local/lib/python3.10/d
ist-packages (from tsplib95) (0.8.10)
Requirement already satisfied: wrapt<2,>=1.10 in /usr/local/lib/python3.10/di
st-packages (from Deprecated~=1.2.9->tsplib95) (1.14.1)
```

##Carga de los datos del problema

In [ ]:
```
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


#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://comopt.ifi.uni-heidelberg.de/software/TSPLI
!gzip -d swiss42.tsp.gz     #Descomprimir el fichero de datos

#Coordendas 51-city problem (Christofides/Eilon)
#file = "eil51.tsp" ; urllib.request.urlretrieve("http://comopt.ifi.uni-heidel

#Coordenadas - 48 capitals of the US (Padberg/Rinaldi)
#file = "att48.tsp" ; urllib.request.urlretrieve("http://comopt.ifi.uni-heidel
```

```
gzip: swiss42.tsp already exists; do you wish to overwrite (y or n)? n
        not overwritten
```

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

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

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

```
NOMBRE: swiss42
TIPO: TSP
COMENTARIO: 42 Staedte Schweiz (Fricker)
DIMENSION: 42
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
0 15 30 23 32 55 33 37 92 114 92 110 96 90 74 76 82 72 78 82 159 122 131 206 112 57 28 43 70
  15 0 34 23 27 40 19 32 93 117 88 100 87 75 63 67 71 69 62 63 96 164 132 131 212 106 44 33 5
  30 34 0 11 18 57 36 65 62 84 64 89 76 93 95 100 104 98 57 88 99 130 100 101 179 86 51 4 18
  23 23 11 0 11 48 26 54 70 94 69 75 75 84 84 89 92 89 54 78 99 141 111 109 89 89 11 11 11 54
  32 27 18 11 0 40 20 58 67 92 61 78 65 76 83 89 91 95 43 72 110 141 116 105 190 81 34 19 35
  55 40 57 48 40 0 23 55 96 123 78 75 36 36 66 66 63 95 34 34 137 174 156 129 224 90 15 59 75
  33 19 36 26 20 23 0 45 85 111 75 82 69 60 63 70 71 85 44 52 115 161 136 122 210 91 25 37 54
  37 32 65 54 58 55 45 0 124 149 118 126 113 80 42 42 40 40 87 87 94 158 158 163 242 135 65 6
  92 93 62 70 67 96 85 124 0 28 29 68 63 122 148 155 156 159 67 129 148 78 80 39 129 46 82 65
 114 117 84 94 92 123 111 149 28 0 54 91 88 150 174 181 182 181 95 157 159 50 65 27 102 65 11
  92 88 64 69 61 78 75 118 29 54 0 39 34 99 134 142 141 157 44 110 161 103 109 52 154 22 63 6
 110 100 89 89 78 75 82 126 68 91 39 0 14 80 129 139 135 167 39 98 187 136 148 81 186 28 61 9
  96 87 76 75 65 62 69 113 63 88 34 14 0 72 117 128 124 153 26 88 174 136 142 82 187 32 48 79
  90 75 93 84 76 36 60 80 122 150 99 80 72 0 59 71 63 116 56 25 170 201 189 151 252 104 44 95
  74 63 95 84 83 56 63 42 148 174 134 129 117 59 0 11 8 63 93 35 135 223 195 184 273 146 71 9
```

In [ ]:
```
#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)
```

Out[4]: 15

Definición de Funciones

In [ ]:

```
#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]}) - s
  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], p
```

.

.

BÚSQUEDA ALEATORIA

In [ ]:
```
##############################################################################
# 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 valo
  mejor_distancia = float('inf')                     #Inicializamos con un valo

  for i in range(N):                                 #Criterio de parada: repet
    solucion = crear_solucion(Nodos)                 #Genera una solucion aleat
    distancia = distancia_total(solucion, problem)   #Calcula el valor objetivo

    if distancia < mejor_distancia:                  #Compara con la mejor obte
      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, 41, 39, 21, 19, 17, 33, 4, 26, 3, 27, 8, 29, 10, 24,
40, 9, 35, 36, 31, 34, 25, 30, 14, 16, 32, 1, 6, 12, 23, 11, 38, 20, 28, 2,
5, 22, 7, 37, 15, 13]
Distancia     : 3561
```

.

.

BÚSQUEDA LOCAL

In [ ]:

```
##############################################################################
# BUSQUEDA LOCAL
##############################################################################
def genera_vecina(solucion):
  #Generador de soluciones vecinas: 2-opt (intercambiar 2 nodos) Si hay N nodo
  #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 b
    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.:
      vecina = solucion[:i] + [solucion[j]] + solucion[i+1:j] + [solucion[i]]

      #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, 3
print("Distancia Solucion Incial:" , distancia_total(solucion, problem))


nueva_solucion = genera_vecina(solucion)
print("Distancia Mejor Solucion Local:", distancia_total(nueva_solucion, probl
```

```
Distancia Solucion Incial: 3561
Distancia Mejor Solucion Local: 3287
```

In [ ]:
```
#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 e
    distancia_vecina = distancia_total(vecina, problem)

    #Si no mejoramos hay que terminar. Hemos llegado a un minimo local(según n
    if distancia_vecina < mejor_distancia:
      #mejor_solucion = copy.deepcopy(vecina)   #Con copia profunda. Las copia
      mejor_solucion = vecina                   #Guarda la mejor solución enco
      mejor_distancia = distancia_vecina

    else:
      print("En la iteracion ", iteracion, ", la mejor solución encontrada es:
      print("Distancia     :" , mejor_distancia)
      return mejor_solucion

    solucion_referencia = vecina


sol = busqueda_local(problem )
```

```
En la iteracion  30 , la mejor solución encontrada es: [0, 37, 15, 16, 14, 8,
39, 21, 40, 24, 22, 38, 32, 7, 6, 26, 18, 23, 9, 29, 30, 28, 5, 13, 19, 34, 3
3, 20, 35, 36, 31, 17, 12, 11, 25, 41, 10, 4, 2, 27, 3, 1]
Distancia     : 1990
```

.

.

SIMULATED ANNEALING (*Recocido Simulado*)

In [ ]:

```
##############################################################################
# SIMULATED ANNEALING
##############################################################################

#Generador de 1 solucion vecina 2-opt 100% aleatoria (intercambiar 2 nodos)
#Mejorable eligiendo otra forma de elegir una vecina.
def genera_vecina_aleatorio(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 a
  return solucion[:i] + [solucion[j]] + solucion[i+1:j] + [solucion[i]] + solu


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

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

In [ ]:
```
def recocido_simulado(problem, TEMPERATURA ):
  #problem = datos del problema
  #T = Temperatura

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

  mejor_solucion = []             #x* del seudocodigo
  mejor_distancia = 10e100        #F* del seudocodigo


  N=0
  while TEMPERATURA > .0001:
    N+=1
    #Genera una solución vecina
    vecina =genera_vecina_aleatorio(solucion_referencia)

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

    #Si es la mejor solución de todas se guarda(siempre!!!)
    if distancia_vecina < mejor_distancia:
        mejor_solucion = vecina
        mejor_distancia = distancia_vecina

    #Si la nueva vecina es mejor se cambia
    #Si es peor se cambia según una probabilidad que depende de T y delta(dist
    if distancia_vecina < distancia_referencia or probabilidad(TEMPERATURA, ab
      #solucion_referencia = copy.deepcopy(vecina)
      solucion_referencia = vecina
      distancia_referencia = distancia_vecina

    #Bajamos la temperatura
    TEMPERATURA = bajar_temperatura(TEMPERATURA)

  print("La mejor solución encontrada es " , end="")
  print(mejor_solucion)
  print("con una distancia total de " , end="")
  print(mejor_distancia)
  return mejor_solucion

sol  = recocido_simulado(problem, 10000000)
```

```
La mejor solución encontrada es [0, 12, 11, 13, 19, 16, 15, 37, 4, 41, 25, 1
8, 26, 6, 31, 20, 33, 35, 36, 17, 3, 10, 23, 9, 8, 28, 32, 34, 30, 29, 22, 3
8, 21, 40, 24, 39, 27, 2, 5, 14, 7, 1]
con una distancia total de 2150
```

.

.

.

#---

La **Búsqueda Local** tiene como ***desventaja*** que intensifica pero no diversifica. Para escapar de máximos/mínimos locales se tienen algunas alternativas:

1. Usar ***Búsqueda por Entornos Variables***.
2. Usar ***Búsqueda Tabú*** o ***Simulated Annealing*** para permitir movimientos peores respecto de la solución actual.
3. usar ***Búsquedas Multi-arranque*** para volver a comenzar con otras soluciones iniciales.

Búsqueda Local Mejorada con Entornos Variables

VNS: Variable Neighbourhood Search

Referencia:

- Búsqueda por Entornos Variables para Planificación Logística: https://jamoreno.webs.ull.es/www/papers/VNS2PL.pdf (https://jamoreno.webs.ull.es/www/papers/VNS2PL.pdf)

De acuerdo con **Moreno & Mladenović**, se tienen 4 tipos de VNS:

1. VNS Descendente
2. VNS ReducidaS
3. VNS Básica
4. VNS General

Se ejemplifica a continuación la *VNS Básica*.

La característica principal la **VNS Básica** es la alternancia sistemática entre el movimiento a través de diferentes estructuras de vecindad (mediante el proceso de "*shaking*") para explorar el espacio de soluciones y la aplicación de una búsqueda local para intentar mejorar la solución dentro de cada una de esas estructuras.

Pasos de la VNS Básica:

1. ***Shaking***: Se genera una solución vecina de la solución actual pero utilizando una estructura de vecindad diferente en cada iteración. Esto permite explorar nuevas regiones del espacio de soluciones que no son accesibles mediante la búsqueda local desde la solución actual.
2. ***Búsqueda Local***: Después del paso de shaking, se aplica una búsqueda local para mejorar la solución obtenida en el paso anterior. La idea es encontrar un óptimo local dentro del nuevo entorno generado.

3. ***Criterio de Movimiento***: La solución obtenida tras la búsqueda local se compara con la solución actual. Si la nueva solución es mejor, entonces se convierte en la nueva solución actual. Esto ayuda a la búsqueda a converger hacia soluciones de mejor calidad.
4. ***Actualización de la Estructura de Vecindad***: Si no se encuentra una mejor solución, se incrementa el índice de la estructura de vecindad (*k*) y se repite el proceso desde el paso de shaking. Si se encuentra una mejor solución, se reinicia el índice de la estructura de

In [ ]:
```
# Generador de vecinos 3-OPT mejorado
def generar_vecino_3opt_mejorado(solucion):
    mejor_solucion = solucion
    mejor_distancia = distancia_total(solucion, problem)
    for i in range(1, len(solucion) - 2):  # Ajuste en el rango para evitar ín
        for j in range(i + 1, len(solucion) - 1):
            for k in range(j + 1, len(solucion)):
                vecinos = [
                    solucion[:i] + solucion[j:k+1] + solucion[i:j] + solucion[
                    solucion[:i] + solucion[j:k+1][::-1] + solucion[i:j] + sol
                    solucion[:i] + solucion[i:j][::-1] + solucion[j:k+1] + sol
                    #
                ]
                for vecino in vecinos:
                    distancia_vecino = distancia_total(vecino, problem)
                    if distancia_vecino < mejor_distancia:
                        mejor_distancia = distancia_vecino
                        mejor_solucion = vecino
    return mejor_solucion
```

In [ ]:
```
#solucion = [1, 47, 13, 41, 40, 19, 42, 44, 37, 5, 22, 28, 3, 2, 29, 21, 50, 3
print("Distancia Solución Inicial:" , distancia_total(solucion, problem))

otra_solucion = generar_vecino_3opt_mejorado(solucion)
print("Distancia Mejor Solución:", distancia_total(otra_solucion, problem))
```

```
Distancia Solución Inicial: 3561
Distancia Mejor Solución: 3212
```

In [ ]:
```
# Búsqueda Local 3-OPT
def busqueda_local_con_3opt_mejorada(problem):
    solucion_referencia = crear_solucion(Nodos)
    mejor_distancia = distancia_total(solucion_referencia, problem)
    mejor_solucion = solucion_referencia
    iteracion = 0
    while True:
        iteracion += 1
        solucion_mejorada = generar_vecino_3opt_mejorado(solucion_referencia)
        distancia_mejorada = distancia_total(solucion_mejorada, problem)
        if distancia_mejorada < mejor_distancia:
            mejor_solucion = solucion_mejorada
            mejor_distancia = distancia_mejorada
        else:
            print(f"En la iteración {iteracion}, la mejor solución encontrada
            print(f"Distancia: {mejor_distancia}")
            return mejor_solucion
        solucion_referencia = solucion_mejorada
```

In [ ]:
```
# Ejemplo
sollocal_con_entornos = busqueda_local_con_3opt_mejorada(problem)
```

```
En la iteración 30, la mejor solución encontrada es: [0, 7, 17, 31, 35, 36, 3
7, 15, 16, 14, 19, 13, 5, 26, 18, 12, 11, 25, 10, 8, 9, 41, 23, 40, 24, 21, 3
9, 22, 38, 34, 33, 20, 32, 30, 29, 28, 27, 2, 3, 4, 6, 1]
Distancia: 1301
```

.

.

Búsqueda Local Mejorada con Simulated Annealing (*Recocido Simulado*)

In [ ]:
```python
import random
import math

# Generador de 1 solución vecina 3-opt 100% aleatoria
def generar_vecina_aleatoria_3opt(solucion):
    # Selecciona tres nodos aleatoriamente y genera vecinos
    i, j, k = sorted(random.sample(range(1, len(solucion)), 3))

    vecinos = [
        solucion[:i] + [solucion[j]] + solucion[i+1:j] + [solucion[i]] + soluc
        solucion[:i] + [solucion[k]] + solucion[i+1:j] + [solucion[j]] + soluc
        #
    ]

    # Escoge un vecino aleatoriamente
    vecino_aleatorio = random.choice(vecinos)
    return vecino_aleatorio
```

In [ ]:
```python
def probabilidad(temperatura, delta):
    if temperatura <= 0:
        return False
    return math.exp(-delta / temperatura) > random.random()

def bajar_temperatura(temperatura_actual):
    factor_enfriamiento = 0.99  # Factor de enfriamiento: configurable
    return temperatura_actual * factor_enfriamiento
```

In [ ]:
```python
def recocido_simulado(problem, temperatura_inicial):
    solucion_actual = crear_solucion(Nodos)
    distancia_actual = distancia_total(solucion_actual, problem)

    mejor_solucion = solucion_actual
    mejor_distancia = distancia_actual

    temperatura = temperatura_inicial
    while temperatura > 0.0001:
        vecina = generar_vecina_aleatoria_3opt(solucion_actual)
        distancia_vecina = distancia_total(vecina, problem)

        if distancia_vecina < mejor_distancia:
            mejor_solucion = vecina
            mejor_distancia = distancia_vecina

        delta_distancia = distancia_vecina - distancia_actual
        if delta_distancia < 0 or probabilidad(temperatura, delta_distancia):
            solucion_actual = vecina
            distancia_actual = distancia_vecina

        temperatura = bajar_temperatura(temperatura)

    print("La mejor solución encontrada es:", mejor_solucion)
    print("con una distancia total de:", mejor_distancia)
    return mejor_solucion
```

In [ ]:
```python
# Ejemplo
sollocal_con_recocido = recocido_simulado(problem, 10000)
```

```
La mejor solución encontrada es: [0, 1, 6, 26, 5, 3, 4, 38, 22, 39, 24, 40, 2
1, 10, 12, 18, 14, 16, 15, 37, 17, 33, 20, 34, 30, 29, 9, 8, 23, 41, 25, 11,
13, 19, 7, 36, 35, 31, 32, 28, 2, 27]
con una distancia total de: 1787
```

.

Conclusiones

Todos los métodos de búsqueda local y metaheurísticas han logrado mejorar considerablemente la solución inicial, lo que demuestra su efectividad para explorar el espacio de soluciones y encontrar soluciones más óptimas.

1. **Superioridad de la Búsqueda Local con VNS Básica**

La búsqueda local mejorada con VNS Básica ha proporcionado la mejor solución de todas, con la ***distancia más corta*** (`1301`). Esto indica que la estrategia de cambiar sistemáticamente entre diferentes estructuras de vecindad puede ser particularmente efectiva para escapar de óptimos locales y explorar más exhaustivamente el espacio de soluciones.

2. **Eficiencia del Simulated Annealing** (*Recocido Simulado*)

El Recocido Simulado y su variante mejorada presentan resultados prometedores, especialmente la versión mejorada, que muestra una mejora notable respecto al Recocido Simulado estándar. Esto sugiere que la adaptación de la temperatura y la aceptación de soluciones peores bajo ciertas condiciones pueden ayudar a evitar quedarse atrapado en mínimos locales, aunque no fue tan efectivo como la VNS Básica en este caso.

.