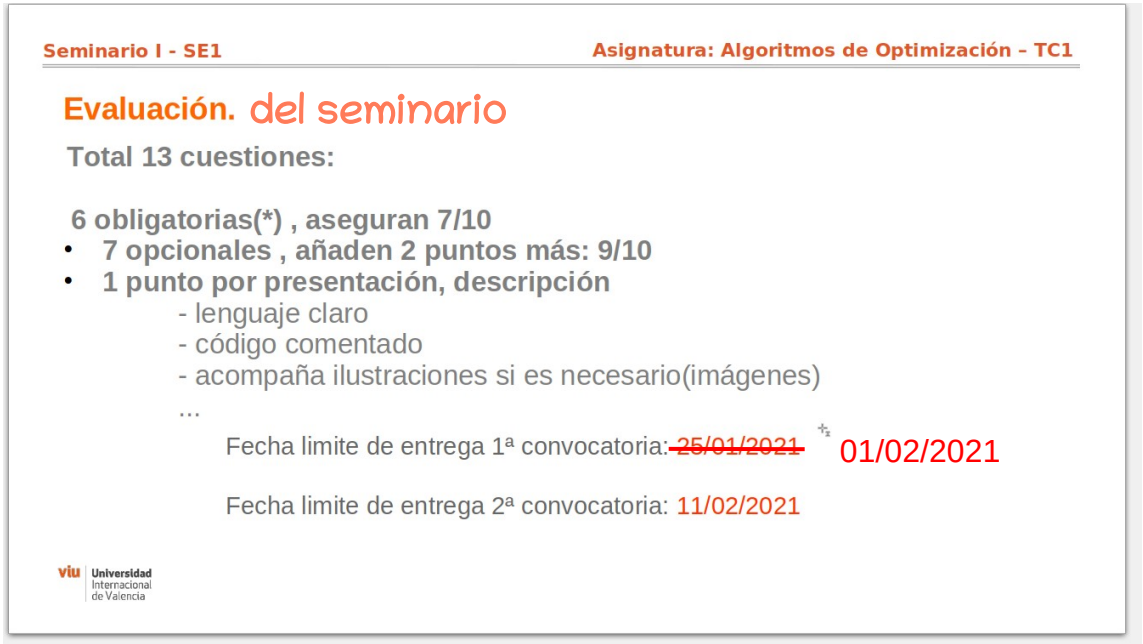
# AG3 – Actividad Guiada 3

## 03MAIR – Algoritmos de optimización

viu Universidad Internacional de Valencia

# Cambio fecha de entrega del Trabajo de seminario



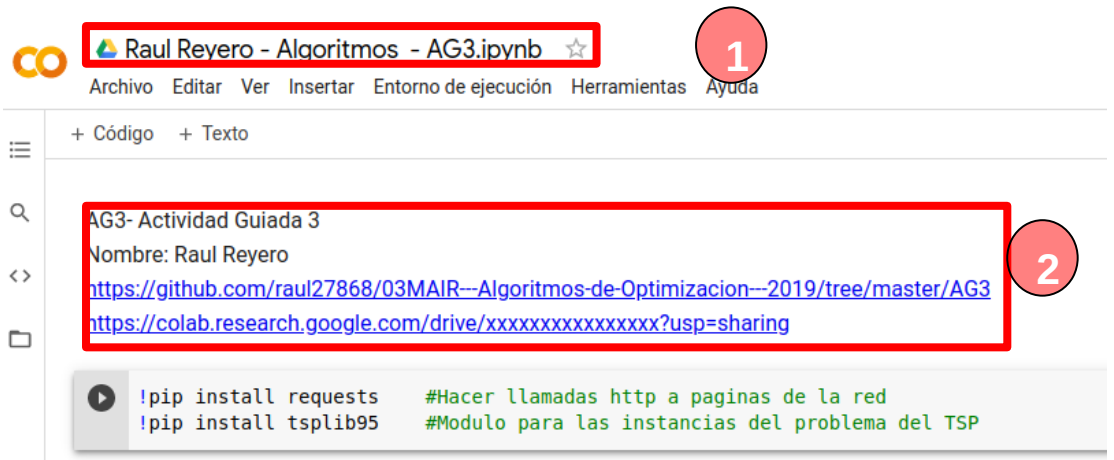
# Agenda

1. Librería TSPLIB para el agente viajero - TSP
2. Resolución por búsqueda aleatoria
3. Resolución por Búsqueda local
4. Resolución por recocido simulado
5. Planteamiento por Colonia de Hormigas - ACO

# Preparar la actividad en Google Colaboratory.

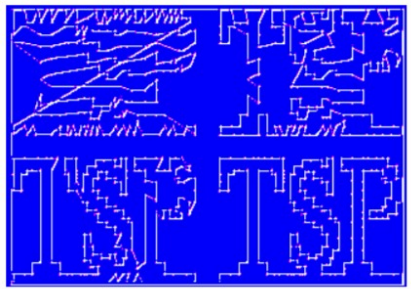
- Copiar con notebook en Google Colaboratory
  https://colab.research.google.com/drive/15WHTAWbuk3M9x4Xm5Isyi5j1NRKzqdsw?usp=sharing

- Renombra el documento python : **<nombre apellido>-AG3**
- Crear un texto con:
  * AG3- Actividad Guiada 3
  * Nombre Apellidos
  * Url a la carpeta **AG3** de GitHub



# El problema del agente viajero – TSP. TSPLIB

Juegos de datos para poner a prueba nuestros diseños para resolver el problema del TSP



http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html

## TSPLIB

TSPLIB is a library of sample instances for the TSP (and related problems) from various sources and of various types. Instances of the following problem classes are available.

### Symmetric traveling salesman problem (TSP)

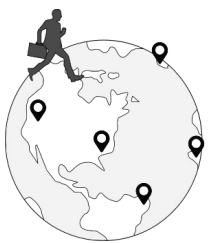
Given a set of n nodes and distances for each pair of nodes, find a roundtrip of minimal total length visiting each node exactly once. The distance from node i to node j is the same as from node j to node i.

TSP data

Best known solutions for symmetric TSPs

# El problema del agente viajero - TSP

- Es el problema más estudiado.
- Sirve de test para los diseños de nuevos algoritmos o técnicas.
- Para simplificar, suponemos todos los nodos conectados y comenzamos y terminamos por el nodo 0.

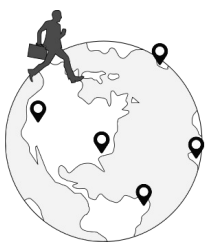


# Preparación de los datos

## Instalación del módulo tsplib y descargar el juego de datos

```
[1]  !pip install requests    #Hacer llamadas http a paginas de la red
     !pip install tsplib95    #Modulo para las instancias del problema del TSP
```

```
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)
```



# Preparación de los datos

## Cargar una instancia del problema: swiss42.tsp

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

#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)
```

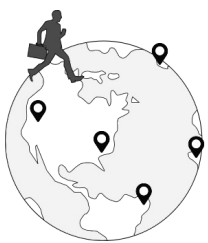
```
('swiss42.tsp', <http.client.HTTPMessage at 0x7fa34ac46a90>)
```

# Preparación de los datos

## Cargar datos del problema

```
problem = tsplib95.load_problem(file)


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

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

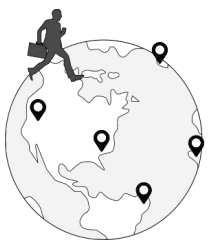
```
NOMBRE: swiss42
TIPO: TSP
COMENTARIO: 42 Staedte Schweiz (Fricker)
DIMENSION: 42
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
0 15 30 23 32 55 33 37 92 114 92 110 96 90 74 76 82 72 78 82 159 122 131 206 112 57 28 43 70
  15 0 34 23 27 40 19 32 93 117 88 100 87 75 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
```



# Preparación de los datos

Probamos algunas funciones del problema

```
#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)
```



# Preparación de los datos

## Datos del problema

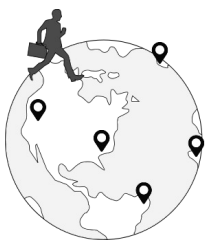
Para n=42 ciudades(nodos) el total de soluciones es:

$$(n-1)!/2 = 334525266131638071081700620534407516651520000000000/2$$

La distancia para la mejor solución encontrada al problema swiss42.tsp es:

*http://elib.zib.de/pub/mp-testdata/tsp/tsplib/stsp-sol.html*

**1273**



# Preparación de los datos

## Algunas funciones generales

```python
#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)
```

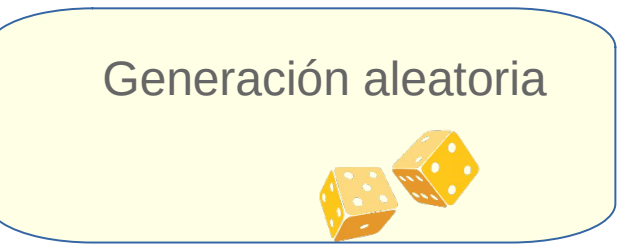
Toma un nodo no elegido anteriormente

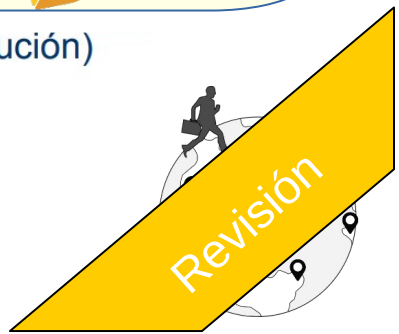
# Metaheurísticas de búsquedas

## Búsqueda aleatoria

**Definición:** Es un proceso por el que se van generando soluciones aleatorias en cada iteración y se devuelve la mejor.

```
Inicio
    GENERA(Solución Inicial)
    Solución Actual ← Solución Inicial;
    Mejor Solución ← Solución Actual;
  Repetir
    GENERA(Solución Actual);
    Si Objetivo(Solución Actual) es mejor que Objetivo(Mejor Solución)
          entonces Mejor Solución ← Solución Actual;
  Hasta (Criterio de parada);
  DEVOLVER (Mejor Solución);
Fin
```




viu Universidad Internacional de Valencia

# Metaheurísticas de búsquedas

## Búsqueda aleatoria

```python
def busqueda_aleatoria(problem, N):

  Nodos = list(problem.get_nodes())

  mejor_solucion = []
  mejor_distancia = 10e100                          #Inicializamos con un valor alto

  for i in range(N):                                #Criterio de parada: repetir N veces pero podemos incluir otros
    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



solucion = busqueda_aleatoria(problem, 5000)
```

```
Mejor solución: [0, 5, 10, 18, 8, 12, 15, 16, 35, 21, 33, 38, 22, 39, 40, 34, 4, 36, 31, 32, 24, 7, 13, 37, 19, 28, 29, 27, 23,
Distancia     : 3712
```

# Metaheurísticas de búsquedas

## Búsqueda local. Generador de vecindad

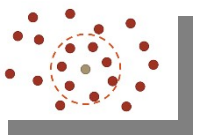
```python
def genera_vecina(solucion):
  #Generador de soluciones vecinas: 2-opt (intercambiar 2 nodos) Si hay N nodos se generan (N-1)x(N-2)/2 soluciones
  #print(solucion)
  mejor_solucion = []
  mejor_distancia = 10e100
  for i in range(1,len(solucion)-1):          #Recorremos todos los nodos en bucle doble para evaluar todos los intercambios 2-op
    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] = [1,2,3]
      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, 49, 10, 39, 33, 45, 15, 24, 4
print("Distancia Solucion Incial:" , distancia_total(solucion, problem))


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

```
Distancia Solucion Incial: 3712
Distancia Solucion Local: 3255
```

Prueba todas los posibles intercambios 2-opt
Total 41*40/2= 820 posibilidades

¿Cómo serian otros generadores?

- se elige una sub-lista y se invierte el orden
- se elige una sub-lista y se baraja
- otros



# Metaheurísticas de búsquedas

## Búsqueda local

```
#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 terminas. Hemos llegado a un minimo local(según nuestro operador de vencindad)
    if distancia_vecina < mejor_distancia:
      mejor_solucion = copy.deepcopy(vecina)   #Con copia profunda. Las copias en python son por referencia
      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  39 , la mejor solución encontrada es: [0, 1, 7, 15, 14, 39, 24, 40, 21, 23, 41, 9, 29, 2, 27,
Distancia     : 1846
```

1273

# Metaheurísticas de búsquedas basadas en trayectorias

## Búsqueda local. Desventajas(intensifica pero no diversifica)

- Escapar de máximos(mínimos) locales. 3 opciones:

  - Modificar la estructura de entornos
    búsqueda en **entornos variables**

  - Permitir movimientos peores respecto a la solución actual
    búsqueda tabú, **recocido simulado**

  - Volver a comenzar con otras soluciones iniciales
    búsquedas **multi-arranque**

Propuesta de mejora para mejorar nota

Revisión

# Metaheurísticas: Recocido simulado - SA

## Esquema básico

$T \leftarrow T_0$

Temperatura inicial alta

Generar una solución inicial $x_1$ en $X$;

$F^* \leftarrow F(x_1)$

$x^* \leftarrow x_1$

**While** la condición de parada no se satisfaga **do**

Criterio de parada:
T=0
ó
n.º de iteraciones

Generar aleatoriamente un $x$ en el entorno $V(x_n)$ de $x_n$

**if** $F(x) \leq F(x_n)$, **then** $x_{n+1} \leftarrow x$

Criterio de aceptación se solución actual

**if** $F(x) \leq F^*$, **then** $F^* \leftarrow F(x)$ y $x^* \leftarrow x$

**else**, generamos un número $p$ aleatorio entre [0,1]

**end if**

**if** $p \leq p(n)$ **then** $x_{n+1} \leftarrow x$

También se acepta con probabilidad p(n)

**end if**

Se desminuye la temperatura según el programa de enfriamiento

**end do**

Revisión

# Metaheurísticas: Recocido simulado - SA

## Función de probabilidad p(n) para aceptar soluciones peores

- Depende la temperatura(T) y de de la **diferencia de costes de las soluciones**

$$P_{aceptación} = \exp(-\delta/T)$$

- A mayor temperatura => mayor probabilidad de aceptar peores soluciones
- A menor diferencia de costes => mayor probabilidad de aceptar peores soluciones

**δ = C(s') – C(s)**
**s = solución actual**
**s' = solución vecina**

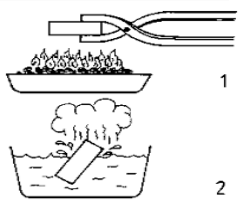
Revisión

# Metaheurísticas: Recocido simulado - SA

## Generar una vecina aleatoria con operador 2-opt

```python
#Generador de 1 solucion vecina 2-opt aleatoria (intercambiar 2 nodos)
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 azar
  return solucion[:i] + [solucion[j]] + solucion[i+1:j] + [solucion[i]] + solucion[j+1:]


genera_vecina_aleatorio(solucion)
```

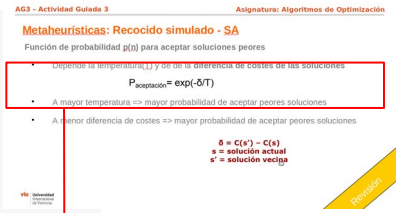
Se eligen aleatoriamente 2 nodos y se intercambian

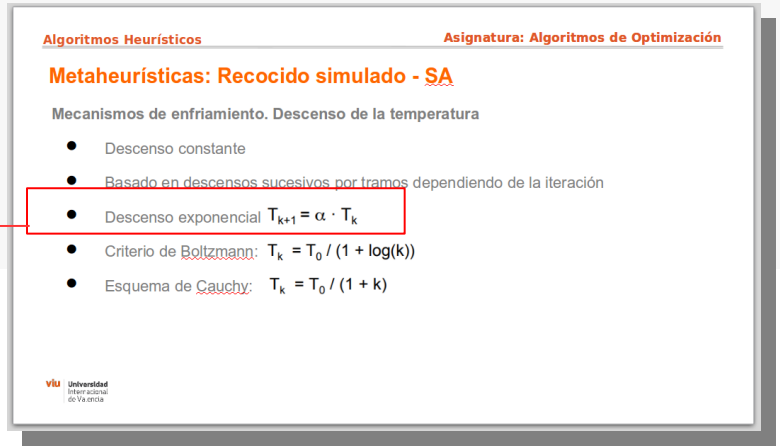


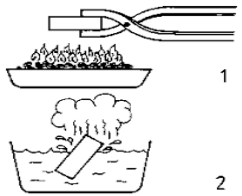
# Metaheurísticas: Recocido simulado - SA

## Funciones para Probabilidad, Temperatura

```
#Funcion de probabilidad para determinar si se cambia
#    a una solución peor respecto a la de referencia(exponencial)
def probabilidad(T,d):
  if random.random() <= math.exp(-1*d / T)    :
    return True
  else:
    return False


def bajar_temperatura(T):
  return T*.999
```



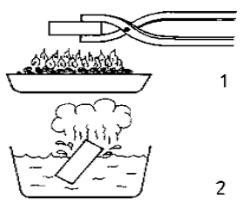



# Metaheurísticas: Recocido simulado - SA

## Recocido simulado(I)

```python
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 = []
  mejor_distancia = 10e100
```

continua



# Metaheurísticas: Recocido simulado - SA

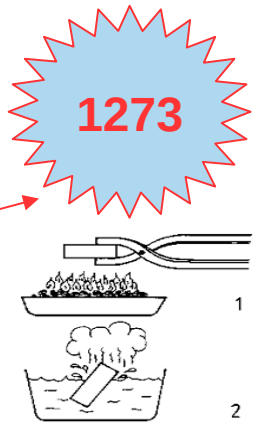
## Recocido simulado(II)

```
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(distancia_referencia - distancia_vecina)
    if distancia_vecina < distancia_referencia or probabilidad(TEMPERATURA, abs(distancia_referencia - distancia_vecina) ) :
      solucion_referencia = copy.deepcopy(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, 17, 15, 14, 6, 1, 27, 3, 4, 5, 26, 11, 12, 18, 13, 19, 16, 36, 35, 31, 37, 7, 20, 33, 34, 
con una distancia total de 1906
```

Criterio de parada

Nos quedamos con una solución peor en algunas ocasiones



# Heurísticas. Práctica individual.

**Tareas opcionales para mejorar nota:**

- **Búsqueda local con Entornos variables**.

  ¿Se puede mejorar con otros operadores de vecindad variables?

- **Recocido simulado**

  ¿Se puede mejorar con otra elección no tan aleatoria( función genera_vecina_aleatorio() )?

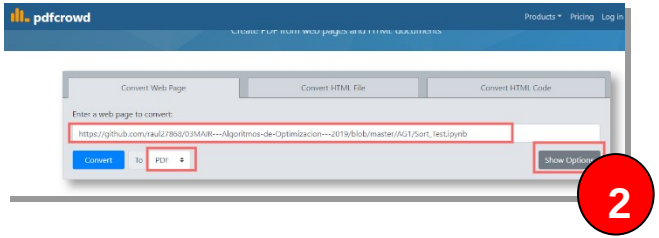
# Finalizar la actividad. Grabar, subir a GitHub, Generar pdf (I)

- Guardar en GitHub

  Repositorio: 03MAIR ---Algoritmos de Optimizacion

  Ruta de Archivo con **AG3**


- Generar pdf con https://pdfcrowd.com




- Descargar pdf y adjuntar el documento generado a la actividad en la plataforma

  - Adjuntar .pdf en la actividad
  - URL GitHub en el texto del mensaje de la actividad

# Heurísticas. Prácticas para compartir en el Foro

- **Búsqueda Tabú** Usar diferentes tipo de listas

- **GRASP**. ¿Se puede mejorar con un algoritmo voraz, aleatorio y adaptativo?

# VC5 – Algoritmos Heurísticos

## 03MAIR – Colonia de Hormigas

viu Universidad Internacional de Valencia

Recuperación VC5

# Metaheurísticas. Métodos constructivos. Colonias de hormigas

**Definición:** Es un proceso **multiagente** de búsqueda en el que cada agente se encarga de **construir probabilísticamente** soluciones aprovechando **información de otros agentes**.
(*Ant Colony Optimization – ACO, Marco Dorigo, 1992* )

Donde:

- **Multiagente** se refiere a la utilización de varios agentes que realizan una determinada tarea de búsqueda basada en instrucciones sencillas.

- **Construcción probabilista** se refiere a que cada agente tomará decisiones según las instrucciones de acuerdo a alguna función de probabilidad.

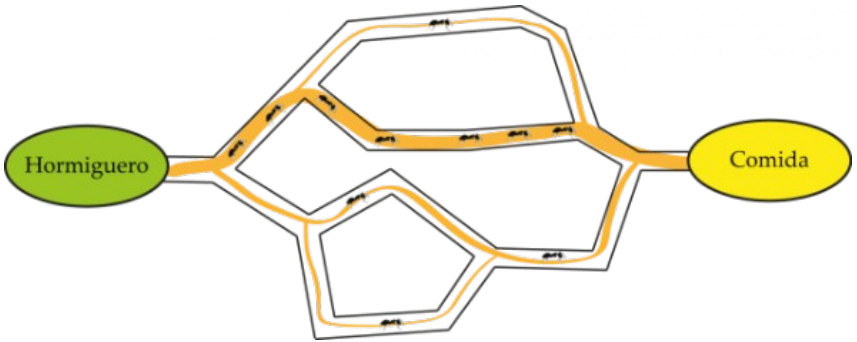
- La información generada por cada agente es compartida con el resto de la comunidad de agentes (feromonas).

Recuperación VC5

# Metaheurísticas. Métodos constructivos. Colonias de hormigas

## Fundamentos

- Se basa en el comportamiento de comunidades de individuos generan comportamientos aparentemente complejos pero basados en reglas sencillas.
- Un rastro de feromonas indica a otros agentes(hormigas) que deben seguir ese rastros.
- Un componente aleatorio para seguir el rastro puede provocar el descubrimiento de nuevas soluciones mejores que se verán reforzadas por que el rastro permanecerá más tiempo que otros.

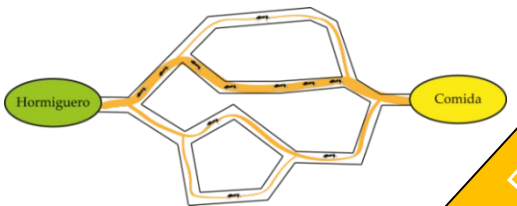



Recuperación VC5

# Metaheurísticas. Métodos constructivos. Colonias de hormigas

## Definición de los rastros de feromonas

- Depositar en cada componente de la solución una cantidad de feromonas proporcional a la calidad de la solución.(En problemas de minimización, inversamente proporcional a valor de la función objetivo)
- Evaporación de feromonas en el tiempo para evitar convergencia a mínimos locales (función exponencial en el tiempo para que sea intensa al principio)
- Incorporación de reglas centralizadas (sin interpretación biológica) que en algunos problemas pueden mejorar el rendimiento (procedimiento demonio). P.Ej. Rastros iniciales de feromonas o rastros extras en algunos nodos o elementos de las soluciones.

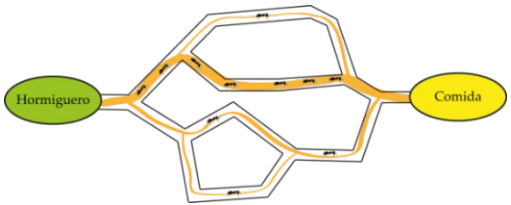



Recuperación VC5

# Metaheurísticas. Métodos constructivos. Colonias de hormigas

## Ventajas sobre otros métodos constructivos

- La componente probabilista permite encontrar gran variedad de soluciones
- Compartir información sirve de guía para encontrar mejores soluciones(aprendizaje reforzado).
- Es posible mejorar la robustez del algoritmo aumentando el número de agentes.
- Permite un buen balance entre intensificación(con el refuerzo de la feromona) y diversificación(con el componente probabilístico y la evaporación)




Recuperación VC5

# Metaheurísticas. Métodos constructivos. Colonias de hormigas

## Esquema básico

```
Depositar una cantidad de feromona inicial en todas las aristas
Crear m hormigas
Repetir:
    Reiniciar hormigas
    Cada hormiga Construir solución usando feromonas y coste de aristas
    Cada hormiga: Depositar feromona en aristas de la solución
    Evaporar feromona de todas las aristas
Devolver: la mejor solución
```

Recuperación VC5

# Metaheurísticas. Métodos constructivos. Colonias de hormigas

## Colonia de hormigas - ACO

Ejemplo: TSP (fue el primer problema abordado con esta técnica)

- Cada agente debe guardar el recorrido parcial.
- En cada paso elegir entre las ciudades más cercanas según una probabilidad asociada a la cantidad de feromonas existente(*)
- Depositar(incrementar) una cantidad de feromona sobre la arista utilizada.

*(*) Inicializar con alguna cantidad inicial de feromonas*

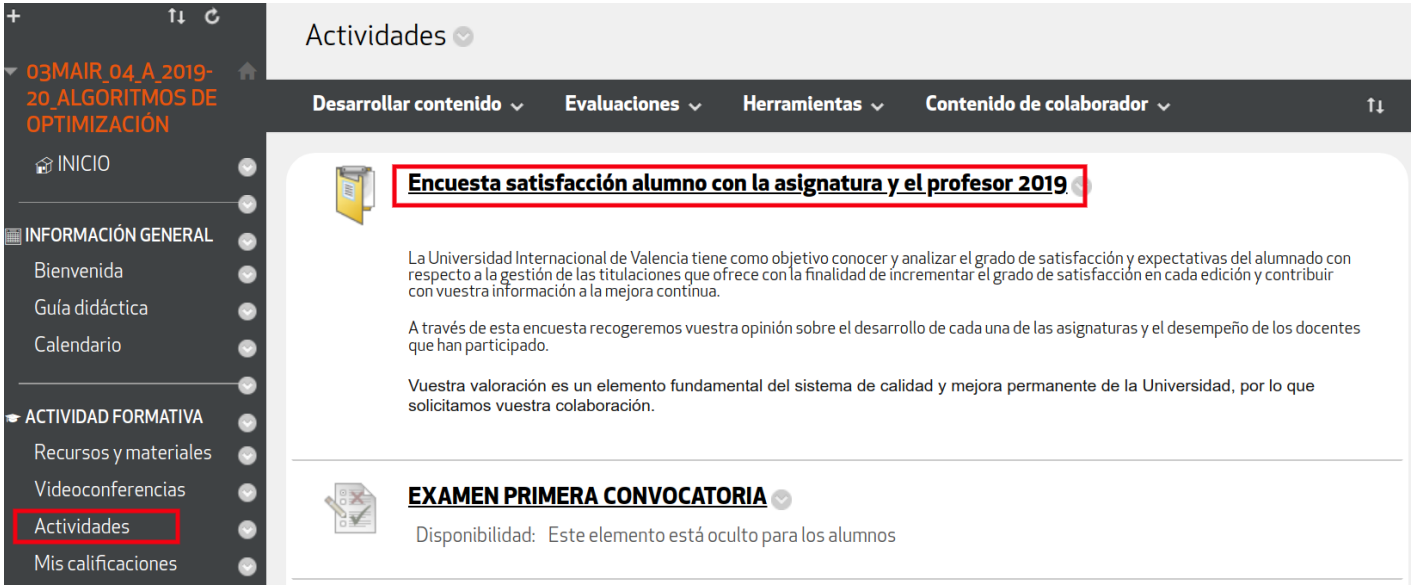
Recuperación VC5

## Próximo día, algoritmos genéticos:

1. Algoritmos evolutivos y genéticos(GA)
2. Introducción práctica a la aplicación de Algoritmos Genéticos(GA) y Colonia de Hormigas(ACO) al problema del agente viajero(TSP)

# Actividad. Encuesta de satisfacción

- Disponible

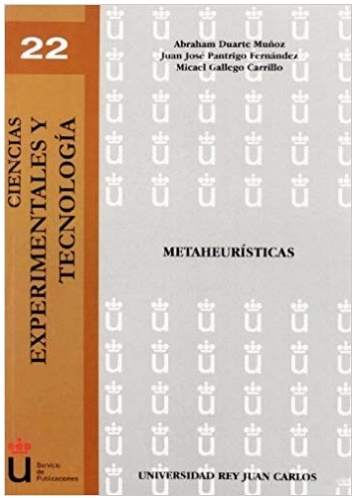


# Ampliación de conocimientos y habilidades

Buscar

- Bibliografía

- Duarte, A. (2008). Metaheurísticas. Madrid: Dykinson.




- Practicar

# Gracias

raul.reyero@campusviu.es

viu Universidad Internacional de Valencia