AG4 – Actividad Guiada 4

03MIAR – Algoritmos de optimización

Abrir el cuaderno de google colab:

https://colab.research.google.com/drive/1ynPeKMTJDJPQ9ctRwCln_LA9Kk1ycXS9

viu Universidad Internacional de Valencia

Christian Velo Abella Hace 14 horas

¿Algoritmo Dijkstra aplicando modulo Heapq?

En la presente contribución en el foro voy a exponer un caso práctico de aplicación del algoritmo Dijkstra y una posible mejora al respecto.

Hace algún tiempo realice un script en el que aplicaba el algoritmo de Dijkstra para localizar el camino más optimo que debía recorrer el cableado eléctrico para unir los diferentes elementos terminales. Empecé a implementar el script sobre un proyecto de una central hidroeléctrica que superaba los 6000 circuitos. El script era capaz de completar el enrutado de gran cantidad de circuitos por ejecución, pero con una velocidad mejorable. Aprovechando lo estudiado en la asignatura he retomado el código y he empezado a indagar en que puntos se podría mejorar y una de las posibles mejoras que he localizado ha sido el sustituir el uso de la función min () sobre un diccionario donde almacenaba los nodos de destinos próximos. No era conocedor del módulo de Python Heapq, pero creo que puede ser interesante para sustituir la función min(diccionario) que tenía implementada en mi código.

*"**Propósito:** heapq implementa un algoritmo de ordenación de montículo adecuado para ser usado con listas de Python.*

*Un montículo es una estructura de datos similar a un árbol en la que los nodos hijos tienen una orden de la relación con los padres. Los montículos binarios pueden ser representados mediante una lista o matriz organizada para que los hijos del elemento N estén en las posiciones 2 * N + 1 y 2 * N + 2 (para índices basados en cero). Este diseño permite reorganizar los montículos en su lugar, por lo que no es necesario reasignar tanta memoria al agregar o eliminando elementos."[0]*

En un principio, aunque sería necesario continuar con el análisis, podríamos pasar de un coste computacional dado en esa línea de código de O(n) en la búsqueda del menor nodo con mi código original a un coste de O(log n). He realizado un pequeño código para analizar los tiempos obtengo lo siguiente:

Diego Rodríguez Atencia

Hace 2 días

Algoritmos de multiplicación de matrices

En la era del **big data**, la necesidad de computación a alta capacidad para procesar cantidades hercúleas de datos ha aumentado significativamente. Esto se debe a la creciente demanda de tratamiento de información en aplicaciones como la resolución numérica de ecuaciones diferenciales, la estimación paramétrica de modelos de lenguaje grandes o incluso en operaciones más simples como la regresión lineal. Una de las operaciones más utilizadas en estos contextos es la **multiplicación matricial**.

Esta operación está presente en prácticamente cualquier modelo matemático multidimensional, surgiendo de manera natural en aplicaciones como la regla de la cadena en funciones compuestas multivariable. Dada su importancia, se han desarrollado varios algoritmos para optimizar su ejecución, mejorando la complejidad computacional en tiempo y espacio.

A continuación, se describen algunos de los algoritmos más relevantes para la multiplicación matricial:

---

1. Multiplicación Ingenua

La **multiplicación ingenua** es la implementación más directa y trivial del producto matricial. Utiliza la definición clásica de multiplicación de matrices. Dadas dos matrices A de dimensiones $(m \times k)$ y B de dimensiones $(k \times n)$, el resultado es una matriz C de dimensiones $(m \times n)$, donde cada elemento c_{ij} se calcula mediante la siguiente fórmula:

Malena Pérez Sevilla Hace 16 días

Resolución del problema de Fibonacci: recursión vs iteración

Hola a todos,

Quiero compartir mi propuesta de solución al problema planteado sobre la sucesión de Fibonacci y mis observaciones al respecto. Además, he incluido ejemplos prácticos para ilustrar la diferencia entre los enfoques recursivo e iterativo.

---

3. ¿Algo raro usando recursión?

Al usar recursión, los cálculos redundantes se vuelven problemáticos para valores grandes de n, haciendo que el algoritmo sea ineficiente. Además, la recursión puede causar un desbordamiento de la pila (*stack overflow*) para valores muy altos de n, dependiendo del lenguaje de programación y la implementación del compilador o intérprete.

Comparación práctica

- **Recursivo:** Simple de implementar, pero ineficiente debido a los cálculos repetidos. Para $n = 10$, puede hacer decenas de llamadas recursivas.
- **Iterativo:** Más eficiente y directo. Solo realiza 8 iteraciones para $n = 10$.

Ambos enfoques llegan al mismo resultado ($F(10) = 55$), pero la iteración es más adecuada para valores grandes de n.

¿Qué opinan? ¿Han encontrado otros enfoques interesantes para abordar este problema?

Paula Pérez Morgado Hace 15 días

RE: Resolución del problema de Fibonacci: recursión vs iteración

Hola Malena!

Me ha gustado mucho como has comparado ambos métodos, destacando las ventajes y desventajas de cada uno. Como respuesta a la pregunta que dejas al final de tu entrada, me gustaría añadir algunos puntos complementarios.

Con respecto al **método iterativo**, has mencionado que este enfoque reduce la complejidad temporal a $O(n)$, por lo que es más eficiente y directo en términos de tiempo. Pero además, este enfoque también reduce la complejidad espacial a $O(1)$, ya que solo se utilizan dos variables (a y b) para almacenar los dos últimos números de Fibonacci, sin la necesidad de reservar memoria adicional para los estados intermedios como sucede en la recursividad. Por tanto, el enfoque iterativo no solo es más rápido, sino también más eficiente en términos de memoria.

Por otro lado, propongo un nuevo método que he encontrado en GeeksforGeeks [1], que es más eficiente en términos de tiempo (lo que más nos suele preocupar). Se trata de la **Exponenciación Matricial:**

Los números de Fibonacci pueden representarse mediante una matriz especial llamada matriz de transformación, que tiene la siguiente forma:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Esta matriz refleja la relación matemática que genera la secuencia de Fibonacci. Si la elevamos a diferentes potencias a través de multiplicaciones sucesivas, podemos obtener cualquier número de la serie.

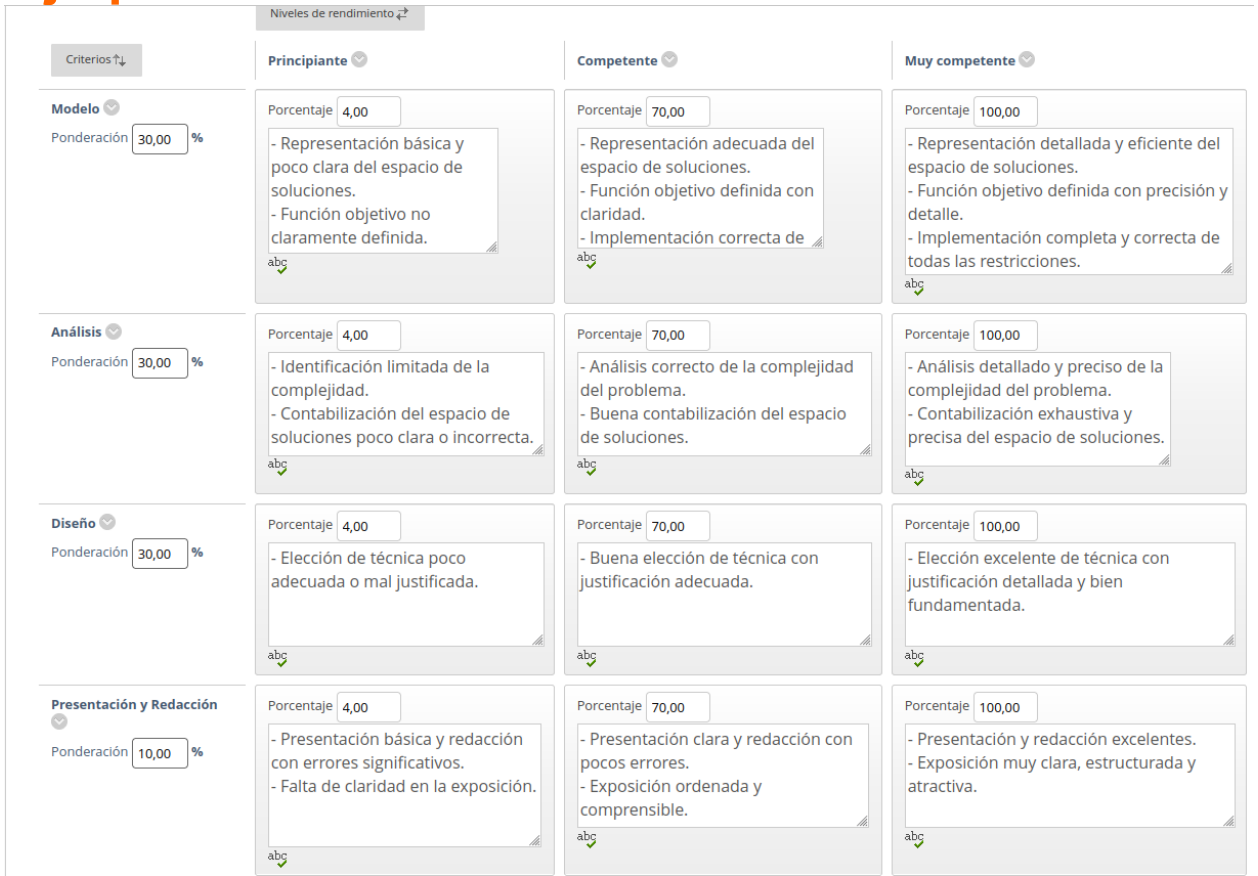
La ecuación matricial para la secuencia de Fibonacci es la siguiente:

$$\begin{bmatrix} F(n) & F(n-1) \\ F(n-1) & F(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$$

De esta forma, para calcular el n-ésimo número de Fibonacci, tenemos que elevar la matriz de transformación a la potencia y quedarnos con el valor que queda en la esquina superior izquierda $F(n)$.

[1] GeeksforGeeks, "Program for nth Fibonacci number," disponible en: https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/#expected-approach1-memoization-approach-on-time-and-on-space. [Accedido: 12-ene-2025].
[2] Aprende Olimpiada Informática, "Algoritmia: Exponenciación rápida," disponible en: https://aprende.olimpiada-informatica.org/algoritmia-exponenciacion-rapida. [Accedido: 12-ene-2025].

Rúbrica del trabajo práctico



Niveles de rendimiento

Criterios	Principiante	Competente	Muy competente
Modelo Ponderación 30,00 %	Porcentaje 4,00 - Representación básica y poco clara del espacio de soluciones. - Función objetivo no claramente definida.	Porcentaje 70,00 - Representación adecuada del espacio de soluciones. - Función objetivo definida con claridad. - Implementación correcta de	Porcentaje 100,00 - Representación detallada y eficiente del espacio de soluciones. - Función objetivo definida con precisión y detalle. - Implementación completa y correcta de todas las restricciones.
Análisis Ponderación 30,00 %	Porcentaje 4,00 - Identificación limitada de la complejidad. - Contabilización del espacio de soluciones poco clara o incorrecta.	Porcentaje 70,00 - Análisis correcto de la complejidad del problema. - Buena contabilización del espacio de soluciones.	Porcentaje 100,00 - Análisis detallado y preciso de la complejidad del problema. - Contabilización exhaustiva y precisa del espacio de soluciones.
Diseño Ponderación 30,00 %	Porcentaje 4,00 - Elección de técnica poco adecuada o mal justificada.	Porcentaje 70,00 - Buena elección de técnica con justificación adecuada.	Porcentaje 100,00 - Elección excelente de técnica con justificación detallada y bien fundamentada.
Presentación y Redacción Ponderación 10,00 %	Porcentaje 4,00 - Presentación básica y redacción con errores significativos. - Falta de claridad en la exposición.	Porcentaje 70,00 - Presentación clara y redacción con pocos errores. - Exposición ordenada y comprensible.	Porcentaje 100,00 - Presentación y redacción excelentes. - Exposición muy clara, estructurada y atractiva.

Examen(*). Doble franja

- Fecha 1ª convocatoria : 28 de febrero a 12:00 y 20:00
- Fecha 2ª convocatoria : 28 de marzo a 12:00 y 20:00
- Duración: 1 hora
- 20 preguntas tipo test: Acierto: **+.5**. Fallo: **-0.167**

smowl

✓Obligatorio

✓Realizar pasos indicados, cualquier problema contactar con soporte

✓Hacedlo con tiempo

viu Universidad Internacional de Valencia

(*)Es requisito indispensable para superar la asignatura aprobar este apartado.

Agenda

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

AG3 – Actividad Guiada 3

Librería TSPLIB

Agenda

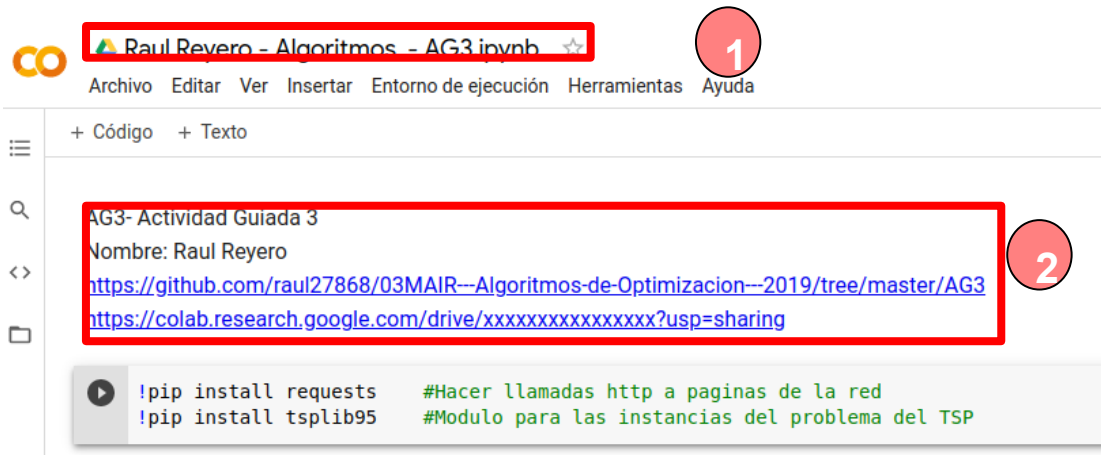
1. Librería TSPLIB para el agente viajero - TSP
2. Resolución por búsqueda aleatoria
3. Resolución por Búsqueda local
4. Resolución por recocido simulado
5. Planteamiento por Colonia de Hormigas - ACO

viu | Universidad Internacional de Valencia

Preparar la actividad en Google Colaboratory.

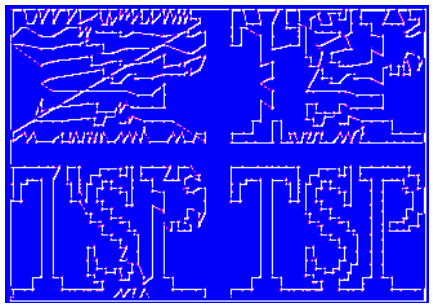
- Copiar con notebook en Google Colaboratory
  https://colab.research.google.com/drive/1ynPeKMTJDJPQ9ctRwCln_LA9Kk1ycXS9?usp=sharing

- Renombra el documento python : **<nombre apellido>-AG4**
- Crear un texto con:
  * AG4- Actividad Guiada 4
  * Nombre Apellidos
  * Url a la carpeta **AG4** 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://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/

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.

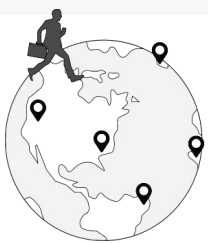
Preparación de los datos

Carga de librerias

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

!pip install requests                       #Hacer llamadas http a paginas de la red
!pip install tabulate>=0.9 networkx>=3.0  # Actualiza las librerías 'tabulate' y 'networkx' a version compatibles con tsplib95
                                          # tabulate: Ayuda a crear tablas de texto legibles para presentar datos.
                                          # networkx:  Sirve para trabajar con grafos y redes, y realizar análisis sobre estas estructuras.
!pip install tsplib95 --no-deps           #Modulo para las instancias del problema del TSP
```

Carga de los datos del problema

```
[2] 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
```



Preparación de los datos

Cargar una instancia del problema: swiss42.tsp

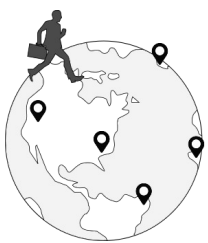
```
import urllib.request #Hacer llamadas http a paginas de la red
import tsplib95       #Modulo para las instancias del problema del TSP
import math           #Modulo de funciones matematicas. Se usa para exp

#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/TSPLIB95/tsp/swiss42.tsp.gz", file + '.gz')
!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-heidelberg.de/software/TSPLIB95/tsp//eil51.tsp.gz", file)

#Coordenadas - 48 capitals of the US (Padberg/Rinaldi)
#file = "att48.tsp" ; urllib.request.urlretrieve("http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp//att48.tsp.gz",
```

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())
```

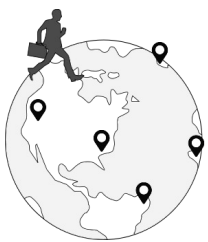
```
NOMBRE: swiss42
TIPO: TSP
COMENTARIO: 42 Staedte Schweiz (Fricker)
DIMENSION: 42
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
0 15 30 23 32 55 33 37 92 114 92 110 96 90 74 76 82 72 78 82 159 122 131 206 112 57 28 43 70
  15 0 34 23 27 40 19 32 93 117 88 100 87 75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

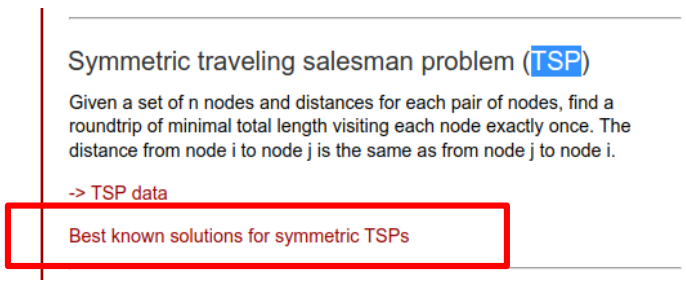
Datos del problema

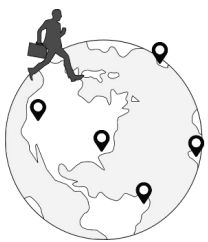
Para n=42 ciudades(nodos) el total de soluciones es:

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

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

1273





http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html

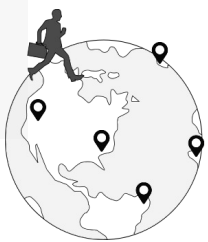
Preparación de los datos

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

Se puede mejorar con **`functools.reduce`** (programación funcional)



AG3 – Actividad Guiada 3

Búsqueda aleatoria

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

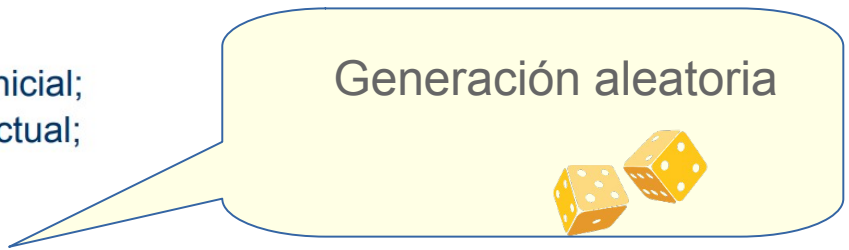
viu Universidad Internacional de Valencia

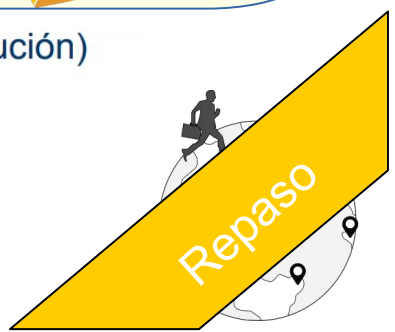
Metaheurísticas de búsquedas

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
```





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
  mejor_distancia = float('inf')                   #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


#Busqueda aleatoria con 5000 iteraciones
solucion = busqueda_aleatoria(problem, 5000)
```

```
Mejor solución: [0, 31, 33, 34, 26, 38, 22, 18, 35, 15, 5, 19, 28, 25, 8, 24, 13, 12, 11, 32, 30, 39, 20, 6, 4, 3, 37, 17
Distancia     : 3495
```

AG3 – Actividad Guiada 3

Búsqueda local

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

viu | Universidad Internacional de Valencia

Metaheurísticas de búsquedas

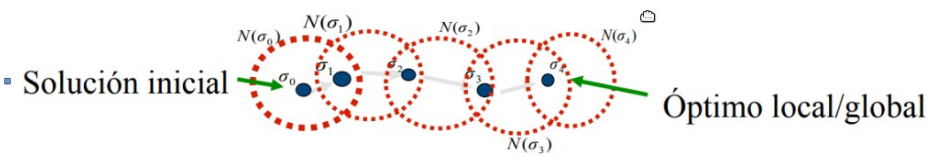
Búsqueda local. Generador de vecindad

Metaheurísticas de búsquedas basadas en trayectorias

Búsqueda local

Elementos básicos:

- ✓ – Establecer **solución inicial**
- ✓ – Establecer una **codificación** para las soluciones
- – Establecer un **operador de generación de vecino**(estructura de entorno)
- – Establecer un **criterio de parada** para finalizar la iteración



Metaheurísticas de búsquedas

Búsqueda local. Generador de vecindad. 1 paso

```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 de vecindad?

- se elige una sub-lista y se invierte el orden
- se elige una sub-lista y se baraja
- ¿se te ocurren otros? (entornos variables)

Metaheurísticas de búsquedas

Búsqueda local. Iteraciones

```
#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 terminar. Hemos llegado a un minimo local(según nuestro operador de vencindad 2-opt)
    if distancia_vecina < mejor_distancia:
      #mejor_solucion = copy.deepcopy(vecina)   #Con copia profunda. Las copias en python son por referencia
      mejor_solucion = vecina                   #Guarda la mejor solución encontrada
      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  42 , la mejor solución encontrada es: [0, 3, 4, 10, 25, 11, 12, 18, 31, 36, 35, 30, 22, 38, 34, 33, 20,
Distancia     : 1749
```

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**

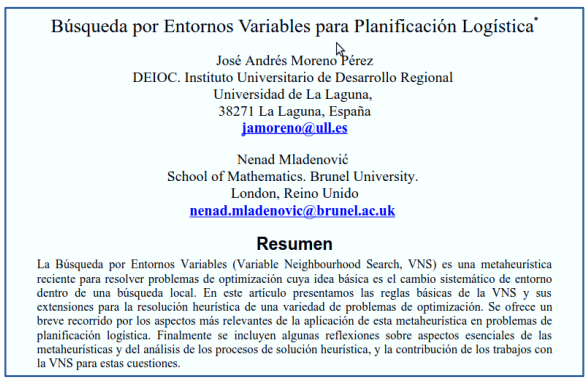
  - Volver a comenzar con otras soluciones iniciales búsquedas **multi-arranque**

Propuesta de mejora para aumentar nota(8/10)

+1 para mejorar

(*)Búsqueda por Entornos Variables para Planificación Logística:

https://jamoreno.webs.ull.es/www/papers/VNS2PL.pdf



Búsqueda por Entornos Variables para Planificación Logística*

José Andrés Moreno Pérez
DEIOC. Instituto Universitario de Desarrollo Regional
Universidad de La Laguna,
38271 La Laguna, España
jamoreno@ull.es

Nenad Mladenović
School of Mathematics. Brunel University.
London, Reino Unido
nenad.mladenovic@brunel.ac.uk

Resumen

La Búsqueda por Entornos Variables (Variable Neighbourhood Search, VNS) es una metaheurística reciente para resolver problemas de optimización cuya idea básica es el cambio sistemático de entorno dentro de una búsqueda local. En este artículo presentamos las reglas básicas de la VNS y sus extensiones para la resolución heurística de una variedad de problemas de optimización. Se ofrece un breve recorrido por los aspectos más relevantes de la aplicación de esta metaheurística en problemas de planificación logística. Finalmente se incluyen algunas reflexiones sobre aspectos esenciales de las metaheurísticas y del análisis de los procesos de solución heurística, y la contribución de los trabajos con la VNS para estas cuestiones.

AG3 – Actividad Guiada 3

Recocido Simulado(Simulated Annealing -SA)

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

viu Universidad Internacional de Valencia

Metaheurísticas: Recocido simulado - SA

Esquema básico

$T \leftarrow T_0$
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**
  Generar aleatoriamente un x en el entorno $V(x_n)$ de x_n
  if $F(x) \leq F(x_n)$, **then** $x_{n+1} \leftarrow x$
    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** $x_{n+1} \leftarrow x$
  end if
  Se desminuye la temperatura según el programa de enfriamiento
end do

Temperatura inicial alta

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

Criterio de aceptación de solución actual

También se acepta con probabilidad p(n) aunque sea peor

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

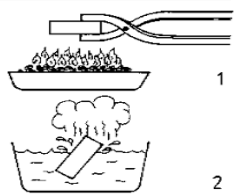
Revisión

Metaheurísticas: Recocido simulado - SA

Generar una vecina aleatoria con operador 2-opt

```
#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)
```

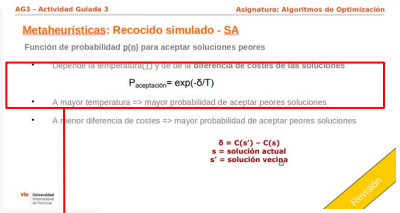
Se eligen aleatoriamente 2 nodos y se intercambian

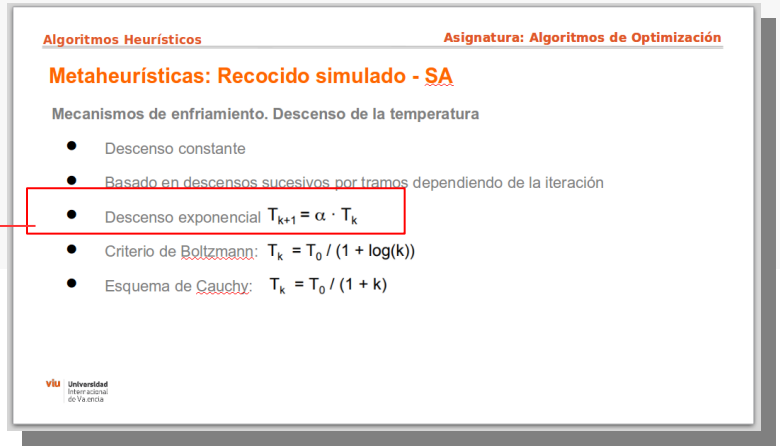


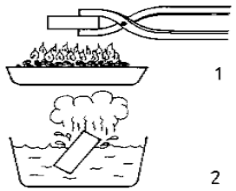
Metaheurísticas: Recocido simulado - SA

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
```





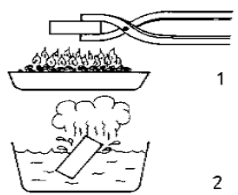


Metaheurísticas: Recocido simulado - SA

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

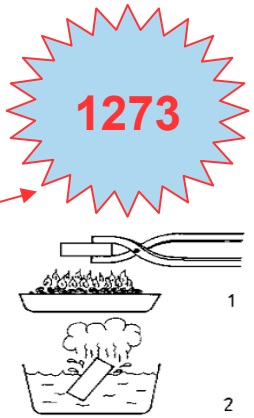
Recocido simulado(II)

```
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. +1 para mejorar

  ¿Se puede mejorar con otros operadores de vecindad variables?

- Recocido simulado +1 para mejorar

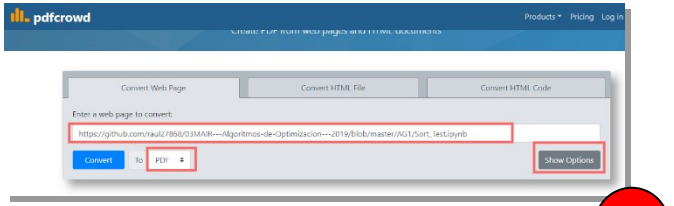
  ¿Se puede mejorar con otra elección no tan aleatoria(función genera_vecina_aleatorio()) ?

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

- Guardar en GitHub

  Repositorio: 03MIAR ---Algoritmos de Optimizacion

  Ruta de Archivo con **AG3**



- 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?

AG3 – Actividad Guiada 3

Colonia de Hormigas - ACO

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

viu | Universidad Internacional de Valencia

Metaheurísticas. Método Constructivo. Colonia de Hormigas

Esquema básico

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

Metaheurísticas. Método Constructivo. Colonia de Hormigas

Esquema básico (I) (1ª iteración)

```
def hormigas(problem, N) :
  #problem = datos del problema
  #N = Número de agentes(hormigas)

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

  #Inicializa las aristas con una cantidad inicial de feromonas:1
  #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)]
```

Propuesta de mejora

Se inicializa todo con ceros

continua

Metaheurísticas. Método Constructivo. Colonia de Hormigas

Esquema básico (II) (1ª iteración)

```
#Recorre cada agente construyendo la solución
for h in range(N) :
  #Para cada agente se construye 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] )
  #print("Feromonas(1)", T)

  #Evapora Feromonas
  T = Evaporar_Feromonas(T)
  #print("Feromonas(2)", 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
```

N=Nº de hormigas

Añade un nuevo nodo

T=Lista de aristas

Metaheurísticas. Método Constructivo. Colonia de Hormigas

Funciones auxiliares

```
def Add_Nodo(problem, H ,T ) :
  #Mejora:Establecer una funcion de probabilidad para
  # añadir un nuevo nodo dependiendo de los nodos mas cercanos y de las feromonas depositadas
  Nodos = list(problem.get_nodes())
  return random.choice(   list(set(range(1,len(Nodos))) - set(H) )  )
```

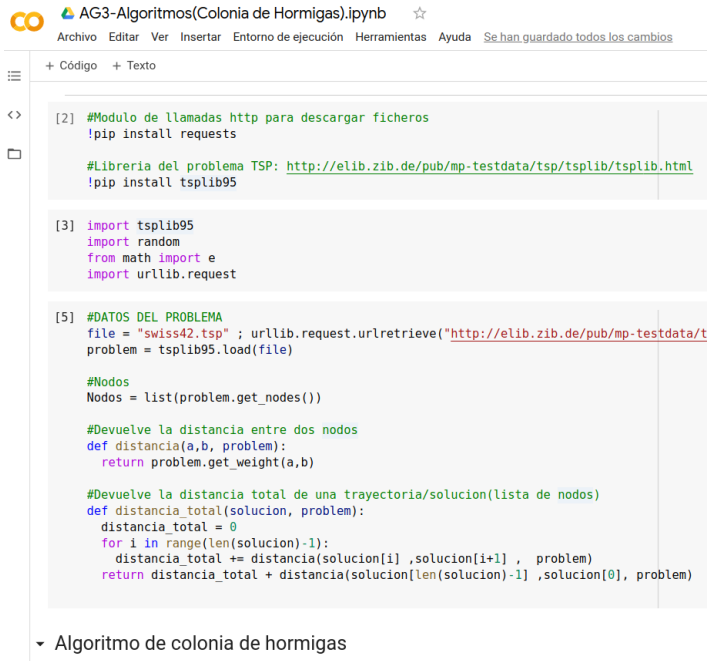
Poco eficiente, demasiado aleatoria

```
def Incrementa_Feromona(problem, T, H ) :
  #Incrementa segun la calidad de la solución. Añadir una cantidad inversamente proporcional a la 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
```

Metaheurísticas. Método Constructivo. Colonia de Hormigas

Trabajar sobre

https://colab.research.google.com/drive/1nX4FZJGCCmh31ps8znXMSxbdgimGW0T2



Metaheurísticas. Método Constructivo. Colonia de Hormigas

Función de probabilidad para elegir el siguiente nodo

Cantidad de feromonas en la arista (i,j)

Inversa de la distancia de i a j

$$p_{ij}^k(t) = \frac{[\tau_{ij}(t)]^\alpha [\nu_{ij}]^\beta}{\sum_{l \in J_i^k} [\tau_{il}(t)]^\alpha [\nu_{il}]^\beta}, \text{ si } j \in J_i^k$$

$$p_{ij}^k(t) = 0, \text{ si } j \notin J_i^k$$ → No se elije si no está en los nodos pendientes

α	Peso relativo que se da a la elección por el rastro
β	Peso relativo que se da a la elección por la distancia entre dos nodos

Si α es igual a 0 , se asemeja a una técnica voraz
Si β es igual a 0 , solo intervine la feromona que puede no ser lo ideal

Metaheurísticas. Método Constructivo. Colonia de Hormigas

Algoritmo ACO aplicado al TSP: Resumen de una experiencia práctica

Benjamín Arenas F., benarenas@hotmail.com
Alejandro Pavez S., apavez@mi.terra.cl
Rodrigo Vidal K., rovika@yahoo.com
Universidad Técnica Federico Santa María
Departamento de Informática

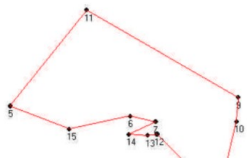
Abstract: La metaheurística Ant Colony Optimization (ACO) [4] es uno de los nuevos paradigmas de resolución de problemas combinatorios del tipo NP. En este artículo se presentan los resultados obtenidos al aplicar un algoritmo variante de la metaheurística ACO en torno al mundialmente conocido Traveling Salesman Problem (TSP) [10], se discuten las oportunidades de desarrollo futuro y se presentan nuevos tuors solución con mejores resultados que los mundialmente encontrados hasta hoy.

Palabras claves: TSP, ACO, Simulated Annealing, Metaheurística

1. Introducción

La importancia del TSP dentro del mundo de problemas NP radica en que es utilizado como conjunto de pruebas para los nuevos algoritmos asociados a la resolución de esta clase de problemas. Su naturaleza NP-Completo hace posible que se puedan resolver una gran familia de problemas equivalentes mediante transformaciones particulares para cada caso [8]. Además, como el TSP presenta una gran cantidad de aplicaciones en el mundo real, se ha convertido en uno de los problemas más estudiados por la comunidad científica mundial. Las características de la nueva metaheurística ACO, aplicadas al TSP [2], [3], [7], nos puede ayudar a encontrar soluciones

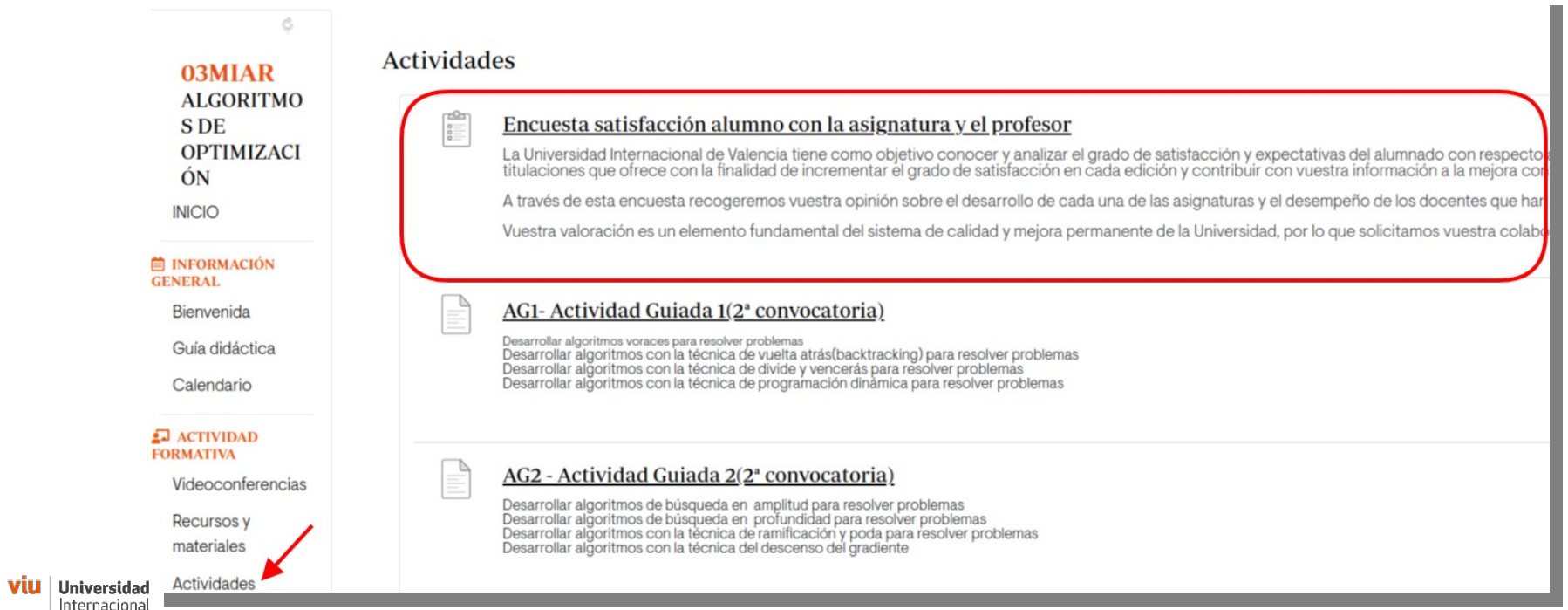
dos veces una misma ciudad. El objetivo es minimizar la longitud de la secuencia de visita de las ciudades, bajo el criterio de minimización (maximización) de la función objetivo. Un ejemplo de un tour es el que se muestra a continuación en el figura 1.



https://www.academia.edu/6676146/Algoritmo_ACO_aplicado_al_TSP_Resumen_de_una_experiencia_pr%C3%A1ctica

Actividad. Encuesta de satisfacción

- Disponible



Ampliación de conocimientos y habilidades

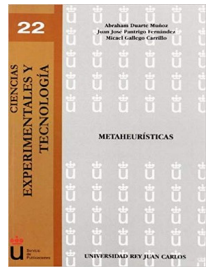
Investigar

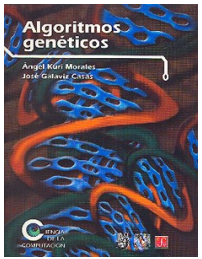
- **Bibliografía**

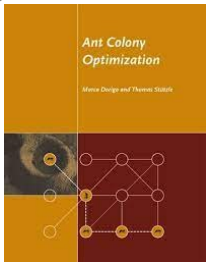
- Duarte, A. (2008). Metaheurísticas. Madrid: Dykinson.
https://elibro-net.universidadviu.idm.oclc.org/es/ereader/universidadviu/35696

- Kuri, Á. (2002). Algoritmos genéticos. Instituto Politécnico Nacional.
https://elibro-net.universidadviu.idm.oclc.org/es/ereader/universidadviu/71925?page=1

- Dorigo, Marco, et al. Ant Colony Optimization, MIT Press, 2004. ProQuest Ebook Central,
https://ebookcentral.proquest.com/lib/universidadviu/detail.action?docID=3338887







Practicar

viu Universidad Internacional de Valencia

Próximo día:

1. Algoritmos genéticos y evolutivos(*)
   - Introducción conceptos evolucionistas
   - Fundamentos de los algoritmos evolutivos y genéticos
   - Características de los algoritmos evolutivos y genéticos
   - Componentes de los algoritmos evolutivos y genéticos

2. Problemas del trabajo práctico(repaso)

3. Artículo científico

(*)Visionar: BIOBOTS y Algoritmos Evolutivos(DotCSV)
https://www.youtube.com/watch?v=ULh2lXR-6O4

¿Preguntas?

Gracias

raul.reyero@professor.universidadviu.com

viu Universidad Internacional de Valencia