

# PROGRAMACIÓN III

## TRABAJO PRÁCTICO

Tecnicatura Universitaria en Inteligencia Artificial

---

## CONTENTS

---

I	CONCEPTOS PREVIOS	2
1	INTRODUCCIÓN	3
1.1	tp-tsp . . . . .	3
1.2	Formulación del problema . . . . .	5
1.3	La clase <code>OptProblem</code> . . . . .	7
1.4	La clase <code>LocalSearch</code> . . . . .	9
2	HILL CLIMBING	10
2.1	Descripción . . . . .	10
2.2	Implementación . . . . .	10
3	RANDOM RESTART HILL CLIMBING	13
3.1	Descripción . . . . .	13
3.2	Implementación . . . . .	13
4	TABU SEARCH	14
4.1	Descripción . . . . .	14
4.2	Implementación . . . . .	14

# Part I

## CONCEPTOS PREVIOS

---

## INTRODUCCIÓN

---

El «**W** Problema del Viajante (TSP)» es un desafío de optimización combinatoria en el campo de las ciencias de la computación y las matemáticas. Consiste en encontrar la ruta más corta posible que un agente debe seguir para visitar un conjunto de ciudades y regresar al punto de partida, recorriendo cada ciudad exactamente una vez.

Formalmente, se define un grafo ponderado y completo, donde cada ciudad es un nodo y cada arista entre dos ciudades tiene un peso que representa la distancia entre ellas. El objetivo es encontrar un ciclo hamiltoniano de longitud mínima en este grafo.


El trabajo práctico consiste en implementar y comparar tres algoritmos de búsqueda local para resolver el TSP. En particular, se abordarán los siguientes algoritmos:

- «**W** Ascensión de colinas» (*hill climbing*).
- «**W** Ascensión de colinas con reinicio aleatorio» (*random restart hill climbing*).
- «**W** Búsqueda tabú» (*tabu search*).

### 1.1 TP-TSP

Antes de empezar a trabajar, necesitamos familiarizarnos con la herramienta que utilizaremos en este trabajo práctico.

Empecemos por clonar el repositorio:

```
 Bash  
git clone https://github.com/maurolucci/tuia-prog3.git  
→ --depth=1
```

A continuación debemos instalar los paquetes requeridos. Para ello ingresamos al repositorio y ejecutamos el siguiente comando:

 Bash

```
pip install -r requirements.txt
```

Ya estamos listos para empezar a trabajar. Ejecutemos el programa y veamos que pasa:

 Bash

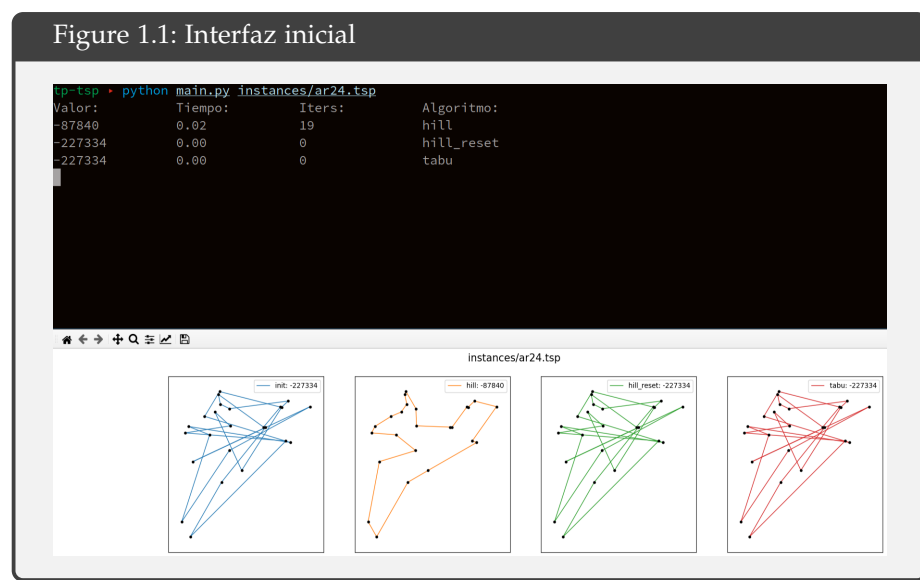
```
python3 main.py instances/ar24.tsp
```

### ! Observación

Para poder ejecutar el programa, necesitamos Python 3.10 o superior.

Si hicimos todo bien, deberíamos ver una pantalla como esta:

Figure 1.1: Interfaz inicial



- En la consola podemos observar estadísticas sobre los tres algoritmos:  
 VALOR Resultado de la función objetivo.  
 TIEMPO Tiempo empleado para resolver el problema.  
 ITES Cantidad de iteraciones realizadas por el algoritmos.  
 ALGORITMO Nombre del algoritmo empleado.
- En la ventana gráfica podemos observar cuatro grafos:  
 INIT Representa el estado inicial del problema.  
 HILL Representa la solución encontrada por el algoritmo de ascensión de colinas.  
 HILL\_RESET Representa la solución encontrada por el algoritmo de ascensión de colinas con reinicio aleatorio.  
 TABU Representa la solución encontrada por el algoritmo de búsqueda tabú.

#### ! Observación

Podemos deducir a partir de la gráfica, que el algoritmo de ascensión de colinas ya está implementado.

## 1.2 FORMULACIÓN DEL PROBLEMA

### 1.2.1 Estados

Consideremos el *TSP* para  $n$  ciudades enumeradas de 0 hasta  $n - 1$  donde la ciudad 0 es el punto de partida, luego nuestros estados serán listas con  $n + 1$  números de la siguiente forma:

$$[0] \oplus \text{permutacion}(1, n - 1) \oplus [0]$$

*TSP son las iniciales de «Traveling Salesman Problem».*

#### ! Observación

La cantidad total de estados es  $(n - 1)!$

### 1.2.2 Estado inicial

Convenimos en utilizar como estado inicial al estado  $[0, 1, 2, 3, \dots, n-1, 0]$ , pero cualquier otro estado también sería válido.

## 1.2.3 Acciones

Consideraremos como acciones posibles al "cruce" de vértices entre dos aristas del tour. A esta familia de acciones se las conoce como «**W** 2-opt».

Podemos representar a cada acción con una tupla de números  $(i, j)$ , donde:

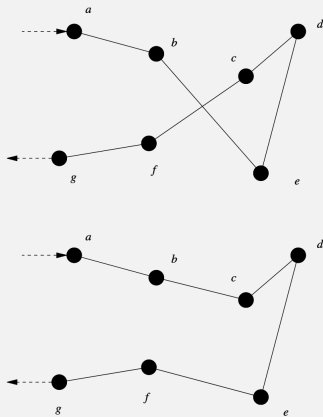
- $i$  representa a la  $i$ -ésima arista; es decir, a la arista que une la  $i$ -ésima ciudad visitada con la siguiente. Llamemos a esta arista  $(b, e)$ .
- $j$  representa a la  $j$ -ésima arista; es decir, a la arista que une la  $j$ -ésima ciudad visitada con la siguiente. Llamemos a esta arista  $(c, f)$ .
- $0 \leq i \leq n - 3, i + 2 \leq j \leq n - 1$

Luego, la acción  $(i, j)$  representa el cruce de vértices entra las aristas de modo que:

- La arista  $i$ -ésima pasa a ser la arista  $(b, c)$ .
- La arista  $j$ -ésima pasa a ser la arista  $(e, f)$ .

En la Figura 1.2: 2-opt se puede ver un ejemplo de cómo se modifica el estado al ejecutar la acción  $(i, j)$ .

Figure 1.2: 2-opt



## ! Observación

Notar que las aristas elegidas no deben ser adyacentes.

## 1.2.4 Resultado

Definimos el resultado de aplicar la acción  $a = (i, j)$  al estado  $s = [v_0, \dots, v_n]$ , donde la  $i$ -ésima arista es  $(v_i, v_{i+1})$  y la  $j$ -ésima arista es  $(v_j, v_{j+1})$ , como sigue:

$$\text{resultado}(a, s) = [v_0, \dots, v_i] \oplus [v_j, \dots, v_{i+1}] \oplus [v_{j+1}, \dots, v_n]$$

## ! Observación

Notar que  $[v_j, \dots, v_{i+1}]$  es el reverso de  $[v_{i+1}, \dots, v_j]'$ .

## 1.2.5 Función objetivo

Para resolver el problema nos proponemos minimizar las distancias, o dicho de otra forma, maximizar el opuesto de las distancias. En definitiva la función objetivo que consideraremos será:

$$\text{obj}([v_0, v_1, \dots, v_{n-1}, v_n]) = -d(v_0, v_1) - \dots - d(v_{n-1}, v_n)$$

donde  $d$  es la función distancia.

## 1.3 LA CLASE OPTPROBLEM

La clase `OptProblem` es la clase que utilizaremos para representar un problema de optimización en general.

`problem.py`

</> Código

```
class OptProblem:
    """Clase que representa un problema de optimizacion
    general."""
```



Nos resultaran de interés los siguientes métodos y atributos:

**RESULT** Nos permite saber a que estado transicionamos, luego de aplicar una determinada acción a un determinado estado.

</> Código

```
def result(self, state: State, action: Action) -> State:
    """Determina el estado resultado de aplicar una
    accion a un estado."""
```

**OBJ\_VAL** A partir de un estado, nos devuelve el valor objetivo.

</> Código

```
def obj_val(self, state: State) -> float:
    """Determina el valor objetivo de un estado."""
```

**VAL\_DIFF** Nos devuelve un diccionario donde para cada acción posible en un determinado estado actual, nos da la diferencia de valor objetivo entre el estado sucesor y el estado actual.

</> Código

```
def val_diff(self, state: State) -> dict[Action, float]:
    """Determina la diferencia de valor objetivo al
    aplicar cada accion.

    El objetivo es que este metodo sea mas eficiente que
    generar cada estado sucesor y calcular su valor
    objetivo."""
```

**RANDOM\_RESTART** Ademas de los métodos vistos, la subclase TSP nos provee un método que devuelve un estado aleatorio.

</> Código

```
def random_reset(self) -> list[int]:
    """Devuelve un estado del TSP con un tour aleatorio.
    """
```

**INIT** También sera indispensable el atributo init de la clase TSP, que almacena el estado inicial.

## 1.4 LA CLASE LOCALSEARCH

Para implementar nuestros algoritmos deberemos heredar de la clase `LocalSearchsearch.py` y reemplazar el método `solve`. Al finalizar nuestro algoritmo, los siguientes atributos deberán almacenar la información correspondiente:

NITERS Numero de iteraciones totales.

TIME Tiempo de ejecución.

TOUR Mejor solución encontrada tras finalizar la ejecución.

VALUE Valor objetivo de la mejor solución encontrada.

---

## HILL CLIMBING

---

### 2.1 DESCRIPCIÓN

El algoritmo de Ascensión de Colinas, también conocido como Hill Climbing en inglés, es un algoritmo de búsqueda local utilizado para resolver problemas de optimización. Su objetivo es encontrar una solución óptima dentro de un espacio de búsqueda mediante la mejora iterativa de soluciones vecinas.

El algoritmo comienza con una solución inicial y evalúa su calidad mediante una función objetivo, que asigna un valor numérico que representa la bondad de la solución. A continuación, se generan soluciones vecinas realizando pequeños cambios o perturbaciones en la solución actual. Estas perturbaciones pueden ser intercambios de ciudades, o de aristas (2-opt) como se propone en este trabajo práctico.

Una vez que se generan las soluciones vecinas, se selecciona la mejor solución de acuerdo con la función objetivo. Si esta solución es mejor que la solución actual, se convierte en la nueva solución actual y se repite el proceso. Este paso se repite hasta que no sea posible encontrar una solución vecina que sea mejor que la solución actual, momento en el cual el algoritmo termina y devuelve la mejor solución encontrada, que puede no ser óptima si la ascensión se atasca en un máximo local.

### 2.2 IMPLEMENTACIÓN

Observemos su implementación para comprender como fue programado.

1. Comenzamos tomando nota de la hora actual, para poder calcular el tiempo empleado en el futuro.

```
</> Código
```

```
# Inicio del reloj  
start = time()
```

2. Comenzamos por el estado inicial y su correspondiente valor objetivo.

</> Código

```
# Arrancamos del estado inicial
actual = problem.init
value = problem.obj_val(problem.init)
```

3. Una vez que terminamos estas tareas de inicialización, estamos dispuestos a comenzar el algoritmo con un bucle `while`.

- (a) Obtenemos el diccionario de acciones y sus respectivos valores objetivos.

</> Código

```
# Determinar las acciones que se pueden aplicar
# y las diferencias en valor objetivo que resultan
diff = problem.val_diff(actual)
```

- (b) Filtramos únicamente aquellas acciones que maximizan el incremento en el valor objetivo.

</> Código

```
# Buscar las acciones que generan el mayor incremento
# de valor obj
max_acts = [act for act, val in diff.items() \
             if val == max(diff.values())]
```

- (c) De todas las acciones seleccionadas en el paso anterior, elegimos una al azar (será nuestro criterio de desempate).

</> Código

```
# Elegir una accion aleatoria
act = choice(max_acts)
```

- (d) Si estamos en un optimo local, ya podemos terminar el algoritmo, completando previamente los valores necesarios.

&lt;/&gt; Código

```
# Retornar si estamos en un optimo local  
# (diferencia de valor objetivo no positiva)  
if diff[act] <= 0:  
    self.tour = actual  
    self.value = value  
    end = time()  
    self.time = end-start  
    return
```

(e) De lo contrario nos movemos al estado sucesor.

&lt;/&gt; Código

```
# Sino, nos movemos al sucesor  
else:  
    actual = problem.result(actual, act)  
    value = value + diff[act]  
    self.niters += 1
```

---

## RANDOM RESTART HILL CLIMBING

---

### 3.1 DESCRIPCIÓN

Random Restart Hill Climbing (Ascensión de Colinas con Reinicio Aleatorio) es una variante del algoritmo de Ascensión de Colinas que busca superar la limitación de quedar atrapado en óptimos locales subóptimos.

En lugar de realizar un solo proceso de ascensión de colinas desde una solución inicial, el Random Restart Hill Climbing realiza múltiples iteraciones del algoritmo, cada una comenzando desde una solución inicial aleatoria. Después de cada iteración, si el algoritmo alcanza un máximo local y no puede mejorar más la solución actual, se reinicia con una nueva solución aleatoria y se repite el proceso.

El objetivo de reiniciar el algoritmo en diferentes puntos aleatorios del espacio de estados es escapar de los máximos locales subóptimos y explorar nuevas áreas en busca de una mejor solución. Cada reinicio brinda la oportunidad de comenzar desde un punto diferente y, potencialmente, encontrar un máximo local mejor o incluso la solución óptima global.

### 3.2 IMPLEMENTACIÓN

Deberemos implementar este algoritmo en la clase `HillClimbingReset`. El método `random_reset` del problema será de vital importancia.

---

## TABU SEARCH

---

### 4.1 DESCRIPCIÓN

Al igual que el algoritmo anterior, la Búsqueda Tabú utiliza una estrategia para evitar quedar atrapado en óptimos locales y explorar de manera más efectiva el espacio de estados en busca de soluciones óptimas.

Utiliza una lista tabú para mantener un registro de los movimientos previamente realizados. Esta lista tabú evita que el algoritmo regrese a soluciones que ya han sido visitadas recientemente, incluso si esas soluciones son mejores que las soluciones actuales.

El uso de la lista tabú permite explorar nuevas soluciones incluso si son peores en términos de la función objetivo, evitando quedar atrapado en ciclos o en óptimos locales subóptimos. La estrategia de caducidad de la lista tabú asegura que los movimientos que han estado en la lista durante mucho tiempo puedan ser considerados nuevamente en el futuro.

### 4.2 IMPLEMENTACIÓN

Deberemos implementar este algoritmo en la clase Tabu. Será indispensable establecer un criterio de parada, elegir un diseño para la lista tabú y experimentar el ajuste de los parámetros involucrados.