

PROGRAMACIÓN III

TRABAJO PRÁCTICO

TRAVELLING SALESMAN PROBLEM

Tecnicatura Universitaria en Inteligencia Artificial

ÍNDICE GENERAL

1	INTRODUCCIÓN	2
1.1	tp-tsp	2
1.2	Formulación del problema	4
1.3	La clase <code>OptProblem</code>	6
1.4	La clase <code>LocalSearch</code>	8
2	HILL CLIMBING	9
2.1	Descripción	9
2.2	Implementación	9
3	RANDOM RESTART HILL CLIMBING	11
3.1	Descripción	11
3.2	Implementación	11
4	TABU SEARCH	12
4.1	Descripción	12
4.2	Implementación	12

INTRODUCCIÓN

El «**W** Problema del Viajante (TSP)» es un problema de Optimización Combinatoria, sumamente importante para las Ciencias de la Computación y las Matemáticas. Consiste en encontrar la ruta más corta posible que un viajante debe seguir para visitar un conjunto de ciudades y regresar al punto de partida, recorriendo cada ciudad exactamente una vez.

Este problema se puede formalizar usando Teoría de Grafos. Primero, se define un grafo ponderado y completo, donde cada ciudad es un nodo y cada arista entre dos ciudades tiene un peso (o costo) que representa la distancia entre ellas. El objetivo es encontrar un ciclo hamiltoniano de peso mínimo 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. Además de especificar el archivo «main.py» a ejecutar, también debemos pasar como argumento una instancia de TSP, que es un archivo que contiene los datos que la definen. Las instancias disponibles se localizan en el directorio «instances». En particular, la instancia «ar24.tsp» contiene las capitales argentinas.

 Bash

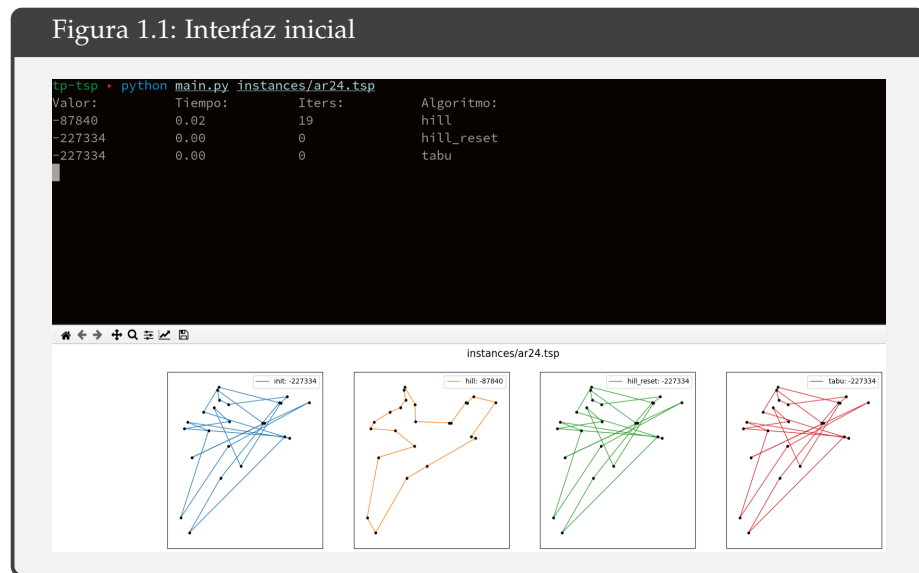
```
python3 main.py instances/ar24.tsp
```

! Observación

Para poder ejecutar el programa, necesitamos Python 3.12 o superior.

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

Figura 1.1: Interfaz inicial



- En la consola podemos observar estadísticas sobre los tres algoritmos:
 - VALOR Valor objetivo de la solución encontrada.
 - TIEMPO Tiempo empleado para resolver el problema.
 - ITERS 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:

TSP son las iniciales de «Traveling Salesman Problem».

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

! 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:

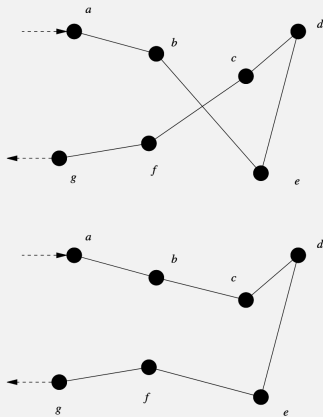
- i representa a la i -ésima arista; es decir, a la arista que une la i -ésima ciudad visitada con la siguiente. Llamemos a esta arista (b, e) .
- j representa a la j -ésima arista; es decir, a la arista que une la j -ésima ciudad visitada con la siguiente. Llamemos a esta arista (c, f) .
- $0 \leq i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

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) .

Figura 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 maximizar:

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

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
```

Esta clase nos provee la siguiente interfaz:

ACTIONS Retorna la lista de acciones aplicables en un estado dado.

```
</> Código  
  
def actions(self, state: State) -> list[Action]
```

RESULT Retorna el estado sucesor, luego de aplicar una determinada acción a un determinado estado.

```
</> Código  
  
def result(self, state: State, action: Action) -> State
```

OBJ_VAL A partir de un estado, nos devuelve su valor objetivo.

```
</> Código  
  
def obj_val(self, state: State) -> float
```

MAX_ACTION Retorna la acción que genera el sucesor con mayor valor objetivo para un estado dado. También retorna el valor objetivo del sucesor.

```
</> Código  
  
def max_action(self, state: State) -> tuple[Action, float]
```

! Observación

Es posible definir este método a partir de los anteriores (por ej. construir un diccionario donde las claves son `self.actions(state)` y el valor asociado a cada clave `action` es `self.obj_val(self.result(state, action))` y luego buscar la clave cuyo valor asociado sea máximo). Sin embargo, dado que este método será llamado en cada iteración de los algoritmos de búsqueda, necesitamos que sea lo más eficiente posible, por lo tanto en su definición no se llamarán a los otros métodos, sino que accederá directamente a la representación del problema para optimizar su funcionamiento.

RANDOM_RESTART Devuelve un estado aleatorio.

```
</> Código  
  
def random_reset(self) -> State
```


INIT También será indispensable el atributo `init`, que almacenará el estado inicial.

! Observación

La subclase `TSP` hereda de `OptProblem` e implementa los atributos y métodos específicos para el TSP, considerando la formulación presentada en la sección [1.2](#).

1.4 LA CLASE LOCALSEARCH

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

search.py

NITERS Numero total de iteraciones.

TIME Tiempo total 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 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, de aristas (2-opt) como se propone en este trabajo práctico, u otras.

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
```

```
start = time()
```

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

</> Código

```
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 la acción que conduce al mejor sucesor y su respectivo valor objetivo.

</> Código

```
act, succ_val = problem.max_action(actual)
```

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

</> Código

```
if succ_val <= value:  
    self.tour = actual  
    self.value = value  
    end = time()  
    self.time = end-start  
    return
```

- c) De lo contrario nos movemos al estado sucesor.

</> Código

```
actual = problem.result(actual, act)  
value = succ_val  
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`. Considerar los siguientes puntos:

- Definir los parámetros necesarios para este algoritmo como atributos de clase en un método `__init__`.
- Al inicio, debe comenzar desde el estado inicial, y a partir de ahí debe llamar al método `random_reset` cada vez que se atasque en un máximo local.
- El número total de iteraciones (`self.niters`) y el tiempo total (`self.time`) deben contabilizar a todos los reinicios.
- Experimentar el ajuste de los parámetros involucrados.

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.

Utiliza una lista tabú para llevar un registro de los estados alcanzados (o acciones aplicadas). Esta lista tabú evita que el algoritmo regrese a soluciones que ya han sido visitadas recientemente.

La búsqueda tabú permite explorar nuevas soluciones incluso si son peores en términos de la función objetivo, evitando quedar atrapado en óptimos locales subóptimos. El tenor de tabú o capacidad limitada de la lista tabú aseguran 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. Considerar los siguientes puntos:

- Elegir parámetros adecuados para este algoritmo y definirlos como atributos de clase en un método `__init__`.
- Elegir una estructura de datos para implementar a la lista tabú. Tenga cuidado que en la formulación propuesta para el TSP, la acción que revierte una acción (i,j) es la misma (i,j) (notar que no existe la acción (j,i) pues la primer componente tiene que ser menor que la segunda).
- Recomendamos agregar un argumento adicional al método `max_action` que sea la lista tabú (con un valor por defecto igual a la lista vacía) y reescribir este método para que la acción retornada no pertenezca a la lista tabú.
- Experimentar el ajuste de los parámetros involucrados.