Práctica 2

- Conjuntos Disjuntos (DS)
- Árboles Abarcadores Mínimos (MST)
- Problema del Viajante

I. TAD CONJUNTO DISJUNTO

I-A. TAD Conjunto Disjunto

Vamos a implementar un conjunto disjunto (CD) S sobre un conjunto universal {0, 1, ..., n-1} con n índices utilizando como estructura de datos un array o bien de padres o bien de rangos negativos según se ha descrito en clase.

1. Escribir una función

init_cd(n: int)-> np.ndarray:

que devuelve un array con valores -1 en las posiciones {0, 1, ..., n-1}.

2. Escribir una función

union(rep_1: int, rep_2: int, p_cd: np.ndarray)-> int:

que devuelve el representante del conjunto obtenido como la unión por rangos de los representados por los índices *rep_1*, *rep_2* en el CD almacenado en el array *p_cd*.

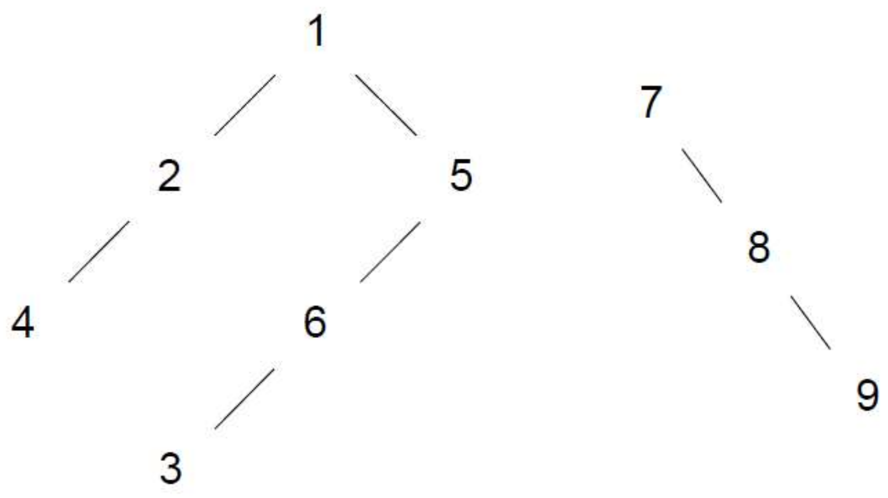
3. Escribir una función

find(ind: int, p_cd: np.ndarray)-> int:

que devuelve el representante del índice *ind* en el CD almacenado en *p_cd* realizando compresión de caminos.

Un Ejemplo

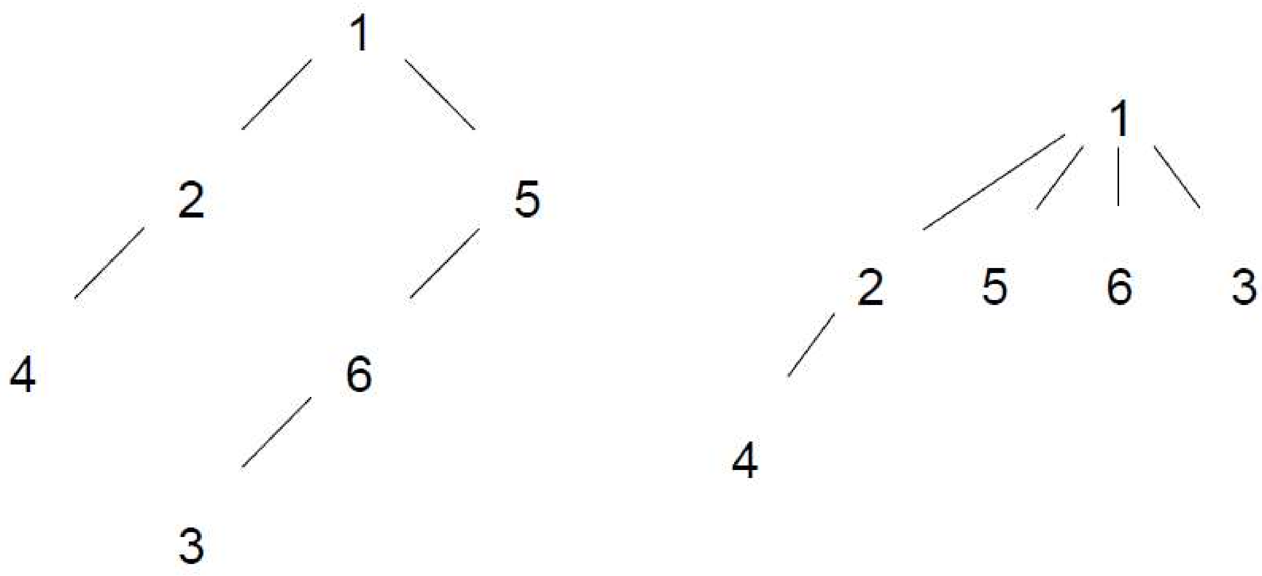
- Para una partición en subcojuntos del conjunto universal [1, 2, 3, 4, 5, 6, 7, 8, 9]



the associated table would be
[-1, 1, 6, 2, 1, 5, -1, 7, 8]

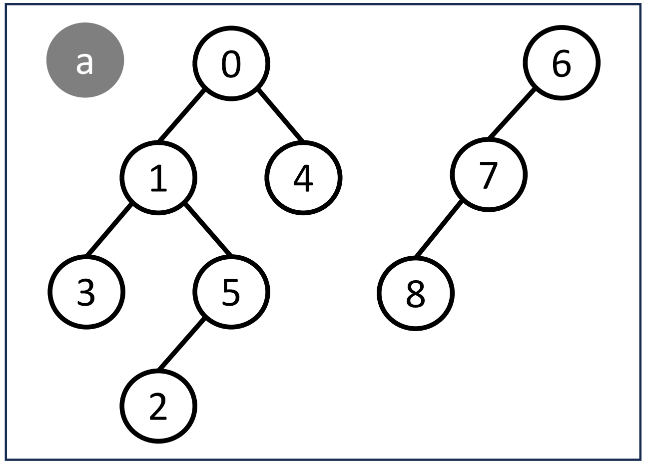
El efecto de la compresión de caminos

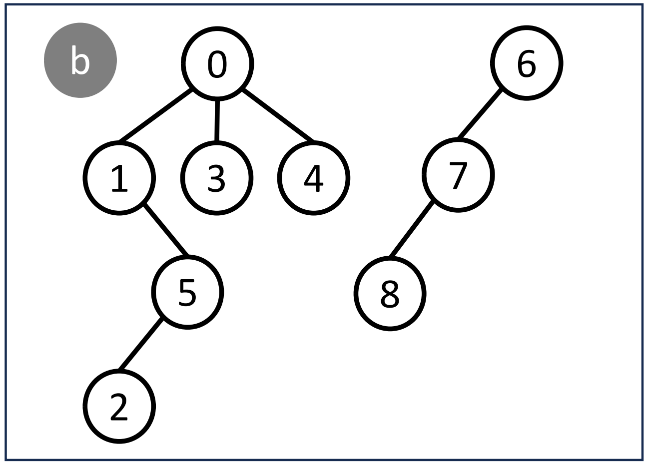
- Izquierda: estado del árbol después de `find(3)`; derecha: estado después de `find_cc(3)`

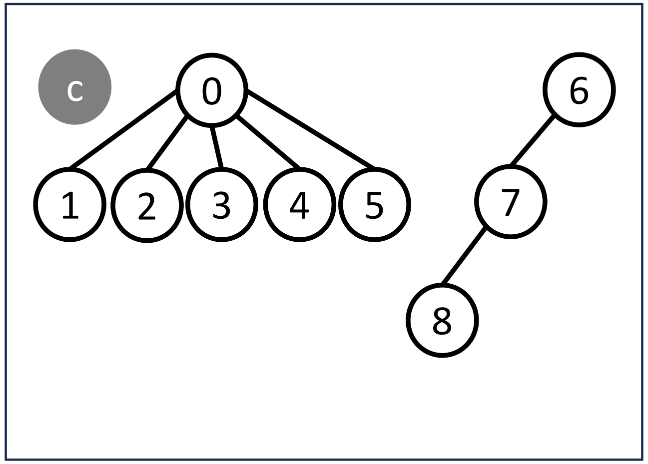


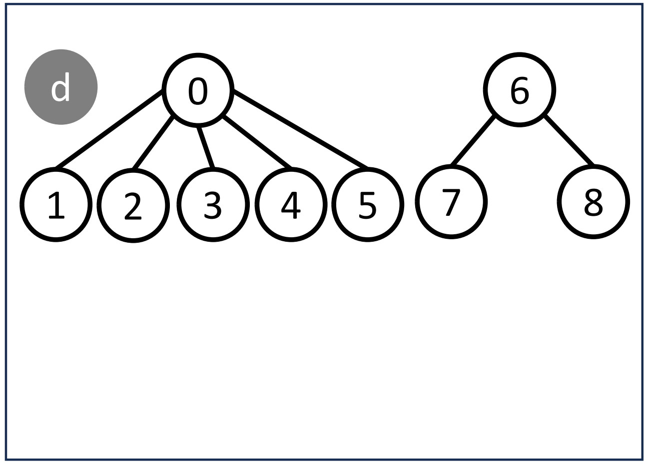
```python
cd = [-3, 0, 5, 1, 0, 4, -2, 6, 7]
print("(a)", cd)
r1 = find(3, cd)
print("(b)", r1, cd)
r2 = find(2, cd)
print("(c)", r2, cd)
r3 = find(8, cd)
print("(d)", r3, cd)
r4 = union(0, 6, cd)
print("(e)", r4, cd)
```

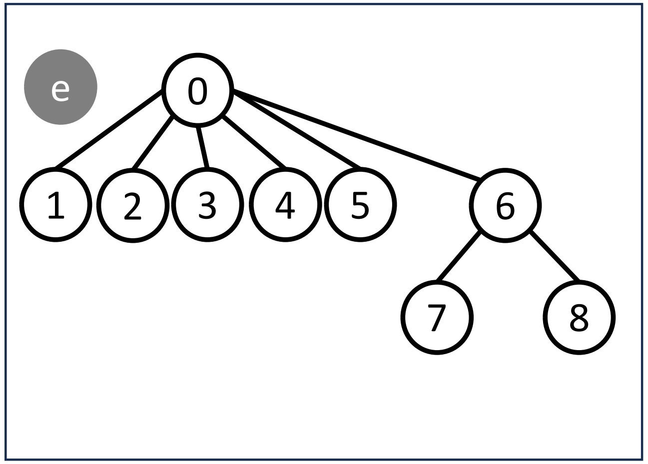
```
(a) [-3, 0, 5, 1, 0, 4, -2, 6, 7]
(b) 0 [-3, 0, 5, 0, 0, 4, -2, 6, 7]
(c) 0 [-3, 0, 0, 0, 0, 0, -2, 6, 7]
(d) 6 [-3, 0, 0, 0, 0, 0, -2, 6, 6]
(e) 0 [-3, 0, 0, 0, 0, 0, 0, 6, 6]
```











II. ÁRBOLES ABARCADORES MÍNIMOS

II-A. Algoritmo de Kruskal

En esta sección vamos a implementar el algoritmo de Kruskal para encontrar un árbol abarcador mínimo de un grafo no dirigido ponderado. Para ello supondremos que en un grafo no dirigido con *n* vértices, estos vienen dados como índices *0, 1, ..., n-1* y que los vértices están dados por una lista *l_g* cuyas ramas se representan como tuplas *(u, v, w)* , donde *u, v* son dos ints que representan la rama y *w* otro int que da el peso de la rama *(u, v)* . Por lo tanto, nuestros grafos vendrán representados como una tupla *(n, l_g)* .

El primer paso en el algoritmo de Kruskal es insertar las distintas ramas de un grafo en una cola de prioridad. Para ello vamos a usar la clase *PriorityQueue* del módulo *queue* , que debemos de importar como

```
import queue
from queue import PriorityQueue
```

En una tal cola *pq* se inserta un elemento *item* con prioridad *w* como *pq.put((w, item))* y se extrae con *w, item = pq.get()*.

1. Escribir una función

```
create_pq(n: int, l_g: List)-> queue.PriorityQueue
```

que inserte las ramas del grafo no dirigido dado por el par *n, l_g* en una cola de prioridad y la devuelva.

2. Completar a continuación el desarrollo del algoritmo de Kruskal escribiendo una función

```
kruskal(n: int, l_g: List)-> Tuple[int, List]
```

que devuelve, si lo hay, un árbol abarcador mínimo (AAM) para el grafo *n, l_g* como un nuevo grafo *n, l_t* , donde *l_t* son las ramas de dicho árbol. La implementación de Kruskal debe vaciar la cola de prioridad antes de volver. Si no hay un tal árbol, debe devolver *None*.

II. ÁRBOLES ABARCADORES MÍNIMOS

II-B. Coste de Kruskal

Vamos a comparar el rendimiento del algoritmo de Kruskal de acuerdo a diferentes variantes en su implementación, trabajando con grafos no dirigidos completos (esto es, donde todos los vértices están conectados entre sí).

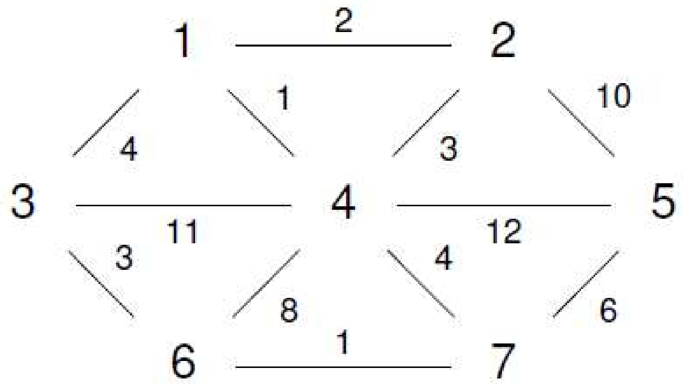
1. Escribir una función
`def complete_graph(n_nodes: int, max_weight=50)-> Tuple[int, List]:`
que genere un grafo completo con *n_nodes* nodos generando ramas *u, v* con *u* < *v* , y pesos *w* obtenidos mediante *random.randint(1, max_weight)* . La función devolverá la tupla *n_nodes, l_g* donde en la lista *l_g* insertamos los elementos *u, v, w* que vamos generando.

2. Escribir una función
`time_kruskal(n_graphs: int, n_nodes_ini: int, n_nodes_fin: int, step: int)-> List`
que genera *n_graphs* grafos completos con pesos aleatorios según la función anterior y devuelve una lista con los tiempos medios de ejecución de nuestra función kruskal correspondientes a cada número de nodos entre *n_nodes_ini, n_nodes_fin* incrementando éste según *step*.

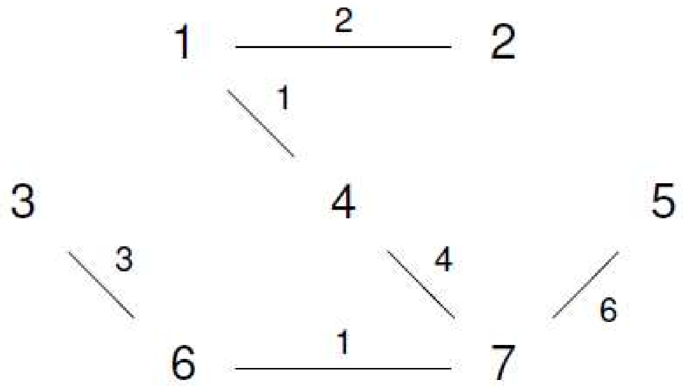
3. El tiempo de ejecución de Kruskal está dominado por la inserción en la cola de prioridad. Modificar la función anterior a una nueva
`time_kruskal_2(n_graphs: int, n_nodes_ini: int, n_nodes_fin: int, step: int)-> List`
para que mida únicamente los tiempos de ejecución asociados a la gestión del CD.
Para ello modificar nuestra función `kruskal` para obtener una función `kruskal_2` que además de un AAM, devuelva también los tiempos de ejecución acumulados sobre las primitivas del conjunto disjunto.

MST Examples

- On the graph

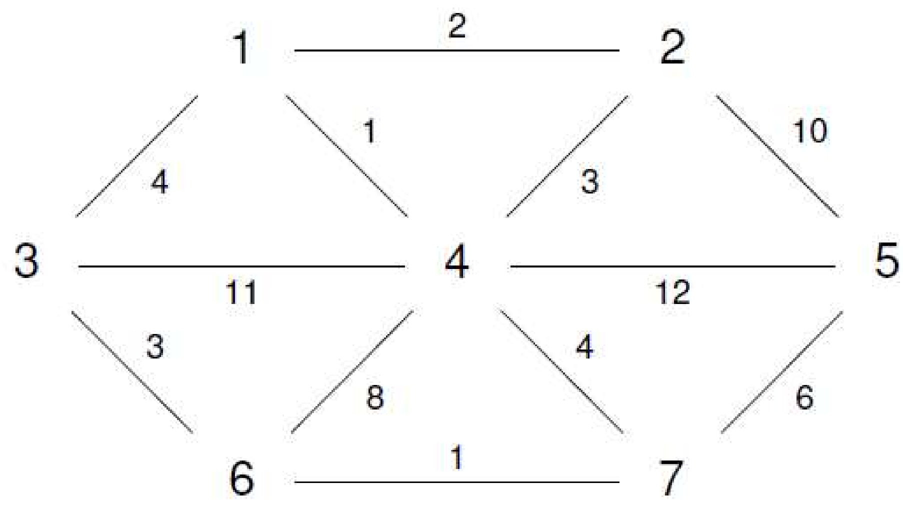


a first MST with cost 17 is



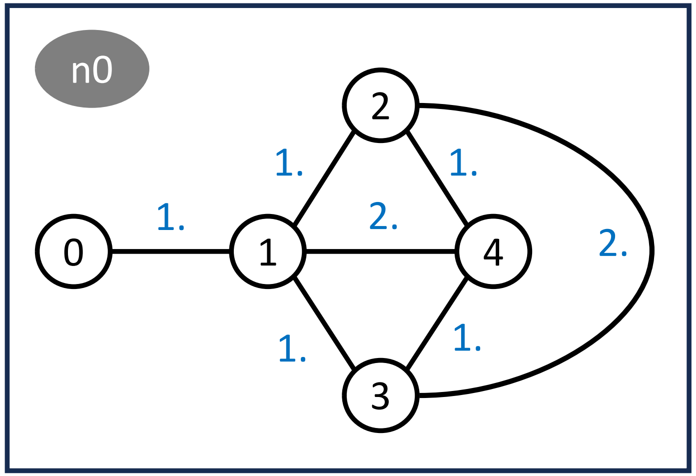
Applying Kruskal's Algorithm

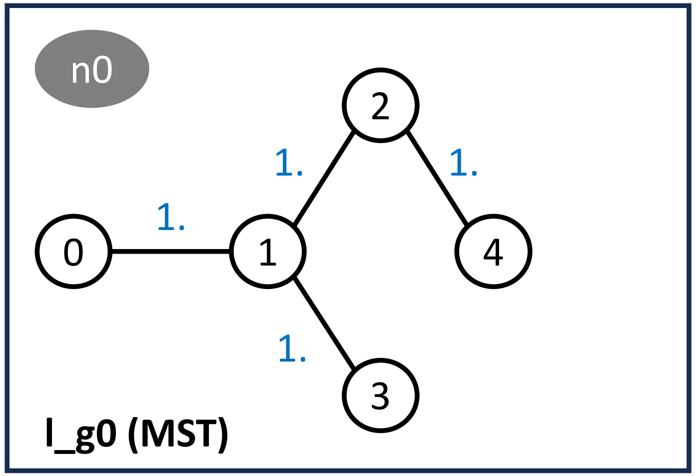
- We apply it on the previous graph



- The PQ is `(1,4), (6,7), (1,2), (2,4), (3,6), (1,3), (4,7), (5,7), (4,6), (2,5), (3,4), (4,5)`

```
n0 = 5
l_g0 = [
(0, 1, 1.),
(1, 0, 1.),
(1, 2, 1.),
(1, 3, 1.),
(1, 4, 2.),
(2, 1, 1.),
(2, 3, 2.),
(2, 4, 1.),
(3, 1, 1.),
(3, 2, 2.),
(3, 4, 1.),
(4, 2, 1.),
(4, 3, 1.),
(4, 1, 2.)]
```

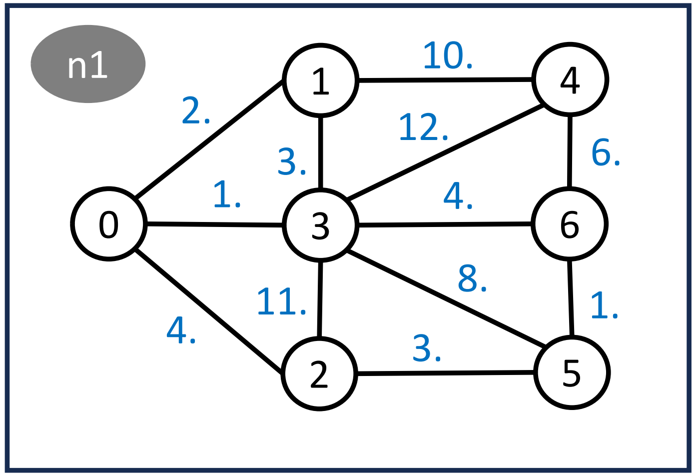


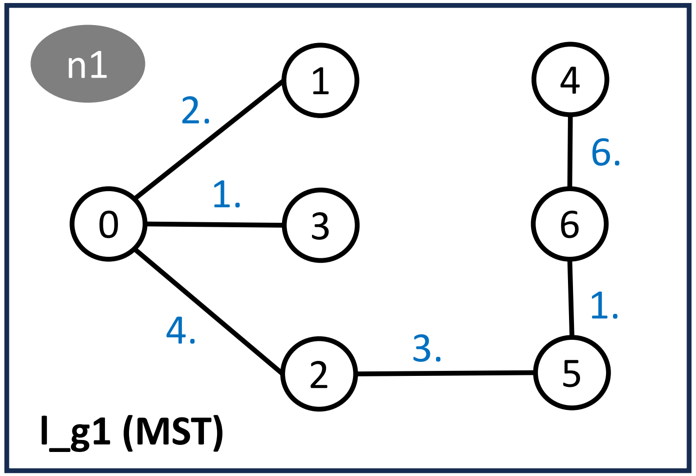



```
print(kruskal(n0, l_g0))
print(kruskal(n1, l_g1))
```

```
(5, [(0, 1, 1.0), (1, 2, 1.0), (1, 3, 1.0), (2, 4, 1.0)])
(7, [(0, 3, 1.0), (5, 6, 1.0), (0, 1, 2.0), (2, 5, 3.0), (0, 2, 4.0), (4, 6, 6.0)])
```

```
n1 = 7
l_g1 = [
(0, 1,  2.),
(0, 2,  4.),
(0, 3,  1.),
(1, 3,  3.),
(1, 4, 10.),
(2, 3, 11.),
(2, 5,  3.),
(3, 4, 12.),
(3, 5,  8.),
(3, 6,  4.),
(4, 6, 6.),
(5, 6, 1.)
]
```





III. EL PROBLEMA DEL VIAJANTE DE COMERCIO

III-A. Algoritmo del Vecino Más Cercano

Vamos a explorar el algoritmo codicioso basado en el vecino más cercano para encontrar un circuito que dé una solución razonable al problema del viajante *(Travelling Salesman Problem, TSP)*. Para ello usaremos la función siguiente

```
def dist_matrix(n_nodes: int, w_max=10) -> np.ndarray:
    """
    """
    m = np.random.randint(1, w_max+1, (n_nodes, n_nodes))
    m = (m + m.T) // 2
    np.fill_diagonal(m, 0)
    return m
```

que genera la matriz de distancias de un grafo con *n_nodes* nodos, valores enteros con un máximo *w_max* ; observar que la función trabaja con *arrays* de *Numpy* y nos devuelve una matriz simétrica con diagonal 0.

III-A. Algoritmo del Vecino Más Cercano

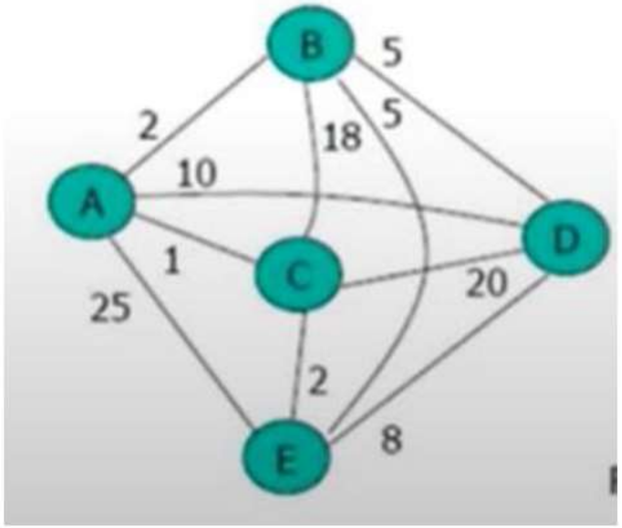
1. Escribir una función
greedy_tsp(dist_m: np.ndarray, node_ini=0)-> List
que reciba una matriz de distancias y un nodo inicial y devuelva un circuito codicioso como una lista con valores entre 0 y el número de nodos menos 1.

2. Escribir una función
len_circuit(circuit: List, dist_m: np.ndarray)-> int
que reciba un circuito y una matriz de distancias y devuelva la longitud de dicho circuito.

3. TSP repetitivo. Una forma sencilla de mejorar nuestro primer algoritmo TSP codicioso es aplicar nuestra función *greedy_tsp* a partir de todos los nodos del grafo y devolver el circuito con la menor longitud. Escribir una función
repeated_greedy_tsp(dist_m: np.ndarray)-> List
que implemente esta idea.

4. TSP exhaustivo. Para grafos pequeños podemos intentar resolver TSP simplemente examinando todos los posibles circuitos y devolviendo aquel con la distancia más corta. Escribir una función
exhaustive_tsp(dist_m: np.ndarray)-> List
que implemente esta idea usando la librería *itertools* . Entre los métodos de iteración implementados en la biblioteca, se encuentra la función *permutations(iterable, r=None)* que devuelve un objeto iterable que proporciona sucesivamente todas las permutaciones de longitud *r* en orden lexicográfico. Aquí *r* es por defecto la longitud del iterable pasado como parámetro, es decir, se generan todas las permutaciones con *len(iterable)* elementos.

- Vecino más cercano:
  - Es una buena aproximación, pero no proporciona la solución óptima.
  - El método consiste en una vez establecido el nodo de partida, evaluar y seleccionar su vecino más cercano.

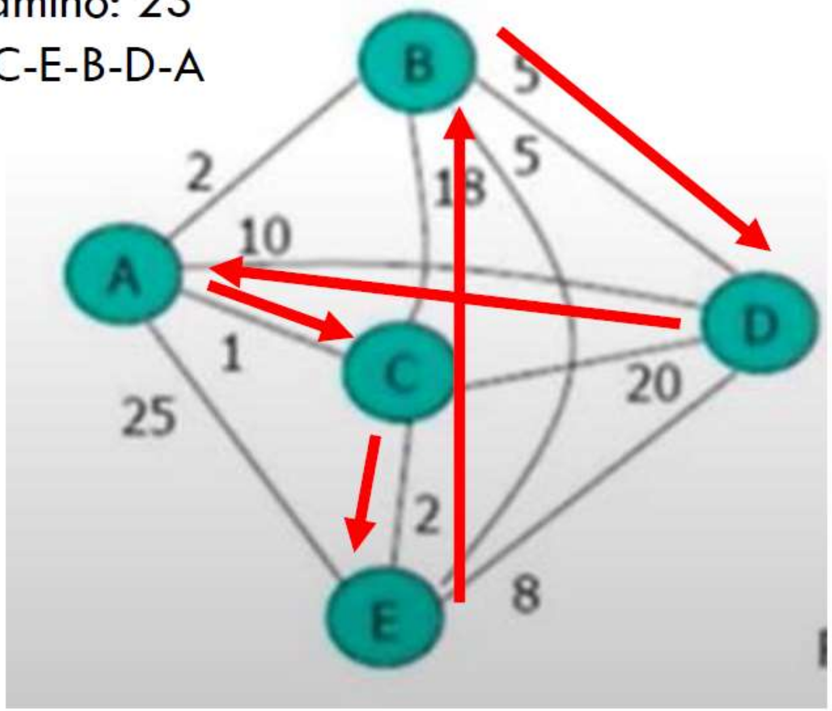


	A	B	C	D	E
A	-	2	1	10	25
B	2	-	18	5	5
C	1	18	-	20	2
D	10	5	20	-	8
E	25	5	2	8	-

Ciudad origen: A

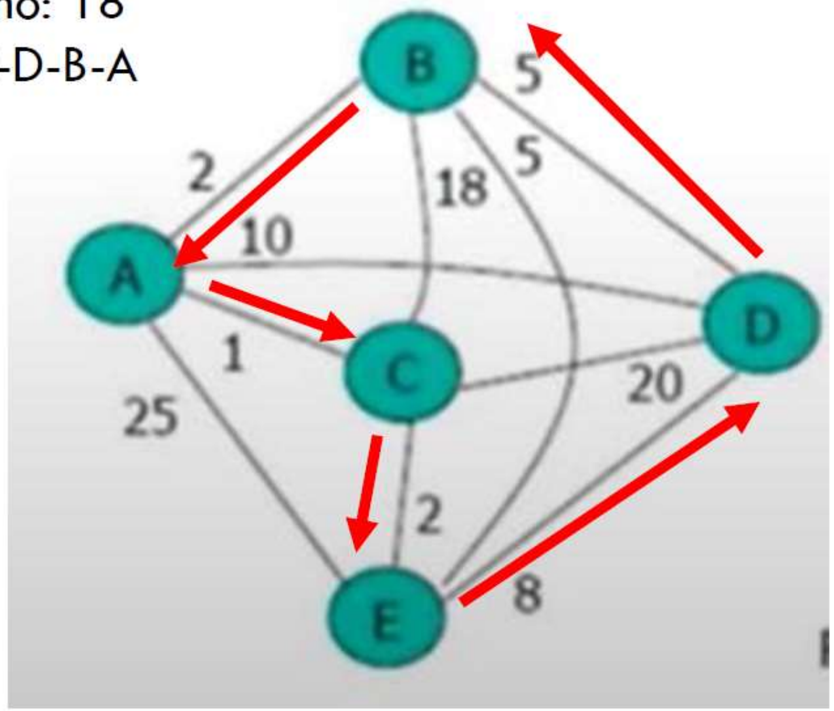
Camino: 23
A-C-E-B-D-A

Camino: 23
A-C-E-B-D-A



VECINO MÁS CERCANO

Camino: 18
A-C-E-D-B-A



SOLUCIÓN ÓPTIMA

- Repetitivo:
  - Aplicar vecino más cercano a todos los nodos del grafo y obtener aquel trayecto con menor longitud.

	A	B	C	D	E
A	0	2	1	10	25
B	2	0	18	5	5
C	1	18	0	20	2
D	10	5	20	0	8
E	25	5	2	8	0

Ciudad origen: A
Camino: 23
A-C-E-B-D-A

Ciudad origen: B
Camino: 23
B-A-C-E-D-A

Ciudad origen: C
Camino: 18
C-A-B-D-E-C

Ciudad origen: D
Camino: 18
D-B-A-C-E-D

Ciudad origen: E
Camino: 18
E-C-A-B-D-E

- Exhaustivo:
  - Examinar todos los posibles circuitos, devolviendo aquel con la distancia más corta.
  - Librería itertools (https://docs.python.org/3/library/itertools.html).
  - Orden lexicográfico: orden de diccionario.
  - Ejemplo: las palabras de cinco letras en el alfabeto A, B.
    - En orden lexicográfico sería: AAAAA, AAAAB, AAABA, AAABB, etc.