Práctica 2: Conjuntos Disjuntos y Componentes Conexas. El Problema del Viajante

Algoritmos y Estructura de Datos Avanzada 2022-2023

Pareja: 11
José Manuel García Giráldez
Alejandro Monterrubio Navarro

Cuestiones

Índice:

Cuestiones sobre CDs y CCs:

Cuestión 1

Sin darnos cuenta, en nuestro algoritmo de encontrar CCs podemos pasar listas con ramas repetidas o donde los vértices coinciden con los de una que ya está aunque en orden inverso. ¿Afectará esto al resultado del algoritmo? ¿Por qué?

No, no afectará al resultado, esto se debe a que en nuestro algoritmo se busca el representante de cada conjunto, de no coincidir los representantes se realizará una unión, si un nodo ya está en el conjunto detectará que el representante es igual, si por ejemplo pasamos de argumento [(1,0),(0,1)] veremos que el resultado será {1:[1,0]}.

```
representante de 1 : 1
representante de 0 : 0
representante de 0 : 1
representante de 1 : 1
{1: [1, 0]}
```

Podemos probar con algo más elaborado:

```
g = [(0, 12), (10, 0), (7, 12), (1, 9), (3, 8), (3, 9), (11, 6), (2,
      5), (4, 5), (0, 2)]
```

```
g = [(0, 12), (10, 0), (7, 12), (1, 9), (3, 8), (3, 9), (11, 6), (2,
      5), (4, 5), (0, 2), (12, 7), (1,9)]
```

Hemos repetido (1,9) e invertido (12,7), resultados:

Resultado en ambos casos:

```
{0: [0, 2, 4, 5, 7, 10, 12], 3: [3, 1, 8, 9], 11: [11, 6]}
```

Cuestión 2

Argumentar que nuestro algoritmo de encontrar componentes conexas es correcto, esto es, que a su final en los distintos subconjuntos disjuntos se encuentran los vértices de las distintas componentes del grafo dado.

Podemos comprobar esto modificando varias cosas, primero modificamos css para que nos devuelva el array sin transformar a diccionario:

```
if len(l) <= 0 :
    print("El argumento lista está vacío, se devolverá un dict vacío")
    return {}

table = init_cd(n)
# Recorremos las ramas del grafo
for u,v in l:
    # Hace un find de cada conjunto
    rep_u = find(u,table)
    rep_v = find(v,table)

    # Si los representantes son distintos, hace una unión
    if rep_u != rep_v:
        # Une los conjuntos
        union(rep_u, rep_v, table)

# Lo convierte en un diccionario y lo devuelve
return cd_2_dict(table), table
```

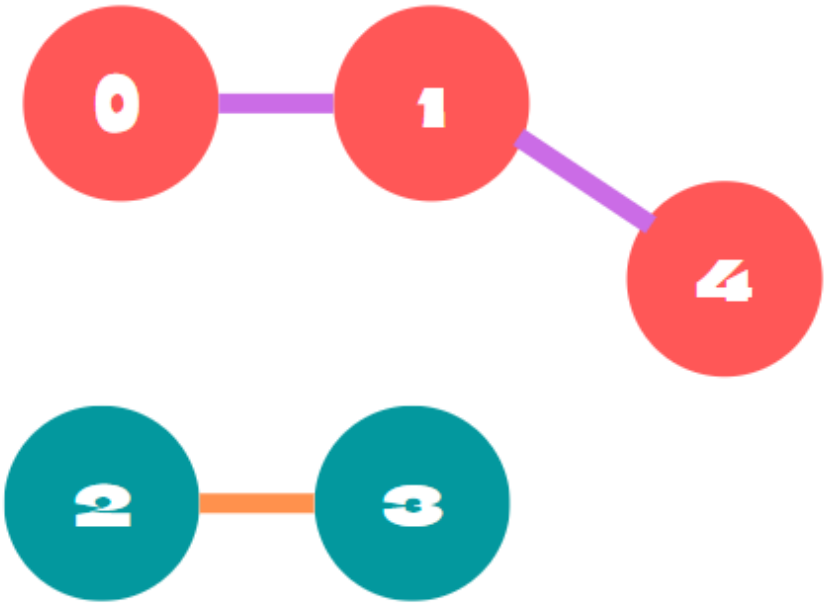
Y creamos código para comprobar los representantes de cada nodo:

```
g = [(0,1), (2,3), (0,4)]
d_cc, ddc = ccs(5, g)
list = []
for x in range(5):
    for y in range(5):
        if find(x,ddc) == find(y, ddc):
            list.append((x,y,True))
        else:
            list.append((x,y,False))
print(d_cc)
print(list)
```

Esto nos mostrará las conexiones existentes:

```
{0: [0, 1, 4], 2: [2, 3]}
[(0, 0, True), (0, 1, True), (0, 2, False), (0, 3, False), (0, 4, True), (1, 0, True), (1, 1, True), (1, 2, False), (1, 3, False), (1, 4, True), (2
, 0, False), (2, 1, False), (2, 2, True), (2, 3, True), (2, 4, False), (3, 0, False), (3, 1, False), (3, 2, True), (3, 3, True), (3, 4, False), (4,
0, True), (4, 1, True), (4, 2, False), (4, 3, False), (4, 4, True)]
```

Como se puede observar 0 está conectado consigo mismo, con 1 y con 4, 1 está conectado con 0, consigo mismo, y con 4, 2 está conectado consgo mismo y con 3 y por último 3 está conectado con 2 y consigo mismo, una representación visual sería:



Cuestión 3

El tamaño de un grafo no dirigido viene determinado por el número n de nodos y la longitud de la lista l de ramas. Estimar razonadamente en función de ambos el coste del algoritmo de encontrar las componentes conexas mediante conjuntos disjuntos.

En nuestro código realizamos L uniones, entonces el coste sería de: $O(L + M \lg^* N)$ teniendo en cuenta que $M = \Omega(N)$ finds con compresión de caminos, en nuestro código son dos finds con compresión, por lo tanto el coste será de $O(L + 2N \lg^* N)$

Cuestiones sobre la solución greedy de TSP

Cuestión 1

Estimar razonadamente en función del número de nodos del grafo el coste codicioso de resolver el TSP. ¿Cuál sería el coste de aplicar la función exhaustive_tsp ? ¿Y el de aplicar la función repeated_greedy_tsp ?

Greedy_tsp:

```
ciudades = dist_m.shape[0]
circuito = [node_ini]
while len(circuito) < ciudades:
    # ciudad actual es la ultima del circuito
    ciudad_act = circuito[-1]

    # se ordenan las distancias para obtener la menor
    distancias = np.argsort(dist_m[ciudad_act])

    # para cada ciudad se comprueba
    for ciudad in distancias:
        if ciudad not in circuito:
            circuito.append(ciudad)
            break
```

Observando la imagen de nuestro algoritmo vemos que recorremos los N nodos, para cada uno de ellos obtenemos la ciudad actual, obtenemos las distancias ordenadas ascendentemente con argsort(coste NlogN) y comprobamos que la ciudad no se encuentre ya en el circuito, recorriendo otra vez N veces. Entonces tenemos N*NlogN que es igual a $O(N^2 logN)$.

Exhaustive_tsp:

```
for circuito in itertools.permutations(range(ciudades)):
    # diccionario con el circuito y su longitud
    circuito = list(circuito)
    circuito.append(circuito[0])
    circuitos[tuple(circuito)] = len_circuit(circuito, dist_m)
```

Se consiguen todas las posibles combinaciones y se coge la de menor longitud, al tomar todas las combinaciones posibles el coste es O(N!).

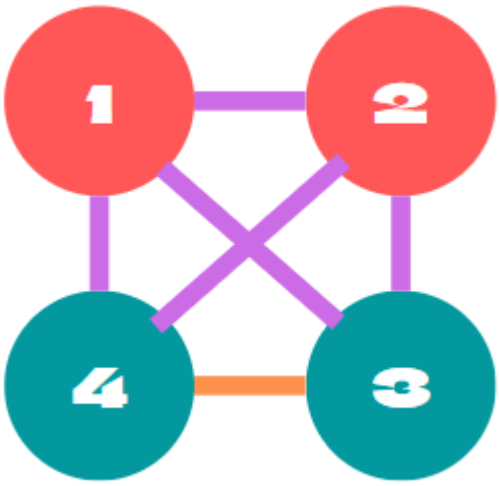
Repeated_greedy_tsp:

Esta función lo que hace es llamar N(por cada nodo) veces a greedy_tsp, por lo tanto el coste se puede ver facilmente que es N*coste greedy_tsp es decir $O(N^3 logN)$

Cuestión 2

A partir del código desarrollado en la práctica, encontrar algún ejemplo de grafo para el que la solución greedy del problema TSP no sea óptima.

No es muy difícil encontrar algún ejemplo que no dé una solución óptima. Por ejemplo podemos forzar que la única opción que le quede sea escoger un camino de gran longitud:



	1	2	3	4
1	0	2	1	2
2	2	0	1	1000
3	1	1	0	2
4	2	1000	2	0

Podemos observar que el circuito elegido si empieza desde 1 sería 1-3-2-4-1 que tendría una longitud de 1004, el camino óptimo sería 1-2-3-4-1 con longitud 7 o 1-4-3-2 con la misma longitud 7,