PRÁCTICA 2 EN PYTHON

13/11/2022

DIMITRIS GIANNAKOS KAISARIS

EDUARDO JUNOY ORTEGA

ALGORITMIA Y ESTRUCTURAS DE DATOS

GRUPO 14
LABORATORIO 02

I-A. TAD Conjunto Disjunto

1. Inicialización de un conjunto disjunto:

```
def init_cd(n: int)-> np.ndarray:
    """  Inicializa un conjunto disjunto

        Argumentos:

            n(int): numero de posiciones del CD

        Devuelve:

            np.full(n, -1)(np.ndarray): un array de n posiciones lleno de -1

        Autor:

            Eduardo

    """
    return np.full(n, -1)
```

2. Unión por rangos de dos representantes:

```
def union(rep_1: int, rep_2: int, p_cd: np.ndarray)-> int:
    """  Une dos conjuntos disjuntos

        Argumentos:

            rep_1(int): representante del primer subconjunto
            rep_2(int): representante del segundo subconjunto
            p_cd(np.ndarray): array del CD

        Devuelve:

            p_cd(np.ndarray): el array del CD con la unión de los subconjuntos

        Autor:

            Eduardo y Dimitris

    """
    p_cd[rep_2-1] = rep_1 #join second tree to first
    return p_cd
```

3. Devuelve el representante:

```
def find(ind: int, p_cd: np.ndarray)-> int:
    """  Busca el representante de un CD
```

```
    Argumentos:

        ind(int): índice del cual se busca su representante
        p_cd(np.ndarray): array del CD

    Devuelve:

        p_c[ind-n+1](int): el elemento el cual representa al índice dado

    Autor:

        Eduardo

    """
    n=1
    while p_cd[ind-n] > 0:
        n=n+1
    return p_cd[ind-n+1]
```

4. Diccionario de claves y elementos del conjunto:

```
def cd_2_dict(p_cd: np.ndarray)-> dict:
    """  Imprime un diccionario del CD

    Argumentos:

        p_cd(np.ndarray): array del CD

    Devuelve:

        p_cd(np.ndarray): el array del CD con la unión de los subconjuntos

    Autor:

        Dimitris

    """
    d = {}
    i = 0
    u = find(i, p_cd)
    j = 0
    k = 0

    for i in range(len(p_cd)):
        if p_cd[i] > 0:
            u = find(i, p_cd)
            d[u] = p_cd[i]
```

```
            k=p_cd[i]
            print (d)
            j=j-1
        elif p_cd[i]<0:
            if j>0:
                k=k+1
                d[u]=k
                print (d)

            if p_cd[i]==-1:
                k=k+1
                d[k]=k
                j=0
                print (d)
                continue

            j=p_cd[i]*(-1)
    if j>0:
        k=k+1
        d[u]=k
    return d
```

I-B. CDs y componentes conexas

1. Función que devuelve las componentes conexas de un grafo:

```
def ccs(n: int, l: List)-> dict:
    """  Devuelve las componentes conexas de un tal grafo.

        Argumentos:

            n(int): número de vértices del grafo
            l(list): la lista que describe las aristas del grafo

        Devuelve:

            graph(dict): las componentes conexas del grafo

        Autor:

            Eduardo

    """
    p = init_cd(n)
    for u, v, in l:
        r_u = find(u, p)
        r_v = find(v, p)
        if r_u != r_v:
            union(r_u, r_v, p)
    d_cc = cd_2_dict(p)
```

```
    return d_cc
```

I-C. Cuestiones sobre CDs y CCs

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 afecta, ya que las componentes conexas resultantes son las mismas en el caso de que la rama se repita o el vértice coincida. Únicamente afecta a la eficiencia del algoritmo, pero no a su resultado.

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.

Es correcto ya que, a parte de comprobarse experimentalmente, la lógica de la función es cierta: para cada iteración hace un find de cada componente y si son distintas las une, devolviendo como resultado final el conjunto disjunto.

3. El tamaño de un grafo no dirigido viene determinado por el número 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 función de l, el coste del algoritmo es l(l-1), ya que busca en cada nodo las posibles conexiones con el resto de nodos.

II-A. Algoritmo del vecino más cercano

1. Función que genera la matriz de distancias de un grafo

```
def dist_matrix(n_nodes: int, w_max=10) -> np.ndarray:
    """  Genera la matriz distrancia de un grafo

        Argumentos:

            n_nodes(int): número de nodos del grafo
            w_max(10): máximo valor del los nodos del grafo

        Devuelve:

            dist_m(np.ndarray): la matriz distancia del grafo dado

        Autor:

            Eduardo

    """

    dist_m = np.random.randint(0, w_max, (n_nodes, n_nodes))
    dist_m = (dist_m + dist_m.T) // 2
```

```
    dist_m = dist_m - np.diag( np.diag(dist_m) )

    return dist_m
```

2. Función que devuelve un circuito codicioso con una lista de valores entre 0 y el número de nodos - 1:

```
def greedy_tsp(dist_m: np.ndarray, node_ini=0) -> List:
    """  Algoritmo codicioso que encuentra el circuito óptimo de un grafo

        Argumentos:

            dist_m(np.ndarray): la matriz distancia del grafo dado
            node_ini(0): nodo inicial del que parte a realizar la búsqueda

        Devuelve:

            circuit + [node_ini]: el circuito más corto desde el nodo inicial

        Autor:

            Eduardo

    """

    num_cities = dist_m.shape[0]
    circuit = [node_ini]

    while len(circuit) < num_cities:
        current_city = circuit[-1]
        options = list(np.argsort(dist_m[current_city]))
        for city in options:
            if city not in circuit:
                circuit.append(city)
                break

    return circuit + [node_ini]
```

3. Función que devuelve la distancia de un circuito:

```
    """  Determina la longitud de un circuito a partir de una matriz distancia

        Argumentos:

            circuit(list): el circuto del cual se quiere saber la distancia
            dist_m(np.ndarray): la matriz distancia del grafo dado

        Devuelve:

            dist(int): la distancia del circuito
```

```
    Autor:

        Eduardo

    """

    dist = 0
    i = 0

    for i in range(len(dist_m)):
        dist += dist_m[circuit[i], circuit[i+1]]

    return dist
```

4. TSP Repetitivo:

```
def repeated_greedy_tsp(dist_m: np.ndarray)-> List:
    """ Algoritmo codicioso que encuentra el circuito óptimo de un grafo
        aplicando la función greedy_tsp a cada nodo del grafo

        Argumentos:

            dist_m(np.ndarray): la matriz distancia del grafo dado

        Devuelve:

            minim(list): el circuito más corto encontrado

        Autor:

            Eduardo

    """
    i=0
    minim = greedy_tsp(dist_m, i)

    for i in range(len(dist_m)):
        if (minim < greedy_tsp(dist_m, i-1)):
            minim = greedy_tsp(dist_m, i)

    return minim
```

5. TSP Exhaustivo:

```
def exhaustive_tsp(dist_m: np.ndarray) -> List:
    """ Algoritmo exhaustivo que que examina todos los posibles circuitos
        y encuentra aquel de distancia más corta
```

```
    Argumentos:

        dist_m(np.ndarray): la matriz distancia del grafo dado

    Devuelve:

        minim(list): el circuito más corto encontrado

    Autor:

        Eduardo

    """

    int_max = 2**128
    len_min = int_max
    for perm in itertools.permutations(range(len(dist_m))):
        new_len = len_circuit(list(perm) + [list(perm)[0]], dist_m)
        if new_len < len_min:
            len_min = new_len

    return list(perm) + [list(perm)[0]]
```

II-B. Cuestiones sobre la solución greedy de TSP

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 ?

Al resolver el TSP el coste codicioso es de n(n^2logN), n(n^3logN) y n*n! por sus características determinadas del código en cada función. En el último ejercicio se puede observar claramente que al realizar las permutaciones da una cantidad de resultados del orden de n! los cuales en la comparación se hace otro bucle y resulta en n*n!. y así sucesivamente con las distintas funciones.

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.

La solución no es óptima para un grafo en el que las distancia entre todos sus nodos es la misma, ya que simplemente sabiendo el número de aristas que separa los nodos sabríamos el resultado de la distancia del circuito con una sencilla multiplicación.

Las cuestiones y este documento han sido realizadas por Eduardo.