Ivan Fernandez Paris
Javier Fraile Iglesias
Pareja 12

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

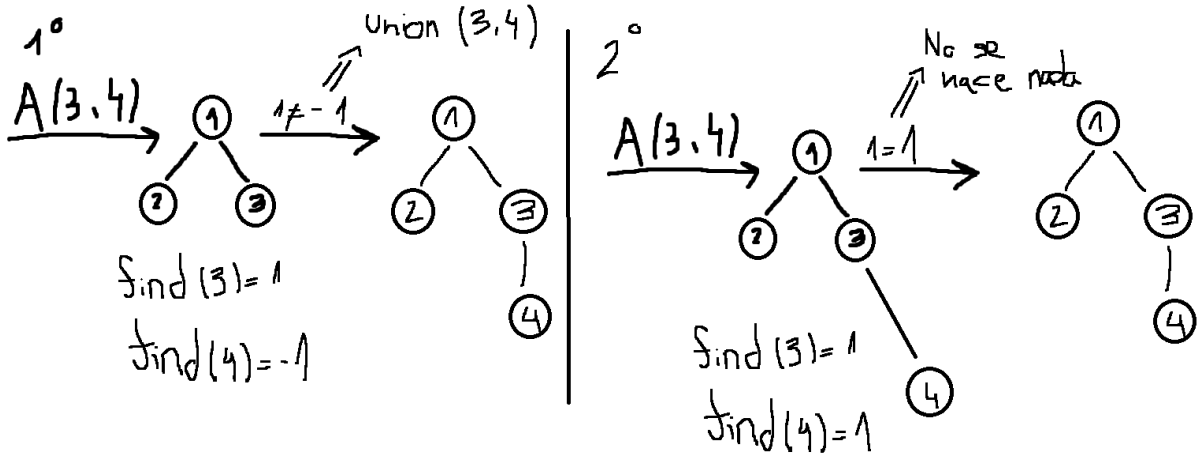
I-C. Cuestiones sobre CDs y CCs

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?**

   Aunque en la lista de ramas se incluyan ramas repetidas o donde los vértices de una rama coinciden con los de otra rama pero en orden inverso, el resultado final del CD no se verá afectado.

   Esto es así porque cuando se examina una rama, por cada uno de los vértices indicados en ella se obtienen sus representantes y en caso de ser iguales no se hace nada (puesto que ya pertenecen al mismo subconjunto), pero en caso de tener diferentes representantes, los subconjuntos a los que pertenece cada vértice se unen.

   Analizando las dos posibles situaciones:
   - Si las ramas están repetidas, la primera vez que se examine una rama se producirá o bien una unión o nada pero para el resto de ramas que se examinan (y que hayan sido examinadas previamente) se descartará la acción a realizar puesto que ambos vértices tendrán el mismo representante. A continuación mostraremos un ejemplo ilustrativo de lo que acabamos de comentar:



- La otra posible situación es que dos ramas se repitan pero el orden de los vértices sea el inverso, según nuestro algoritmo implementado da igual el orden de los vértices, puesto que se hace un find de ambos, entonces el orden sea inverso o no al de otra rama examinada previamente la situación que ocurrirá será la misma que para la otra situación comentada, cuando se examine una rama con un orden inverso de los vértices en comparación con otra rama de la lista, no se producirá una unión de subconjunto porque al hacer un find a cada vértice de la rama el representante de ambos será el mismo y por tanto su examinación terminará

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.**

   Si atendemos a la definición de grafo conexo, un grafo G es conexo si existe un camino entre cualesquiera dos vértices V1 y V2. Esto quiere decir que en el subconjunto resultante deberán de aparecer los vertices V1 y V2 si estos tenían una rama que los uniera, es decir, exista la rama (V1, V2) o (V2, V1) esto es indiferente.

   Nuestra implementación es correcta porque en caso de que algún vértice no tenga ninguna conexión, en el CD tendrá un subconjunto con rango -1 puesto que el si dibujamos el subconjunto como un árbol este solo tendrá la raíz que será dicho vértice y por tanto la profundidad será 1. Pero en caso de que para cada rama si los representantes de los vértices son diferentes unirá sus subconjuntos por lo que en el CD final estarán todos los componentes del grafo.

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.**

   El coste del algoritmo ccs en el peor de los casos es: **O(L*(log N) + L)**

   Cómo hemos llegado a dicha conclusión, bien, el coste de realizar un find() es log N y el coste del union() es 1 como por cada rama se tienen que realizar find() y un union() entonces es cuestión de multiplicar el número de ramas por el coste del find() y el coste del union().

   Destacar que hemos obviado el coste de inicializar el cd.

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***?**

   El coste de **len_circuit()** es **O(N)** puesto que hay que recorrer los N elementos del circuito (lista)

   El coste de **greedy_tsp** es **O(2*N + N*(log N)) = O(N*(log N))**, puesto que por cada nodo se aplica una ordenación sobre su array de distancia (esto supone un coste de O(log N)) y en el peor de los casos recorrer la lista de distancias enteras (esto supone un coste de N)

   El coste de aplicar la función **exhaustive_tsp** es: **O(N*N!)** esto se debe a que debemos crear una permutación de N elementos (lo que pasa que el coste de esta operación no se tiene en cuenta porque es coste es constante), pero por cada elemento de la permutación hay que calcular la distancia, lo que supone recorrer la permutación completa (esto supone un coste de N!) y calcular la distancia de cada permutación (esto supone un coste de N por permutación)

   El coste de aplicar la función **repeated_greedy_tsp** es: N veces el coste de greedy_tsp porque por cada nodo se se obtiene su circuito más corto aplicando greedy_tsp sobre el nodo, más el coste de calcular la distancia de cada circuito con la función len_circuit (que en comparación con el coste de greedy_tsp es despreciable), es por ello que el coste total es **O(N*(N*(log N))) = O(N²*(log N))**

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.**

   Un ejemplo sería el siguiente:

   [[0 17 20 33]
   [17 0 20 2]
   [20 20 0 43]
   [33 2 43 0]]

   Si dicha matriz la usamos en el algoritmo **greedy_tsp** nos devuelve el siguiente circuito [0, 1, 3, 2, 0] cuya longitud es 82 mientras que el algoritmo **exhaustive_tsp** nos devuelve el siguiente circuito [0, 2, 1, 3, 0] cuya longitud es 75.

Porque ocurre esto, el algoritmo **greedy_tsp** obtiene el circuito a partir de la menor distancia entre un nodo y otro si es que el nodo destino no se encuentra ya en el circuito (esto limita que si la distancia a otro es menor pero el nodo destino ya se encuentra en la lista, descarta a ese nodo y comprueba el siguiente), mientras que **exhaustive_tsp** comprueba la distancia de cada permutación (o circuito). Si volvemos al ejemplo:

[0, 1, 3, 2, 0] => 0–17–>1–2–>3–43–>2–33–>0 => 82

[0, 2, 1, 3, 0] => 0–20–>2–20–>1–2–>3–33–>0 => 75

A–X–>B representa que el coste de ir de A a B es X