Estructuras de Datos y Algoritmos – IIC2133

I3

30 de mayo, 2018

1. Algoritmo de Dijkstra sobre un grafo direccional $G = (V, E)$

El algoritmo de Dijkstra puede resumirse así: Primero, colocamos el vértice fuente s en la solución — un árbol de rutas mínimas, T, originalmente vacío. Luego, construimos T de a una arista a la vez, agregando siempre a continuación la arista que da una ruta más corta desde s a un vértice que no está en T; es decir, agregamos vértices a T en el orden de sus distancias (a través de T) a s. Esta es la versión abstracta del algoritmo.

a) Una versión concreta particular del algoritmo es la que estudiamos en clase, y que toma tiempo $O(E \log V)$. Este tiempo es conveniente cuando el número de aristas de G es $O(V)$, o en general, significativamente menor que V^2. Pero si G es muy denso, es decir, si el número de aristas es más bien $O(V^2)$, entonces sería preferible una versión del algoritmo que tome tiempo $O(V^2)$, y que por lo tanto es lineal en el número de aristas de G. Describe una versión del algoritmo con esta propiedad y justifica que es así.

Respuesta: Mantenemos un arreglo, indexado por los números de los vértices, con las distancias de cada vértice a s, todas inicialmente *infinito*, excepto la del propio s (que es 0). Cada vez que agregamos un vértice v a T, actualizamos, via la operación *reduce*, las distancias a s de los vecinos de v — cada actualización toma tiempo $O(1)$ y a lo más hay $|V|-1$ actualizaciones— y luego buscamos el vértice w que quedó más cerca de s —hay que hacer $|V|-1$ comparaciones—; w hará las veces de v en la próxima iteración. Es decir, en cada iteración hacemos $O(V)$ operaciones; y en total hacemos $|V|-1$ iteraciones.

b) Muestra que en general el algoritmo de Dijkstra efectivamente no encuentra (todas) las rutas más cortas a partir de s si G tiene algunas aristas con costos o pesos negativos.

Respuesta: Basta con dar un (contra)ejemplo. Lo primero que hace el algoritmo es mirar a todos los vecinos de s y agregar a T el más cercano de estos. Supongamos que los vecinos son dos, u y v, y que los costos de las aristas son $w(s, u) = 2$ y $w(s, v) = 3$. Entonces el algoritmo agrega u a T, con distancia 2. Sin embargo, si hay una arista (u, v) con costo $w(u, v) = -1$, significa que la distancia de s a v es realmente 1 (= 2 + –1, yendo a través de u) y no 3 (yendo directo), y por lo tanto v está más cerca de s que u y debería haber sido agregado a T antes que u. O bien, si hay una arista (v, u) con costo $w(v, u) = -2$, entonces la distancia de s a u es 1 (= 3 + –2, yendo a través de v) y no 2 (yendo directo).

2. Componentes fuertemente conectadas de un grafo direccional $G = (V, E)$

La nación de Atlanto está compuesta por un grupo de islas pequeñas en el océano atlántico. La gente suele moverse entre las islas usando botes a vela, pero durante el invierno el viento y las corrientes no permiten hacer cualquier ruta. Los habitantes de Atlanto quieren saber a cuáles islas pueden ir durante el invierno de manera de poder volver a sus casas. Haz un programa que reciba como input un grafo direccional en que las islas son los nodos y las aristas son las rutas disponibles en invierno, y que retorne los grupos de islas entre los cuales se puede navegar tranquilamente.

3. MST de un grafo no direccional $G = (V, E)$

El algoritmo de Kruskal se puede resumir así: Primero, ordenamos las aristas, según sus pesos o costos, de menor a mayor, y el árbol T que queremos encontrar está inicialmente vacío. Luego, vamos considerando una a una las aristas en el orden dado, y unimos una arista a T a menos que cierre un ciclo. El algoritmo termina cuando hay $|V|-1$ aristas en T.

a) ¿Cuál es la complejidad de este algoritmo si simplemente usamos DFS para decidir si una arista cierra un ciclo? Deduce paso a paso tu respuesta.

Respuesta: Primero, ordenar las aristas de $G \rightarrow E \log E$; luego, aplicar DFS en un grafo con a lo más V aristas (el número de aristas que finalmente va a tener el árbol T) $\rightarrow V$; como esto último hay que repetirlo en las E iteraciones $\rightarrow EV$; así, en total $\rightarrow E \log E + EV = O(EV)$.

b) Si el algoritmo es implementado usando conjuntos disjuntos, ¿cuál es la relación entre los conjuntos disjuntos —específicamente, cuando los representamos como árboles— y los árboles que constituyen las componentes conectadas de T, en cada paso del algoritmo?

Respuesta: Los árboles que representan los conjuntos disjuntos son las mismas componentes —es decir, tienen los mismos vértices— que los árboles que constituyen las componentes conectadas de T ; solo que las formas de los árboles pueden ser diferentes.

c) [Aparte de a y b] Sea $G(V, E)$ un grafo conectado con costos. Si C es cualquier ciclo de G, entonces demuestra que la arista más costosa (o pesada) de C no puede pertenecer a un MST de G.

Respuesta: Partimos de la propiedad del "corte": dado un corte en G, la arista de menor costo que cruza el corte pertenece a algún mst de G, y todo mst de G contiene una arista de menor costo que cruza algún corte.

4. Conjuntos disjuntos

a) Para cada una de las siguientes implementaciones de conjuntos disjuntos, ¿cuál es la complejidad —peor caso— de ejecutar una única operación *union*? ¿y una única operación *find*? Justifica, preferentemente ayudado por un dibujo.

	union	*find*
arreglo simple	**O(*n*)**	**O(1)**
lista ligada lineal	**O(1)**	**O(*n*)**
árboles con raíz (representados mediante arreglos)	**O(1)**	**O(*n*)**
árboles con raíz, en que el árbol más pequeño apunta al más grande	**O(1)**	**O(log*n*)**

b) Considera el problema de crear un laberinto a partir de un arreglo rectangular de $n \times m$ celdas, en el que la entrada está en la celda de la esquina de arriba a la izquierda y la salida está en la celda de la esquina de abajo a la derecha; las celdas están separadas de sus celdas vecinas por murallas. Inicialmente, están todas las murallas, excepto a la entrada y a la salida. La idea es repetidamente elegir una muralla de manera aleatoria y botarla si las celdas que la muralla separa no están conectadas entre ellas, y así las conectamos (por supuesto, nunca botamos las murallas del perímetro del arreglo). Terminamos cuando las celdas de la entrada y la salida queden conectadas entre ellas (aunque para obtener un "mejor" laberinto conviene seguir botando murallas hasta que cualquier celda sea alcanzable desde cualquier otra celda).

Explica cómo puedes resolver este problema usando una estructura de datos para conjuntos disjuntos.

Respuesta:

Cualquiera de las estructuras de datos para conjuntos disjuntos sirve para resolver el problema; lo importante es entender bien lo que representan los conjuntos y el rol de las operaciones *find* y *union*.

Inicialmente, dado que no hay dos celdas conectadas entre ellas, cada celda está en un conjunto por sí misma.

Entonces, repetidamente, elegimos una muralla de manera aleatoria —representada por las dos celdas adyacentes a la muralla— y ejecutamos dos operaciones *find* para determinar a qué conjunto pertenece cada celda.

Si las celdas pertenecen al mismo conjunto, significa que ya están conectadas entre ellas y por lo tanto no conviene botar esa muralla.

Por el contrario, si las celdas pertenecen a conjuntos distintos, significa que no están conectadas entre ellas, por lo que botamos la muralla, es decir, realizamos una operación *union* entre los conjuntos.

Podemos detener el algoritmo cuando la celda de la entrada y la celda de la salida queden en el mismo conjunto, o, mejor, cuando solo quede un conjunto (todas las celdas están conectadas entre ellas).