Apuntes OIFem 2020

Addendum Grafos

Ordenamiento topológico

Dentro de los grafos unidireccionales (con aristas dirigidas), hay un tipo que se llama "grafo dirigido acíclico" (DAG por sus siglas en inglés), es decir, un grafo donde, empieces donde empieces, no hay forma de volver al nodo inicial si sigues las aristas en su dirección.

Es posible, con grafos de este tipo, elaborar un orden con los nodos de forma que si un nodo i va antes de otro nodo j en ese orden, no hay forma de llegar del nodo j al i siguiendo las aristas del grafo. Un ejemplo clásico de aplicación de este algoritmo que ahora veremos en detalle es la construcción de un orden de asignaturas a partir de parejas de prioridad. Es decir, sabiendo que para cursar la asignatura B hay que cursar antes la A, que para la D hay que cursar antes la S, etc., podemos encontrar un orden en el que tomarlas de forma que cuando estudies una asignatura ya hayas aprobado sus prerrequisitos.

Extra 1: Es importante tener en cuenta que si el grafo tiene ciclos no va a ser posible obtener este orden. ¿Sabrías decir por qué es esto?

Extra 2: ¿Sabrías hacer un programa que encuentra si un grafo dirigido tiene ciclos? Pista: adapta uno de los algoritmos que se encuentra en los apuntes normales de grafos.

Para encontrar el ordenamiento topológico de un DAG, hay dos algoritmos alternativos: uno basado en DFS (por si queréis indagar: https://www.geeksforgeeks.org/topological-sorting/) y el algoritmo de Kahn, que es el que usaremos. El algoritmo de Kahn nos permite, además de obtener un ordenamiento topológico para un DAG, saber si un grafo dirigido tiene ciclos (y frenar en tal caso), además de que el ordenamiento que hace va "por capas". Es decir, al basarse en BFS y no DFS, primero van los nodos cuyo grado (número de aristas que apuntan a ellos) es igual a 0, luego igual a 1, etc. Esto es útil ya que hay problemas (por ejemplo, UVa Beverages, que podéis buscar online y resolver en el juez) que te piden un orden así.

El algoritmo de Kahn funciona de la siguiente manera:

1. Encontramos el grado de cada nodo y creamos una cola a la que añadimos los nodos con grado 0.
2. Vamos sacando nodos de la cola hasta que esta queda vacía. Para cada nodo que sacamos, lo añadimos al orden y lo eliminamos de la cola y del grafo. Esto quiere decir que se elimina el nodo y las aristas que salen de él (no hay aristas que lleguen a él, ya que si no no estaría en la cola). Estas aristas se eliminan simplemente restando 1 al grado de los nodos a los que apuntan estas aristas. Si el grado de alguno de estos vecinos pasa a ser 0, lo añadimos a la cola.
3. Sabremos que hay un ciclo en el grafo si la cola queda vacía y el orden tiene menos nodos del total de nodos que hay en el grafo.

```cpp
vector<int> ordenamientoTopologico(vector<vector<int>> & listaAdyacencia) {
        int N = (int) listaAdyacencia.size();
        // encontramos el grado de cada nodo
        vector<int> grado(N, 0);
        for (int i = 0; i < N; i++) {
                for (int vecino: listaAdyacencia[i])
                        grado[vecino]++;
        }
        // creamos la cola y añadimos los nodos de grado 0
        queue<int> colaNodos;
        for (int i = 0; i < N; i++)
                if (!grado[i])
                        colaNodos.push(i);
        // encontramos el ordenamiento
        vector<int> ordenamiento = {};
        int nodoActual;
        while(!colaNodos.empty()) {
                nodoActual = colaNodos.front();
                colaNodos.pop();
                ordenamiento.push_back(nodoActual);
                for (int vecino: listaAdyacencia[nodoActual]) {
                        grado[vecino]--;
                        if (grado[vecino] == 0)
                                colaNodos.push(vecino);
                }
        }
        if (ordenamiento.size() == N)
                return ordenamiento; // retornamos el ordenamiento encontrado si
                ↪  no hay ciclos
        else return {}; // retornamos el ordenamiento vacío si hay un ciclo
}
```

Aristas y vértices de corte

Una arista de corte en un grafo no dirigido (bidireccional) es una que, en el caso de quitarla del grafo, aumentaría el número de componentes conexas.

Un vértice (nodo) de corte en un grafo no dirigido es uno que, en el caso de removerlo del grafo, aumentaría el número de componentes conexas.

El método para encontrar el número de estos en un grafo (y cuáles son) se basa en DFS. Invocamos una función DFS algo modificada desde una raíz arbitraria.

No hace falta conocer a gran profundidad cómo funciona para las Olimpiadas pero es útil entender el código para poder aplicarlo y la intuición que hay detrás del método.

Lo que nos interesa es ver lo que ocurre en el 'árbol DFS', que es el recorrido que hace DFS para llegar de un nodo a otro en sus invocaciones.

En lo que concierne a nodos, es importante ver que un nodo u es de corte en dos casos:

1. El nodo u es raíz y tiene al menos dos hijos en el árbol de DFS. Cuidado que esto no significa dos vecinos en el grafo, ya que los hijos pueden estar conectados entre sí y llegar de uno a otro, en cuyo caso ese segundo hijo no se descubriría de forma directa a través de u.

2. El nodo u no es raíz y ningún nodo v en un subárbol de u esté conectado a un ancestor de u (es decir, los nodos v más profundos que u y que se los descubra en DFS de la que se baja por el nodo u no pueden estar conectados a nodos w menos profundos que u desde los que se haya descendido hasta u).

Por lo tanto, desde la función de DFS comprobaremos estos dos casos, guardando qué nodos son de corte.

Además, una arista entre los nodos u y v es de corte si no hay forma de llegar a un ancestor de u desde el subárbol de v salvo por esa arista. Comprobamos esto en el DFS. Si u es el padre de v y la arista $u-v$ es de corte, u es un nodo de corte.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

vector<vector<int>> listaAdyacencia;
vector<int> ordenDescubrimiento, bajo; // empiezan como (N, -1)
vector<bool> nodoCorte; // empieza como (N, false)
int contador, N, vertices, aristas, hijos;

void articulacion(int u, int par) {
    ordenDescubrimiento[u] = bajo[u] = contador++;
    for (int v: listaAdyacencia[u]) {
        if (ordenDescubrimiento[v] == -1) {
            articulacion(v, u); // recurrimos
            if (par == -1)
                hijos++; // v es hijo de u, la raíz, en el árbol de DFS
            if (bajo[v] > ordenDescubrimiento[u]) {
                // no hay forma de llegar a ningún ancestor de u desde v salvo por
                ↪ la arista u-v
                nodoCorte[u] = true;
                // u -> nodo de corte
                aristas++;
                // u-v -> arista corte
            }
            else if (bajo[v] == ordenDescubrimiento[u]) {
                // no hay forma de llegar de vuelta a ningún ancestor de u desde
                ↪ el subárbol de v salvo a través de u
                // pero sí a u (y por tanto la arista u-v no es de corte pero u
                ↪ sí)
                nodoCorte[u] = true;
                // u -> nodo de corte
            }
            bajo[u] = min(bajo[u], bajo[v]);
        } else if (v != par)
            bajo[u] = min(bajo[u], ordenDescubrimiento[v]);
    }
}

void corte() {
    // aquí contamos con los valores de N y la lista de adyacencia ya rellenos
    contador = 0;
    aristas = 0;
```

```
42      ordenDescubrimiento.assign(N, -1); // ordenDescubrimiento guarda el orden en
        ↪  el que el DFS encuentra cada nodo
43      bajo.assign(N, -1); // bajo[u] guarda el orden de descubrimiento más pequeño
        ↪  que pertenece a un nodo alcanzable desde el subárbol de u (incluyendo el
        ↪  de u)
44      nodoCorte.assign(N, false);
45      for (int i = 0; i < N; i++) {
46          hijos = 0;
47          if (ordenDescubrimiento[i] == -1) {
48              // aún no hemos explorado esta componente -> tratamos al nodo i como
                ↪  su raíz
49              articulacion(i, -1); // hacemos DFS
50              nodoCorte[i] = hijos > 1; // i será vértice de corte solo en el caso
                ↪  de que tenga más de un hijo en el árbol DFS
51          }
52      }
53      vertices = count(nodoCorte.begin(), nodoCorte.end(), true);
54      cout << vertices << " " << aristas << '\n';
55  }
```