# Apuntes OIFem II Nivel 2

# Grafos Parte 1

Podrás visualizar la mayoría de algoritmos de esta unidad haciendo clic aquí.

## Qué son los grafos y cómo podemos representarlos

Un grafo es un conjunto de nodos conectados por aristas. Un ejemplo sencillo son las amistades. Los nodos son las personas: unidades individuales. Las aristas son los lazos de amistad, es decir, las conexiones entre personas.

Las aristas pueden ser unidireccionales (dirigidos) o bidireccionales (no dirigidos). Una arista unidireccional puede ser un enamoramiento: que Juan se enamore de Ana no implica que Ana se enamore de Juan. Una arista bidireccional puede ser una amistad: si Ana es amiga de Juan, Juan es amigo de Ana.

Además, las aristas pueden tener un coste. Por ejemplo: los nodos representan distintos lugares de encuentro de la ciudad y las aristas son las calles por las que se puede llegar a estos. El coste de una arista puede ser la longitud de la calle que representa o el tiempo que se tarda en recorrerla.

Hay dos formas principales de representar un grafo computacionalmente: la matriz de adyacencia y la lista de adyacencia.

Una matriz de adyacencia es un vector de dos dimensiones que sirve para comprobar en tiempo constante ($O(1)$) si dos nodos $u$ y $v$ están conectados directamente por una arista. Si `matriz[u][v]` $= 1$, entonces $u$ está conectado a $v$. En un grafo bidireccional, `matriz[u][v]` será igual a `matriz[v][u]`. Además, si las aristas tienen un coste, podemos representarlo poniendo el valor del coste de la arista en lugar de ese 1 y poniendo un $-1$ para las parejas no conectadas por una arista, en el caso de saber que $-1$ nunca será un coste. Sin embargo, para ver todos los nodos a los que está conectado un nodo $u$, tendremos que recorrer toda la lista de nodos, con un coste de $O(n)$ en el caso de haber $n$ nodos, estrategia muy poco eficiente para grafos con nodos que igual tienen una o dos aristas.

Una lista de adyacencia es un vector de dos dimensiones que guarda para cada nodo $u$ todos los nodos a los que este está conectado directamente por una arista. En grafos cuyas aristas todas tienen el mismo coste (aristas sin costes/coste 1), `lista[u]` es un vector de una dimensión que contiene los índices de los nodos conectados directamente a $u$. En grafos con aristas con coste, `lista[u]` contiene un vector de parejas (índice, coste) con los nodos conectados directamente con el nodo $u$. La lista de adyacencia es el método recomendado para casi todos los problemas, muchas veces incluso cuando el usuario da la entrada como matriz de adyacencia, ya que no hace falta iterar sobre todos los nodos para ver las conexiones directas del nodo $u$.

# Breadth-first search (BFS)

La búsqueda en anchura (BFS) es una técnica para explorar todos los nodos de un grafo que se aprovecha de una cola para visitar todos los nodos en orden de lejanía de una raíz (arbitraria o no) $x$.

Empezamos con una cola vacía a la que añadimos el nodo $x$. Mientras que la cola no esté vacía, decolamos el primer elemento y comprobamos si sus conexiones ya han sido visitadas. Si este no es el caso, las añadimos a la cola.

En este ejemplo, usamos BFS para imprimir todos los nodos de un grafo.

```
void bfs(int raiz, vector<vector<int>> & listaAdyacencia) {
        queue<int> colaBFS;
        vector<bool> visitado(listaAdyacencia.size(), false);
        colaBFS.push(raiz);
        visitado[raiz] = true;
        int nodoActual;
        while(!colaBFS.empty()) {
                nodoActual = colaBFS.front();
                colaBFS.pop();
                cout << nodoActual << '\n';
                for (int conexion: listaAdyacencia[nodoActual]) {
                        if (!visitado[conexion]) {
                                colaBFS.push(conexion);
                                visitado[conexion] = true;
                        }
                }
        }
}
```

# Depth-first search (DFS)

La búsqueda en profundidad (DFS) es una técnica de exploración de grafos que explora las conexiones de un nodo de manera profunda hasta encontrarse en un camino sin salida, momento en el que vuelve a subir y explora el siguiente nodo no visitado. Lo más frecuente es implementar DFS de forma recursiva:

```
void dfs(int nodo, vector<bool> & visitado, vector<vector<int>> & listaAdyacencia)
↪  {
        cout << nodo << '\n';
        visitado[nodo] = true;
        for (int conexion: listaAdyacencia[nodo]) {
                if (!visitado[conexion])
                        dfs(conexion, visitado, listaAdyacencia);
        }
}
```

**Extra: la función DFS puede representarse de manera iterativa en vez de con esta función recursiva. Esto es útil para grafos muy alargados y con muchos nodos, que pueden aparecer en concursos. El compilador de C++, cuando llega a una cierta profundidad con la recursión**

**(por ejemplo, 50000 invocaciones encadenadas), corta el programa. Esto no ocurre con ese número de iteraciones de un bucle. Prueba a implementar DFS usando un stack y un bucle, de una forma similar a cómo BFS usaba la cola.**

# Componentes conexas

Las funciones que mostré antes de BFS y DFS asumen que se puede llegar a todos los nodos del grafo empezando por la raíz elegida. Este no tiene por qué ser el caso. Llamamos componente conexa en un grafo no dirigido a un grupo de nodos conectados todos entre sí. La componente conexa contiene todas las conexiones de los nodos que forman parte de ella.

**Extra: modifica las funciones iterativas de BFS y DFS para que se tenga en cuenta este caso. La función recursiva de DFS se mantiene; lo que hay que cambiar es las veces que se la llama y con qué nodo.**

Podemos encontrar las componentes conexas de un grafo usando cualquiera de BFS y DFS. Yo suelo usar DFS por ser más corto de programar, pero cada uno tiene su preferencia. El siguiente código guarda, para cada nodo, el índice de la componente a la que pertenece.

```
1  void dfsComponente(int nodo, int ccActual, vector<int> & componente,
   ↪  vector<vector<int>> & listaAdyacencia) {
2          cout << nodo << " pertenece a " << ccActual << '\n';
3          componente[nodo] = ccActual;
4          for (int conexion: listaAdyacencia[nodo]) {
5                  if (componente[conexion] == -1)
6                          dfsComponente(conexion, ccActual, componente,
                           ↪  listaAdyacencia);
7          }
8  }
9
10 void componentesConexas(vector<vector<int>> & listaAdyacencia) {
11         int N = (int) listaAdyacencia.size();
12         vector<int> componente(N, -1);
13         int numeroComponentes = 0;
14         for (int i = 0; i < N; i++) {
15                 if (componente[i] == -1) {
16                         numeroComponentes++;
17                         dfsComponente(i, numeroComponentes, componente,
                           ↪  listaAdyacencia);
18                 }
19         }
20 }
```

# Componentes fuertemente conexas (algoritmo de Kosaraju)

En un grafo dirigido, una componente fuertemente conexa es un grupo de nodos desde el cual cualquier nodo dentro del grupo puede llegar al resto de nodos en el grupo. Un nodo $u$ pertenece a una sola componente fuertemente conexa, es decir, que una componente fuertemente conexa contiene a todos los nodos para los que eso se cumple y, por lo tanto, tiene el mayor tamaño posible.

Encontrar las componentes fuertemente conexas es algo más tedioso que encontrar las componentes conexas. Para ello, usaremos DFS una vez para obtener un orden reverso de los nodos, es decir, añadir cada nodo al stack cuando terminas de visitarlo en vez de cuando primero lo visitas. Así, tendremos

encima del stack los últimos nodos con los que terminamos, es decir, que el nodo arriba del todo será la raíz que hayamos elegido.

Procederemos a invertir las aristas del grafo, es decir, a obtener el mismo grafo pero cambiando la dirección de cada arista para que vaya en sentido contrario.

Dejaremos el stack vacío quitando nodo a nodo. Si este nodo no ha sido visitado todavía, usamos la función `dfsComponente()` que definimos en la página anterior para obtener los nodos en las componente fuertemente conexa del nodo actual.

Para entender más a fondo por qué este programa funciona, puede ser útil la siguiente referencia: https://www.geeksforgeeks.org/strongly-connected-components/.

```
1  void dfsCFC(int nodo, vector<bool> & visitado, stack<int> & S, vector<vector<int>>
   ↪  & listaAdyacencia) {
2          visitado[nodo] = true;
3          for (int conexion: listaAdyacencia[nodo]) {
4                  if (!visitado[conexion])
5                          dfsCFC(conexion, visitado, S, listaAdyacencia);
6          }
7          S.push(nodo);
8  }
9
10 void componentesFuertementeConexas(vector<vector<int>> & listaAdyacencia) {
11         int N = (int) listaAdyacencia.size();
12         // Paso 1: DFS guardando cada nodo tras visitar todas sus conexiones de
           ↪  mayor profundidad
13         stack<int> S;
14         vector<bool> visitado(N, false);
15         for (int i = 0; i < N; i++) {
16                 if (!visitado[i])
17                         dfsCFC(i, visitado, S, listaAdyacencia);
18         }
19         // Paso 2: Invertir el grafo
20         vector<vector<int>> grafoInv(N, vector<int>());
21         for (int i = 0; i < N; i++) {
22                 for (int u: listaAdyacencia[i]) {
23                         grafoInv[u].push_back(i);
24                 }
25         }
26         // Paso 3: Obtener las componentes
27         vector<int> componente(N, -1);
28         int numeroComponentes = 0, nodoActual;
29         while (!S.empty()) {
30                 nodoActual = S.top();
31                 S.pop();
32                 if (componente[nodoActual] == -1) {
33                         numeroComponentes++;
34                         dfsComponente(nodoActual, numeroComponentes, componente,
                           ↪  grafoInv);
35                 }
36         }
37 }
```

# Programación dinámica sobre grafos

La programación dinámica es una técnica que puede sernos muy útil para problemas de grafos y de árboles en particular. La diferencia es que los estados de la DP son los nodos del grafo.

# Ejemplo: CSES Subordinates

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

void dfs(int nodo, vector<int> & DP, vector<vector<int>> & listaAdyacencia) {
        // DP[nodo] = 0
        for (int conexion: listaAdyacencia[nodo]) {
                dfs(conexion, DP, listaAdyacencia);
                DP[nodo] += 1 + DP[conexion];
        }
}

int main() {
        ios::sync_with_stdio(false);
        cin.tie(NULL);
        int N, padre;
        cin >> N;
        vector<vector<int>> listaAdyacencia(N, vector<int>());
        vector<int> DP(N, 0);
        for (int i = 1; i < N; i++) {
                cin >> padre;
                listaAdyacencia[padre-1].push_back(i);
        }
        dfs(0, DP, listaAdyacencia);
        for (int i = 0; i < N; i++)
                cout << DP[i] << ' ';
        return 0;
}
```

# Addendum

## 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.

```
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;
    ordenDescubrimiento.assign(N, -1); // ordenDescubrimiento guarda el orden en el que el DFS encuentra cada nodo
    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)
    nodoCorte.assign(N, false);
    for (int i = 0; i < N; i++) {
        hijos = 0;
        if (ordenDescubrimiento[i] == -1) {
            // 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  }
```