Apuntes OIFem 2020

Grafos Parte 1

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);
        int nodoActual;
        while(!colaBFS.empty()) {
                nodoActual = colaBFS.front();
                visitado[nodoActual] = true;
                cout << nodoActual << '\n';
                for (int conexion: listaAdyacencia[nodoActual]) {
                        if (!visitado[conexion])
                                colaBFS.push(conexion);
                }
        }
}
```

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

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/.

```
void dfsCFC(int nodo, vector<bool> & visitado, stack<int> & S, vector<vector<int>>
↪  & listaAdyacencia) {
        visitado[nodo] = true;
        for (int conexion: listaAdyacencia[nodo]) {
                if (!visitado[conexion])
                        dfsCFC(conexion, visitado, S, listaAdyacencia);
        }
        S.push(nodo);
}

void componentesFuertementeConexas(vector<vector<int>> & listaAdyacencia) {
        int N = (int) listaAdyacencia.size();
        // Paso 1: DFS guardando cada nodo tras visitar todas sus conexiones de
        ↪  mayor profundidad
        stack<int> S;
        vector<bool> visitado(N, false);
        for (int i = 0; i < N; i++) {
                if (!visitado[i])
                        dfsCFC(i, visitado, S, listaAdyacencia);
        }
        // Paso 2: Invertir el grafo
        vector<vector<int>> grafoInv(N, vector<int>());
        for (int i = 0; i < N; i++) {
                for (int u: listaAdyacencia[i]) {
                        grafoInv[u].push_back(i);
                }
        }
        // Paso 3: Obtener las componentes
        vector<int> componente(N, -1);
        int numeroComponentes = 0, nodoActual;
        while (!S.empty()) {
                nodoActual = S.top();
                S.pop();
                if (componente[nodoActual] == -1) {
                        numeroComponentes++;
                        dfsComponente(nodoActual, numeroComponentes, componente,
                        ↪  grafoInv);
                }
        }
}
```

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;
}
```