Apuntes OIFem 2020

Árboles

Qué es un árbol y sus propiedades más útiles

Un árbol es un grafo conectado, es decir, formado por una sola componente conexa. Además, si un árbol tiene N nodos, tiene $N-1$ aristas. En esta clase vamos a trabajar con árboles con aristas bidireccionales. Lo que esto significa es que hay una sola forma de llegar desde cualquier nodo u del grafo a cualquier otro nodo v. Esta propiedad simplifica varios problemas relacionados con grafos:

- Aristas y vértices de corte- en un árbol todos los nodos son nodos de corte y todas las aristas son aristas de corte.
- Camino más corto entre dos nodos- la semana pasada aprendimos a encontrar el camino más corto entre dos nodos en un grafo con aristas sin costes usando BFS (ejemplo: problema de Erdös). La semana que viene veremos cómo hacer esto con grafos con aristas con costes que no tengan por qué ser árboles. Cuando un grafo es un árbol, podemos encontrar el camino entre el nodo u y el nodo v de manera sencilla, iniciando un DFS con u como raíz y, para cada invocación al DFS, retornando 10^9 (infinito) para los nodos donde no se encuentre v en su subárbol y el coste en el caso de que v esté en ese subárbol, sumando la arista entre el nodo actual y ese subárbol.
- Camino más corto entre todos los nodos- la semana que viene veremos cómo, para cualquier grafo, obtener una matriz donde la casilla (u, v) contenga el coste del camino más corto para llegar de u a v. En un árbol podemos hacer esto de forma más sencilla, repitiendo la anterior función con cada nodo u como raíz. **Extra: ¿sabrías programar este algoritmo? Puedes mandar tu código a Blanca por Discord para que te lo revise.**
- Diámetro- podemos encontrar el diámetro de un árbol de dos maneras, como vimos en la última clase: con dos BFS, uno para encontrar el nodo u más lejano de nuestra raíz arbitraria, y otro para encontrar el nodo v más lejano de u, retornando la distancia $u-v$, o usando programación dinámica (el código está en la web, en los materiales de la clase 7).

Código: Camino más corto entre u y v

```cpp
vector<vector<pair<int, int>>> listaAdyacencia; // formato: [nodo v, coste u-v]

int caminoMasCorto(int u, int v, int padre) {
        if (u == v)
                return 0;
        int respuesta = 1e9;
        for (auto x: listaAdyacencia[u]) {
                if (x.first == padre)
                        continue;
                respuesta = min(respuesta, caminoMasCorto(x.first, v, u) +
                ↪  x.second);
        }
        return respuesta;
}
```

Código: Diámetro con BFS

```cpp
int diametro(int raiz, vector<vector<int>> & listaAdyacencia) {
        int N = (int) listaAdyacencia.size();
        queue<int> colaBFS;
        colaBFS.push(raiz);
        int nodoActual;
        vector<int> dist(N, 1e9);
        dist[raiz] = 0;
        int nodoLejano = 0;
        // buscamos el nodo más lejano de 0
        while(!colaBFS.empty()) {
                nodoActual = colaBFS.front();
                colaBFS.pop();
                nodoLejano = nodoActual;
                for (int conexion: listaAdyacencia[nodoActual]) {
                        if (dist[conexion] > dist[nodoActual]+1) {
                                dist[conexion] = dist[nodoActual]+1;
                                colaBFS.push(conexion);
                        }
                }
        }
}
        colaBFS.push(nodoLejano);
        dist.assign(N, 1e9);
        dist[nodoLejano] = 0;
        // buscamos el nodo más lejano de nodoLejano
        while(!colaBFS.empty()) {
                nodoActual = colaBFS.front();
                colaBFS.pop();
                nodoLejano = nodoActual;
                for (int conexion: listaAdyacencia[nodoActual]) {
                        if (dist[conexion] > dist[nodoActual]+1) {
                                dist[conexion] = dist[nodoActual]+1;
                                colaBFS.push(conexion);
```

```
                        }
                }
        }
        return dist[nodoLejano];
}
```

Programación dinámica sobre árboles

Los problemas más frecuentes de programación dinámica sobre grafos son los relacionados con árboles. Aunque la técnica es la misma, es fundamental tener práctica con este tipo de problemas. Aquí incluyo un ejemplo práctico, además de los que veréis entrenando (Traffic Congestion, por ejemplo).

Ejemplo: CSES Tree Distances 2

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

int N;
long long int caminosRaiz; // guarda la suma de los caminos a la raíz = suma de la profundidad de todos los nodos
vector<vector<int>> listaAdyacencia;
vector<long long int> tamanoSubarbol, sumaCaminos;
/*
 * tamanoSubarbol -> tamaño del subárbol bajo cada nodo, incluyendo dicho nodo
 * sumaCaminos -> respuesta final
 */

long long int dfs(int S, int padre, int prof) {
        // retorna el tamaño del subárbol con raíz S
        if (tamanoSubarbol[S] != -1)
                return tamanoSubarbol[S];
        caminosRaiz += prof;
        tamanoSubarbol[S] = 1;
        for (int v: listaAdyacencia[S]) {
                if (v != padre) {
                        dfs(v, S, prof+1);
                        tamanoSubarbol[S] += tamanoSubarbol[v];
                }
        }
        return tamanoSubarbol[S];
}

void calcularCaminos(int S) {
        for (int v: listaAdyacencia[S]) {
                if (sumaCaminos[v] != -1) continue; // v es el padre de S
                // los caminos a v son la suma de caminos a su padre S con dos cambios:
                // los caminos u-S donde u está fuera del subárbol de v tienen una arista más (S-v) (sumamos N - tamanoSubarbol[v])
                // los caminos u-S donde u está en el subárbol de v tienen una arista menos (v-S) (restamos tamanoSubarbol[v])
                sumaCaminos[v] = sumaCaminos[S] + N - 2*tamanoSubarbol[v];
```

```
                calcularCaminos(v);
        }
}

int main() {
        ios::sync_with_stdio(false);
        cin.tie(NULL);
        cin >> N;
        listaAdyacencia.assign(N, vector<int>());
        tamanoSubarbol = sumaCaminos = vector<long long int>(N, -1);
        int a, b;
        for (int i = 0; i < N-1; i++) {
                cin >> a >> b;
                listaAdyacencia[a-1].push_back(b-1);
                listaAdyacencia[b-1].push_back(a-1);
        }
        dfs(0, -1, 0);
        sumaCaminos[0] = caminosRaiz;
        calcularCaminos(0);
        cout << sumaCaminos[0];
        for (int i = 1; i < N; i++) cout << " " << sumaCaminos[i];
        cout << "\n";
        return 0;
}
```

Conjuntos distintos

Esta estructura de datos (UFDS por sus siglas en inglés) es una forma de obtener las componentes conexas de un grafo de forma constructiva. Es decir, es una forma sencilla de empezar con una serie de conjuntos distintos e ir juntando los conjuntos de forma progresiva. Normalmente, el estado inicial con n nodos es tener n conjuntos, uno con cada nodo.

Si tenemos un conjunto 1 con dos nodos a y b y un conjunto 2 con dos nodos c y d y conectamos b y d, pasamos a tener un solo conjunto con los 4 nodos. Una vez hayamos procesado todas las aristas/conexiones, obtendremos una serie de conjuntos con los nodos.

Esta estructura se construye como un bosque de árboles, donde cada conjunto es un árbol. Para describir en qué conjunto está un nodo u, tenemos de información a su padre, un nodo S. Seguimos subiendo por el árbol hasta llegar a su raíz. Por lo tanto, dos nodos estarán en el mismo conjunto solo si comparten raíz. Esto significa que, para unir los nodos u y v, hay que unir las raíces de estos dos conjuntos. Para que sea más eficiente, haremos que la raíz con más nodos debajo sea la raíz del nuevo conjunto. Utilizaremos el vector `alturaAprox` como forma de representar la profundidad de un árbol. Además, podemos ir acercando a los nodos a la raíz de su conjunto con cada búsqueda que hagamos, ya que lo que nos importa es la raíz de su conjunto, no el camino para llegar a ella.

Código: Conjuntos distintos

```cpp
int numeroConjuntos; // en main() debería inicializarse a n, el número de nodos
vector<int> padre, alturaAprox; // padre debería inicializarse a n posiciones,
↪  donde padre[i] = i para cada nodo de 0 a n-1 y alturaAprox = 0 en cada una de
↪  las n posiciones

int encontrarRaiz(int a) {
        if (padre[a] == a) return a; // raíz encontrada
        return padre[a] = encontrarRaiz(padre[a]); // seguimos buscando la raíz de
        ↪  a y, tras encontrarla, la guardamos como el padre de a, para que las
        ↪  futuras búsquedas sean más rápidas
}

bool mismoConjunto(int a, int b) {
        return encontrarRaiz(a) == encontrarRaiz(b);
}

void unirConjuntos(int a, int b) {
        int raiz_a, raiz_b;
        raiz_a = encontrarRaiz(a);
        raiz_b = encontrarRaiz(b);
        if (raiz_a == raiz_b) return;
        numeroConjuntos--;
        if (alturaAprox[raiz_a] > alturaAprox[raiz_b]) {
                padre[raiz_b] = raiz_a;
        } else if (alturaAprox[raiz_b] > alturaAprox[raiz_a]) {
                padre[raiz_a] = raiz_b;
        } else {
                padre[raiz_a] = raiz_b;
                alturaAprox[raiz_b]++;
        }
}
```

Árboles recubridores mínimos

Dado un conjunto de n nodos y m aristas, un árbol recubridor mínimo será una selección de $n-1$ aristas de forma que todos los nodos estén conectados y la suma de los costes de estas $n-1$ aristas sea la menor posible.

Hay dos métodos que se utilizan para encontrar este árbol recubridor mínimo (MST por sus siglas en inglés): el de Prim y el de Kruskal. Hoy aprenderemos Kruskal, ya que los problemas donde se puede usar Prim se pueden resolver también con Kruskal, pero no necesariamente con Prim los problemas de Kruskal. Puede ser interesante leer sobre cómo funciona Prim, pero creo que os compensa el triple hacer los problemas de entrenamientos anteriores que no hayáis logrado resolver.

Dicho esto, el método de Kruskal es bastante intuitivo. Ordenamos las aristas en orden de coste ascendiente y vamos construyendo el árbol hasta que este tiene $n-1$ aristas y, por tanto, sabemos que está completo. Cuando llegamos a la arista $u-v$, la seleccionamos si no forma un ciclo y, si no, pasamos a la siguiente arista. Para ver si dicha arista formará un ciclo, basta con asegurarnos de que u y v aún no estén conectados. Podemos hacer esto de forma sencilla usando una estructura ya aprendida: conjuntos distintos. Sabiendo esto, el código es trivial.

Código: Algoritmo de Kruskal

```cpp
// Incluir código de UFDS para usar sus funciones (tenéis que acordaros de hacer esto; yo me lo salté para evitar ocupar más espacio con lo mismo).

int kruskalMST(int N, vector<pair<int, pair<int, int>>> & aristas) {
    // aristas es un vector que contiene las aristas en formato [coste, [u, v]]
    sort(aristas.begin(), aristas.end());
    padre.assign(N, 0);
    for (int i = 1; i < N; i++) padre[i] = i;
    alturaAprox.assign(N, 0);
    numeroConjuntos = N;
    int j = 0, u, v, respuesta = 0;
    while(numeroConjuntos > 1) {
        u = aristas[j].second.first;
        v = aristas[j].second.second;
        if (mismoConjunto(u, v)) {
            j++;
            continue;
        }
        respuesta += aristas[j].first;
        unirConjuntos(u, v);
        j++;
    }
    return respuesta;
}
```