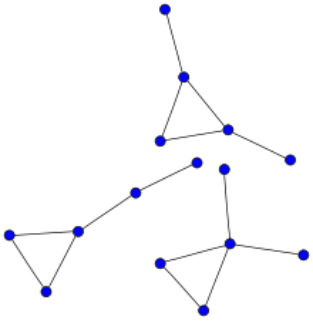
GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: COMPONENTES CONEXAS

1. Introducción

Dado un grafo no dirigido G con n nodos y m aristas. Queremos encontrar en él todas las componentes conexas, es decir, varios grupos de vértices tales que dentro de un grupo se puede llegar a cada vértice desde otro y no existe camino entre diferentes grupos. Por ejemplo en la imagen de abajo tenemos un grafo con tres componentes conexas.



2. Conocimientos previos

2.1. Componente conexa

En teoría de grafos, una componente conexa es un subgrafo inducido de un grafo en que cualquiera dos vértices que integran este subgrafo están conectados mediante un camino. Un vértice aislado, el grafo trivial o un grafo conexo son en sí mismos componentes. Es importante que el termino de componente conexa solo es aplicable a un **grafo no dirigido**.

3. Desarrollo

Para resolver el problema, podemos usar la búsqueda primero en profundidad o la búsqueda primero en amplitud. De hecho, haremos una serie de rondas de DFS: la primera ronda comenzará desde el primer nodo y todos los nodos en el primer componente conectado serán atravesados (encontrados). Luego encontramos el primer nodo no visitado de los nodos restantes y ejecutamos la primera búsqueda en profundidad en él, encontrando así un segundo componente conectado. Y así sucesivamente, hasta que se visiten todos los nodos.

Las funciones profundamente recursivas son en general malas. Cada llamada recursiva requerirá un poco de memoria en la pila y, por defecto, los programas solo tienen una cantidad limitada de espacio en la pila. Entonces, cuando realiza un DFS recursivo sobre un gráfico conectado con millones de nodos, es posible que se encuentre con desbordamientos de pila. Siempre es posible convertir un programa recursivo en un programa iterativo, manteniendo manualmente una estructura de datos de pila. Dado que esta estructura de datos se asigna en el montón, no se producirá ningún desbordamiento de pila.

4. Implementación

4.1. C++

```
int n; //cantidad de nodos del grafo
vector< vector<int> > adj; // lista de adyacencia para representar el grafo
vector<bool> used;
vector<int> comp;

void dfs(int v) {
   stack<int> st;
   st.push(v);

   while (!st.empty()) {
      int curr = st.top();
      st.pop();
      if (!used[curr]) {
         used[curr] = true;
         comp.push_back(curr);
         for(int i = adj[curr].size()-1;i>=0;i--){
            st.push(adj[curr][i]);
         }
      }
   }
}

void find_comps() {
   fill(used.begin(), used.end(), 0);
   for (int v = 0; v < n ; ++v) {
      if (!used[v]) {
         comp.clear();
         dfs(v);
         cout << "Los nodos de esta componente son:" ;
         for (int u : comp)
            cout << ' ' << u;
         cout << endl ;
      }
   }
}
```

La función más importante que se utiliza es *find_comps()* la que encuentra y muestra los componentes conectados en el grafo. El grafo se almacena en representación de lista de adyacencia, es decir, $adj[v]$ contiene una lista de vértices que tienen aristas desde el vértice v. El vector *comp* contiene una lista de nodos en el componente conectado actual.

4.2. Java

```
private class Graph {
```

```
  private int nodes;
  private List<List<Integer>> ady;

  public Graph(int n) {
     this.nodes = n;
     this.ady = new ArrayList<List<Integer>>();

     for (int i = 0; i < this.nodes; i++)
                         this.ady.add(new ArrayList<Integer>());
  }

  public void addEdge(int a, int b) {
     this.ady.get(a).add(b); this.ady.get(b).add(a);
  }

  public void dfs(int v, boolean[] used, ArrayList<Integer> comp) {
     Stack<Integer> st = new Stack<Integer>();
     st.push(v);

     while (!st.empty()) {
        int curr = st.pop();
        if (!used[curr]) {
           used[curr] = true; comp.add(curr);
           for (int i = this.ady.get(curr).size() - 1; i >= 0; i--)
              st.push(this.ady.get(curr).get(i));
        }
         }
  }

  public void find_comps() {
     boolean[] used = new boolean[this.nodes];
     Arrays.fill(used, false);
     ArrayList<Integer> comp = new ArrayList<Integer>();
     for (int v = 0; v < this.nodes; ++v) {
        if (!used[v]) {
           comp.clear(); dfs(v, used, comp);
           System.out.printf("Los nodos de esta componente son:\n");
           for (Integer u : comp) System.out.printf(" %d", u);
           System.out.printf("\n");
        }
      }
   }
}
```

5. Complejidad

Como sabemos la complejidad del DFS es igual $O(V + E)$ siendo V y E los nodos y aristas respectivamente del grafo, y como es llamado dentro de un ciclo que se ejecuta n la complejidad

es igual a $O(n(V+E))$ donde n es igual a V. Por lo que la complejidad del algoritmo es $O(N(V+E))$. Pero no se alarmen por esta complejidad, veremos que no es alta como parece.

Vamos analizar los dos casos extremos:

- **Un grafo con V vértices con una sola componente conexa:** En un grafo con una sola componente conexa el DFS solo se ejecutará en la primera iteración del ciclo y en el resto de la iteraciones no será así porque ya los nodos estarán visitados y el tiempo será constante $O(1)$. Por lo que para ese caso la complejidad es $O(V+E)$.
- **Un grafo con V vértices con V componentes conexa:** En un grafo con V vértices y con V componentes conexa significa que es un grafo sin aristas esto significa que realizar un DFS sobre cualquier nodo de ese grafo va ser en tiempo constante $OO(1)$ porque es un solo nodo de esa supuesta componente conexas que no tiene arista y dada que este procedimiento se va repetir para los V vértices del grafo la complejidad para este caso es $O(V)$.

Cualquier otro caso va oscilar en este rango definido por estos dos casos explicados anteriormenente. Es por eso que en el peor de los casos este algoritmo va tener una complejidad de $O(V+E)$.

6. Aplicaciones

Las aplicaciones de saber detectar las componentes conexa de un grafo son varias sobre todo en el desarrollo de software que se basan en la detección de contornos. En el caso de los ejercicios y problemas es bastante aplicable sobre grafos que se representan en tableros o matrices. Lo más común en los ejercicios abordan esta temática puramente es saber la cantidad de componentes conexa, cual es la componente conexa con mayor o menor cantidad de nodos, existen otros ejercicios donde el comienzo de la solución pasa por saber detectar las componentes conexa.

7. Ejercicios propuestos

A continuación una lista de ejercicios que se resuelven si se es capaz de detectar las componentes conexas de un grafo:

- DMOJ - Avoid the lake
- DMOJ - Castillo medieval
- DMOJ - Conteo de Pozos
- DMOJ - Manchas en el Papel
- DMOJ - Closing the Farm