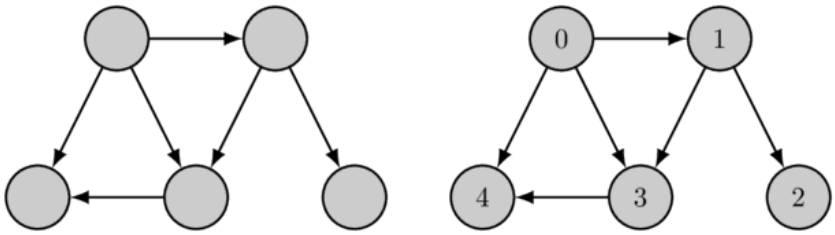
GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ORDENAMIENTO TOPOLÓGICO

1. Introducción

Se tiene un grafo dirigido con n vértices y m aristas. Tienes que encontrar un orden de los vértices, de modo que cada arista conduzca desde el vértice con un índice más pequeño a un vértice con uno más grande.

En otras palabras, desea encontrar una permutación de los vértices (orden topológico) que corresponda al orden definido por todas las aristas del grafo. Aquí hay un grafo dado junto con su orden topológico:



2. Conocimientos previos

2.1. Grafo dirigido

Un grafo dirigido o digrafo es un tipo de grafo en el cual las aristas tienen un sentido definido, a diferencia del grafo no dirigido, en el cual las aristas son relaciones simétricas y no apuntan en ningún sentido.

2.2. Componente fuertemente conexa

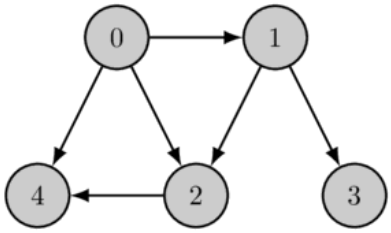
En grafos dirigidos, una componente fuertemente conexa (SCC por sus siglas en inglés, *Strongly Connected Component*) es un conjunto de nodos y aristas del grafo que cumplen 2 condiciones:

- Desde cada nodo de la componente existe un camino hacia todos los otros nodos que usa aristas únicamente de la componente.
- No hay un nodo del grafo fuera de la componente desde el cual se pueda llegar a la componente y al que se pueda llegar desde ella (es maximal, si existiera estaría en la componente).

3. Desarrollo

El orden topológico puede **no ser único** (por ejemplo, si existen tres vértices a , b , c para los que existen caminos desde a a b y de a a c pero no caminos de b a c o de c a b). El grafo de ejemplo también tiene múltiples órdenes topológicos, un segundo orden topológico es el siguiente:

Un orden topológico puede **no existir** en absoluto. Solo existe si el grafo dirigido no contiene ciclos. De lo contrario porque hay una contradicción: si hay un ciclo que contiene los vértices a y b,



entonces a necesita tener un índice más pequeño que b (ya que puedes llegar b de a) y también uno más grande (como se puede alcanzar a de b). El algoritmo descrito en esta guia también muestra por construcción que cada grafo dirigido acíclico contiene al menos un orden topológico.

Para resolver este problema utilizaremos la búsqueda en profundidad (DFS). Partimos que el grafo es acíclico. ¿Qué hace la búsqueda primero en profundidad?.

Al partir de algún vértice v, DFS intenta recorrer a lo largo de todas las aristas que salen de v. Se detiene en las aristas cuyos extremos ya han sido visitados previamente, y recorre el resto de las aristas y continúa recursivamente en sus extremos.

Por lo tanto, en el momento de la llamada a la función dfs(v) ha terminado, todos los vértices que son alcanzables desde v han sido visitados directamente (a través de un borde) o indirectamente por la búsqueda.

Agreguemos el vértice v a una lista, cuando terminemos dfs(v). Dado que ya se han visitado todos los vértices accesibles, ya estarán en la lista cuando agreguemos v. Hagamos esto para cada vértice del grafo, con una o varias ejecuciones de búsqueda en profundidad. Para cada arista dirigida $v \to u$ en el grafo, u aparecerá antes en esta lista que v, porque u es accesible desde v. Entonces, si solo etiquetamos los vértices en esta lista con $n-1, n-2, \ldots, 1, 0$, hemos encontrado un orden topológico del grafo. En otras palabras, la lista representa el orden topológico inverso.

Estas explicaciones también se pueden presentar en términos de tiempos de salida del algoritmo DFS. El tiempo de salida para el vértice v es el momento en que la función llama dfs(v) terminado (los tiempos se pueden numerar de 0 a $n-1$). Es fácil entender que el tiempo de salida de cualquier vértice v siempre es mayor que el tiempo de salida de cualquier vértice accesible desde él (ya que fueron visitados antes de la llamada dfs(v) o durante el mismo). Así, el ordenamiento topológico buscado son los vértices en orden decreciente de sus tiempos de salida.

4. Implementación

Aquí las implementaciones que asume que el grafo es acíclico, es decir, existe el ordenamiento topológico deseado.

4.1. C++

```
int n; // numero de vertices
```

```
vector< vector<int> > adj; //lista de adyacencia del grafo
vector<bool> visited;
vector<int> ans;

void dfs(int v){
   visited[v] = true;
   for (int u : adj[v]) {
      if (!visited[u]) dfs(u);
   }
   ans.push_back(v);
}

void topological_sort() {
   visited.assign(n, false);
   ans.clear();
   for (int i = 0; i < n; ++i) {
      if (!visited[i]) dfs(i);
   }
   reverse(ans.begin(), ans.end());
}
```

La función principal de la solución es *topological_sort*, que inicializa las variables DFS, inicia DFS y recibe la respuesta en el vector *ans*.

4.2. Java

```
public void dfs(List<Integer>[] graph, boolean[] used,
                List<Integer> order, int u){
   used[u] = true;
   for (int v : graph[u])
      if (!used[v])
         dfs(graph, used, order, v);
   order.add(u);
}

public List<Integer> topologicalSort(List<Integer>[] graph) {
   int n = graph.length;
   boolean[] used = new boolean[n];
   List<Integer> order = new ArrayList<>();
   for(int i = 0; i < n; i++)
      if(!used[i])
         dfs(graph, used, order, i);
   Collections.reverse(order);
   return order;
}
```

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 fuertemente 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 fuertemente conexa:** En un grafo con V vértices y con V componentes fuertemente 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 O$O(1)$ 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 anteriormente. Es por eso que en el peor de los casos este algoritmo va tener una complejidad de $O(V+E)$.

6. Aplicaciones

Un problema común en el que se produce la ordenación topológica es el siguiente. Hay n variables con valores desconocidos. Para algunas variables sabemos que una de ellas es menor que la otra. Debe verificar si estas restricciones son contradictorias y, de lo contrario, generar las variables en orden ascendente (si son posibles varias respuestas, generar cualquiera de ellas). Es fácil notar que este es exactamente el problema de encontrar el orden topológico de un gráfico con n vértices.

7. Ejercicios propuestos

A continuación una lista de ejercicios que se puede resolver aplicando este algoritmo:

- SPOJ - TOPOSORT - Topological Sorting
- UVA - 10305 - Ordering Tasks
- UVA - 124 - Following Orders
- UVA - 200 - Rare Order
- Codeforce - C. Fox And Names
- SPOJ - RPLA - Answer the boss!

- CSES - Course Schedule
- CSES - Longest Flight Route
- CSES - Game Routes