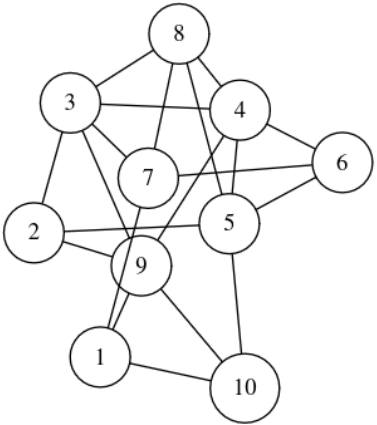
Introducción a los grafos

Sebastian Mestre

¿Qué es un grafo?

- Un **grafo** (V,E) tiene conjuntos V de **vértices** (o nodos) y E de **aristas** que los conectan.
- Nos permite modelar relaciones binarias: "está conectado con", "hay una ruta entre", etc.
- Super general: muchas cosas se pueden ver como casos particulares de grafos.



Aplicaciones clásicas

- Detectar partes de una red que estén **desconectadas**.
- Encontrar **caminos más cortos** en mapas (por ejemplo, rutas de GPS).
- Modelar **redes** de todo tipo: rutas aéreas, internet, redes sociales, etc.
- Resolver problemas clásicos como los **puentes de Königsberg**, que motivaron la teoría de grafos.

Grafos dirigidos y no dirigidos

- **Grafo dirigido** (dígrafo):
  - Las aristas tienen dirección: de un nodo u hacia un nodo v.
  - Una arista de u a v **no** implica una de v a u.
- **Grafo no dirigido**:
  - Las aristas no tienen dirección, la conexión es simétrica.
  - Usando la notación (u, v) se entiende que también existe (v, u).

Caminos y ciclos

- Un **camino** es una secuencia de vértices $v_0, v_1, \ldots, v_k$ tal que para cada i existe una arista entre v_i y v_{i+1}.
- Un **camino simple** es un camino donde no se repite ningún vértice.
- Un **ciclo simple** es un camino que repite solamente el primero y último, que son iguales.
- Si existe un camino entre dos vértices, decimos que **están conectados** o **son conexos**.
- La **distancia** entre dos nodos es la longitud (número de aristas) del camino más corto que los conecta. (no definida para nodos no-conexos)
- En un grafo no dirigido, una **componente conexa** es un conjunto de vértices tal que:
  - Cualquier par de vértices del conjunto está conectado por algún camino.
  - Ningún vértice fuera del conjunto está conectado con los de adentro.

Subgrafos, árboles y árboles recubridores

- Un **subgrafo** es un grafo formado por un subconjunto de los vértices y aristas del grafo original.
- Un **árbol** es un grafo **conexo** y **sin ciclos**.
  - Un árbol con n nodos tiene exactamente $n - 1$ aristas.
- Un **árbol recubridor** de un grafo es un subgrafo que:
  - Incluye **todos** los nodos del grafo original.
  - Es un árbol (conexo y sin ciclos).

Listas de adyacencia

- Una de las representaciones más usadas (sobre todo para grafos **ralos** / poco densos) es la **lista de adyacencia**.
- Para cada vértice u, guardamos una lista con sus vecinos.

C++

```cpp
int n; // cantidad de nodos (0..n-1)
vector<int> adj[maxn]; // adj[u] = vecinos de u
// o bien:
// vector<vector<int>> adj(n);

```

Agregando aristas

- Para una arista **dirigida** de u a v:

Dirigido

```
1 adj[u].push_back(v);
2
```

- En un grafo **no dirigido** debemos agregar la arista en "ambos sentidos":

No dirigido

```
1 adj[u].push_back(v);
2 adj[v].push_back(u);
3
```

Recorridos con “bolsas”

- Queremos recorrer todos los nodos de una componente conexa a partir de un nodo inicial s.
- Idea general:
  - Tomamos un nodo y agregamos todos sus vecinos a una **estructura de datos** (la “bolsa”).
  - Mientras la bolsa no esté vacía:
    - Sacamos un nodo u de la bolsa.
    - Agregamos sus vecinos aún no visitados a la bolsa.
- Según la estructura que usemos (pila, cola, etc.) obtenemos distintos recorridos.

DFS iterativo

- **DFS** (Depth-First Search) explora lo más profundo posible antes de retroceder.
- Versión iterativa usando una **pila**:

Implementación

```
vector<bool> visitado(n, false);
stack<int> S;
S.push(s); // nodo de partida
while (!S.empty()) {
  int u = S.top(); S.pop();
  if (visitado[u]) continue;
  visitado[u] = true;
  // procesar el nodo u aqui
  for (int v : adj[u])
    S.push(v);
}
```

BFS

- **BFS** (Breadth-First Search) recorre el grafo por **capas** de distancia desde el origen.
- Igual al DFS, pero usando una **cola**:

Implementación

```
vector<bool> visitado(n, false);
queue<int> Q;
Q.push(s);
while (!Q.empty()) {
  int u = Q.front(); Q.pop();
  if (visitado[u]) continue;
  visitado[u] = true;
  // procesar el nodo u aqui
  for (int v : adj[u])
    Q.push(v);
}
```

DFS recursivo

- Hay una version recursiva que nos permite procesar un nodo:
  - Antes de recorrer sus vecinos.
  - Al recorrer una arista (u, v).
  - Después de haber recorrido todos los vecinos.

Esquema típico

```
vector<bool> visitado(n, false);
void dfs(int u) {
  visitado[u] = true;
  // procesar el nodo u aqui (pre-orden)
  for (int v : adj[u]) {
    if (!visitado[v]) {
      // procesar la arista (u, v) aqui
      dfs(v);
    }
  }
  // procesar el nodo u aqui (post-orden)
}
```

Distancias en grafos

- En un grafo la distancia entre dos nodos es:
  - El número mínimo de aristas de un camino que los conecta.
- **BFS** calcula simultáneamente:
  - La distancia mínima desde un origen s a todos los nodos alcanzables.
  - Un **árbol de expansión** que permite reconstruir caminos mínimos.
  - La idea es: la primera vez que el BFS descubre un nodo guardamos el nodo del que llegamos, y la cantidad de pasos de distancia (que va a ser una mas que la del nodo del que llegamos).

BFS para distancias y camino

Cálculo de distancias

```
vector<int> dist(n, -1);
vector<int> prev(n, -1);
queue<int> Q;
Q.push(s);
dist[s] = 0;
while (!Q.empty()) {
  int u = Q.front(); Q.pop();
  if (visitado[u]) continue;
  visitado[u] = true;
  for (int v : adj[u]) {
    if (dist[v] == -1) {
      dist[v] = dist[u] + 1;
      prev[v] = u;
    }
    Q.push(v);
  }
}
```

Reconstruyendo el camino

De s a t usando prev

```
vector<int> path;
int u = t; // nodo final
while (u != -1) {
  path.push_back(u);
  u = prev[u];
}
reverse(begin(path), end(path));

```

- El vector path contiene un camino mínimo desde s hasta t.
- No olvidar darlo vuelta porque está ordenado del final al principio

Grafos de estados

- A veces el nodo del grafo no representa solo una "posición física", sino un **estado** más complejo.
- Ejemplo: ciudades conectadas por carreteras, algunas peligrosas; queremos llegar con a lo sumo K carreteras peligrosas.
- Un estado puede representarse como (u, c):
  - u: ciudad actual.
  - c: cuántas carreteras peligrosas usamos hasta ahora.
- La idea es construir un grafo donde cada nodo representa un estado posible y aplicar BFS sobre ese grafo.

Implementación explícita por capas

- Podemos construir un grafo con $n \cdot (k+1)$ nodos, separados en $k+1$ **capas**.
- Necesitamos asignar los pares (u,c) a numeros en el intervalo de 0 a $n \cdot (k+1) - 1$:

Mapeo $(u, c) \mapsto$ índice

```
1 int idx(int u, int c) { return c * n + u; }
```

- Luego construimos las aristas según si son peligrosas o no.

Construcción del grafo

```
vector<vector<int>> g(n * (k + 1));
for (int i = 0; i < m; ++i) {
  int u, v, w;
  cin >> u >> v >> w;
  --u; --v;
  if (w == 0) { // arista segura
    for (int c = 0; c <= k; ++c) {
      g[idx(u, c)].push_back(idx(v, c));
      g[idx(v, c)].push_back(idx(u, c));
    }
  } else { // arista peligrosa
    for (int c = 0; c < k; ++c) {
      g[idx(u, c)].push_back(idx(v, c + 1));
      g[idx(v, c)].push_back(idx(u, c + 1));
    }
  }
}

```

Implementación implícita

- Alternativa: mantener el grafo original con etiquetas en las aristas y modificar el BFS para tener en cuenta el contador c.
- O sea, representamos el grafo con `vector<vector<pair<int, bool>> g(n);`
- Ventajas / desventajas:
  - Menos memoria, pero código más específico del problema.

BFS multisource y super fuente

- A veces queremos, para cada nodo, la distancia al **más cercano** entre varios orígenes.
- **BFS multisource**: inicializar la cola con todos los nodos fuente.

BFS multisource

```
queue<int> Q;
for (int s : sources) Q.push(s);
while (!Q.empty()) {
  // codigo del BFS normal
}

```

- Variante equivalente: crear una **super fuente** conectada a todos los orígenes y hacer BFS desde ella.

Invertir las aristas

- A veces nos interesa preguntarnos "**quién puede llegar a este nodo**" en lugar de "a dónde puedo ir desde este nodo".
- Truco estándar: **invertir las aristas** del grafo y trabajar en el grafo transpuesto.
- Esto también es útil como truco de implementación en ciertos problemas de caminos mínimos o de propagación.
- Muchos problemas de competencias (por ejemplo, el A del Regional ICPC 2023) usan este tipo de idea.

Problemas sugeridos

- cses.fi/problemset/task/1666
- cses.fi/problemset/task/1193
- cses.fi/problemset/task/1668
- cses.fi/problemset/task/1194
- Codeforces Group contest 104252 A (Latam 2023 A)

Enlaces útiles

- Codeforces blog: referencias y problemas de grafos

Ideas clave

- Los grafos modelan una enorme variedad de problemas de programación competitiva.
- **Listas de adyacencia**, **DFS** y **BFS** son las herramientas básicas para trabajar con ellos.
- Extender el modelo a **grafos de estados** y usar trucos como BFS multisource o invertir aristas permite resolver problemas más complejos.

Preguntas

Q&A