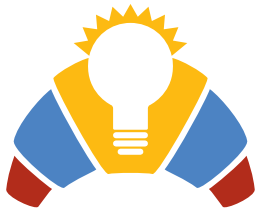
Grafos en ICPC

Franco De Rico[1]

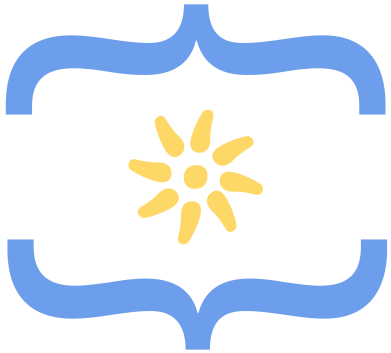
[1]Universidad Nacional de Rosario
Don Gato

Training Camp 2026



¡Gracias sponsors!

Organiza



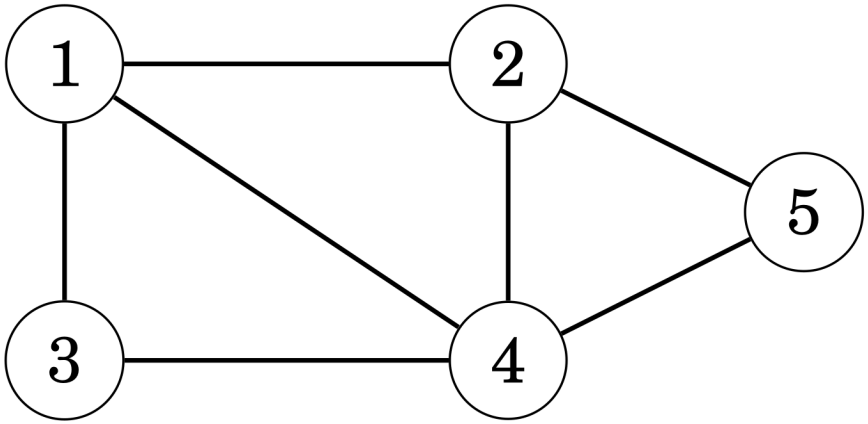
Facultad de INFORMÁTICA

UNIVERSIDAD NACIONAL DE LA PLATA

Diamond Sponsor

Definición

- Un grafo es un conjunto de puntitos unidos por palitos.

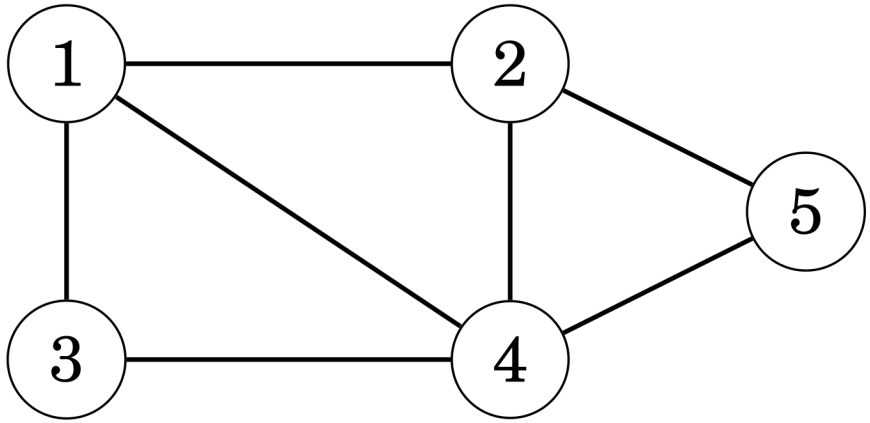


- A los puntitos se los llama **nodos**, y a los palitos se los llama **aristas**.

Representación

- Para trabajar con grafos en nuestros programas, tenemos que elegir una forma de guardar la información.
- La forma más cómoda es que cada nodo sea un número entero. Las aristas no las guardamos explícitamente, sino que cada nodo guarda una lista de sus nodos vecinos.
- Esto se llama **lista de adyacencia**.

Representación




```
vecinos[1]: 2, 3, 4
vecinos[2]: 1, 4, 5
vecinos[3]: 1, 4
vecinos[4]: 1, 2, 3
vecinos[5]: 2, 4
```

Problema

Rumor

La Plata es una ciudad con n personas. Entre ellas hay m pares de amigos. Uli quiere divulgar un rumor por toda la ciudad. Sabe que si le paga a una persona de expandir el rumor, ella se lo contará a todos sus amigos, y cada uno de ellos se lo contará a todos sus amigos, y así sucesivamente.
La persona i pide que le paguen c_i monedas para empezar a divulgar el rumor. ¿Cuál es la menor cantidad de monedas que deberá gastar Uli para asegurarse de que el rumor llegue a todos los habitantes?

Problema

Ejemplo:

```
5 3
2 10 8 9 3
1 2
2 3
4 5
```

Solución

```
const ll MAXN = 2e5+100; // max cantidad de personas
vector<int> grafo[MAXN]; // lista de adyacencia (el grafo)
ll costo[MAXN];          // array con el costo de cada nodo

int main(){
  int n, m;
  cin >> n >> m;
  forn(i, n)
    cin >> costo[i];
  forn(j, m){
    int a, b;
    cin >> a >> b;
    a --; b --; // nos gusta indexar en 0 en vez de en 1
    grafo[a].push_back(b);  // agregamos b como vecino de a
    grafo[b].push_back(a);  // agregamos a como vecino de b
  }

  // ?
}
```

Solución

```
bool vis[MAXN];             // array de nodos visitados

ll dfs(int v){
  ll m = costo[v];

  vis[v] = true;

  for(auto x : grafo[v]){ // recorremos los vecinos de v
    if(not vis[x]){ // pero solo los que aun no visitamos
      m = min(m, dfs(x));
    }
  }

  return m;
}
```

Solución

```
int main(){
  // aca va el input que leimos antes

  ll ans = 0;

  forn(i, n)
    if(not vis[i])
      ans += dfs(i);

  cout << ans << '\n';
}
```

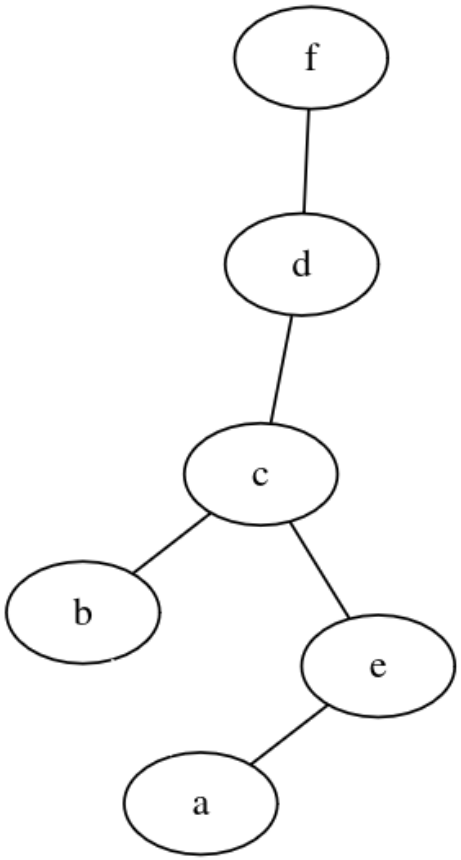
DFS

```
vector<int> grafo[MAXN]; // lista de adyacencia (el grafo)
bool vis[MAXN];          // array de nodos visitados

void dfs(int v){
  vis[v] = true;

  // aca podemos procesar cada nodo antes que a sus vecinos.
  // sugerencia: probar que pasa si aca hacemos:
  cout << v << '\n';

  for(auto x : grafo[v]){ // recorremos los vecinos de v
    if(not vis[x]){ // pero solo los que aun no visitamos
      dfs(x);
    }
  }

  // aca podemos procesar cada nodo despues que a sus vecinos.
}
```

Complejidad: $\mathcal{O}(n+m)$.

Árboles

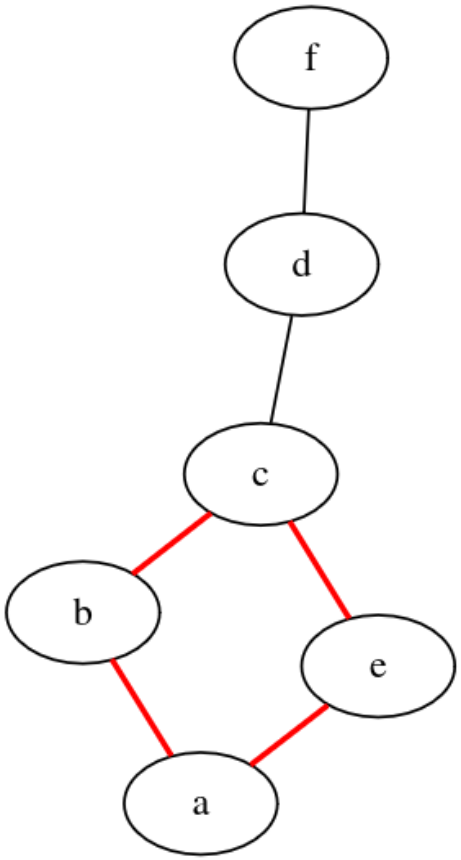
Un **árbol** es un grafo conexo que no tiene ciclos.



Esto es un árbol.

Árboles

Un **árbol** es un grafo conexo que no tiene ciclos.



Esto NO es un árbol.

Árboles

Propiead muy útil: si sabemos que tenemos un grafo conexo con n nodos y $n-1$ aristas, entonces es un árbol.

Árboles

- Cuando hacemos funciones recursivas, los árboles aparecen naturalmente, incluso si el problema no es de grafos.
- Cuando recorremos un grafo usando una dfs, la estructura que se arma es la de un árbol que cubre el grafo original, más algunas aristas que van "hacia atrás" (las de los vecinos que no recorrimos). Este se llama **dfs tree**.

Problema

Calles

La Plata es una ciudad muy ordenada que puede representarse como un tablero de $10^9 \times 10^9$.
Joaco está en su casa en la celda (a, b) y quiere llegar a la facultad, que está en la celda (c, d).
Puede caminar de una casilla a otra vecina (por lado o por vértice), pero hay solo $k \leq 10^5$ casillas válidas por las que se puede pasar.
Hallar la mínima cantidad de movimientos que tiene que hacer Joaco para llegar de su casa a la facultad, pasando solo por casillas válidas.

BFS

```
vector<ll> bfs(int v){
  vector<ll> dist(n, -1); // inicialmente no sabemos llegar a
                          // ningun nodo
  dist[v] = 0;  // el nodo inicial esta a dist 0 de si mismo
  queue<int> q; // cola para el orden en que procesamos los nodos
  q.push(v);

  while(not q.empty()){
    int x = q.front();
    q.pop();
    for(auto y : grafo[x])
      // solo procesamos nodos que aun no vimos
      if(dist[y] == -1){
        q.push(y);
        dist[y] = dist[x]+1;
      }
  }
  return dist;
}
```

Complejidad: $\mathcal{O}(n+m)$.

Problema

Ciudades

En un mapa hay $n \leq 10^3$ ciudades. Hay $m \leq 10^4$ rutas entre las ciudades. Cada ruta es de una sola mano, y tiene cierta longitud $l \leq 10^9$. Hallar la distancia mínima para ir desde una ciudad inicial p hasta cualquiera de las otras ciudades.

```cpp
int main(){
  int n, m;
  cin >> n >> m;
  forn(j, m){
    int a, b; cin >> a >> b;
    ll largo; cin >> largo;
    grafo[a].push_back(b);      //    agregamos b como vecino de a
    // grafo[b].push_back(a);  // NO agregamos a como vecino de b
  }

  // ?
}
```

Grafos dirigidos y ponderados

- Este grafo es **dirigido** porque cada arista tiene dirección (es una flecha). Un grafo no dirigido es como un grafo dirigido, pero siempre tenemos las 2 flechas.

Grafos dirigidos y ponderados

- Este grafo es **dirigido** porque cada arista tiene dirección (es una flecha). Un grafo no dirigido es como un grafo dirigido, pero siempre tenemos las 2 flechas.
- Este grafo es **ponderado** porque cada arista tiene un peso.

Grafos ponderados

```
vector<pair<int, ll>> grafo[MAXN];

int main(){
  int n, m;
  cin >> n >> m;
  forn(j, m){
    int a, b; cin >> a >> b;
    ll largo; cin >> largo;
    grafo[a].push_back({b, largo}); // agregamos b como vecino de a
                                    // con longitud 'largo'
  }

  // ?
}
```

Dijkstra

- En cada iteración, toma el vértice más cercano de todos los que no procesamos.

Dijkstra

- En cada iteración, toma el vértice más cercano de todos los que no procesamos.
- **Invariante:** antes de la k-ésima iteración calcula correctamente la distancia a los k vértices más cercanos al origen.

Dijkstra

- En cada iteración, toma el vértice más cercano de todos los que no procesamos.
- **Invariante:** antes de la k-ésima iteración calcula correctamente la distancia a los k vértices más cercanos al origen.
- En este punto podemos actualizar la distancia de todos los vecinos del k-ésimo nodo.

Dijkstra

```
vector<ll> dijkstra(int v){
  vector<ll> dist(n, INF);
  dist[v] = 0;
  priority_queue<pair<ll, int>> q;
  q.push({0, v});

  while(not q.empty()){
    // sigue...
  }

    return dist;
}
```

Dijkstra

```
{
  while(not q.empty()){
    auto [d, x] = q.top();
    q.pop();

    if(dist[x] != -d) continue; // ya lo procesamos

    // invariante: la distancia a x es la minima posible

    for(auto [y, largo] : grafo[x])
    {
      if(dist[x] + largo < dist[y]){
        dist[y] = dist[x] + largo;
        q.push({-dist[y], y});
      }
    }
  }
}
```

Complejidad: $\mathcal{O}(m \log m)$.

Bellman-Ford

- Si hay alguna arista de peso negativo, Dijkstra no anda.

Bellman-Ford

- Si hay alguna arista de peso negativo, Dijkstra no anda.
- Bellman-Ford es un algoritmo que resuelve el mismo problema, pero soporta aristas con peso negativo.
- Complejidad: $\mathcal{O}(nm)$.

Bellman-Ford

- Si hay alguna arista de peso negativo, Dijkstra no anda.
- Bellman-Ford es un algoritmo que resuelve el mismo problema, pero soporta aristas con peso negativo.
- Complejidad: $\mathcal{O}(nm)$.
- Hacemos $n-1$ pasadas del algoritmo, y en cada una iteramos por todas las aristas.

Bellman-Ford

- Si hay alguna arista de peso negativo, Dijkstra no anda.
- Bellman-Ford es un algoritmo que resuelve el mismo problema, pero soporta aristas con peso negativo.
- Complejidad: $\mathcal{O}(nm)$.
- Hacemos $n-1$ pasadas del algoritmo, y en cada una iteramos por todas las aristas.
- **Invariante:** luego de la k-ésima iteración, ya tenemos las distancias mínimas mirando todos los caminos de k aristas o menos.

Bellman-Ford

```
vector<ll> bf(int v)
{
  vector<ll> dist(n, INF);
  dist[v] = 0;

  forn(i, n-1)
    for(auto [x, y, peso] : aristas)
      dist[y] = min(dist[y], dist[x] + peso);

  return dist;
}
```

Bellman-Ford

```
vector<ll> bf(int v)
{
  vector<ll> dist(n, INF);
  dist[v] = 0;

  forn(i, n-1)
    forn(x, n)
      for(auto [y, peso] : grafo[x])
        dist[y] = min(dist[y], dist[x] + peso);

  return dist;
}
```

Floyd-Warshall

- Si queremos las distancias entre cada par de nodos, podemos correr un Bellman-Ford que salga de cada nodo: $\mathcal{O}(n^2m)$.

Floyd-Warshall

- Si queremos las distancias entre cada par de nodos, podemos correr un Bellman-Ford que salga de cada nodo: $\mathcal{O}(n^2 m)$.
- Floyd-Warshall calcula todas las distancias en $\mathcal{O}(n^3)$.
- **Invariante:** luego de la k-ésima iteración, tenemos todas las distancias mínimas en caminos que solo pasan por los primeros k nodos.

Floyd-Warshall

```
ll dist[MAXN][MAXN];

void floyd(){
  forn(x, n) forn(y, n) dist[x][y] = INF;
  forn(x, n) dist[x][x] = 0;
  forn(x, n) for(auto [y, peso] : grafo[n])
    dist[x][y] = min(dist[x][y], peso);

  forn(k, n)
    forn(x, n) forn(y, n)
      dist[x][y] = min(dist[x][y], dist[x][k] + dist[k][y]);
}
```

Problema

MST

Dado un grafo ponderado, sin dirigir, con $n \leq 10^5$ nodos y $m \leq 10^5$ aristas, hallar un subgrafo del mismo que incluya a todos los nodos.
Por cada par de nodos debe haber un único camino posible.
Además, la suma de los pesos de las aristas elegidas debe ser lo más chica posible.

Kruskal

- También hay un algoritmo conocido para esto: el algoritmo de Kruskal.
- La idea es (otra vez) simple: vamos agregando las aristas de menor a mayor, siempre que no agreguen un ciclo al árbol que estamos armando.

Kruskal

- También hay un algoritmo conocido para esto: el algoritmo de Kruskal.
- La idea es (otra vez) simple: vamos agregando las aristas de menor a mayor, siempre que no agreguen un ciclo al árbol que estamos armando.
- ¿Cómo chequeamos que una arista no agregue un ciclo?

Kruskal

- También hay un algoritmo conocido para esto: el algoritmo de Kruskal.
- La idea es (otra vez) simple: vamos agregando las aristas de menor a mayor, siempre que no agreguen un ciclo al árbol que estamos armando.
- ¿Cómo chequeamos que una arista no agregue un ciclo? Usando una estructura de datos, que se llama union-find (o DSU).
- Complejidad: $\mathcal{O}(m \, log \, m)$.

Kruskal

```
vector<tuple<ll, int, int>> aristas; // (peso, x, y)

ll kruskal(){
  sort(aristas.begin(), aristas.end());
  ll res = 0;
  for(auto [peso, x, y] : aristas){
    if(find(x) == find(y)) continue;
    // si llegamos aca, x e y estan en componentes
    // distintas. Por lo tanto agregamos la arista
    join(x, y); // marcamos que estan en la misma componente
    res += peso;
  }
  return res;   // costo total del MST
}
```

Extras

- Representar el grafo de otras formas, como lista de aristas o matriz de adyacencia.
- Grafos bipartitos: pueden coloreares de rojo y azul de manera que todas las aristas tienen Un extremo rojo y uno azul.
- Los árboles se pueden pensar con una raíz y subárboles. Esto sirve para hacer DP. A veces es útil representarlos así explícitamente.
- Grafo dirigido acíclico (DAG): es la estructura que aparece de fondo en la DP. Se puede hacer un orden topológico.

Problemas

- https://codeforces.com/contest/893/problem/C
- https://codeforces.com/contest/242/problem/C
- https://codeforces.com/contest/520/problem/B
- https://cses.fi/problemset/task/1667
- https://cses.fi/problemset/task/1671
- https://cses.fi/problemset/task/1669
- https://cses.fi/problemset/task/1672
- https://cses.fi/problemset/task/1673
- https://cses.fi/problemset/task/1195
- https://cses.fi/problemset/task/1675
- https://cses.fi/problemset/task/1676