Grafos

Nivel Inicial

Mariano Feresin

Universidad Tecnológica Nacional - Facultad Regional Santa Fe

Training Camp 2025

Gracias Sponsors!

Organizador

Diamond Plus

Gracias Sponsors!

Platino

FOLDER IT

INTERNATIONAL
SOFTWARE COMPANY

Gold

NeuralSoft

Oro

JS JERÁRQUICOS

Gracias Sponsors!

Aliado

Outline

1. Motivación
2. Definiciones
3. Representaciones
4. Recorridos
5. Camino Mínimo
6. Árbol Generador Mínimo
7. Cierre

Motivación

- ¿Cómo hace Google Maps para decirte cuál es el camino más rápido?
- ¿Como hacemos para eligir qué materias cursar? Algunas tienen correlativas, es decir, hay que hacer una antes que otra.
- ¿O cómo Instagram te muestra cuentas de personas que quizás conozcas?
- Todas esas cosas tienen algo en común: usan *grafos*.
- Un grafo es simplemente una forma de representar cosas (como personas, ciudades, páginas web...) y cómo esas cosas están conectadas entre sí.

Outline

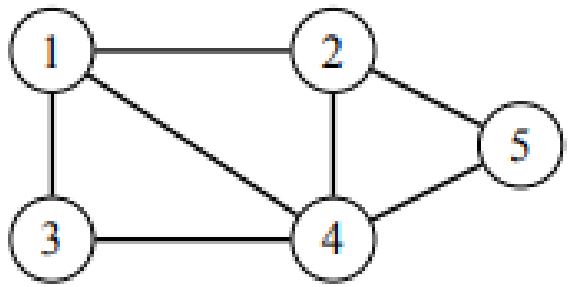
1. Motivación
2. Definiciones
3. Representaciones
4. Recorridos
5. Camino Mínimo
6. Árbol Generador Mínimo
7. Cierre

Grafo

Grafo $G = (V, E)$

- V es un conjunto de vértices, alias nodos.
- Algunos pares de nodos están conectados por aristas, alias ejes. E es el conjunto de aristas.
- **Notación:** n es la cantidad de nodos, m la cantidad de aristas.

Fig. 7.1 A graph with 5 nodes and 7 edges



Caminos y Ciclos

- Un **camino** lleva de un vértice a otro usando aristas del grafo. Por ejemplo, en la figura, $1 \to 3 \to 4 \to 5$. La longitud de un camino es la cantidad de aristas.
- Un **ciclo** es un camino que empieza y termina en el mismo vértice. Ejemplo: $1 \to 3 \to 4$.

Fig. 7.2 A path from node 1 to node 5

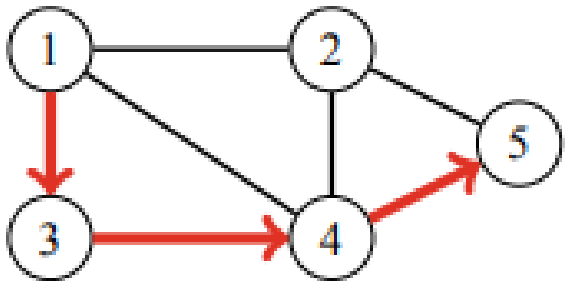
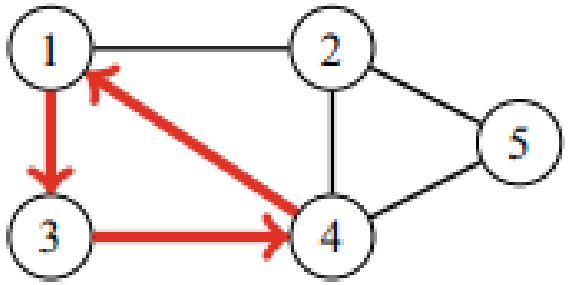


Fig. 7.3 A cycle of three nodes



Conexidad y Componentes

- Decimos que un grafo es **conexo** si existe camino entre todo par de vértices.
- A cada conjunto maximal de vértice que sea conexo le llamamos **componente**.

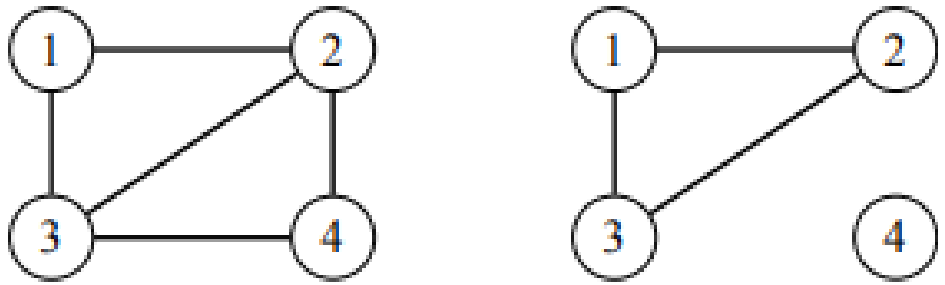


Fig. 7.4 The left graph is connected, the right graph is not

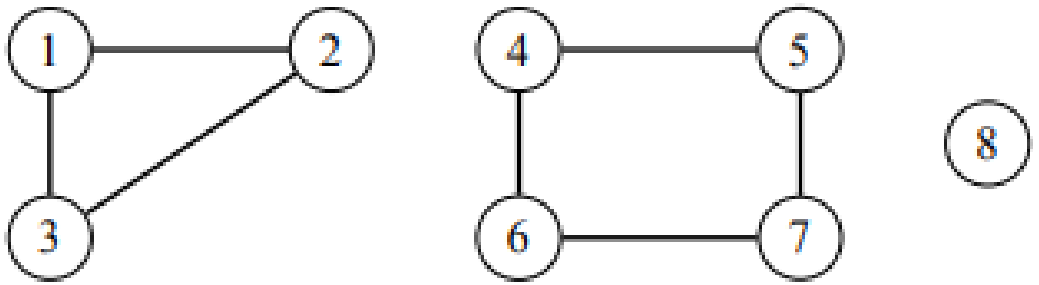


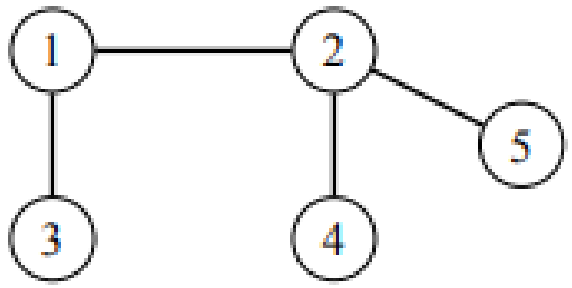
Fig. 7.5 Graph with three components

Árboles

Un árbol es un grafo:

1. Conexo
2. Sin ciclos
3. Con $m = n - 1$

Fig. 7.6 A tree



Otros Tipos de Grafos

- En un grafo **dirigido** las aristas se pueden recorrer en un único sentido.
- En un grafo **pesado**, cada arista tiene un peso asociado.

Fig. 7.7 Directed graph

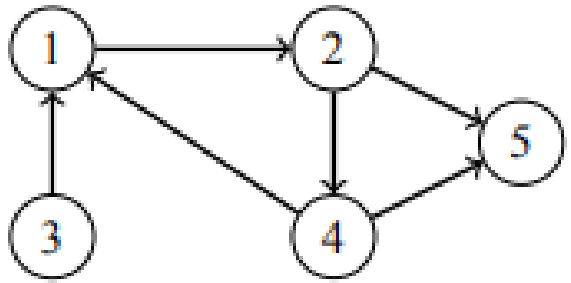
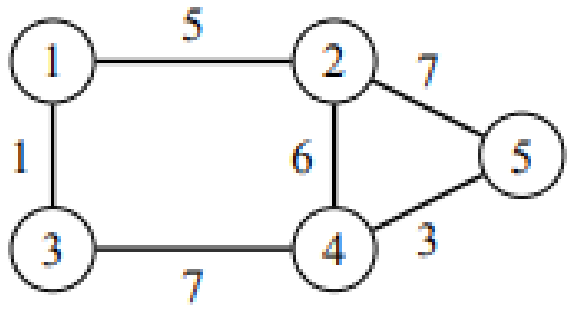


Fig. 7.8 Weighted graph



Outline

1. Motivación
2. Definiciones
3. Representaciones
4. Recorridos
5. Camino Mínimo
6. Árbol Generador Mínimo
7. Cierre

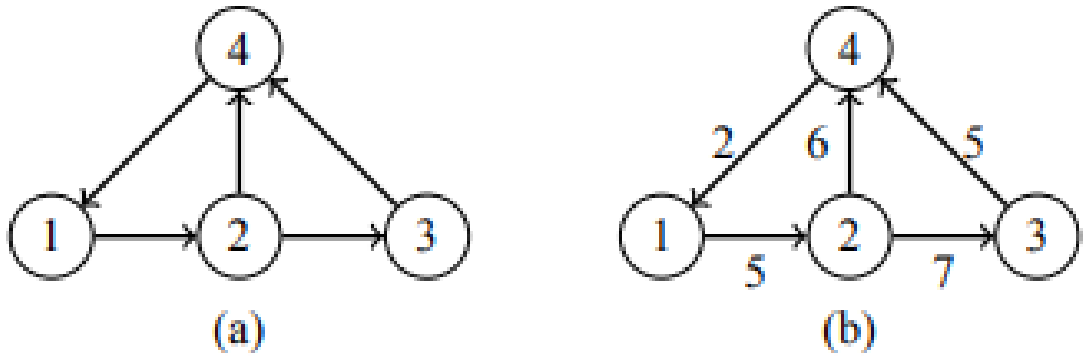
Representaciones Comunes

Para resolver con grafos, hay que representarlos en la computadora. Elegir la representación que sea más conveniente según características del grafo y qué operaciones necesite realizar el algoritmo. Vemos tres representaciones comunes:

1. Lista de adyacencias.
2. Lista de aristas.
3. Matriz de adyacencia.

Lista de Adyacencias

Fig. 7.12 Example graphs



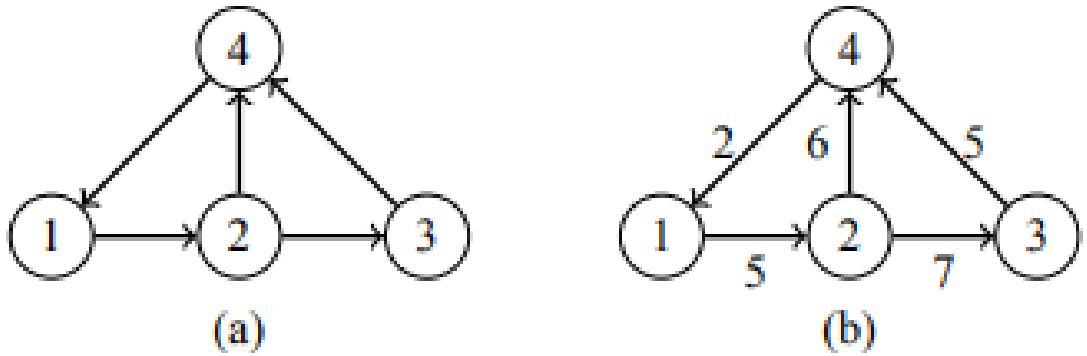

```
vector<int> adj[N];
```

The constant N is chosen so that all adjacency lists can be stored. For example, the graph in Fig. 7.12a can be stored as follows:

```
adj[1].push_back(2);
adj[2].push_back(3);
adj[2].push_back(4);
adj[3].push_back(4);
adj[4].push_back(1);
```

Lista de Adyacencias para Grafo Pesado

Fig. 7.12 Example graphs



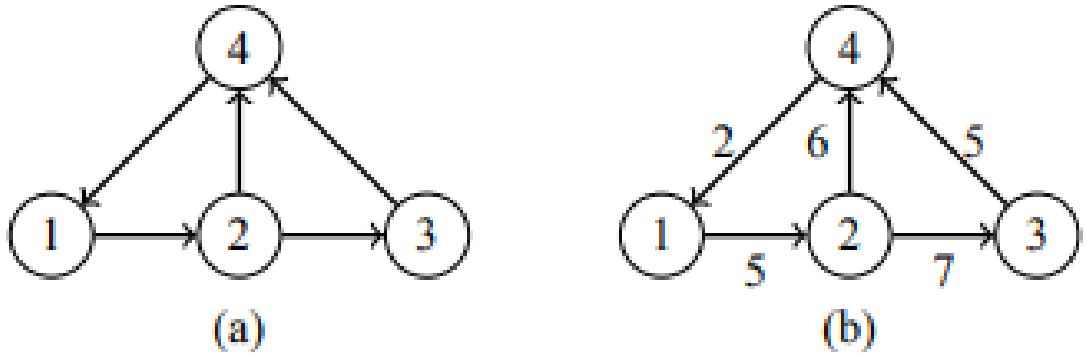

```
vector<pair<int,int>> adj[N];
```

In this case, the adjacency list of node a contains the pair (b, w) always when there is an edge from node a to node b with weight w. For example, the graph in Fig. 7.12b can be stored as follows:

```
adj[1].push_back({2,5});
adj[2].push_back({3,7});
adj[2].push_back({4,6});
adj[3].push_back({4,5});
adj[4].push_back({1,2});
```

Lista de Aristas

Fig. 7.12 Example graphs



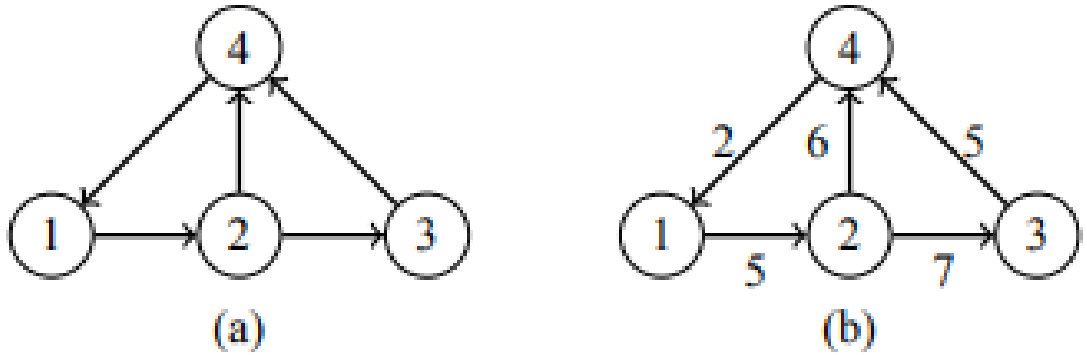

```
vector<pair<int,int>> edges;
```

where each pair (a, b) denotes that there is an edge from node a to node b. Thus, the graph in Fig. 7.12a can be represented as follows:

```
edges.push_back({1,2});
edges.push_back({2,3});
edges.push_back({2,4});
edges.push_back({3,4});
edges.push_back({4,1});
```

Lista de Aristas para Grafo Pesado

Fig. 7.12 Example graphs



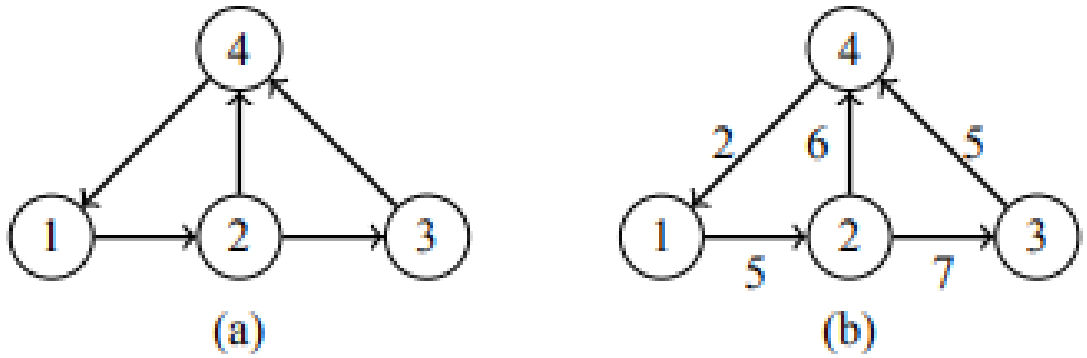

```
vector<tuple<int,int,int>> edges;
```

Each element in this list is of the form (a, b, w), which means that there is an edge from node a to node b with weight w. For example, the graph in Fig. 7.12b can be represented as follows[1]:

```
edges.push_back({1,2,5});
edges.push_back({2,3,7});
edges.push_back({2,4,6});
edges.push_back({3,4,5});
edges.push_back({4,1,2});
```

Matriz de Adyacencia

Fig. 7.12 Example graphs



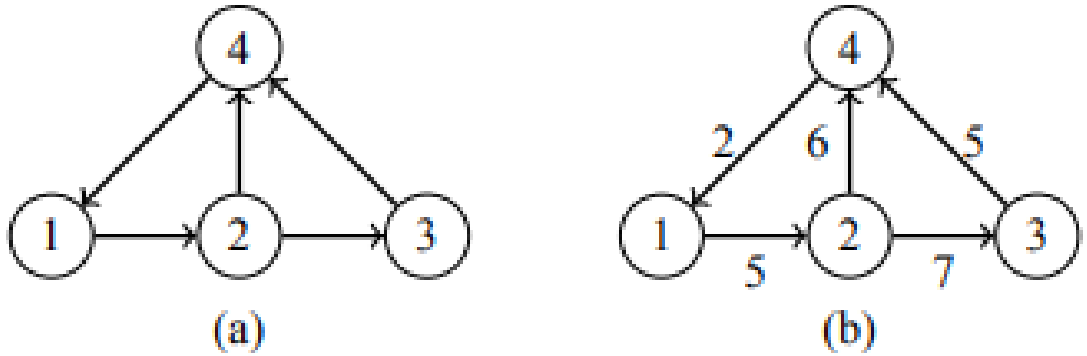

```
int adj[N][N];
```

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$ady[u][v] = 1$ si y sólo si hay arista de vértice u a v.

Matriz de Adyacencia para Grafos Pesados

Fig. 7.12 Example graphs




```
int adj[N][N];
```

$$\begin{bmatrix} 0 & 5 & 0 & 0 \\ 0 & 0 & 7 & 6 \\ 0 & 0 & 0 & 5 \\ 2 & 0 & 0 & 0 \end{bmatrix}$$

Outline

1. Motivación
2. Definiciones
3. Representaciones
4. Recorridos
5. Camino Mínimo
6. Árbol Generador Mínimo
7. Cierre

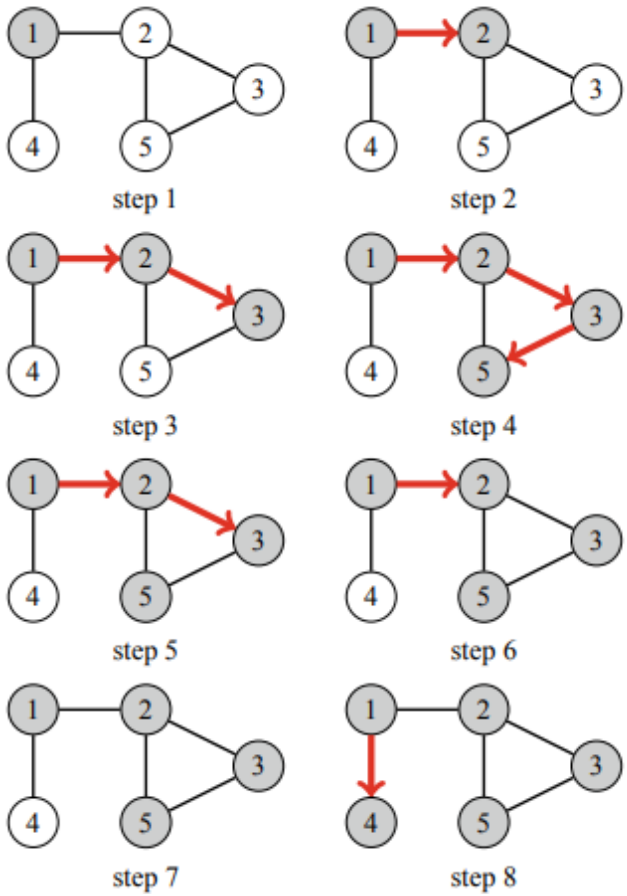
Recorridos

Vemos dos algoritmos fundamentales de grafos: depth-first search (DFS) y breadth-first search (BFS). Ambos recorren todos los nodos que pueden ser alcanzados desde un v inicial, pero en distinto orden.

DFS

- Recorrido “en profundidad”.
- Sigue por un camino mientras encuentre nodos sin explorar. Luego, vuelve a nodos previos a explorar otras partes del grafo.

Fig. 7.13 Depth-first search



Código DFS

```
void dfs(int s) {
    if (visited[s]) return;
    visited[s] = true;
    // process node s
    for (auto u: adj[s]) {
        dfs(u);
    }
}
```

El grafo es representado por *adj*, una lista de adyacencias. Visited es un vector de booleanos.

Problema

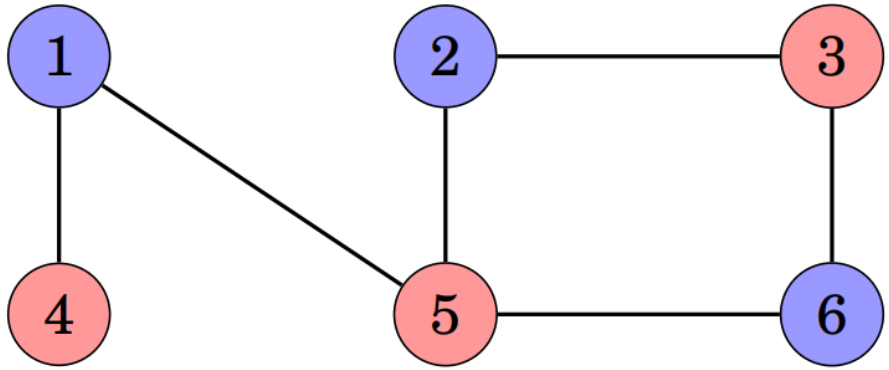
En un curso de programación, los estudiantes quieren organizar un partido de fútbol. Para que el partido sea justo, deben dividirse en dos equipos con la misma cantidad de jugadores.
Sin embargo, hay una regla:

- Si dos jugadores comparten una misma habilidad (por ejemplo, son buenos en defensa o patean con la zurda), no pueden estar en el mismo equipo.

La pregunta es: ¿Es posible formar los equipos de manera que todos puedan jugar respetando esa regla?

Colorear un grafo

- Colorear un grafo es asignar a cada nodo un *color* de manera que no existan dos nodos adyacentes del mismo color.
- Un grafo es **bipartito** si es posible colorear el grafo usando dos colores.
- Un grafo es bipartito cuando no contiene un ciclo con una cantidad impar de aristas



Código Bipartito

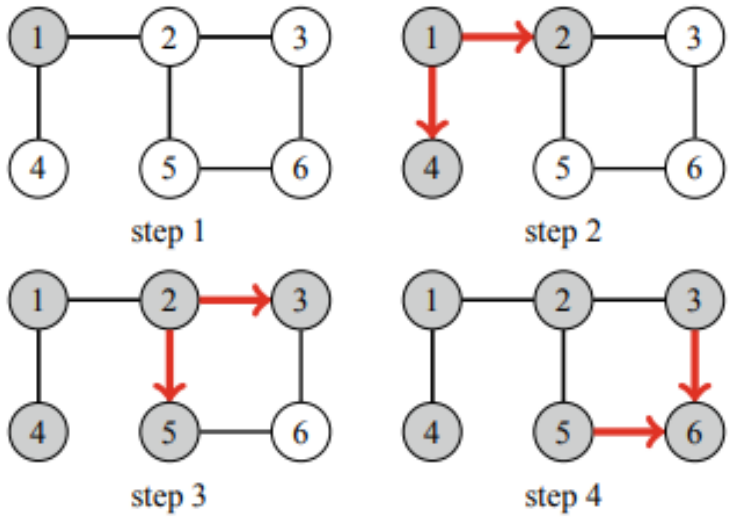
Para saber si un grafo es bipartito utilizamos DFS, asignamos un primer nodo con un color arbitrario, a los vecinos de este el color contrario y llamamos recursivamente. Si nos encontramos con un vecino del mismo color el grafo no es bipartito.

```
vector<vector<int>> g; //lista de adyacencia
/*
Inicializamos el color de cada nodo en -1
indicando que no lo hemos visitando aun,
luego asignamos 0 a un grupo y 1 al otro
*/
vector<int> color;
bool esBipartito(int nodo, int c){
    color[nodo] = c;
    bool bipartito = true;
    for(auto vecino: g[nodo]){
        if(color[vecino] == c)
            return false;
        if(color[vecino] == -1)
            bipartito = min(bipartito, esBipartito(vecino, c ^ 1));
    }
    return bipartito;
}
```

BFS

- Recorrido “a lo ancho”.
- Va explorando el grafo en orden creciente de distancia desde un origen.

Fig. 7.14 Breadth-first search



Código BFS

```
queue<int> q;
bool visited[N];
int distance[N];
```

```
visited[x] = true;
distance[x] = 0;
q.push(x);
while (!q.empty()) {
    int s = q.front(); q.pop();
    // process node s
    for (auto u : adj[s]) {
        if (visited[u]) continue;
        visited[u] = true;
        distance[u] = distance[s]+1;
        q.push(u);
    }
}
```

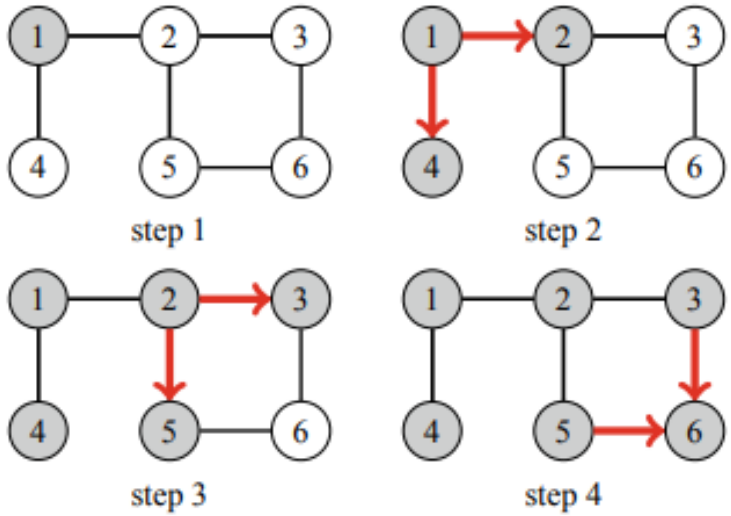
Outline

1. Motivación
2. Definiciones
3. Representaciones
4. Recorridos
5. Camino Mínimo
6. Árbol Generador Mínimo
7. Cierre

Miremos un poco

- Volvamos al ejemplo del recorrido BFS

Fig. 7.14 Breadth-first search



- ¿Qué tienen en común los nodos 2 y 4?
- ¿Y los nodos 3 y 5?

Camino Mínimo

- Un problema recurrente es encontrar el camino de mínimo costo entre dos vértices.
- Hoy menciono cuatro algoritmos para resolver este problema:
  - BFS,
  - Bellman Ford,
  - Dijkstra y
  - Floyd-Warshall.

Bellman Ford

- En grafos pesados necesitamos otros algoritmos.
- Bellman Ford calcula camino mínimo desde un vértice hacia todos.
- Si hay ciclos negativos, el problema del camino mínimo se indefine. Bellman Ford detecta si hay un ciclo negativo.
- **Invariante:** luego de su k-ésima iteración, calcula la distancia mínima a todo v usando a lo sumo k aristas[1].
- **Complejidad:** $O(nm)$.

[1] En realidad, no exactamente...

Código Bellman Ford

```
for (int i = 1; i <= n; i++) {
    distance[i] = INF;
}
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

Detectando ciclos negativos

- ¿Cómo detectamos un ciclo negativo?
- **Intuición:** Sabemos que si no hay ciclos negativos, un camino minimo tiene a los sumo $n-1$ aristas, es decir, como mucho pasa 1 vez por cada nodo del grafo, si tendria más aristas significaría que pasamos dos veces por un nodo lo cual no seria óptimo.
- Una vez que terminamos las $n-1$ iteraciones de Bellman-Ford, hacemos una pasada extra.
- Si en esa última pasada alguna arista puede seguir relajándose, significa que hay un ciclo con peso negativo
- Podríamos seguir mejorando distancias infinitamente. Esa es la señal de que existe un ciclo negativo.

```
bool hayCicloNegativo(){
    for(auto [a, b, w]: edges){
        if(distance[b] > distance[a] + w)
            return true;
    }
    return false;
}
```

Dijkstra

- Camino mínimo de uno a todos.
- Requiere que el costo de las aristas sea no negativo.
- **Invariante:** antes de la k-ésima iteración, calcula correctamente la distancia hacia los k vértices más cercanos al origen.
- En cada iteración, toma al vértice no procesado que esté a distancia mínima del origen.
- **Complejidad:** $O(min\{n^2, m\, lg\, n\})$.

Código Dijkstra

```
vector<vector<pair<int,int>>> G; // lista de adyacencia
vector<int> dist(N, INF);
// usamos una lista de padres para reconstruir el camino
vector<int> padre(N, -1);
priority_queue<pair<int,int>> Q;
Q.push(make_pair(0, origen)), dist[origen] = 0;
while(not Q.empty()){
    int nodo = Q.top().second;
    int d = -Q.top().first; // distancia hasta nodo
    Q.pop();
    if (d > dist[node]) continue;
    for(auto [peso, vecino]: G[nodo]){
        if(d + peso < dist[vecino]){
            dist[vecino] = d + peso;
            padre[vecino] = nodo;
            //un truco es guardar las distancias en la pq en negativo
            //para tener la menor distancia en el "top" de la pq
            Q.push({-dist[vecino], vecino});
        }
    }
}
```

Floyd-Warshall

- Computa distancia entre todo par de vértices.
- **Invariante:** luego de la k-ésima iteración, computa los caminos k-internos[2] mínimos.
- **Complejidad:** $O(n^3)$.

[2]Decimos que un camino es k-interno si todos los vértices intermedios (o sea, exluyendo al primero y al último) tienen un índice menor o igual a k

Código Floyd-Warshall

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (i == j) distance[i][j] = 0;
        else if (adj[i][j]) distance[i][j] = adj[i][j];
        else distance[i][j] = INF;
    }
}
```

```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            distance[i][j] = min(distance[i][j],
                                 distance[i][k]+distance[k][j]);
        }
    }
}
```

Tips

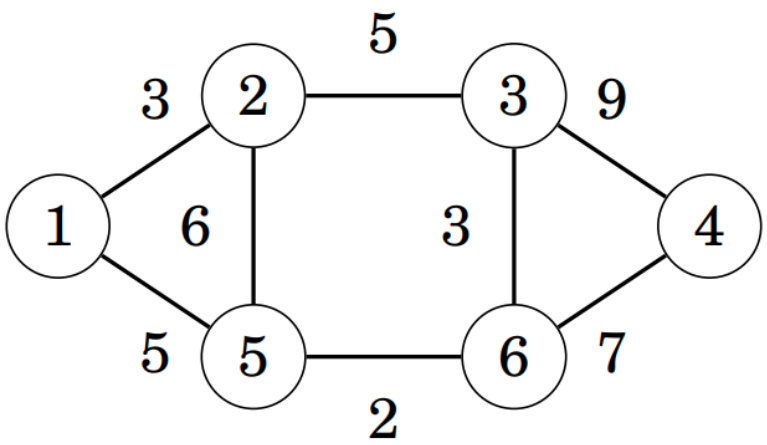
- Los algoritmos de 1 a todos se pueden transformar en algoritmos para encontrar caminos de todos a uno (revirtiendo el sentido de los ejes).
- Se pueden usar para computar caminos de todos a todos (haciendo n ejecuciones del algoritmo, una desde cada origen).
- Pueden usar los algoritmos para calcular distancias y también para obtener el árbol de caminos mínimos. Ojo, puede haber varios caminos mínimos, sólo alguno pertenece a este árbol.
- Si sienten que les falta información en el grafo, consideren agregar vértices para codificar esa información faltante.
- Se pueden correr estos algoritmos teniendo mas de un nodo origen

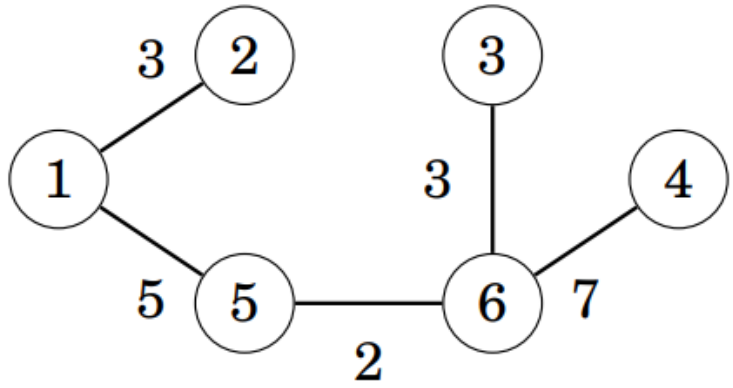
Outline

1. Motivación
2. Definiciones
3. Representaciones
4. Recorridos
5. Camino Mínimo
6. Árbol Generador Mínimo
7. Cierre

Árbol Generador Mínimo

- Otro problema recurrente.
- Un árbol generador (AG) contiene a todos los nodos del grafo, a algunas $(n-1)$ de sus aristas y conecta a todos los vértices.
- El costo de un AG es la suma de los costos de las aristas que contiene.





Union Find

- Si bien esta no es una clase de estructura de datos, la estructura Union Find va a ser útil para encontrar el árbol generador mínimo de un grafo.
- Union Find es una estructura que mantiene una colección de conjuntos. Los conjuntos son disjuntos, es decir, ningun elemento pertenece a más de un conjunto.
- Cada conjunto tine un elemento que es el representante de este y hay una cadena desde cada elemento del conjunto hasta el representante.

Union Find

- Si bien esta no es una clase de estructura de datos, la estructura Union Find va a ser útil para encontrar el árbol generador mínimo de un grafo.
- Union Find es una estructura que mantiene una colección de conjuntos. Los conjuntos son disjuntos, es decir, ningun elemento pertenece a más de un conjunto.
- Cada conjunto tine un elemento que es el representante de este y hay una cadena desde cada elemento del conjunto hasta el representante.

Union Find

Union Find soporta dos tipos de operaciones:

- *unite* : nos permite unir dos conjuntos
- *find* : nos permite encontrar el representante de un elemento
- No vamos a analizar la complejidad de estas operaciones pero pueden asumir que son prácticamente constantes.[3]

[3] Para los curiosos pueden más leer sobre esto acá: codeforces.com/blog/entry/98275

Código Union Find

- Encontrar el representante de un elmento es simplemente recorrer su cadena de representantes
- Para unir dos conjuntos conectamos el reprensentante del conjunto mas chico al representante del conjunto mas grande para así lograr que la estructura sea eficiente.

```
int find(int u){
    if(repre[u] != u) {
        repre[u] = find(repre[u]);
    }
    return repre[u];
}
```

```
void unite(int u, int v){
    u = find(u);
    v = find(v);
    if(u == v)
        return;
    if(tam[u] < tam[v])
        swap(u, v);
    repre[v] = u;
    tam[u] += tam[v];
}
```

Implementación Kruskal

Ahora que conocemos la estructura Union Find podemos ver como encontrar el árbol generador mínimo de un grafo.

- Se ordenan las arista por costo, de menor a mayor.
- Se procesa cada arista u, v: si u y v están en distintas componentes, se agrega la arista al bosque.
- Para responder eficientemente si están en la misma componente, usamos la estructura union-find.

```
//ordenamos las aristas por peso de menor a mayor
sort(aristas.begin(),aristas.end());
for(auto [peso,a,b]: aristas){
    if(find(a) != find(b)){
        //encontramos una arista que pertenece al árbol
        unite(a,b);
    }
}
```

Outline

1. Motivación
2. Definiciones
3. Representaciones
4. Recorridos
5. Camino Mínimo
6. Árbol Generador Mínimo
7. Cierre

Referencias

Les dejo el libro Guide to Competitive Programming, de Antti Laaksonen.

- https://cses.fi/book/book.pdf

En la sección de grafos van a poder ver mas sobre los temas que vimos hoy y chusmear mas cosas sobre grafos.

Problemas para practicar

- Les dejo problemas de CSES:
  - `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`

Consultas

Pueden consultarme durante estas semanas, o me pueden escribir al Telegram:

- @marianoferesin

Saludo

Gracias a todos por su atención y mucha suerte en el contest de la tarde.