Grafos

Luis Santiago Re[1]

[1]Universidad Tecnológica Nacional - FRSF

Training Camp 2020

Definición

Grafo [Wikipedia]

Un grafo es un conjunto de objetos llamados **vértices** o **nodos**, unidos por enlaces llamados **aristas** o **arcos**.

Definición

Grafo [Wikipedia]

Un grafo es un conjunto de objetos llamados **vértices** o **nodos**, unidos por enlaces llamados **aristas** o **arcos**.

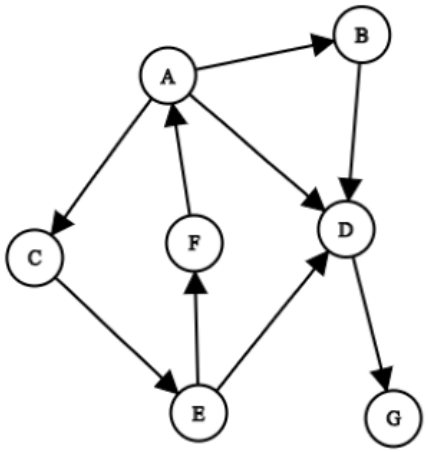


Figure: Grafo dirigido

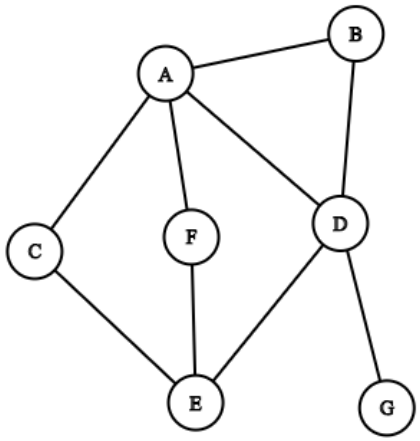


Figure: Grafo no dirigido

Ejemplos

Ejemplo: redes sociales

- En Facebook, dos usuarios, pueden o no ser amigos. Esta relación es mutua, es decir que puede pensarse como un **grafo no dirigido**, en donde cada usuario es un nodo y cada relación de amistad es una arista.
- En Instagram, un usuario puede seguir a otro, sin necesidad que ese otro usuario lo siga a uno. Es decir que puede pensarse como un **grafo dirigido**, en donde cada usuario es un nodo y cada relación seguidor-seguido es una arista.

En general, dada cualquier situación en la que tenemos **pares** de cosas relacionadas, probablemente sirve verla como un grafo.

Representación

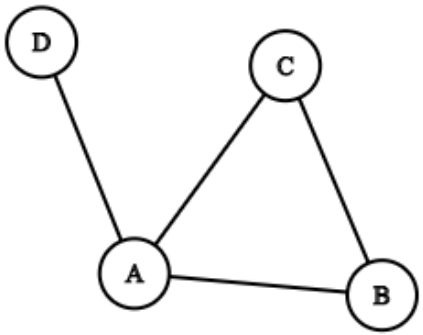
Matriz de adyacencia

La **matriz de adyacencia** es una matriz de $n \times n$ donde n es la cantidad de nodos del grafo, que en la posición (i, j) tiene un 1 (o true) si hay una arista entre los nodos i y j, o 0 (o false) en caso contrario. Uso de memoria: $O(n^2)$

Representación

Matriz de adyacencia

La **matriz de adyacencia** es una matriz de $n \times n$ donde n es la cantidad de nodos del grafo, que en la posición (i, j) tiene un 1 (o true) si hay una arista entre los nodos i y j, o 0 (o false) en caso contrario. Uso de memoria: $O(n^2)$



$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Implementación matriz de adyacencia

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    int n,m;
6    cin>>n>>m;
7    int graph[n][n] = { 0 };
8    for(int i=0;i<m;i++){
9      int a,b;
10     cin>>a>>b;
11     graph[a][b] = 1;
12     graph[b][a] = 1;
13   }
14 }
```

Representación

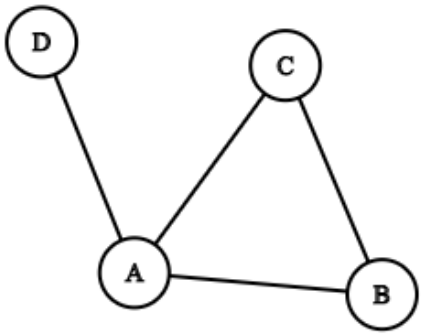
Listas de adyacencia

La **lista de adyacencia** es un vector de vectores de enteros, que en el i-ésimo vector tiene el número j si hay una arista entre los nodos i y j. Uso de memoria: $O(m)$, donde m es la cantidad de aristas.

Representación

Listas de adyacencia

La **lista de adyacencia** es un vector de vectores de enteros, que en el i-ésimo vector tiene el número j si hay una arista entre los nodos i y j. Uso de memoria: $O(m)$, donde m es la cantidad de aristas.



A :	{B,C,D}
B :	{A,C}
C :	{A,B}
D :	{A}

Implementaciones lista de adyacencia

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  vector<int> graph[10000];
6  // voy a tener n<=10000
7  int n,m;
8
9  int main(){
10   cin>>n>>m;
11   for(int i=0;i<m;i++){
12     int a,b;
13     cin>>a>>b;
14     graph[a].push_back(b);
15     graph[b].push_back(a);
16   }
17 }
```

Implementaciones lista de adyacencia

```cpp
#include <iostream>
#include <vector>
using namespace std;

vector<int> graph[10000];
// voy a tener n<=10000
int n,m;

int main(){
  cin>>n>>m;
  for(int i=0;i<m;i++){
    int a,b;
    cin>>a>>b;
    graph[a].push_back(b);
    graph[b].push_back(a);
  }
}
```

```cpp
#include <iostream>
#include <vector>
using namespace std;

vector<vector<int > > graph;
int n,m;

int main(){
cin>>n>>m;
graph.resize(n);
  for(int i=0;i<m;i++){
    int a,b;
    cin>>a>>b;
    graph[a].push_back(b);
    graph[b].push_back(a);
  }
}
```

Algunas definiciones

Definiciones: vecino y grado

Dada una arista que conecta dos vértices u y v, decimos que u es **vecino** de v (y que v es vecino de u). Además, a la cantidad de vecinos de u se le llama el **grado** de u.

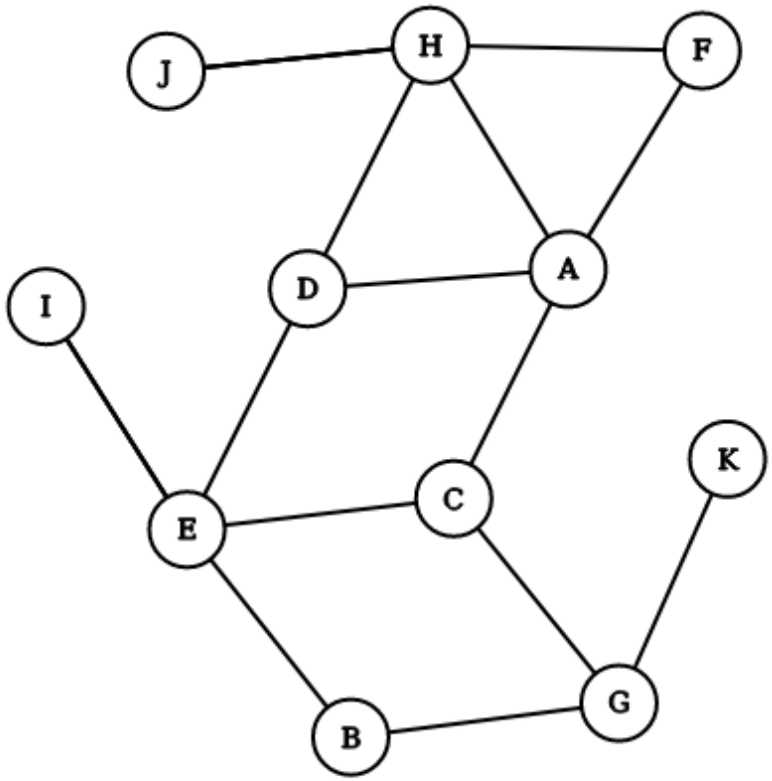
Definición: distancia

Definimos la **distancia** de u a v como el **menor** n tal que hay un camino de largo n de u a v.

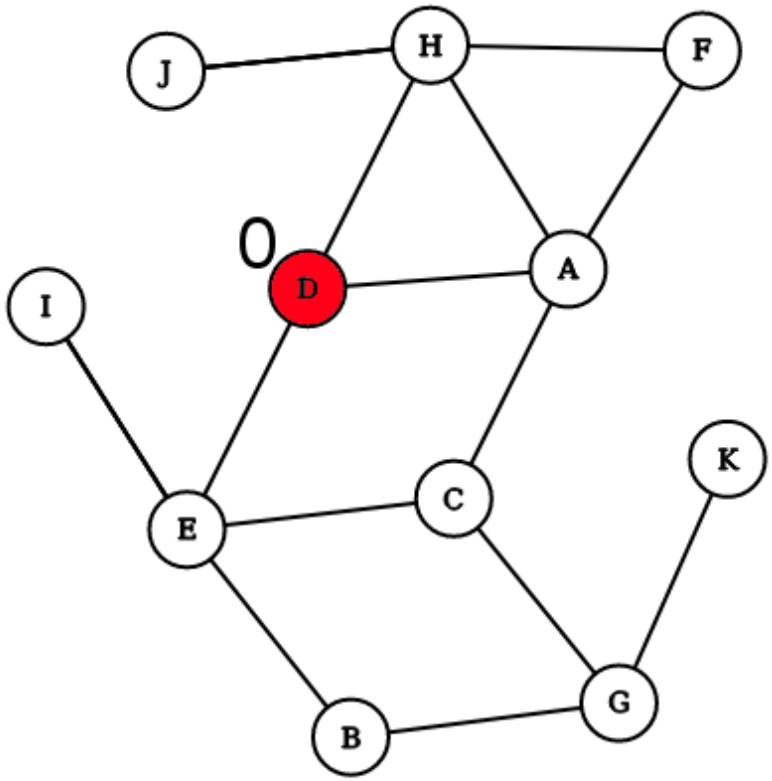
BFS (siglas de *breadth-first-search* o "búsqueda en anchura") es un algoritmo para recorrer grafos, que a su vez sirve para calcular la **distancia mínima** desde un nodo v a cada uno de los otros.

- Inicialmente se procesa v, que tiene distancia a sí mismo 0, por lo que seteamos $d_v = 0$. Además es el único a distancia 0.
- Los vecinos de v tienen sí o sí distancia 1, por lo que seteamos $d_y = 1$ para todo y vecino de v. Además, estos son los únicos a distancia 1.
- Ahora tomamos los nodos a distancia 1, y para cada uno nos fijamos en sus vecinos cuya distancia aún no calculamos. Estos son los nodos a distancia 2.
- Así sucesivamente, para saber cuáles son los nodos a distancia $k+1$, tomamos los vecinos de los nodos a distancia k que no hayan sido visitados aún.

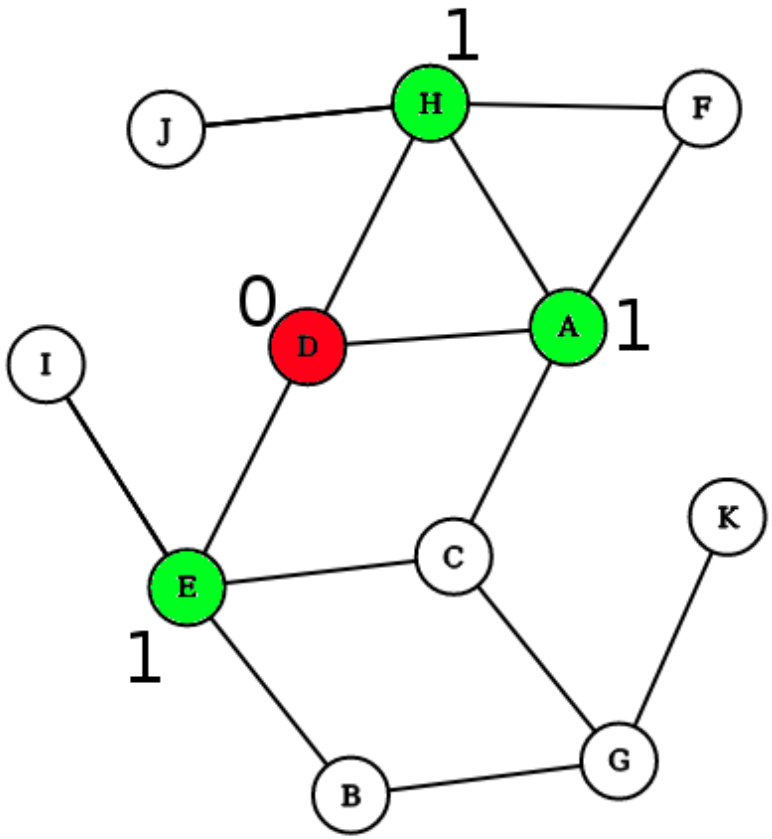
Ejemplo



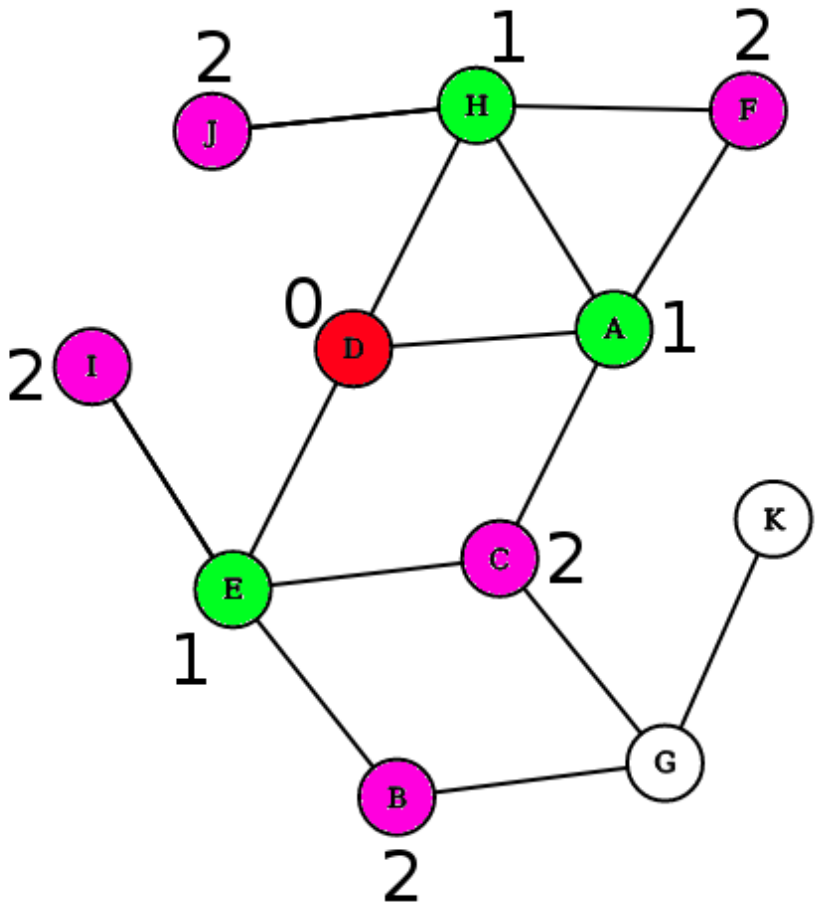
Ejemplo



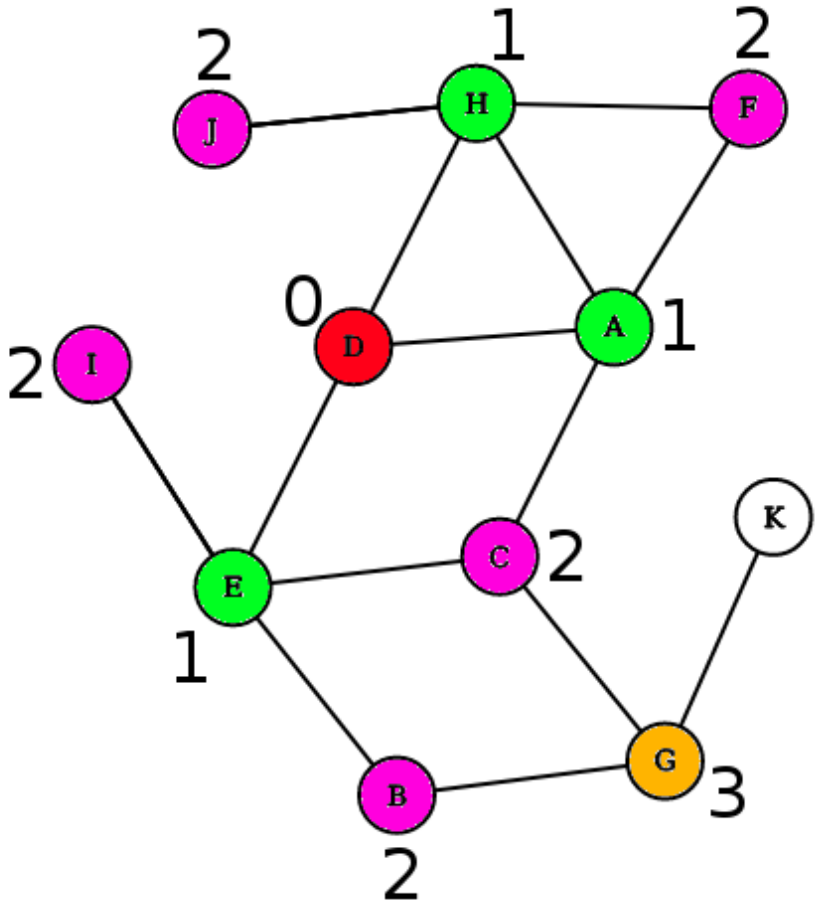
Ejemplo



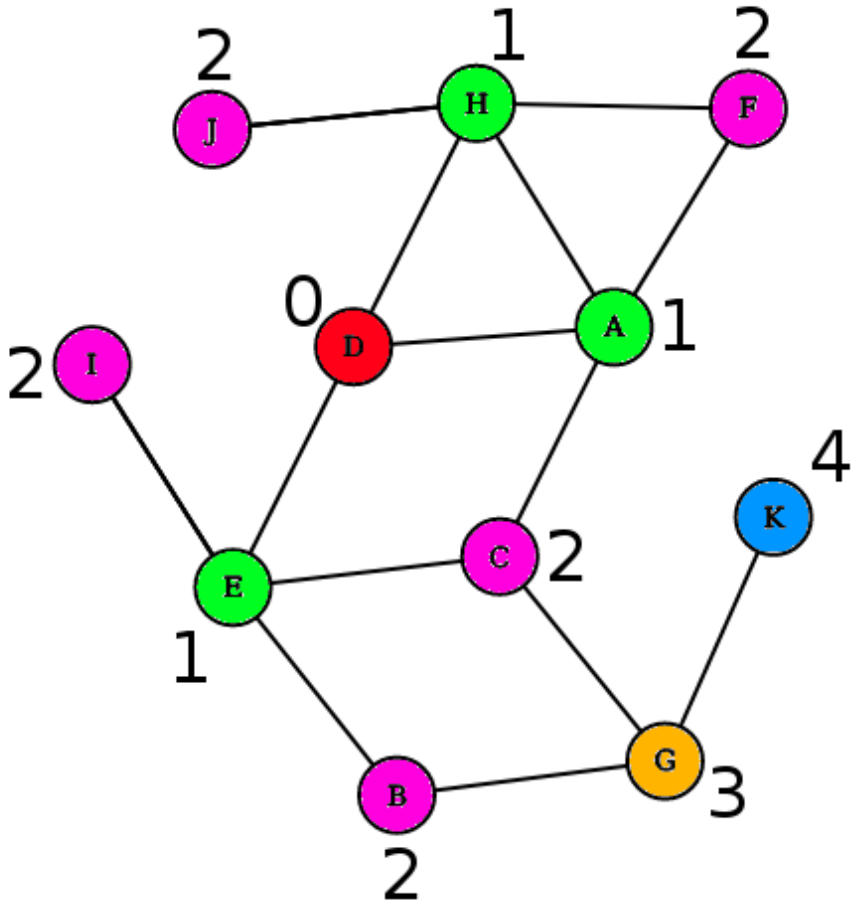
Ejemplo



Ejemplo



Ejemplo



Implementación

```
1  vector<int> bfs(vector<vector<int>>& graph , int v) {
2    vector<int> d(graph.size(), -1);
3    queue<int> q;
4    d[v] = 0;
5    q.push(v);
6    while(!q.empty()) { // mientras haya nodos por procesar
7      int x = q.front();
8      q.pop();
9      for(int y: graph[x]) { // para cada y vecino de x
10       if(d[y] == -1) { // si y no fue encolado aun
11         d[y] = d[x] + 1;
12         q.push(y);
13       }
14     }
15   }          // d contiene las distancias desde v
16   return d; // (o -1 para los nodos inalcanzables)
17 }
```

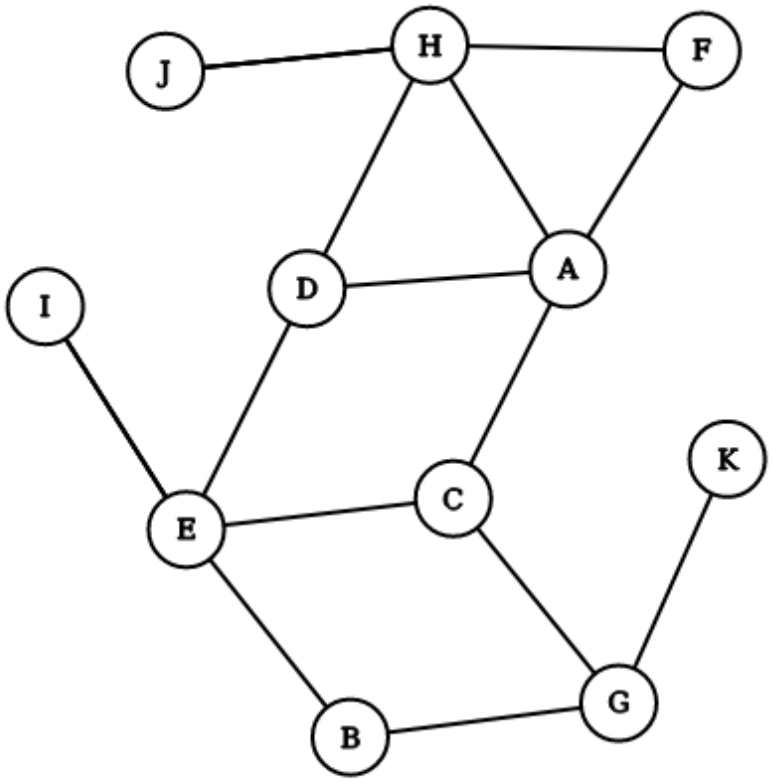
Análisis

- **Cada nodo** lo procesamos una sóla vez, porque cuando lo agregamos a la *queue*, también inicializamos su distancia, por lo que no volverá a ser agregado.
- Como cada nodo es procesado una sóla vez, entonces **cada arista** es procesada una sóla vez (o una vez en cada sentido para no-dirigidos).
- Entonces, la complejidad de BFS es $O(m)$, donde m es la cantidad de aristas.

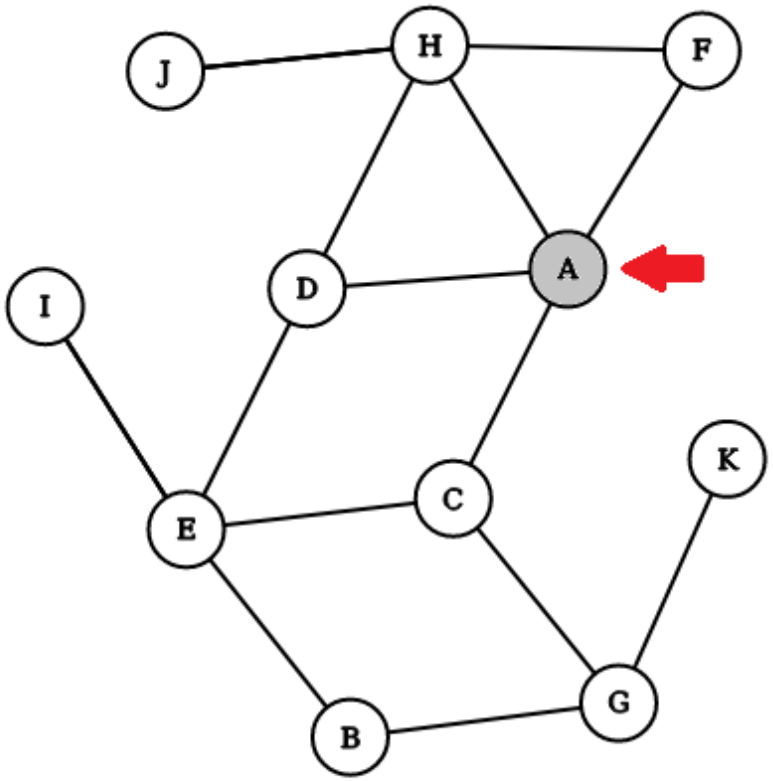
Otra manera de recorrer un grafo es con DFS (*depth-first-search* o "búsqueda en profundidad"). Funciona así:

- Arrancamos de cierto nodo y lo marcamos como visitado.
- Luego, evaluamos sus vecinos de a uno, y cada vez que encontramos uno que no esté marcado como visitado, seguimos procesando a partir de él. Este procedimiento es **recursivo**, por lo que se puede implementar así.
- Cuando no quedan vecinos por visitar salgo para atrás (vuelvo en la recursión).
- Cuando no se conoce el máximo del **stack de recursión** que usa nuestro juez, no es mala idea implementar el DFS con nuestro propio stack aunque sea más código.

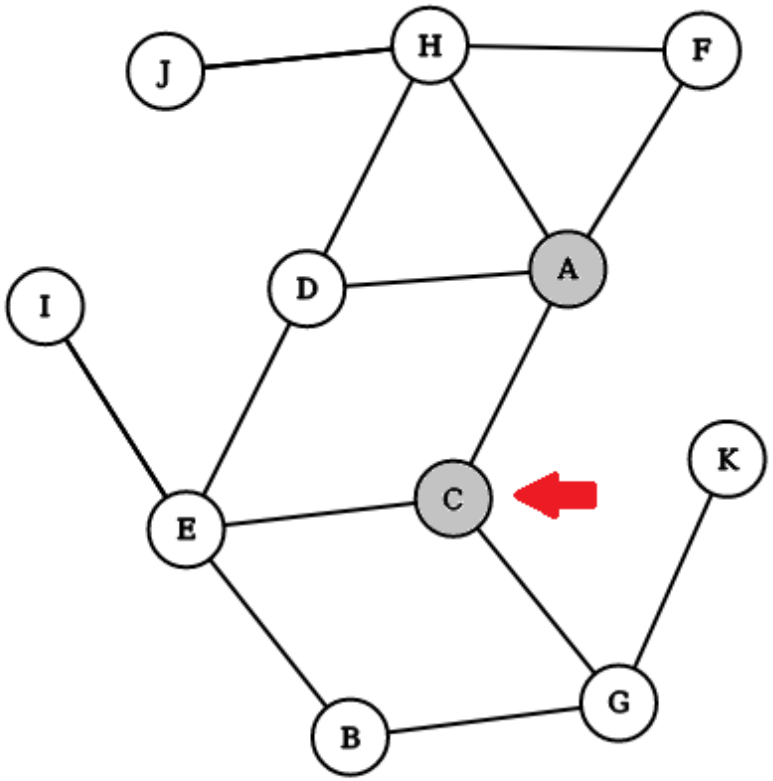
Ejemplo



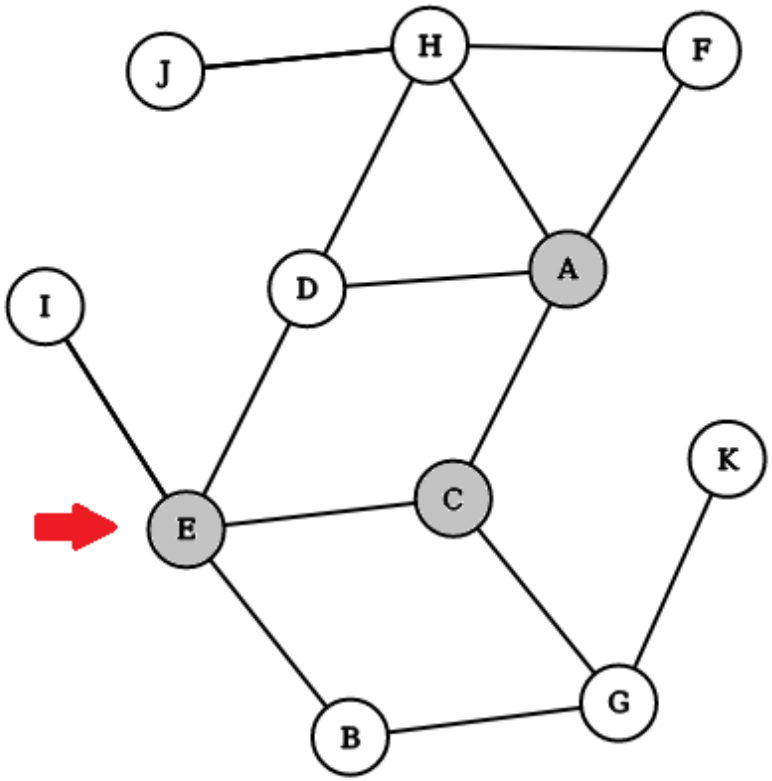
Ejemplo



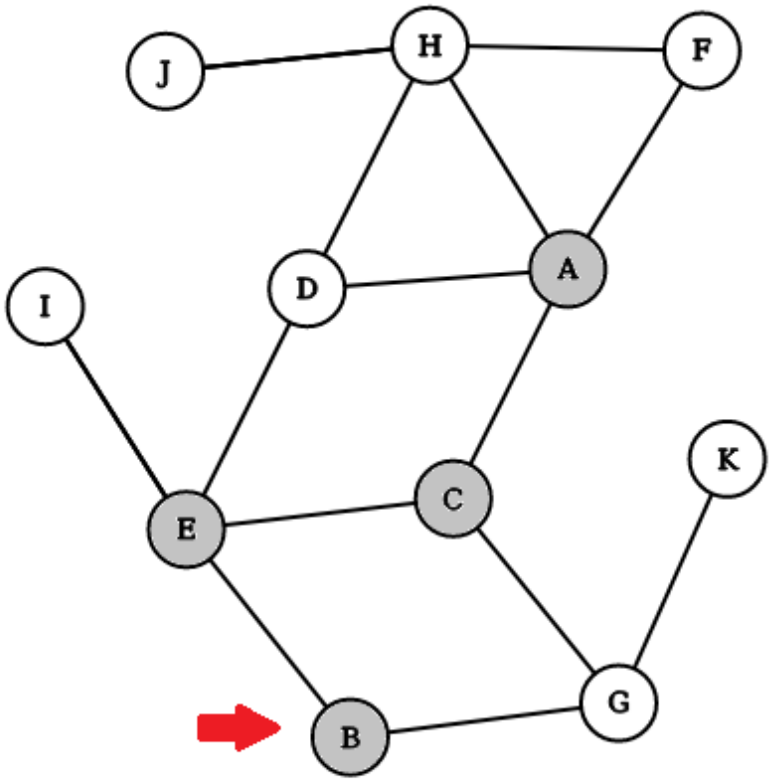
Ejemplo



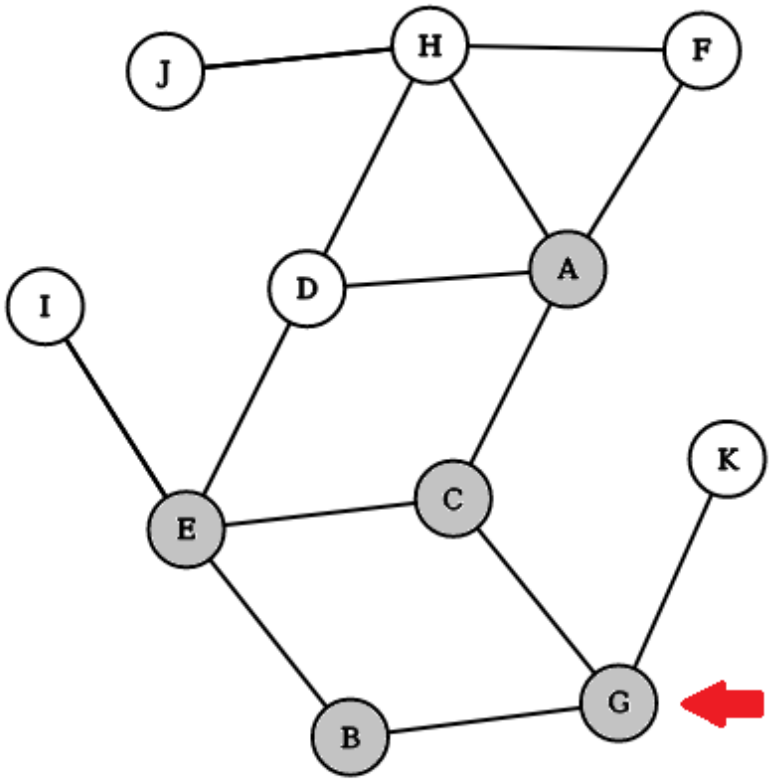
Ejemplo



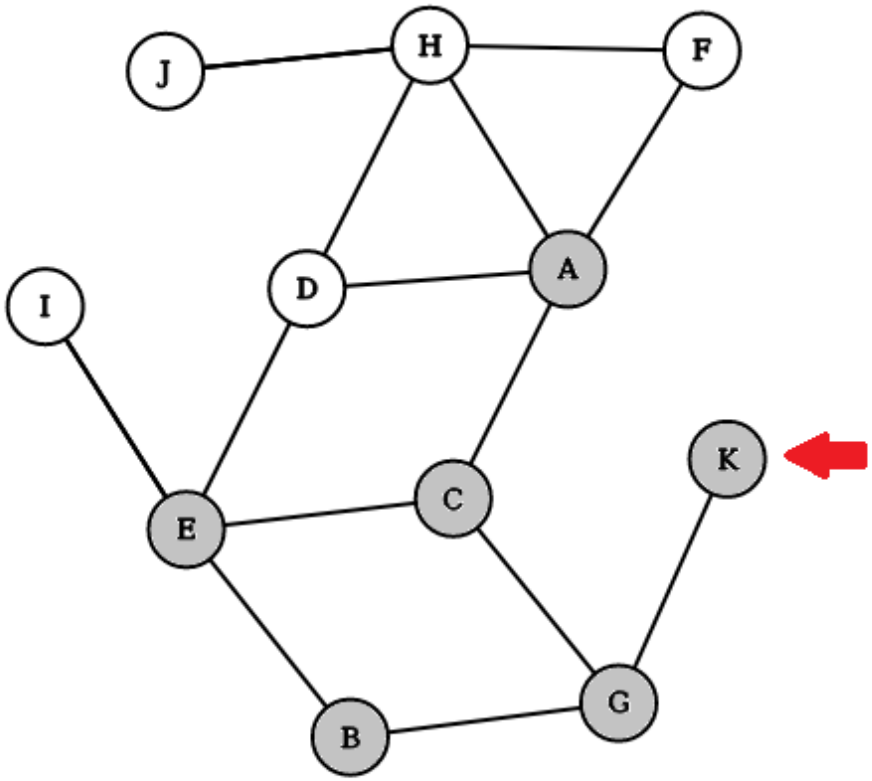
Ejemplo



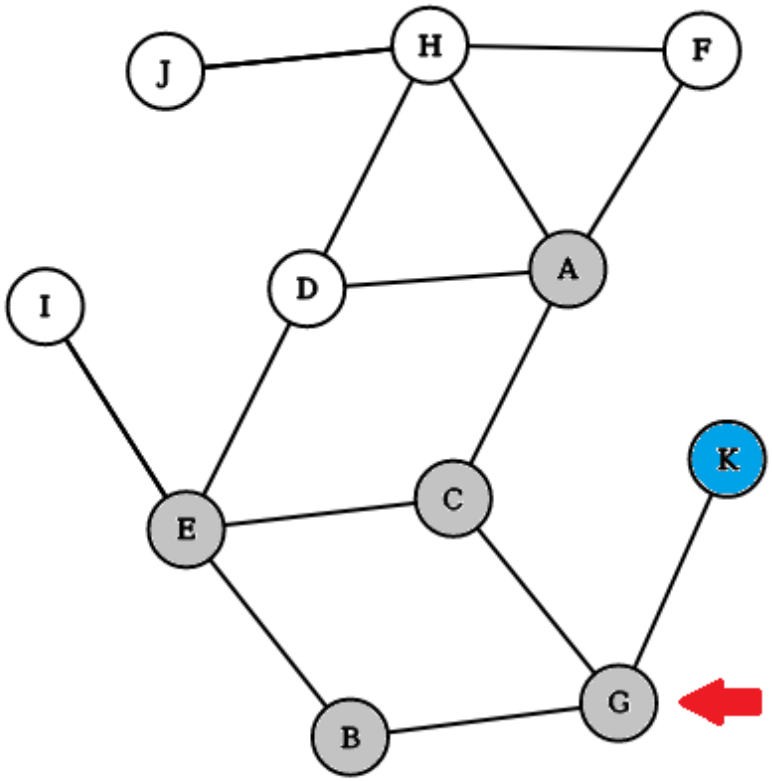
Ejemplo



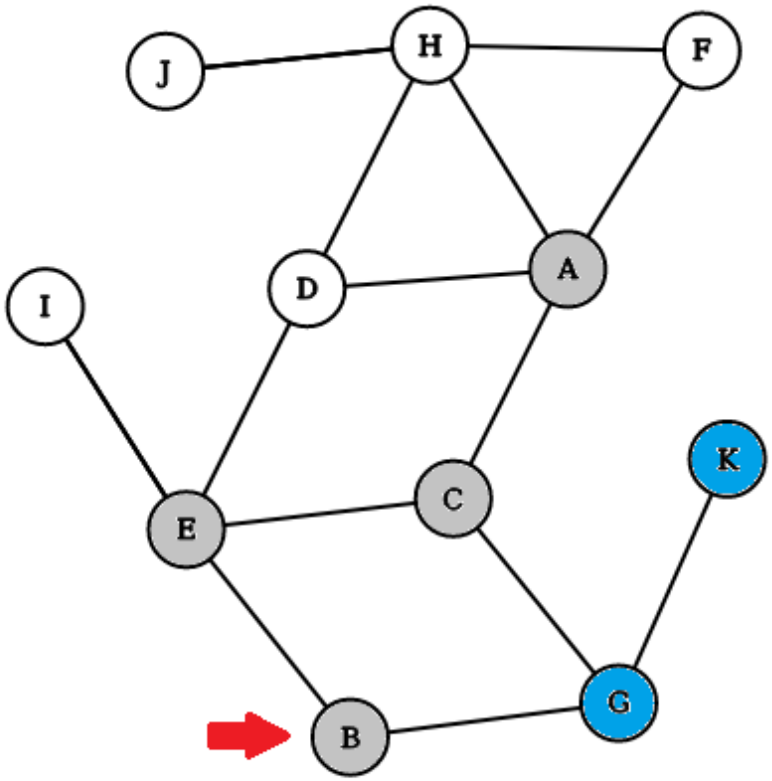
Ejemplo



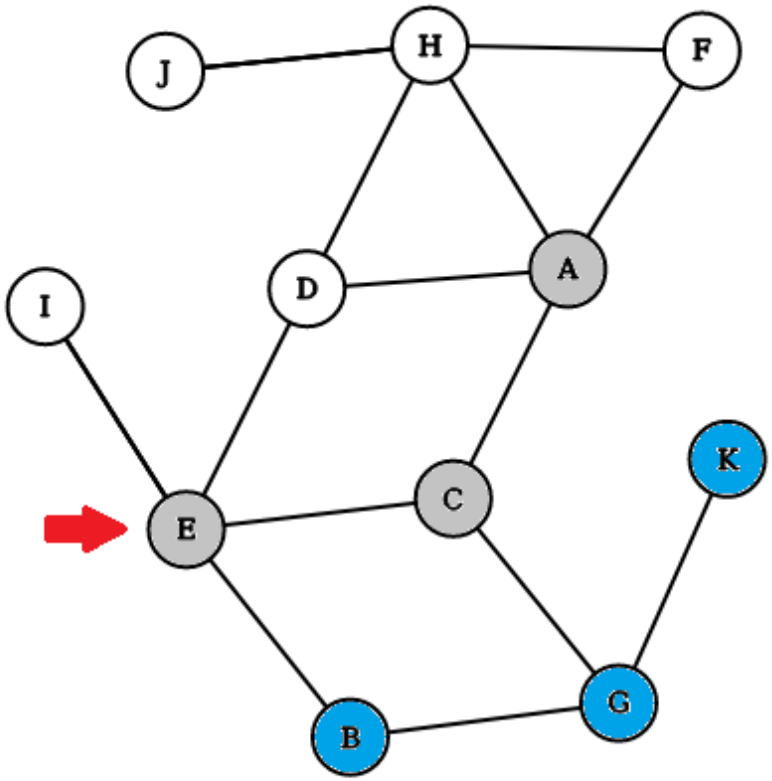
Ejemplo



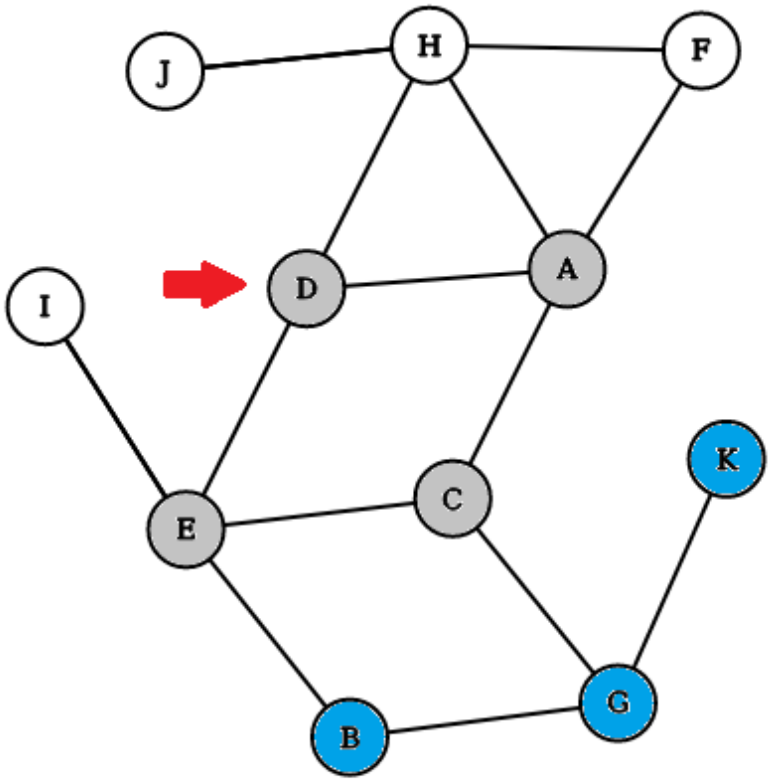
Ejemplo



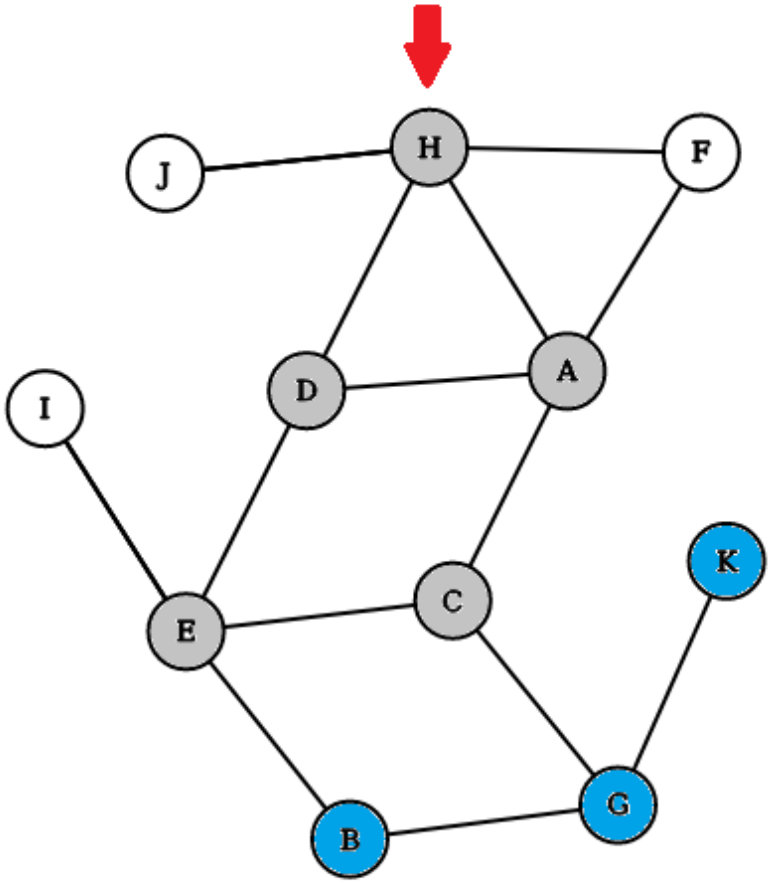
Ejemplo



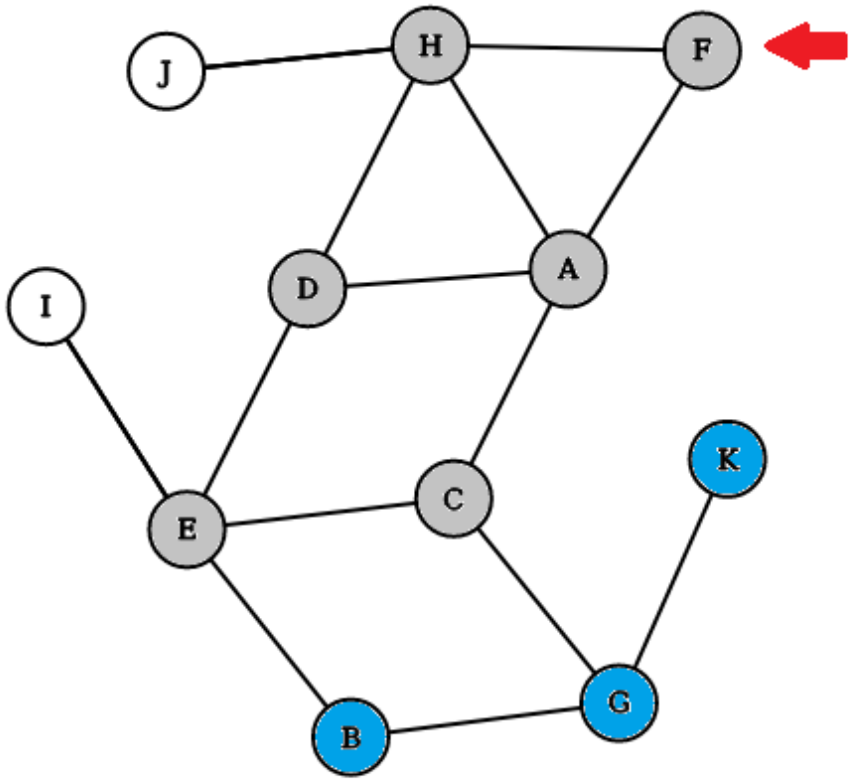
Ejemplo



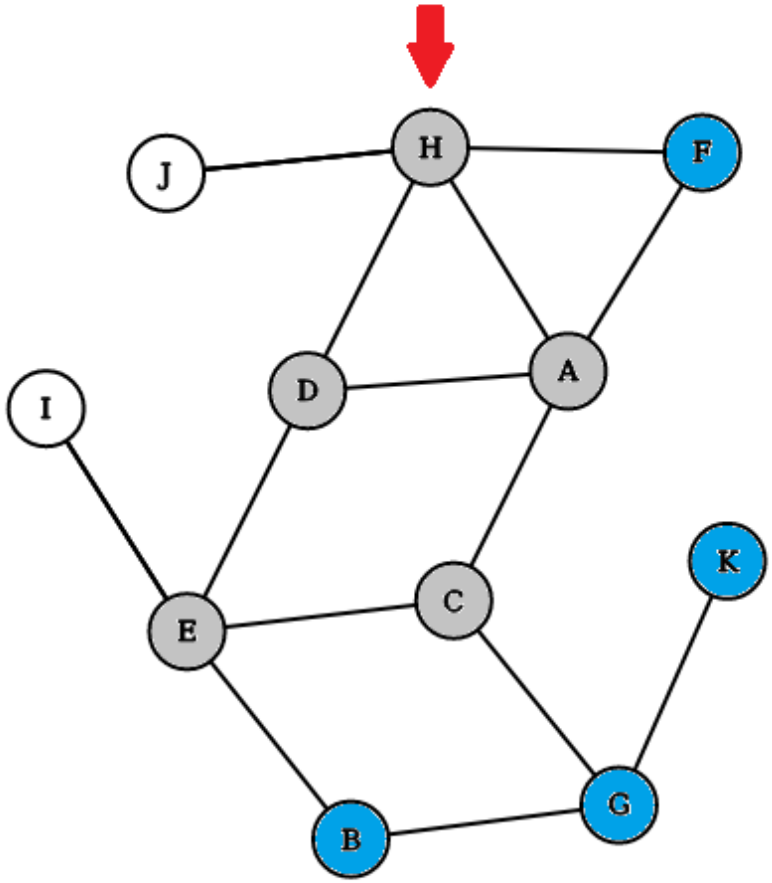
Ejemplo



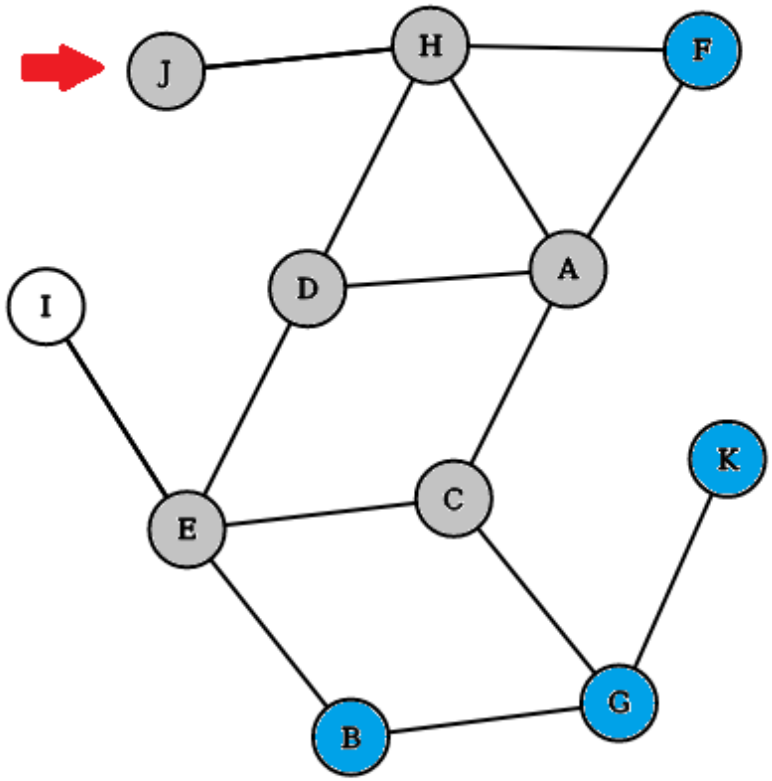
Ejemplo



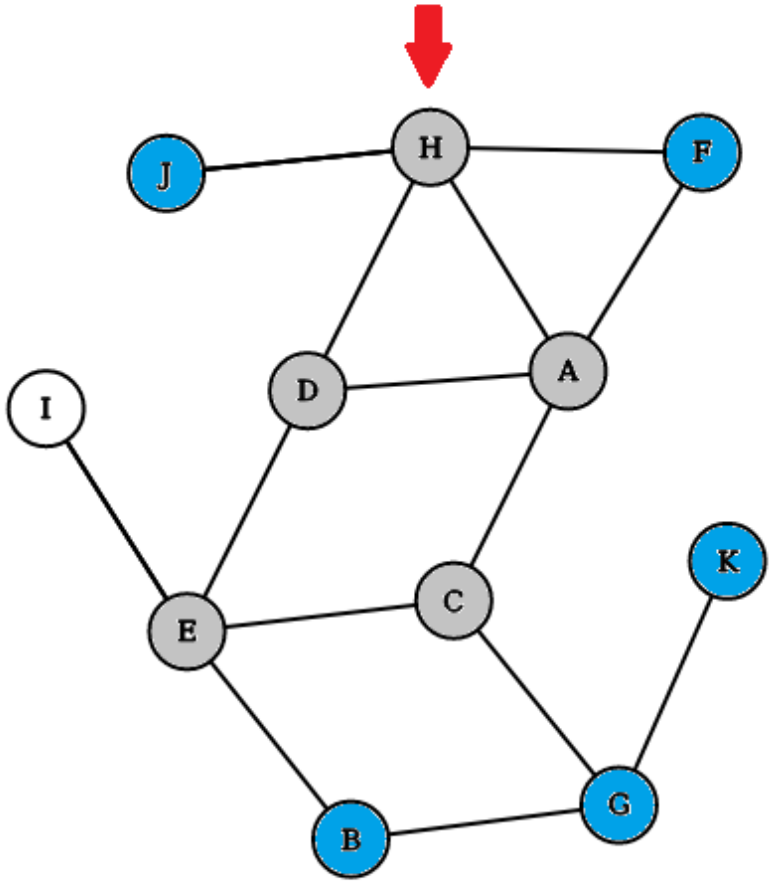
Ejemplo



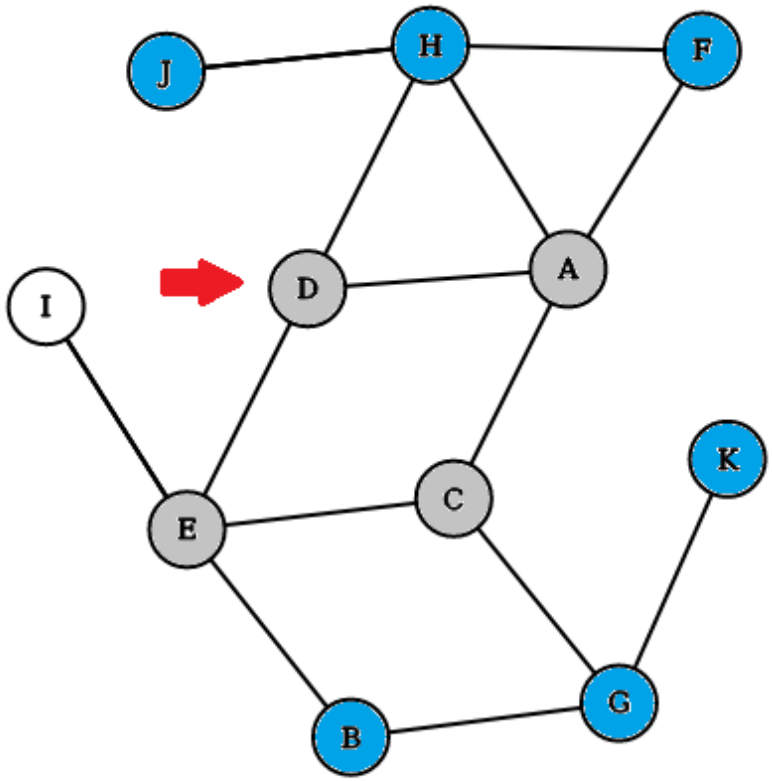
Ejemplo



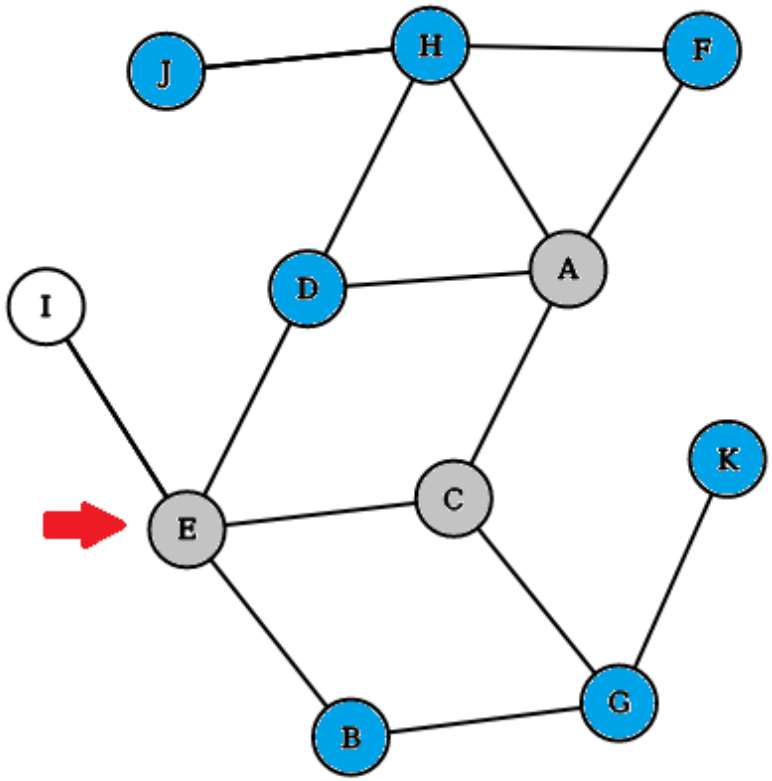
Ejemplo



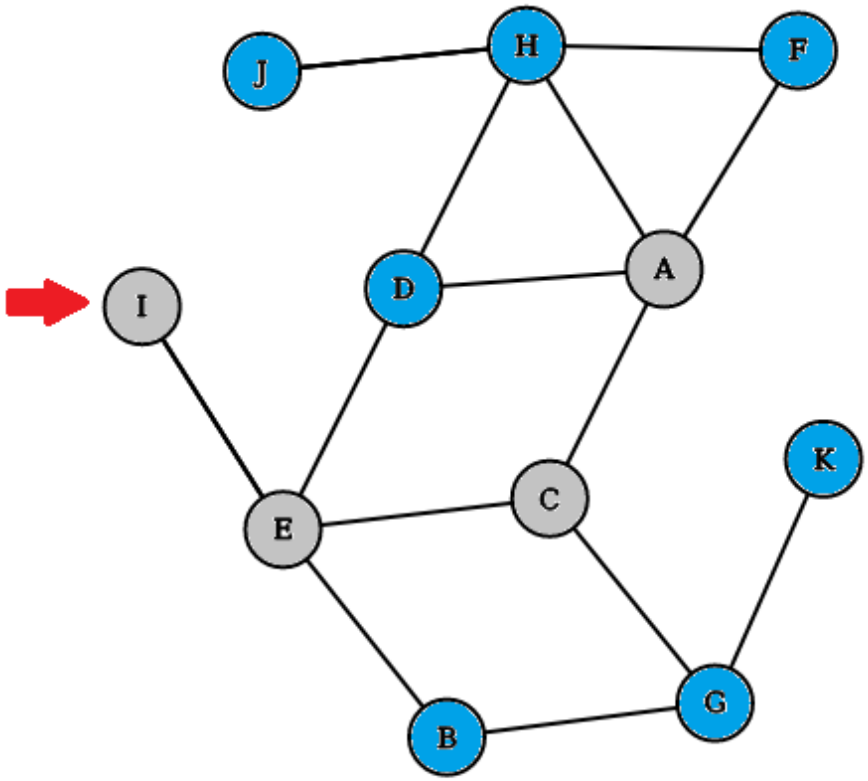
Ejemplo



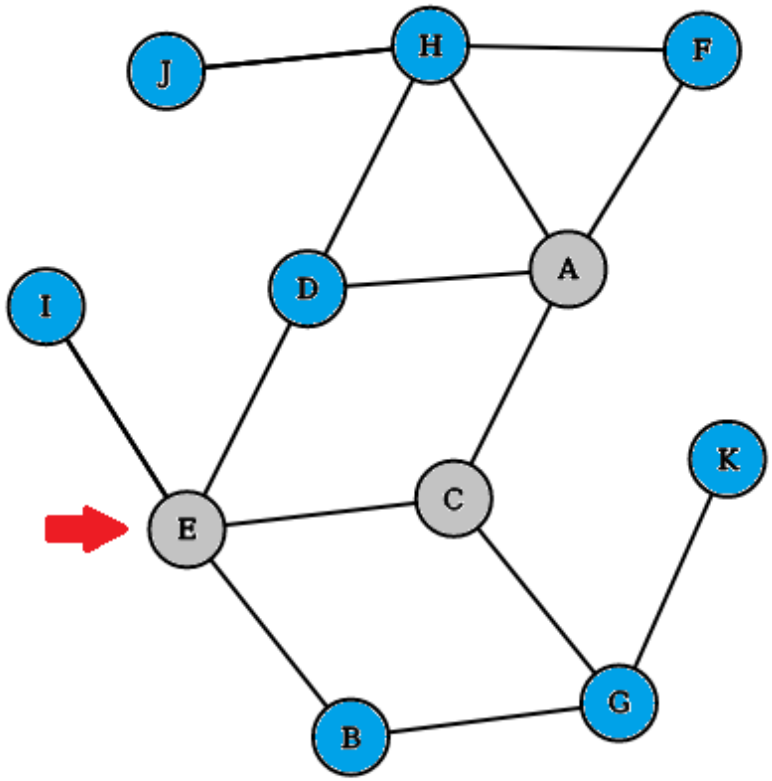
Ejemplo



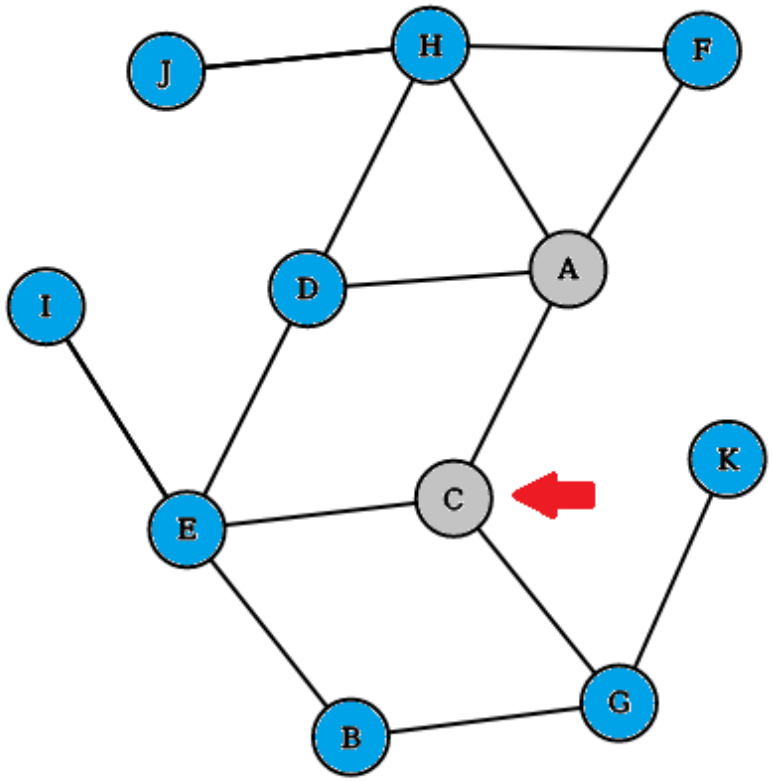
Ejemplo



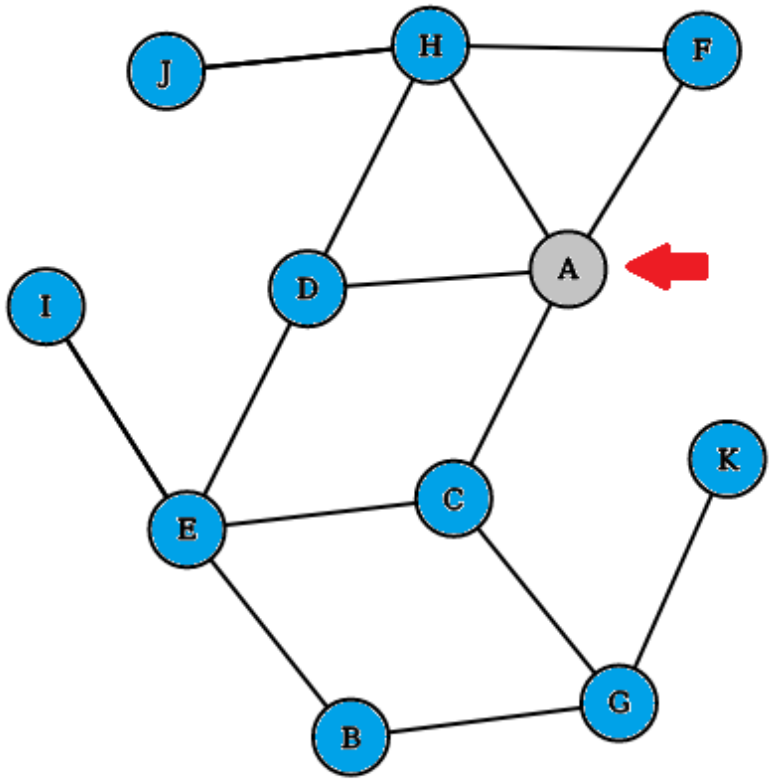
Ejemplo



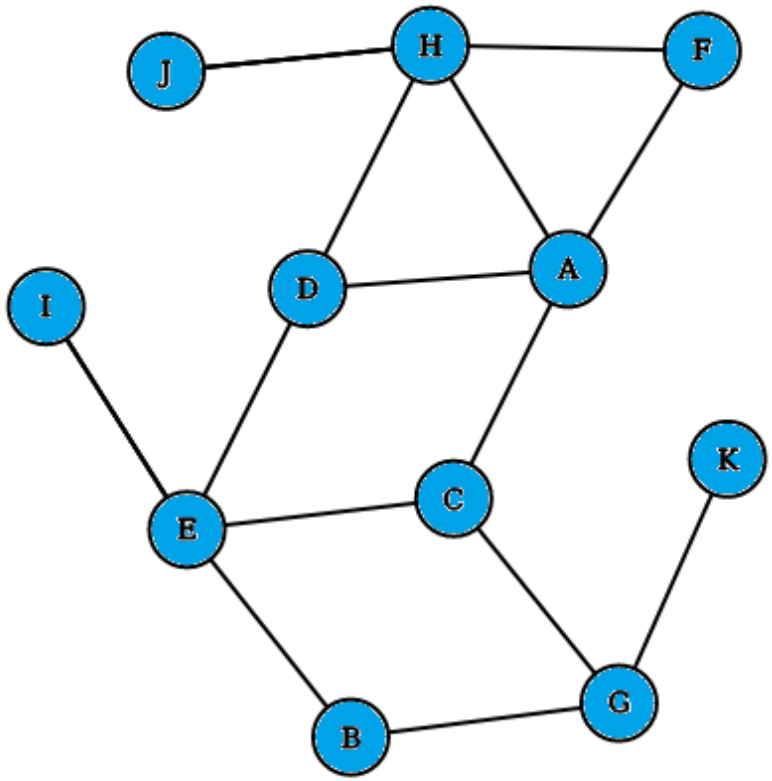
Ejemplo



Ejemplo



Ejemplo



Implementación iterativa

```
1  void DFS(int node) {
2    stack<int> s;
3    s.push(node);
4    while(s.size()) { // mientras haya nodos por procesar
5      int x = s.top();
6      visitado[x] = true;
7      s.pop();
8      for(int y: graph[x]) { // para cada y vecino de x
9        if(!visitado[y]) { // si y no fue visitado
10         s.push(y);
11       }
12     }
13   }
14 }
```

Implementación recursiva

```
1 void DFS(int node) {
2   visitado[node] = true;
3   for(int y: graph[node]) { // para cada y vecino de node
4     if(!visitado[y]) { // si y no fue visitado
5       DFS(y);
6     }
7   }
8 }
```

Análisis

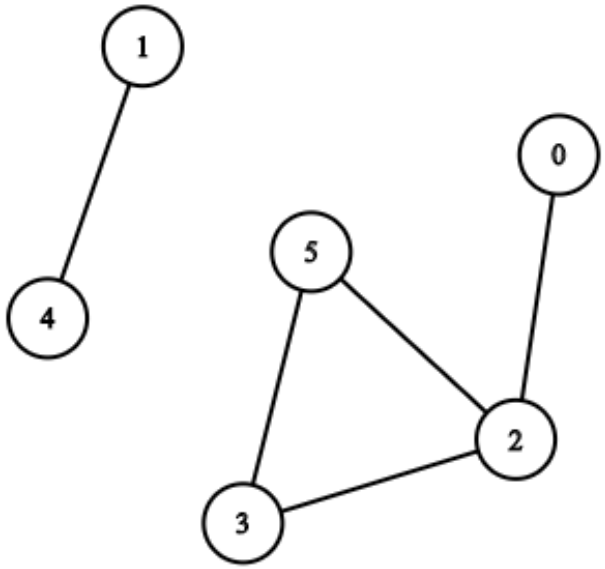
- Un análisis similar al que hicimos para BFS concluye que DFS también es $O(m)$.
- BFS y DFS son similares: ambos procesan todos los nodos alcanzables desde cierto nodo, pero lo hacen en distinto orden.
- Muchos problemas salen con ambas técnicas, por lo que podemos usar la que nos quede más cómodo.
- Algunos problemas sólo se resuelven con una de ellas:
  - BFS: distancias mínimas.
  - DFS: algunos problemas que tienen que ver con la estructura del grafo, como por ejemplo encontrar puentes y puntos de articulación.

Definición: conexión

Decimos que dos vértices u y v están **conectados** si hay un camino (secuencia de aristas con vértice en común) que los une.

- Para todo grafo **no-dirigido**, podemos particionar el conjunto de nodos en varios **subconjuntos**, tales que dos nodos están conectados si y sólo si pertenecen al mismo subconjunto.
- Estos subconjuntos se denominan **componentes conexas** del grafo.

Ejemplo



Las componentes conexas son $\{1, 4\}$ y $\{0, 2, 3, 5\}$

Problema

Problema: Rumor

Vova necesita **expandir un rumor** sobre los N habitantes de una ciudad. Hay M pares de habitantes que son amigos, y si un habitante empieza a divulgar un rumor, sus amigos también lo van a divulgar, y los amigos de sus amigos también, y así sucesivamente. Sin embargo ninguno de ellos es amigo de Vova, si no que cada uno le pide cierta cantidad de **monedas** para empezar a divulgar su rumor. Formalmente, el i-ésimo habitante le pide c_i monedas. Cuál es la **mínima** cantidad de monedas que tiene que gastar Vova?

Link: `https://codeforces.com/problemset/problem/893/C`

Solución

Enunciado resumido

Dado un grafo no dirigido, donde cada nodo tiene peso, encontrar el nodo con menor peso de cada componente conexa.

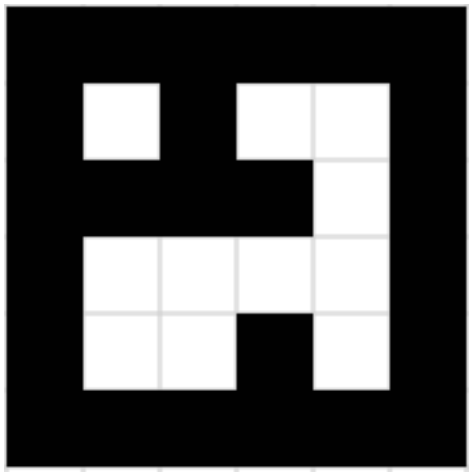
Solución

Enunciado resumido

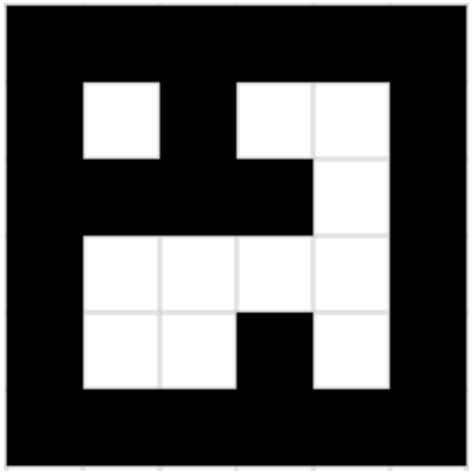
Dado un grafo no dirigido, donde cada nodo tiene peso, encontrar el nodo con menor peso de cada componente conexa.

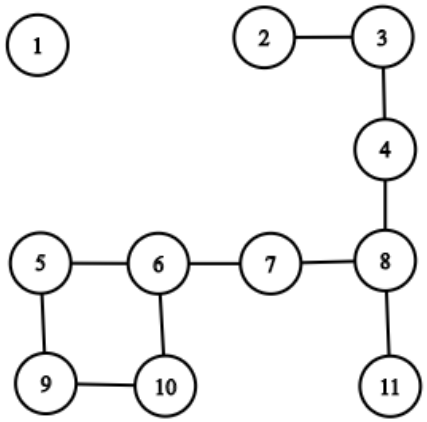
```
1  // aca declarar variables globales necesarias
2  int dfs(int nodo) {
3    used[nodo] = true;
4    int ret = v[nodo];
5    for(int y: graph[node]) if(!used[y]) ret = min(ret, dfs(y));
6    return ret;
7  }
8  int main() {
9    // aca declarar variables necesarias y leer datos
10   for(int i = 0; i < n; i++)  if(!used[i]) ans += dfs(i);
11   cout << ans << "\n";
12 }
```

Floodfill

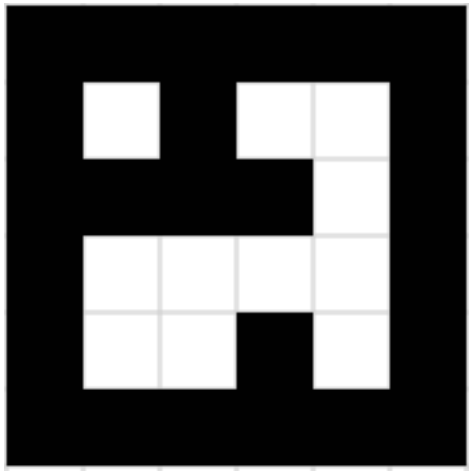


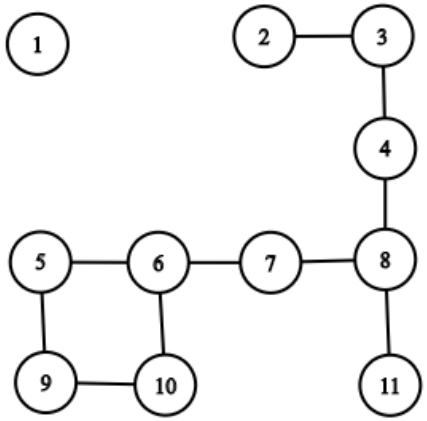
Floodfill





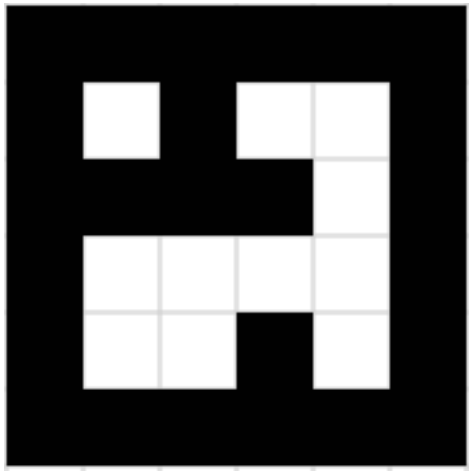
Floodfill

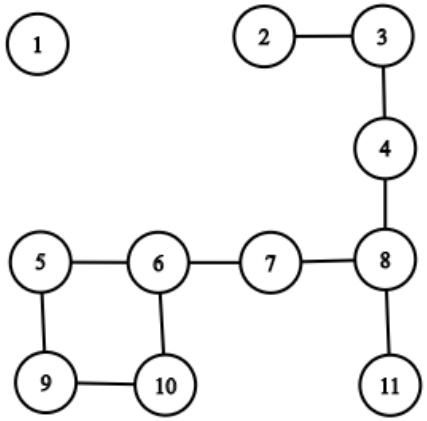




Se puede usar la matriz como grafo **implícito**

Floodfill





Se puede usar la matriz como grafo **implícito**

Problema

Dado un tablero donde un 0 representa el color negro y un 1 el color blanco. Contar la cantidad de componentes blancas, y cuantas celdas tiene cada una.

DFS en tablero

```
1  int n, m; // dimensiones del tablero
2  int dr[4] = {0,0,1,-1}, dc[4] = {1,-1,0,0}; // para implementar
       movimientos
3  vector<vector<int>> table, used;
4
5  int floodfill(int r,int c) {
6    if(r < 0 || c < 0 || r >= n || c >= m || used[r][c] || table[r
         ][c] == 0)
7      return;
8    used[r][c] = true;
9    int ret = 1;
10   for(int i = 0; i < 4; i++) ret += floodfill(r + dr[i], c + dc[
         i]);
11   return ret;
12 }
```

Definición: Bipartito

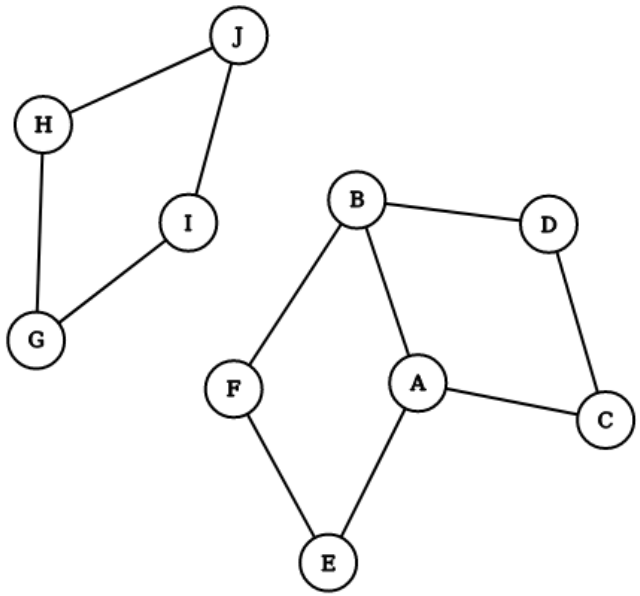
Un grafo se dice **bipartito** si le podemos pintar los vértices usando dos colores, de forma que toda arista conecte a dos vértices de distinto color.

Definición: Bipartito

Un grafo se dice **bipartito** si le podemos pintar los vértices usando dos colores, de forma que toda arista conecte a dos vértices de distinto color. *Un grafo es bipartito si y solo si no tiene ciclos impares*.

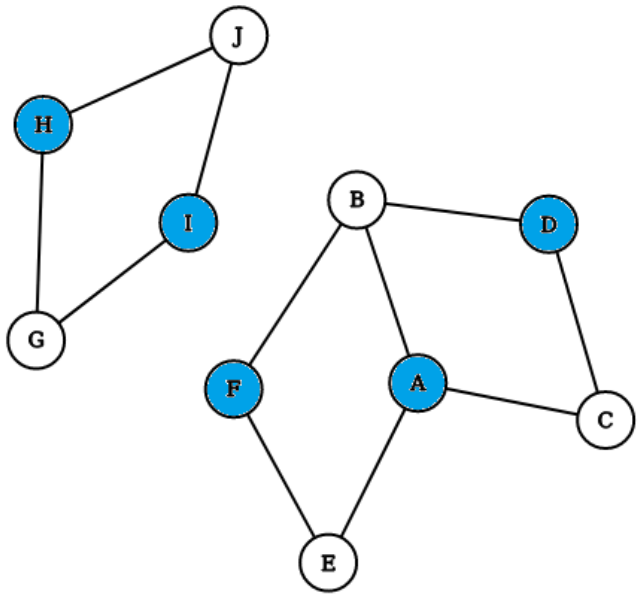
Definición: Bipartito

Un grafo se dice **bipartito** si le podemos pintar los vértices usando dos colores, de forma que toda arista conecte a dos vértices de distinto color. *Un grafo es bipartito si y solo si no tiene ciclos impares*.



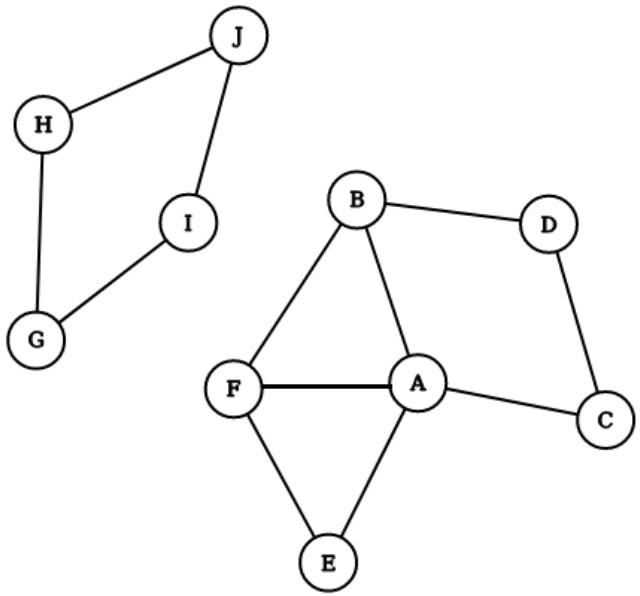
Definición: Bipartito

Un grafo se dice **bipartito** si le podemos pintar los vértices usando dos colores, de forma que toda arista conecte a dos vértices de distinto color. *Un grafo es bipartito si y solo si no tiene ciclos impares*.



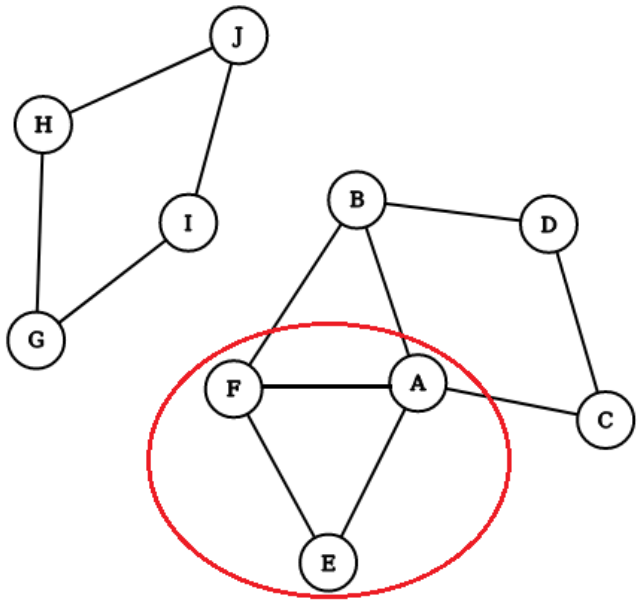
Definición: Bipartito

Un grafo se dice **bipartito** si le podemos pintar los vértices usando dos colores, de forma que toda arista conecte a dos vértices de distinto color. *Un grafo es bipartito si y solo si no tiene ciclos impares*.



Definición: Bipartito

Un grafo se dice **bipartito** si le podemos pintar los vértices usando dos colores, de forma que toda arista conecte a dos vértices de distinto color. *Un grafo es bipartito si y solo si no tiene ciclos impares*.



Problema

Problema: See World

En un acuario hay N orcas, y solamente tiene 2 tanques donde las orcas pueden vivir. Cada orca habla algún idioma, y un par de orcas pueden hablar entre si, si ambas hablan el mismo idioma. Además, si dos orcas que no pueden comunicarse viven en el mismo tanque, se van a pelear. ¿Es posible dividir las orcas en los tanques para que no se peleen?

Link: `https://www.urionlinejudge.com.br/judge/en/problems/view/1955`

Solución

Enunciado resumido

Dado un grafo no dirigido, determinar si se puede dividir los nodos en dos grupos, tal que exista una arista entre todo par de nodos del mismo grupo.

Solución

Enunciado resumido

Dado un grafo no dirigido, determinar si se puede dividir los nodos en dos grupos, tal que exista una arista entre todo par de nodos del mismo grupo.

Parece un problema difícil, donde hay que probar sobre los subconjuntos.

Solución

Enunciado resumido

Dado un grafo no dirigido, determinar si se puede dividir los nodos en dos grupos, tal que exista una arista entre todo par de nodos del mismo grupo.

Parece un problema difícil, donde hay que probar sobre los subconjuntos. $N <= 10^3$

Solución

Enunciado resumido

Dado un grafo no dirigido, determinar si se puede dividir los nodos en dos grupos, tal que exista una arista entre todo par de nodos del mismo grupo.

Parece un problema difícil, donde hay que probar sobre los subconjuntos. $N <= 10^3$

Otro grafo

Si pensamos el grafo que es el **complemento** del definido arriba ¿A que se traduce el enunciado?

Solución

Enunciado resumido

Dado un grafo no dirigido, determinar si se puede dividir los nodos en dos grupos, tal que exista una arista entre todo par de nodos del mismo grupo.

Parece un problema difícil, donde hay que probar sobre los subconjuntos. $N <= 10^3$

Otro grafo

Si pensamos el grafo que es el **complemento** del definido arriba ¿A que se traduce el enunciado? **A verificar que este grafo sea bipartito**

Solución

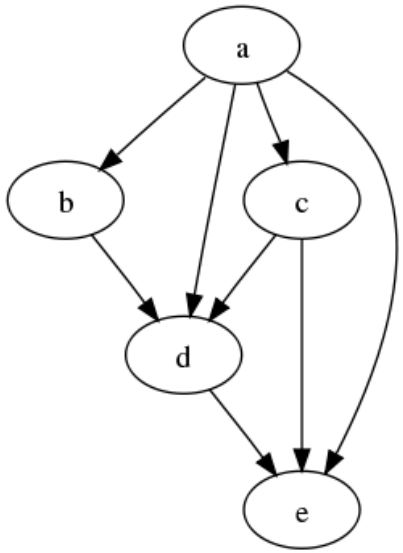
```
1  // aca declarar variables globales necesarias
2  bool dfs(int node, int clr) {
3          color[node] = clr;
4          for(int i = 0; i < n; i++) if(matAdy[node][i] == 1) {
5                  if(color[i] == color[node]) return false;
6                  if(color[i] == -1 && !dfs(i, 1-clr)) return
                       false;
7          }
8          return true;
9  }
10 int main() {
11   // aca declarar variables necesarias y leer datos
12         bool ok = true;
13         forn(i,n) if(color[i] == -1 && !dfs(i,0)) ok = false;
14   // aca mostrar la respuesta como pide el problema
15 }
```

Definición: orden topológico

Dado un grafo **dirigido**, un *orden topológico* es un ordenamiento de los nodos de modo que para cada arista $x \rightarrow y$, x viene antes que y en el orden.

Definición: orden topológico

Dado un grafo **dirigido**, un *orden topológico* es un ordenamiento de los nodos de modo que para cada arista $x \rightarrow y$, x viene antes que y en el orden.



Este grafo tiene dos órdenes topológicos:

- a,b,c,d,e
- a,c,b,d,e

- **No** todo grafo tiene orden topológico.
- Concretamente, un grafo tiene orden topológico si y sólo si no tiene ciclo (camino desde algún nodo a sí mismo). A estos grafos se los llama **DAG** (*directed-acyclic-graph* o "grafo dirigido sin ciclos").
- Si el grafo no tiene ciclo, podemos encontrar el orden topológico de la siguiente forma:
  - Obtenemos el **grado de entrada** de cada nodo y metemos los de grado 0 en una **cola**.
  - Mientras que la cola no este vacía, tomamos el nodo del frente y "lo **eliminamos**" del grafo, actualizando el grado de entrada de sus vecinos y metiendo a la cola los que pasen a tener grado 0.

Implementación

```
1  vector<int> topSort(vector<vector<int>> g, vector<int> inGrade)
2  {
3    int n = inGrade.size();
4    vector<int> topSorted;
5    queue<int> q;
6    for(int i = 0; i < n; i++) if(inGrade[i] == 0) q.push(i);
7    while(!q.empty()) {
8      int node = q.front(); q.pop();
9      topSorted.push_back(node);
10     for(int y : g[node]) if(--inGrade[y] == 0) q.push(y);
11   }
12   //Si hay ciclos retornar un vector vacio
13   if(topSorted.size() < n) topSorted.clear();
14   return topSorted;
15 }
```

Ejes con peso

- Hasta ahora en los grafos que vimos la distancia entre dos nodos vecinos siempre fue 1.

Ejes con peso

- Hasta ahora en los grafos que vimos la distancia entre dos nodos vecinos siempre fue 1.
- Un grafo **ponderado** es un grafo en el cuál los ejes tienen peso. Los pesos de los ejes pueden ser números en $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, etc.

Ejes con peso

- Hasta ahora en los grafos que vimos la distancia entre dos nodos vecinos siempre fue 1.
- Un grafo **ponderado** es un grafo en el cuál los ejes tienen peso. Los pesos de los ejes pueden ser números en $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, etc.
- Los grafos ponderados se pueden **representar** con lista de adyacencia, guardando además del nodo destino, el peso de la arista (usando un par).

Ejes con peso

- Hasta ahora en los grafos que vimos la distancia entre dos nodos vecinos siempre fue 1.
- Un grafo **ponderado** es un grafo en el cuál los ejes tienen peso. Los pesos de los ejes pueden ser números en $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, etc.
- Los grafos ponderados se pueden **representar** con lista de adyacencia, guardando además del nodo destino, el peso de la arista (usando un par).
- La **distancia** entre dos nodos v y w es el menor d tal que existen $v = v_0, v_1, \ldots, v_n = w$ tales que si p_i es el peso del eje que une v_i y v_{i+1} entonces $d = \sum_{i=0}^{n-1} p_i$.

Ejes con peso

- Hasta ahora en los grafos que vimos la distancia entre dos nodos vecinos siempre fue 1.
- Un grafo **ponderado** es un grafo en el cuál los ejes tienen peso. Los pesos de los ejes pueden ser números en $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, etc.
- Los grafos ponderados se pueden **representar** con lista de adyacencia, guardando además del nodo destino, el peso de la arista (usando un par).
- La **distancia** entre dos nodos v y w es el menor d tal que existen $v = v_0, v_1, \ldots, v_n = w$ tales que si p_i es el peso del eje que une v_i y v_{i+1} entonces $d = \sum_{i=0}^{n-1} p_i$.

BFS no sirve para camino mínimo en grafos ponderados

Dado un nodo (v), el algoritmo de **Dijkstra** calcula la distancia mínima a todos los demás nodos en un grafo ponderado (sin aristas **negativas**). Funciona así:

Dado un nodo (v), el algoritmo de **Dijkstra** calcula la distancia mínima a todos los demás nodos en un grafo ponderado (sin aristas **negativas**). Funciona así:

- v tiene distancia a sí mismo 0, por lo que seteamos $d_v = 0$. Para los demás nodos seteamos inicialmente su distancia a ∞ (es decir, un valor suficientemente grande).

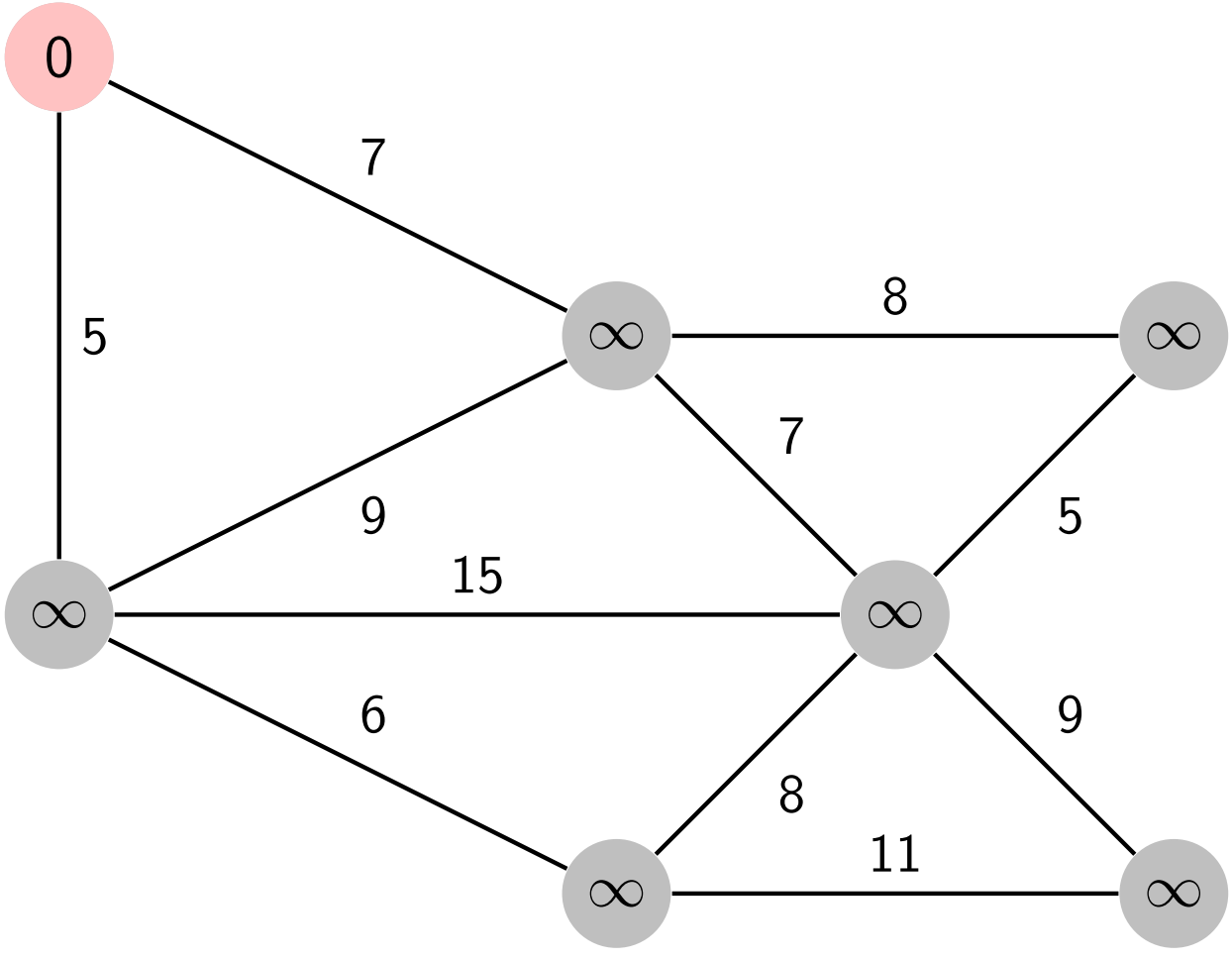
Dado un nodo (v), el algoritmo de **Dijkstra** calcula la distancia mínima a todos los demás nodos en un grafo ponderado (sin aristas **negativas**). Funciona así:

- v tiene distancia a sí mismo 0, por lo que seteamos $d_v = 0$. Para los demás nodos seteamos inicialmente su distancia a ∞ (es decir, un valor suficientemente grande).
- Vamos procesando los nodos uno por uno: En cada paso elegimos el nodo x que está a **menor** distancia de v (entre los que no procesamos).

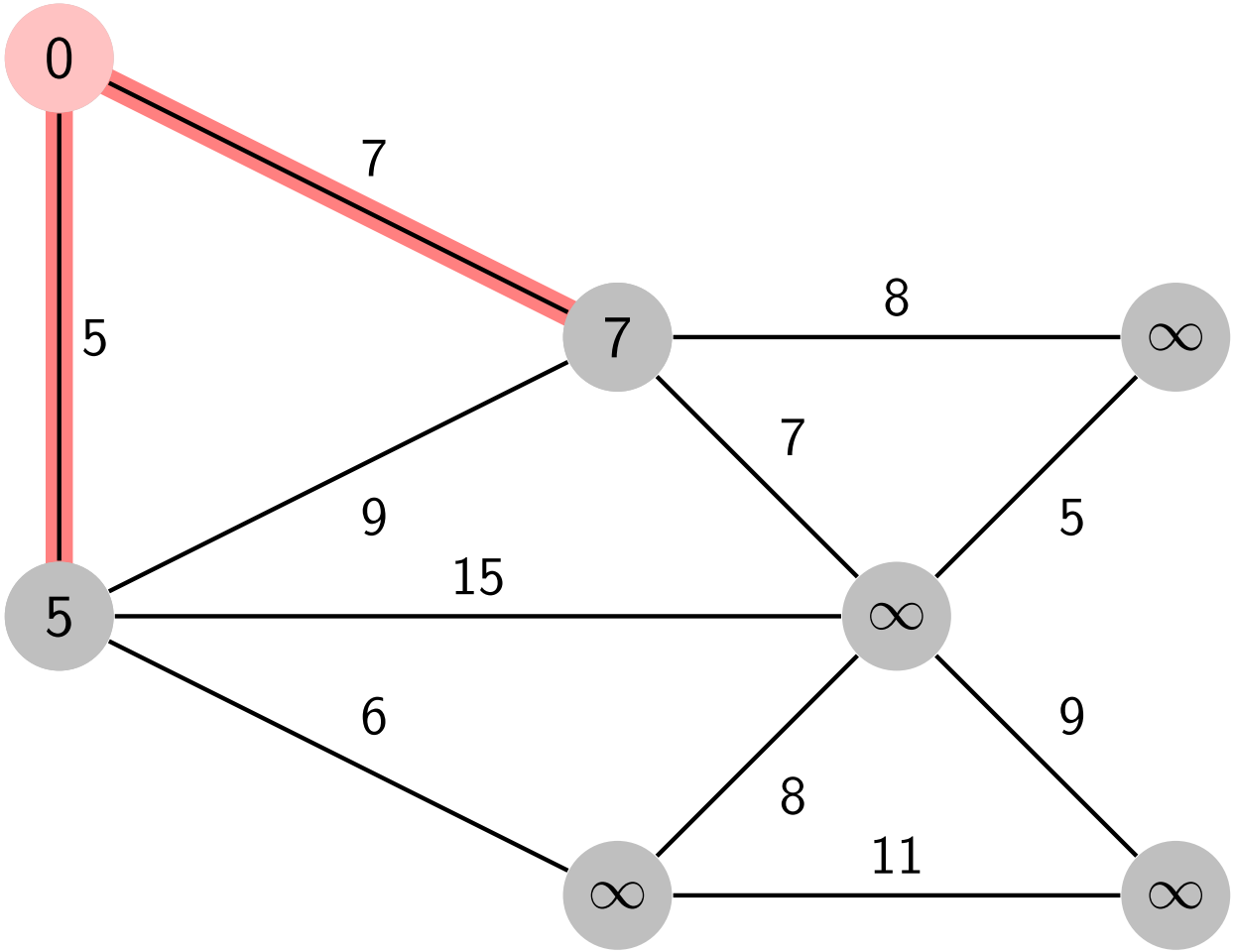
Dado un nodo (v), el algoritmo de **Dijkstra** calcula la distancia mínima a todos los demás nodos en un grafo ponderado (sin aristas **negativas**). Funciona así:

- v tiene distancia a sí mismo 0, por lo que seteamos $d_v = 0$. Para los demás nodos seteamos inicialmente su distancia a ∞ (es decir, un valor suficientemente grande).
- Vamos procesando los nodos uno por uno: En cada paso elegimos el nodo x que está a **menor** distancia de v (entre los que no procesamos).
- Para cada **vecino** y de x, actualizamos su distancia, en caso de que el camino que pasa por x justo antes de ir a y tenga menor costo que d_y (es decir si $d_x + c < d_y$, donde c es el peso de la arista (x, y)).

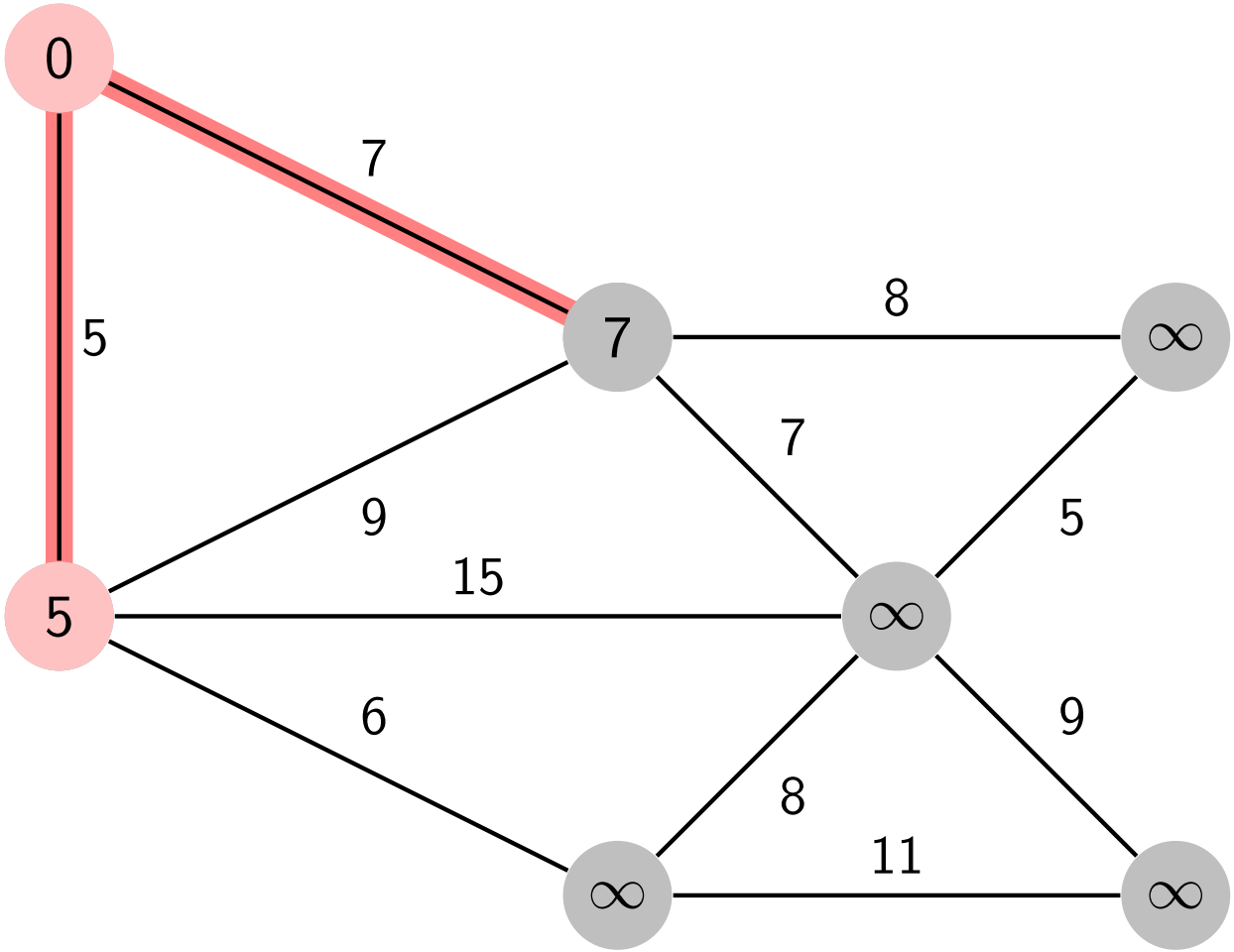
Ejemplo



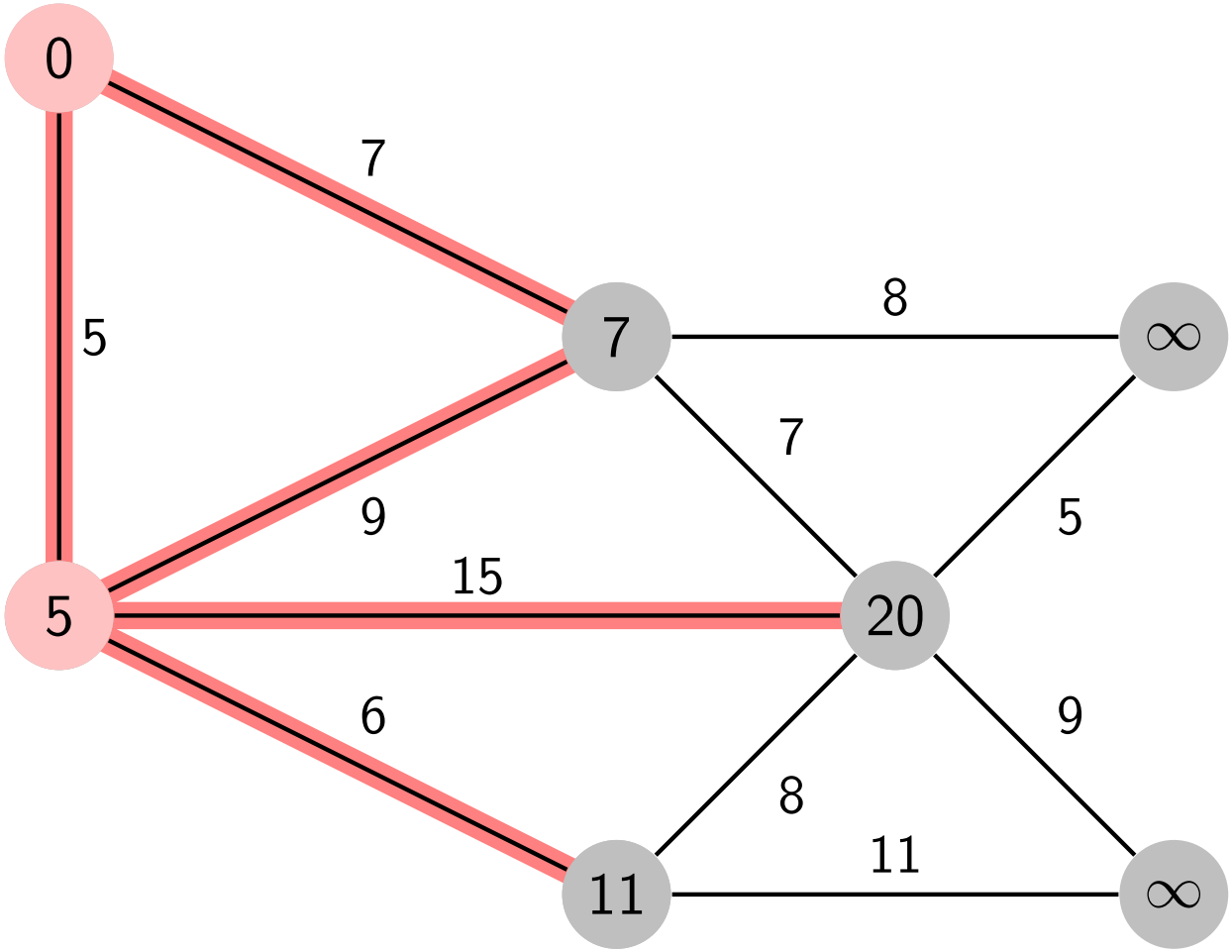
Ejemplo



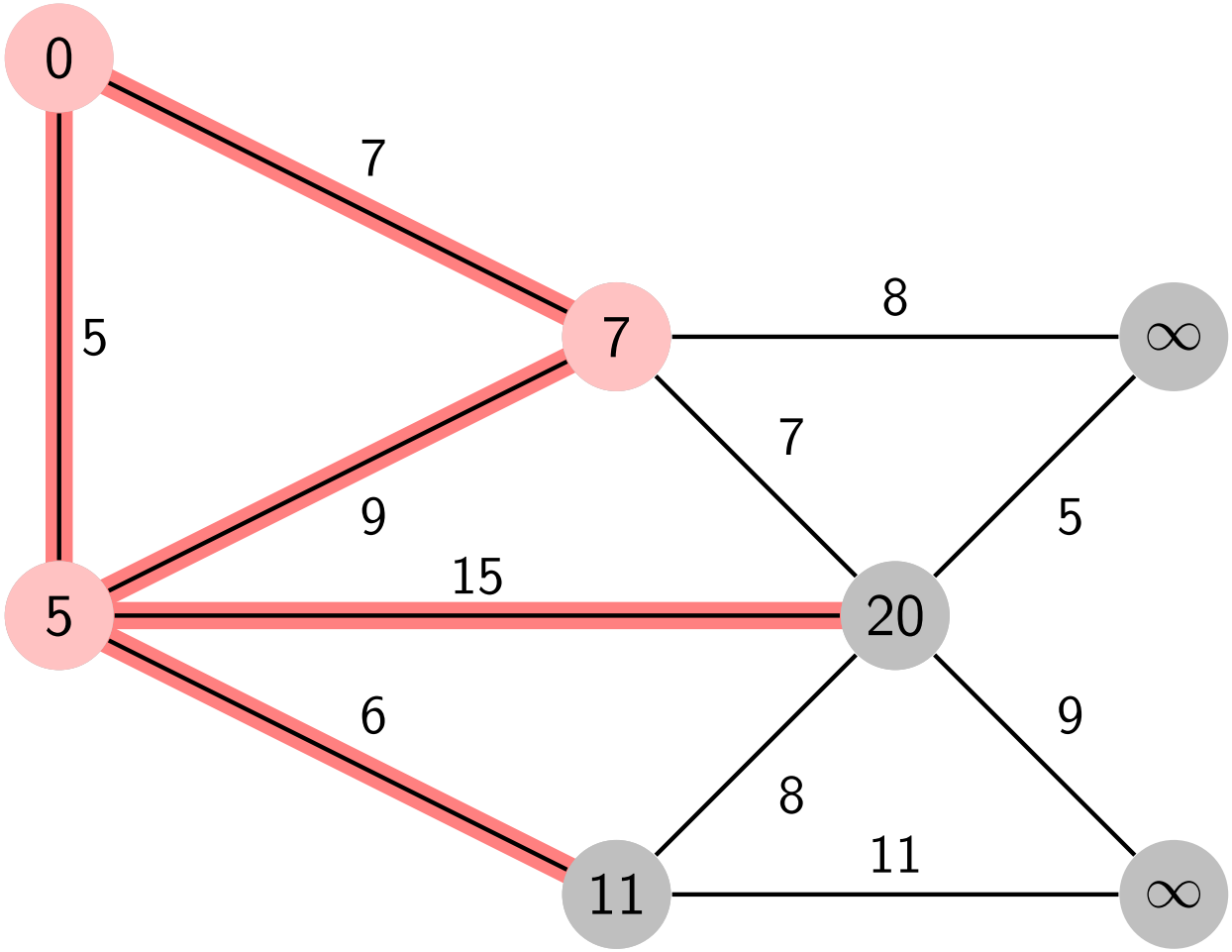
Ejemplo



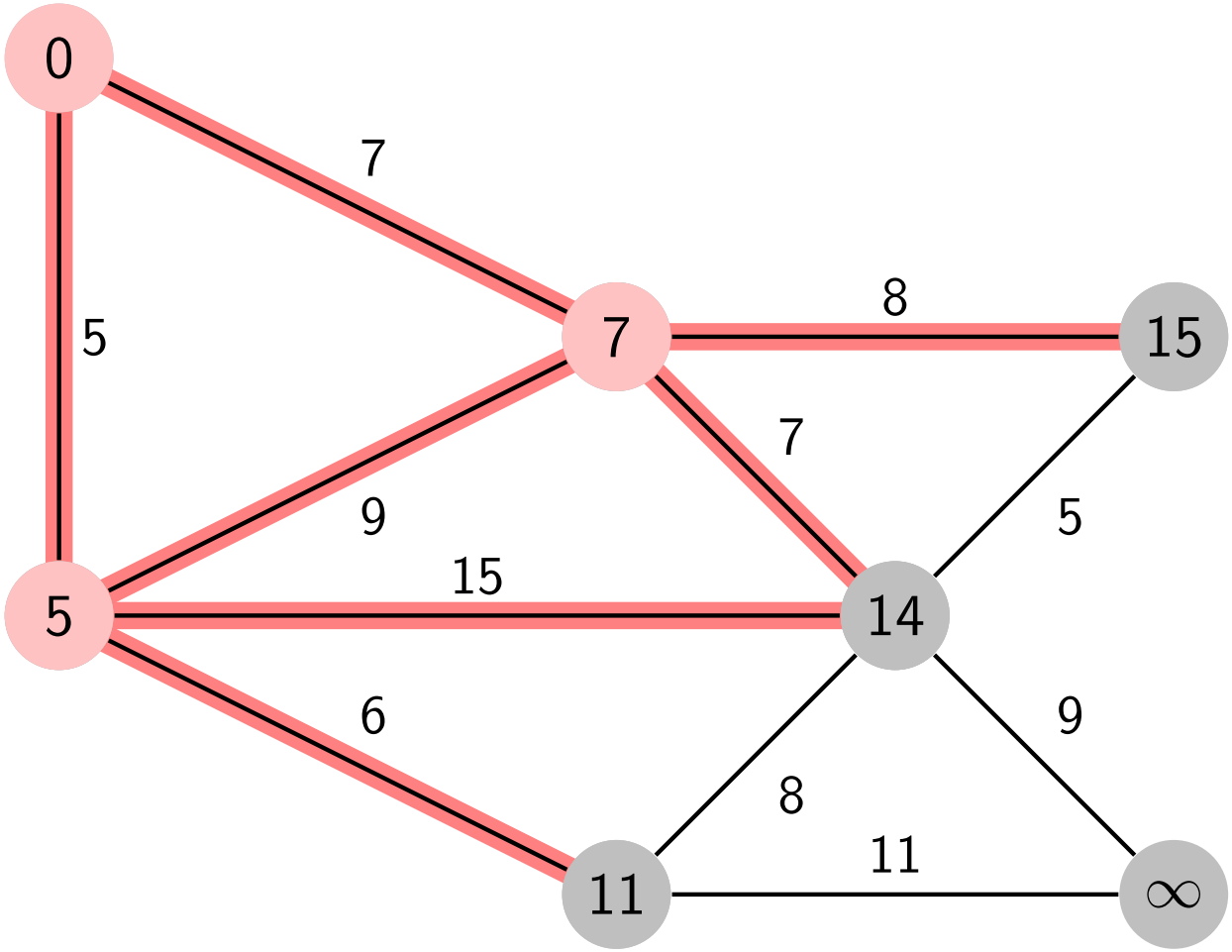
Ejemplo



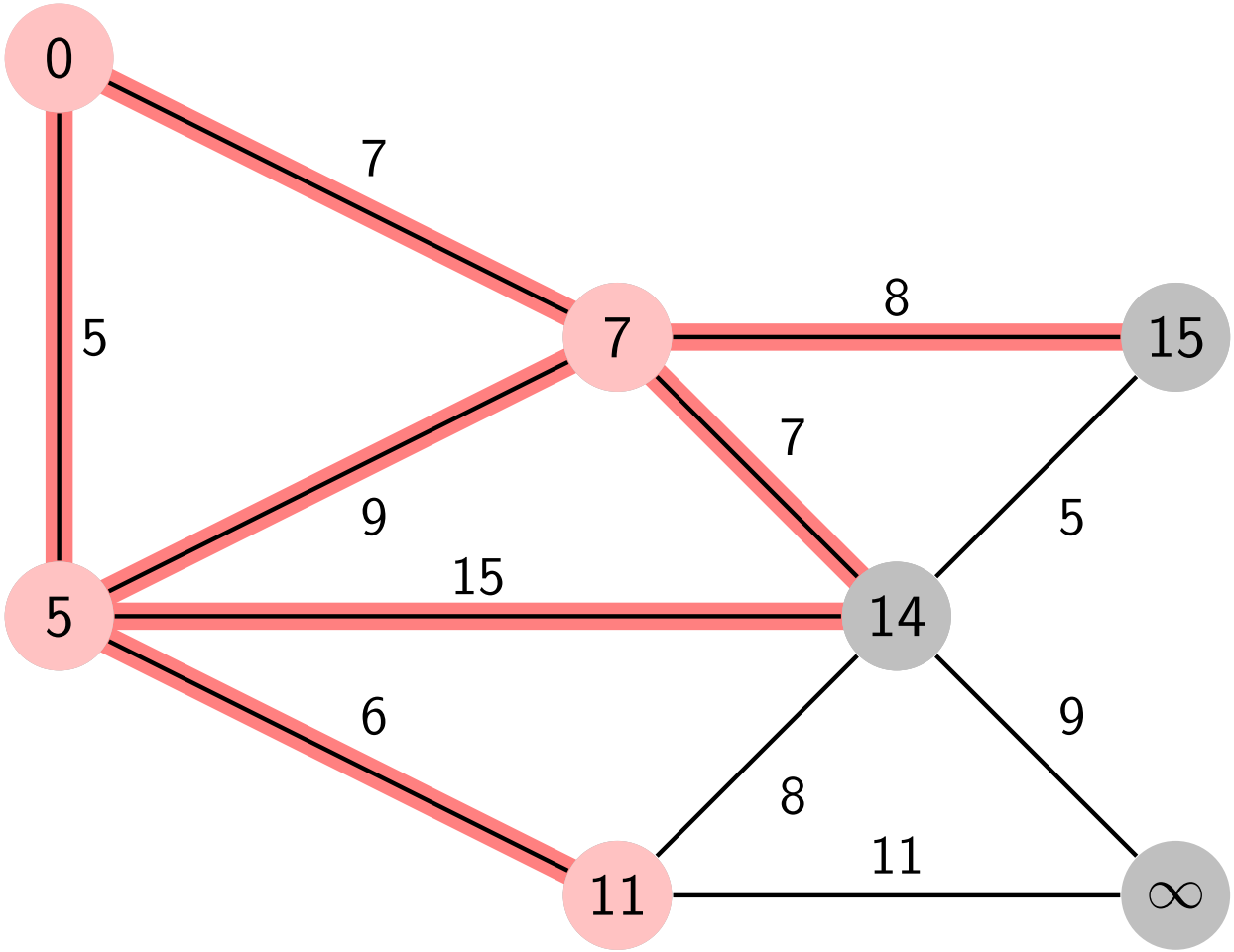
Ejemplo



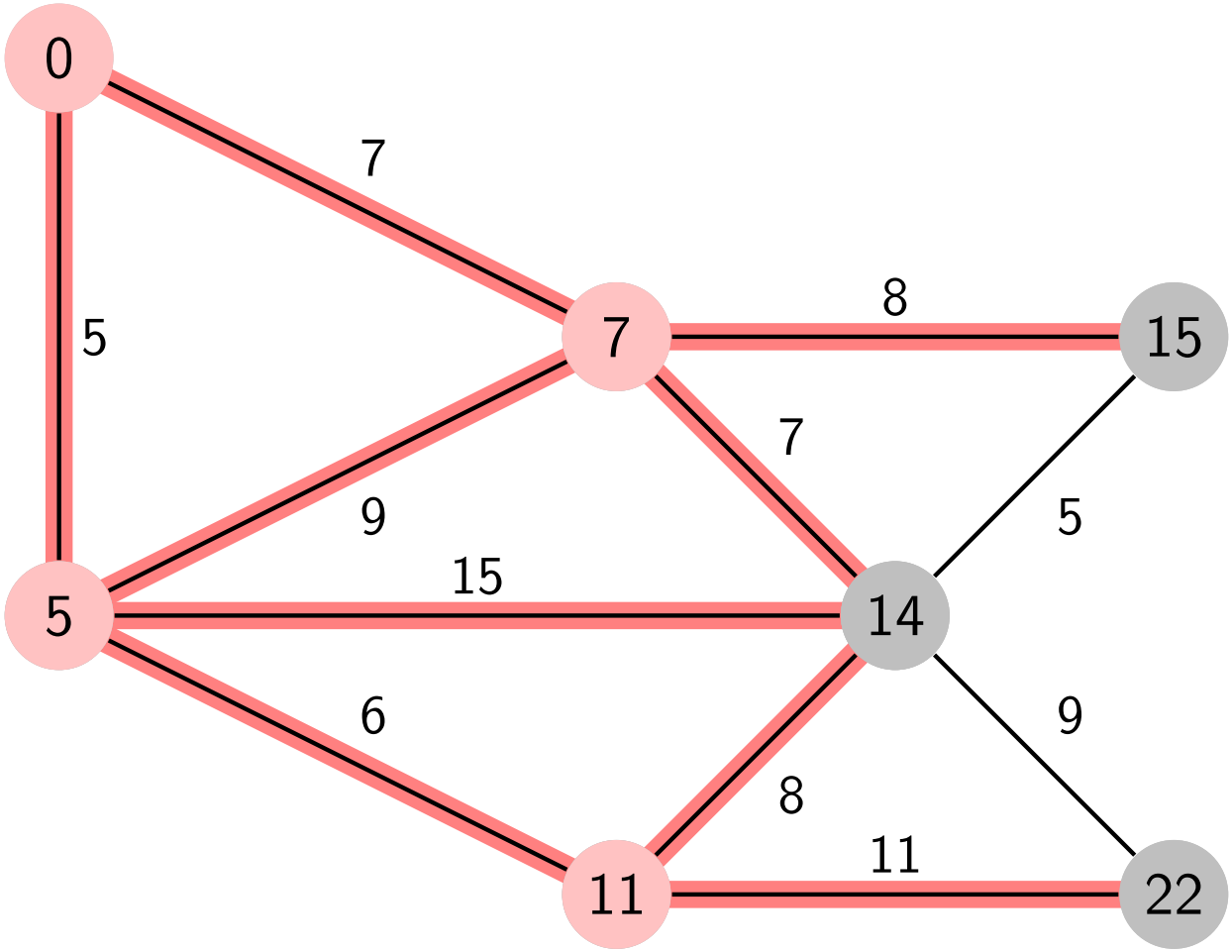
Ejemplo



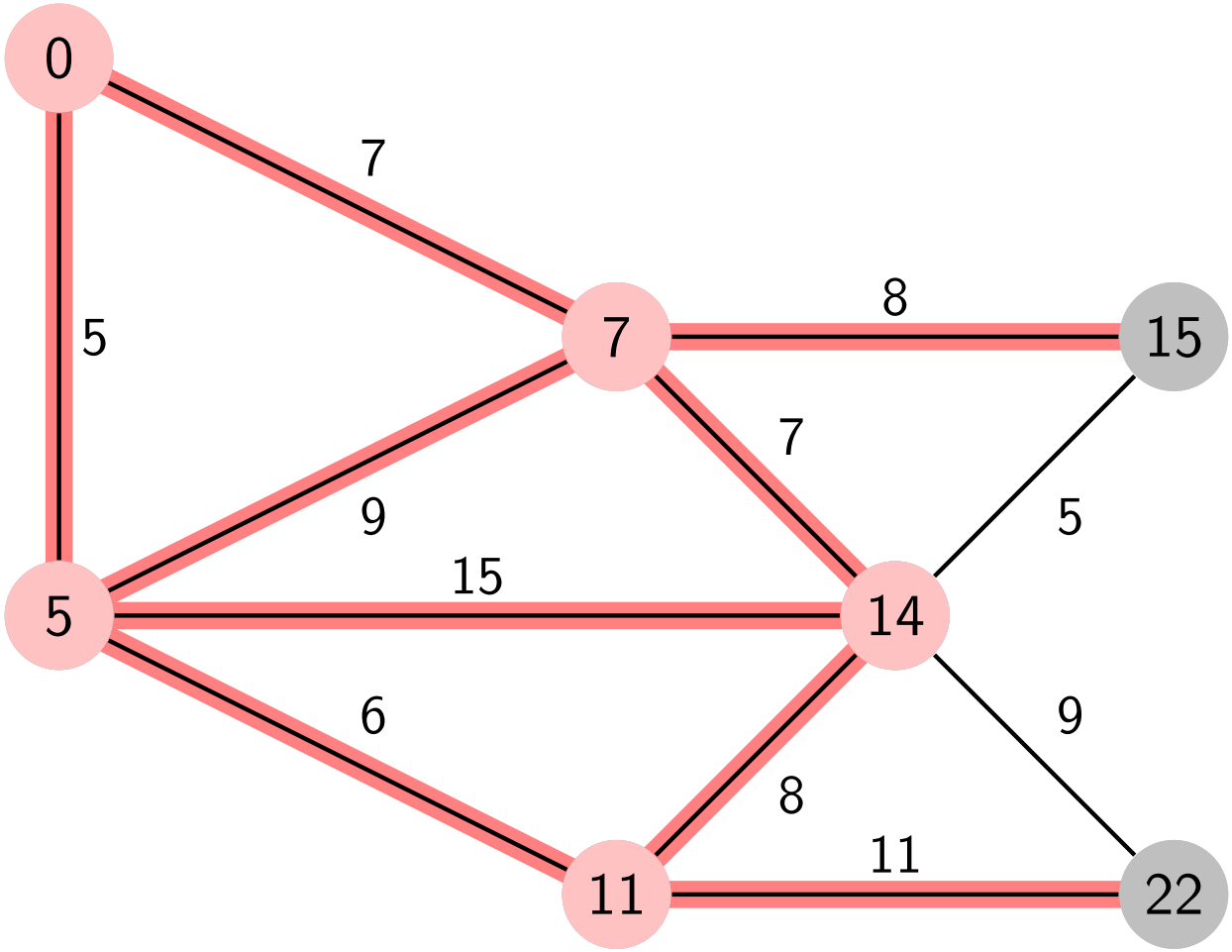
Ejemplo



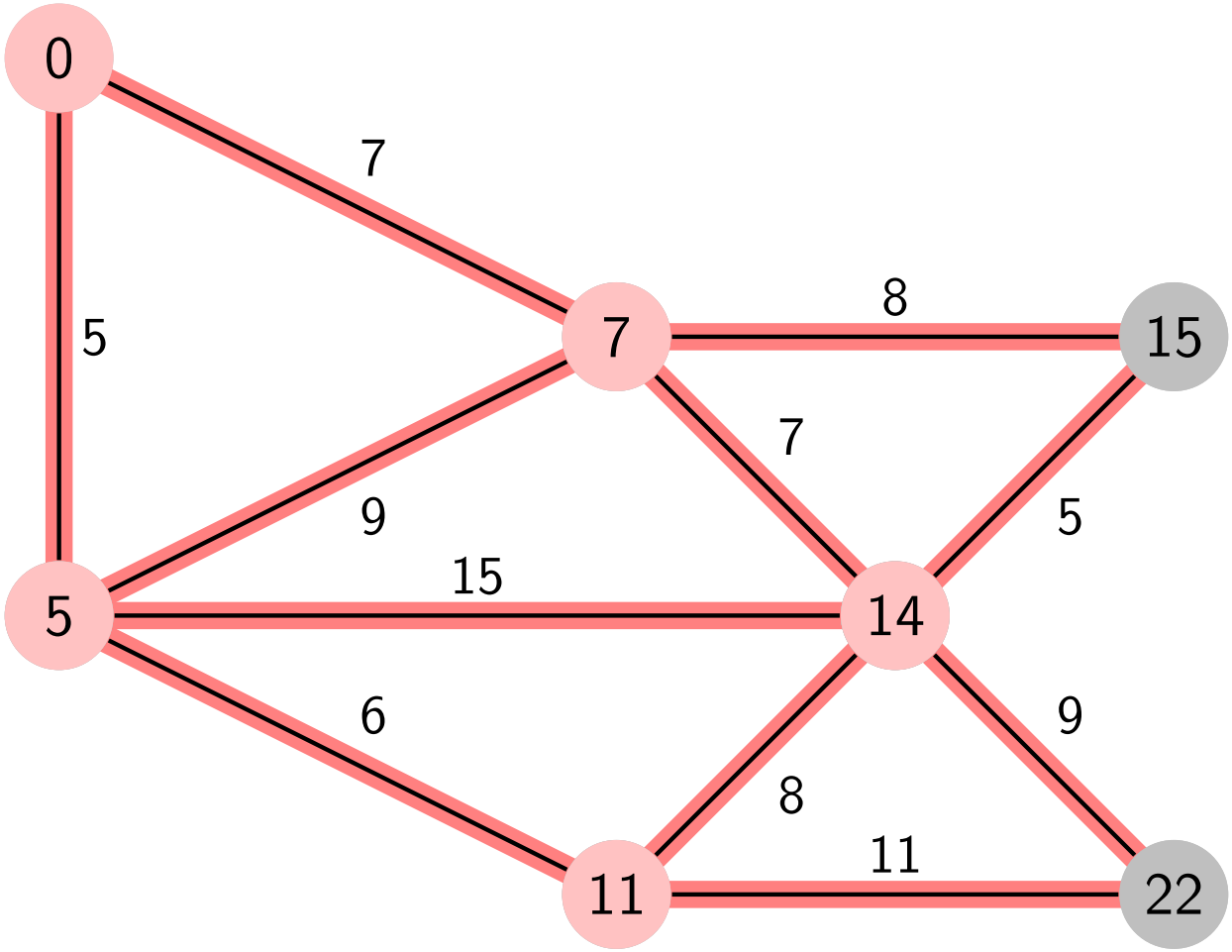
Ejemplo



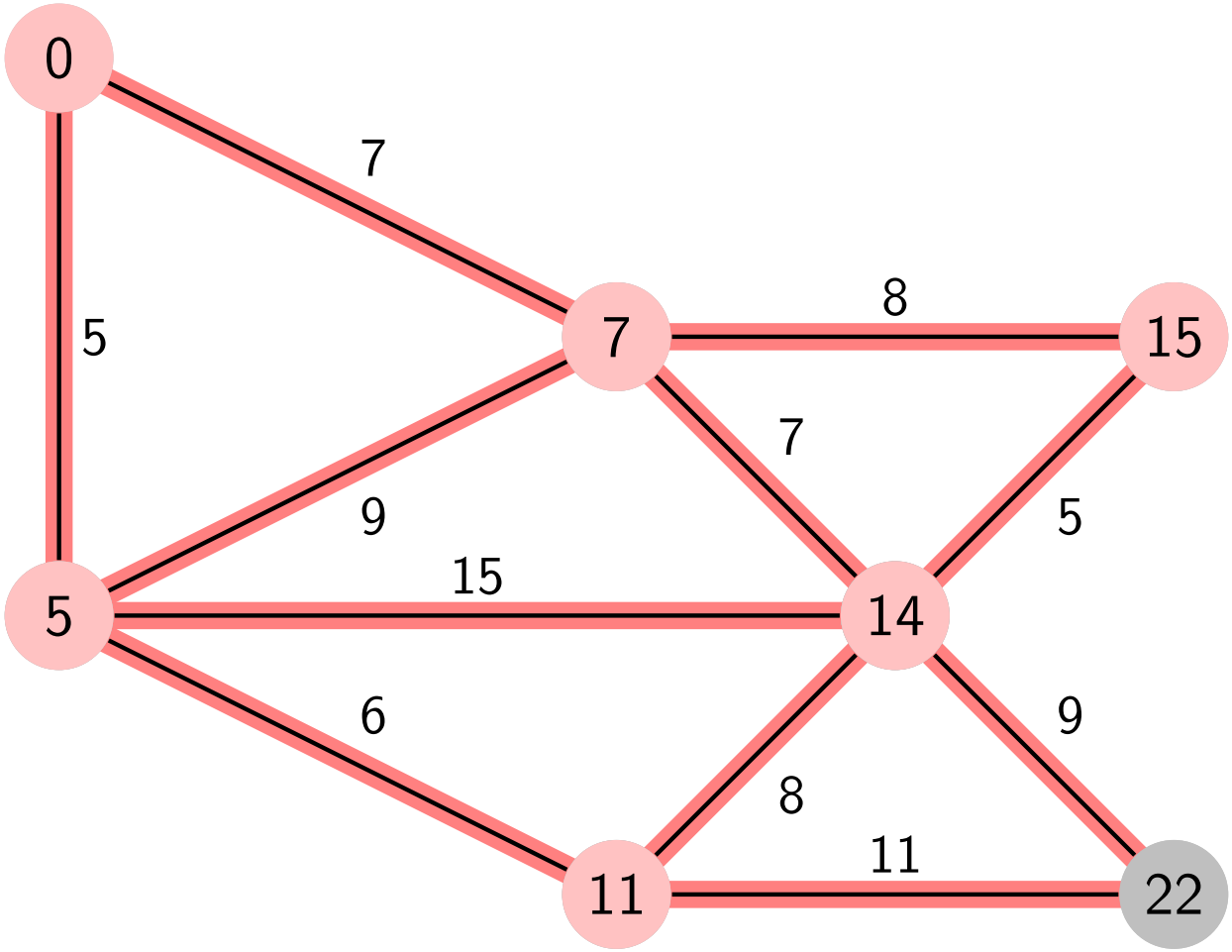
Ejemplo



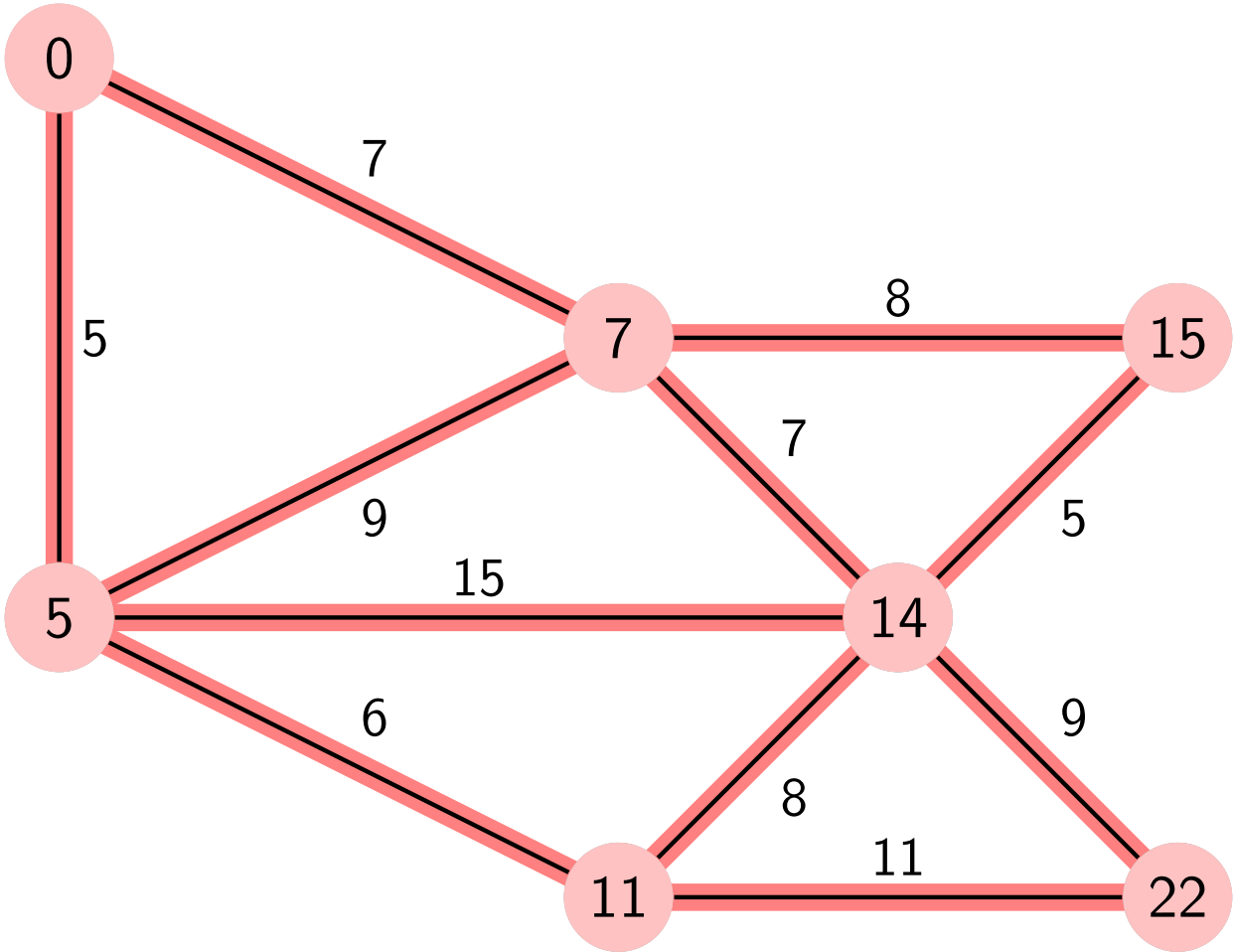
Ejemplo



Ejemplo



Ejemplo



Análisis

- La **complejidad** es $O(|V|^2)$ porque en cada uno de los $|V|$ pasos tenemos que **buscar** el nodo que está a menor distancia, y cada arista se considera una sóla vez.

Análisis

- La **complejidad** es $O(|V|^2)$ porque en cada uno de los $|V|$ pasos tenemos que **buscar** el nodo que está a menor distancia, y cada arista se considera una sóla vez.
- Se puede implementar en $O(|E|log(|E|))$, usando una **cola de prioridad** para elegir el nodo a menor distancia (suele ser necesario usar esta versión en las competencias).

Pesos negativos

- Dijkstra no funciona si algunas aristas tienen pesos **negativos**, porque cuando procesamos un nodo, no podemos estar seguros de que su distancia es efectivamente la que calculamos hasta ese punto.
- Además, cuando hay pesos negativos, puede que la distancia entre algún par de nodos sea $-\infty$, en caso de que exista algún **ciclo negativo**.

Pesos negativos

- Dijkstra no funciona si algunas aristas tienen pesos **negativos**, porque cuando procesamos un nodo, no podemos estar seguros de que su distancia es efectivamente la que calculamos hasta ese punto.
- Además, cuando hay pesos negativos, puede que la distancia entre algún par de nodos sea $-\infty$, en caso de que exista algún **ciclo negativo**.
- Para grafos con pesos negativos conviene usar el algoritmo de **Bellman-Ford** (que calcula la distancia de un nodo a todos los otros, como Dijkstra) o el algoritmo de **Floyd-Warshall** (que calcula la distancia entre todos los pares de nodos).

Dado un nodo (v), el algoritmo de Bellman-Ford calcula la distancia mínima a todos los demás nodos en un grafo ponderado, **incluso si hay aristas negativas**. Funciona así:

Dado un nodo (v), el algoritmo de Bellman-Ford calcula la distancia mínima a todos los demás nodos en un grafo ponderado, **incluso si hay aristas negativas**. Funciona así:

- Asumimos que no hay ciclos negativos.

Dado un nodo (v), el algoritmo de Bellman-Ford calcula la distancia mínima a todos los demás nodos en un grafo ponderado, **incluso si hay aristas negativas**. Funciona así:

- Asumimos que no hay ciclos negativos.
- v tiene distancia a sí mismo 0, por lo que seteamos $d_v = 0$. Para los demás nodos seteamos inicialmente su distancia a ∞ (es decir, un valor suficientemente grande).

Dado un nodo (v), el algoritmo de Bellman-Ford calcula la distancia mínima a todos los demás nodos en un grafo ponderado, **incluso si hay aristas negativas**. Funciona así:

- Asumimos que no hay ciclos negativos.
- v tiene distancia a sí mismo 0, por lo que seteamos $d_v = 0$. Para los demás nodos seteamos inicialmente su distancia a ∞ (es decir, un valor suficientemente grande).
- Iteramos n-1 veces, en cada iteración intentamos **relajar** la distancia a los nodos usando todas las aristas.
- Se puede demostrar que luego de la i-ésima iteración, ya se han encontrado todos los caminos mínimos que usan i o menos aristas.

Dado un nodo (v), el algoritmo de Bellman-Ford calcula la distancia mínima a todos los demás nodos en un grafo ponderado, **incluso si hay aristas negativas**. Funciona así:

- Asumimos que no hay ciclos negativos.
- v tiene distancia a sí mismo 0, por lo que seteamos $d_v = 0$. Para los demás nodos seteamos inicialmente su distancia a ∞ (es decir, un valor suficientemente grande).
- Iteramos n-1 veces, en cada iteración intentamos **relajar** la distancia a los nodos usando todas las aristas.
- Se puede demostrar que luego de la i-ésima iteración, ya se han encontrado todos los caminos mínimos que usan i o menos aristas.
- Hacemos 1 iteración más, si la distancia a algun nodo disminuye, el grafo contiene un ciclo negativo.

Implementación

```
1  struct edge{
2      int a, b, cost;
3  };
4  vector<int> bellmanFord(int n, int m, vector<edge> &e){
5      vector<int> d (n, INF);
6      d[v] = 0;
7      for (int i=0; i<n-1; ++i)
8          for (int j=0; j<m; ++j)
9              if (d[e[j].a] < INF)
10                 d[e[j].b] = min (d[e[j].b], d[e[j].a] + e[j].
                       cost);
11     return d;
12 }
```

La complejidad es claramente $O(n * m)$.

- El algoritmo de Floyd calcula las distancia entre **cada par** de nodos, lo hace usando **programación dinámica**.

- El algoritmo de Floyd calcula las distancia entre **cada par** de nodos, lo hace usando **programación dinámica**.
- Inicialmente, $d_{x,y}$ (distancia de x a y) es el peso de la **arista** de x a y si esta existe, ∞ si no existe, ó 0 si $x = y$.

- El algoritmo de Floyd calcula las distancia entre **cada par** de nodos, lo hace usando **programación dinámica**.
- Inicialmente, $d_{x,y}$ (distancia de x a y) es el peso de la **arista** de x a y si esta existe, ∞ si no existe, ó 0 si $x = y$.
- Procesamos los nodos uno por uno, y mantenemos como invariante que las distancias son las mínimas que se pueden lograr pasando sólo por los nodos que procesamos.

- El algoritmo de Floyd calcula las distancia entre **cada par** de nodos, lo hace usando **programación dinámica**.
- Inicialmente, $d_{x,y}$ (distancia de x a y) es el peso de la **arista** de x a y si esta existe, ∞ si no existe, ó 0 si $x = y$.
- Procesamos los nodos uno por uno, y mantenemos como invariante que las distancias son las mínimas que se pueden lograr pasando sólo por los nodos que procesamos.
- Cuando procesamos el nodo k, para cada par de nodos i, j actualizamos su distancia como el mínimo entre lo que ya había antes y la distancia si pasamos por k (es decir, $d_{i,k} + d_{k,j}$).

Implementación

```
1  void floyd(vector<vector<int> > &d){
2  // d -> matriz inicial de distancias (como se explico en diapo
       anterior)
3    for(int k=0;k<n;k++)
4      for(int i=0;i<n;i++)
5        for(int j=0;j<n;j++)
6          d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
7  }
```

- La complejidad es claramente $O(n^3)$.
- En el caso de pesos negativos, podemos saber si x está en un ciclo negativo, fijándonos si ocurre $d_{x,x} < 0$.
- La distancia de x a y es $-\infty$ si hay un nodo z que está en un ciclo negativo y podemos ir de x a z y de z a y.

Un problema de grafos

Problema: Ahorrando Gasolina

Javier quiere viajar desde la ciudad A a la ciudad B. En total hay N ciudades y M rutas doble mano que conectan un par de ciudades. El tanque del auto de Javier puede almacenar hasta C litros de gasolina. Al comenzar el recorrido Javier llenó el tanque. Para recorrer una ruta que conecta dos ciudades (i, j) debemos gastar W_{ij} litros de gasolina.
Además, en cada ciudad podemos cargar el tanque al precio de D_i pesos por litro. ¿Cuál es la menor cantidad de dinero que debemos gastar para llegar de A a B? ¿Y de A a cada una de las otras ciudades?

- Al buscar la solución, necesitamos saber **en qué ciudad estamos**, pero también **cuánta gasolina nos queda**.

- Al buscar la solución, necesitamos saber **en qué ciudad estamos**, pero también **cuánta gasolina nos queda**.
- Podemos definir un grafo donde cada nodo esta compuesto por este par de valores y trabajar sobre este. A estos nodos los llamaremos **estados**.

- Al buscar la solución, necesitamos saber **en qué ciudad estamos**, pero también **cuánta gasolina nos queda**.
- Podemos definir un grafo donde cada nodo esta compuesto por este par de valores y trabajar sobre este. A estos nodos los llamaremos **estados**.

¿Cuáles son nuestras posibles transiciones (aristas) si actualmente estamos en la ciudad i con l litros de gasolina en el tanque? ¿Qué decisiones podemos tomar?

- Al buscar la solución, necesitamos saber **en qué ciudad estamos**, pero también **cuánta gasolina nos queda**.
- Podemos definir un grafo donde cada nodo esta compuesto por este par de valores y trabajar sobre este. A estos nodos los llamaremos **estados**.

¿Cuáles son nuestras posibles transiciones (aristas) si actualmente estamos en la ciudad i con l litros de gasolina en el tanque? ¿Qué decisiones podemos tomar?

- Podemos *movernos a otra ciudad* (si es que nos da la gasolina en el tanque):

- Al buscar la solución, necesitamos saber **en qué ciudad estamos**, pero también **cuánta gasolina nos queda**.
- Podemos definir un grafo donde cada nodo esta compuesto por este par de valores y trabajar sobre este. A estos nodos los llamaremos **estados**.

¿Cuáles son nuestras posibles transiciones (aristas) si actualmente estamos en la ciudad i con l litros de gasolina en el tanque? ¿Qué decisiones podemos tomar?

- Podemos *movernos a otra ciudad* (si es que nos da la gasolina en el tanque): $(i, l) \xrightarrow{0} (j, l - W_{ij})$.
- Podemos *cargar un litro de gasolina* en la ciudad actual (si el tanque no está lleno):

- Al buscar la solución, necesitamos saber **en qué ciudad estamos**, pero también **cuánta gasolina nos queda**.
- Podemos definir un grafo donde cada nodo esta compuesto por este par de valores y trabajar sobre este. A estos nodos los llamaremos **estados**.

¿Cuáles son nuestras posibles transiciones (aristas) si actualmente estamos en la ciudad i con l litros de gasolina en el tanque? ¿Qué decisiones podemos tomar?

- Podemos *movernos a otra ciudad* (si es que nos da la gasolina en el tanque): $(i, l) \xrightarrow{0} (j, l - W_{ij})$.
- Podemos *cargar un litro de gasolina* en la ciudad actual (si el tanque no está lleno): $(i, l) \xrightarrow{C_i} (i, l + 1)$

- Al buscar la solución, necesitamos saber **en qué ciudad estamos**, pero también **cuánta gasolina nos queda**.
- Podemos definir un grafo donde cada nodo esta compuesto por este par de valores y trabajar sobre este. A estos nodos los llamaremos **estados**.

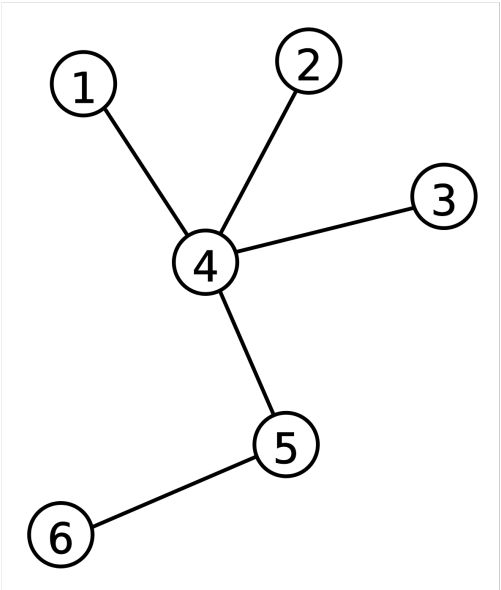
¿Cuáles son nuestras posibles transiciones (aristas) si actualmente estamos en la ciudad i con l litros de gasolina en el tanque? ¿Qué decisiones podemos tomar?

- Podemos *movernos a otra ciudad* (si es que nos da la gasolina en el tanque): $(i, l) \xrightarrow{0} (j, l - W_{ij})$.
- Podemos *cargar un litro de gasolina* en la ciudad actual (si el tanque no está lleno): $(i, l) \xrightarrow{C_i} (i, l + 1)$

Luego, el problema se reduce a encontrar un camino mínimo de (A, C) a algún nodo de la forma (B, l) para algún l.

Para muchos problemas puede ser útil tratar de **modelar** un grafo, de tal forma que lo que pide el problema se transforme en una "consulta" sobre ese grafo. La forma de hacer este modelado suele ser con un **vértice por cada estado posible** distinto que puede haber en el problema y una **arista por cada transición posible entre dos estados**.

- Un **árbol** es un grafo conexo sin ciclos.
- Una particularidad de los árboles es que siempre $|E| = |V| - 1$.



Union-find es una **estructura de datos** que mantiene las componentes conexas de un grafo. Soporta las siguientes dos operaciones:

- $find(x)$: Devuelve un *id* de la componente conexa en la que está el nodo x. Se puede usar para ver si dos nodos están en la misma componente chequeando $find(x) == find(y)$.
- $union(x, y)$: Agrega una arista entre los nodos x e y. Es decir, junta las componentes de x y de y en la misma componente.

Union-find es una **estructura de datos** que mantiene las componentes conexas de un grafo. Soporta las siguientes dos operaciones:

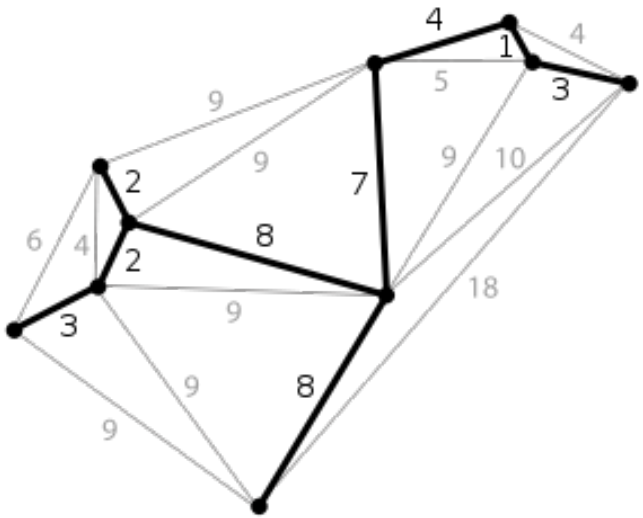
- $find(x)$: Devuelve un *id* de la componente conexa en la que está el nodo x. Se puede usar para ver si dos nodos están en la misma componente chequeando $find(x) == find(y)$.
- $union(x, y)$: Agrega una arista entre los nodos x e y. Es decir, junta las componentes de x y de y en la misma componente.

Ambas operaciones se pueden realizar de manera muy **eficiente** (prácticamente $O(1)$).

Definición: árbol generador mínimo

Dado un grafo conexo G, un árbol generador de G es un árbol que tiene todos los nodos de G y cuyas aristas también son aristas de G.

Si las aristas de G tienen pesos, un "árbol generador mínimo" de G es uno que cumple que la suma de sus aristas es lo más chica posible.



El algoritmo de **Kruskal** calcula el árbol generador mínimo como sigue:

El algoritmo de **Kruskal** calcula el árbol generador mínimo como sigue:

- Mantengo un union-find con las componentes conexas determinadas por las aristas que agregué al árbol.
- Armo un arreglo con todas las aristas. Las **ordeno por peso** de menor a mayor.

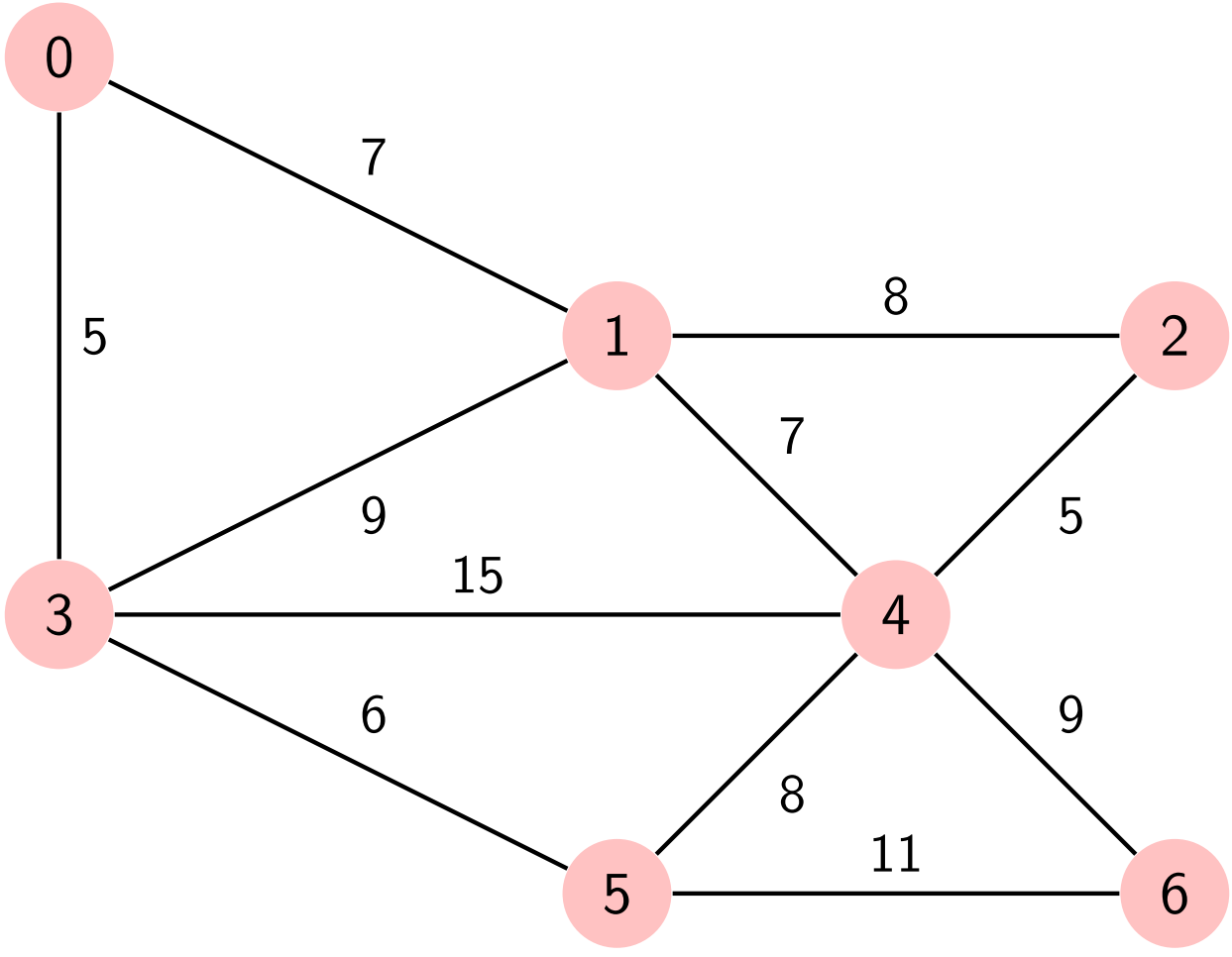
El algoritmo de **Kruskal** calcula el árbol generador mínimo como sigue:

- Mantengo un union-find con las componentes conexas determinadas por las aristas que agregué al árbol.
- Armo un arreglo con todas las aristas. Las **ordeno por peso** de menor a mayor.
- Recorro cada arista: Si los nodos que conecta están en la misma componente, entonces no hago nada. Si están en distintas, agrego la arista al árbol y hago **union** de los dos nodos en el union-find.

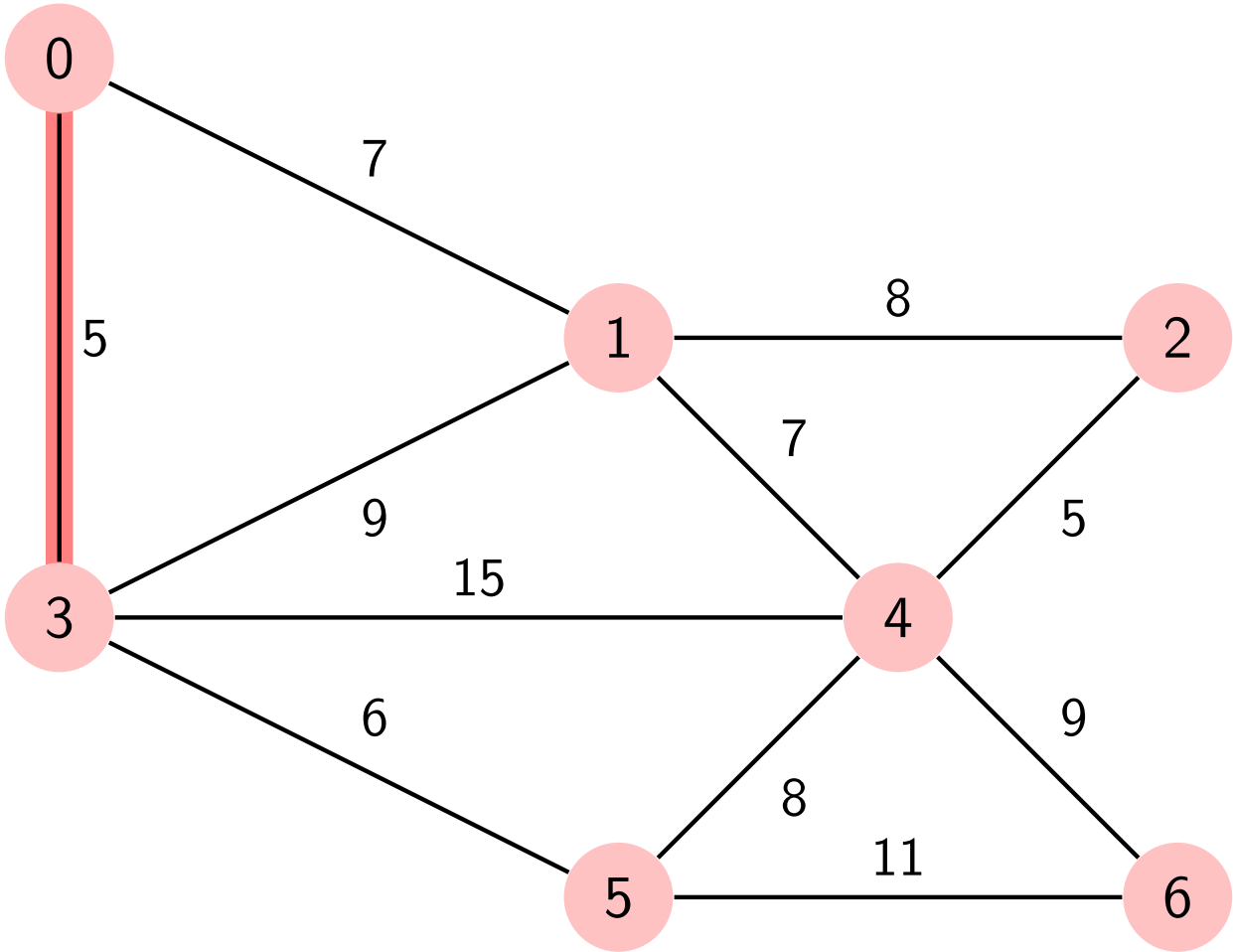
El algoritmo de **Kruskal** calcula el árbol generador mínimo como sigue:

- Mantengo un union-find con las componentes conexas determinadas por las aristas que agregué al árbol.
- Armo un arreglo con todas las aristas. Las **ordeno por peso** de menor a mayor.
- Recorro cada arista: Si los nodos que conecta están en la misma componente, entonces no hago nada. Si están en distintas, agrego la arista al árbol y hago **union** de los dos nodos en el union-find.
- La **complejidad** es lineal, excepto en la parte de ordenar las aristas. Entonces en total es $O(|E|log(|E|))$
- Para Kruskal, no hace falta representar el grafo como lista de adyacencia, sino simplemente como un arreglo con las aristas.

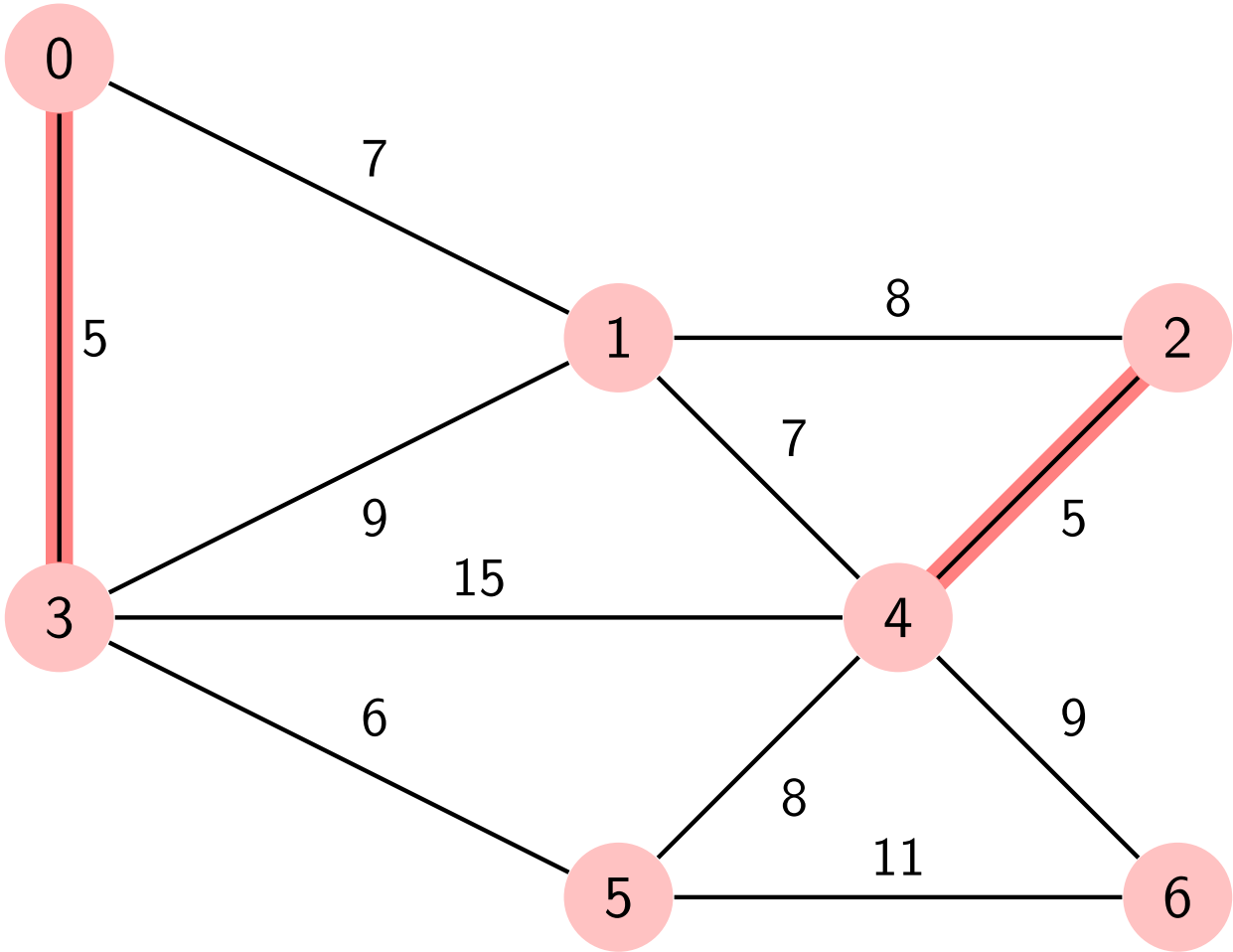
Ejemplo



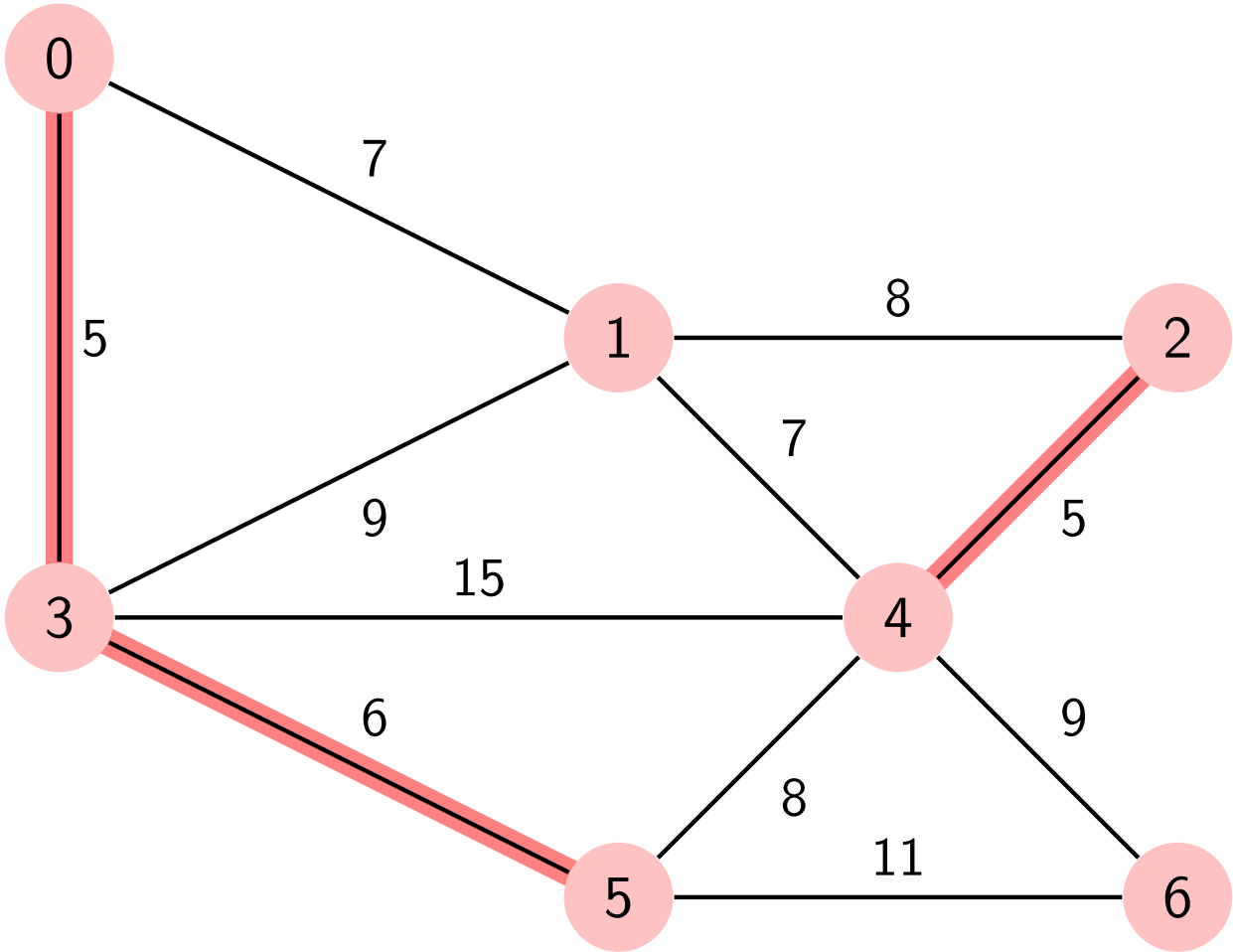
Ejemplo



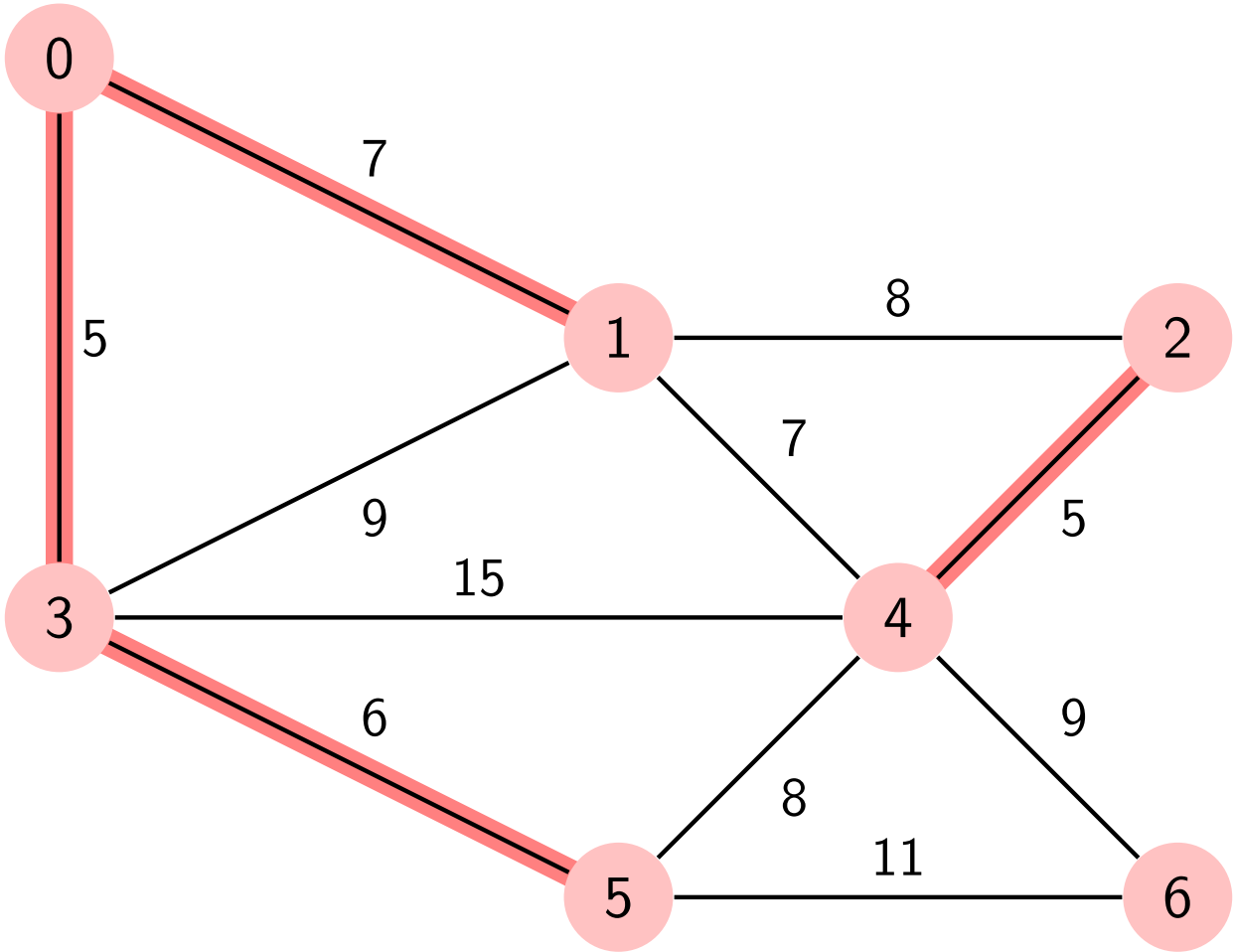
Ejemplo



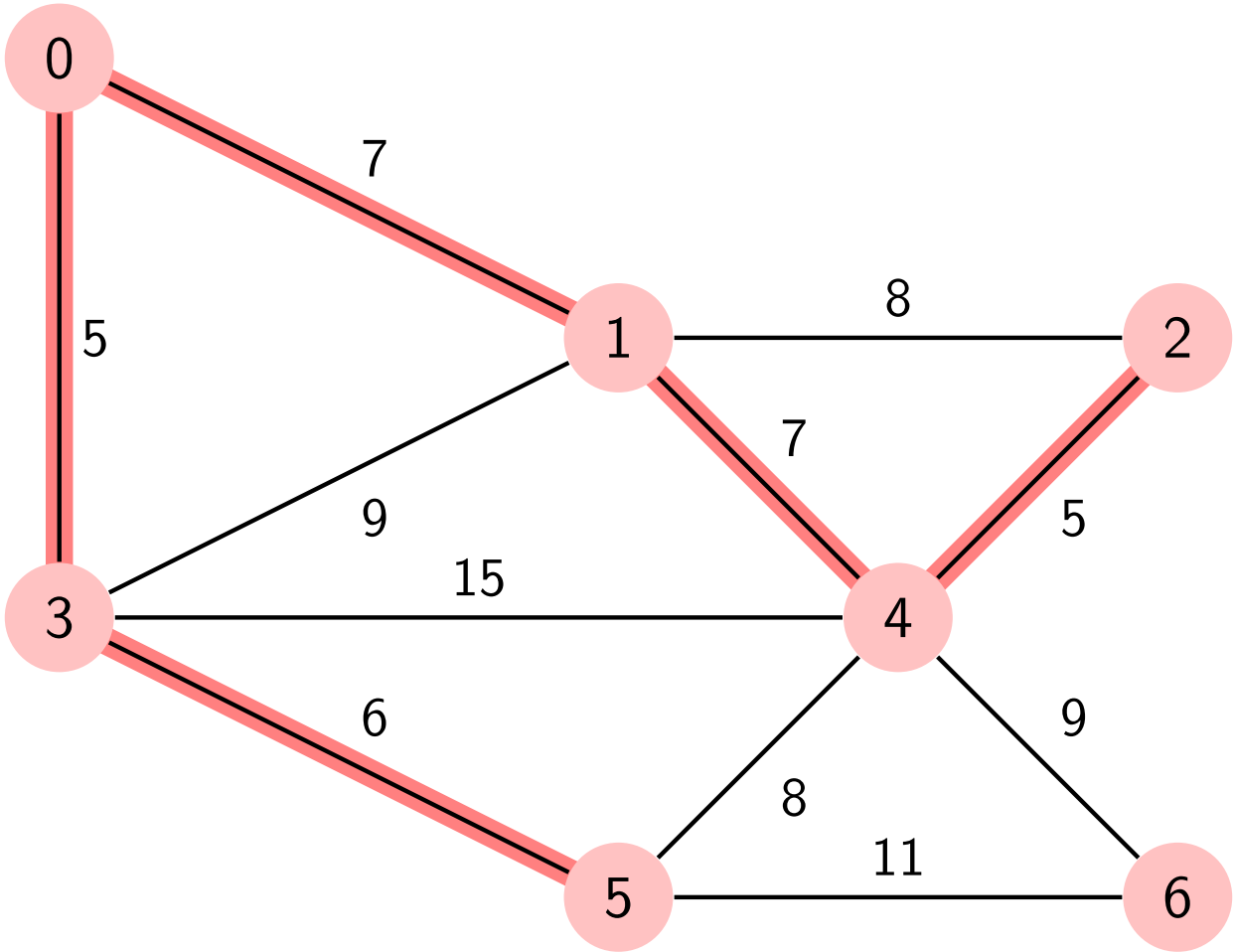
Ejemplo



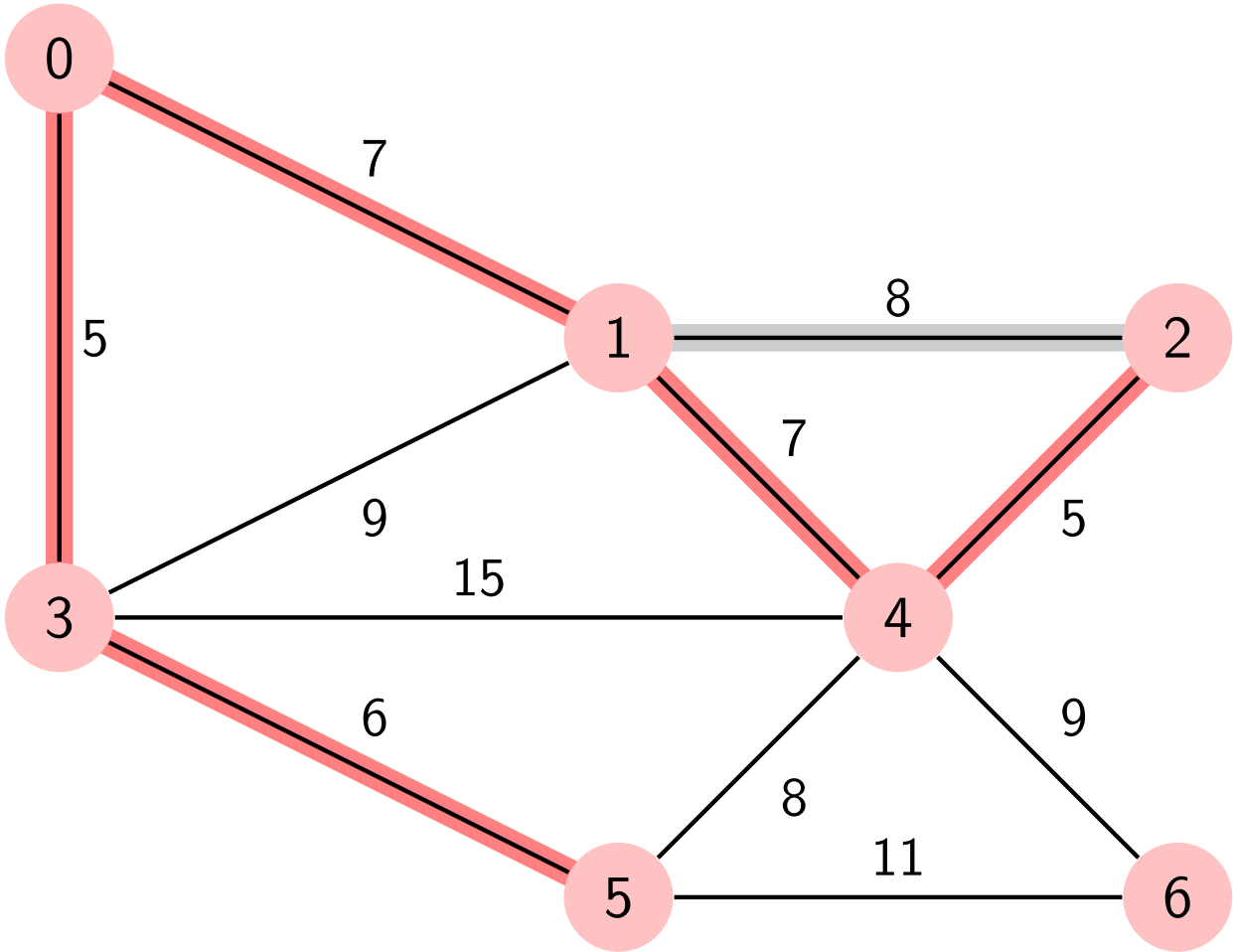
Ejemplo



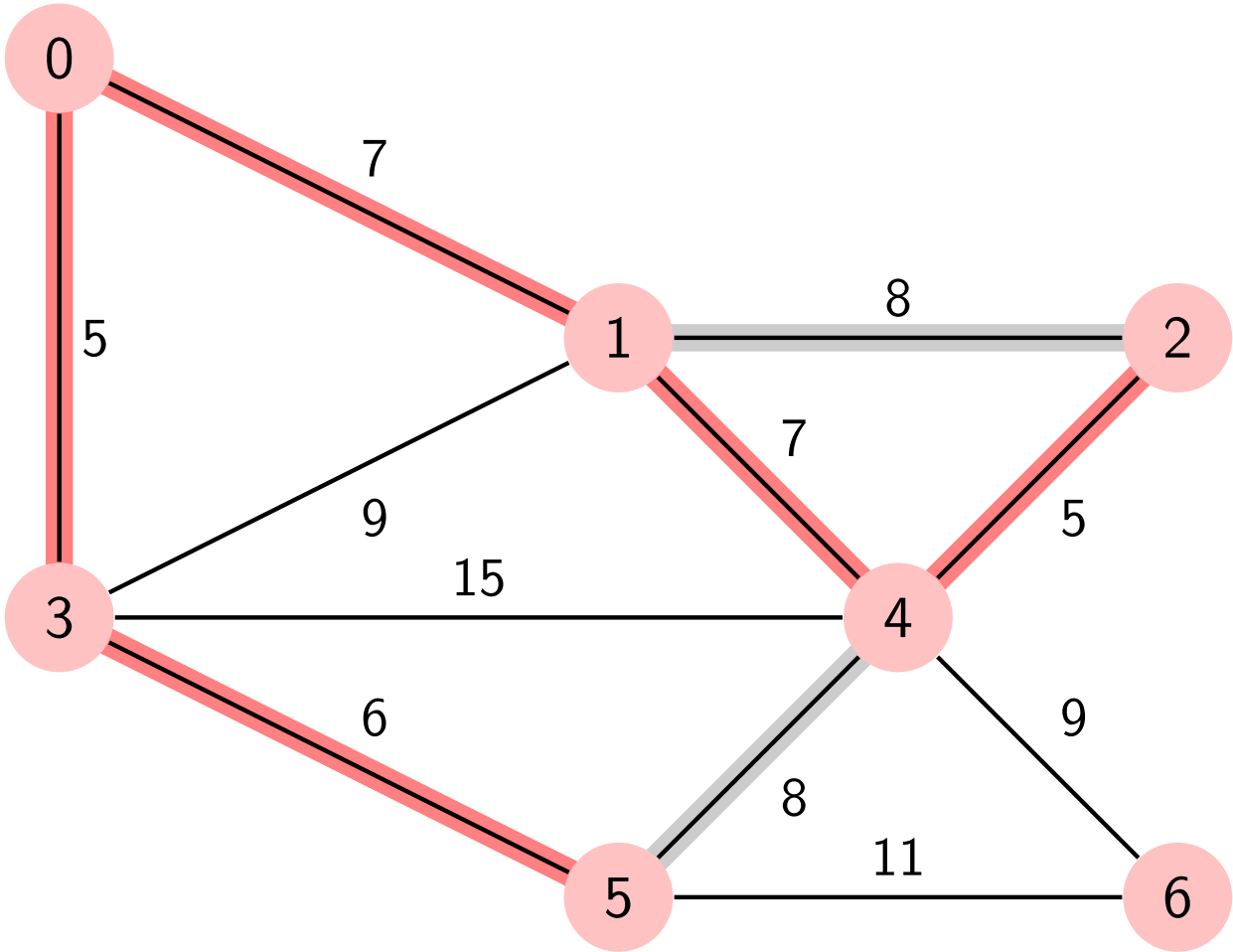
Ejemplo



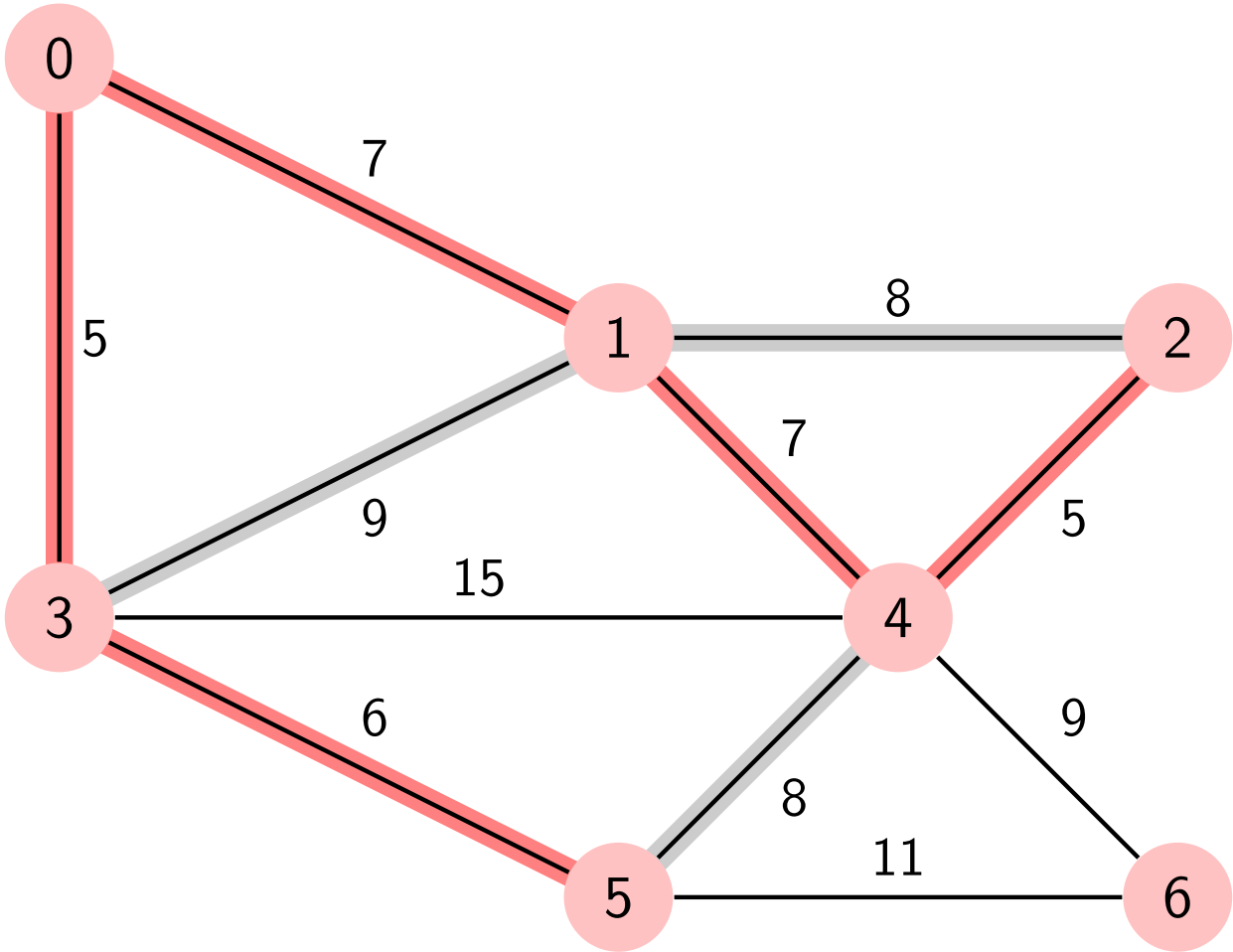
Ejemplo



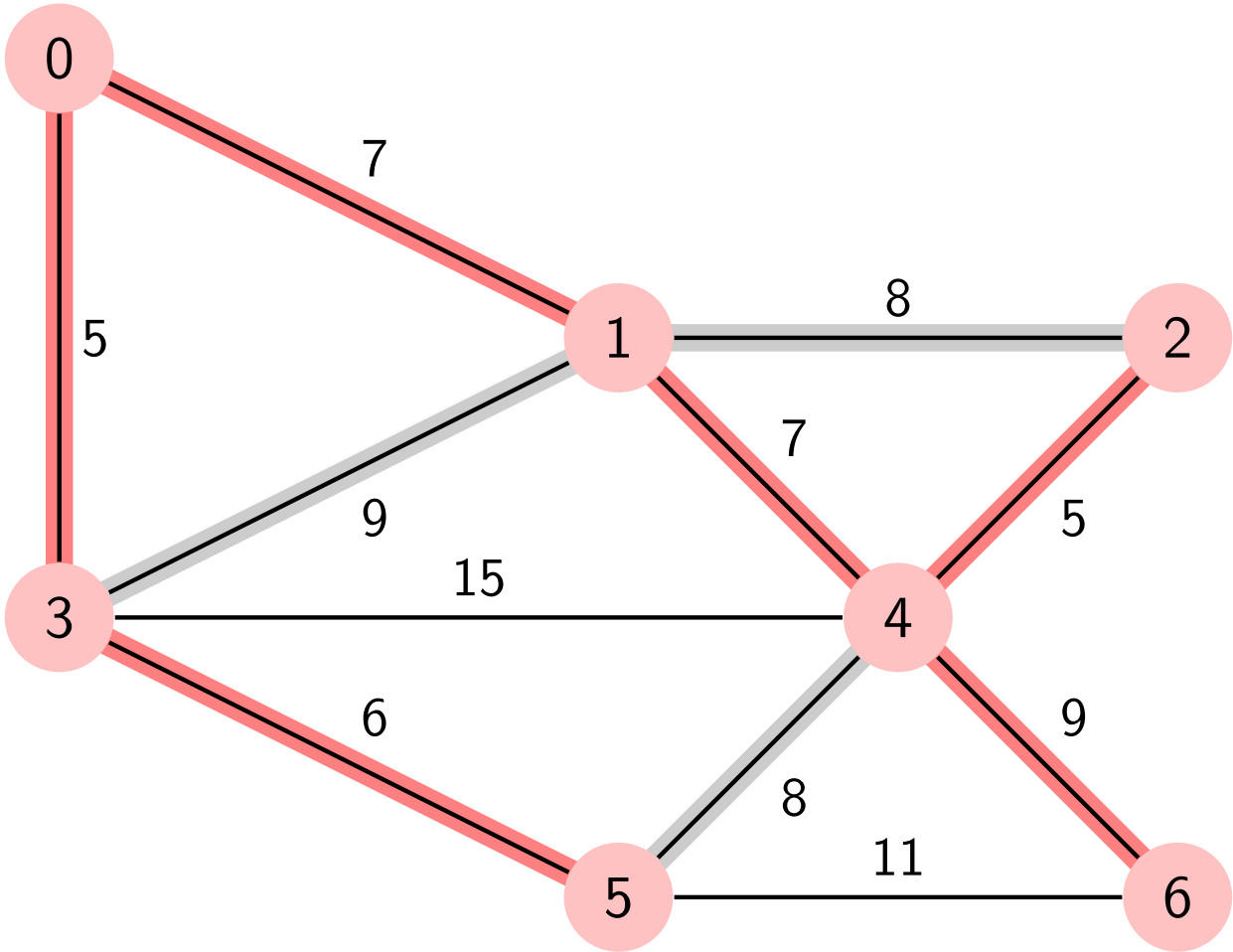
Ejemplo



Ejemplo



Ejemplo



Implementación

```
1  int kruskal(vector<pair<int, pair<int, int>>> edges, int n) {
2    //edges: lista de aristas en la forma {peso, {nodo1, nodo2}}
3    sort(edges.begin(),edges.end()); // ordena por peso
4    init_uf(n);
5    int u = 0, res = 0;
6    for(auto p: edges){
7      int c = p.first, x = p.second.first, y = p.second.second;
8      x = find(x); y = find(y);
9      if(x == y) continue // los nodos ya estan conectados
10     res += c;
11     u++;
12     join(x,y);
13     if(u == n - 1) // completamos el arbol?
14       return res;
15   }
16   return -1; // si llegamos hasta aca entonces no es conexo
17 }
```