Grafos

Héctor Peña Pollastri[1]

[1]FaMAF
Universidad Nacional de Córdoba

Training Camp 2019

Contenidos

Definición de Grafo

Wikipedia

Un grafo es un conjunto, no vacío, de objetos llamados vértices (o nodos) y una selección de pares de vértices, llamados aristas (edges en inglés) que pueden ser orientados (dirigidos) o no.

Real Academia Española

Diagrama que representa mediante puntos y líneas las relaciones entre pares de elementos y que se usa para resolver problemas lógicos, topológicos y de cálculo combinatorio.

La Posta

Un grafo es un conjunto de puntos y líneas que unen pares de esos puntitos.

Ejemplos

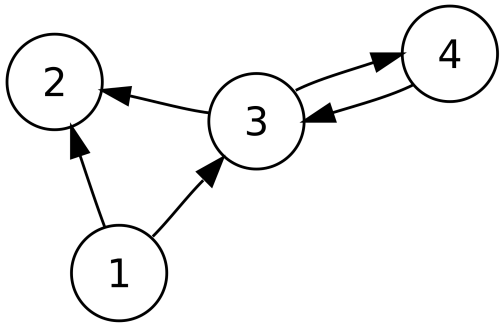


Figura: Grafo dirigido

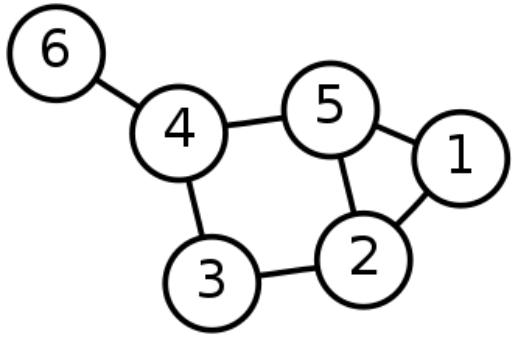


Figura: Grafo no dirigido

Más Ejemplos

Ejemplo: Redes sociales

- En Facebook, solo podemos ver el contenido de la otra persona si son 'mutuamente amigos'. La relación Juancito es amigo de María es la de un grafo no dirigido, pues tienen que ser mutuamente amigos para ver lo que comparten.
- En Instagram o Twitter para uno puede seguir a alguien sin necesidad de que la otra persona nos siga. En tal caso la relación Juancito sigue a María se puede representar como la de un grafo dirigido.

Ejemplo:

Conjunto de personas (vértices), donde algunos pares de personas se conocen (aristas).

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

Formas de representar un Grafo

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.

Lista 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. Esta representación es la que usaremos para los algoritmos que recorren un grafo (DFS Y BFS por ejemplo).

A partir de ahora en nuestras implementaciones n será la cantidad de nodos y m la cantidad de ejes.

Formas de representar un Grafo

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.

Lista 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. Esta representación es la que usaremos para los algoritmos que recorren un grafo (DFS Y BFS por ejemplo).

A partir de ahora en nuestras implementaciones n será la cantidad de nodos y m la cantidad de ejes.

Implementaciones de 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);
}
}
```

Implementaciones de 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 }
```

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  vector<vector<int> > graph;
6  int n,m;
7
8  int main(){
9  cin>>n>>m;
10 graph.resize(n);
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 de 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 }
```

Contenidos

Distancia

- Un camino de largo n de v a w es una lista de $n+1$ vértices $v = v_0, v_1, \ldots, v_n = w$ tales que para $0 \leq i < n$ tenemos $(v_i, v_{i+1}) \in E$.
- Definimos la *distancia* de v a w como el menor n tal que hay un camino de largo n de v a w
- Más coloquialmente: La distancia es la menor cantidad de aristas que tenemos que "recorrer" para ir de un nodo al otro.
- Un problema muy típico es dado un grafo y dos nodos, calcular la distancia de uno a otro. Se puede resolver con un algoritmo llamado BFS.

BFS

BFS (siglas de *breadth-first-search* o “búsqueda en anchura”) calcula la distancia mínima desde un nodo v a cada uno de los otros como sigue:

- v tiene distancia a sí mismo 0, por lo que seteamos $d_v = 0$. Además v es el único nodo 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 vértices a distancia 1.
- Ahora tomamos los vértices a distancia 1, y para cada uno nos fijamos en sus vecinos cuya distancia aún no calculamos. Estos son los vértices a distancia 2.
- Así sucesivamente, para saber cuáles son los vertices a distancia $k + 1$, tomamos los vecinos de un nodo a distancia k que no hayan sido visitados aún.

BFS - Cont.

Podemos implementar BFS como sigue:

- Mantenemos un arreglo d con las distancias desde v que ya calculamos (a las que no calculamos las identificamos con un valor especial, por ejemplo: -1).
- Mantenemos una *queue* que contenga los nodos cuya distancia conocemos y que aún no procesamos.
- Inicialmente asignamos $d_v = 0$ y agregamos v a la *queue*.
- En cada paso tomamos el primer nodo de la *queue* (llamémoslo x), nos fijamos en cada vecino y. Si todavía no lo vimos ($d_y = -1$) entonces asignamos $d_y = d_x + 1$ y lo agregamos a la *queue* para procesarlo más adelante.
- Como de la *queue* vamos sacando el primer elemento que entró, eso nos asegura que vamos recorriendo los nodos en orden creciente de distancia.

BFS - Código

```
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 esta visitado
11         d[y] = d[x] + 1;
12         q.push(y);
13       }
14     }
15   }
16   return d; // d contiene las distancias desde v
17 }           // (o -1 para los nodos inalcanzables)
```

BFS - 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 $O(|E|)$.

Contenidos

DFS

Además de BFS, otra manera de recorrer un grafo es DFS (*depth-first-search* o "búsqueda en profundidad"). Funciona así.

- Se puede implementar igual que un BFS pero cambiando una queue por un stack.
- Mantenemos un arreglo que nos dice para cada nodo si fue visitado o no.
- Arrancamos de cierto nodo. Lo marcamos como visitado. Luego nos fijamos en sus vecinos, y para todos los vecinos que no hayan sido visitados recorremos desde ahí. 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.

Implementación con un stack propio)

```
1  void DFS(int node){
2  stack<int> s;
3  s.push(node);
4  encolado[node] = true;
5  while(s.size()){
6  int current = s.top();
7  s.pop();
8  cout<<"Estoy viendo el nodo: "<<current<<endl;
9  for(int i=0;i<graph[current].size();i++){
10 int t = graph[current][i];
11 if(!encolado[t]){
12 s.push(t);
13 encolado[t] = true;
14 }
15 }
16 }
17 }
```

Implementación recursiva

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  vector<vector<int > > graph;
5  vector<bool> visit;
6  int n,m;
7
8  void DFS(int node){
9  visit[node] = true;
10 cout<<"estoy viendo el nodo: "<<node<<endl;
11 for(int i=0;i<graph[node].size();i++){
12 int current = graph[node][i];
13 if(!visit[current]) DFS(current);
14 }
15 }
```

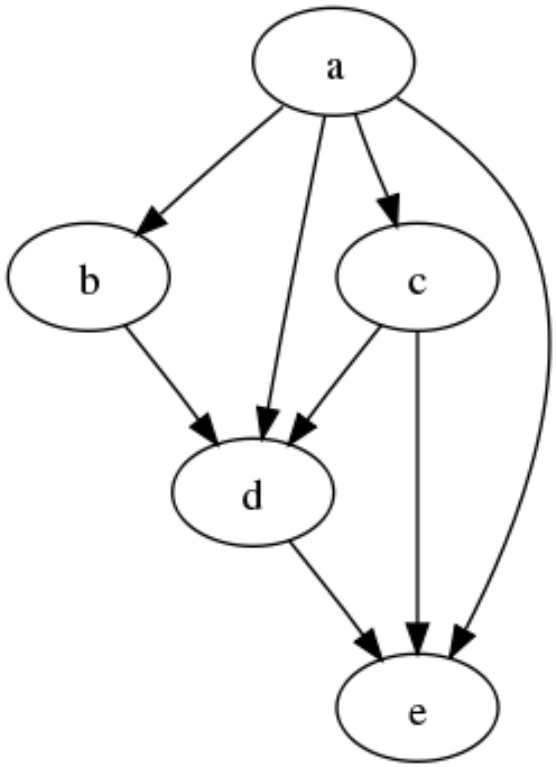
DFS - Análisis

- Un análisis similar al que hicimos para BFS concluye que DFS también es $O(|E|)$.
- 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 el que veremos ahora.

Ejemplo - orden topológico

Orden topológico

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



Este grafo tiene dos órdenes topológicos:

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

Ejemplo - orden topológico (cont.)

- 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).
- Si el grafo no tiene ciclo, podemos encontrar el orden topológico de la siguiente forma:
  - Llamamos a DFS para todos los nodos (excepto los que visitamos en un paso anterior).
  - Cuando termina el DFS desde cada nodo, agregamos el nodo al principio del orden topológico.
- Realizando este algoritmo, tenemos que cuando agregamos un nodo x al principio, entonces ya procesamos todos los nodos que se pueden alcanzar desde x. Entonces, va antes que todos esos en el orden.

Ejemplo - orden topológico - código

```
1  vector<int> topo;
2  bool vis[MAXN];
3  void dfs(int x){
4    vis[x] = true;
5    for(int y: g[x])
6      if(!vis[y])
7        dfs(y);
8    topo.push_back(y);
9  }
10 void init_topo(){
11   topo.clear();
12   for(int i=0; i<n; ++i){
13     if(!vis[i]){
14       dfs(i);
15     }
16   }
17   reverse(topo.begin(), topo.end());
18 }
```

Contenidos

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 $d = \sum_{i=0}^{n-1} p_i$.

Algoritmo de Dijkstra

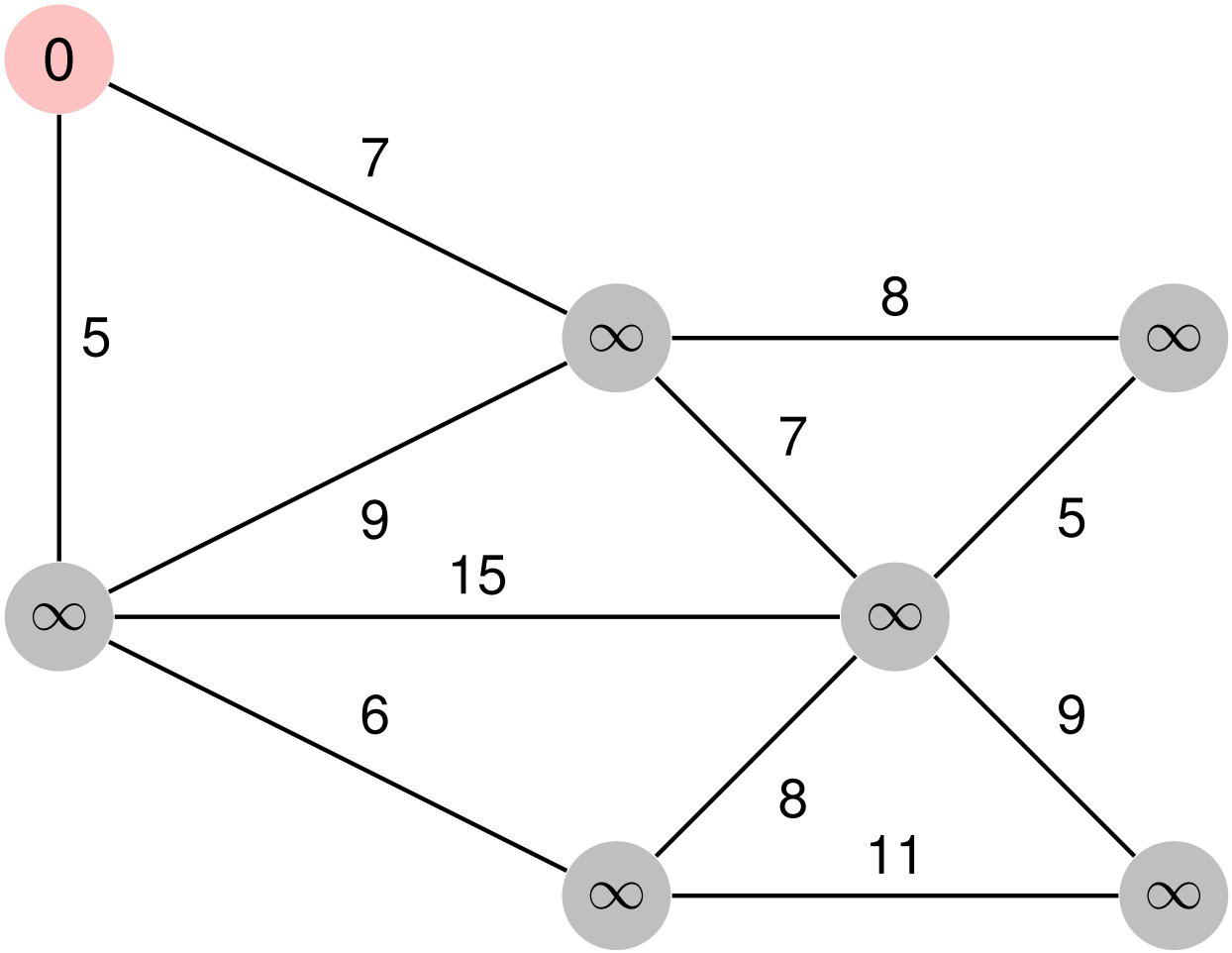
- El BFS ya no sirve para calcular el camino mínimo entre dos nodos de un grafo ponderado.
- Para solucionar ese problema, existe, entre otros, el algoritmo de Dijkstra.
- El algoritmo de Dijkstra calcula dado un vértice la distancia mínima a todos los demás vértices desde ese vértice en un grafo ponderado.

Qué hace el algoritmo de Dijkstra?

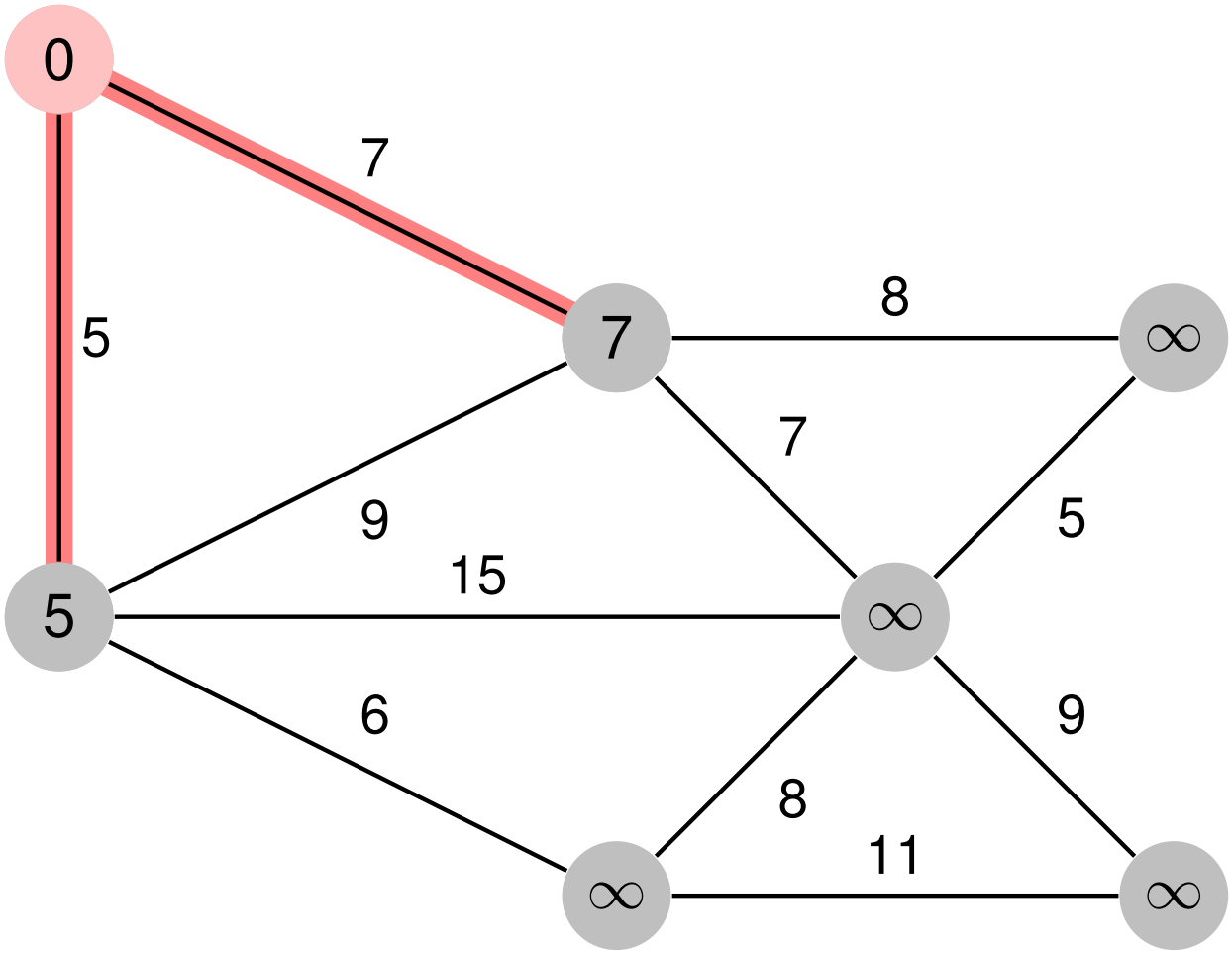
El algoritmo de Dijkstra funciona como sigue:

- El input es un grafo con pesos, y un nodo v desde el cual queremos calcular la distancia a todos los nodos.
- v tiene distancia a sí mismo 0, por lo que seteams $d_v = 0$. 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)).

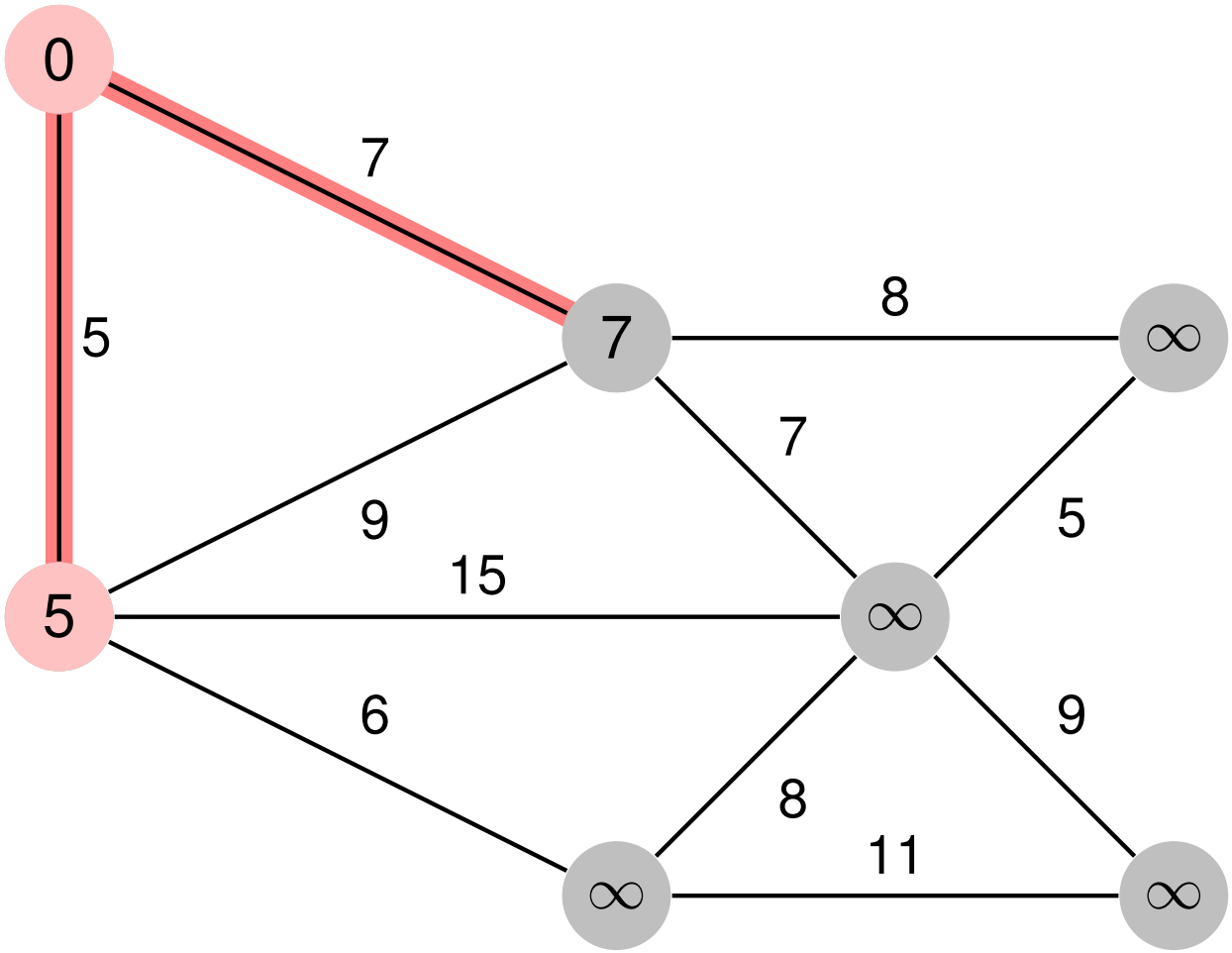
Ejemplo de ejecución de Dijkstra



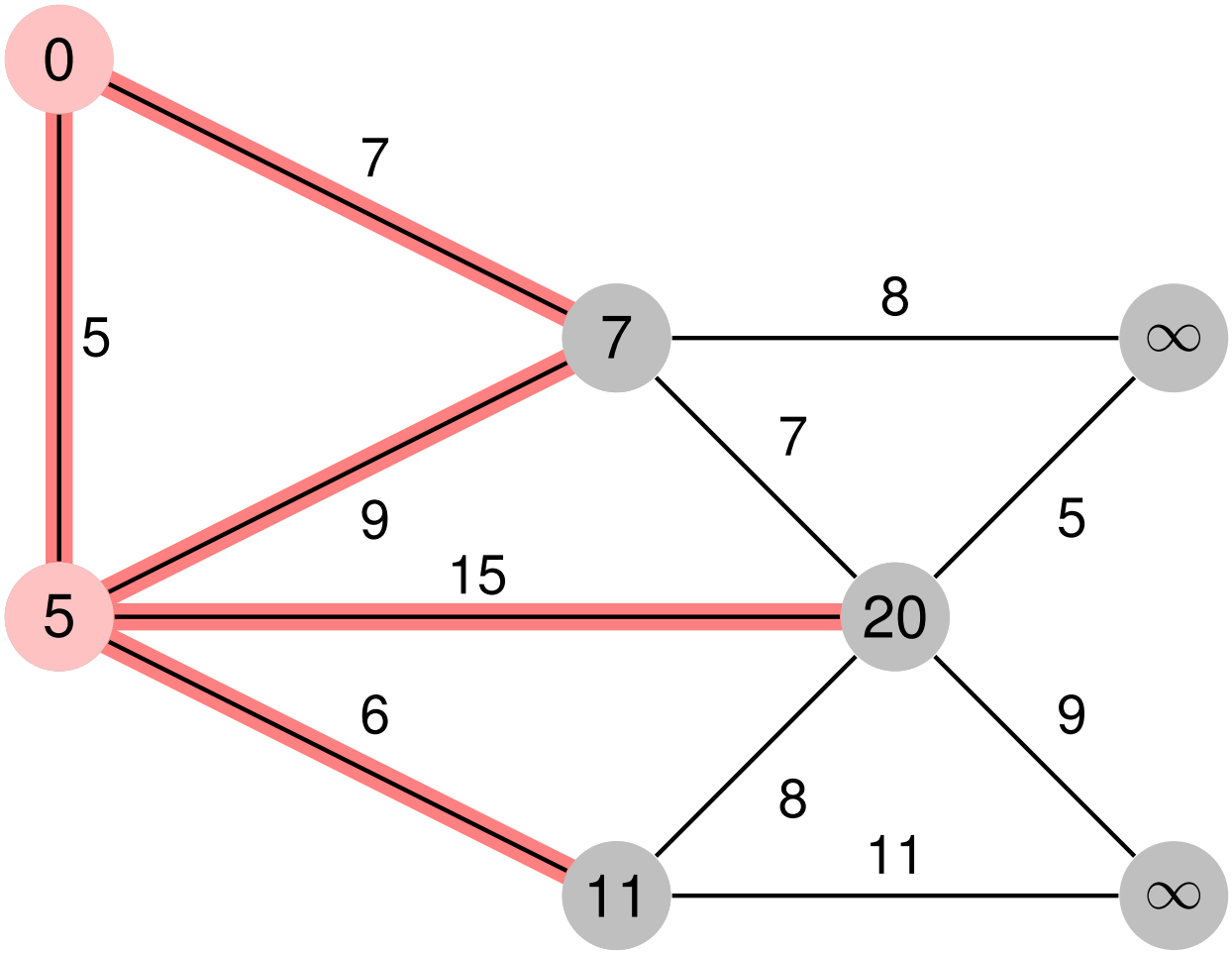
Ejemplo de ejecución de Dijkstra



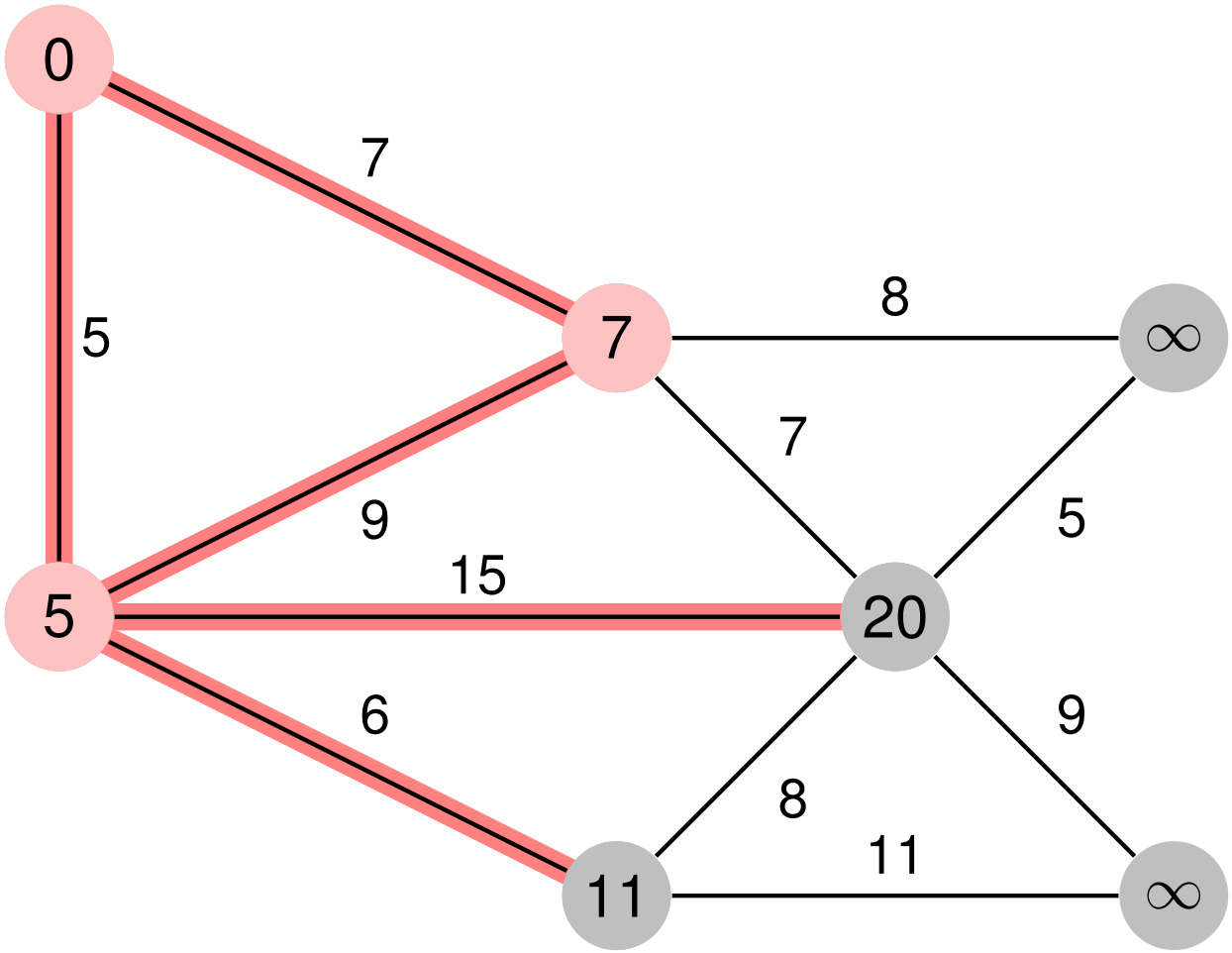
Ejemplo de ejecución de Dijkstra



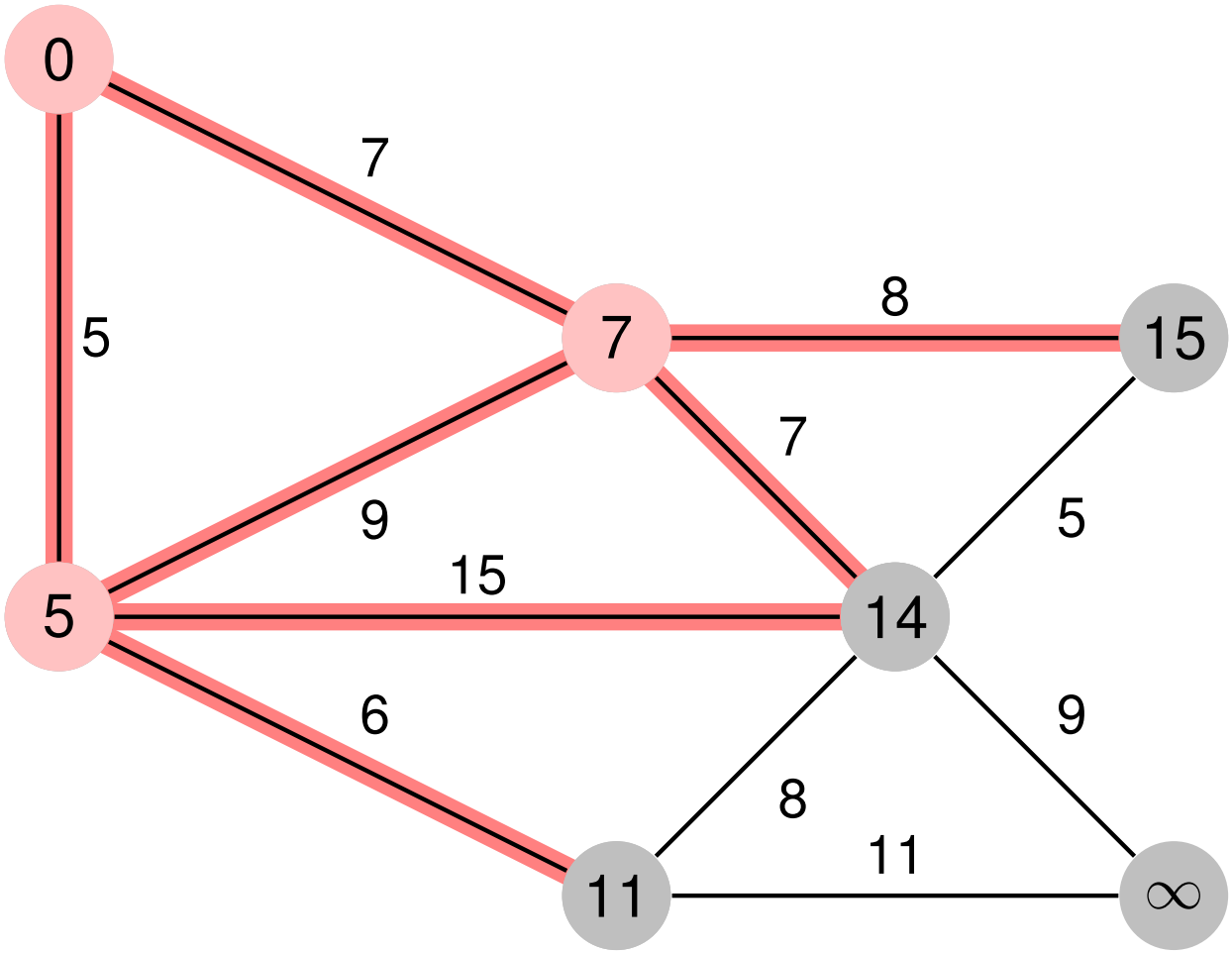
Ejemplo de ejecución de Dijkstra



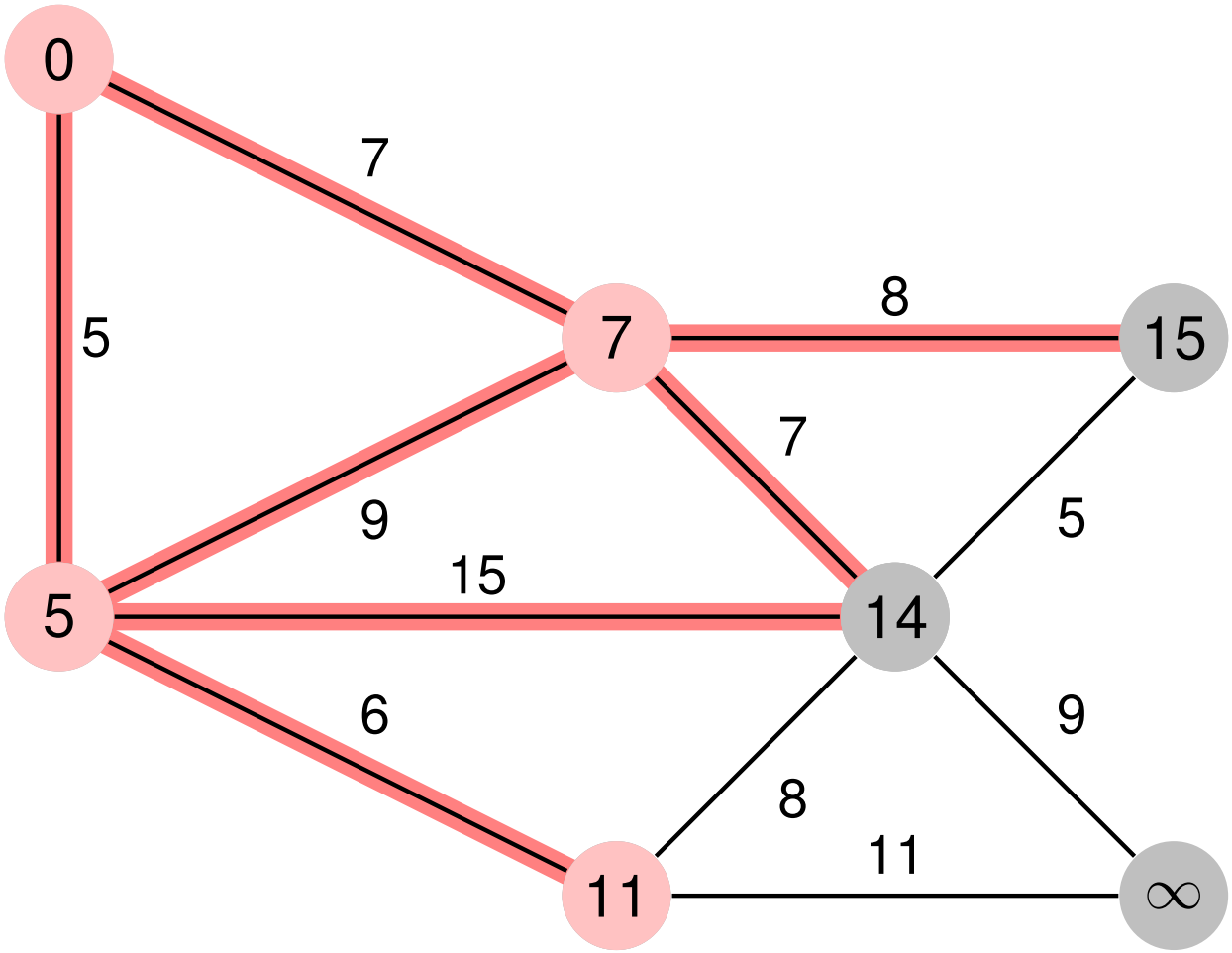
Ejemplo de ejecución de Dijkstra



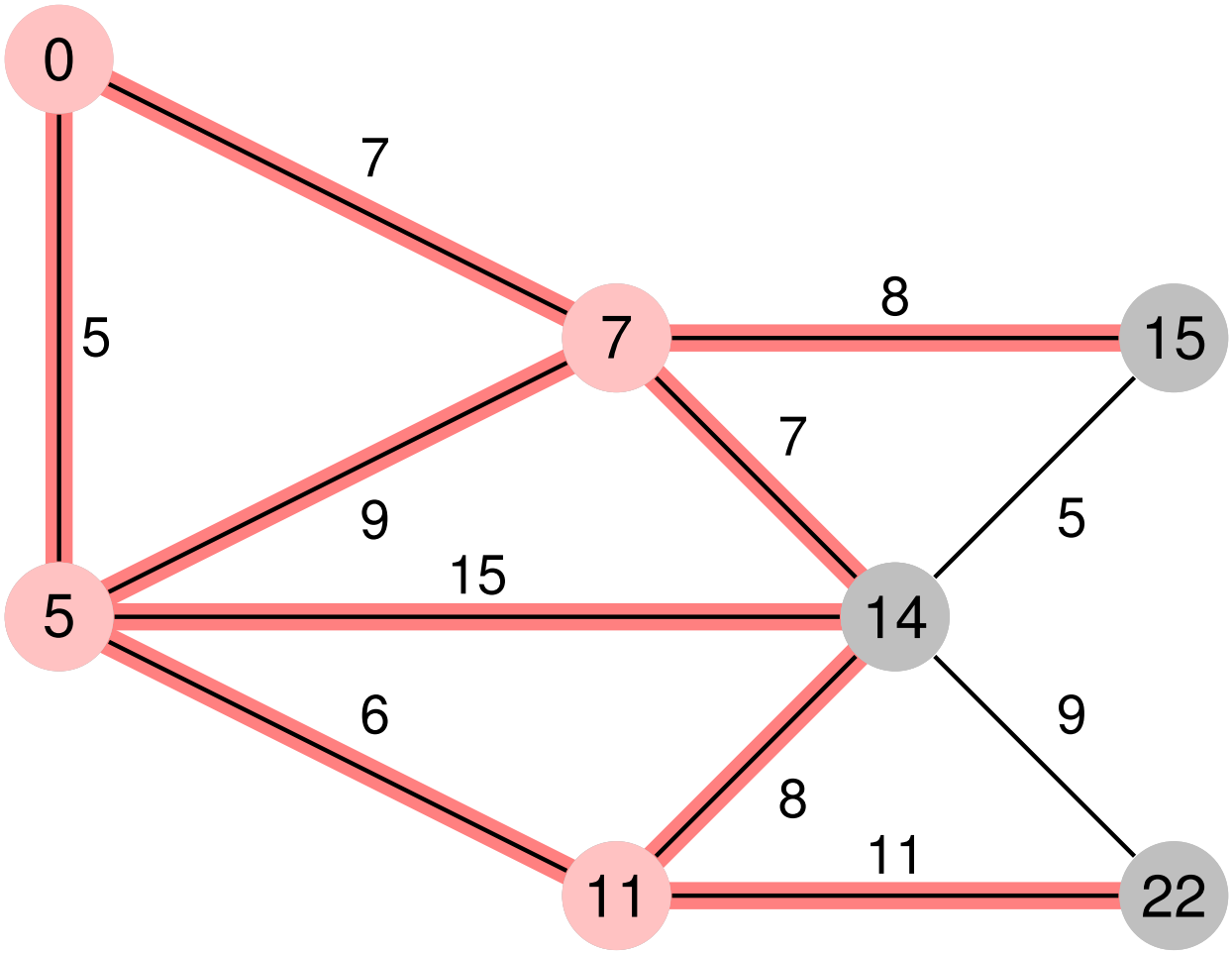
Ejemplo de ejecución de Dijkstra



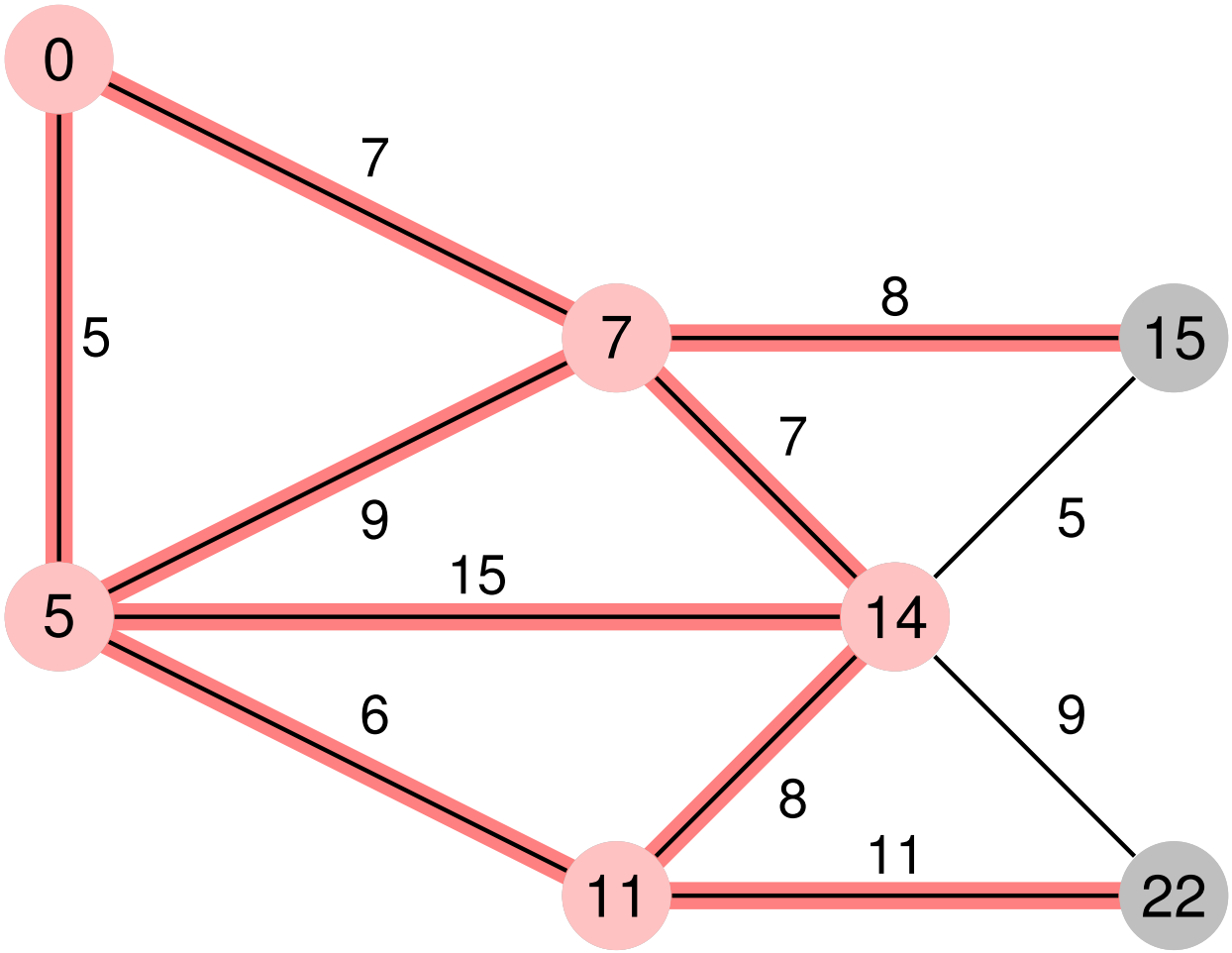
Ejemplo de ejecución de Dijkstra



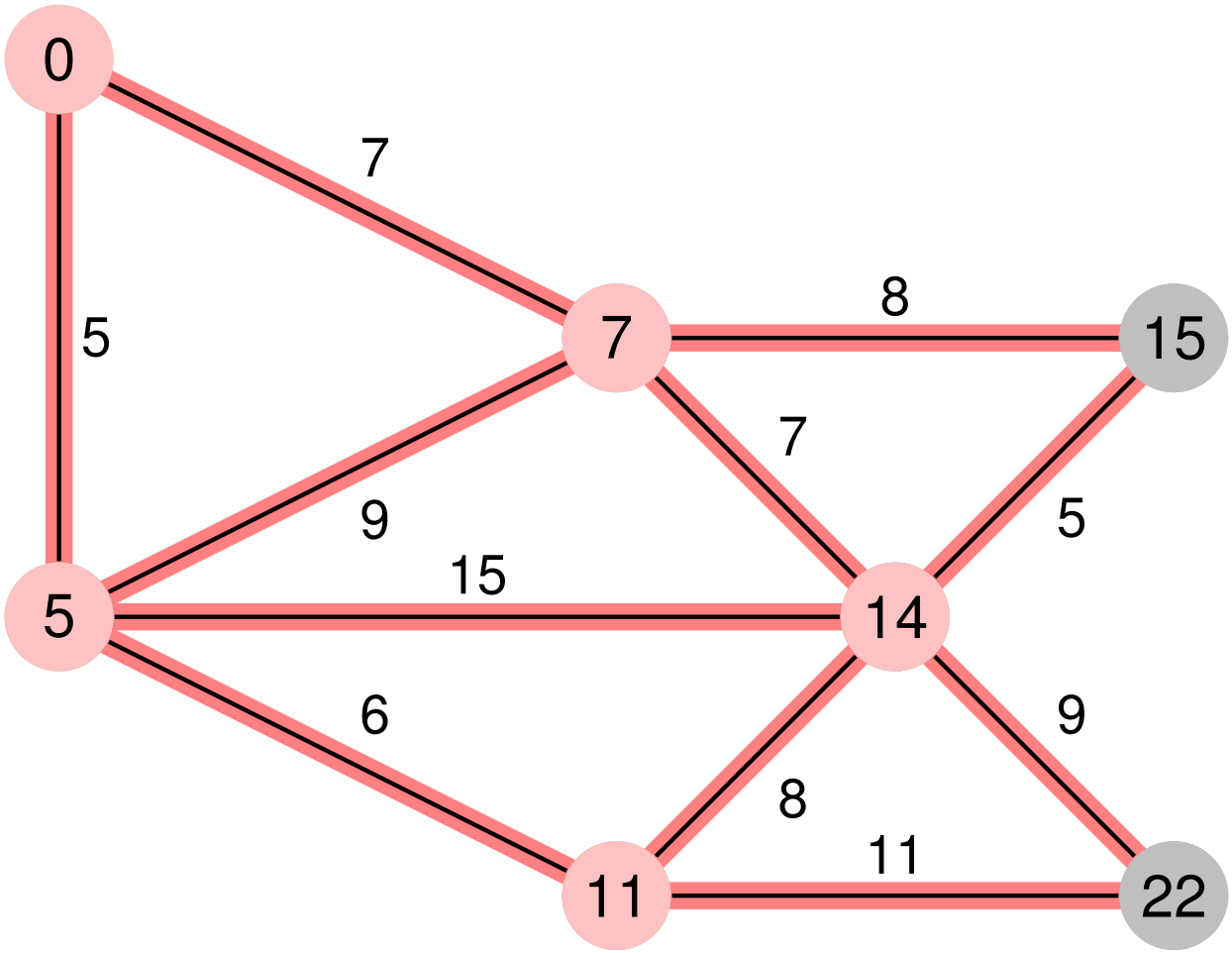
Ejemplo de ejecución de Dijkstra



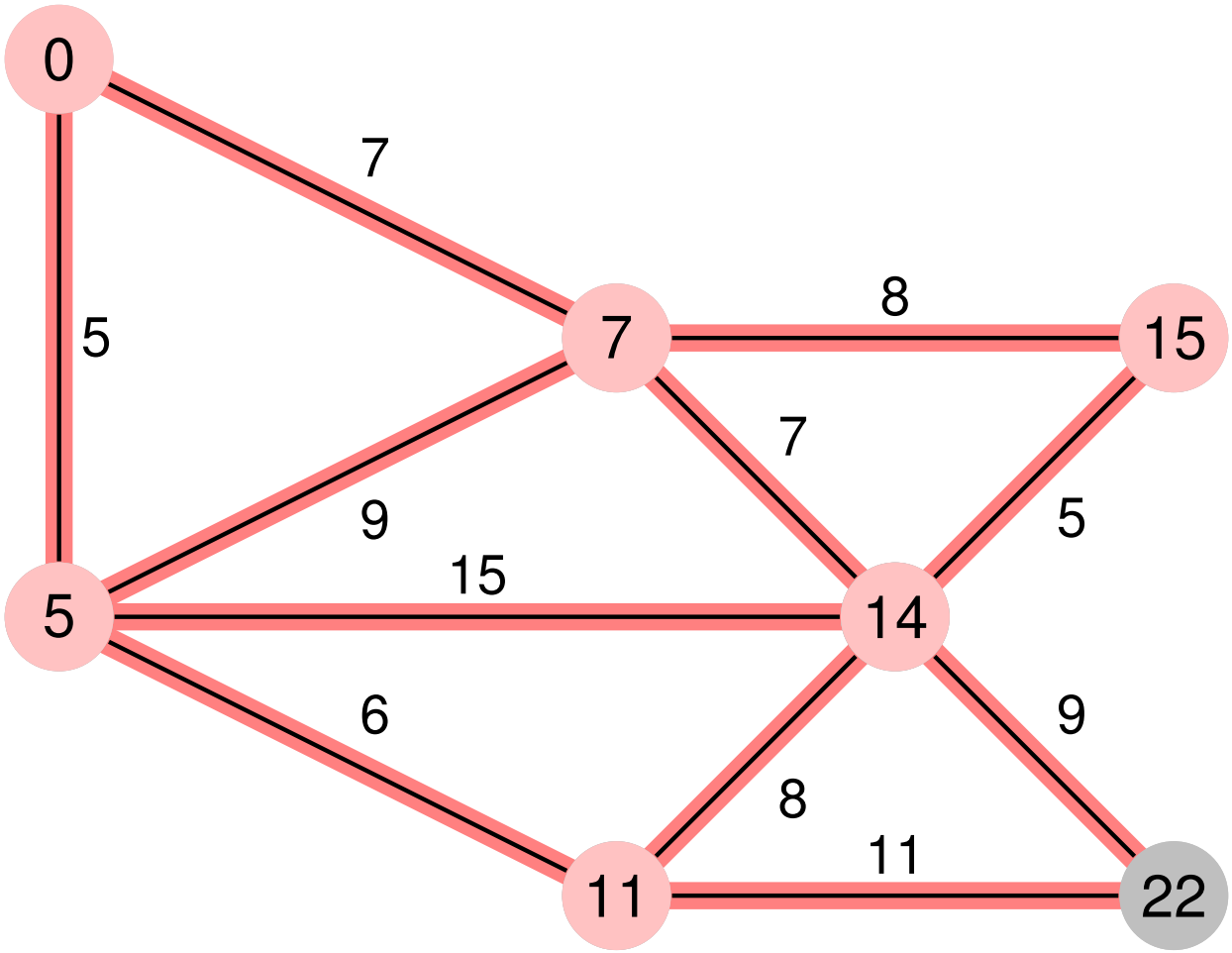
Ejemplo de ejecución de Dijkstra



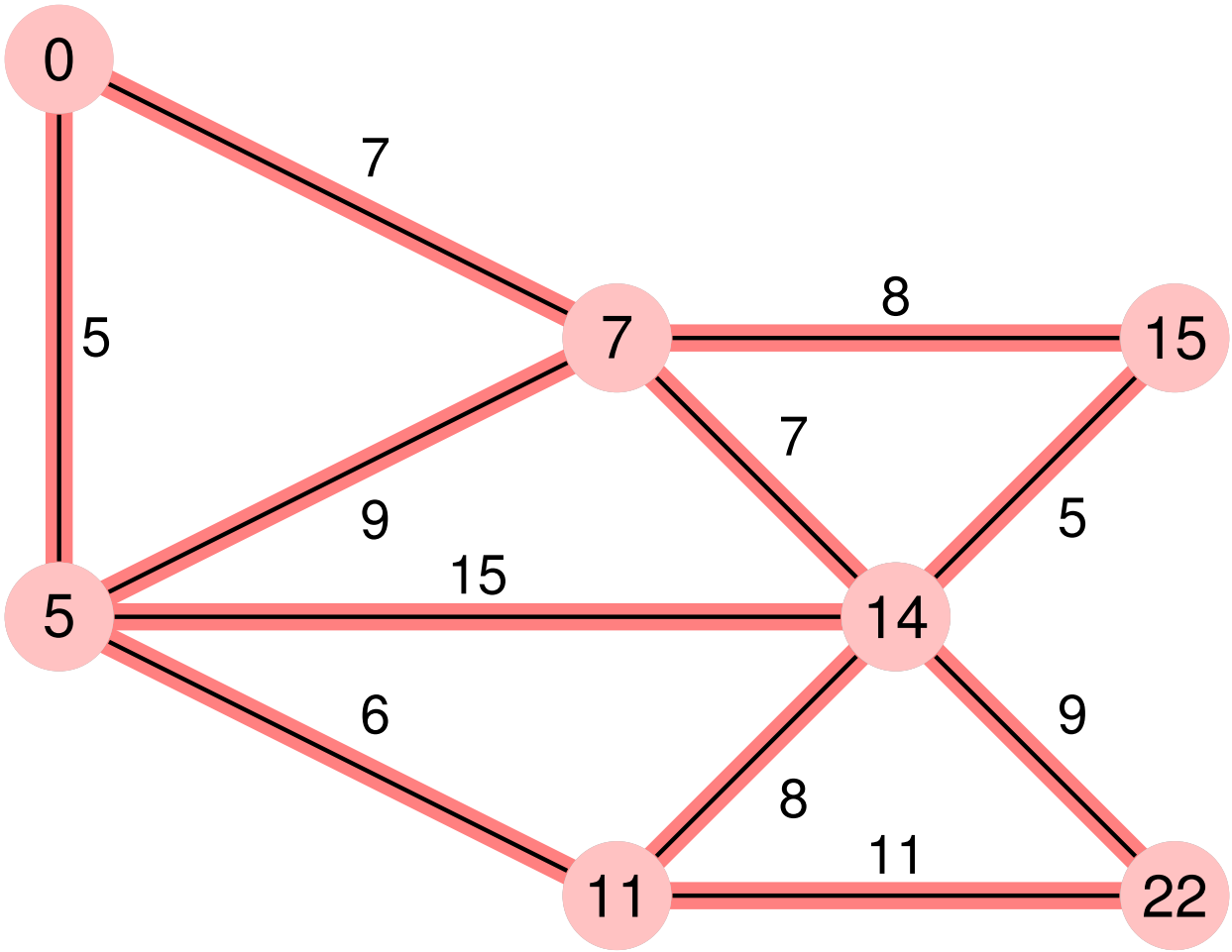
Ejemplo de ejecución de Dijkstra



Ejemplo de ejecución de Dijkstra



Ejemplo de ejecución de Dijkstra



Dijkstra - análisis

- El algoritmo de Dijkstra funciona porque en cada paso mantiene como invariante que la distancia que vamos calculando es la menor entre los caminos que tienen como nodos intermedios a los procesados.
- 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).

Dijkstra - código usando SET

```
1  void djstrak(long long node){
2    set<pair<long long,long long> > s;
3    s.insert(make_pair(0,node));
4    cost[node] = make_pair(0,node);
5    while(s.size()){
6      set<pair<long long,long long> >::iterator it = s.begin();
7      pair<long long,long long> current = *it;
8      s.erase(current);
9      if(visit[current.second]) continue;
10     visit[current.second]=true;
11     for(int i=0;i<graph[current.second].size();i++){
12       pair<long long,long long> p = graph[current.second][i];
13       if(!visit[p.second]){
14         long long newcost = cost[current.second].first + p.first;
15         if((cost[p.second].first > cost[current.second].first + p.first) or
               cost[p.second].first == -1){
16           cost[p.second] = make_pair(newcost,current.second);
17           s.insert(make_pair(newcost,p.second));
18         }
19       }
20     }
```

Distancias mínimas - 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).

Algoritmo de Floyd-Warshall

- 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, $d_{i,k} + d_{k,j}$).

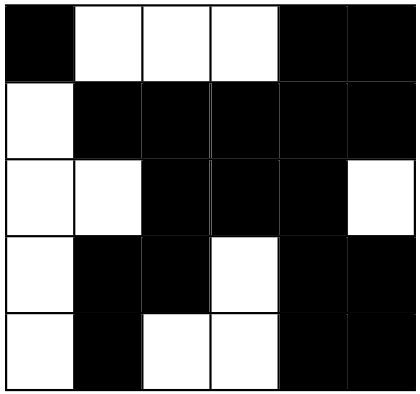
Floyd - código

```
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.

Distancias mínimas con BFS (en un tablero)

Problema: `http://www.spoj.com/problems/BITMAP/`
Tengo un tablero $n \times m$ con casillas blancas y negras.



Para cada casilla, quiero encontrar la mínima distancia a una casilla blanca. Donde la distancia es en el grafo donde los nodos son las casillas, y los vecinos son los casilleros arriba, abajo, izquierda y derecha de la casilla dada.

Solución

- (Intento de solución) Puedo correr un BFS desde cada nodo negro y ver cual es el nodo blanco con distancia mínima. PROBLEMA: Debería hacer n^2 BFS, y se me va en tiempo.
- Debo correr un BFS simultaneo desde todos los nodos blancos. Eso lo puedo hacer agregando un nodo imaginario que se conecta con todos los nodos blancos, y calculo distancias desde ahí.

Solución

- (Intento de solución) Puedo correr un BFS desde cada nodo negro y ver cual es el nodo blanco con distancia mínima. PROBLEMA: Debería hacer n^2 BFS, y se me va en tiempo.
- Debo correr un BFS simultaneo desde todos los nodos blancos. Eso lo puedo hacer agregando un nodo imaginario que se conecta con todos los nodos blancos, y calculo distancias desde ahí.

Código de la solución

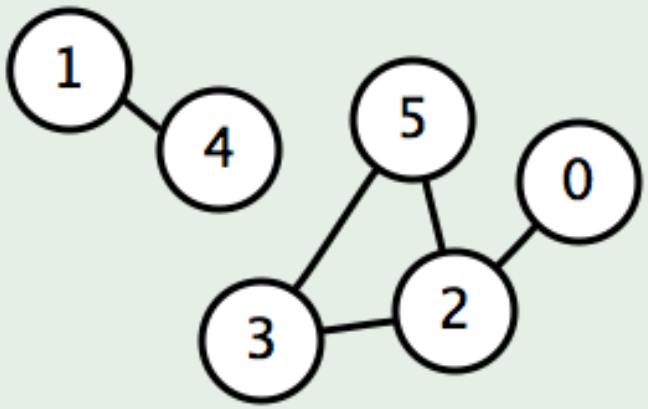
Ver aquí:`https://pastebin.com/LkWjyixi`

Contenidos

Componentes conexas

- Un grafo **no-dirigido** se dice conexo si hay camino entre cada par de nodos.
- Para cualquier grafo, podemos particionar los nodos en subconjuntos, tales que cada subconjunto forma un sub-grafo conexo y no hay ninguna arista entre dos nodos de subconjuntos distintos.
- Estos sub-conjuntos se llaman **componentes conexas** del grafo.

Ejemplo



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

Árboles

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

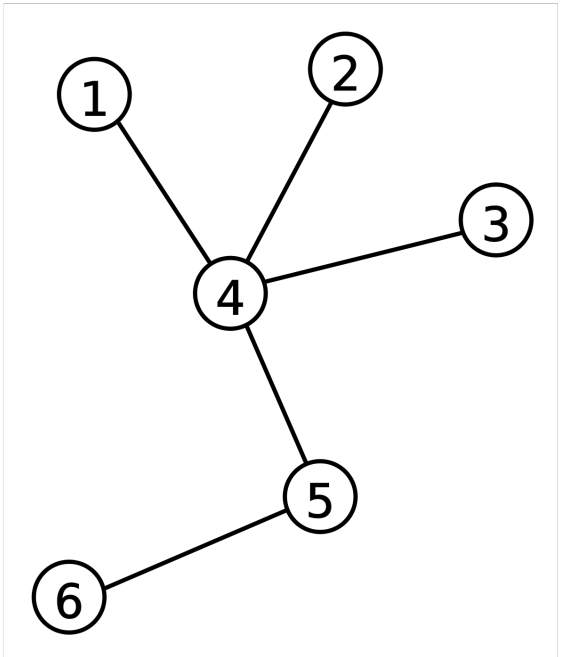


Figura: Árbol (ejemplo)

Contenidos

Union-find

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 (cont.)

- Union-find se puede implementar identificando cada componente con un "representante", que es un nodo de la componente.
- Podemos usar un arreglo que llamamos *uf*, donde para cada nodo guardamos un valor especial para marcar que es un representante, y el índice de otro nodo de la misma componente para los que no son representantes.
- Para realizar la operación $find(x)$: Si x es un representante devolvemos x, sino recursionamos para el nodo que guardamos en la posición x (o sea $uf[x]$).
- Para realizar la operación $union(x, y)$: Primero reemplazamos x e y por sus respectivos representantes. Si son iguales, no hay nada para hacer. Si son distintos, asignamos $uf[x] = y$.

Union-find (cont.)

La implementación anterior puede optimizarse de dos formas:

- Cuando realizamos el $find(x)$ podemos guardar en el arreglo el valor del representante para todos los nodos que recorrimos, para acelerar consultas posteriores.
- Cuando realizamos $union(x, y)$, nos conviene hacer $uf[x] = y$ si la componente de y es mas grande que la componente de x, y de lo contrario hacer $uf[y] = x$.

Con estas dos optimizaciones usadas a la vez, la complejidad de union-find es prácticamente $O(1)$.

- En la implementación que mostraremos, usamos un valor negativo en el arreglo para identificar los representantes. En ese caso, el valor absoluto de ese valor es la cantidad de nodos de la componente.

Union-find - código

```
1  int uf[MAXN];
2  void init_uf(int n){
3    fill(uf, uf+n, -1);
4  }
5  int find(int x){
6    if(uf[x] < 0)
7      return x;
8    return uf[x] = find(uf[x]);
9  }
10 void join(int x, int y){ // union es palabra reservada de C++ :(
11   x = find(x); y = find(y);
12   if(x == y)
13     return;
14   if(uf[x] > uf[y])
15     swap(x,y);
16   uf[x] += uf[y];
17   uf[y] = x;
18 }
```

Contenidos

Árbol generador mínimo

Definición

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.

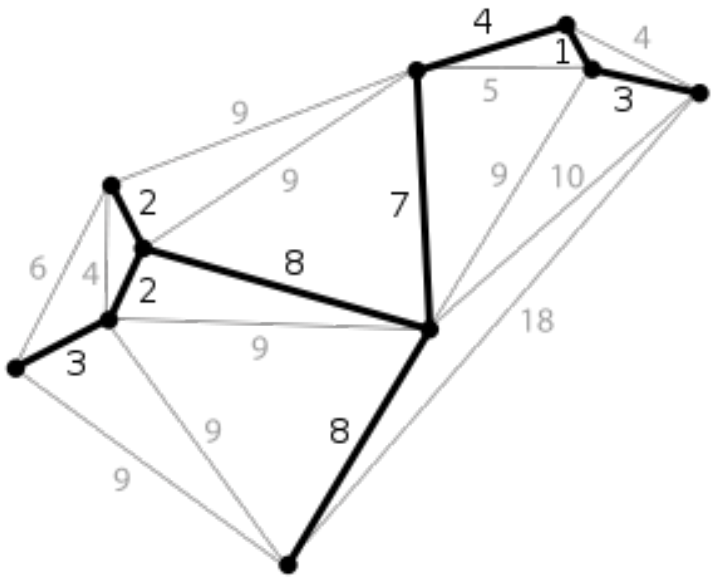


Figura: Árbol generador mínimo (ejemplo)

Algoritmo de Kruskal

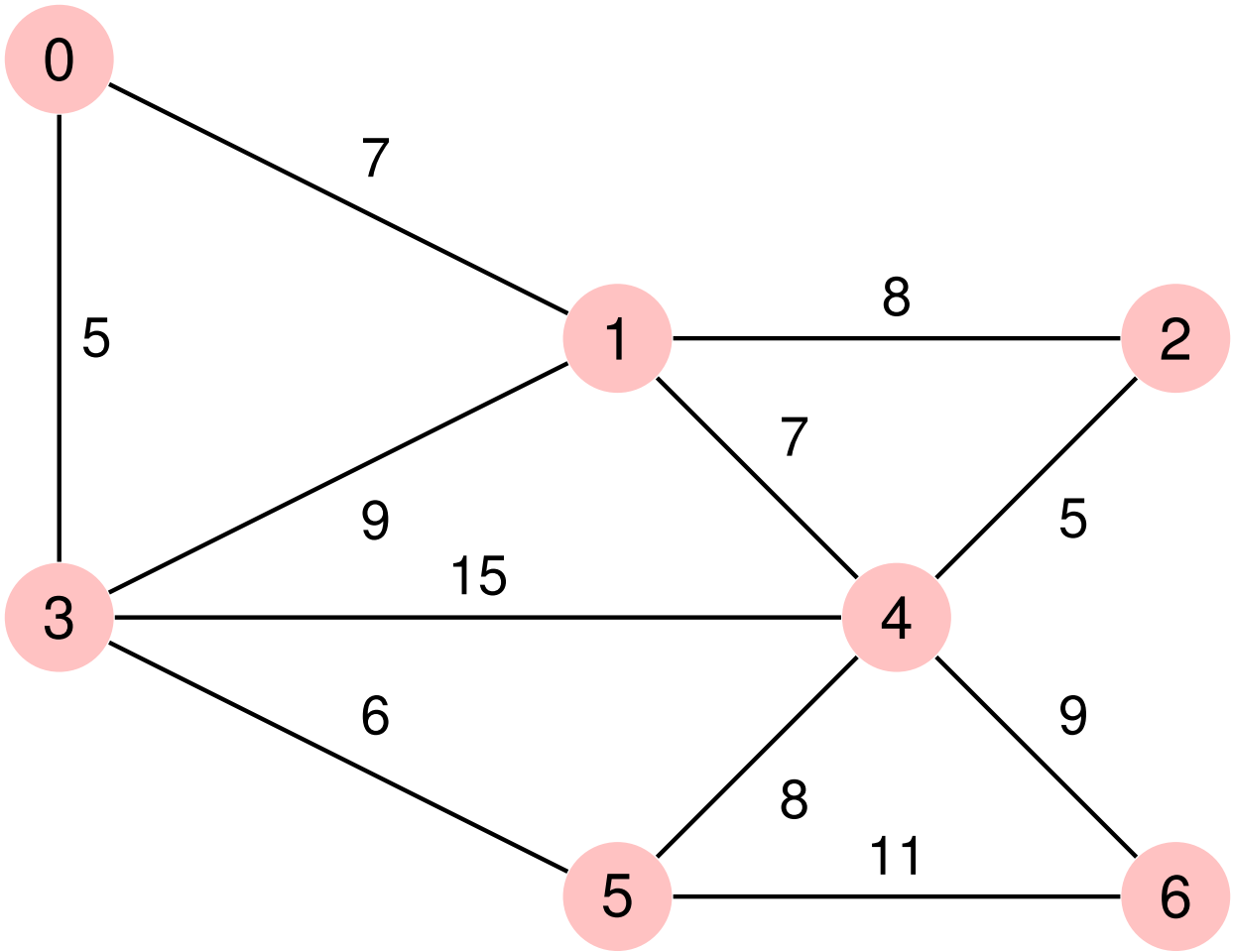
El algoritmo de Kruskal calcula el árbol generador mínimo como sigue:

- 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.

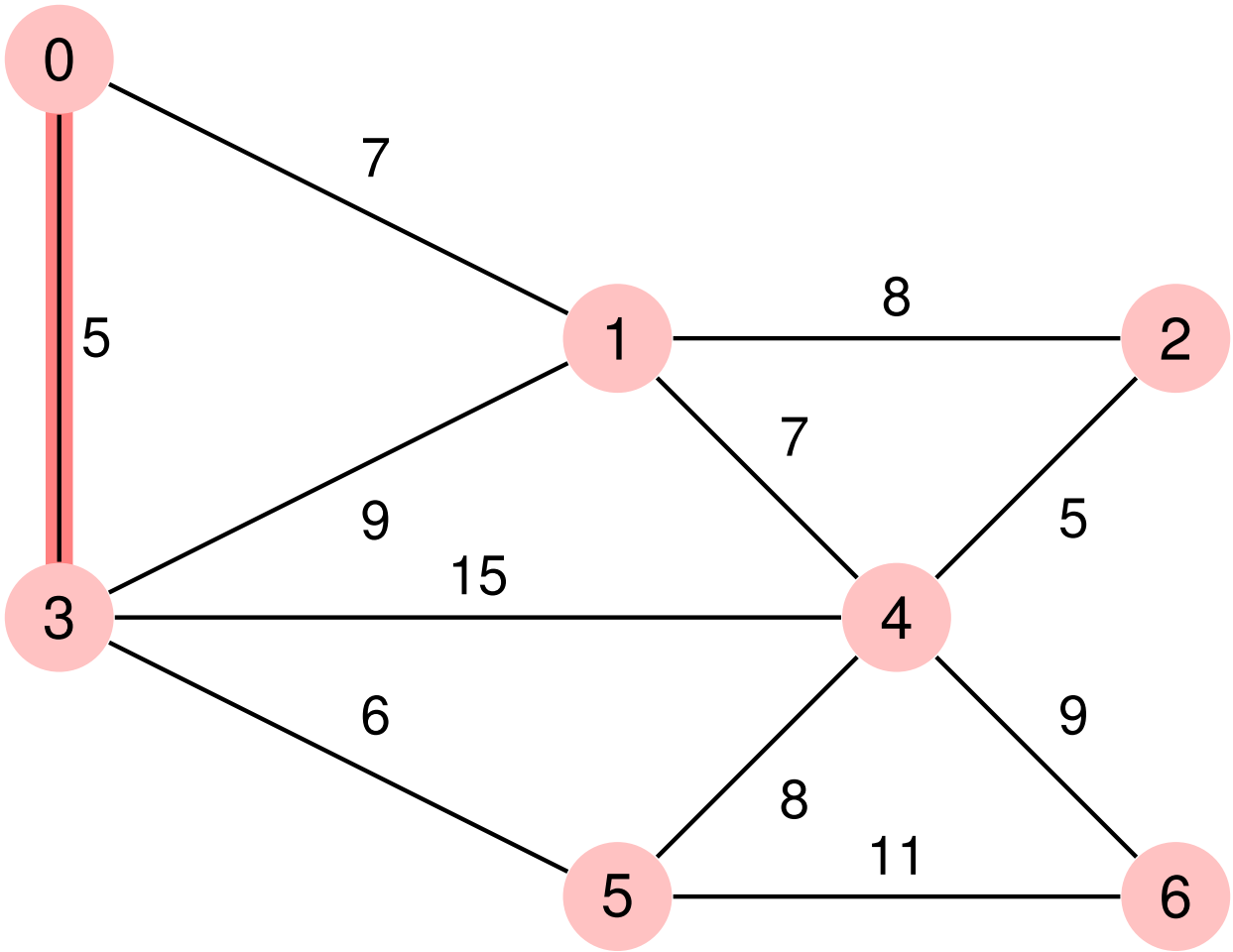
.

- 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.

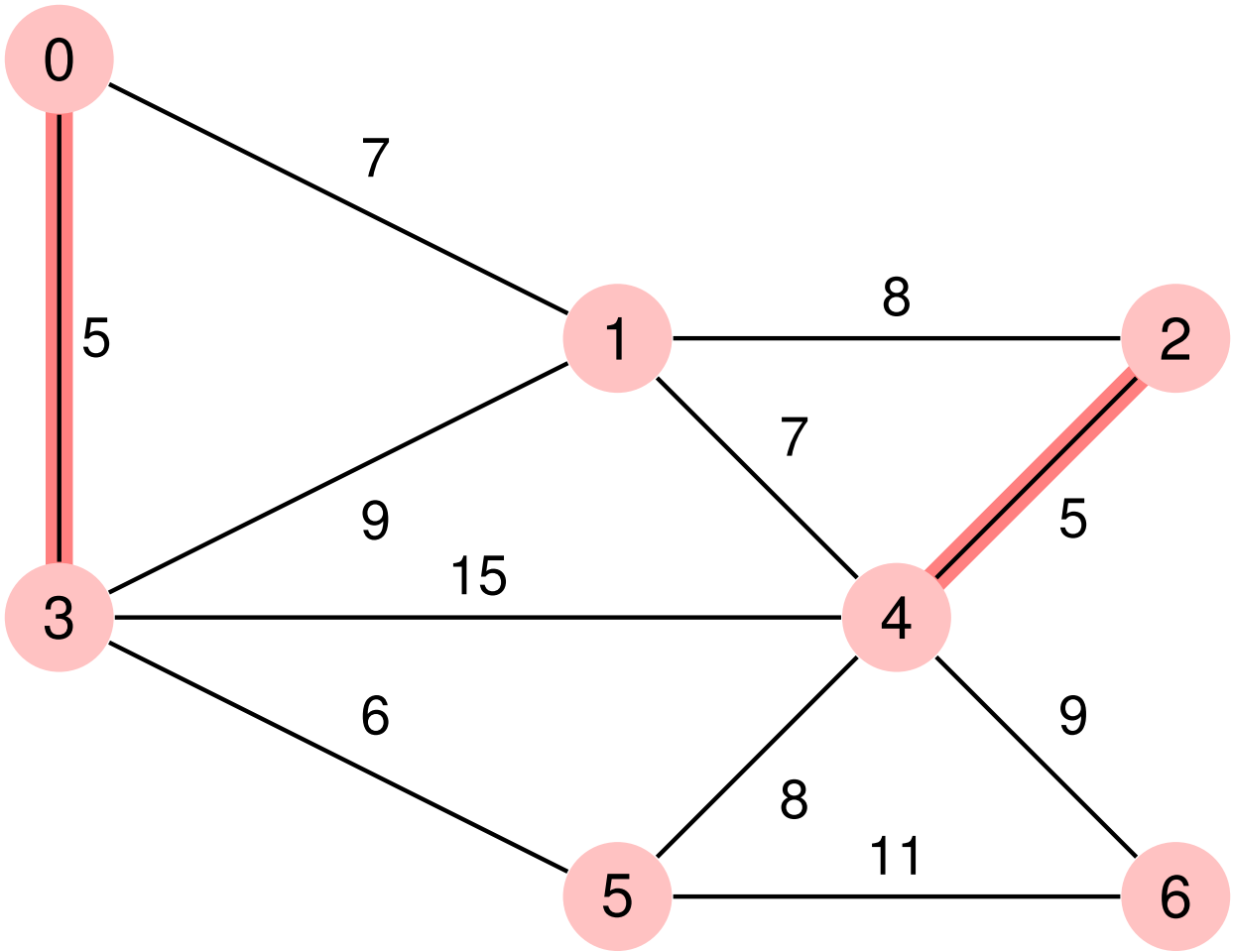
Ejemplo



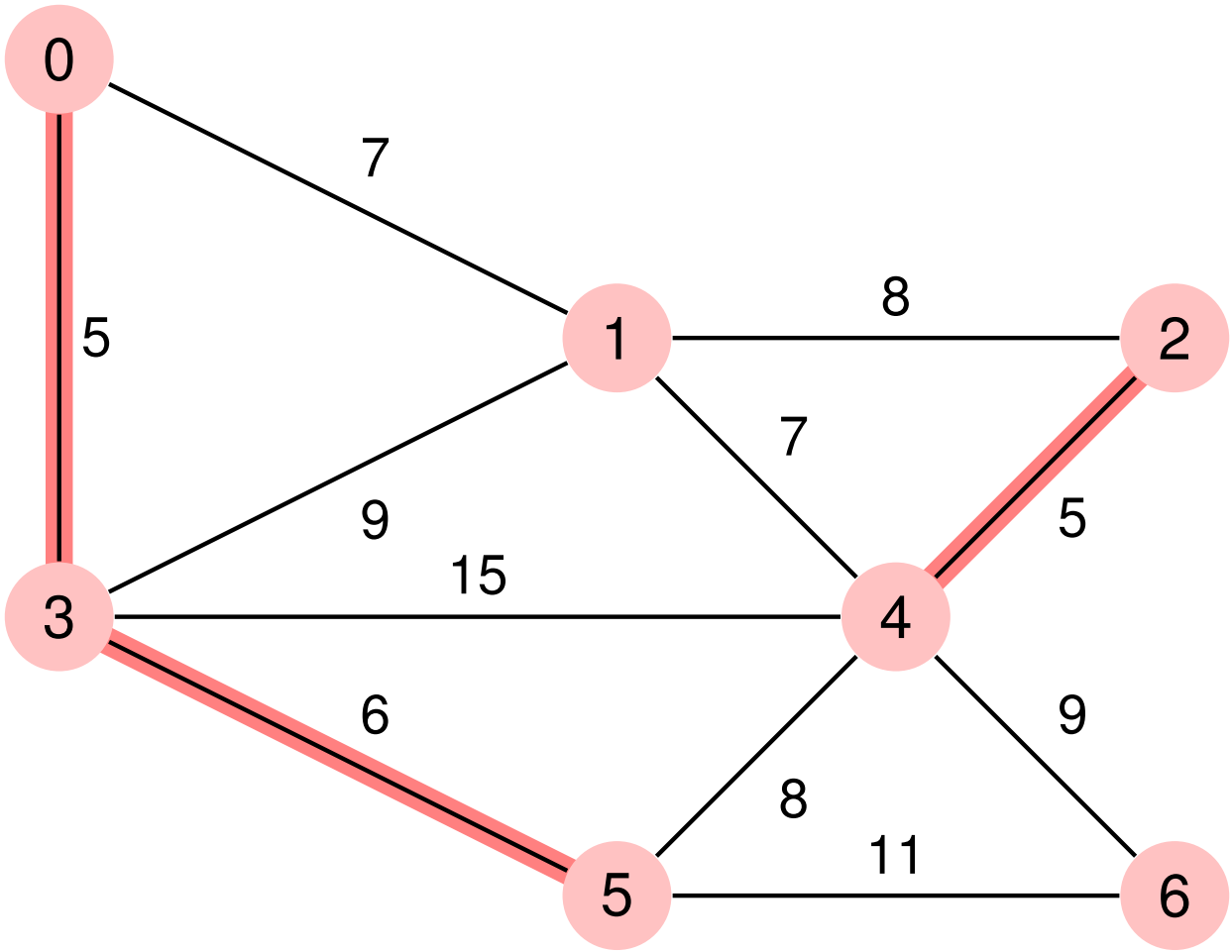
Ejemplo



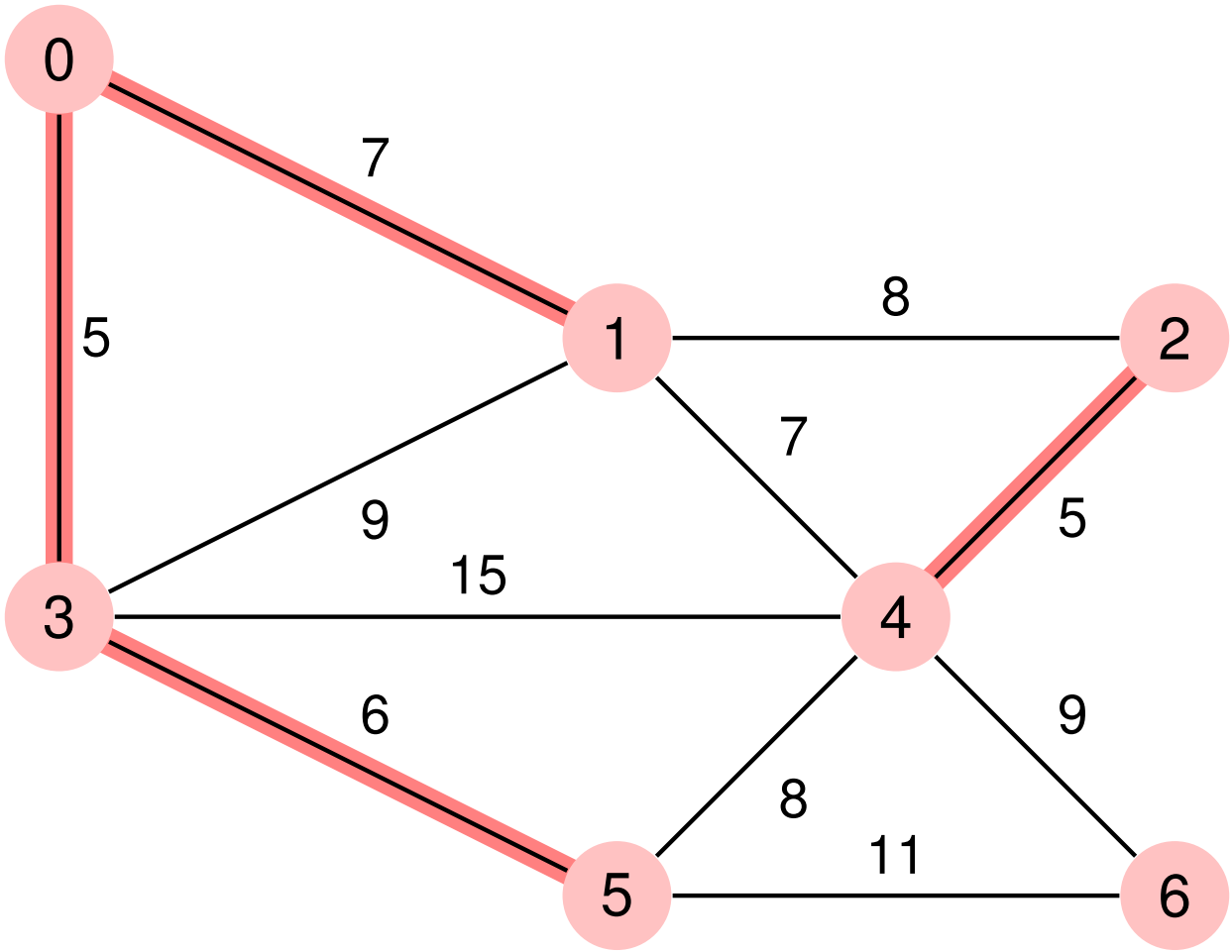
Ejemplo



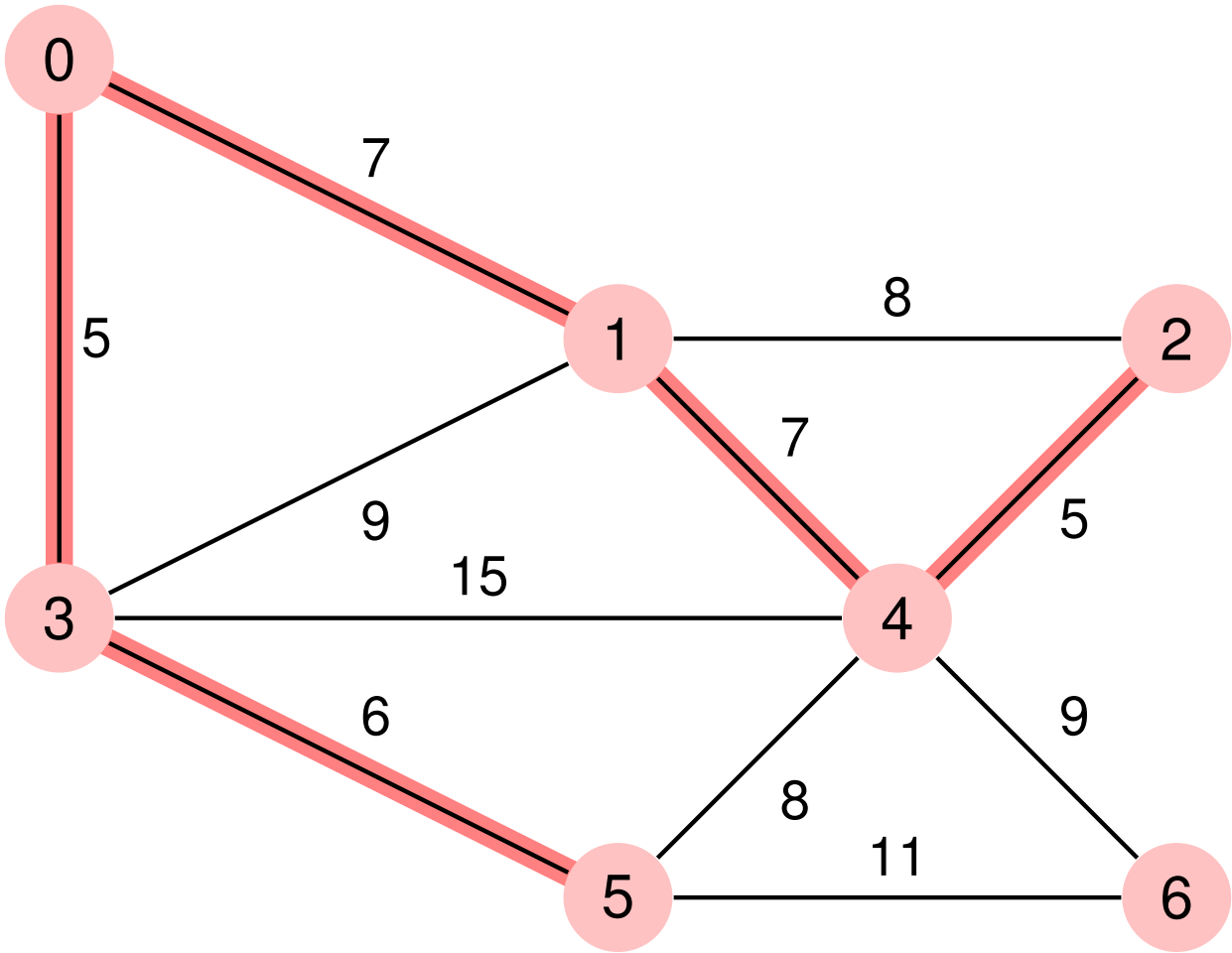
Ejemplo



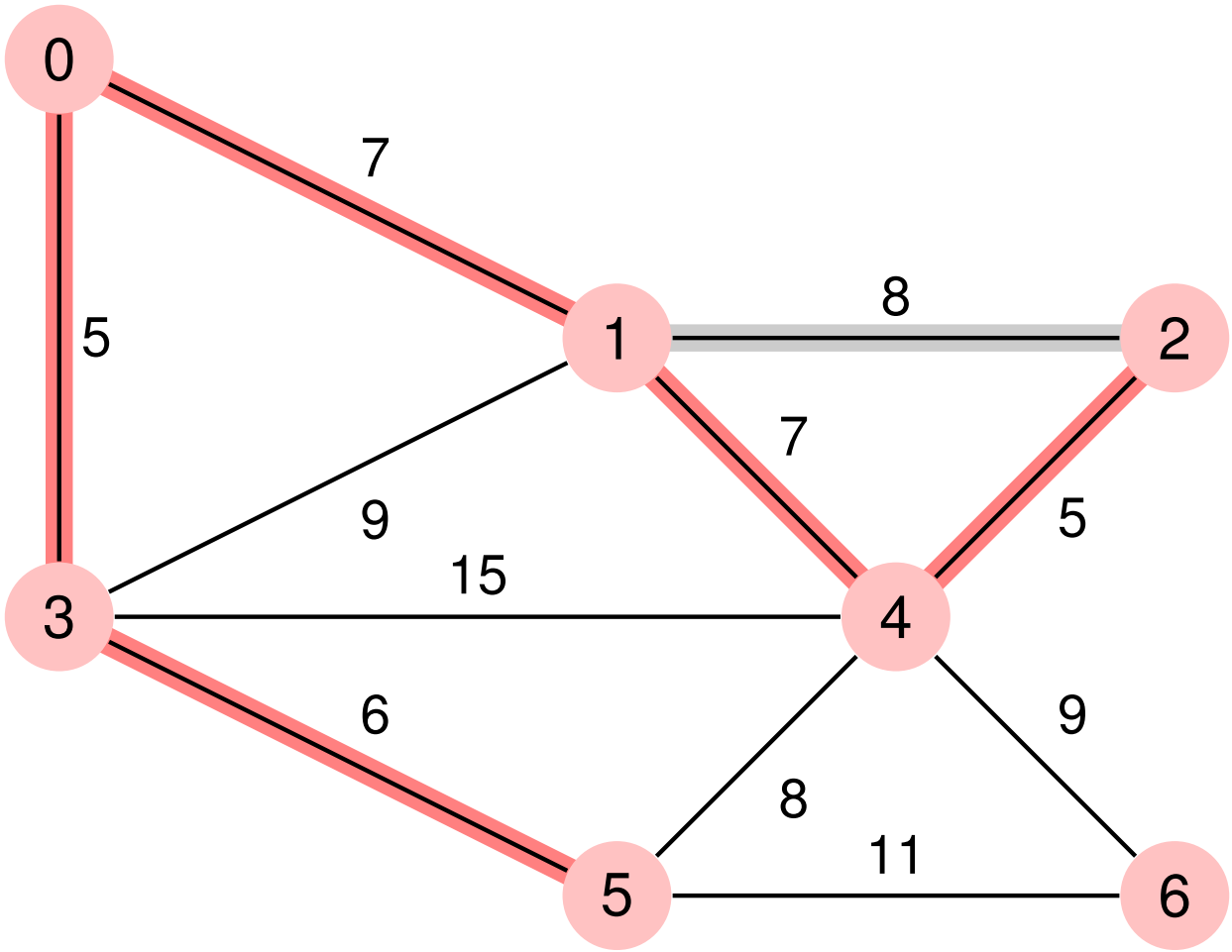
Ejemplo



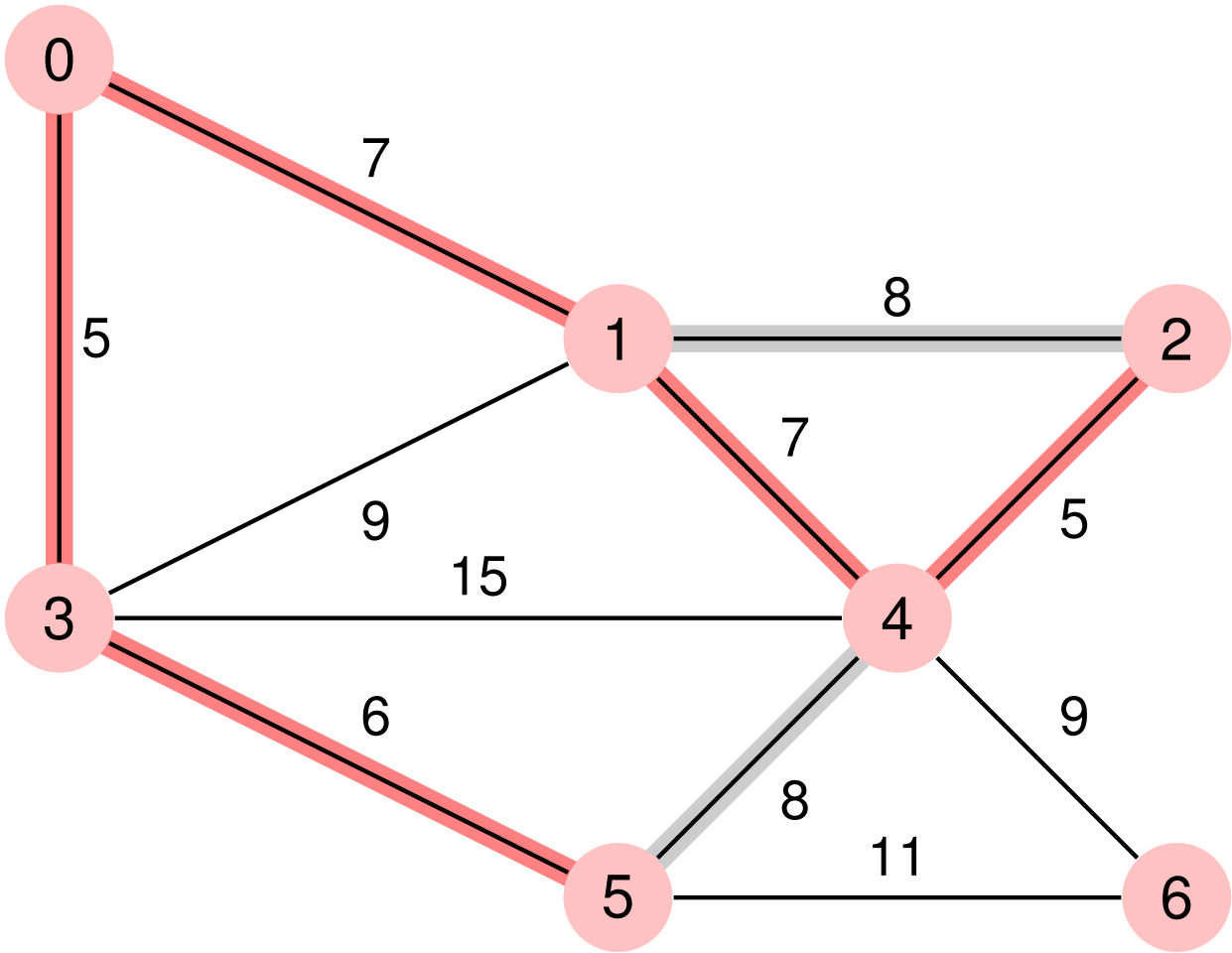
Ejemplo



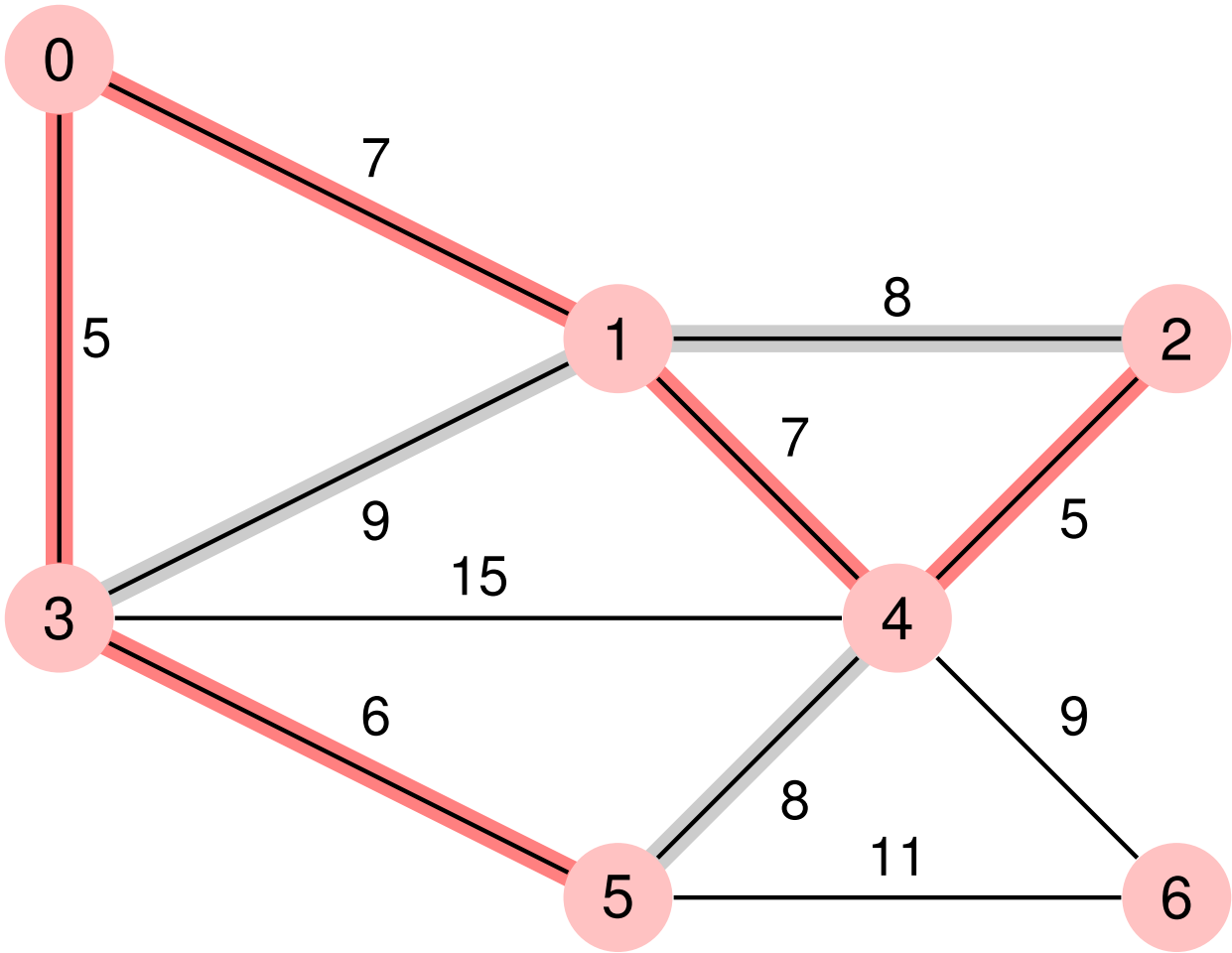
Ejemplo



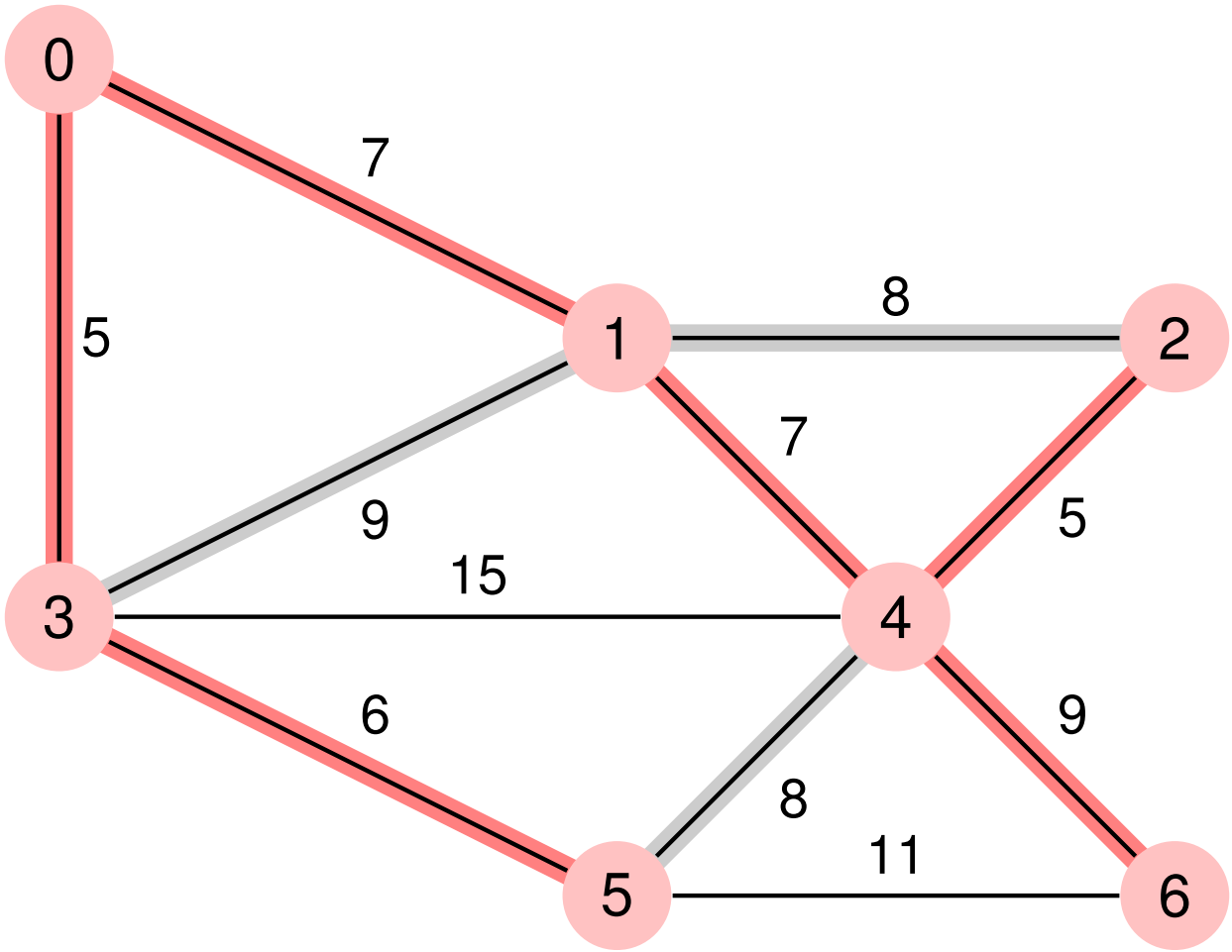
Ejemplo



Ejemplo



Ejemplo



Implementación de Kruskal

```
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)
10       continue;
11     res += c;
12     u++;
13     join(x,y);
14     if(u == n - 1) // completamos el arbol?
15       return res;
16   }
17   return -1; // si llegamos hasta aca entonces no es conexo
18 }
```

Diámetro en un árbol

Ver problema acá: `http://www.spoj.com/problems/PT07Z/`

Definición

Es la máxima distancia entre un par de nodos en un árbol. (Existe tal distancia porque no hay ciclos)

¿Cómo la calculo usando DFS o BFS?

Solución

- Hago un BFS desde un nodo arbitrario v para encontrar un nodo u que esté a una distancia máxima desde donde empecé.
- Hago un nuevo BFS desde el nodo u para encontrar un nodo w a distancia máxima del nodo u.
- La distancia $d(u, w)$ es el diámetro del árbol.

NOTA: Como entre cada par de nodos hay un único camino en un árbol, se puede usar DFS en vez de BFS para calcular distancias!! el código así es más fácil.

Solución

- Hago un BFS desde un nodo arbitrario v para encontrar un nodo u que esté a una distancia máxima desde donde empecé.
- Hago un nuevo BFS desde el nodo u para encontrar un nodo w a distancia máxima del nodo u.
- La distancia $d(u, w)$ es el diámetro del árbol.

NOTA: Como entre cada par de nodos hay un único camino en un árbol, se puede usar DFS en vez de BFS para calcular distancias!! el código así es más fácil.

Solución

- Hago un BFS desde un nodo arbitrario v para encontrar un nodo u que esté a una distancia máxima desde donde empecé.
- Hago un nuevo BFS desde el nodo u para encontrar un nodo w a distancia máxima del nodo u.
- La distancia $d(u, w)$ es el diámetro del árbol.

NOTA: Como entre cada par de nodos hay un único camino en un árbol, se puede usar DFS en vez de BFS para calcular distancias!! el código así es más fácil.

Solución

- Hago un BFS desde un nodo arbitrario v para encontrar un nodo u que esté a una distancia máxima desde donde empecé.
- Hago un nuevo BFS desde el nodo u para encontrar un nodo w a distancia máxima del nodo u.
- La distancia $d(u, w)$ es el diámetro del árbol.

NOTA: Como entre cada par de nodos hay un único camino en un árbol, se puede usar DFS en vez de BFS para calcular distancias!! el código así es más fácil.

Solución

- Hago un BFS desde un nodo arbitrario v para encontrar un nodo u que esté a una distancia máxima desde donde empecé.
- Hago un nuevo BFS desde el nodo u para encontrar un nodo w a distancia máxima del nodo u.
- La distancia $d(u, w)$ es el diámetro del árbol.

NOTA: Como entre cada par de nodos hay un único camino en un árbol, se puede usar DFS en vez de BFS para calcular distancias!! el código así es más fácil.