Grafos (principiantes) - Parte I

Juan Cruz Piñero

Facultad de Informática
Universidad Nacional del Comahue

Training Camp 2017

Contenidos

Contenidos

Qué es un grafo?

"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." - Wikipedia

Qué es un grafo?

"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." - Wikipedia
"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."
- Real Academia Española

Qué es un grafo?

"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." - Wikipedia

"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."
- Real Academia Española

"Un grafo es un conjunto de puntos y líneas que unen pares de esos puntitos" - La posta

Definición formal de grafo

Grafo

Se define como un conjunto V cuyos elementos se denominan vértices o nodos, y un conjunto $E \subseteq V \times V$ cuyos elementos se llaman ejes o aristas. Un grafo puede ser dirigido, es decir, $(a, b) \in E$ no es lo mismo que $(b, a) \in E$ o no dirigido, cuando $(a, b) = (b, a)$.

Definición formal de grafo

Grafo

Se define como un conjunto V cuyos elementos se denominan vértices o nodos, y un conjunto $E \subseteq V \times V$ cuyos elementos se llaman ejes o aristas. Un grafo puede ser dirigido, es decir, $(a, b) \in E$ no es lo mismo que $(b, a) \in E$ o no dirigido, cuando $(a, b) = (b, a)$.

Ejemplo

Mediante un grafo podemos representar, por ejemplo, una ciudad. Las esquinas serían los vértices (elementos de V) y las conexiones por medio de una calle entre dos esquinas serían los ejes (elementos de E). Si las calles son mano y contramano el grafo es dirigido, si en cambio son doble mano, el grafo es no dirigido.

Importancia de los grafos

- Un grafo nos permite representar relaciones de cualquier tipo entre elementos de un conjunto: Dos elementos (vértices) están relacionados si tienen un arista o arco que los relaciona. Si son aristas, la relación entre estos elementos es recíproca.

Importancia de los grafos

- Un grafo nos permite representar relaciones de cualquier tipo entre elementos de un conjunto: Dos elementos (vértices) están relacionados si tienen un arista o arco que los relaciona. Si son aristas, la relación entre estos elementos es recíproca.
- Esto hace que las técnicas generales que veremos en grafos sean aplicables en casos en que el grafo no esté explicitado. Podemos ver las transiciones de estados de una solución recursiva (posiblemente DP) como arcos de un grafo implícito.

Importancia de los grafos

- Un grafo nos permite representar relaciones de cualquier tipo entre elementos de un conjunto: Dos elementos (vértices) están relacionados si tienen un arista o arco que los relaciona. Si son aristas, la relación entre estos elementos es recíproca.
- Esto hace que las técnicas generales que veremos en grafos sean aplicables en casos en que el grafo no esté explicitado. Podemos ver las transiciones de estados de una solución recursiva (posiblemente DP) como arcos de un grafo implícito.
- Comenzaremos trabajando con grafos no dirigidos y daremos algunas definiciones que tienen sentido en grafos no dirigidos, pero que luego podremos adaptar a grafos dirigidos.

Contenidos

Caminos

Adyacencia

Decimos que dos vértices v_1 y v_2 son adyacentes si $(v_1, v_2) \in E$. En este caso decimos que hay una arista entre v_1 y v_2. Es común llamar *"vecinos"* de v_1 a los arcos *ädyacentes"*

Camino

Un camino entre v y w es una lista de $n + 1$ vértices $v = v_0, v_1, \ldots, v_n = w$ tales que para $0 \leq i < n$ los vértices v_i y $v_i + 1$ son adyacentes.

Longitud de un camino

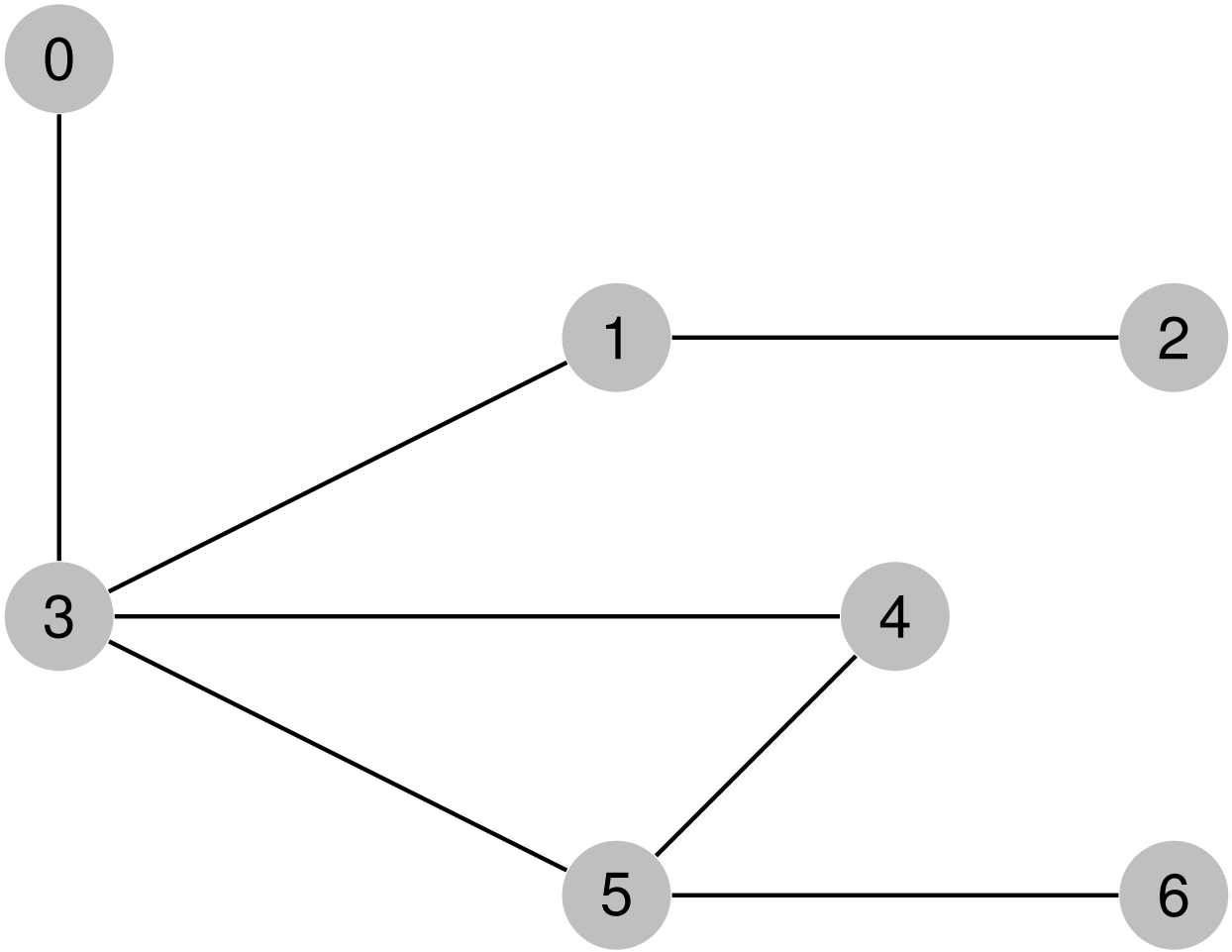
Camino Simple

Un camino entre v y w, se dice un camino simple, si se cumple que ninguno de los vértices que visita, es visitado más de una vez.

Longitud de un camino

Diremos que un camino entre v y w es de longitud n, si ese camino es una lista de vértices de longitud n.

Ejemplo



Distancia

Distancia entre dos vértices

La distancia entre dos vértices v y w se define como el menor número n tal que existe un camino entre v y w de largo n. Si no existe ningún camino entre v y w decimos que la distancia entre v y w es ∞

Contenidos

Camino mínimo

- Si la distancia entre v y w es n, entonces un camino entre v y w de largo n se llama camino mínimo.

Camino mínimo

- Si la distancia entre v y w es n, entonces un camino entre v y w de largo n se llama camino mínimo.
- Un problema muy frecuente es tener que encontrar la distancia entre dos vértices, este problema se resuelve encontrando un camino mínimo entre los vértices.

Camino mínimo

- Si la distancia entre v y w es n, entonces un camino entre v y w de largo n se llama camino mínimo.
- Un problema muy frecuente es tener que encontrar la distancia entre dos vértices, este problema se resuelve encontrando un camino mínimo entre los vértices.
- Este problema es uno de los problemas más comunes de la teoría de grafos y se puede resolver de varias maneras, una de ellas es el algoritmo llamado BFS (Breadth First Search).

BFS

Breadth First Search

El BFS es un algoritmo que calcula las distancias de un nodo de un grafo a todos los demás. Para esto empieza en el nodo desde el cual queremos calcular la distancia a todos los demás y se mueve a todos sus vecinos, una vez que hizo esto, se mueve a los vecinos de los vecinos, y así hasta que recorrió todos los nodos del grafo a los que existe un camino desde el nodo inicial.

BFS

Breadth First Search

El BFS es un algoritmo que calcula las distancias de un nodo de un grafo a todos los demás. Para esto empieza en el nodo desde el cual queremos calcular la distancia a todos los demás y se mueve a todos sus vecinos, una vez que hizo esto, se mueve a los vecinos de los vecinos, y así hasta que recorrió todos los nodos del grafo a los que existe un camino desde el nodo inicial.

Representación del grafo

A la hora de implementar un algoritmo sobre un grafo es importante saber cómo vamos a representar el grafo. Hay más de una forma de representar un grafo, nosotros veremos dos de ellas.

Formas de representar un grafo

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 nuestra implementación del BFS

Formas de representar un grafo

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 nuestra implementación del BFS

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 y 0 (o false) sino.

Formas de representar un grafo

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 nuestra implementación del BFS

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 y 0 (o false) sino.

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

Detalles de implementación del BFS

- Las distancias del nodo inicial a los demás nodos las inicializaremos en un virtual infinito (puede ser, por ejemplo, la cantidad de nodos del grafo, ya que es imposible que la distancia entre dos nodos del grafo sea mayor o igual a ese número.)

Detalles de implementación del BFS

- Las distancias del nodo inicial a los demás nodos las inicializaremos en un virtual infinito (puede ser, por ejemplo, la cantidad de nodos del grafo, ya que es imposible que la distancia entre dos nodos del grafo sea mayor o igual a ese número.)
- Usaremos una cola para ir agregando los nodos por visitar. Como agregamos primero todos los vecinos del nodo inicial, los primeros nodos en entrar a la cola son los de distancia 1, luego agregamos los vecinos de esos nodos, que son los de distancia 2, y así vamos recorriendo el grafo en orden de distancia al vértice inicial.

Detalles de implementación del BFS

- Las distancias del nodo inicial a los demás nodos las inicializaremos en un virtual infinito (puede ser, por ejemplo, la cantidad de nodos del grafo, ya que es imposible que la distancia entre dos nodos del grafo sea mayor o igual a ese número.)
- Usaremos una cola para ir agregando los nodos por visitar. Como agregamos primero todos los vecinos del nodo inicial, los primeros nodos en entrar a la cola son los de distancia 1, luego agregamos los vecinos de esos nodos, que son los de distancia 2, y así vamos recorriendo el grafo en orden de distancia al vértice inicial.
- Cuando visitamos un nodo, sabemos cuáles de sus vecinos agregar a la cola. Tenemos que visitar los vecinos que todavía no han sido visitados.

Implementación del BFS

```
1  vector<int> BFS(vector<vector<int> > &lista, int nodoInicial){
2      int n = lista.size(),t;
3      queue<int> cola;
4      vector<int> distancias(n,n);
5      cola.push(nodoInicial);
6      distancias[nodoInicial] = 0;
7      while(!cola.empty()){
8          t = cola.front();
9          cola.pop();
10         for(int i=0;i<lista[t].size();i++){
11             if(distancias[lista[t][i]]==n){
12                 distancias[lista[t][i]] = distancias[t]+1;
13                 cola.push(lista[t][i]);
14             }
15         }
16     }
17     return distancias;
18 }
```

Ejemplo - BFS

http://www.spoj.com/problems/BITMAP/

There is given a rectangular bitmap of size n*m. Each pixel of the bitmap is either white or black, but at least one is white. The pixel in i-th line and j-th column is called the pixel (i,j).

The distance between two pixels p1=(i1,j1) and p2=(i2,j2) is defined as: $d(p1, p2) = |i1 - i2| + |j1 - j2|$.

Task Write a program which: - reads the description of the bitmap from the standard input, - for each pixel, computes the distance to the nearest white pixel, - writes the results to the standard output.

(1 <= n, m <= 200)

Formas de recorrer un grafo

- Existen varias maneras de recorrer un grafo, cada una puede ser útil según el problema. Una forma de recorrer un grafo es el BFS, que recorre los nodos en orden de distancia a un nodo.

Formas de recorrer un grafo

- Existen varias maneras de recorrer un grafo, cada una puede ser útil según el problema. Una forma de recorrer un grafo es el BFS, que recorre los nodos en orden de distancia a un nodo.
- Otra forma de recorrer un grafo es un DFS (Depth First Search), que recorre el grafo en profundidad, es decir, empieza por el nodo inicial y en cada paso visita un nodo no visitado del nodo donde está parado, si no hay nodos por visitar vuelve para atrás.

Formas de recorrer un grafo

- Existen varias maneras de recorrer un grafo, cada una puede ser útil según el problema. Una forma de recorrer un grafo es el BFS, que recorre los nodos en orden de distancia a un nodo.
- Otra forma de recorrer un grafo es un DFS (Depth First Search), que recorre el grafo en profundidad, es decir, empieza por el nodo inicial y en cada paso visita un nodo no visitado del nodo donde está parado, si no hay nodos por visitar vuelve para atrás.
- Un problema que se puede resolver con un DFS es decidir si un grafo es un árbol.

Árboles

Definición

Un árbol es un grafo conexo acíclico. Un grafo es conexo si para todo par de vértices hay un camino que los une, y es acíclico si no contiene ciclos.

Árboles

Definición

Un árbol es un grafo conexo acíclico. Un grafo es conexo si para todo par de vértices hay un camino que los une, y es acíclico si no contiene ciclos.

- Un DFS empieza en un nodo cualquiera de un grafo, y se expande en cada paso a un vecino del nodo actual, si ya se expandio a todos los vecinos vuelve para atrás.

Árboles

Definición

Un árbol es un grafo conexo acíclico. Un grafo es conexo si para todo par de vértices hay un camino que los une, y es acíclico si no contiene ciclos.

- Un DFS empieza en un nodo cualquiera de un grafo, y se expande en cada paso a un vecino del nodo actual, si ya se expandio a todos los vecinos vuelve para atrás.
- Si el DFS se quiere expandir a un nodo ya visitado desde otro nodo, esto quiere decir que hay un ciclo.

Ejemplo DFS 1

- En el ejemplo anterior vimos cómo recorre el grafo un DFS.

Ejemplo DFS 1

- En el ejemplo anterior vimos cómo recorre el grafo un DFS.
- Cuando llega a un nodo que ya fue visitado se da cuenta de que pudo acceder a ese nodo por dos caminos, eso quiere decir que si recorremos uno de los caminos en un sentido y el otro camino en sentido opuesto formamos un ciclo.

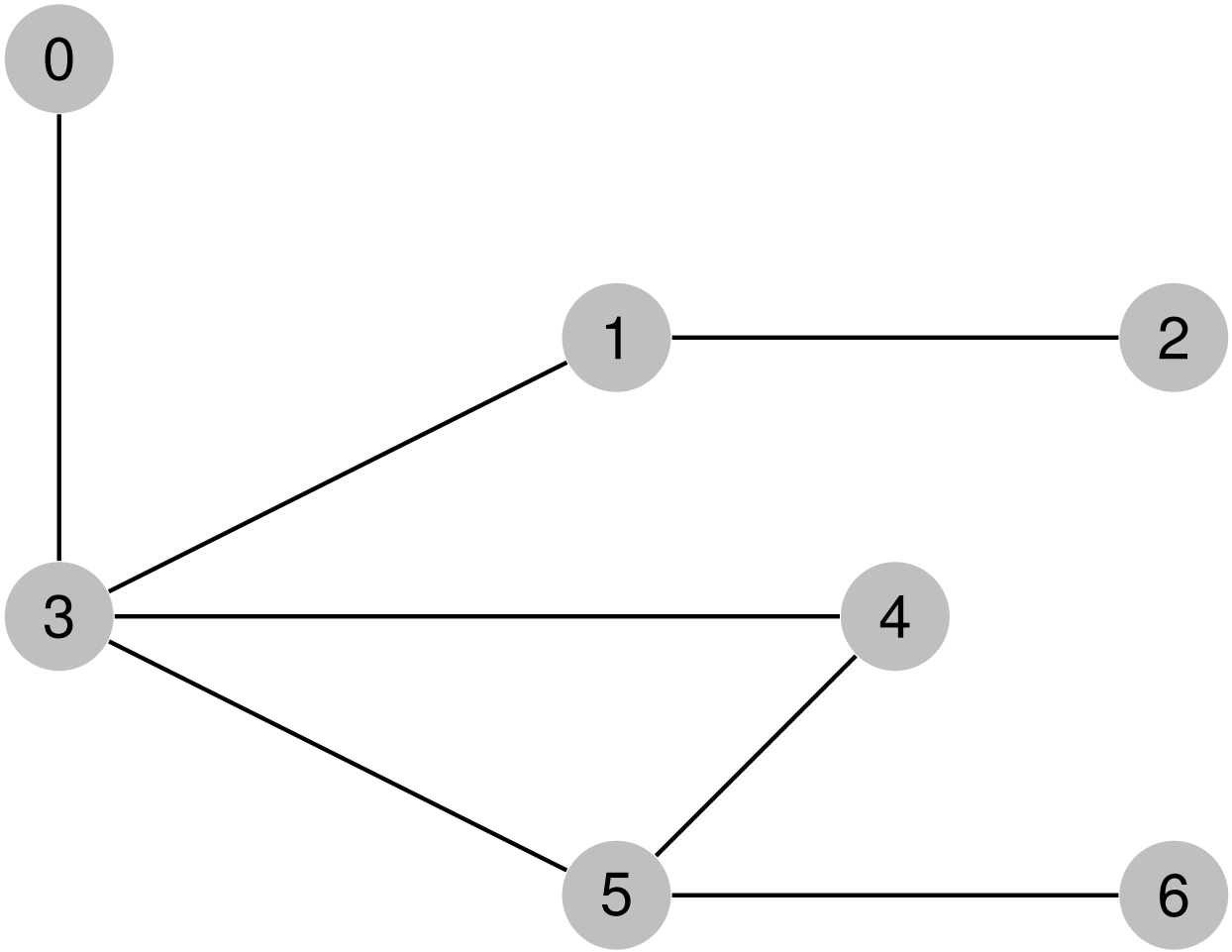
Ejemplo DFS 1

- En el ejemplo anterior vimos cómo recorre el grafo un DFS.
- Cuando llega a un nodo que ya fue visitado se da cuenta de que pudo acceder a ese nodo por dos caminos, eso quiere decir que si recorremos uno de los caminos en un sentido y el otro camino en sentido opuesto formamos un ciclo.
- Así como el BFS se implementa con una cola, el DFS se implementa con una pila.

Implementación del DFS

```
1  bool esArbol(vector<vector<int> > &lista, int t, vector<bool> &toc, int
       padre)
2  {
3      toc[t] = true;
4      for(int i=0;i<lista[t].size();i++)
5      {
6          if((toc[lista[t][i]]==true&&lista[t][i]!=padre))
7              return false;
8          if(toc[lista[t][i]]==false)
9          if(esArbol(lista,lista[t][i],toc,t)==false))
10             return false;
11     }
12     return true;
13 }
```

Ejemplo



Análisis del DFS

- En ningún momento usamos una pila explícita, pero la pila está implícita en la recursión.

Análisis del DFS

- En ningún momento usamos una pila explícita, pero la pila está implícita en la recursión.
- Guardamos el padre del nodo, es decir, el nodo desde el cuál fuimos a parar al nodo actual, para no confundir un ciclo con ir y volver por la misma arista.

Análisis del DFS

- En ningún momento usamos una pila explícita, pero la pila está implícita en la recursión.
- Guardamos el padre del nodo, es decir, el nodo desde el cuál fuimos a parar al nodo actual, para no confundir un ciclo con ir y volver por la misma arista.
- Si el nodo ya lo visitamos y no es el nodo desde el cual venimos quiere decir que desde algún nodo llegamos por dos caminos, o sea que hay un ciclo.

Análisis del DFS

- En ningún momento usamos una pila explícita, pero la pila está implícita en la recursión.
- Guardamos el padre del nodo, es decir, el nodo desde el cuál fuimos a parar al nodo actual, para no confundir un ciclo con ir y volver por la misma arista.
- Si el nodo ya lo visitamos y no es el nodo desde el cual venimos quiere decir que desde algún nodo llegamos por dos caminos, o sea que hay un ciclo.
- Si el nodo no lo visitamos, pero desde uno de sus vecinos podemos llegar a un ciclo, entonces es porque hay un ciclo en el grafo y por lo tanto no es un árbol.

Análisis del DFS

- En ningún momento usamos una pila explícita, pero la pila está implícita en la recursión.
- Guardamos el padre del nodo, es decir, el nodo desde el cuál fuimos a parar al nodo actual, para no confundir un ciclo con ir y volver por la misma arista.
- Si el nodo ya lo visitamos y no es el nodo desde el cual venimos quiere decir que desde algún nodo llegamos por dos caminos, o sea que hay un ciclo.
- Si el nodo no lo visitamos, pero desde uno de sus vecinos podemos llegar a un ciclo, entonces es porque hay un ciclo en el grafo y por lo tanto no es un árbol.
- Faltan tener en cuenta algunas consideraciones. ¿Se dan cuenta cuáles?

Análisis del DFS

- *t* es el nodo en el que estamos parados, *toc* guarda los nodos que ya tocamos, y *padre* es el nodo desde el que venimos.

Análisis del DFS

- *t* es el nodo en el que estamos parados, *toc* guarda los nodos que ya tocamos, y *padre* es el nodo desde el que venimos.
- *toc* empieza inicializado en false y el tamaño de *toc* es el mismo que el de *lista*.

Análisis del DFS

- *t* es el nodo en el que estamos parados, *toc* guarda los nodos que ya tocamos, y *padre* es el nodo desde el que venimos.
- *toc* empieza inicializado en false y el tamaño de *toc* es el mismo que el de *lista*.
- Estamos asumiendo que el grafo es conexo, ya que si no lo es la función *esArbol* puede devolver true sin ser un árbol

Análisis del DFS

- *t* es el nodo en el que estamos parados, *toc* guarda los nodos que ya tocamos, y *padre* es el nodo desde el que venimos.
- *toc* empieza inicializado en false y el tamaño de *toc* es el mismo que el de *lista*.
- Estamos asumiendo que el grafo es conexo, ya que si no lo es la función *esArbol* puede devolver true sin ser un árbol
- La forma de chequear si el grafo es conexo es chequear que hayamos tocado todos los nodos, es decir, que todas las posiciones de *toc* terminen en true.

Ejemplo DFS 2

http://www.spoj.com/problems/PT07Z/ - Longest path in a tree
You are given an unweighted, undirected tree. Write a program to output the length of the longest path (from one node to another) in that tree. The length of a path in this case is number of edges we traverse from source to destination.

Input
The first line of the input file contains one integer N — number of nodes in the tree (0 < N <= 10000). Next N-1 lines contain N-1 edges of that tree — Each line contains a pair (u, v) means there is an edge between node u and node v (1 <= u, v <= N).

Output
Print the length of the longest path on one line.

Complejidades

- Para saber si un algoritmo es bueno en cuanto a su tiempo de ejecución es muy útil conocer su complejidad temporal, es decir, una función evaluada en el tamaño del problema que nos de una cota de la cantidad de operaciones del algoritmo.

Complejidades

- Para saber si un algoritmo es bueno en cuanto a su tiempo de ejecución es muy útil conocer su complejidad temporal, es decir, una función evaluada en el tamaño del problema que nos de una cota de la cantidad de operaciones del algoritmo.
- Por ejemplo, si el problema tiene un tamaño de entrada n la complejidad podría ser $O(< n^2)$, esto quiere decir que existe una constante c tal que el algoritmo tarda menos de cn^2 para todos los casos posibles.

Complejidades

- Para saber si un algoritmo es bueno en cuanto a su tiempo de ejecución es muy útil conocer su complejidad temporal, es decir, una función evaluada en el tamaño del problema que nos de una cota de la cantidad de operaciones del algoritmo.
- Por ejemplo, si el problema tiene un tamaño de entrada n la complejidad podría ser $O(< n^2)$, esto quiere decir que existe una constante c tal que el algoritmo tarda menos de cn^2 para todos los casos posibles.
- El cálculo de complejidades es un análisis teórico y no tiene en cuenta la constante c que puede ser muy chica o muy grande, sin embargo es por lo general una buena referencia para saber si un algoritmo es bueno o malo.

Complejidades

- Para saber si un algoritmo es bueno en cuanto a su tiempo de ejecución es muy útil conocer su complejidad temporal, es decir, una función evaluada en el tamaño del problema que nos de una cota de la cantidad de operaciones del algoritmo.
- Por ejemplo, si el problema tiene un tamaño de entrada n la complejidad podría ser $O(< n^2)$, esto quiere decir que existe una constante c tal que el algoritmo tarda menos de cn^2 para todos los casos posibles.
- El cálculo de complejidades es un análisis teórico y no tiene en cuenta la constante c que puede ser muy chica o muy grande, sin embargo es por lo general una buena referencia para saber si un algoritmo es bueno o malo.
- La complejidad del BFS y del DFS es $O(n + m) = O(m)$

Contenidos

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.
- 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.

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.

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?

Algoritmo de Dijkstra

El algoritmo de Dijkstra empieza en el vértice inicial, que empieza con distancia cero, y todos los demás vértices empiezan en distancia infinito, en cada paso el algoritmo actualiza las distancias de los vecinos del nodo actual cambiándolas, si las hace más chicas, por la distancia al nodo actual más el peso del eje que los une. Luego elige el vértice aún no visitado de distancia más chica y se mueve a ese vértice.

Hay varias versiones del algoritmo de Dijkstra de las cuales nosotros veremos dos, con y sin cola de prioridad, la diferencia está en cómo se elige a qué nodo moverse.

Distintas versiones de Dijkstra

Dijkstra sin cola de prioridad

Una vez que actualiza las distancias para elegir a qué nodo moverse revisa todos los nodos del grafo uno por uno y se queda con el más cercano aún no visitado.

Dijkstra con cola de prioridad

En cada paso guarda en una cola de prioridad los nodos aún no visitados ordenados según su distancia. La cola de prioridad es una estructura en la que se puede insertar y borrar en tiempo logarítmico en función de la cantidad de elementos de la cola y consultar el menor en tiempo constante. Así, siempre sabe en tiempo constante qué nodo es el próximo a visitar.

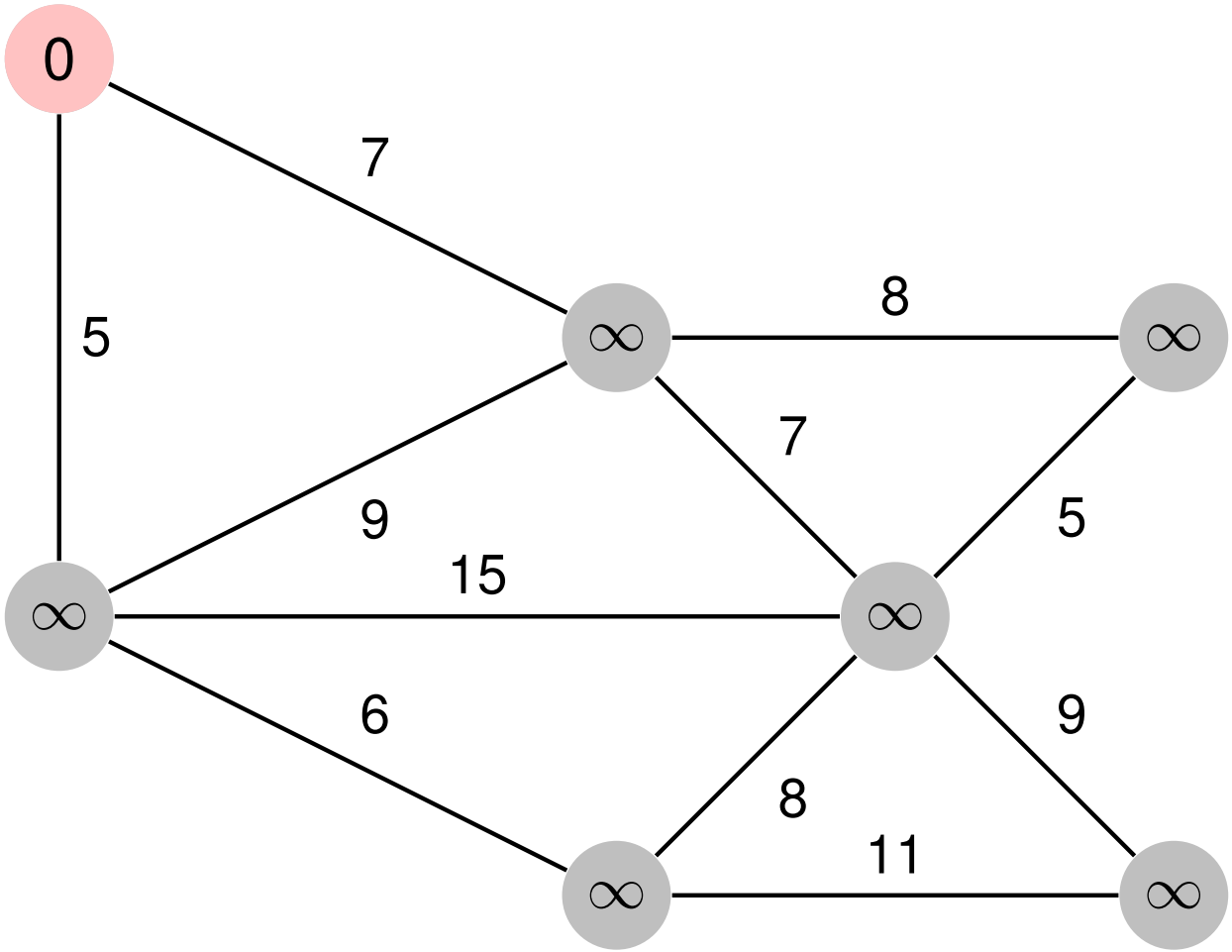
Distintas versiones de Dijkstra

- Nosotros veremos sólo el algoritmo sin cola de prioridad porque es más fácil de implementar el algoritmo sin la cola.

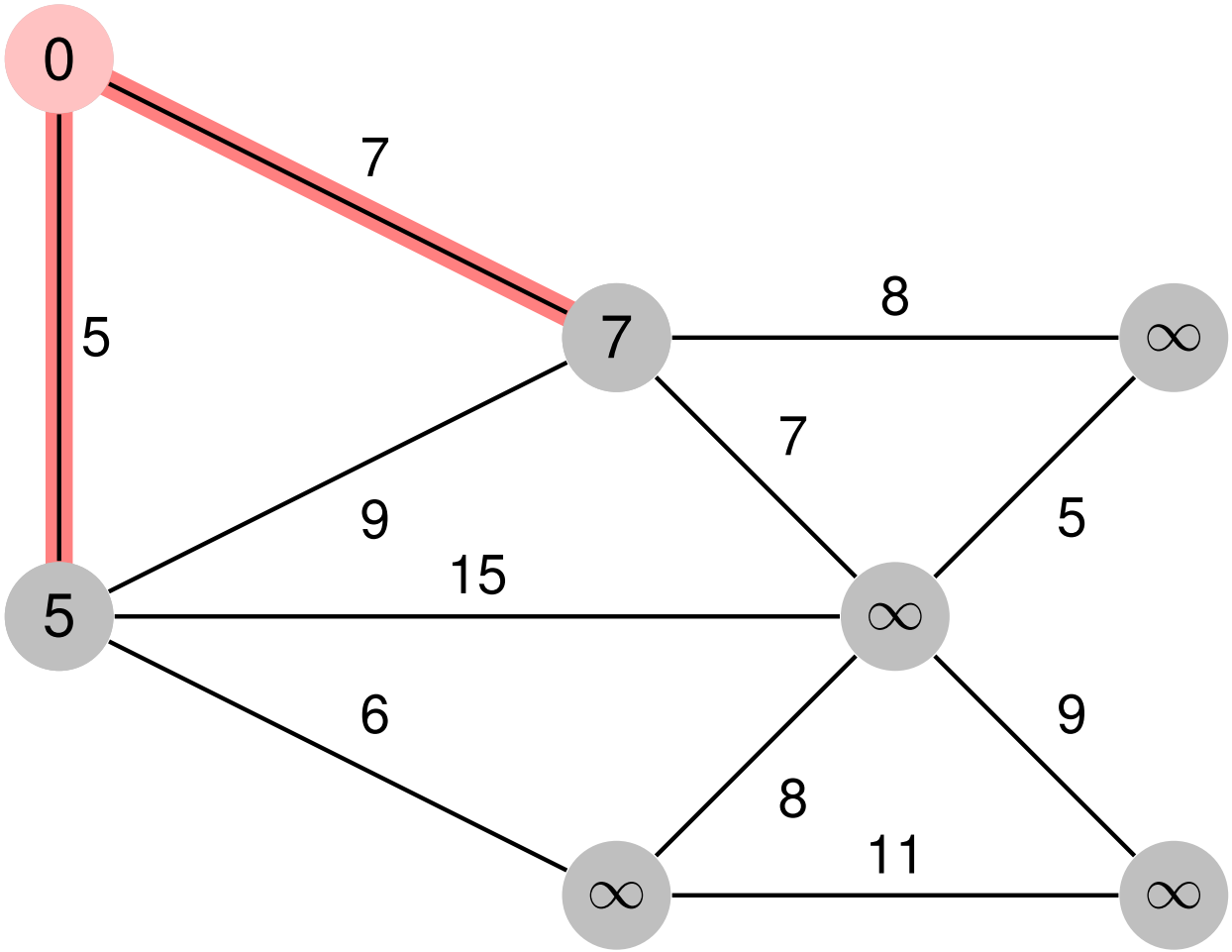
Distintas versiones de Dijkstra

- Nosotros veremos sólo el algoritmo sin cola de prioridad porque es más fácil de implementar el algoritmo sin la cola.
- La cola de prioridad es una estructura difícil de implementar pero el set que viene en la STL de C++ funciona como una cola de prioridad.

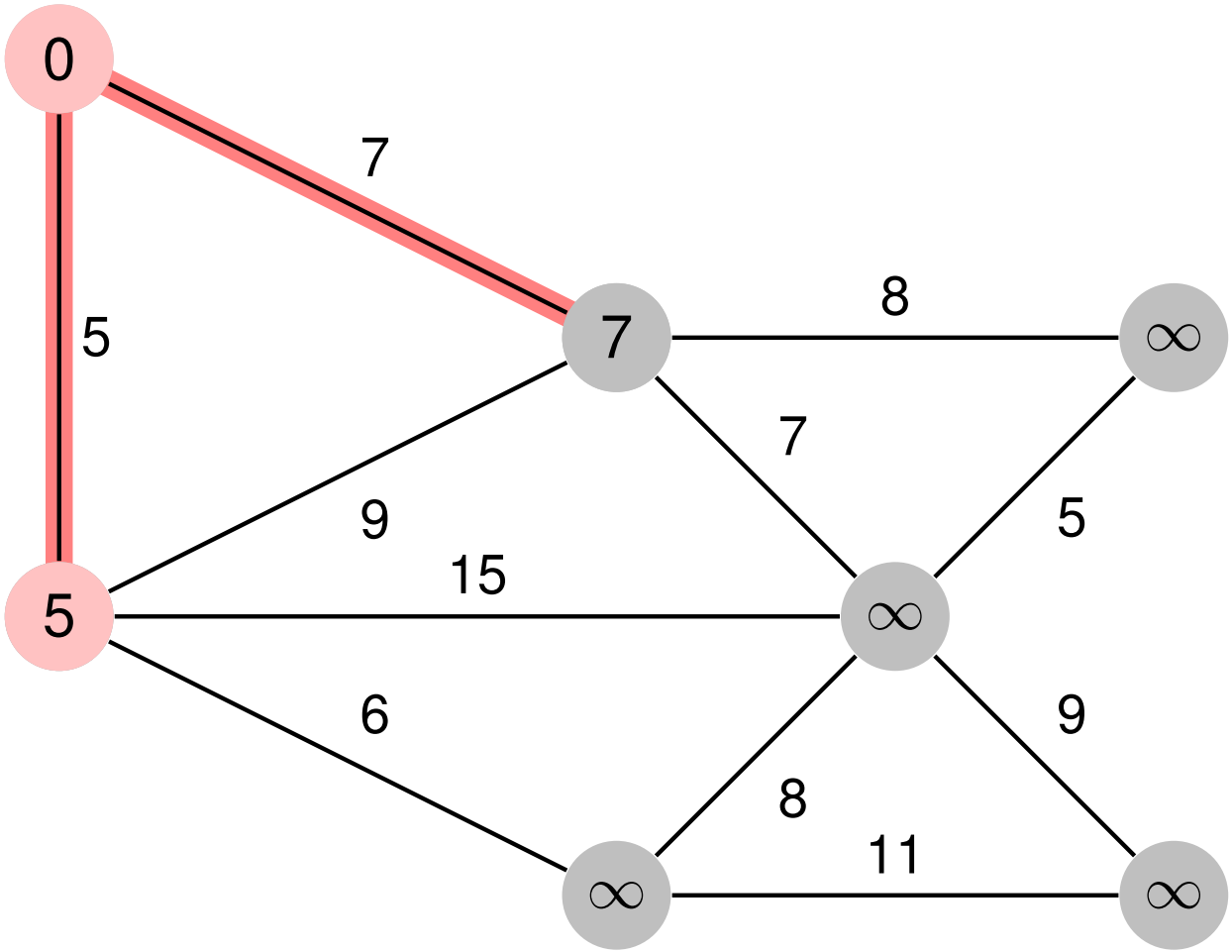
Ejemplo



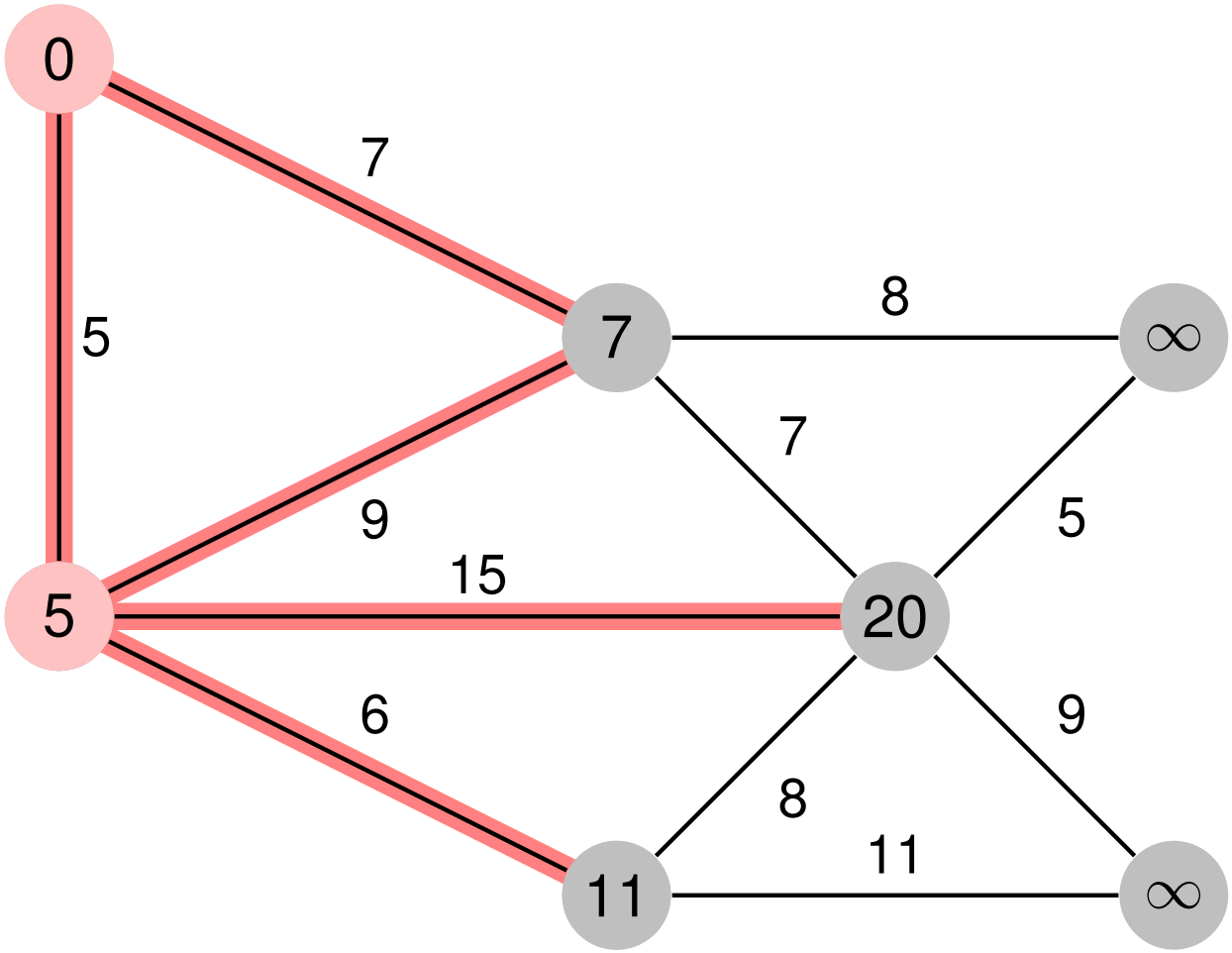
Ejemplo



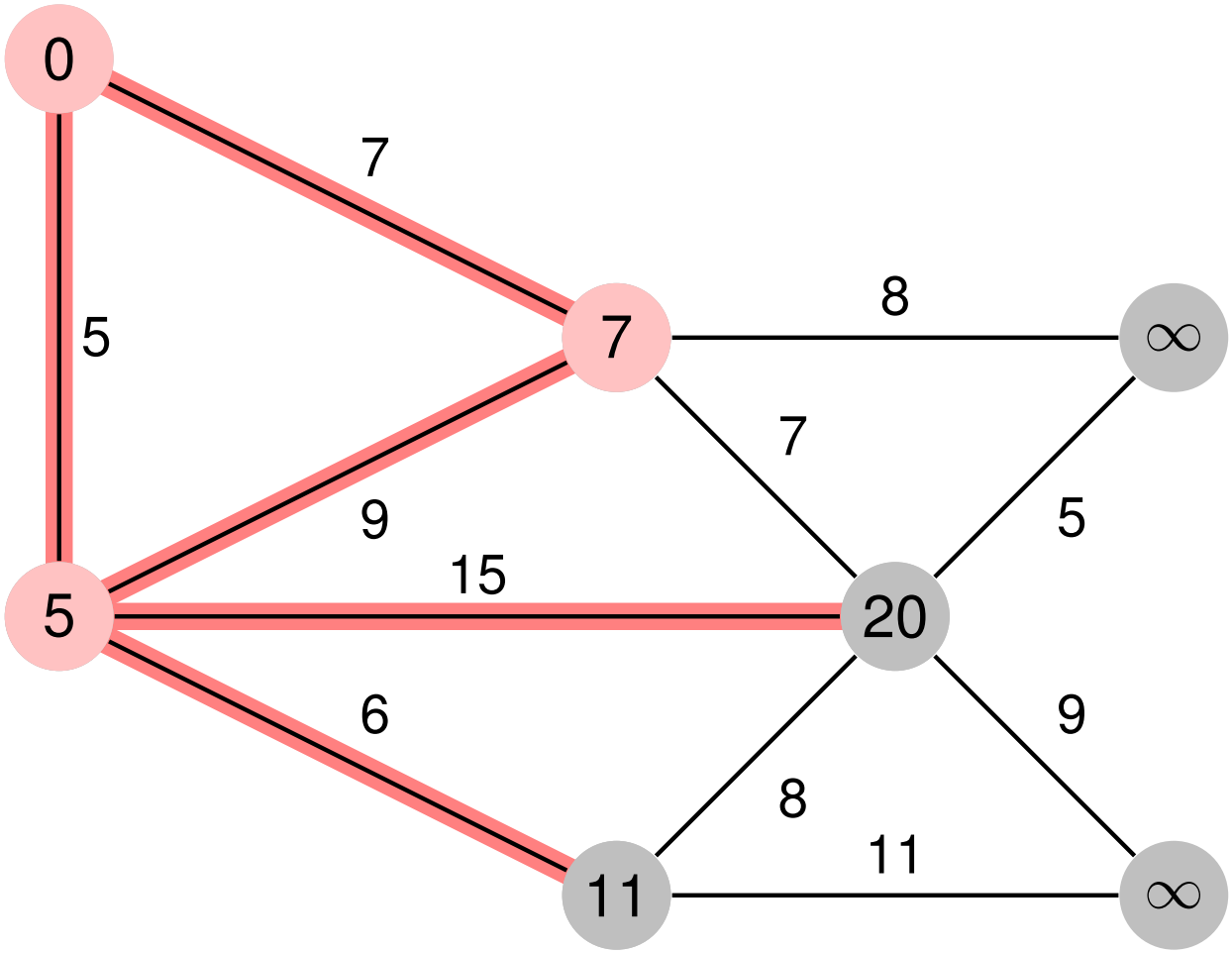
Ejemplo



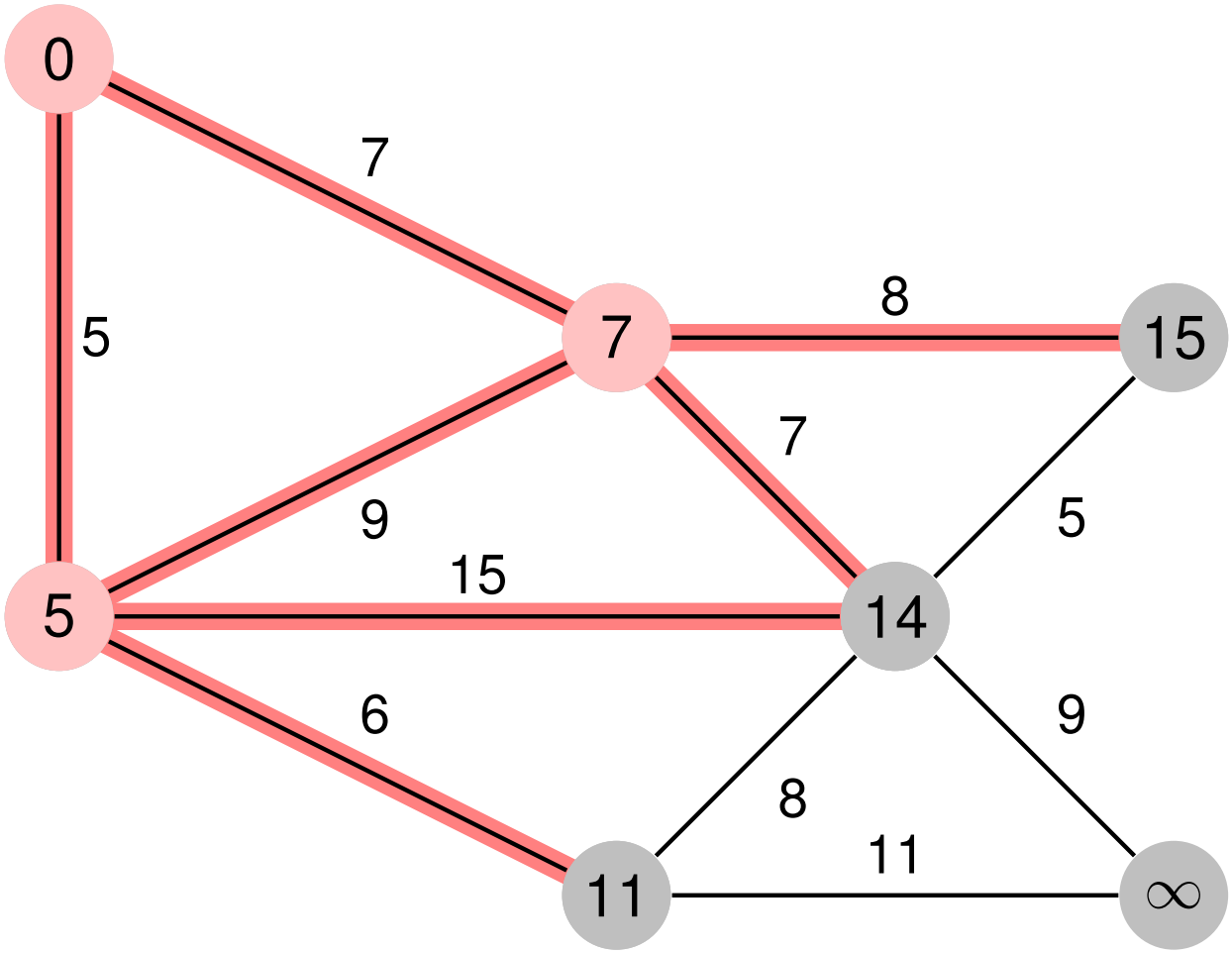
Ejemplo



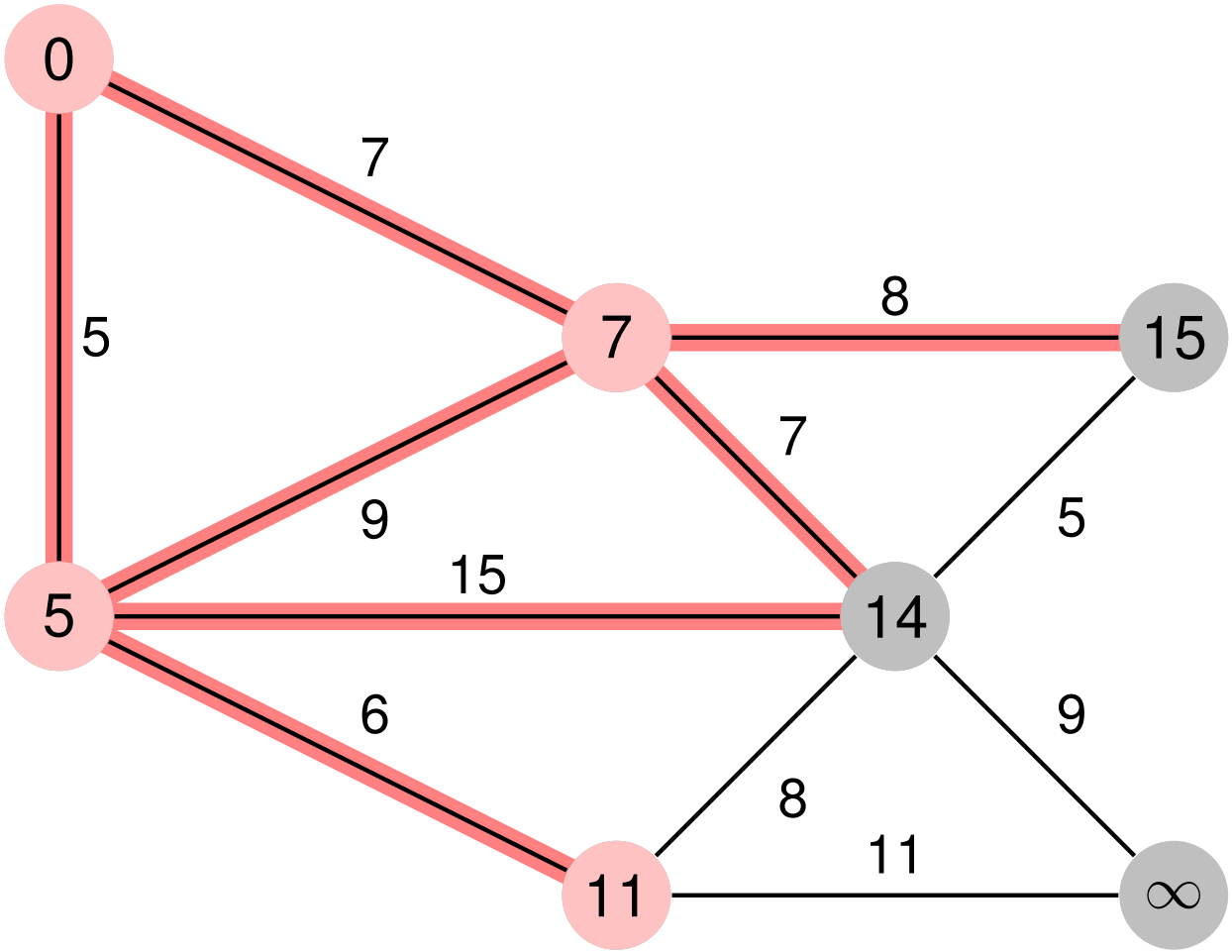
Ejemplo



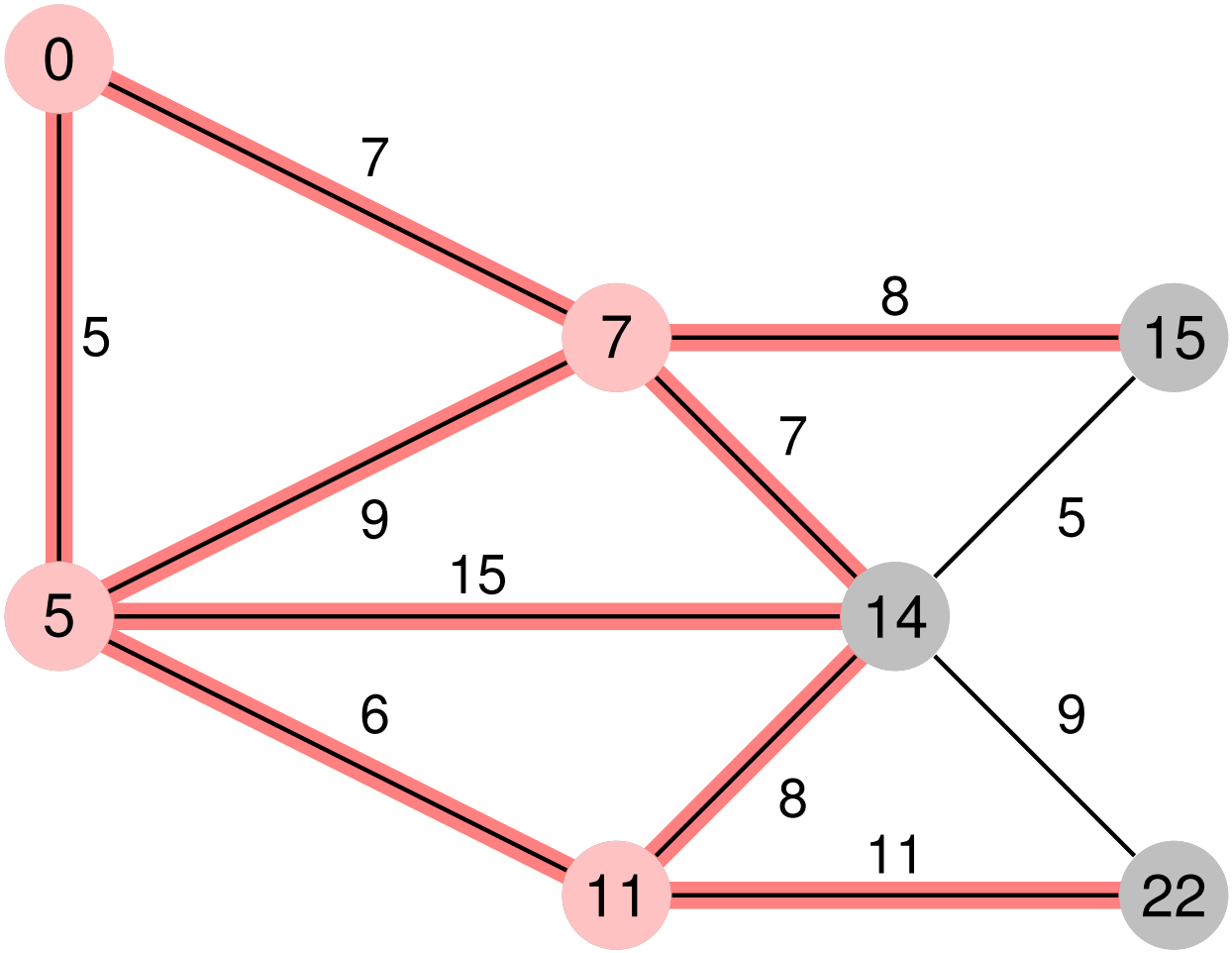
Ejemplo



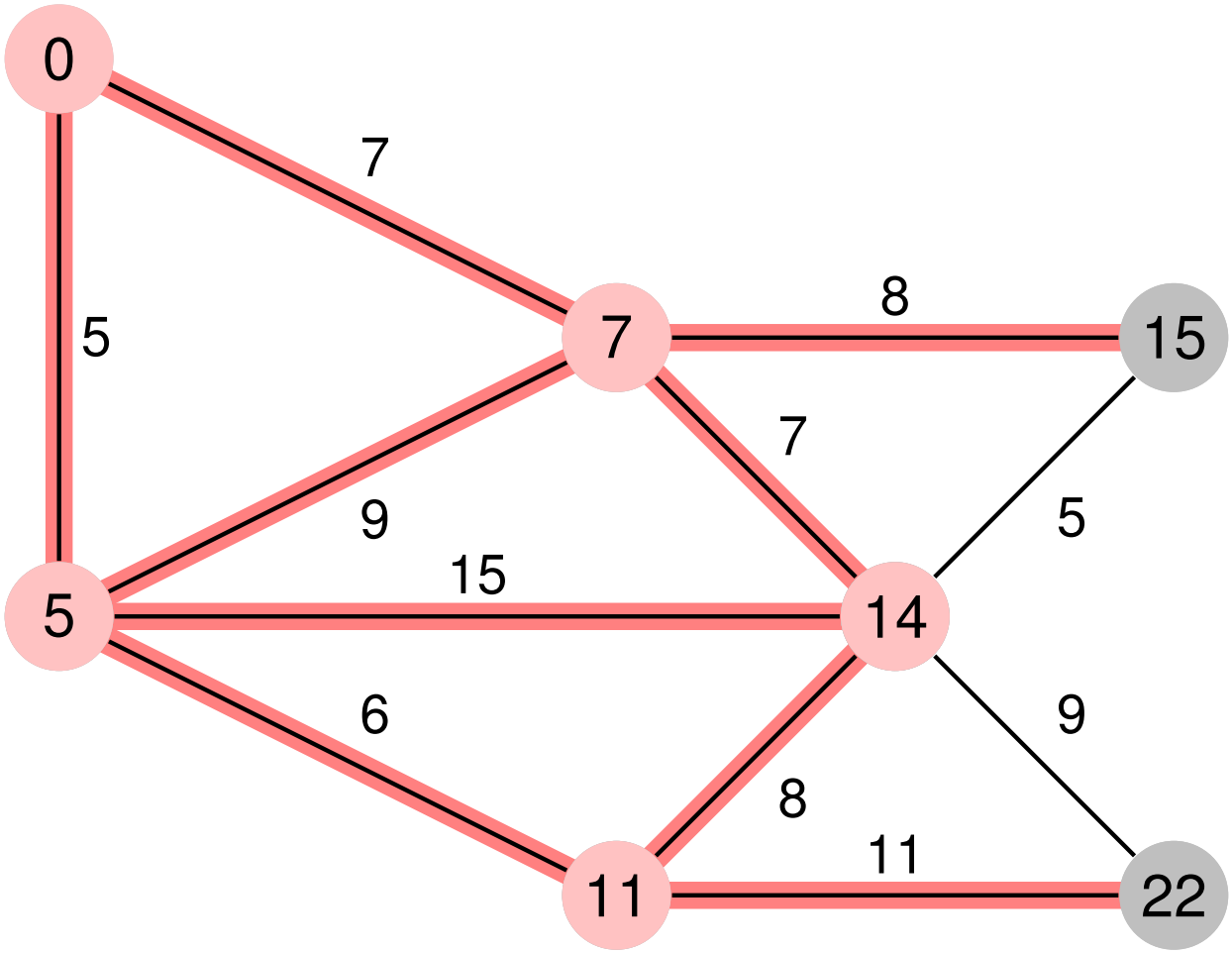
Ejemplo



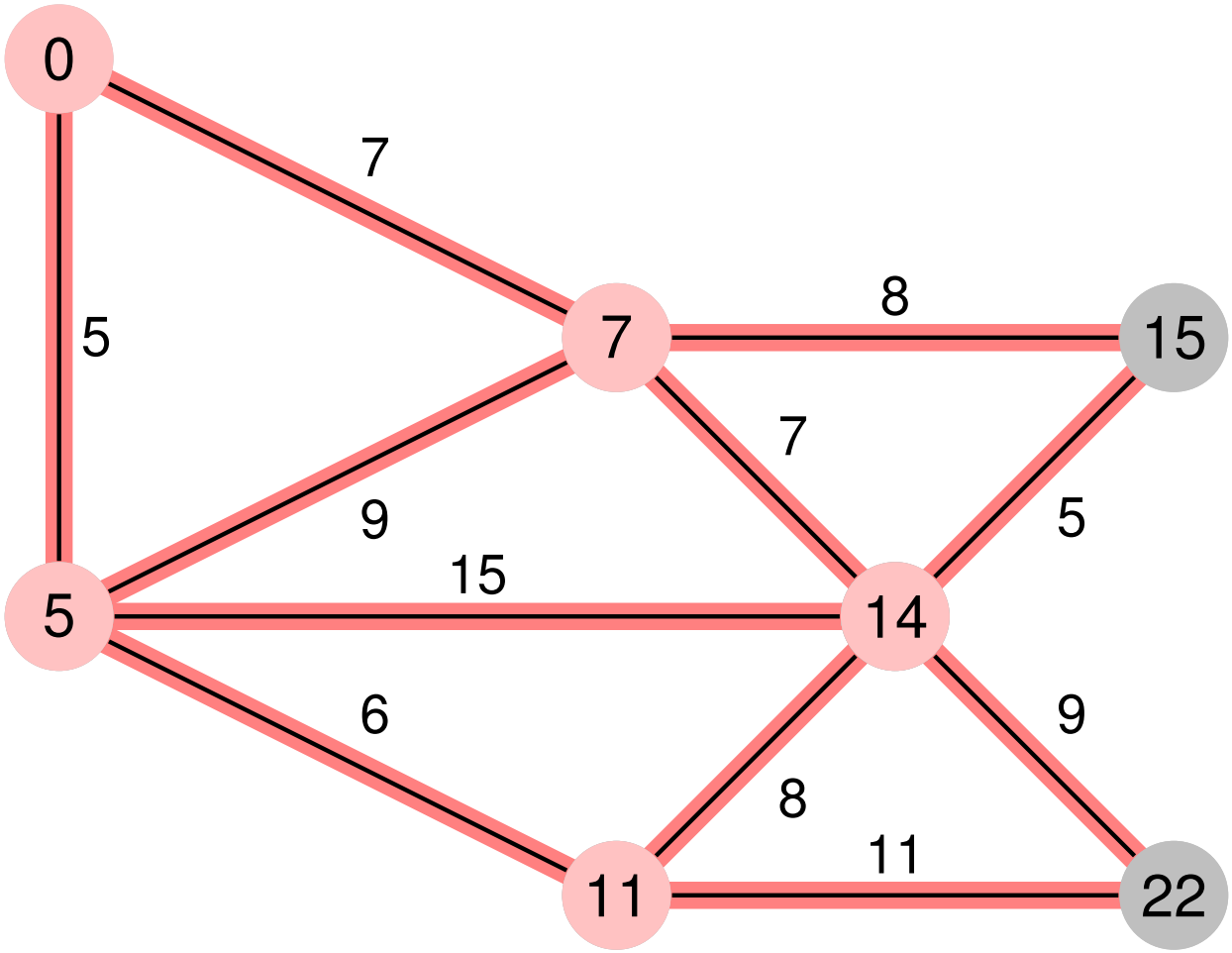
Ejemplo



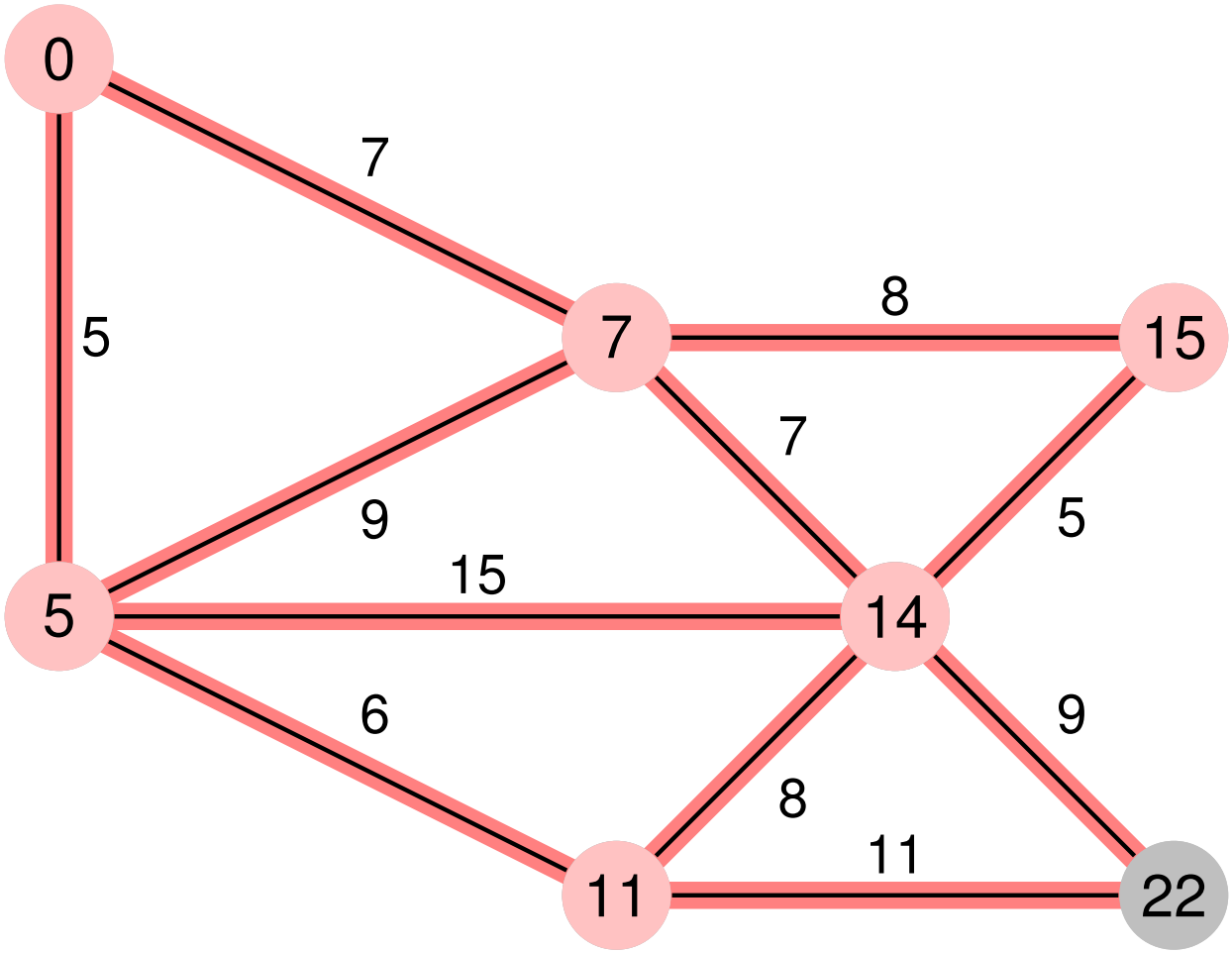
Ejemplo



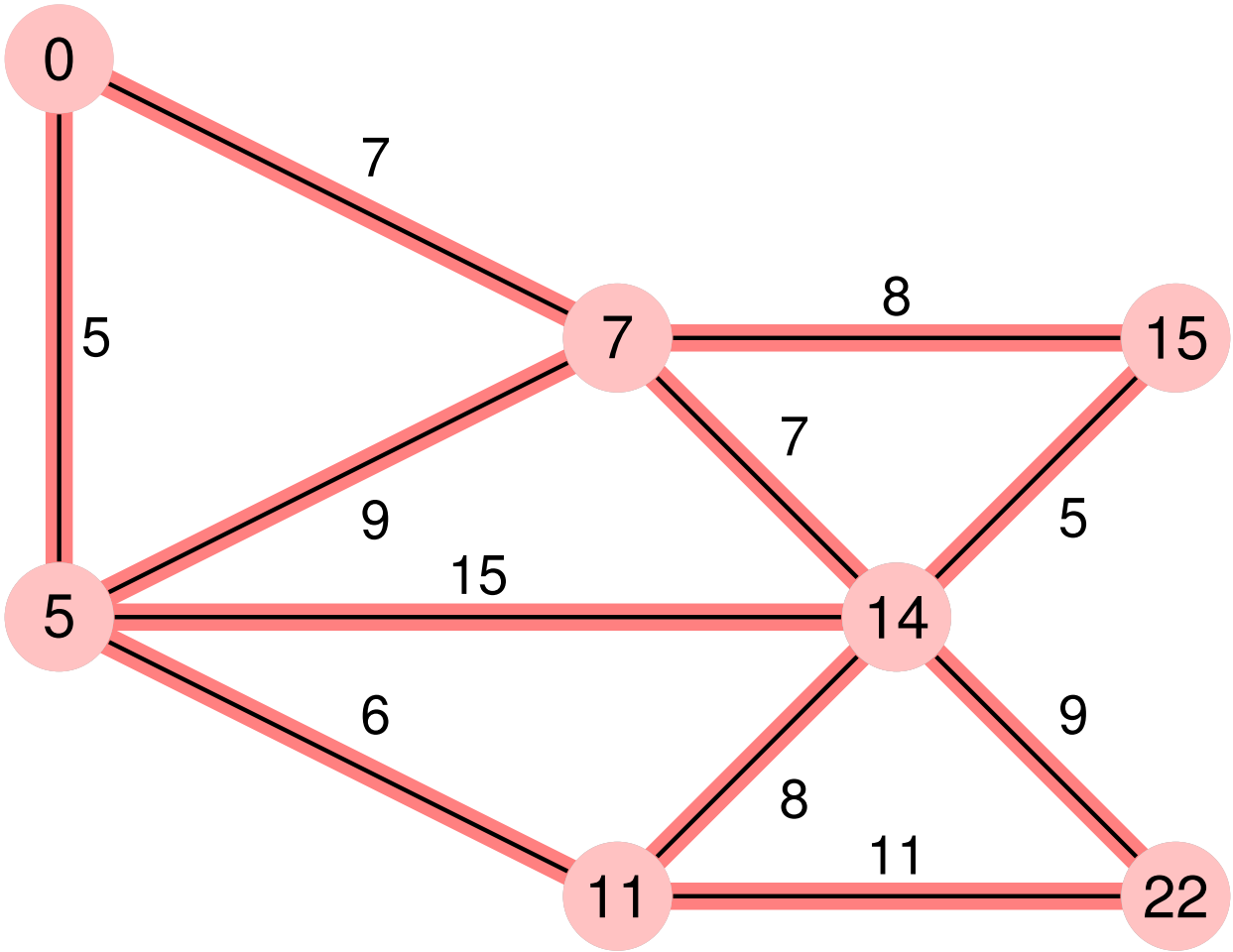
Ejemplo



Ejemplo



Ejemplo



Implementación de Dijkstra

```
1  vector<int> dijkstra(vector<vector<pair<int,int> > > &lista,int nodoInicial)
2  {
3      int n = lista.size();
4      vector<int> dist(n,INF);
5      vector<bool> toc(n,false);
6      dist[nodoInicial] = 0;
7      int t = nodoInicial;
8      for(int i=0;i<n;i++){
9          toc[t] = true;
10         for(int i=0;i<lista[t].size();i++)
11             dist[lista[t][i].first] = min(dist[lista[t][i].first],dist[t]+
                   lista[t][i].second);
12         for(int i=0;i<n;i++)
13         if(toc[t]==true||(toc[i]==false&&dist[i]<dist[t]))
14             t = i;
15     }
16     return dist;
17 }
```

Análisis de Dijkstra

- Cuando usamos el valor INF es un valor previamente definido como un virtual infinito, puede ser, por ejemplo, la suma de todos los pesos de los ejes más uno.

Análisis de Dijkstra

- Cuando usamos el valor INF es un valor previamente definido como un virtual infinito, puede ser, por ejemplo, la suma de todos los pesos de los ejes más uno.
- La complejidad del algoritmo de Dijkstra es $O(n^2 + m)$ y como $E = O(n^2)$ entonces la complejidad es $O(n^2)$. La versión con cola de prioridad tiene complejidad $O((m + n) \log n)$

Análisis de Dijkstra

- Cuando usamos el valor INF es un valor previamente definido como un virtual infinito, puede ser, por ejemplo, la suma de todos los pesos de los ejes más uno.
- La complejidad del algoritmo de Dijkstra es $O(n^2 + m)$ y como $E = O(n^2)$ entonces la complejidad es $O(n^2)$. La versión con cola de prioridad tiene complejidad $O((m + n) \log n)$
- El algoritmo de Dijkstra funciona sólo con pesos no negativos. Para calcular camino mínimo en un grafo con pesos negativos existe el algoritmo de Bellman-Ford, que además de calcular camino mínimo detecta ciclos negativos.
- ... Lo vemos la otra clase