# Grafos: repaso de contenidos de Algoritmos III

Melanie Sclar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Problemas, Algoritmos y Programación

# Formas de representar un grafo

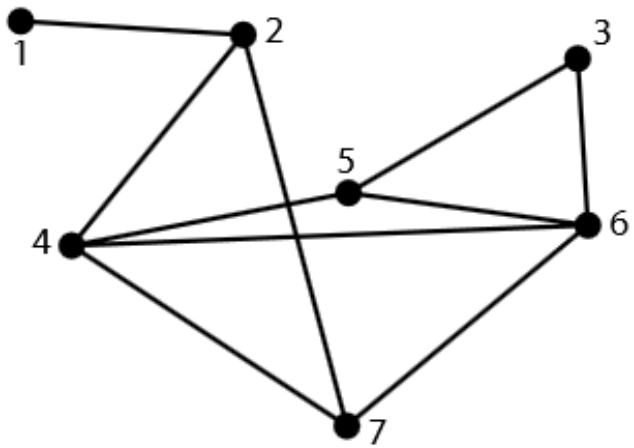
Existen varias maneras de guardar un grafo en memoria para poder luego consultar cosas (por ejemplo, recorrer el grafo, que es lo que haremos hoy).
Veamos las dos más populares.

# Matriz de adyacencia

## 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) si no.

Ej:




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

# Matriz de adyacencia

Esta es una de las representaciones más utilizadas. Si bien el ejemplo es para un grafo no dirigido, también se puede utilizar la misma estructura para grafos dirigidos y grafos con pesos.

**Ventajas**

# Matriz de adyacencia

Esta es una de las representaciones más utilizadas. Si bien el ejemplo es para un grafo no dirigido, también se puede utilizar la misma estructura para grafos dirigidos y grafos con pesos.

**Ventajas**

- Permite saber si existe o no arista entre dos nodos cualesquiera en O(1).
- Es muy fácil de implementar, $matrizAdy[i][j]$ guarda toda la información sobre la arista.

# Matriz de adyacencia

Esta es una de las representaciones más utilizadas. Si bien el ejemplo es para un grafo no dirigido, también se puede utilizar la misma estructura para grafos dirigidos y grafos con pesos.

**Ventajas**

- Permite saber si existe o no arista entre dos nodos cualesquiera en O(1).
- Es muy fácil de implementar, $matrizAdy[i][j]$ guarda toda la información sobre la arista.

**Desventajas**

# Matriz de adyacencia

Esta es una de las representaciones más utilizadas. Si bien el ejemplo es para un grafo no dirigido, también se puede utilizar la misma estructura para grafos dirigidos y grafos con pesos.

**Ventajas**

- Permite saber si existe o no arista entre dos nodos cualesquiera en O(1).
- Es muy fácil de implementar, $matrizAdy[i][j]$ guarda toda la información sobre la arista.

**Desventajas**

- La complejidad espacial: se necesitan $n^2$ casillas para representar un grafo de $n$ nodos.

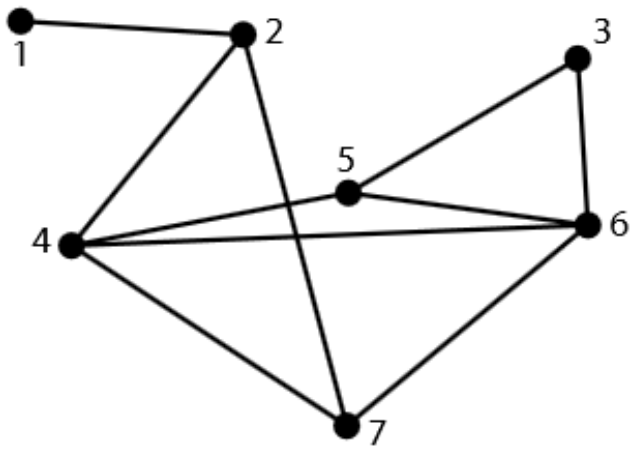
# Lista de adyacencia

## 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*.

Coloquialmente la llamamos *lista de vecinos* pues para cada nodo guardamos la lista de nodos para los que existe una arista que los conecta (o sea, los vecinos).

Ej:




$L_1 : 2$
$L_2 : 1 \longrightarrow 4 \longrightarrow 7$
$L_3 : 5 \longrightarrow 6$
$L_4 : 2 \longrightarrow 5 \longrightarrow 6 \longrightarrow 7$
$L_5 : 3 \longrightarrow 4 \longrightarrow 6$
$L_6 : 3 \longrightarrow 4 \longrightarrow 5 \longrightarrow 7$
$L_7 : 2 \longrightarrow 4 \longrightarrow 6$

# Lista de adyacencia

Nuevamente, con la misma idea también se pueden modelar grafos dirigidos y con pesos.

La complejidad espacial de esta representación será posiblemente mucho menor. ¿Cuánta memoria necesitaremos para un grafo de $n$ nodos y $m$ aristas?

# Lista de adyacencia

Nuevamente, con la misma idea también se pueden modelar grafos dirigidos y con pesos.

La complejidad espacial de esta representación será posiblemente mucho menor. ¿Cuánta memoria necesitaremos para un grafo de $n$ nodos y $m$ aristas? **O(m+n)**

# Recorrer un grafo

A continuación vamos a ver dos algoritmos utilizados para recorrer grafos. Luego, podremos utilizar estos algoritmos para calcular lo que necesitemos (por ejemplo, distancias a un nodo en particular) o para encontrar un nodo en particular, que tenga una propiedad.

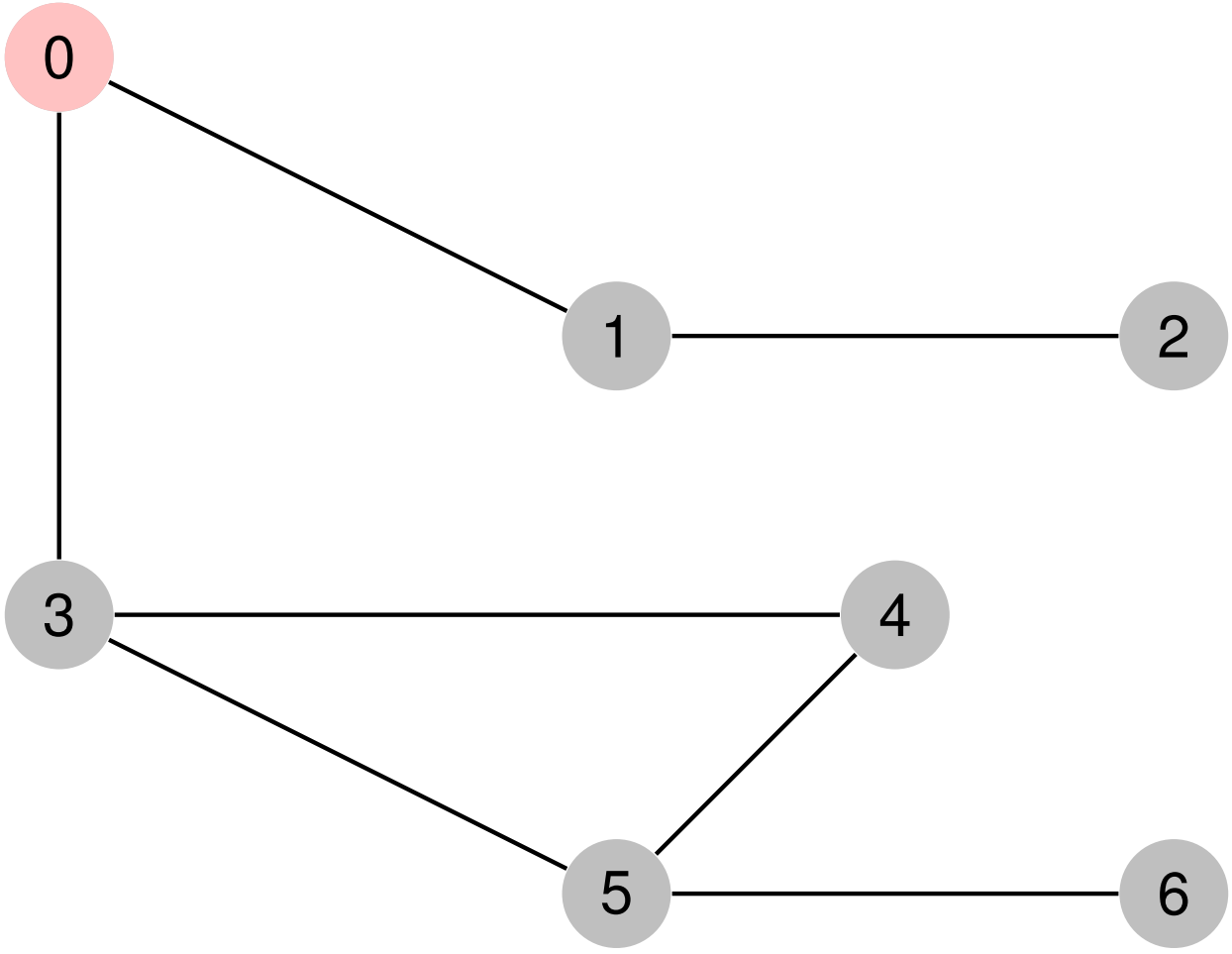
Ambos algoritmos tendrán una complejidad de $O(n + m)$, si $n$ es la cantidad de nodos y $m$ la de aristas, pero cada uno será útil en situaciones específicas.

# Una forma intuitiva de recorrer el grafo: DFS

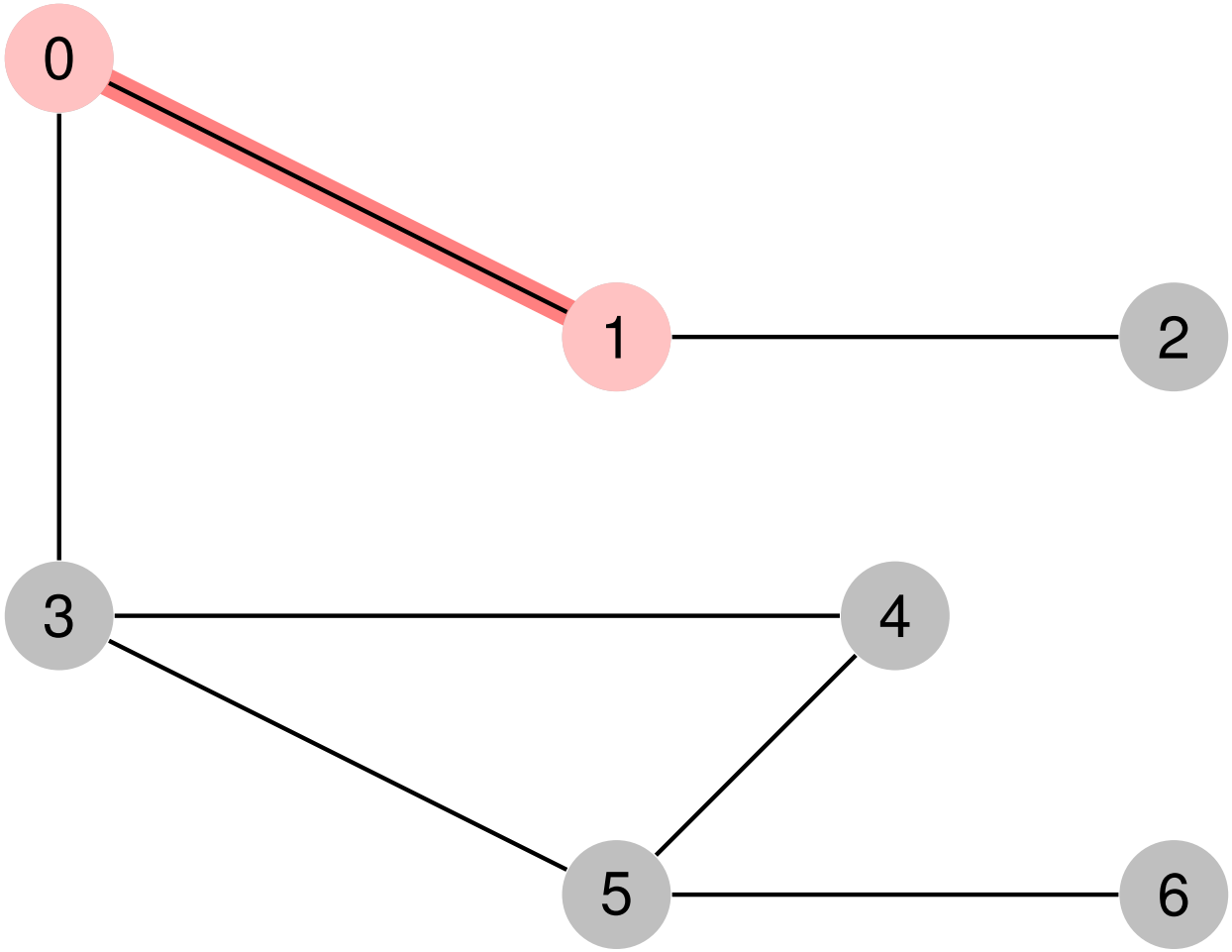
El DFS (Depth First Search) es un tipo de recorrido del grafo. Se dice que lo recorre *en profundidad*, es decir, empieza por el nodo inicial y en cada paso visita un nodo vecino no visitado del nodo donde está parado, si no hay nodos por visitar vuelve para atrás.

Veamos un ejemplo visual de cómo se recorre un grafo con DFS.

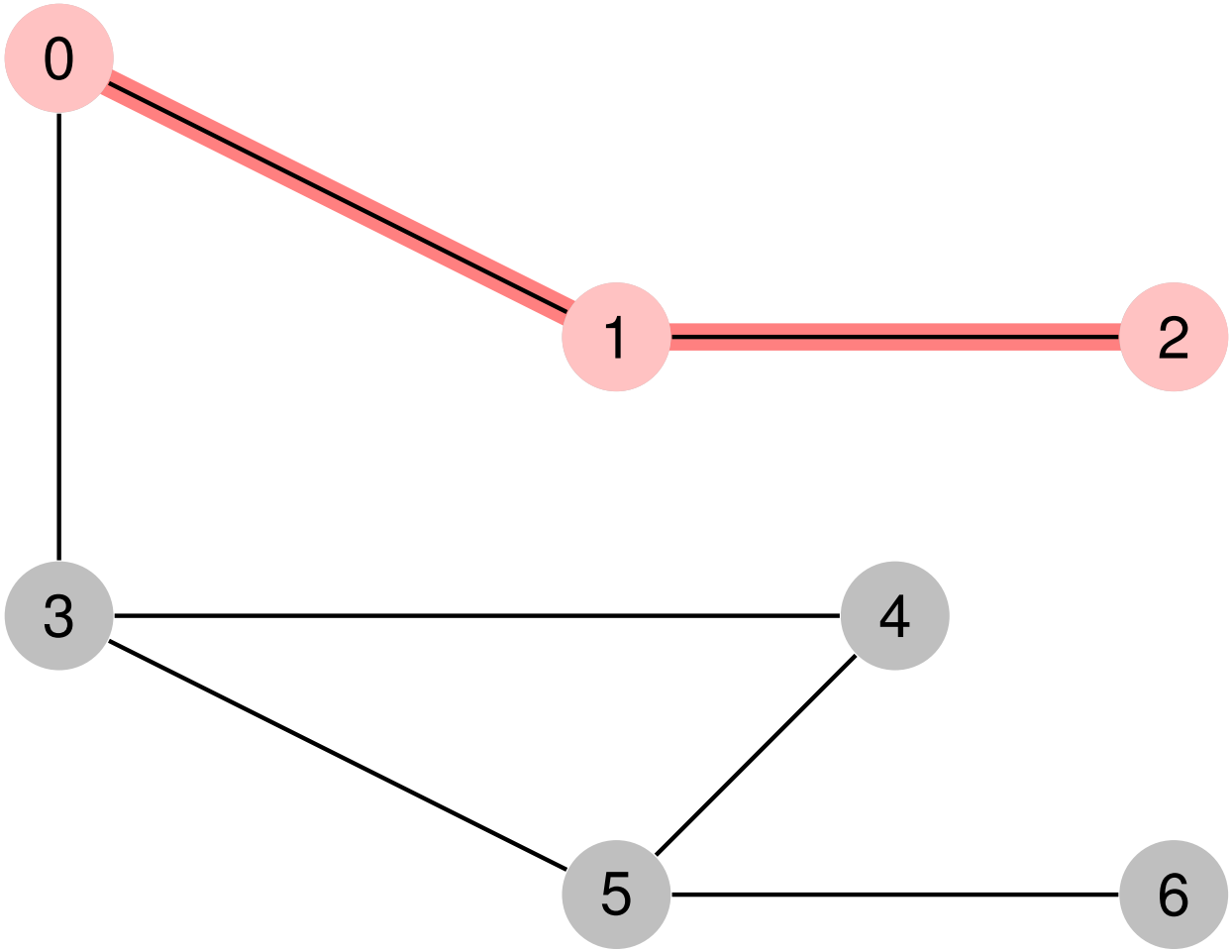
# Ejemplo



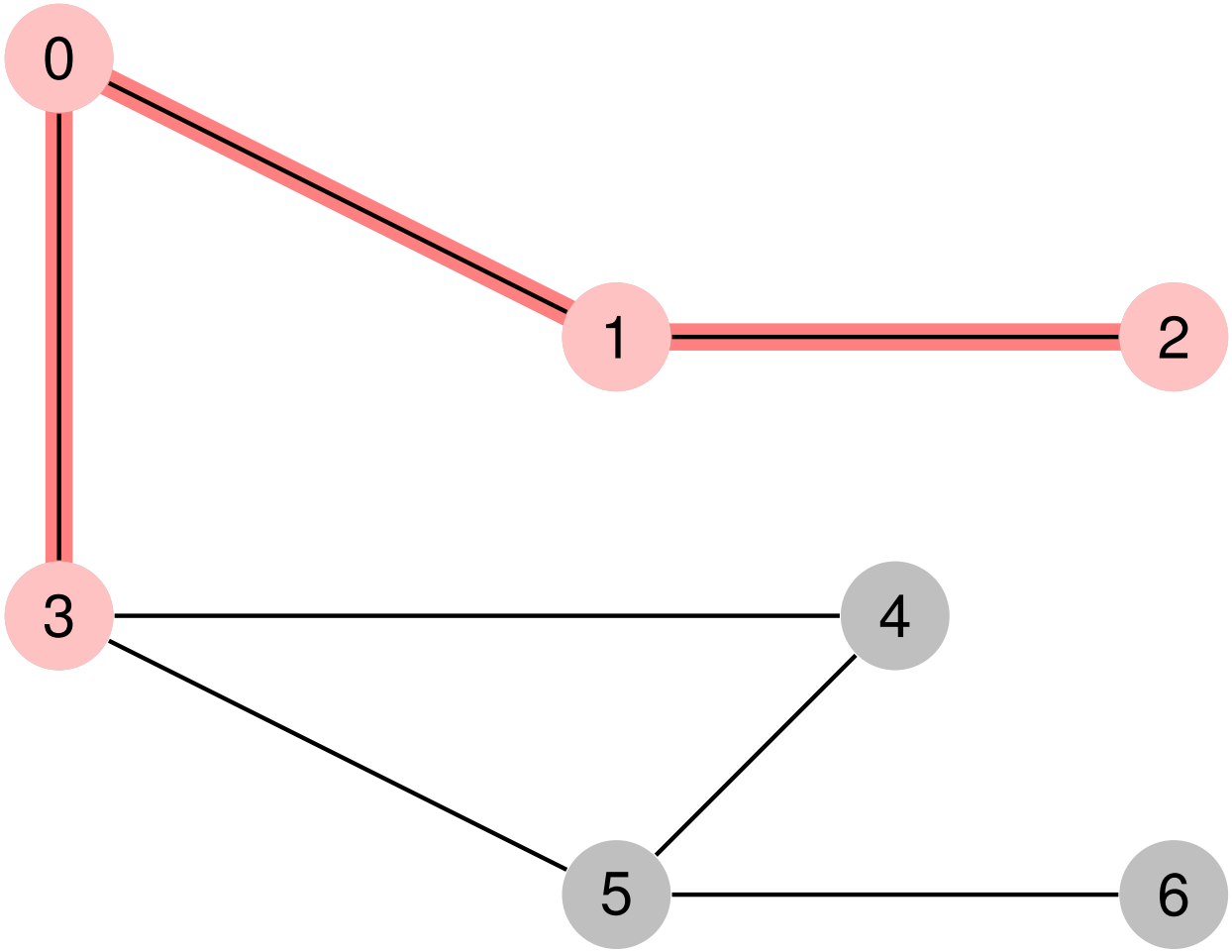
# Ejemplo



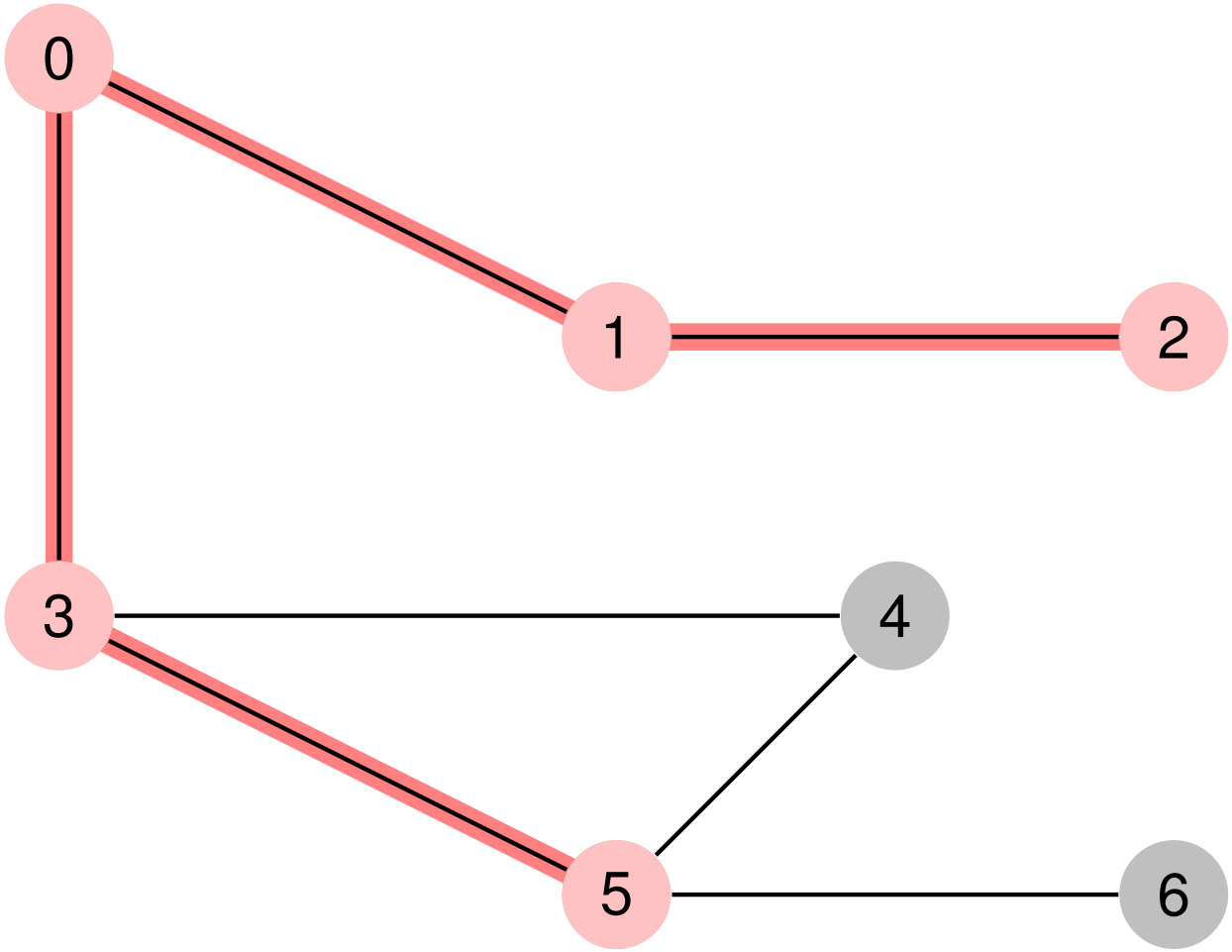
# Ejemplo



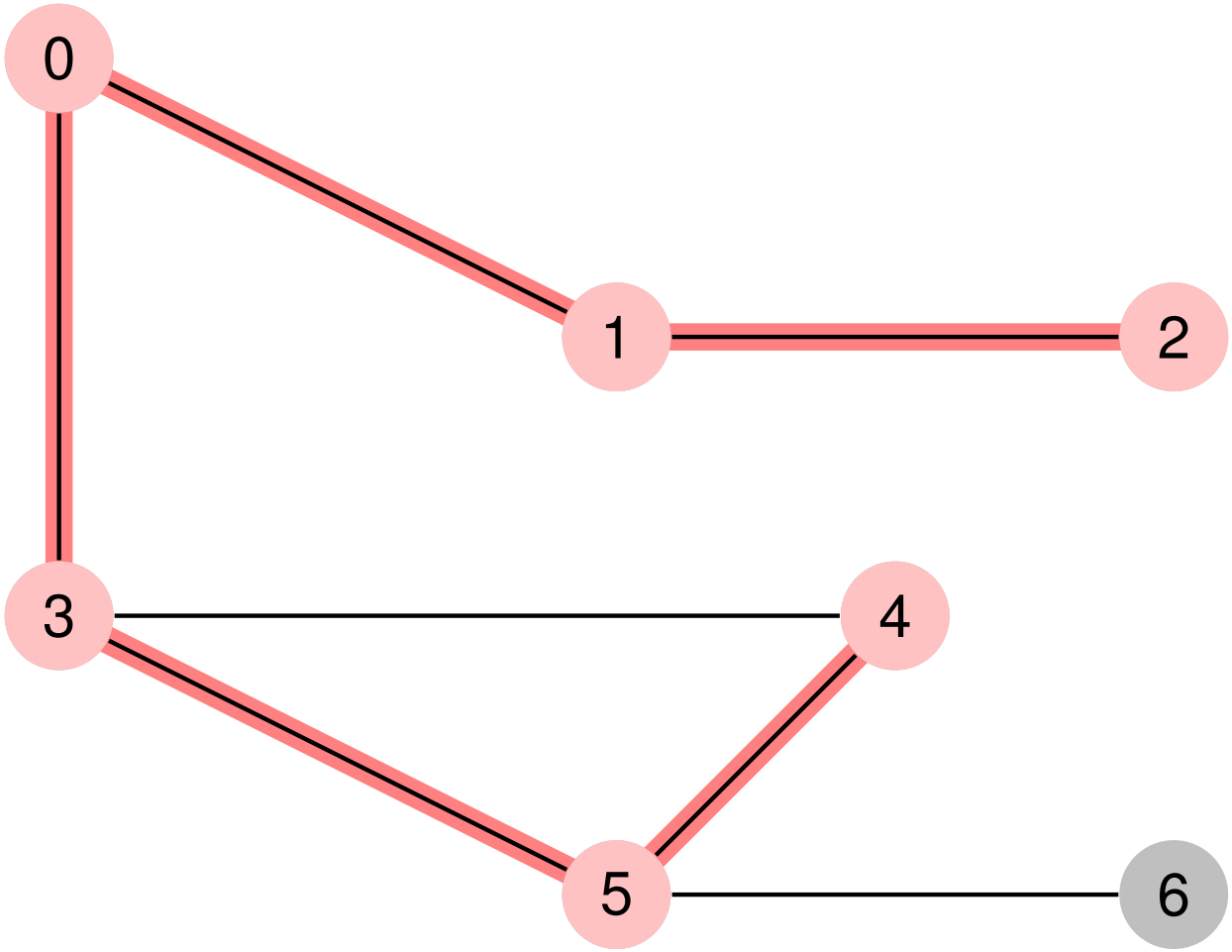
# Ejemplo



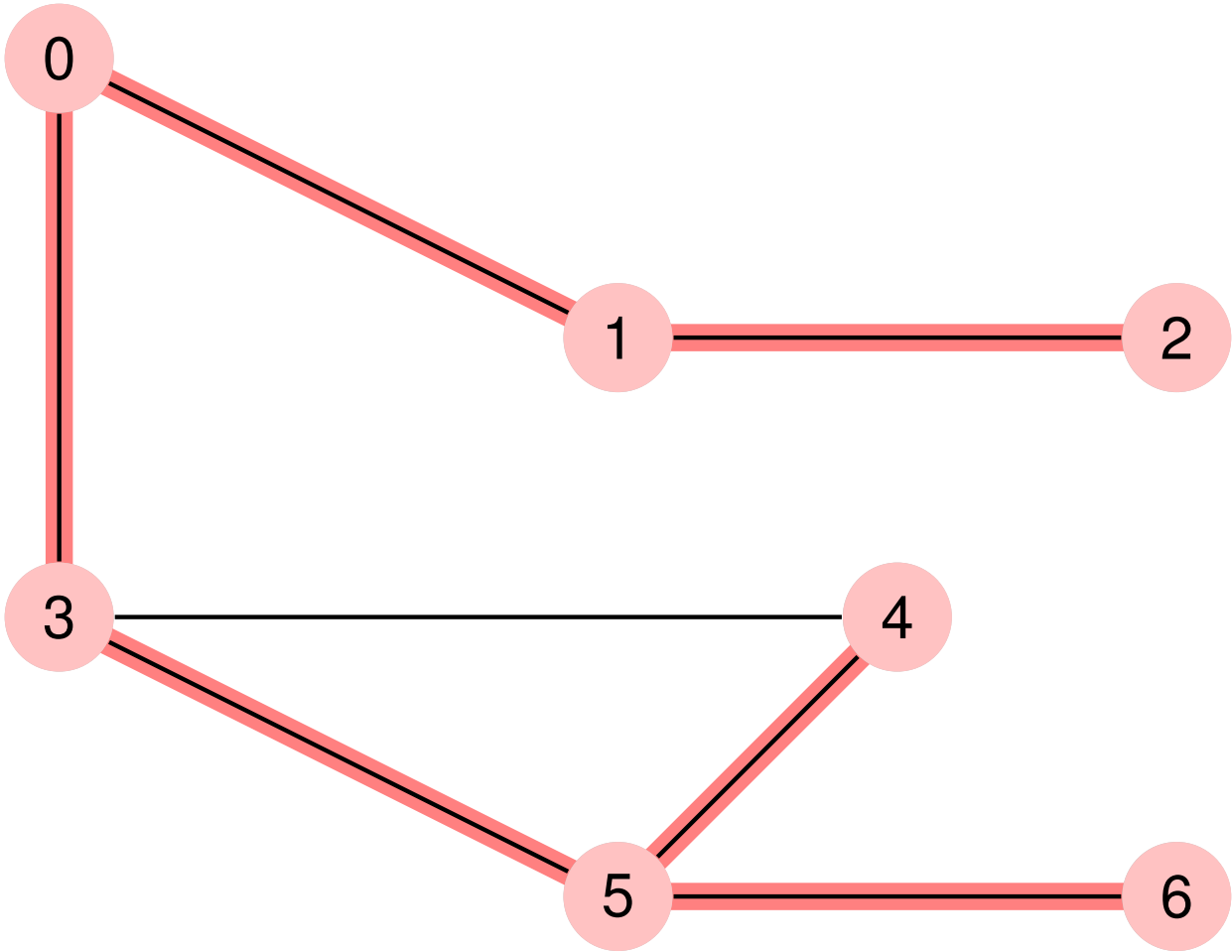
# Ejemplo



# Ejemplo



# Ejemplo



# Ejemplo

En el ejemplo anterior vimos cómo recorre el grafo un DFS. Ahora tratemos de implementar un DFS resolviendo un problema (por supuesto, DFS se puede aplicar a muchos problemas más).

## Problema

Dado un grafo conexo (es decir, que existe al menos un camino entre todo par de nodos) queremos ver si dicho grafo es un **árbol**.

¿Pero qué significa que un grafo sea un **árbol**?

## Árboles

Un árbol es un tipo especial de grafo no dirigido. Es un grafo donde entre cualquier par de nodos existe un único camino que los conecta (notar que en particular, todos los nodos tienen que estar conectados entre sí).

Los árboles tienen mil propiedades interesantes, como $m = n - 1$, que no tiene ciclos y más cosas sobre las que hoy no hablaremos.

# Ideas para la resolución

- Primero que nada, como tenemos un grafo vamos a querer recorrerlo y detectar si existe más de un camino entre algún par de nodos. ¿Pero cómo podemos hacer esto?

# Ideas para la resolución

- Primero que nada, como tenemos un grafo vamos a querer recorrerlo y detectar si existe más de un camino entre algún par de nodos. ¿Pero cómo podemos hacer esto?
- Iremos recorriendo el grafo con DFS, siempre moviéndome a un vecino no visitado aún. Para no visitar un nodo dos veces (eso sería innecesario y empeoraría la performance) marcamos cada nodo cuando lo revisamos. ¿Qué significa si llego a un nodo y un vecino de él (que no es por el que llegué) ya está visitado?

# Ideas para la resolución

- Primero que nada, como tenemos un grafo vamos a querer recorrerlo y detectar si existe más de un camino entre algún par de nodos. ¿Pero cómo podemos hacer esto?
- Iremos recorriendo el grafo con DFS, siempre moviéndome a un vecino no visitado aún. Para no visitar un nodo dos veces (eso sería innecesario y empeoraría la performance) marcamos cada nodo cuando lo revisamos. ¿Qué significa si llego a un nodo y un vecino de él (que no es por el que llegué) ya está visitado?
- Significa que desde mi nodo inicial pude llegar a un nodo de dos formas distintas: es decir que no existe un único camino, lo que contradice la definición de árbol. Ergo, no es árbol.

# Pseudocódigo del DFS

```
1  // inicialmente se llena el vector visitado en false pues ningun nodo fue
       visitado
2  // ademas el nodo inicial no tiene padre y por eso se llama con padre = -1
3  bool esArbol(grafo, int nodoActual, vector<bool> &visitado, int padre):
4      visitado[nodoActual] = true
5
6      para cada vecino v de nodoActual en grafo:
7          si v ya fue visitado y v != padre:
8              devolver false
9              // encontre dos caminos distintos y luego no es arbol
10         si v no fue visitado aun:
11             si no esArbol(grafo, v, visitado, nodoActual):
12             // llamo recursivamente al dfs con el subarbol con raiz en v,
                   ahora el padre es nodoActual
13                 devolver false
14
15     // si para ningun vecino encontre algun ciclo, es un arbol porque se que
           es conexo
16     devolver true
```

# Implementación en C++ del DFS

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

# 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 cual 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 cual 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 cual 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.

# Otro algoritmo de búsqueda: BFS

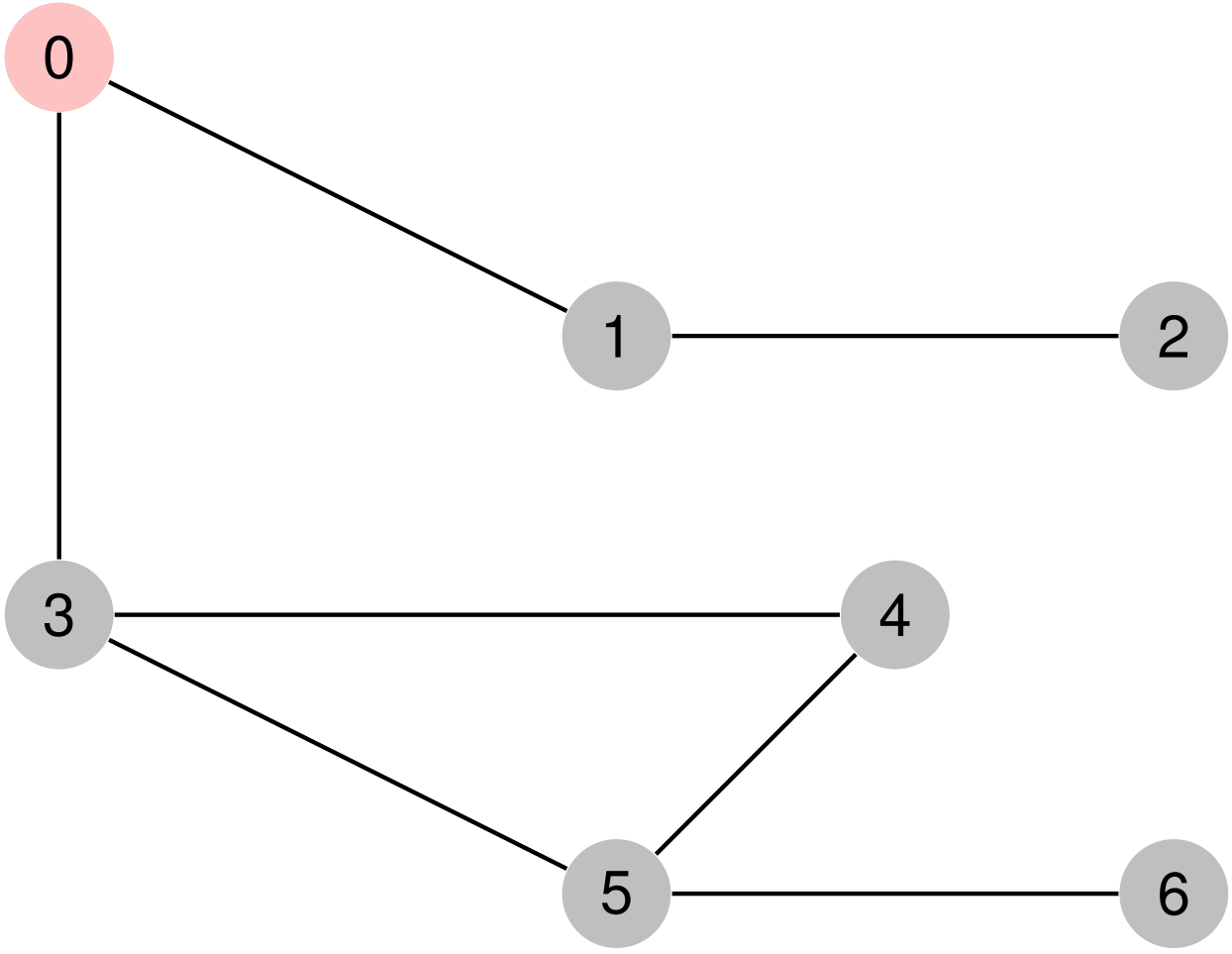
Muchas veces no basta con recorrer el grafo, sino que queremos hacerlo de una forma en particular. BFS suele ser muy útil en muchos problemas, veamos de qué se trata!

## Breadth First Search o *búsqueda en anchura*

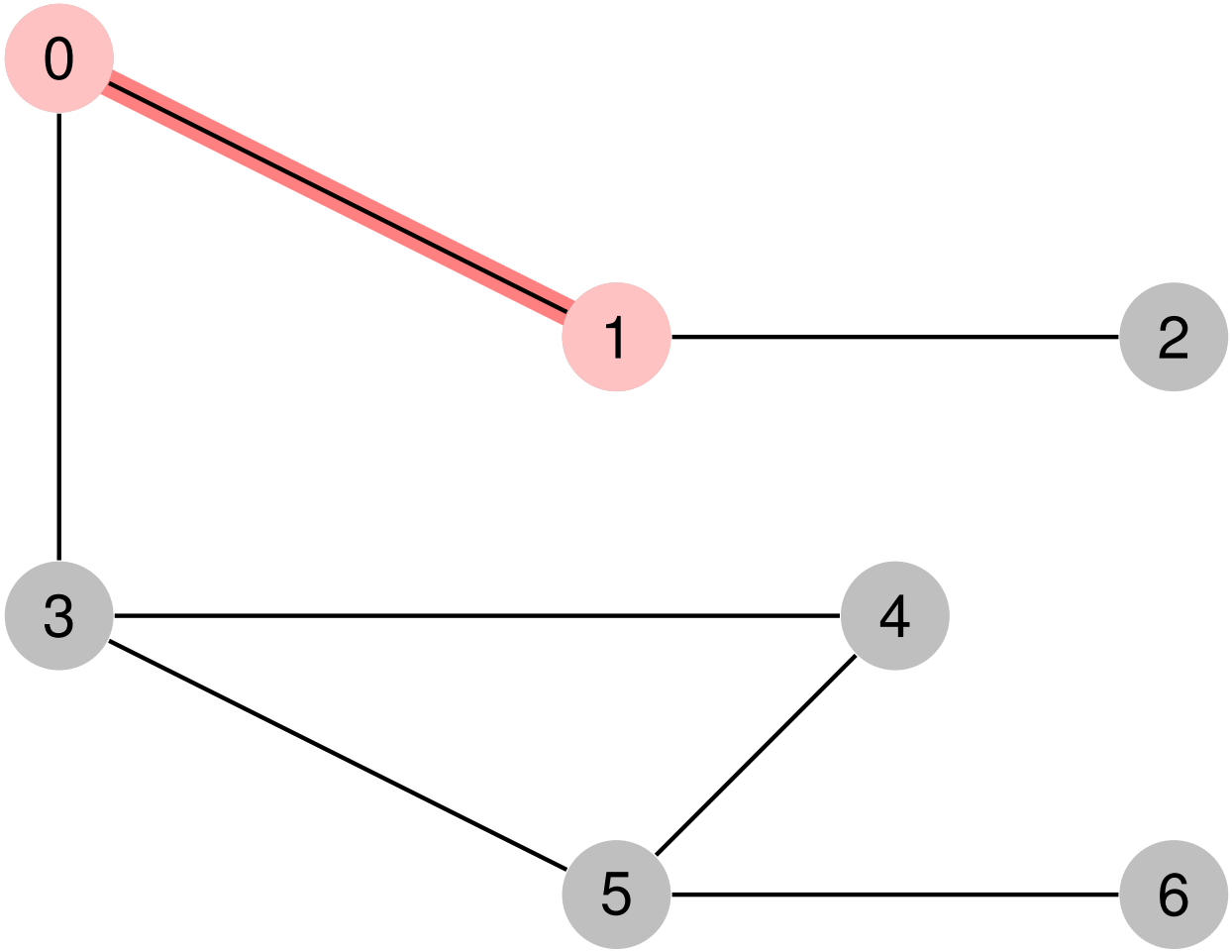
El Breadth First Search o *búsqueda en anchura* es un algoritmo que recorre un grafo comenzando por un nodo en particular, y se exploran todos los vecinos del nodo. Luego, se exploran todos los vecinos de los vecinos del nodo inicial, y así siguiendo hasta recorrer todos los nodos.

Veámoslo gráficamente.

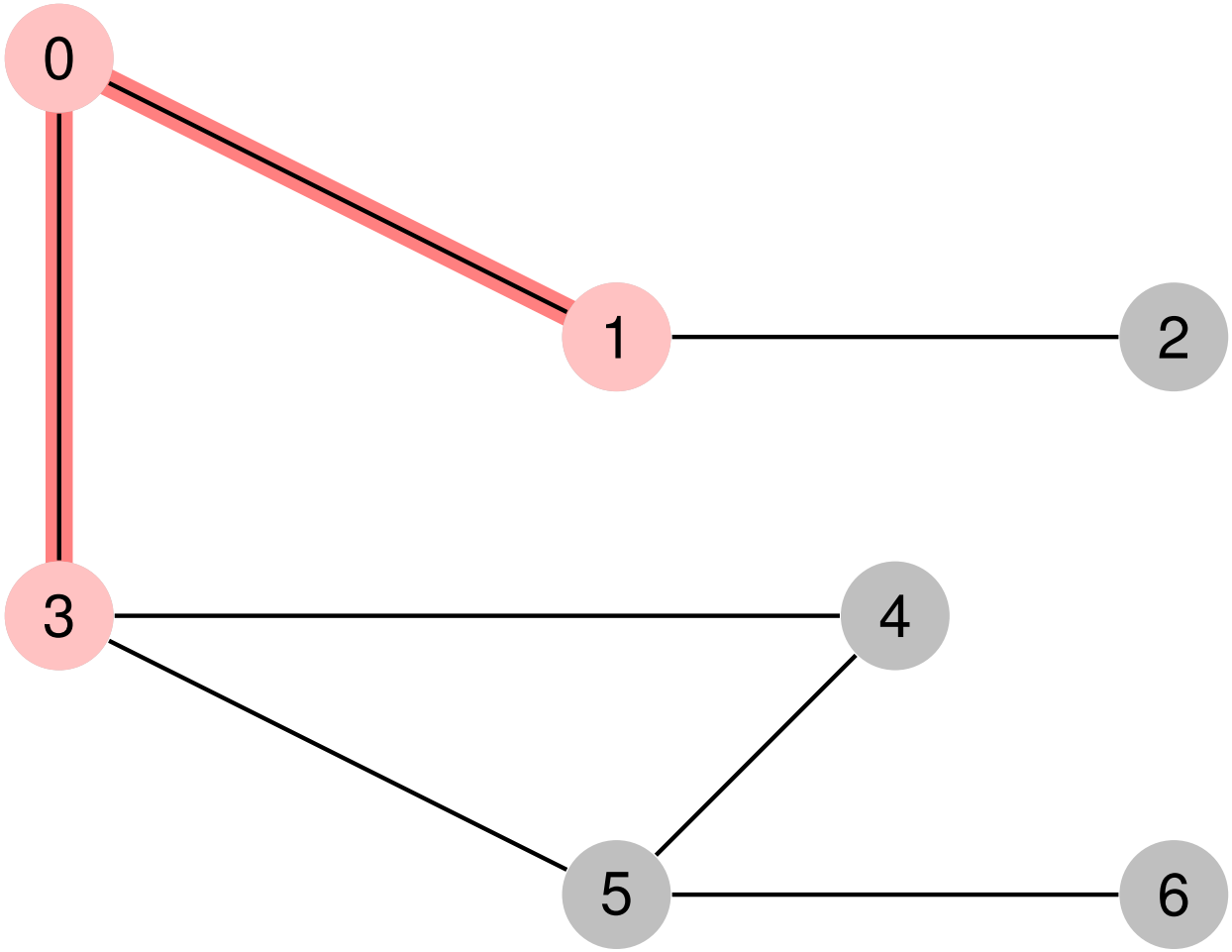
# Ejemplo de recorrido de un grafo usando BFS



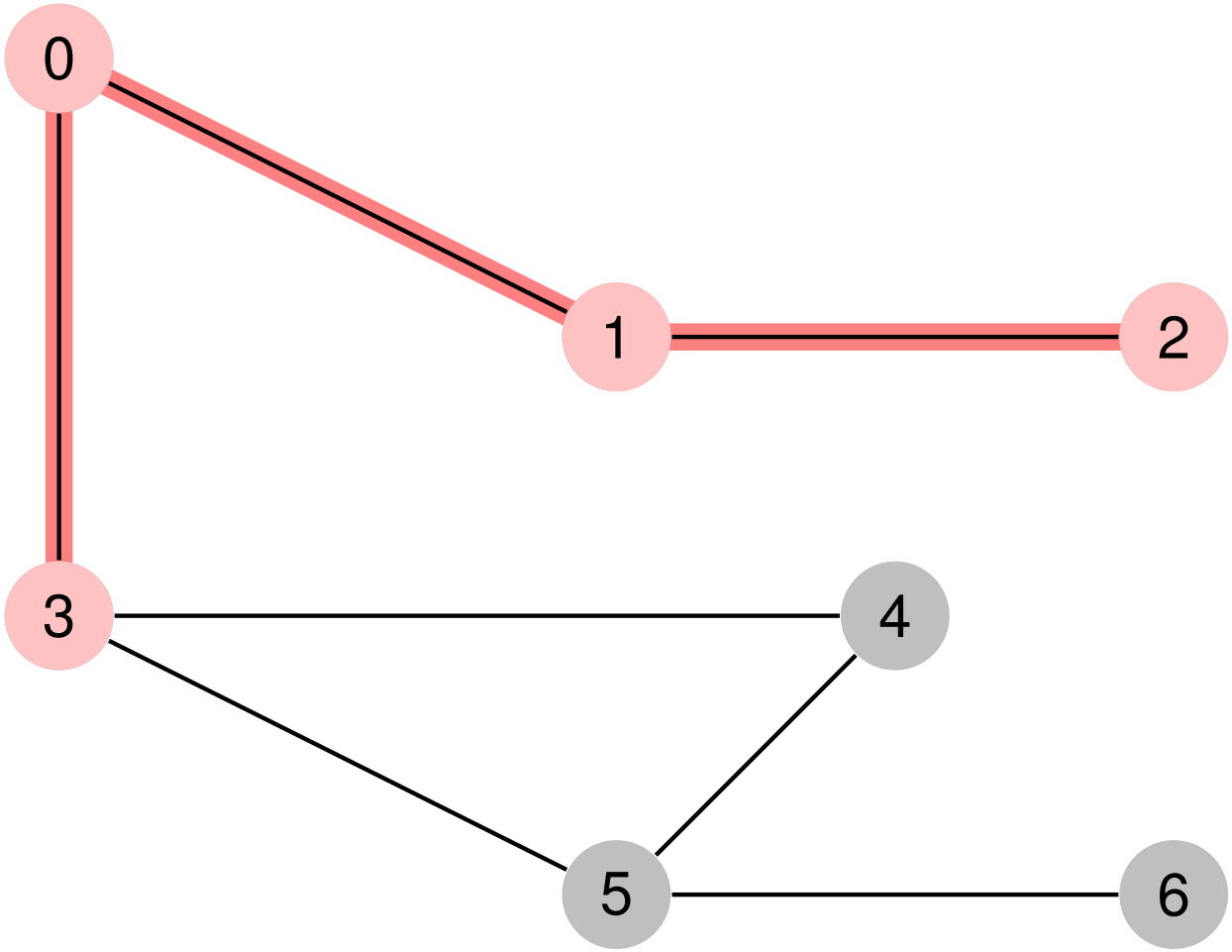
# Ejemplo de recorrido de un grafo usando BFS



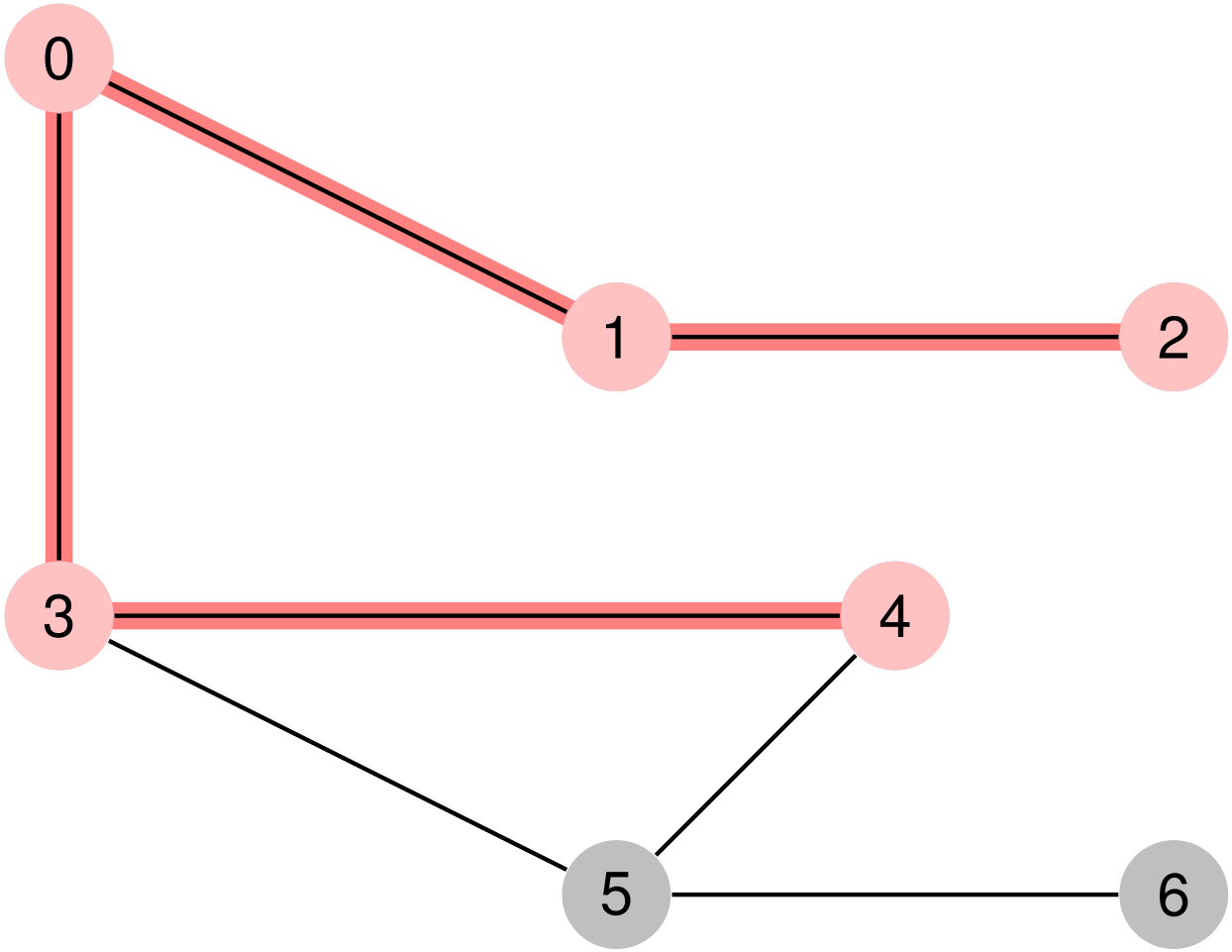
# Ejemplo de recorrido de un grafo usando BFS



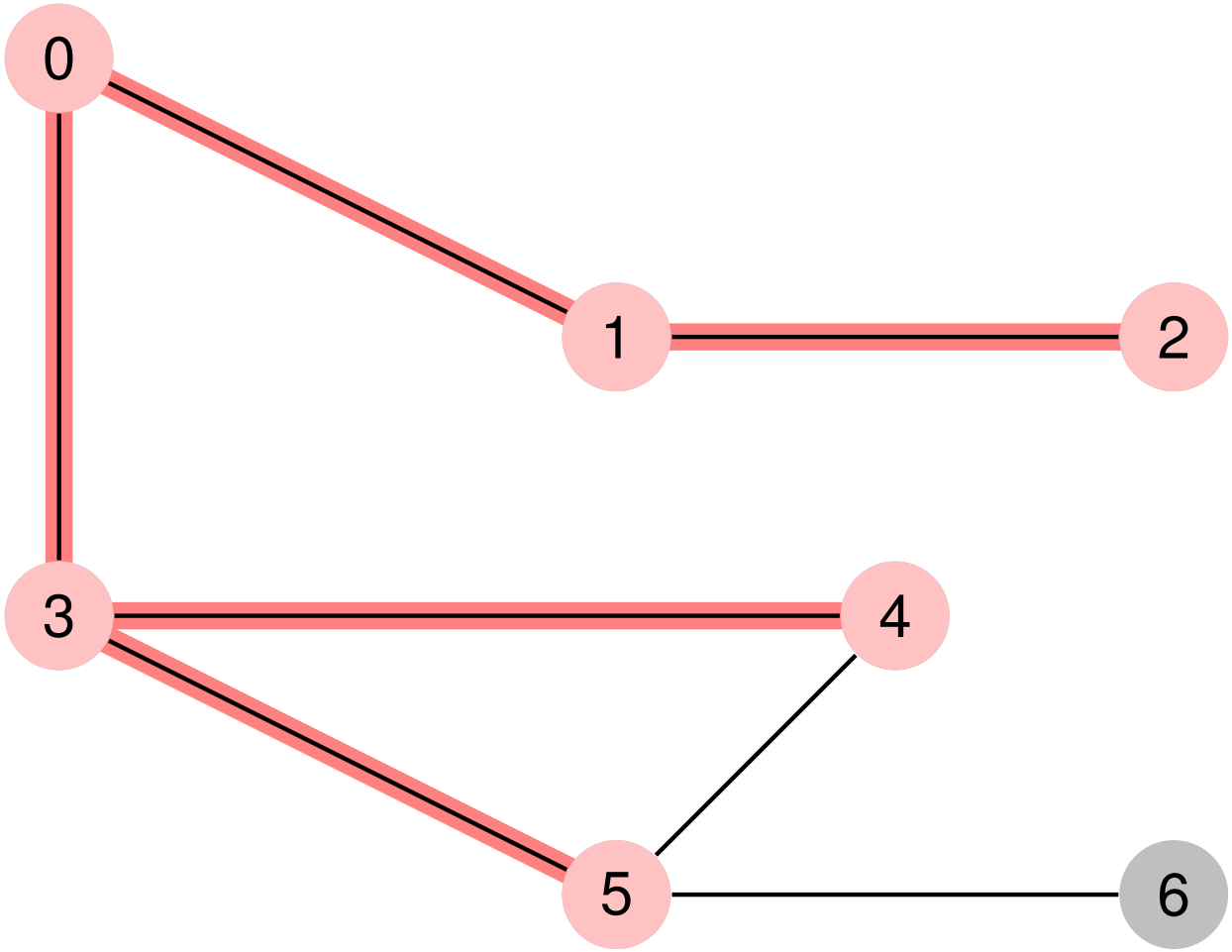
# Ejemplo de recorrido de un grafo usando BFS



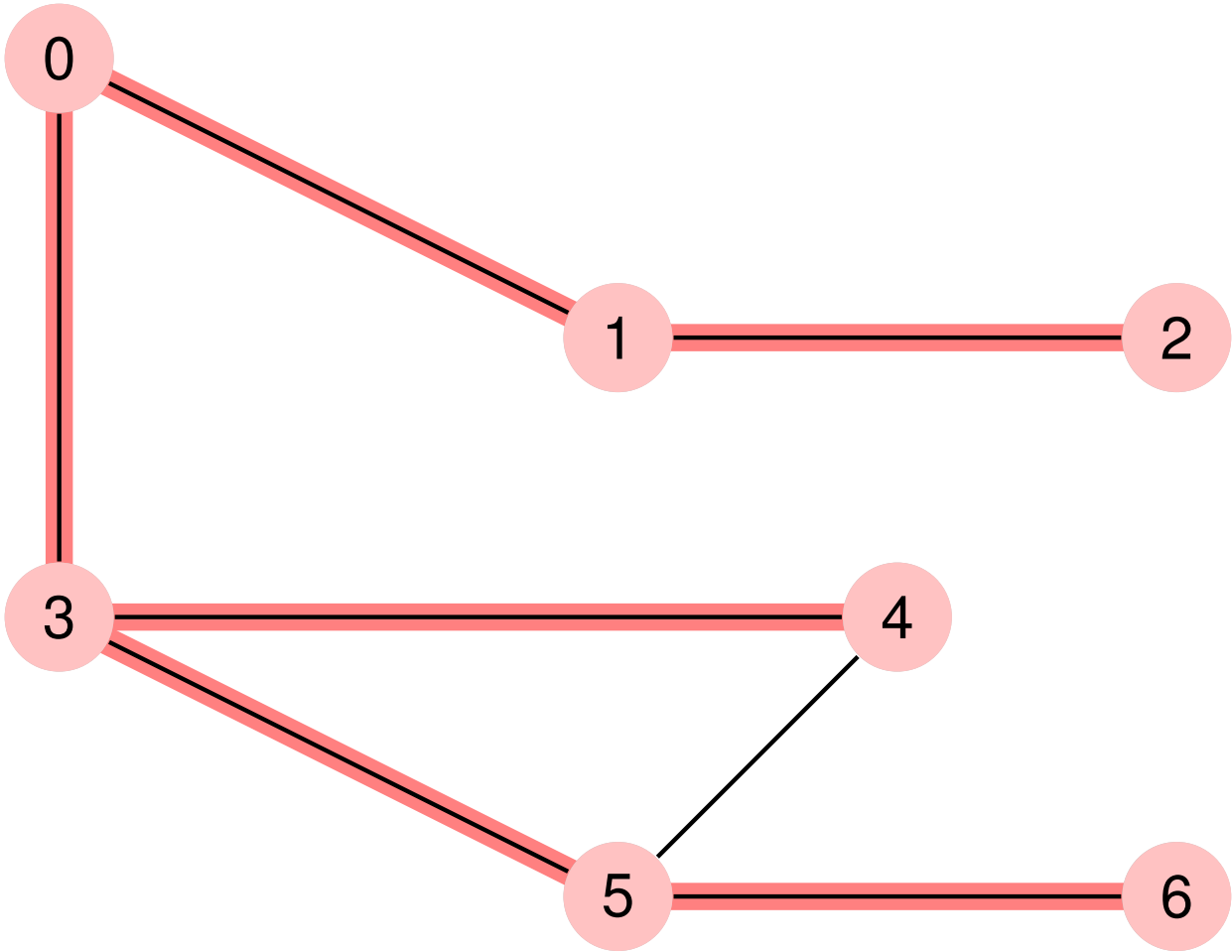
# Ejemplo de recorrido de un grafo usando BFS



# Ejemplo de recorrido de un grafo usando BFS



# Ejemplo de recorrido de un grafo usando BFS



Implementemos un BFS resolviendo un problema. Vale notar que este problema no podía ser resuelto con DFS.

### Calcular las distancias de un nodo a todos los demás

Dado un nodo inicial, queremos hallar la distancia de cada nodo a todos los demás. Recordemos que puede haber más de una forma de ir de un nodo a otro, pero para la distancia siempre tomaremos la mínima.

¿Cómo podemos resolverlo usando BFS? Por ahora no nos centremos en cómo se implementa el algoritmo, sino en cómo usarlo para resolver el problema. Luego intentaremos implementarlo.

# Detalles de implementación del BFS

- Las distancias del nodo inicial a los demás nodos las inicializaremos en infinito (en principio no sabemos a qué distancia están, luego iremos actualizando el valor si encontramos un camino desde el nodo inicial al nodo).

# Detalles de implementación del BFS

- Las distancias del nodo inicial a los demás nodos las inicializaremos en infinito (en principio no sabemos a qué distancia están, luego iremos actualizando el valor si encontramos un camino desde el nodo inicial al nodo).
- 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 infinito (en principio no sabemos a qué distancia están, luego iremos actualizando el valor si encontramos un camino desde el nodo inicial al nodo).
- 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.

# Pseudocódigo del BFS

```
1  vector<int> BFS (Grafo listaAdyacencias, int nodoInicial):
2      cantidadDeNodos = longitud(listaAdyacencias)
3      queue<int> cola // aqui encolare los nodos que todavia no analizamos
4
5      // distancias[i] = distancia de i a nodoInicial, inicialmente es INFINITO
6      vector<int> distancias(cantidadDeNodos, INFINITO)
7
8      cola.encolar(nodoInicial)
9      distancias[nodoInicial] = 0
10     mientras (la cola no esta vacia):
11         tope = cola.tope() // tomo el tope de la cola (y lo saco de la cola)
12
13         para todos los vecinos v de "tope" en el grafo:
14             si la distancia entre v y el nodoInicial es infinito:
15                 distancias[v] = distancias[tope] + 1 // esta un nodo mas
                       lejos que su vecino
16                 cola.encolar(v) // encolo v para analizarlo luego
17
18     devolver distancias
```

# 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 }
```

# Contenidos

# Problema sencillo para empezar

En una ciudad con forma de grilla hay $h$ calles horizontales y $v$ calles verticales. Además, algunas esquinas están obstaculizadas por diversos motivos, y no se puede pasar por allí.

Queremos partir desde nuestra casa y llegar a la facultad. ¿Será posible lograrlo?
Diseñemos un algoritmo eficiente para resolver este problema.

**Pista:** el problema se puede resolver con complejidad temporal $O(vh)$

# Contenidos

# Modelar un problema con un grafo no trivial

Muchas veces, modelar un problema con el grafo que el enunciado casi explícitamente nos describe no es suficiente para cumplir con los requerimientos de eficiencia que se nos piden (o a veces, ni siquiera sabríamos cómo resolverlo con ese grafo que nos dan).

Para eso, durante la clase de hoy intentaremos **pensar en grafos un poco más ricos de los que venimos viendo, con más parámetros**, que nos permitan resolver los problemas eficientemente.

Como suele suceder, esto es un trade-off: **al ganar en complejidad temporal, podemos perder en complejidad espacial**.

# En la altura, la bici no dobla

Melanie vive en la ciudad que mencionamos anteriormente: tiene forma de grilla, con $h$ calles horizontales y $v$ calles verticales. Además, algunas esquinas están obstaculizadas por diversos motivos, y no puede pasar por allí.

Recientemente Melanie se compró una bicicleta y ya aprendió a andar sin rueditas, pero doblar todavía le cuesta. Le gustaría poder ir en bici de su casa a la facultad minimizando la cantidad de veces que tiene que doblar en una esquina. ¿Cómo la podemos ayudar?

# Spoilers - idea 1

Crear un grafo donde cada nodo describa una esquina y la dirección de la que provenimos con la bici. Así, **el nodo será una tupla (fila, columna, dirección)**, y se conectarán dos nodos si y sólo si de un estado puedo pasar al otro. Además, si para pasar de un estado a otro debo doblar en una esquina, la arista tendrá costo 1. De lo contrario, tendrá costo 0.

Luego, tendremos un grafo con aristas de peso 0 ó 1, y queremos saber cuál es el camino de costo mínimo desde la esquina inicial hasta la final. O(vh lg(vh)) si usamos Dijkstra con cola de prioridad.

# Spoilers - idea 2

Ahora cada nodo además contendrá cuántos giros fue necesario realizar para llegar a cada esquina. Así, **el nodo será una tupla (fila, columna, cantidad de giros usados hasta el momento, dirección)**, y se conectarán dos nodos si y sólo si de un estado puedo pasar al otro.

Notar que este grafo no tendrá pesos, y lo que queremos preguntar es si será posible llegar a algún nodo final cualquiera sea la cantidad de giros y la dirección del mismo. Para ello será necesario recorrer el grafo (por ejemplo usando BFS) y ver si el nodo salida y alguna de las llegadas se encuentran en la misma componente conexa. Linealmente recorremos todos los posibles nodos llegada, y retornamos aquel cuya cantidad de giros necesarios sea mínima.

# Spoilers - idea 2 (cálculo de complejidad)

Llamemos $g$ a la cantidad máxima de giros posibles a realizar en el tablero. Así, como BFS es lineal en el tamaño de la entrada, y tenemos $v \times h \times g \times 2$ nodos en nuestro modelado (y cada nodo tiene grado a lo sumo 4), la recorrida del grafo será O(vhg).

Al final recorremos linealmente todos los posibles nodos finales, que como tienen fija la esquina en la que terminan son solamente $2g$ nodos. Así, la complejidad total del algoritmo es
$O(vhg) + O(2g) = O(vhg)$

# Simplificación del F de la WF 2015

Tenemos un teclado con forma de una grilla de $a$ filas y $b$ columnas donde cada casilla contiene una letra.

Queremos escribir una palabra $p$, pero queremos hacerlo gastando la mínima cantidad de energía. Estando nuestro dedo situado en una casilla, el mismo sólo puede moverse a una de las 4 casillas vecinas y pagará una unidad de energía por hacerlo.

Determinar cuál es la menor cantidad de energía que es necesario gastar para escribir la palabra. Puede haber más de una casilla con la misma letra y en el teclado se encuentran todas las letras necesarias para escribirla.

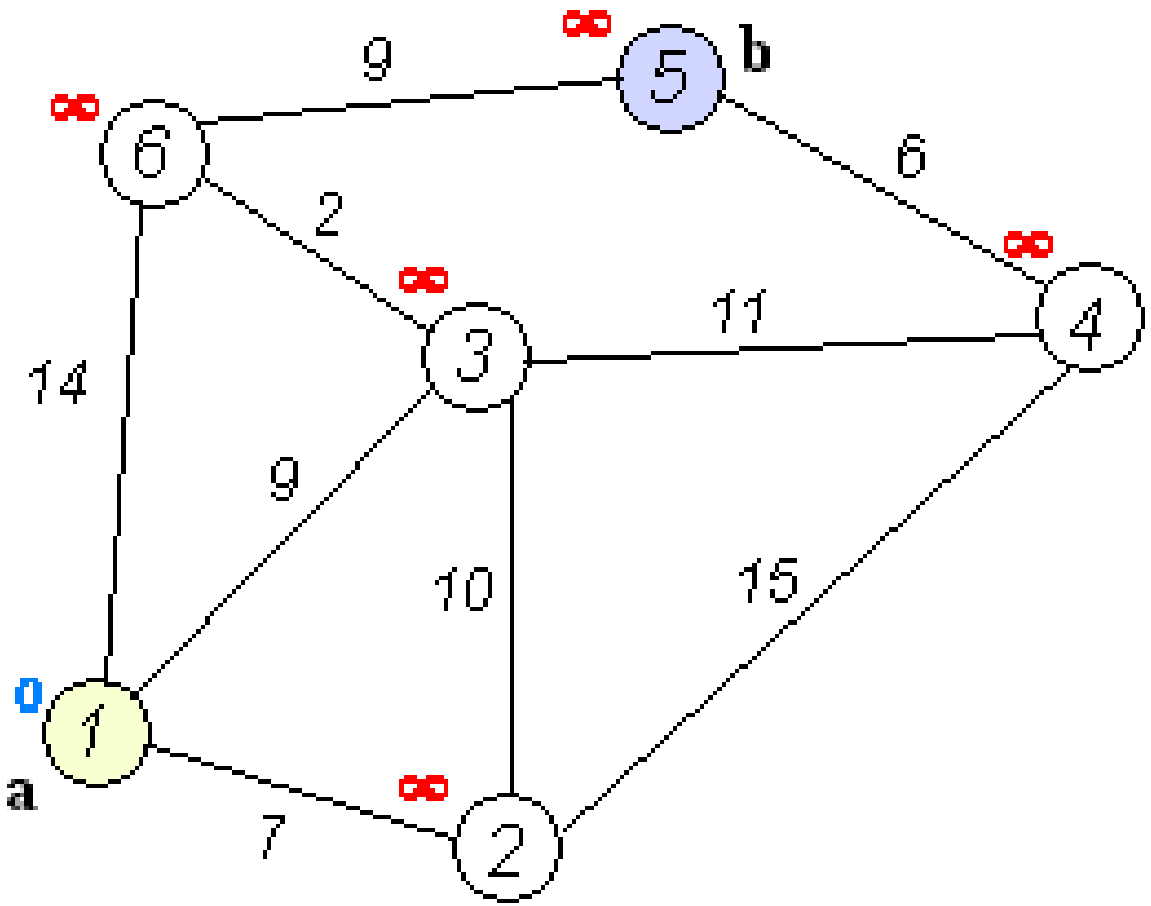
**Pista:** el problema se puede resolver con complejidad temporal $O(ab|p|)$.

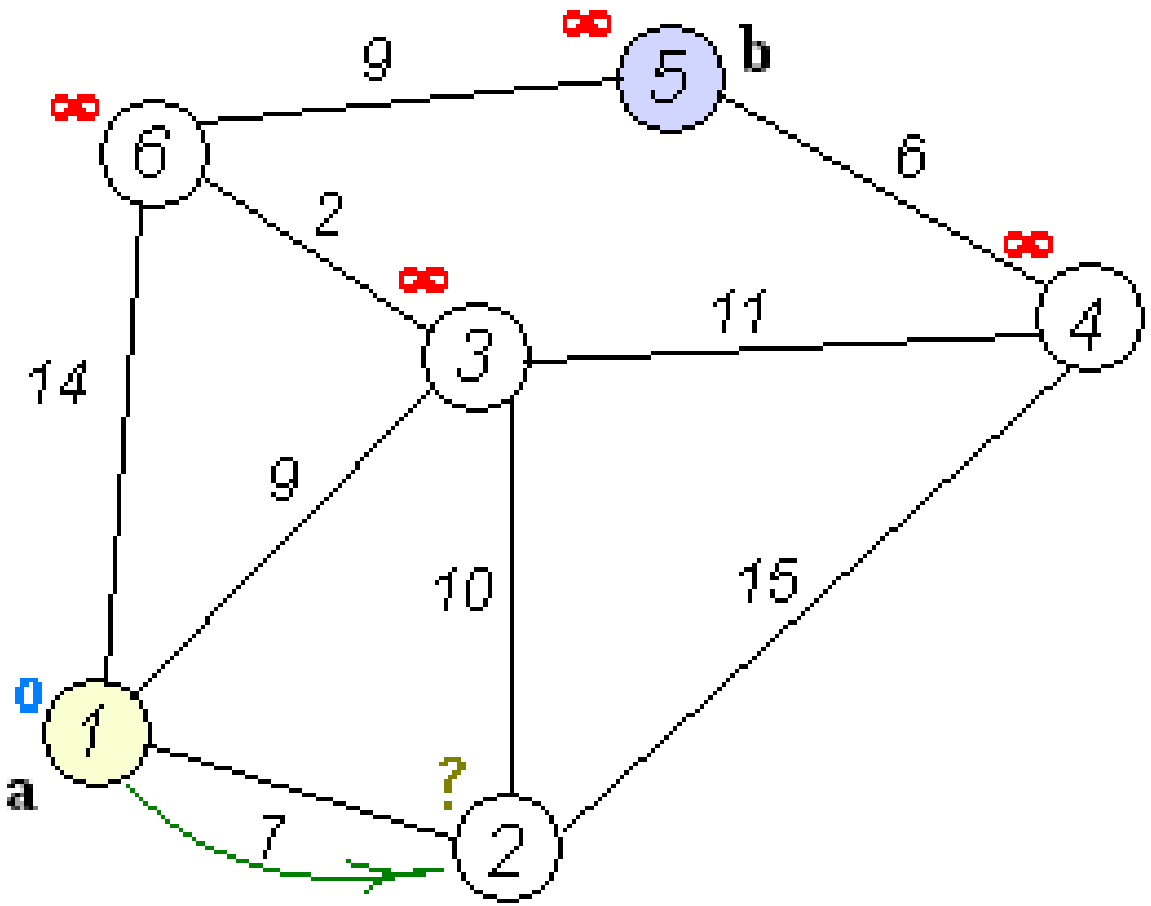
# Camino mínimo - algoritmo de Dijkstra

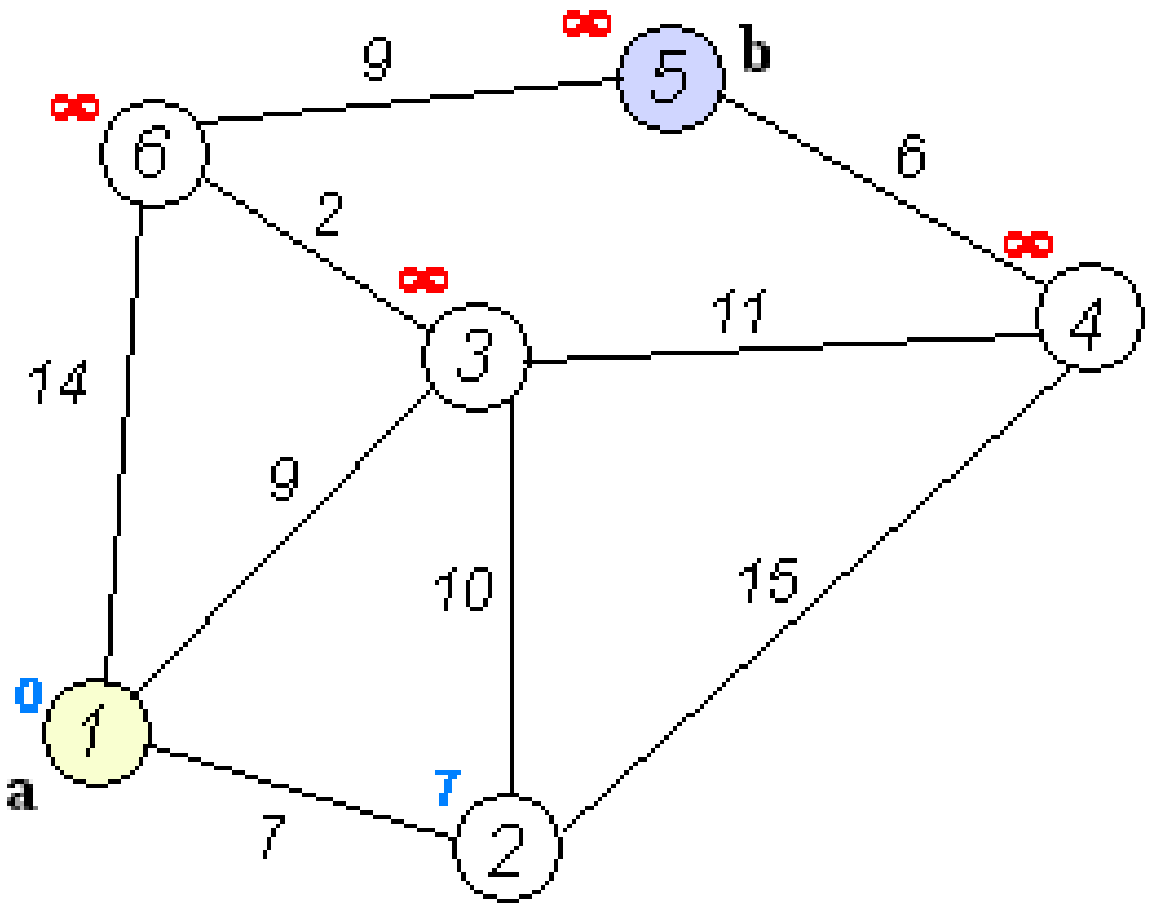
Como todos saben, la forma más conocida de hallar el camino mínimo de un nodo a todos los demás es utilizando el algoritmo de Dijkstra. Sólo se puede utilizar si todos los pesos de las aristas son no negativos.

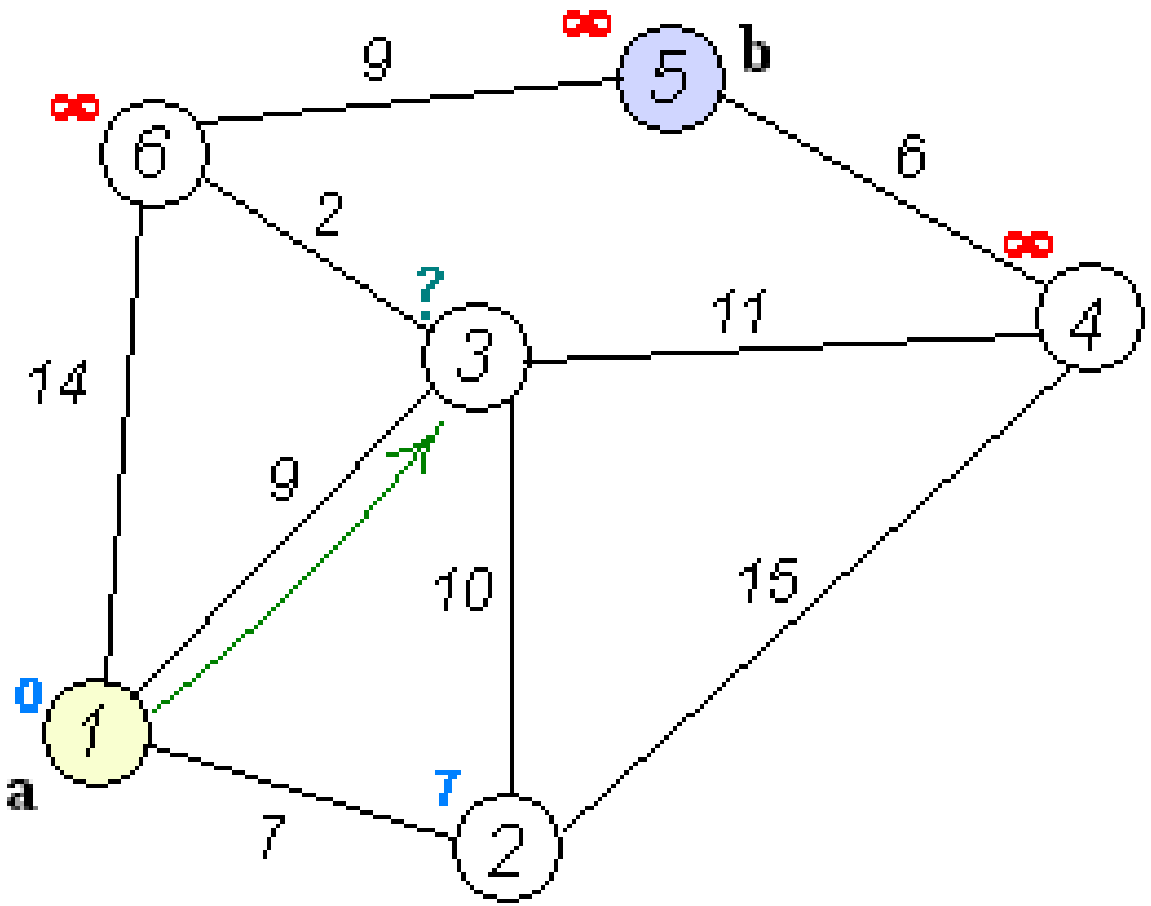
Primero recordaremos cómo es el algoritmo de Dijkstra en abstracto, luego en sus dos implementaciones más famosas y finalmente resolveremos un problema relacionado.

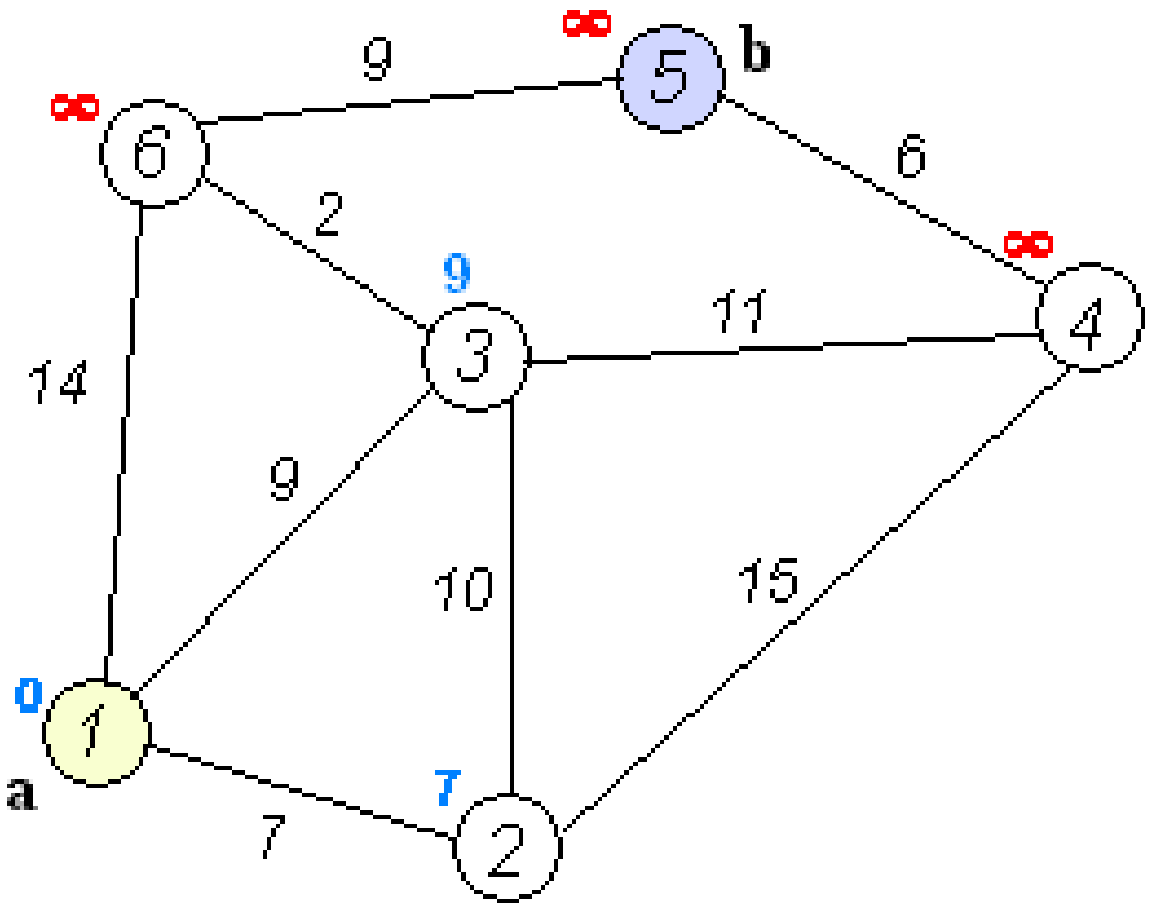
Veamos gráficamente cómo se ejecuta el algoritmo.

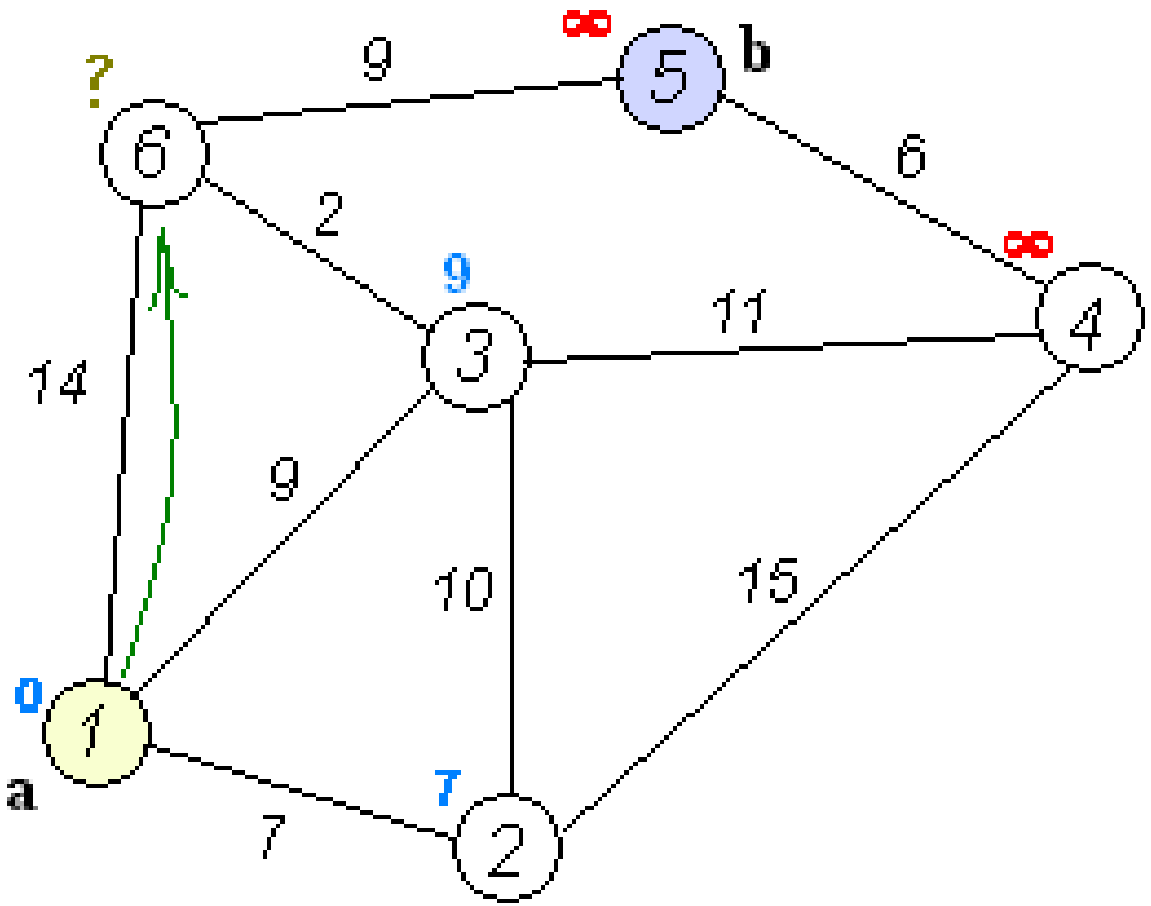

∞
9
5
b
∞
6
6
2
∞
∞
11
4
3
14
9
10
15
0
1
a
∞
7
2

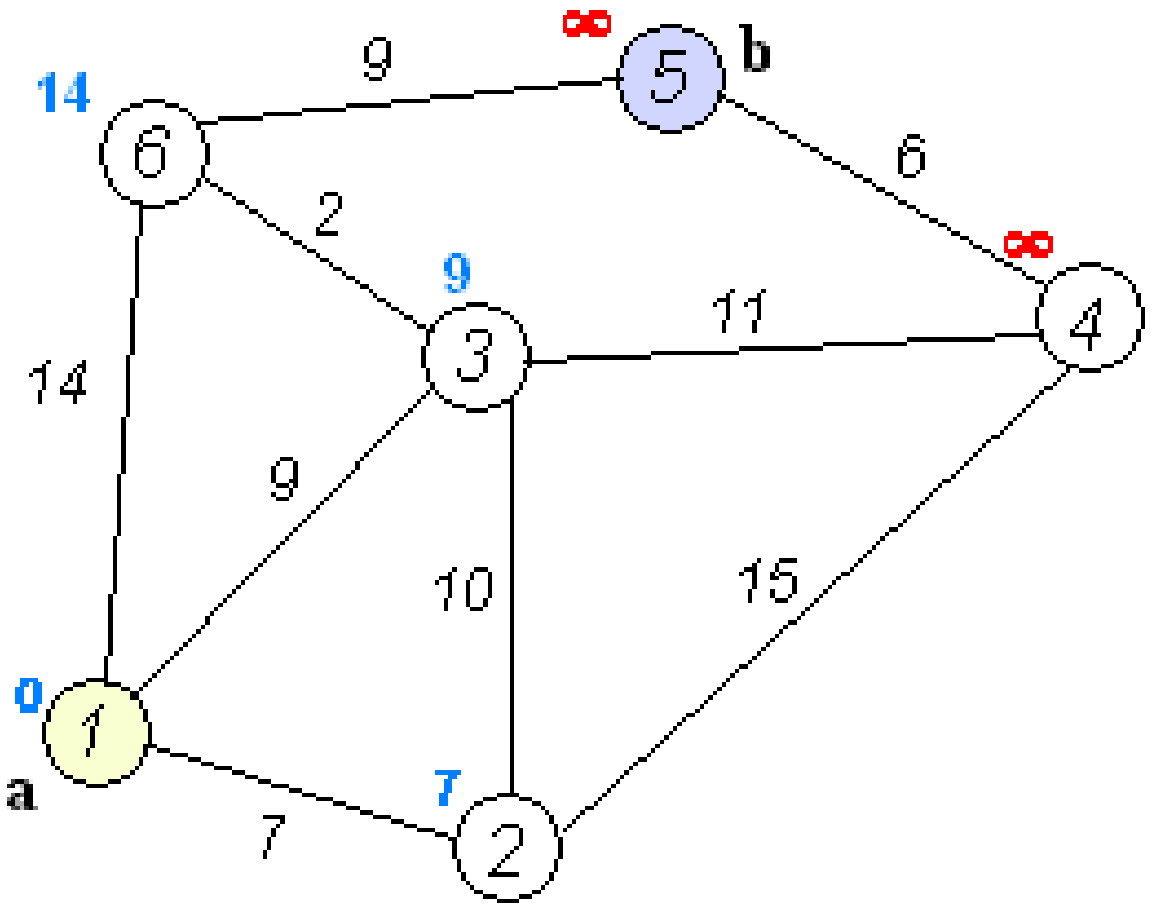

∞
9
5
b
∞
6
6
2
∞
∞
4
11
3
14
9
10
15
0
1
a
?
7
2

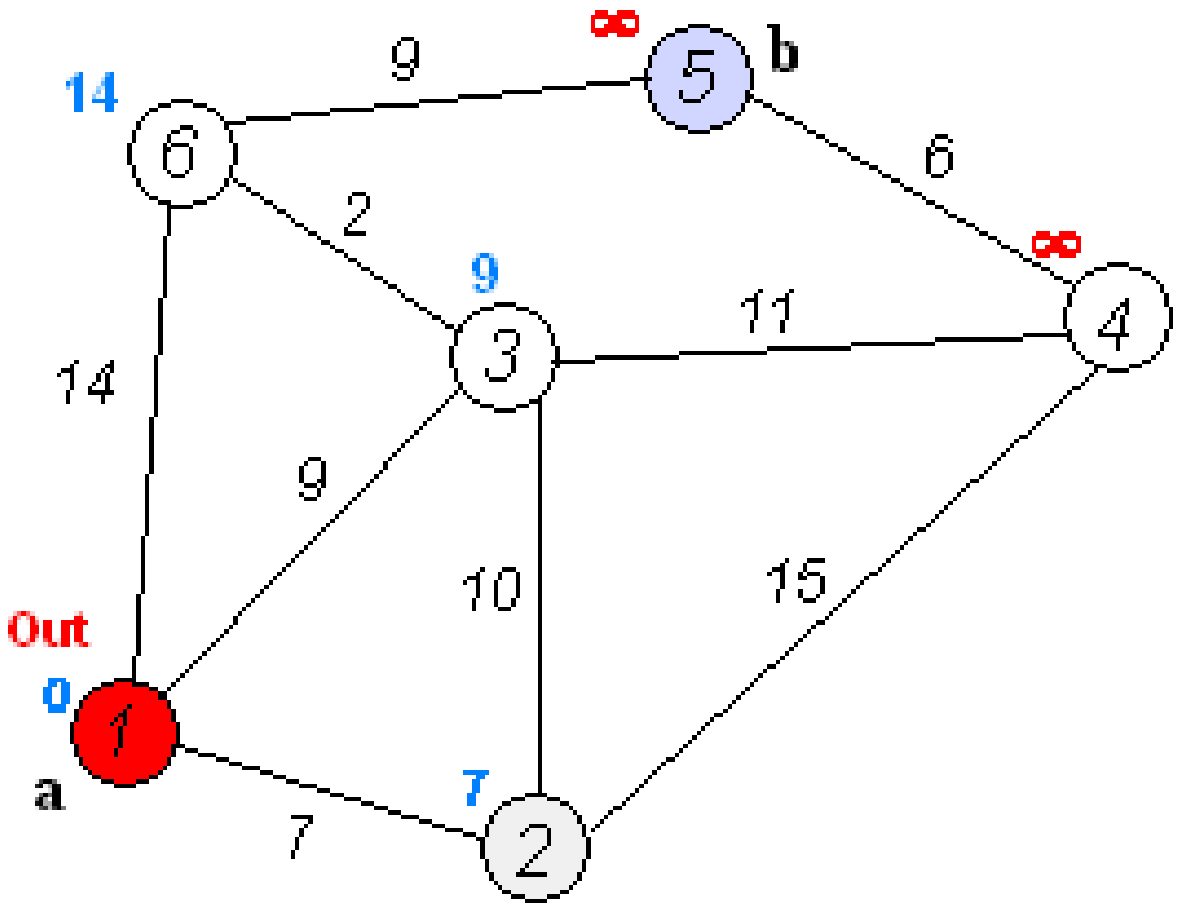

∞
9
5
b
∞
6
6
2
∞
∞
11
4
3
14
9
10
15
0
1
a
7
7
2

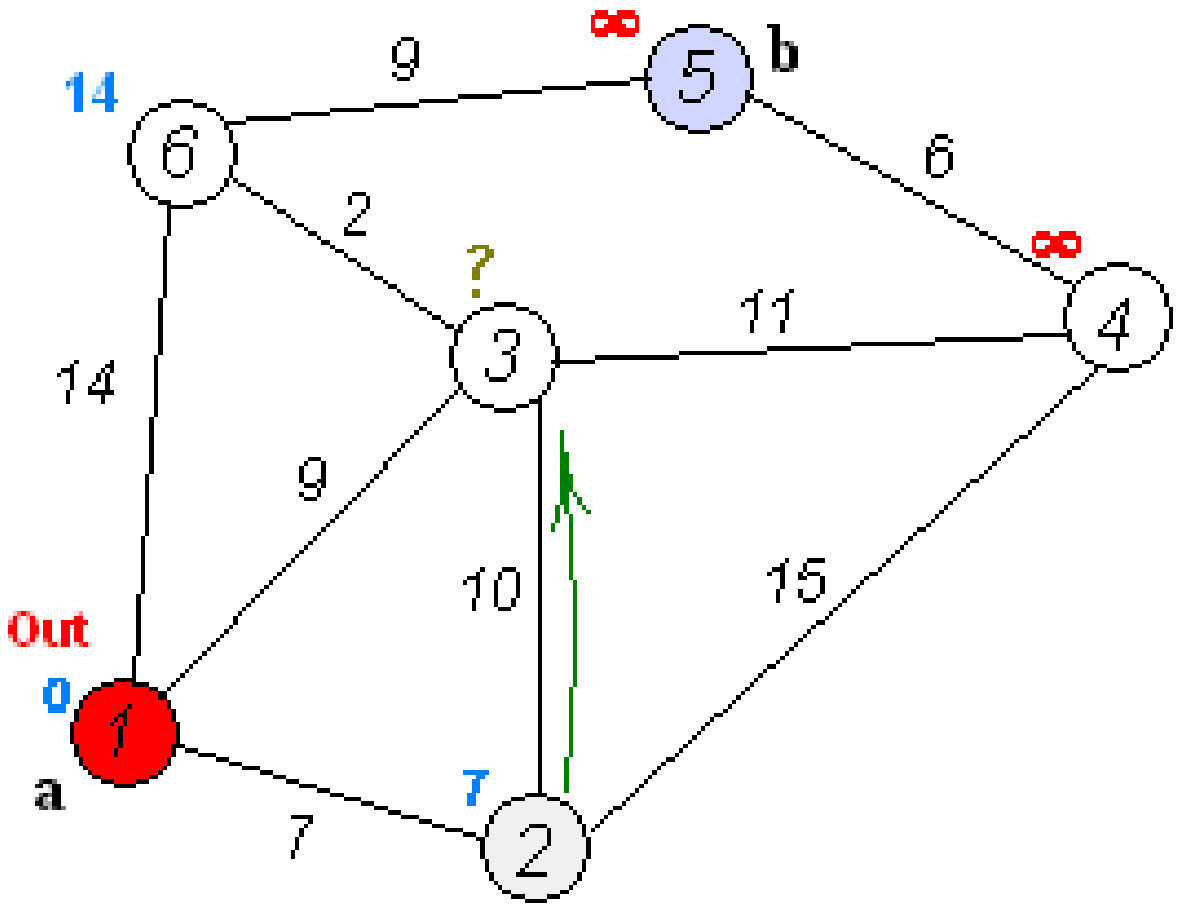

∞
9
5
b
∞
6
6
2
∞
?
11
4
3
14
9
10
15
0
1
a
7
7
2

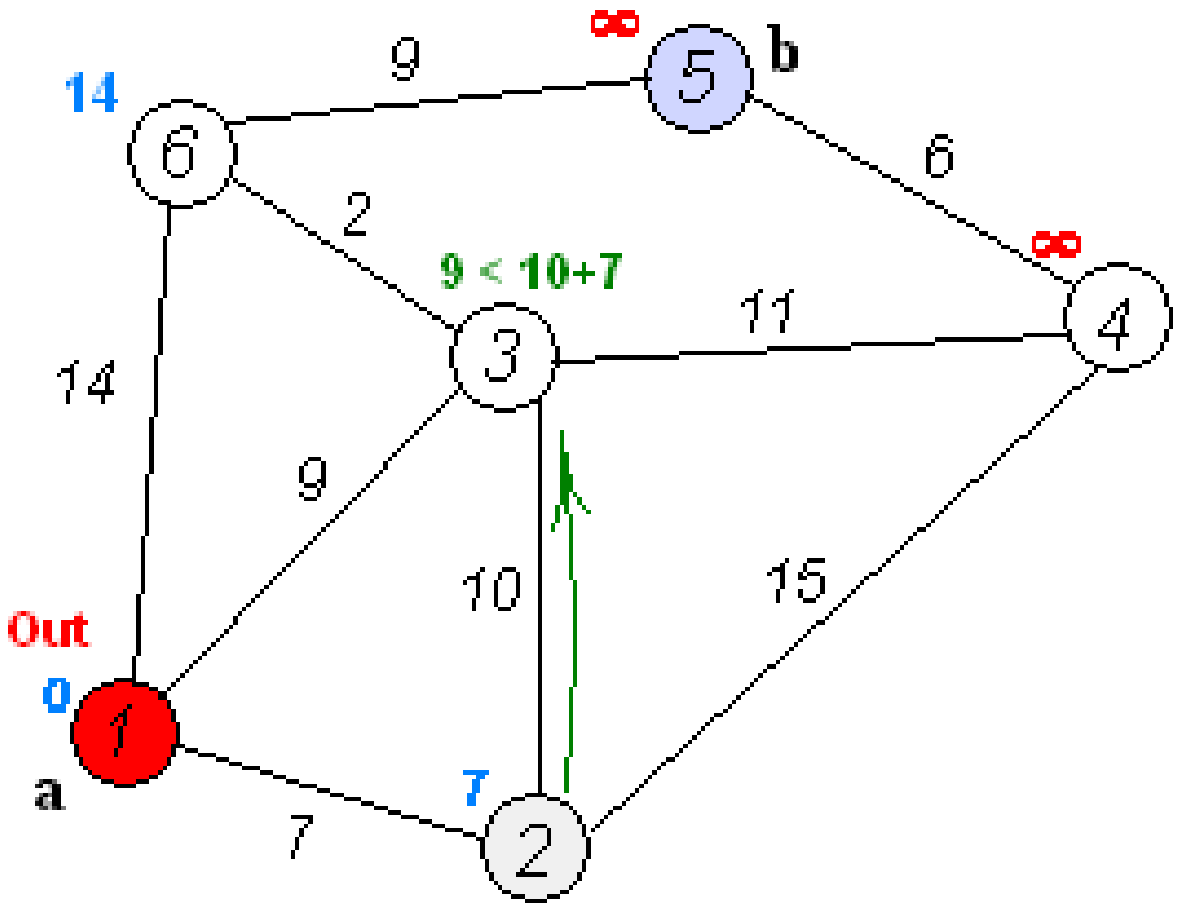

∞
9
5
b
∞
6
6
2
∞
9
4
11
3
14
9
10
15
0
1
a
7
7
2

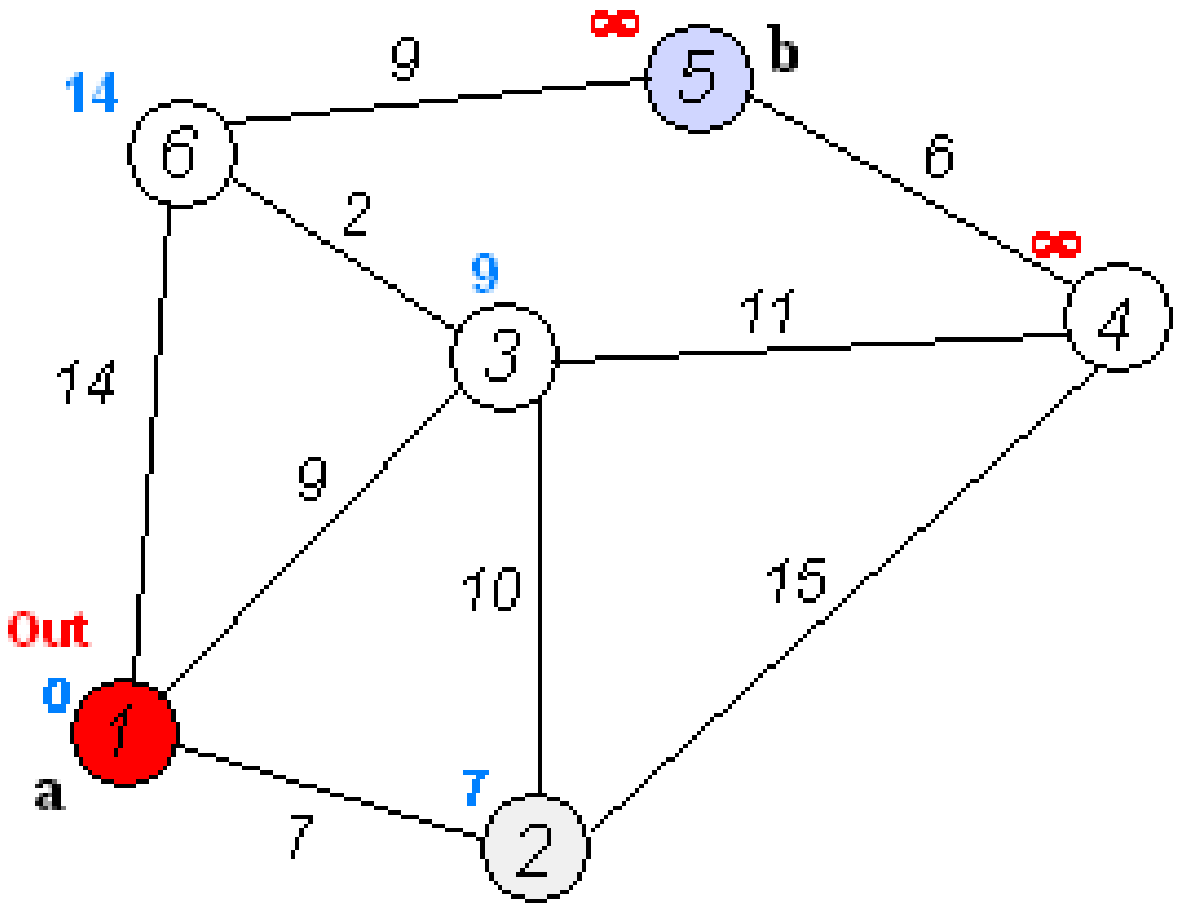

∞
b
5
9
?
6
6
2
∞
9
4
11
3
14
9
10
15
0
1
a
7
7
2

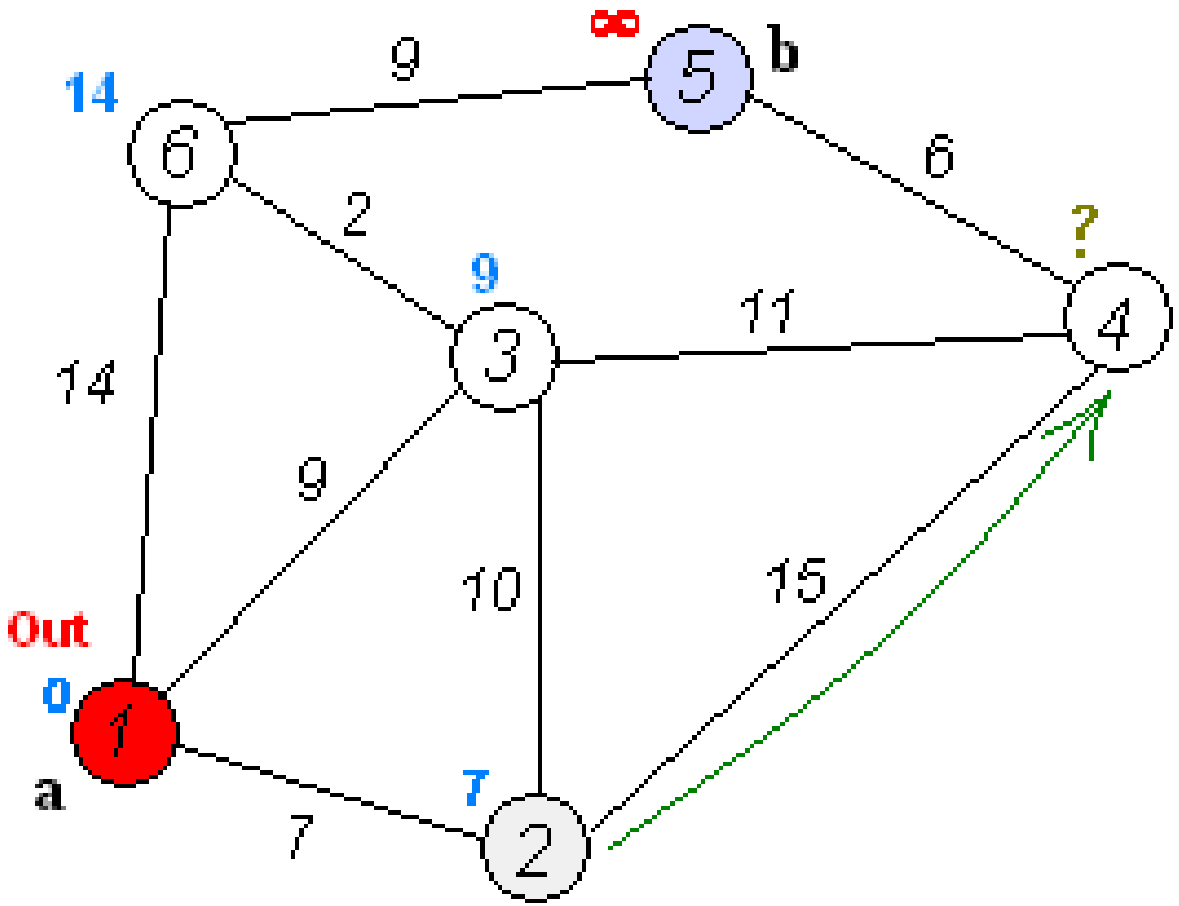

∞
b
5
9
14
6
6
2
∞
9
4
11
3
14
9
10
15
0
1
a
7
7
2

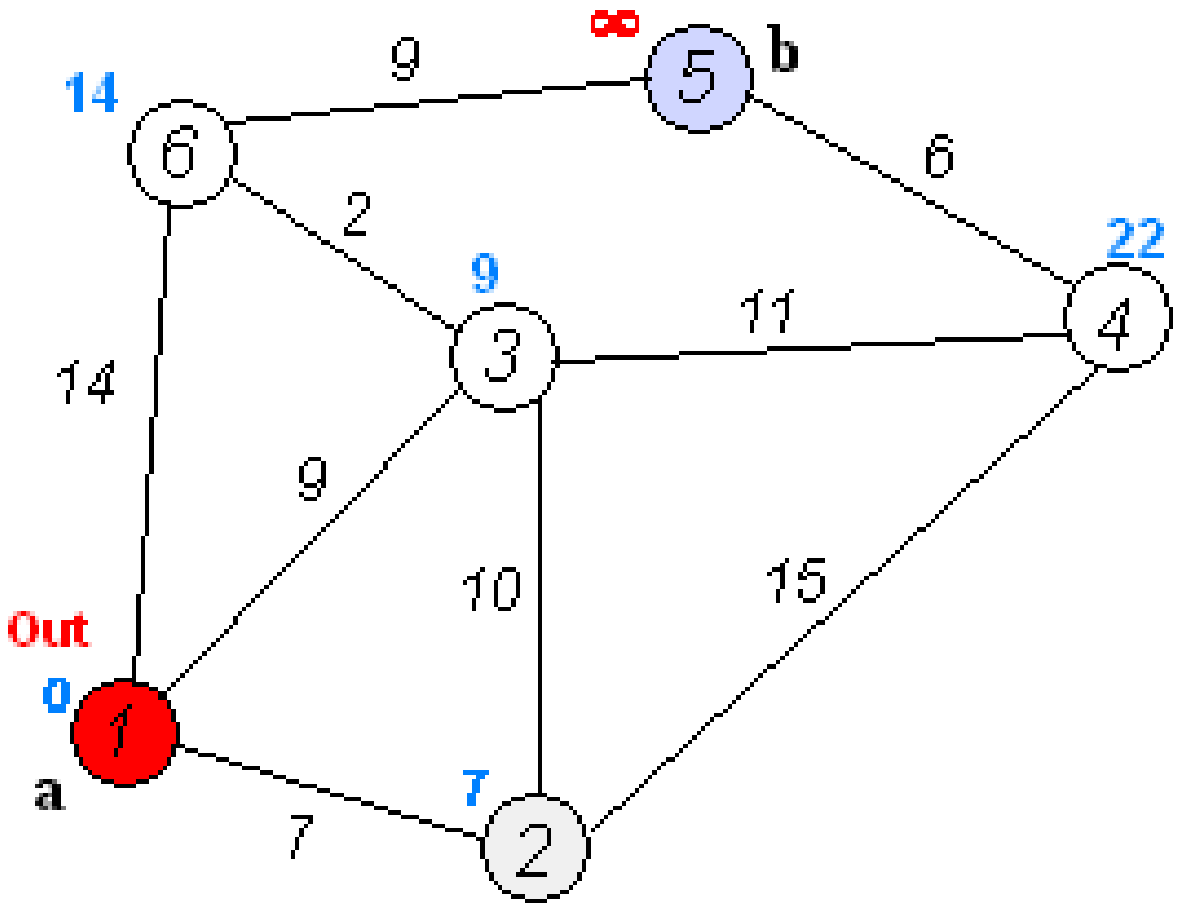

∞
b
5
9
14
6
6
2
∞
9
4
11
3
14
9
10
15
Out
0
1
a
7
7
2

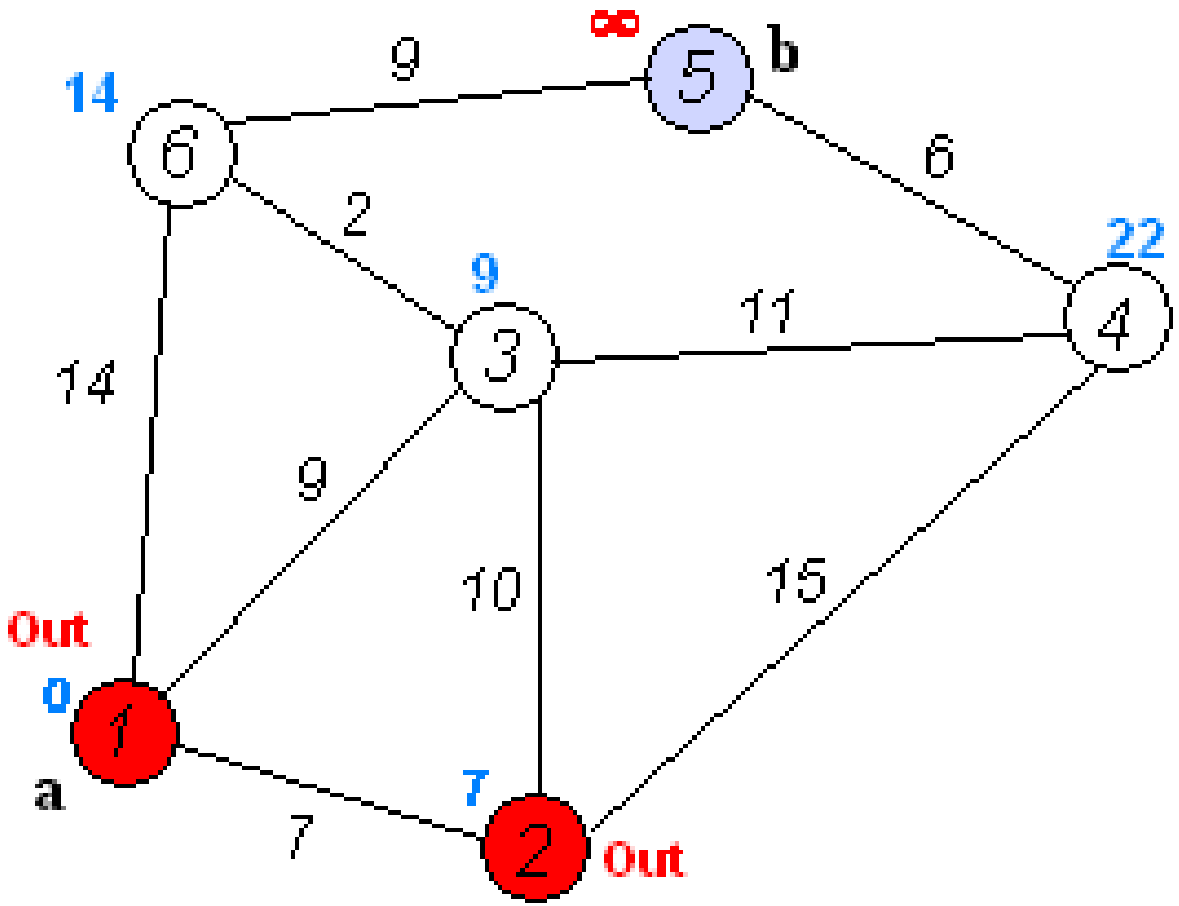

∞
b
5
9
14
6
6
2
∞
?
4
3
11
14
9
10
15
Out
0
1
a
7
7
2

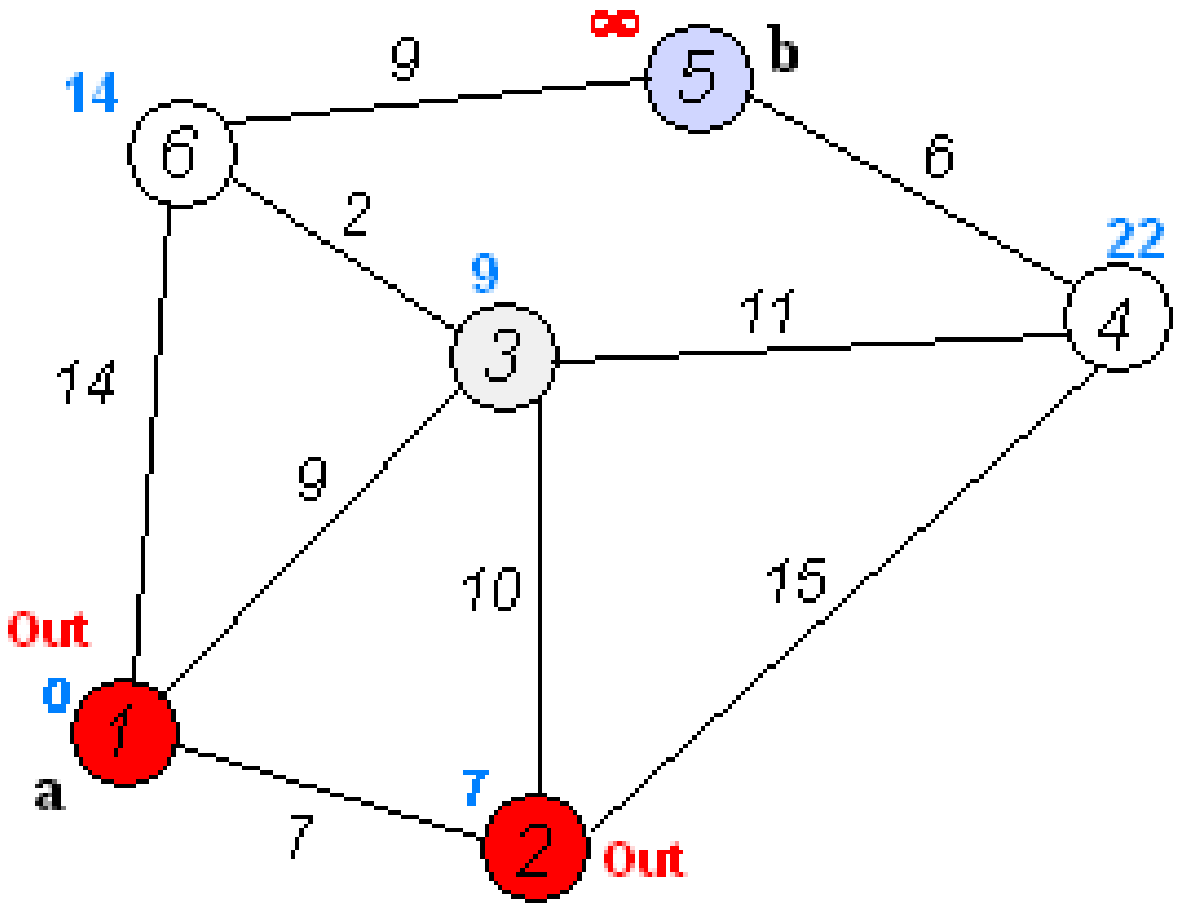

∞
b
5
9
14
6
6
2
∞
9 < 10+7
4
3
11
14
9
10
15
Out
0
1
a
7
7
2

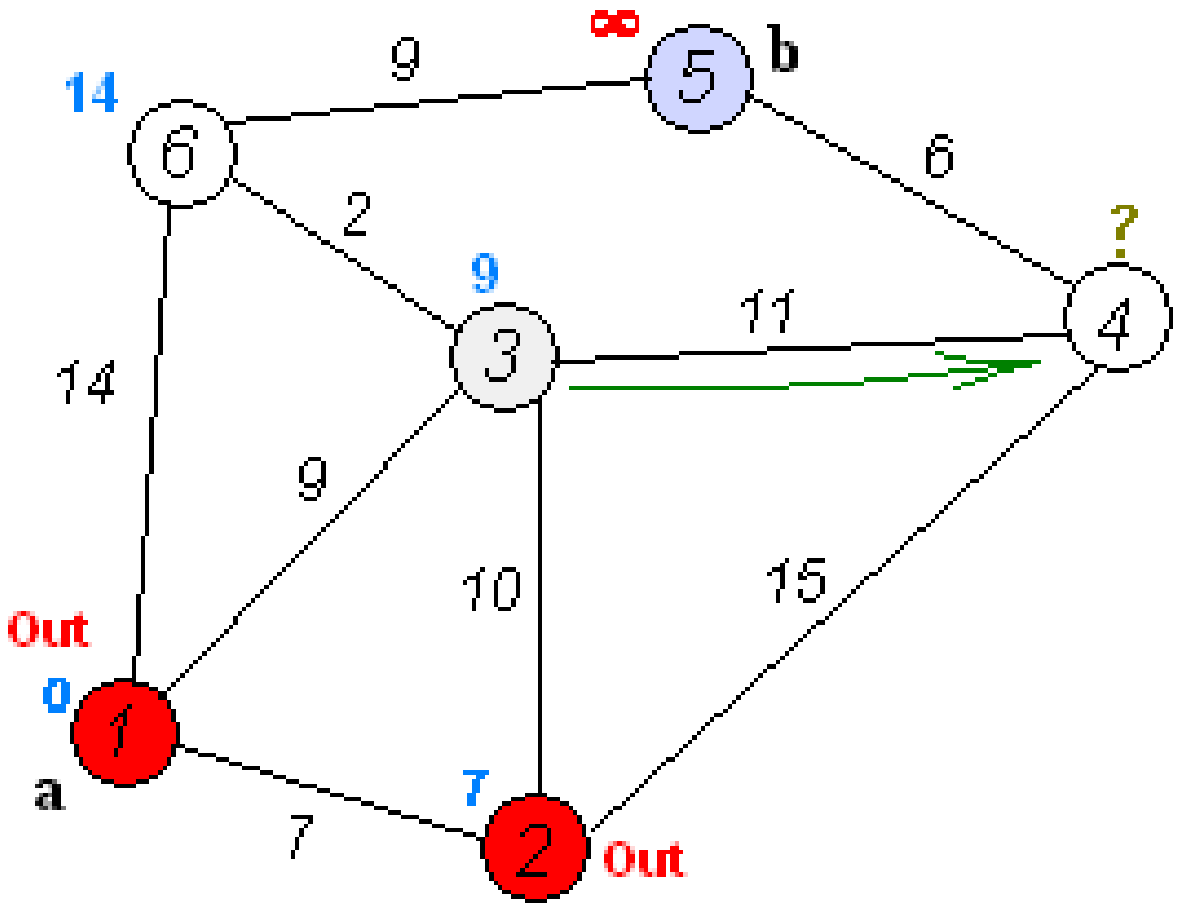

∞
b
5
9
14
6
6
2
∞
9
4
11
3
14
9
10
15
Out
0
1
a
7
7
2

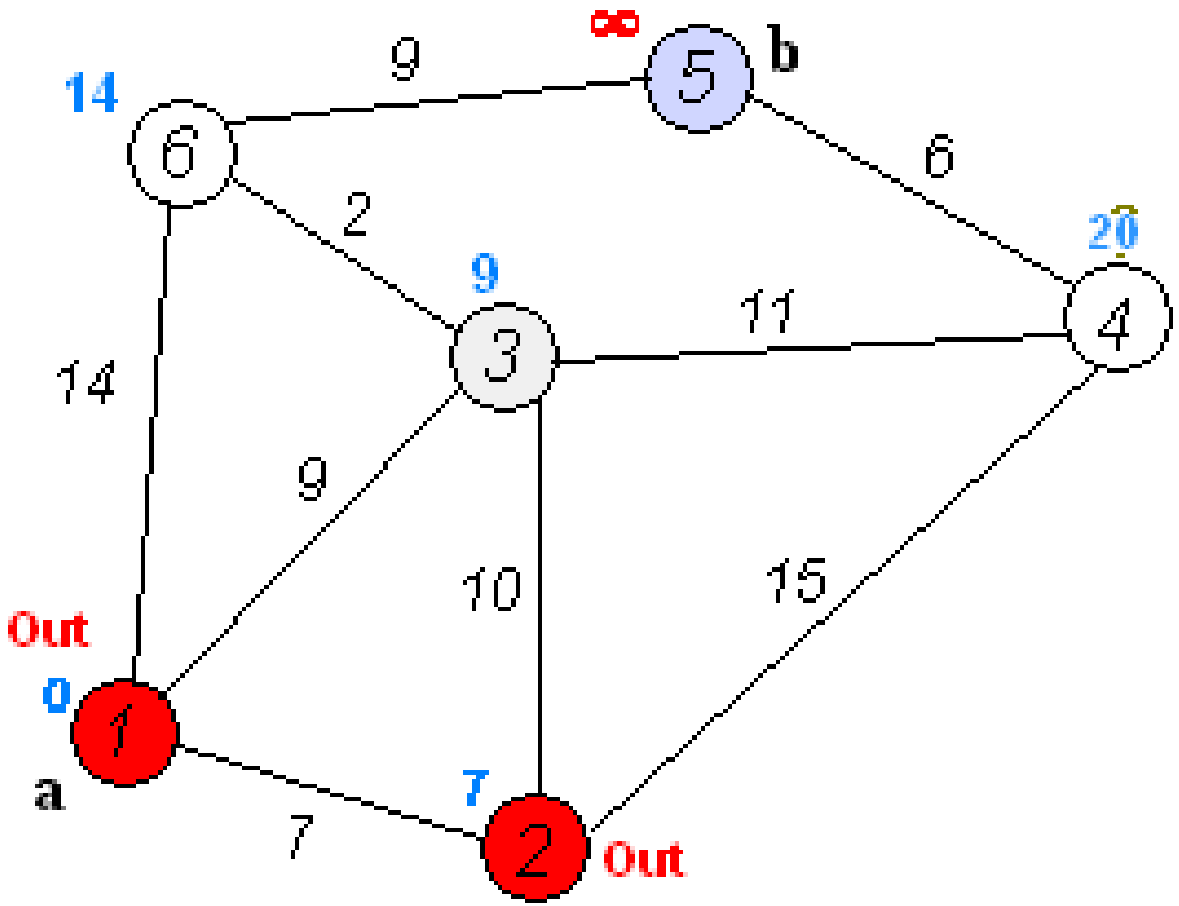

∞
b
5
9
14
6
6
2
?
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2

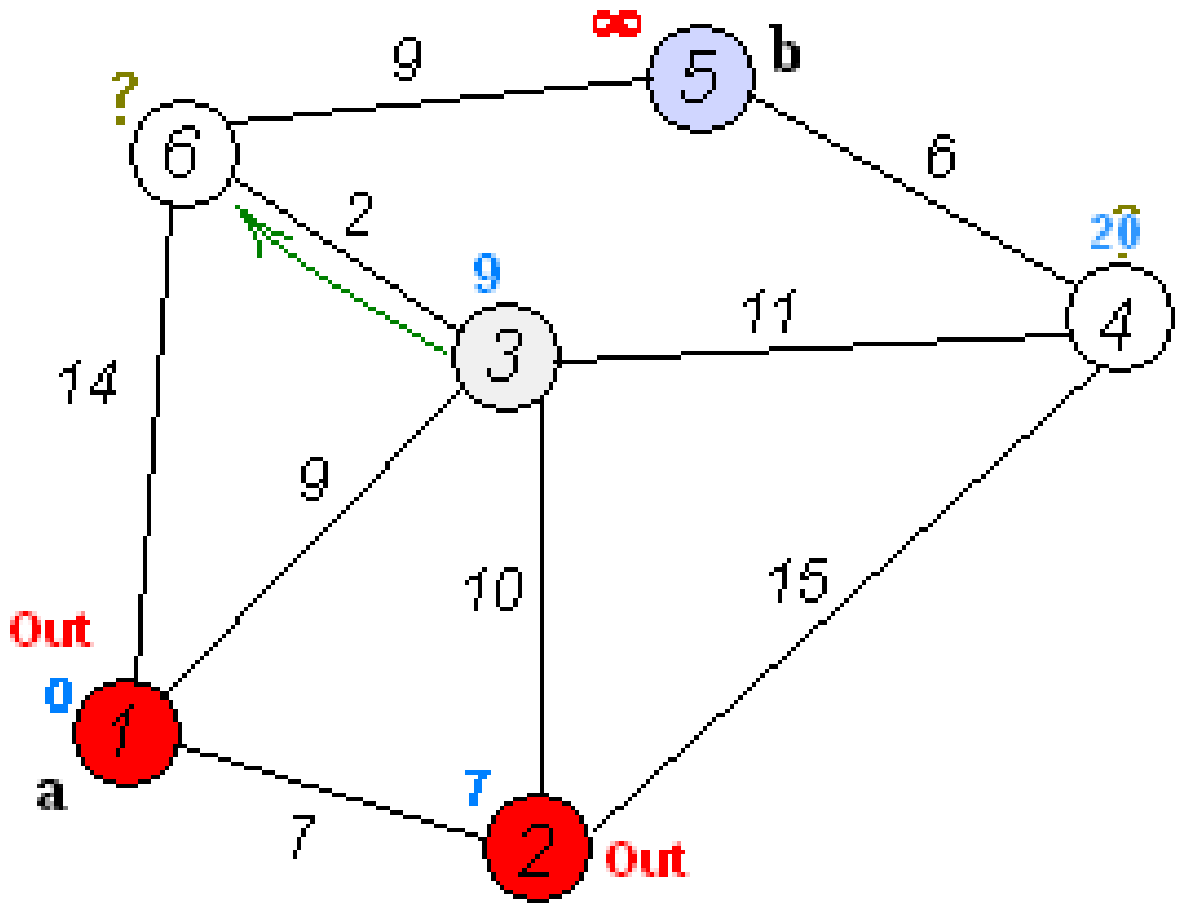

∞
b
5
9
14
6
6
2
22
9
11
3
4
14
9
10
15
Out
0
1
a
7
7
2

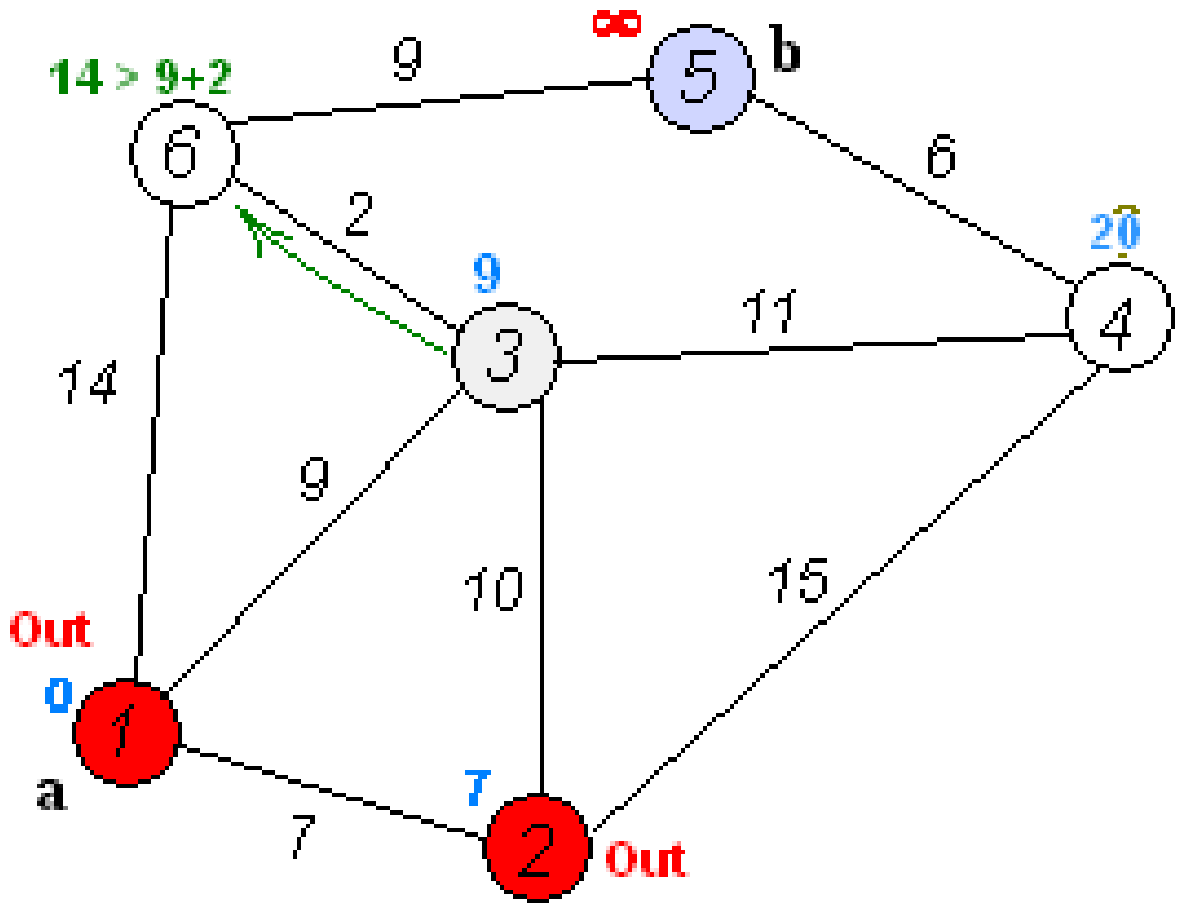

∞
b
5
9
14
6
6
2
22
9
11
3
4
14
9
10
15
Out
0
1
a
7
7
2
Out

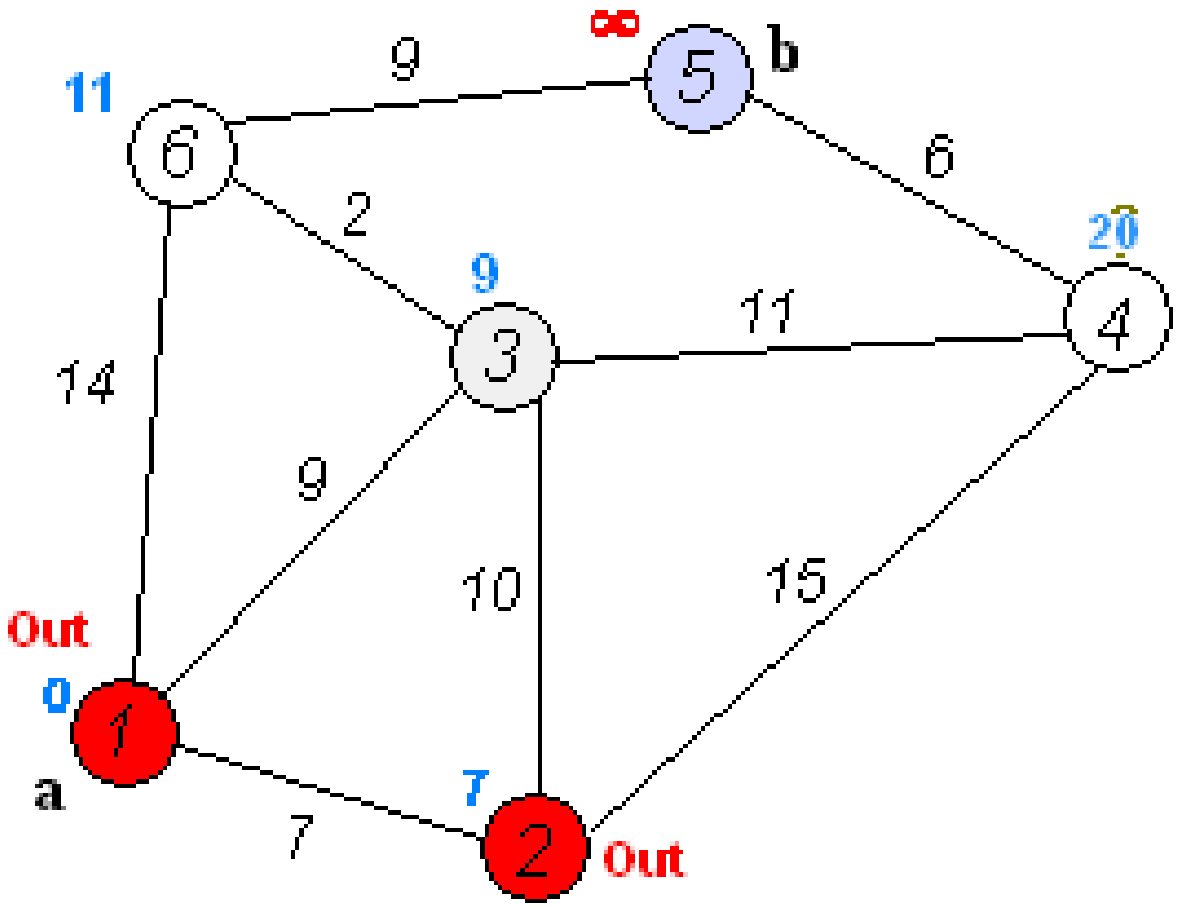

∞
b
5
9
14
6
6
2
22
9
11
3
4
14
9
10
15
Out
0
1
a
7
7
2
Out

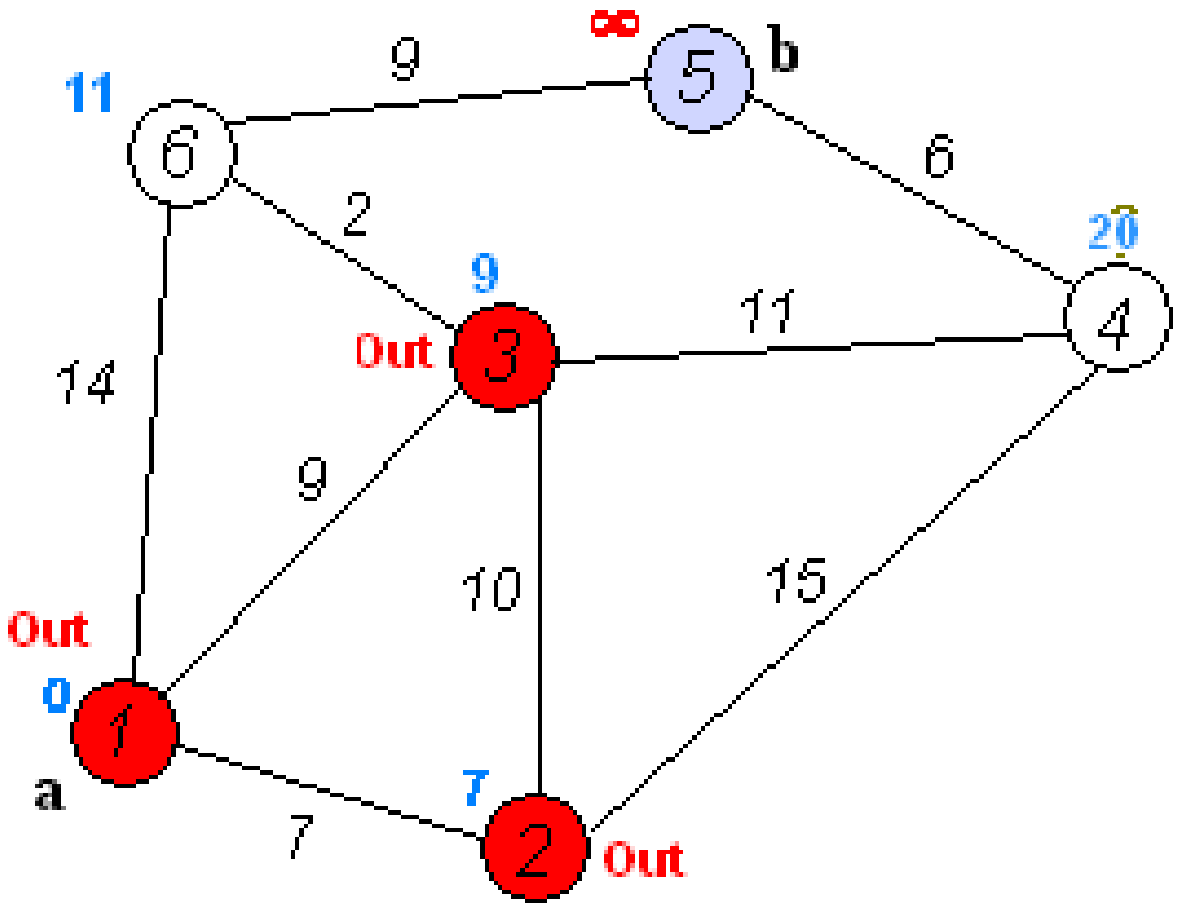

∞
b
5
9
14
6
6
2
?
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out

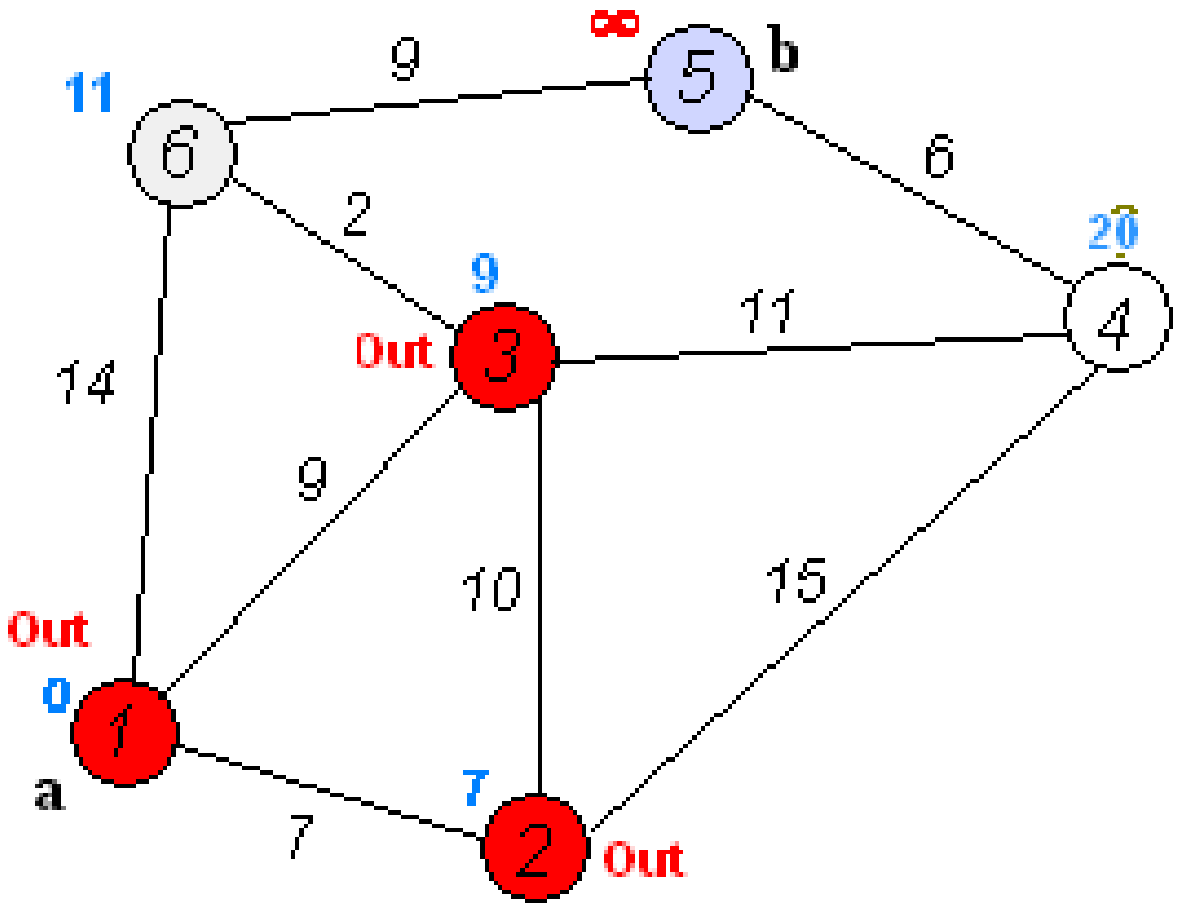

∞
b
5
9
14
6
6
2
20
9
11
3
4
14
9
10
15
Out
0
1
a
7
7
2
Out

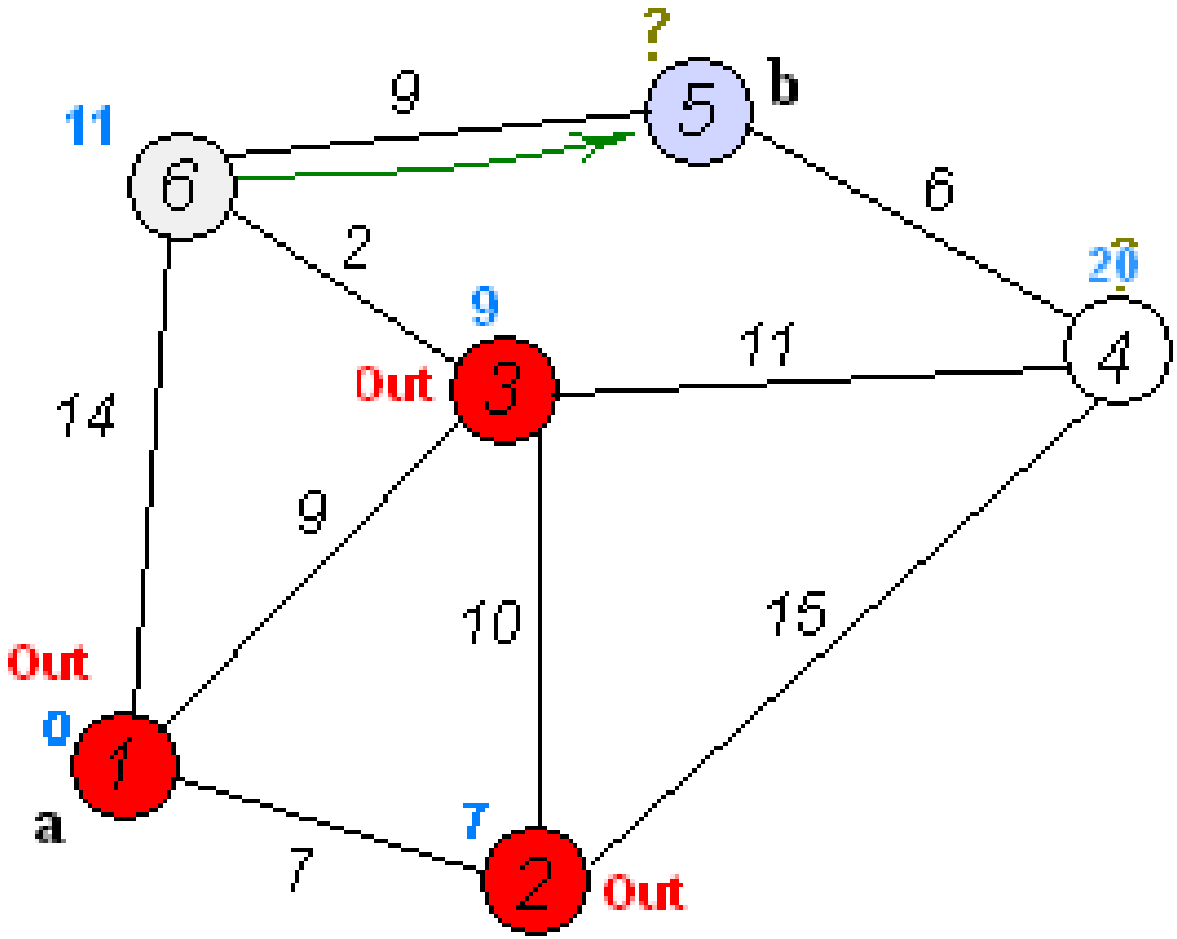

∞
b
5
9
?
6
6
2
20
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out

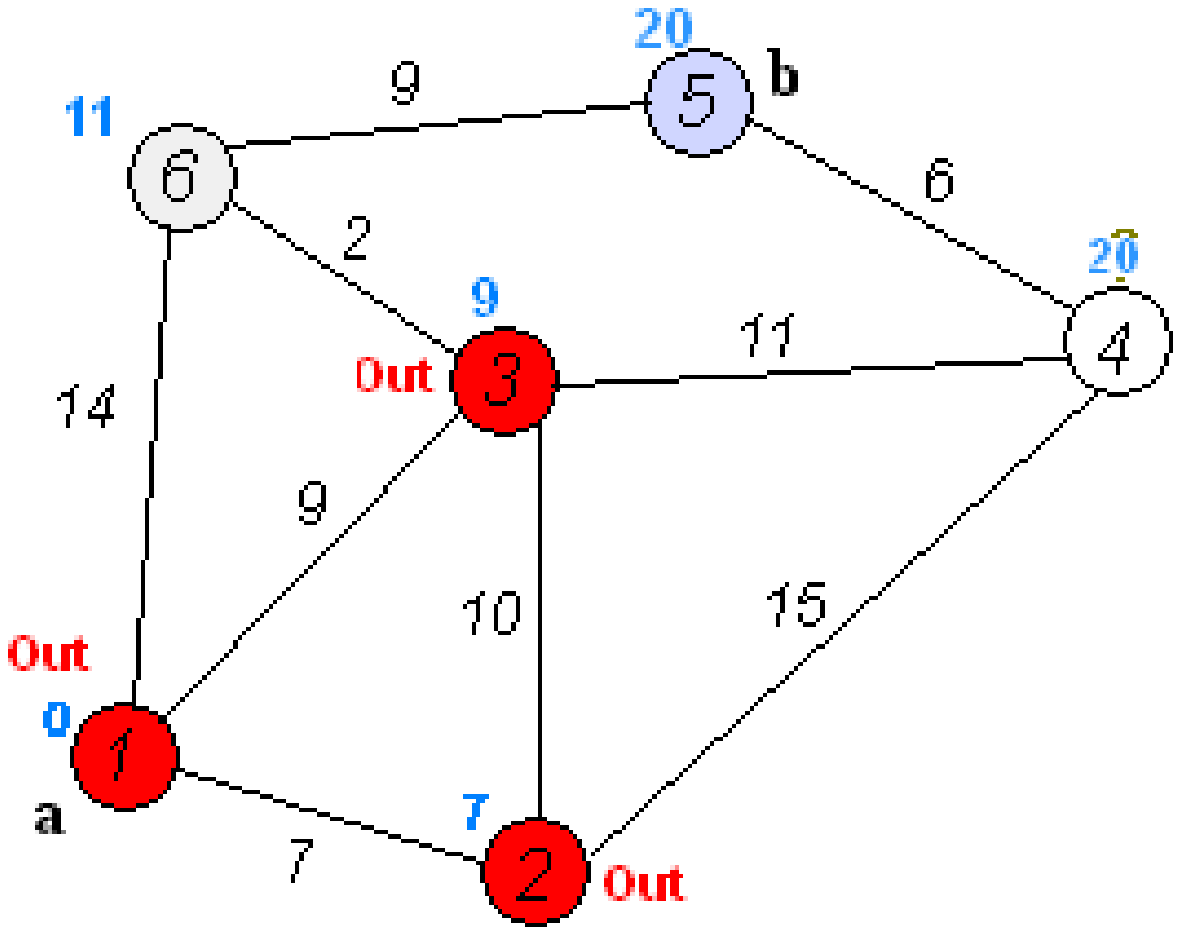

14 > 9+2
9
∞
5
b
6
6
2
20
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out

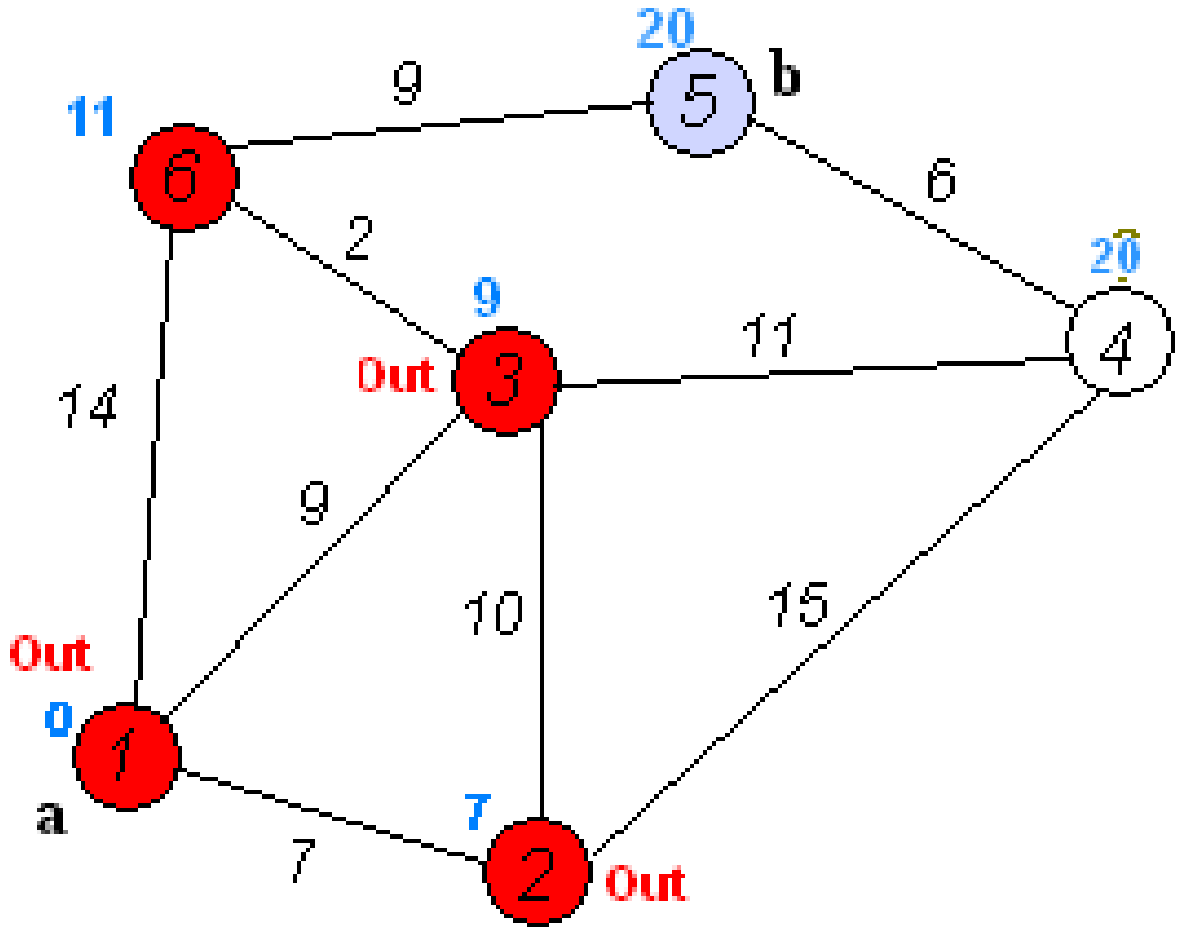

∞
b
5
9
11
6
6
2
20
9
11
4
3
14
9
10
15
Out
0
1
a
7
7
2
Out

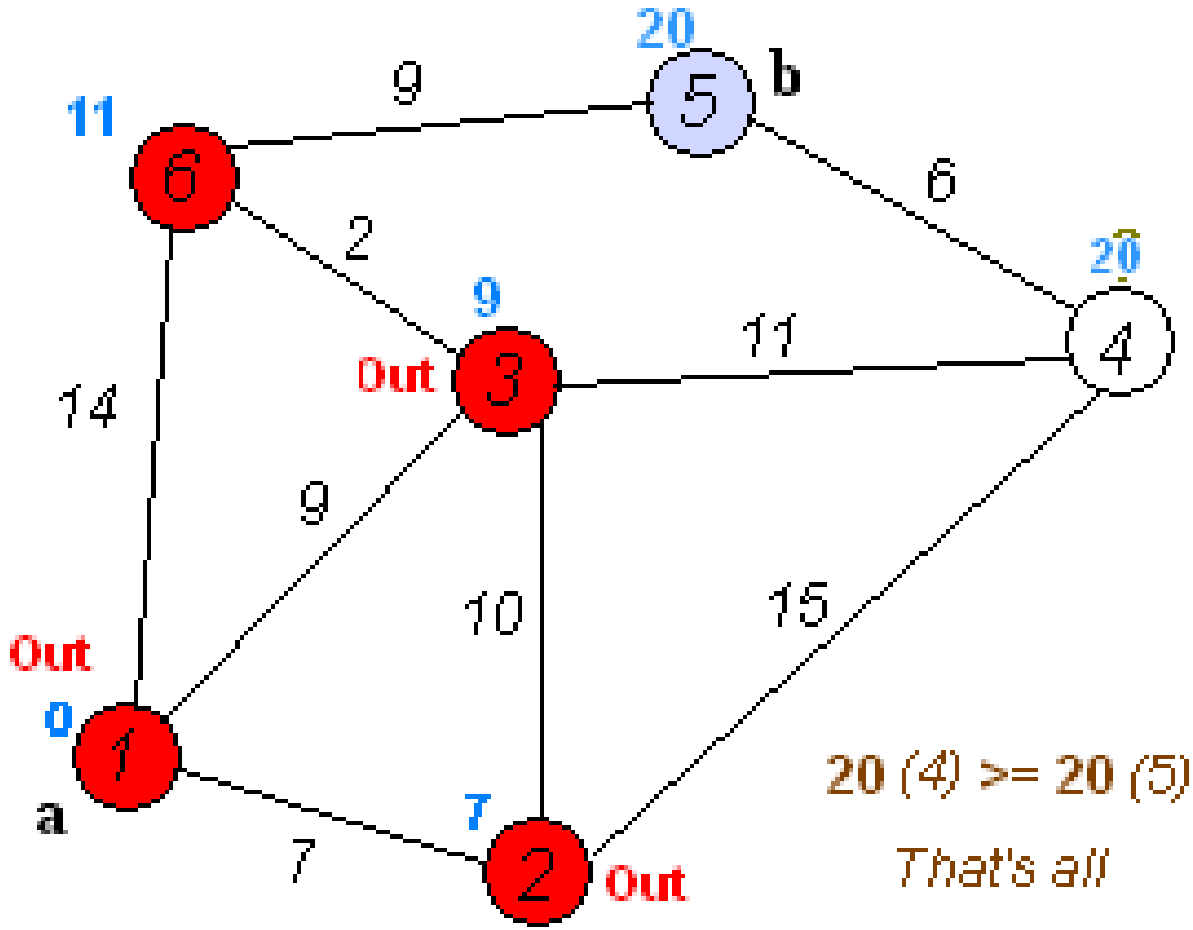

∞
b
5
9
11
6
6
2
20
9
Out
3
11
4
14
9
10
15
Out
0
1
a
7
7
2
Out


∞
b
5
9
11
6
6
2
20
9
Out
3
11
4
14
9
10
15
Out
0
1
a
7
7
2
Out


?
b
5
9
11
6
6
2
20
9
Out
3
11
4
14
9
10
15
Out
0
1
a
7
7
2
Out


20
b
5
9
11
6
6
2
20
9
Out
3
11
4
14
9
10
15
Out
0
1
a
7
7
2
Out


20
b
5
9
11
6
6
2
20
4
9
Out
3
11
14
9
10
15
Out
0
1
a
7
7
2
Out


20
b
5
9
11
6
6
2
20
9
Out
3
11
4
14
9
10
15
Out
0
1
a
7
7
2
Out
20 (4) >= 20 (5)
That's all

# Contenidos

# Posibles implementaciones de Dijkstra

- Existen dos implementaciones muy famosas del algoritmo de Dijkstra. Deberíamos analizar sobre qué tipo de grafo queremos correr este algoritmo y en función de eso elegir cuál programar
- La implementación básica es de $O(|V|^2)$
- La implementación con cola de prioridad es $O(|E|log|V|)$
- Existe una implementación re loca que no vamos a tratar cuya complejidad es de $O(|V|log|V| + |E|)$ (usa *Fibonacci heap*)
- Cuando da lo mismo cuál implementación elegir, **elijan la que sientan que entienden mejor y van a tener menos errores al programar**

# Posibles implementaciones de Dijkstra

- Si el grafo es **ralo**, o sea, tiene pocas aristas, conviene utilizar

# Posibles implementaciones de Dijkstra

- Si el grafo es **ralo**, o sea, tiene pocas aristas, conviene utilizar la implementación con cola de prioridad ($O(|E|log|V|)$)

# Posibles implementaciones de Dijkstra

- Si el grafo es **ralo**, o sea, tiene pocas aristas, conviene utilizar la implementación con cola de prioridad ($O(|E|log|V|)$)
- Si el grafo es **denso**, o sea, tiene muchas aristas, conviene utilizar

# Posibles implementaciones de Dijkstra

- Si el grafo es **ralo**, o sea, tiene pocas aristas, conviene utilizar la implementación con cola de prioridad ($O(|E|log|V|)$)
- Si el grafo es **denso**, o sea, tiene muchas aristas, conviene utilizar la implementación básica ($O(|V|^2)$)

# Pseudocódigo de Dijkstra sin cola de prioridad

```
1  Dijkstra (Grafo G, nodo inicial s)
2    visitado[n] = {false, ..., false} // guarda si un nodo ya fue visitado
3    distancia[n] = {Infinito, ..., Infinito} // guarda las distancias del nodo
         salida al resto
4
5    para cada w en V[G] hacer
6        si existe arista entre s y w entonces
7            distancia[w] = peso (s, w)
8
9    distancia[s] = 0
10   visitado[s] = true
11
12   mientras que no esten visitados todos hacer
13       v = nodo de menor distancia a s que no fue visitado aun
14       visitado[v] = true
15       para cada w en sucesores (G, v) hacer
16           si distancia[w] > distancia[v] + peso (v, w) entonces
17               distancia[w] = distancia[v] + peso (v, w)
```

# Pseudocódigo de Dijkstra con cola de prioridad

```
1  Dijkstra (Grafo G, nodo_fuente s)
2      para todo u en V[G] hacer
3          distancia[u] = INFINITO
4          padre[u] = NULL
5          visitado[u] = false
6
7      distancia[s] = 0
8      adicionar (cola, (s, distancia[s]))
9
10     mientras que cola no sea vacia hacer
11         u = extraer_minimo(cola)
12         visitado[u] = true
13         para todo v en adyacencia[u] hacer
14             si no visitado[v] y distancia[v] > distancia[u] + peso (u, v)
                   hacer
15                 distancia[v] = distancia[u] + peso (u, v)
16                 padre[v] = u
17                 adicionar(cola,(v, distancia[v]))
```

# Charly and Nito

Nito y Charly son amigos y se juntaron en un bar. Tipo 3am, comenzaron a tener sueño y quieren volver cada uno a su casa. Cada uno desea volver utilizando un camino que minimice la distancia a su casa. Sin embargo, a Nito y Charly les gusta caminar juntos, por lo que quieren caminar juntos lo más posible sin que ninguno deba hacer un camino más largo de lo que podría haber hecho.

La ciudad donde viven Nito y Charly se puede modelar como un conjunto de calles de distintas longitudes y esquinas: cada calle conecta dos esquinas distintas. Charly y Nito no viven juntos, y no viven en el bar. Existe al menos un camino desde el bar a lo de Charly y a lo de Nito.

¿Cuál es la distancia máxima que podrán caminar juntos con estas condiciones? ¿Cuánto caminará solo cada uno?
`http://www.spoj.com/problems/CANDN/`

# Contenidos

# Repaso de definiciones

## Definición (Árbol generador)

Sea $G$ un grafo. Decimos que $T$ es un *árbol generador* de $G$ si se cumplen estas condiciones:

- $T$ es subgrafo de $G$.
- $T$ es un árbol.
- $T$ tiene todos los vértices de $G$.

## Definición (Costo de un árbol generador)

Sea $G$ un grafo con pesos en sus aristas. Si $T$ es un árbol generador de $G$, definimos el *costo* de $T$ como la suma de los pesos de las aristas de $T$.

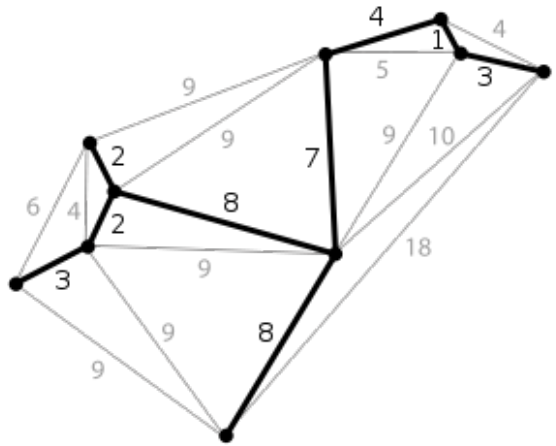
# ¿Qué es un AGM?

## Definición (AGM)

Sea $G$ un grafo. Decimos que un grafo $T$ es un árbol generador mínimo (AGM) de $G$ si:

- $T$ es un árbol generador de $G$.
- El costo de $T$ es mínimo con respecto a todos los árboles generadores de $G$.

Aca vemos un grafo $G$ (en gris) y un subgrafo $T$ (en negro) que es AGM de $G$.



# Contenidos

# ¿Como hallamos un AGM de un grafo?

El algoritmo de Kruskal sirve para resolver el problema de hallar un AGM de un grafo. Un pseudocódigo del algoritmo es el siguiente:

```
1 lista(aristas) Kruskal(lista(aristas)):
2     solucion = lista vacia // inicializo la solucion
3     sort(aristas) // ordeno las aristas por peso (de menor a mayor)
4     for e in aristas:
5         si agregar e a solucion no genera un ciclo:
6             agregar e a la solucion
7     return solucion
```

El código parece muy simple pero hay un paso no trivial clave. ¿Cómo verificamos (rápido) que agregar una arista no genera un ciclo?

# Detectando ciclos

Notemos que al ejecutar Kruskal sobre un grafo $G$:

- En cada paso intermedio la solución (parcial) nos divide los vértices de $G$ en varias componentes conexas.
- Una nueva arista nos generaría un ciclo si (y sólo si) nos une dos vértices de la misma componente conexa.
- Al agregar una nueva arista a la solución juntamos dos componentes conexas en una sola más grande.

Sabiendo esto, la siguiente estructura nos será de utilidad.

# UDS

La estructura de datos Union Disjoint Set (UDS) es una estructura que nos permite manejar conjuntos disjuntos de elementos (como las componentes conexas).
Cada conjunto tiene un representante que lo identifica.
La estructura nos debe proveer las siguientes dos operaciones:

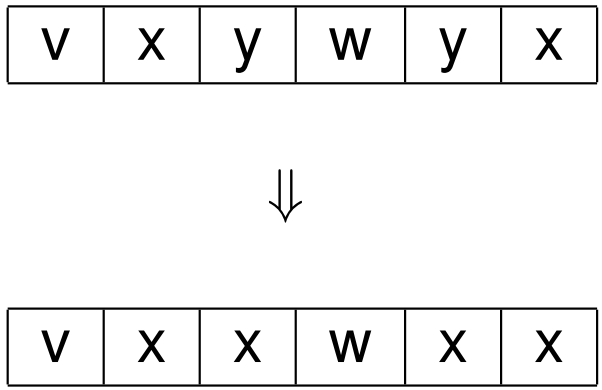
- find($x$): Dado un elemento $x$, nos dice quien es el representante del conjunto al que pertenece $x$.
- union($x$, $y$): Dados dos elemenos $x$ e $y$, une los conjuntos a los que pertenecen $x$ e $y$ en uno solo.

En el caso de Kruskal, inicialmente cada elemento $x$ está en un conjunto distinto y es su propio representante.

# Primer approach

Una posible implementación de DSU es tener un arreglo $C[]$ con los representantes de cada elemento. Podemos implementar las dos operaciones de esta manera:

```
1  find(x):
2      return C[x]
3
4  union(x, y):
5    for z tal que C[z] = C[y]:
6        C[z] = C[x]
```




El problema es que con este algoritmo si hay revisamos todos los $z$ posibles, hacer $O(n)$ unions donde $n$ es la cantidad de elementos es $O(n^2)$. Buscamos algo mejor.

# Optimizaciones

Esto se puede mejorar con las siguientes dos obervaciones:

- Podemos mantener una lista con los elementos del conjunto representado por $x$ para cada $x$.
- Cuando unimos dos conjuntos elegimos recorrer los elementos del conjunto más chico. Para esto hay que llevar una cuenta de cuantos elementos tiene cada conjunto.

Con estas dos optimizaciones se puede ver que el costo de hacer $O(n)$ operaciones union o find es $O(nlog(n))$. Pero como somos ambiciosos vamos a buscar algo mejor todavía.

# Versión con listas

```
1  | init(n):
2  |     for i en [0, n):
3  |         componente[i] = {i} // conjunto solo con el elemento i
4  |         padre[i] = i
5  |
6  | find(x):
7  |     return padre[x]
8  |
9  | union(x, y):
10 |     x = find(x), y = find(y)
11 |     if longitud(componente[x]) > longitud(componente[y]):
12 |         intercambiar x e y
13 |
14 |     for z en componente[x]:
15 |         padre[z] = y
16 |         agregar z a componente[y]
17 |     vaciar componente[x]
```

# Versión con listas

```
1  | init(n):
2  |     for i en [0, n):
3  |         componente[i] = {i} // conjunto solo con el elemento i
4  |         padre[i] = i
5  |
6  | find(x):
7  |     return padre[x]
8  |
9  | union(x, y):
10 |     x = find(x), y = find(y)
11 |     if longitud(componente[x]) > longitud(componente[y]):
12 |         intercambiar x e y
13 |
14 |     for z en componente[x]:
15 |         padre[z] = y
16 |         agregar z a componente[y]
17 |     vaciar componente[x]
```

Como siempre agregamos los elementos del conjunto más chico al conjunto más grande, cada elemento que movemos de un conjunto a otro pasa a estar en un conjunto de al menos el doble de tamaño del anterior. Por lo tanto, cada elemento es agregado a lo sumo $O(\lg n)$ veces: esto implica una complejidad total de $O(n \lg n)$.

# ¡Un conjunto es un árbol!

La idea clave es que vamos a representar cada conjunto como un árbol.
El representante de un conjunto será la raíz del árbol al que pertenece.
Cada elemento sabe quién es su padre en el árbol. Si un elemento es raíz su padre es sí mismo.
Los algoritmos quedan así:

```
1 find(x):
2     if padre[x] != x:
3         return find(padre[x]);
4     return x;
5
6 union(x, y):
7     padre[find(x)] = padre[find(y)];
```

# Optimizando II

Podemos hacer dos mejoras a este algoritmo:

- Colgar el árbol de menor altura al de mayor altura, pues el costo de un find es a lo sumo la altura del árbol.
- Cada vez que realizamos un find($x$) actualizamos el padre de $x$ con el resultado del find para ahorrarnos volver a tener que subir por el árbol ante futuros find.

Con estas dos optimizaciones se puede probar que realizar $O(n)$ operaciones union o find es $O(n \lg^*(n))$.

# El final de la película

Así nos queda nuestro algoritmo que implementa DSU (incluyendo la inicialización):

```
1  init(n): \\ Inicializa los arreglos sabiendo que hay n elementos
2      for i = 1 to n:
3          altura[i] = 1; // "altura" es una cota superior de la altura
4          padre[i] = i;
5
6  find(x):
7      if padre[x] != x:
8          padre[x] = find(padre[x]);
9      return padre[x];
10
11 union(x, y):
12     x = find(x), y = find(y) // Tomo los representantes de cada conjunto
13     if altura[x] < altura[y]:
14         padre[x] = y;
15     else
16         padre[y] = x;
17     if altura[x] = altura[y]:
18         altura[x] = altura[x] + 1;
```

# Contenidos

# La verdad de la milanesa

Ahora sí podemos dar un algoritmo completo y eficiente de Kruskal:

```
1  lista(aristas) Kruskal(lista(aristas), int n):
2      solucion = lista vacia // inicializo la solucion
3      init(n)
4      sort(aristas) // ordeno las aristas por peso (de menor a mayor)
5      for e in aristas:
6          if find(e.inicio) != find(e.fin): // si los vertices que une la
               arista estan en componentes distintas
7              agregar e a la solucion
8              union(e.inicio, e.fin) // uno las dos componentes
9      return solucion
```

Este algoritmo es $O(V + E \lg V)$ con $V$ la cantidad de vértices y $E$ la cantidad de aristas.

# Contenidos

# Arturo y las chinchillas

Arturo se compró una mansión en la cordillera de los Andes con el objetivo de estudiar a las chinchillas. Las chinchillas viven en túneles subterráneos y cada una tiene un circuito de túneles por el que se mueve. Arturo quiere colocar cámaras en los túneles de manera de estudiar el movimiento de las chinchillas. Para ello cuenta con un mapa de los túneles subterráneos que consiste de puntos de intersección entre túneles y la longitud entre dos puntos de intersección.

Sabe que colocar una cámara en un túnel de longitud *l* le cuesta *l* pesos y que cada chinchilla se mueve por un ciclo de túneles cerrados sin pasar dos veces por el mismo punto hasta volver al inicio del ciclo. Quiere colocar las cámaras de manera de que todo ciclo posible de alguna chinchilla posea una cámara y gastando la menor cantidad de dinero.
¿Cuánto le va a costar realizar esto? ¿Cuál es el costo de la cámara más cara que tiene que colocar?

El algoritmo deberá tener una complejidad temporal de $O(V + ElogV)$.

# ¡A pensar!

# Contenidos

## Definición 1

Si $S$ es un subgrafo de $G$ decimos que es *solución* del problema para $G$ si todo ciclo simple de $G$ tiene una arista de $S$

## Definición 1

Si $S$ es un subgrafo de $G$ decimos que es *solución* del problema para $G$ si todo ciclo simple de $G$ tiene una arista de $S$

## Definición 2

Si $S$ es un subgrafo de $G$ llamamos $G - S$ al subgrafo de $G$ que tiene los vértices de $G$ y las aristas de $G$ que no están en $S$.

### Definición 1

Si $S$ es un subgrafo de $G$ decimos que es *solución* del problema para $G$ si todo ciclo simple de $G$ tiene una arista de $S$

### Definición 2

Si $S$ es un subgrafo de $G$ llamamos $G - S$ al subgrafo de $G$ que tiene los vértices de $G$ y las aristas de $G$ que no están en $S$.

### Observación 1

$S$ es una solución minimal para el grafo $G$ si y sólo si $G - S$ es un árbol generador.

## Observación 2

$S$ es una solución de costo mínimo para el grafo $G$ si y sólo si $G - S$ es un árbol generador de costo máximo de $G$.

## Observación 2

$S$ es una solución de costo mínimo para el grafo $G$ si y sólo si $G - S$ es un árbol generador de costo máximo de $G$.

## Observación 3

El algoritmo de Kruskal calcula un árbol generador de costo máximo si ordenamos las aristas de mayor a menor costo (en lugar de menor a mayor).

### Observación 2

$S$ es una solución de costo mínimo para el grafo $G$ si y sólo si $G - S$ es un árbol generador de costo máximo de $G$.

### Observación 3

El algoritmo de Kruskal calcula un árbol generador de costo máximo si ordenamos las aristas de mayor a menor costo (en lugar de menor a mayor).

### Observación 4

La arista más pesada es la primera que no agregamos a la solución en el algoritmo de Kruskal.

# ¿Preguntas?