Grafos

Martin Fixman[1]

[1] Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Training Camp 2014

Contenidos

Contenidos

¿Qué es un grafo?

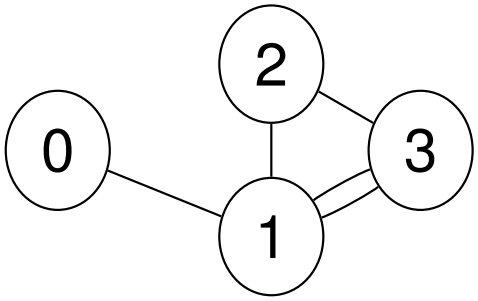
- "Un país tiene *N* ciudades conectadas por *M* calles de diferente largo. ¿Cuál es el camino más corto desde la ciudad *a* hasta la ciudad *b*?"
- "Beto y Carlos tienen un árbol genealógico de sus familias. ¿Cúal es su ancestro común menor?"
- "La ciudad Prusiana de Könisberg (actualmente parte de Rusia y llamada Kaliningrado) tiene 7 puentes que unen las 2 márgenes del río Pregel y otras 2 islas. ¿Es posible, empezando desde cualquiera de las 4 regiones, dar un paseo cruzando exactamente una vez cada puente y regresando al punto de partida?

Un grafo es una representación de un grupo de objetos conectados donde algunos pares están conectados por aristas.

Formalmente,

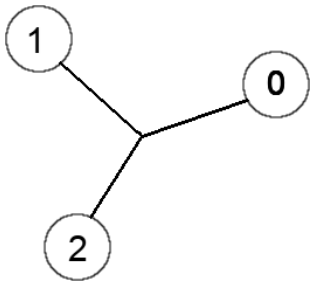
Grafo

Un grafo es representado por un par $G = (V, E)$, donde V es un conjunto de nodos (o vértices), y $E \subseteq V \times V$ es un conjunto de aristas (o ejes). En el caso de un grafo no dirigido, $(a, b) \in E \iff (b, a) \in E$.

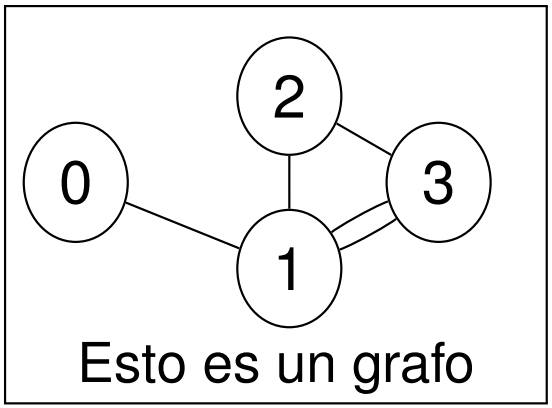


Esto es un grafo

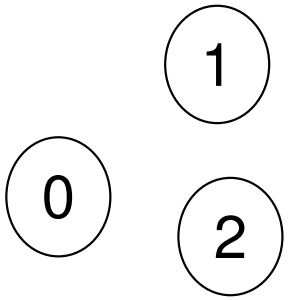
Esto es un grafo



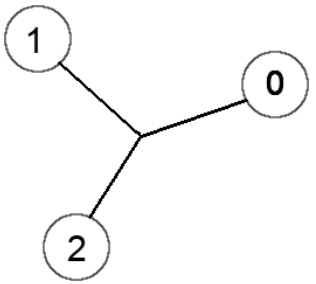
Esto no es un grafo



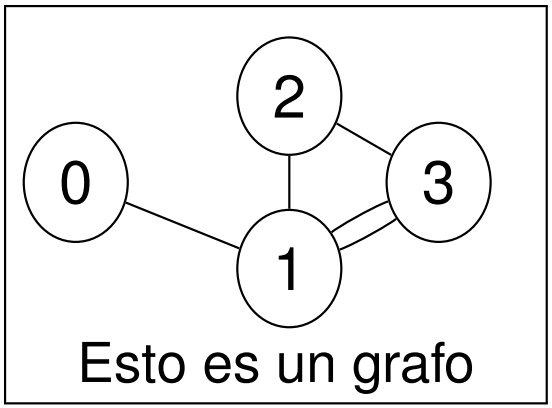
Esto es un grafo



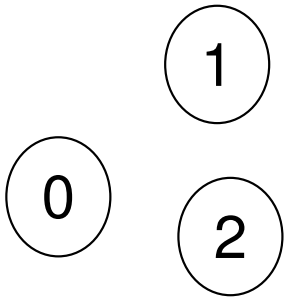
Esto es un grafo



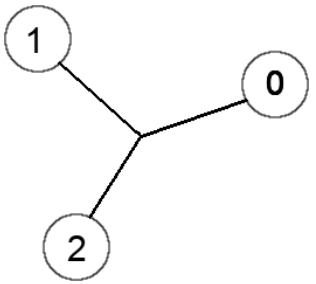
Esto no es un grafo



Esto es un grafo



Esto es un grafo

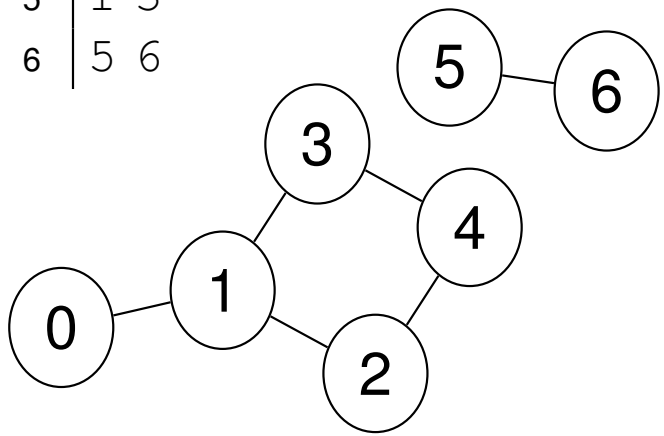


Esto no es un grafo

Contenidos

Pares de Adyacencia

1	0 1
2	1 2
3	2 4
4	3 4
5	1 3
6	5 6



Una lista de pares donde cada par representa una arista.

Ventajas:

- Solo usa $\mathcal{O}(|E|)$ de memoria.
- Es simple de parsear.

Desventajas:

- Hacer cualquier operación es dificil y lento.

Por lo general solo se usan para la entrada en los problemas de grafos.

Matrices de Adyacencia

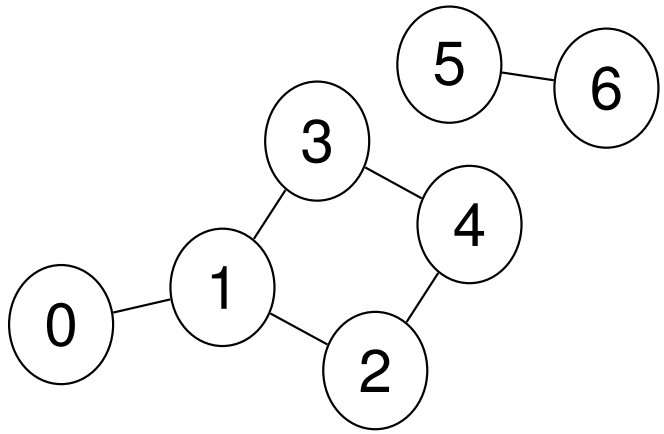
Una matriz donde cada el elemento en (i, j) indica si hay una arista entre el nodo i y el nodo j.

Ventajas:

- Saber si dos nodos están conectados tiene complejidad $\mathcal{O}(1)$
- Se pueden hacer operaciones sobre esta matriz para encontrar propiedades del grafo.

Una matriz donde cada el elemento en (i, j) indica si hay una arista entre el nodo i y el nodo j.

	0	1	2	3	4	5	6
0	0	1	0	0	0	0	0
1	1	0	1	1	0	0	0
2	0	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	0	1	0



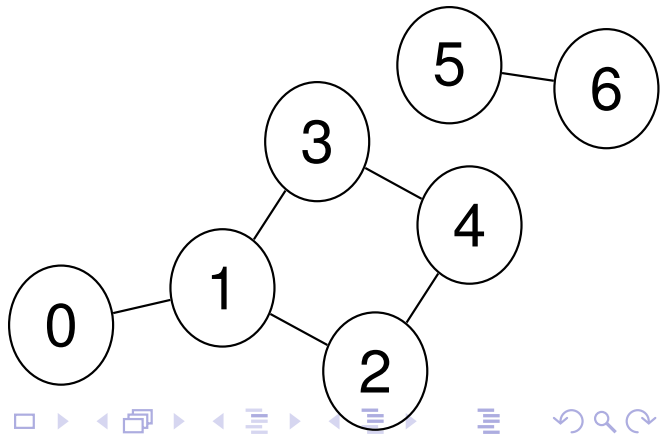
Matrices de Adyacencia

Una matriz donde cada el elemento en (i, j) indica si hay una arista entre el nodo i y el nodo j.

	0	1	2	3	4	5	6
0	0	1	0	0	0	0	0
1	1	0	1	1	0	0	0
2	0	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	0	1	0

Desventajas:

- Buscar los nodos adyacentes a otro nodo tiene complejidad $\mathcal{O}(|V|)$, sin importar cuantos nodos adyacentes tenga el primer nodo.
- Recorrer todo el grafo tiene complejidad $\mathcal{O}(|V|^2)$.
- La matriz tiene una complejidad en memoria de $\mathcal{O}(|V|^2)$.
- No hay ninguna manera obvia de

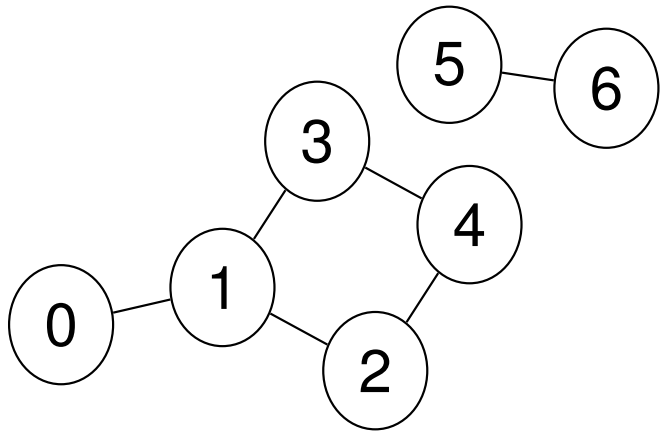


Matrices de Adyacencia

Una matriz donde cada el elemento en (i, j) indica si hay una arista entre el nodo i y el nodo j.

Es ideal para cuando $|V|$ es pequeño y hay que encontrar relaciones entre pares de nodos muy seguido, o cuando el grafo es denso.

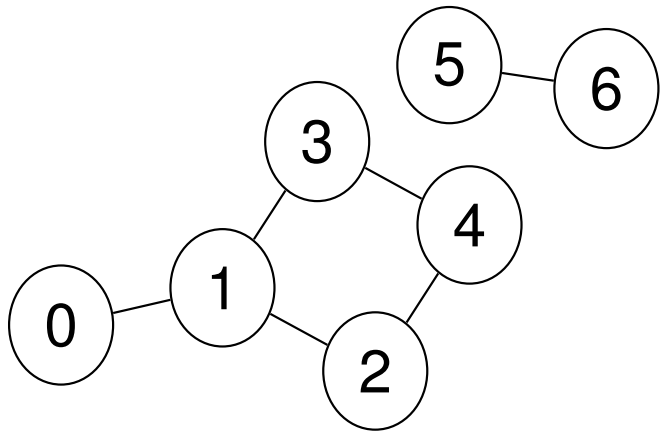
	0	1	2	3	4	5	6
0	0	1	0	0	0	0	0
1	1	0	1	1	0	0	0
2	0	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	0	1	0



Listas de Adyacencia

Una lista con $|V|$ elementos, donde el elemento númeo i tiene la lista de nodos adyacentes al nodo i.

0	1
1	2 0 3
2	1 4
3	4 1
4	3 2
5	6
6	5



Ventajas:

- Tiene $\mathcal{O}(|E|)$ de complejidad de memoria.
- Encontrar la cantidad de nodos adyacentes a cierto nodo tiene complejidad lineal en el tamaño de la respuesta.
- Recorrar todo el grafo tiene complejidad $\mathcal{O}(|V| + |E|)$.

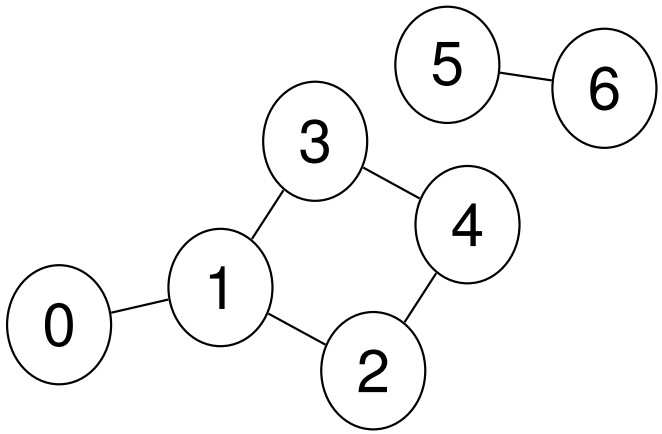
Listas de Adyacencia

0	1
1	2 0 3
2	1 4
3	4 1
4	3 2
5	6
6	5

Una lista con $|V|$ elementos, donde el elemento númeo i tiene la lista de nodos adyacentes al nodo i.

Desventajas:

- Saber si dos nodos están conectados es lineal al grado de alguno de los dos nodos, que en el peor caso es $\mathcal{O}(|V|)$.



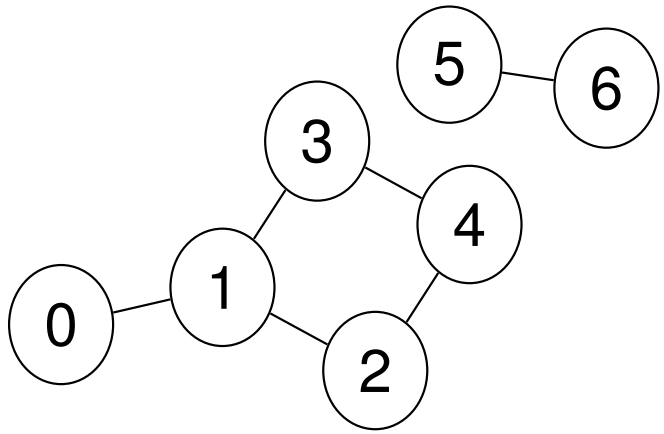
Listas de Adyacencia

Una lista con $|V|$ elementos, donde el elemento númeo i tiene la lista de nodos adyacentes al nodo i.

Por lo general se usa en grafos esparsos, cuando $|V|$ es grande, o cuando los multiejes son significativos.

Si se guardan los elementos de cada lista ordenados o se usa un árbol binario de búsqueda balanceado (como el *set* de la STL) para representar cada lista, tanto agregar un nodo como chequear si dos nodos son adyacentes pasa a tener complejidad $\mathcal{O}(\log(|V|))$.

0	1
1	2 0 3
2	1 4
3	4 1
4	3 2
5	6
6	5



Contenidos

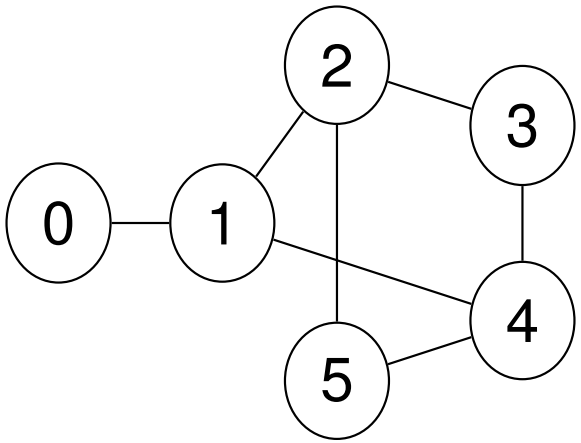
Recorriendo un grafo

Muchas veces la solución de un problema requiere un algoritmo que recorra los nodos de un grafo en algún orden el particular. Por lo general, en grafos conexos, estos algoritmos se pueden representar de esta forma:

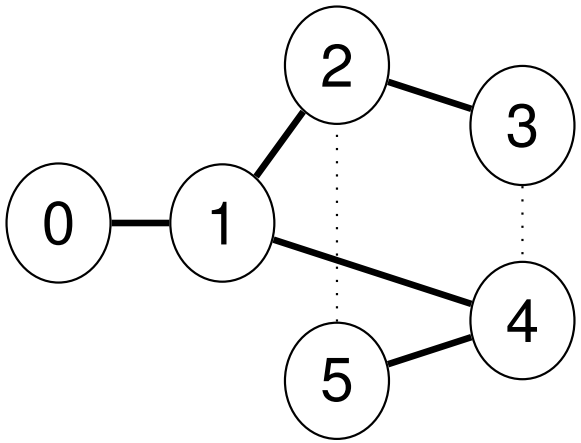
```
visitados ← ∅;
proximo ← {X_0};
while proximo not empty do
    nodo ← proximo.pop() ; // Sacar un elemento de proximo
    if nodo ∈ visitados then
        continue ; // Descartar este nodo
    end
    visitados ← visitados ∪ {nodo};

    for siguiente ∈ adyacentes(nodo) do
        proximo ← proximo ∪ {siguiente};
    end
end
```

En el caso de este pseudocódigo, se puede crear un subgrafo $G' = (V, E')$ con $E' \subseteq E$ donde los nodos i y j, adyacentes en G, son adyacentes en G' si se agrego al nodo j a *proximo* cuando se estaban buscando los nodos adyacentes a i. A este grafo se lo llama **Árbol generador**.
El árbol generador se crea cuando se "ignora" un eje porque uno de los nodos de su extremo ya está visitado.



En el caso de este pseudocódigo, se puede crear un subgrafo $G' = (V, E')$ con $E' \subseteq E$ donde los nodos i y j, adyacentes en G, son adyacentes en G' si se agrego al nodo j a *proximo* cuando se estaban buscando los nodos adyacentes a i. A este grafo se lo llama **Árbol generador**.
El árbol generador se crea cuando se "ignora" un eje porque uno de los nodos de su extremo ya está visitado.



Contenidos

Árboles

Abramos un parentesis para definir que es un árbol.

Un árbol es un grafo G conexo tal que

- G no tiene ciclos.
- Si se le saca cualquier arista a G, G ya no es conexo.
- Si se le agrega cualquier arista a G, G pasa a tener ciclos.
- Hay un solo camino entre cada par de nodos de G
- G tiene n nodos y $n - 1$ aristas.

De hecho, todas esas propiedades son equivalentes. Si se cumple cualquiera, se cumplen todas.

Árboles

Abramos un parentesis para definir que es un árbol.

Un árbol es un grafo G conexo tal que

- G no tiene ciclos.
- Si se le saca cualquier arista a G, G ya no es conexo.
- Si se le agrega cualquier arista a G, G pasa a tener ciclos.
- Hay un solo camino entre cada par de nodos de G
- G tiene n nodos y $n - 1$ aristas.

De hecho, todas esas propiedades son equivalentes. Si se cumple cualquiera, se cumplen todas.

Árboles

Abramos un parentesis para definir que es un árbol.

Un árbol es un grafo G conexo tal que

- G no tiene ciclos.
- Si se le saca cualquier arista a G, G ya no es conexo.
- Si se le agrega cualquier arista a G, G pasa a tener ciclos.
- Hay un solo camino entre cada par de nodos de G
- G tiene n nodos y $n - 1$ aristas.

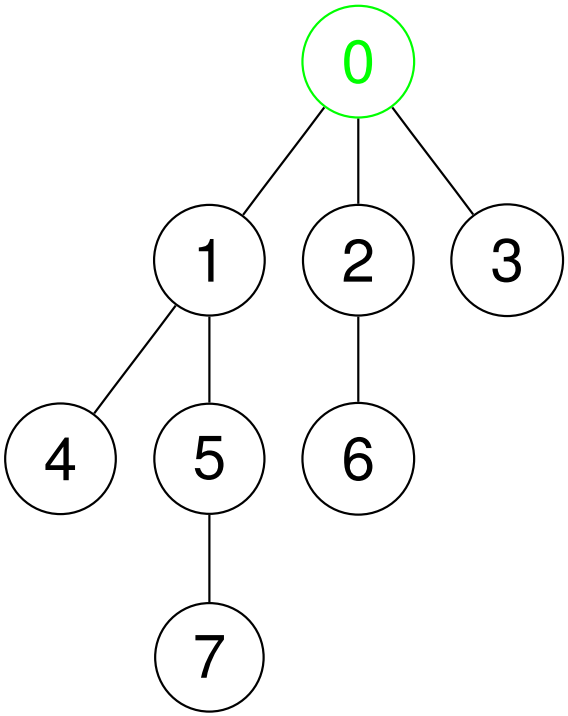
De hecho, todas esas propiedades son equivalentes. Si se cumple cualquiera, se cumplen todas.

Contenidos

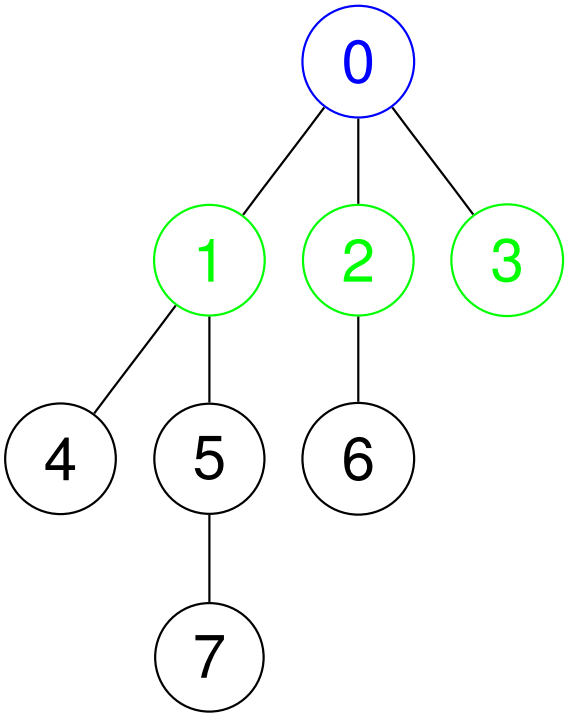
Breath First Search

Una manera de recorrer un árbol (o de generar un árbol generador en un grafo y recorrerlo) es mediante la búsqueda en anchura, usualmente llamada Breath First Search o BFS. En esta *proximo* se representa mediante una **cola**, donde *pop* remueve el elemento que se insertó hace más tiempo.

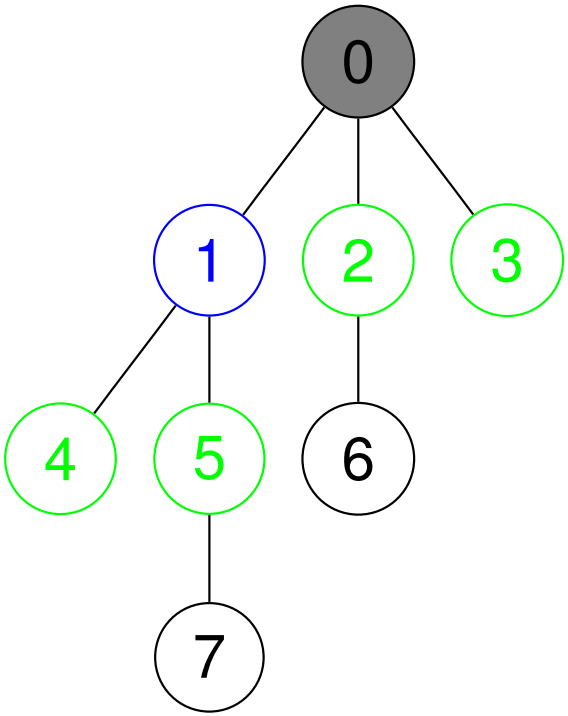
La compejidad de recorrer todo un grafo usando BFS es $\mathcal{O}(|V| + |E|)$.



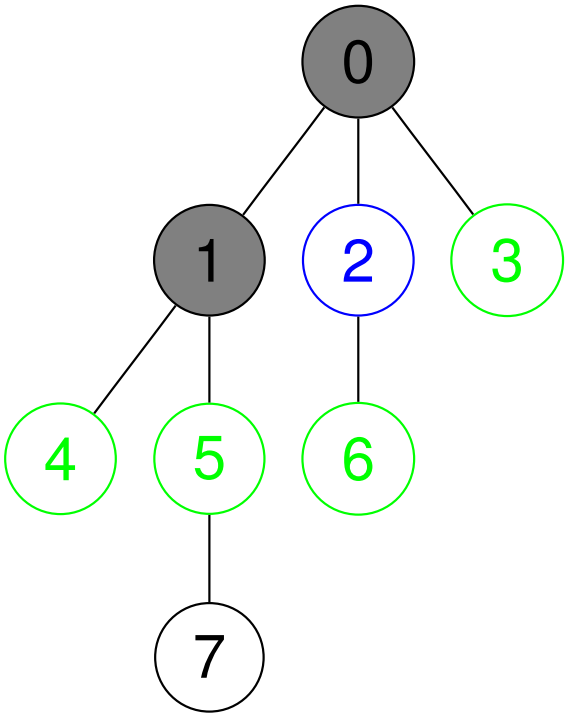
Nodos visitados: []
Nodos próximos: [0]



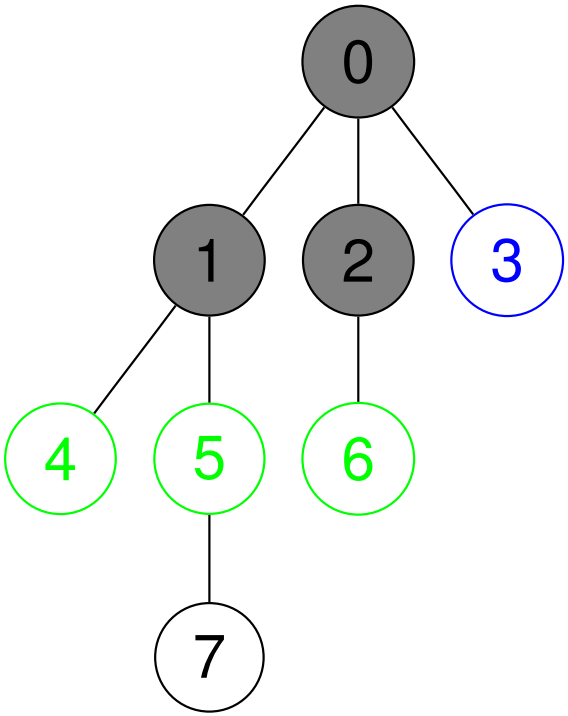
Nodos visitados: []
Nodos próximos: [1 2 3]



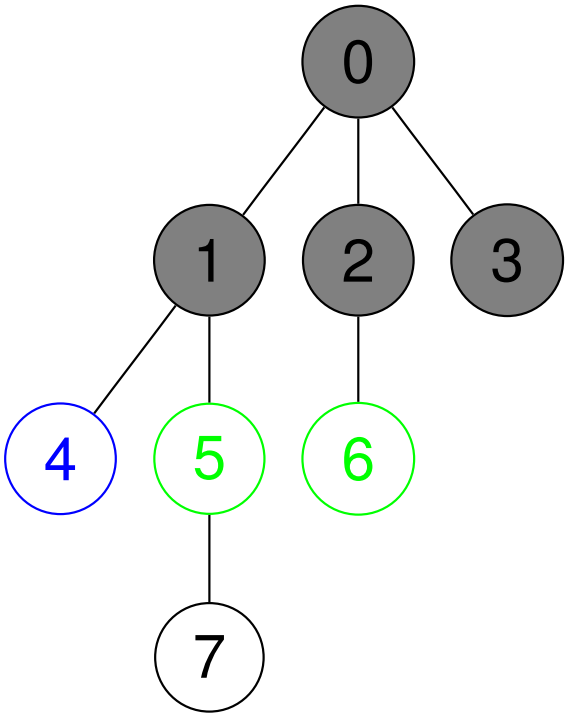
Nodos visitados: [0]
Nodos próximos: [2 3 4 5]



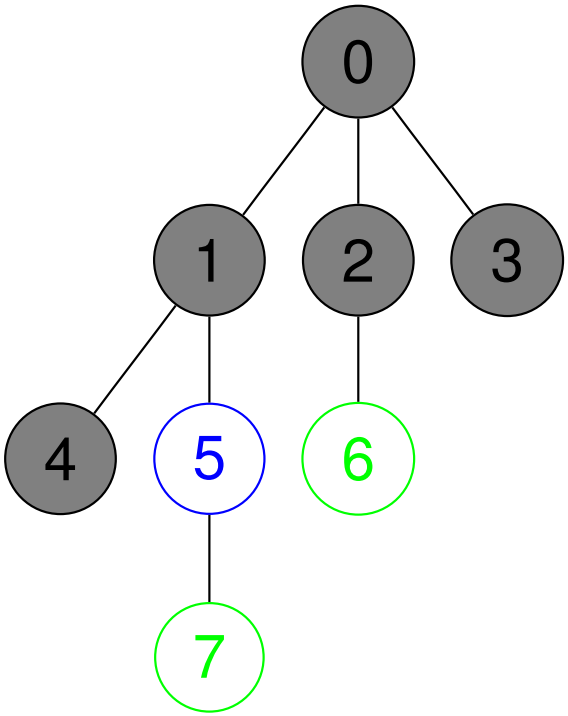
Nodos visitados: [0 1]
Nodos próximos: [3 4 5 6]



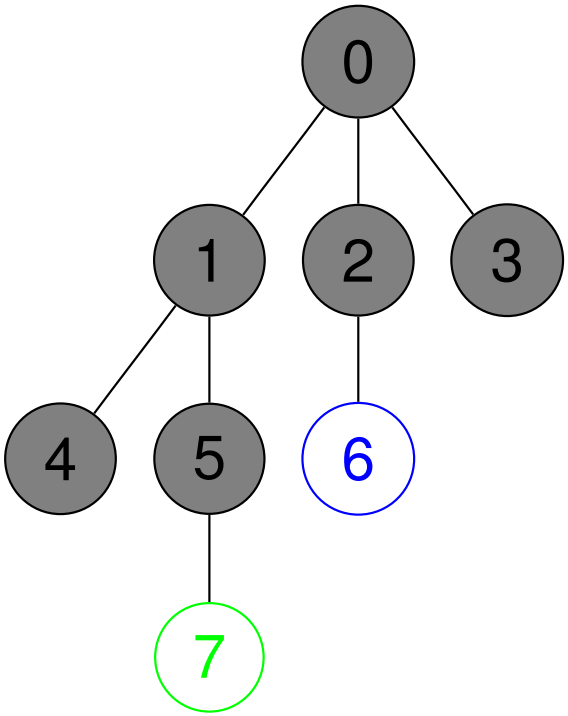
Nodos visitados: [0 1 2]
Nodos próximos: [4 5 6]



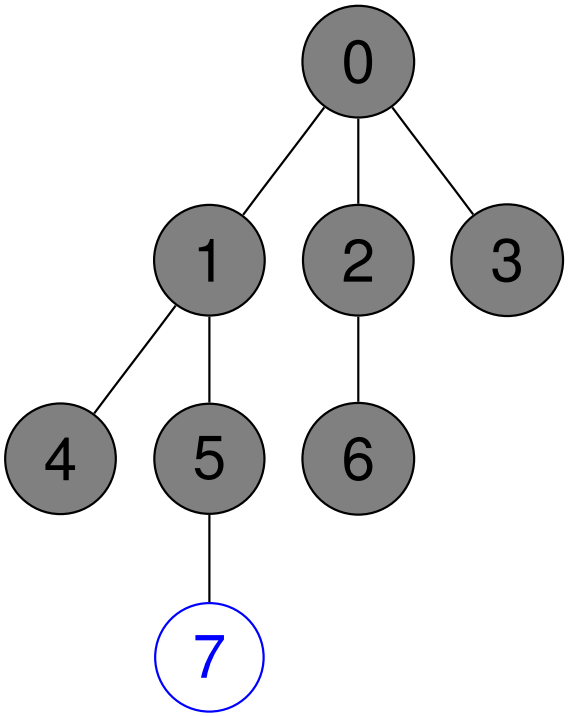
Nodos visitados: [0 1 2 3]
Nodos próximos: [5 6]



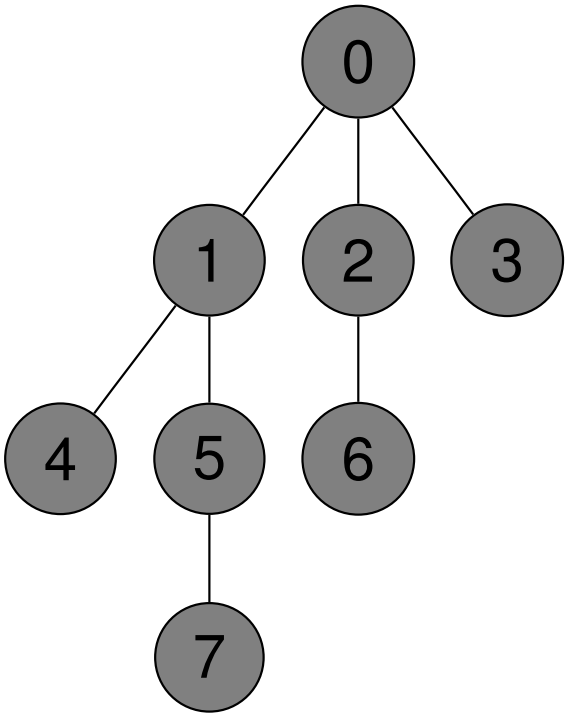
Nodos visitados: [0 1 2 3 4]
Nodos próximos: [6 7]



Nodos visitados: [0 1 2 3 4 5]
Nodos próximos: [7]



Nodos visitados: [0 1 2 3 4 5 6]
Nodos próximos: []



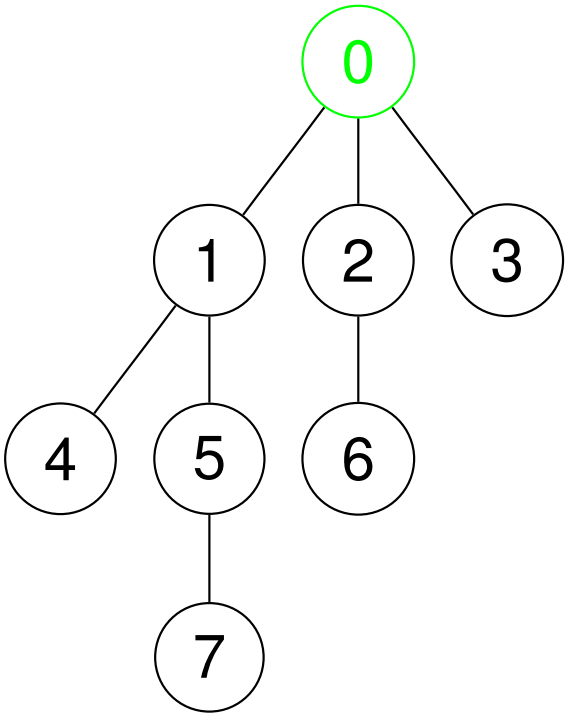
Nodos visitados: [0 1 2 3 4 5 6 7]
Nodos próximos: []

Contenidos

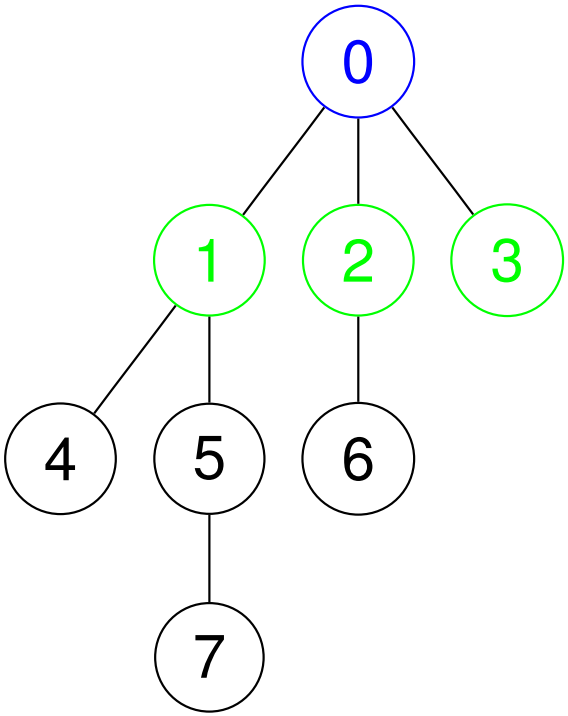
Depth First Search

Otra manera de recorrer un árbol es mediante la búsqueda en altura, usualmente llamada Depth First Search o DFS. En esta *proximo* se representa mediante una **pila**, donde *pop* remueve último elemento que se insertó.

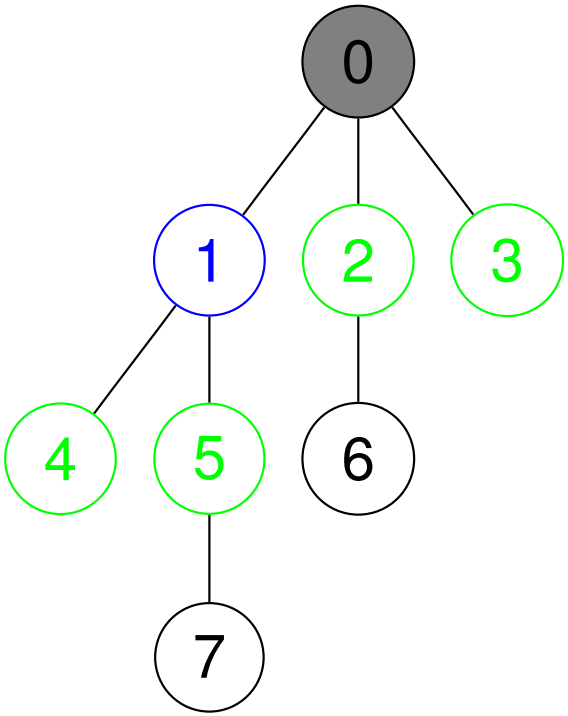
La compejidad de recorrer todo un grafo usando DFS es $\mathcal{O}(|V| + |E|)$.



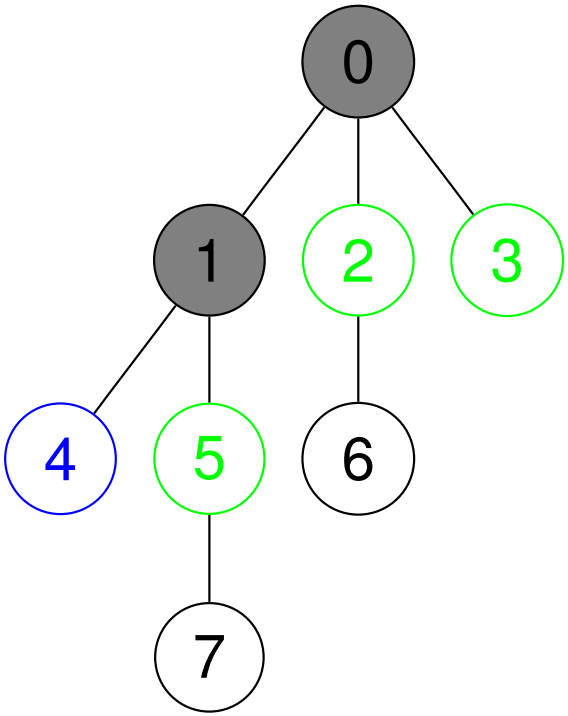
Nodos visitados: []
Nodos próximos: [0]



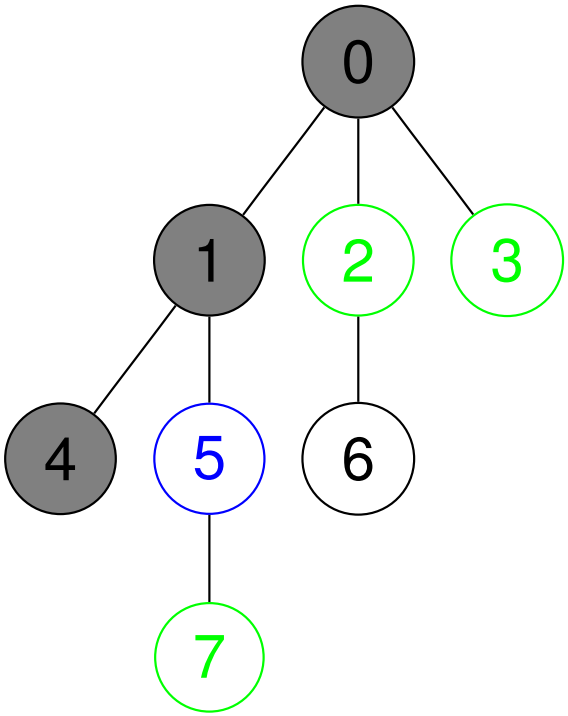
Nodos visitados: []
Nodos próximos: [1 2 3]



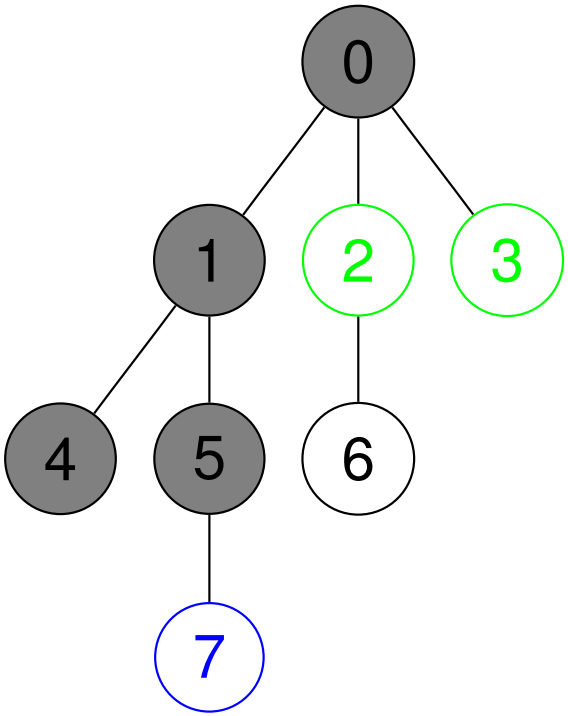
Nodos visitados: [0]
Nodos próximos: [4 5 2 3]



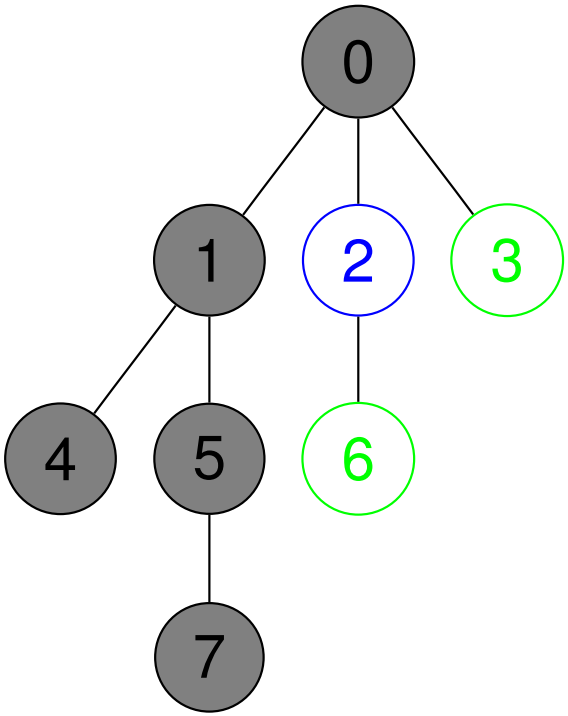
Nodos visitados: [0 1]
Nodos próximos: [5 2 3]



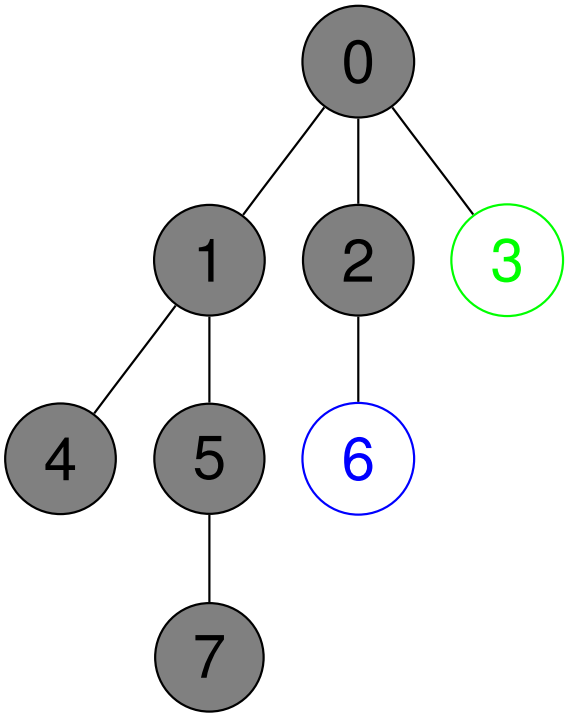
Nodos visitados: [0 1 4]
Nodos próximos: [7 2 3]



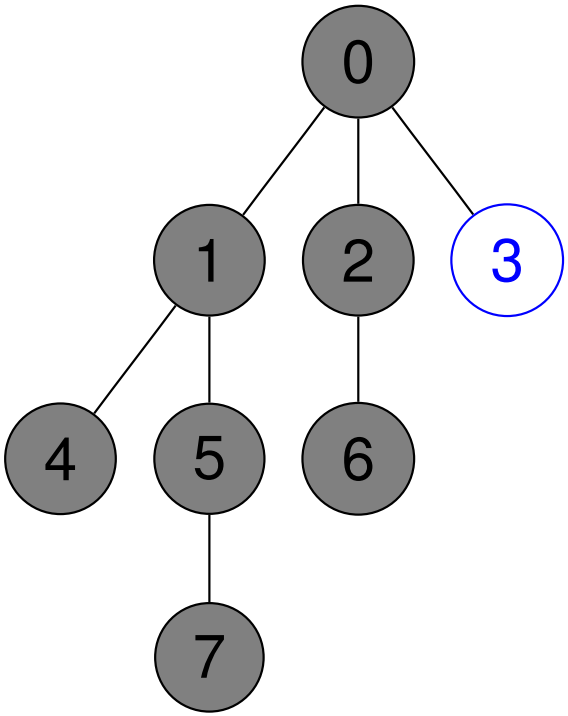
Nodos visitados: [0 1 4 5]
Nodos próximos: [2 3]



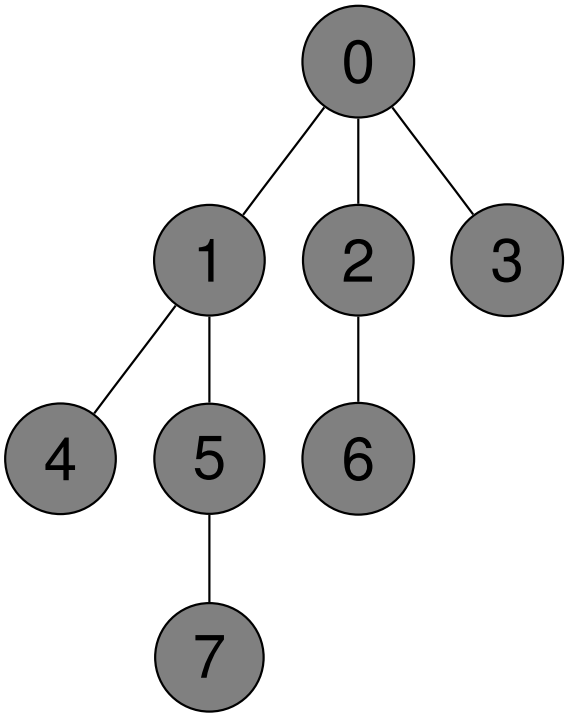
Nodos visitados: [0 1 4 5 7]
Nodos próximos: [6 3]



Nodos visitados: [0 1 4 5 7 2]
Nodos próximos: [3]



Nodos visitados: [0 1 4 5 7 2 6]
Nodos próximos: []



Nodos visitados: [0 1 4 5 7 2 6 3]
Nodos próximos: []

Contenidos

Cámino Mínimo

Dos de los problemas de grafos más estudiados son el la busqueda de cámino mínimo y el de cámino máximo.

El problema en general consiste en encontrar el camino $[v_0, v_1, \ldots, v_n]$ entre dos nodos i y j tal que $v_0 = i$ y $v_n = j$, tal que la distancia $d = \sum_{i=0}^{n} p_{v_i}$ sea lo menor (o lo mayor) posible.

Este problema se podía resolver fácilmente en un árbol o un grafo donde todas las aristas tengan el mismo peso, en tiempo lineal, usando BFS o DFS. Cuando usamos un grafo ponderado, es decir, donde las aristas pueden tener diferentes pesos, estos algoritmos no siempre dan una solución optima.

Cámino Mínimo

Dos de los problemas de grafos más estudiados son el la busqueda de cámino mínimo y el de cámino máximo.

El problema en general consiste en encontrar el camino $[v_0, v_1, \ldots, v_n]$ entre dos nodos i y j tal que $v_0 = i$ y $v_n = j$, tal que la distancia $d = \sum_{i=0}^{n} p_{v_i}$ sea lo menor (o lo mayor) posible.

Este problema se podía resolver fácilmente en un árbol o un grafo donde todas las aristas tengan el mismo peso, en tiempo lineal, usando BFS o DFS. Cuando usamos un grafo ponderado, es decir, donde las aristas pueden tener diferentes pesos, estos algoritmos no siempre dan una solución optima.

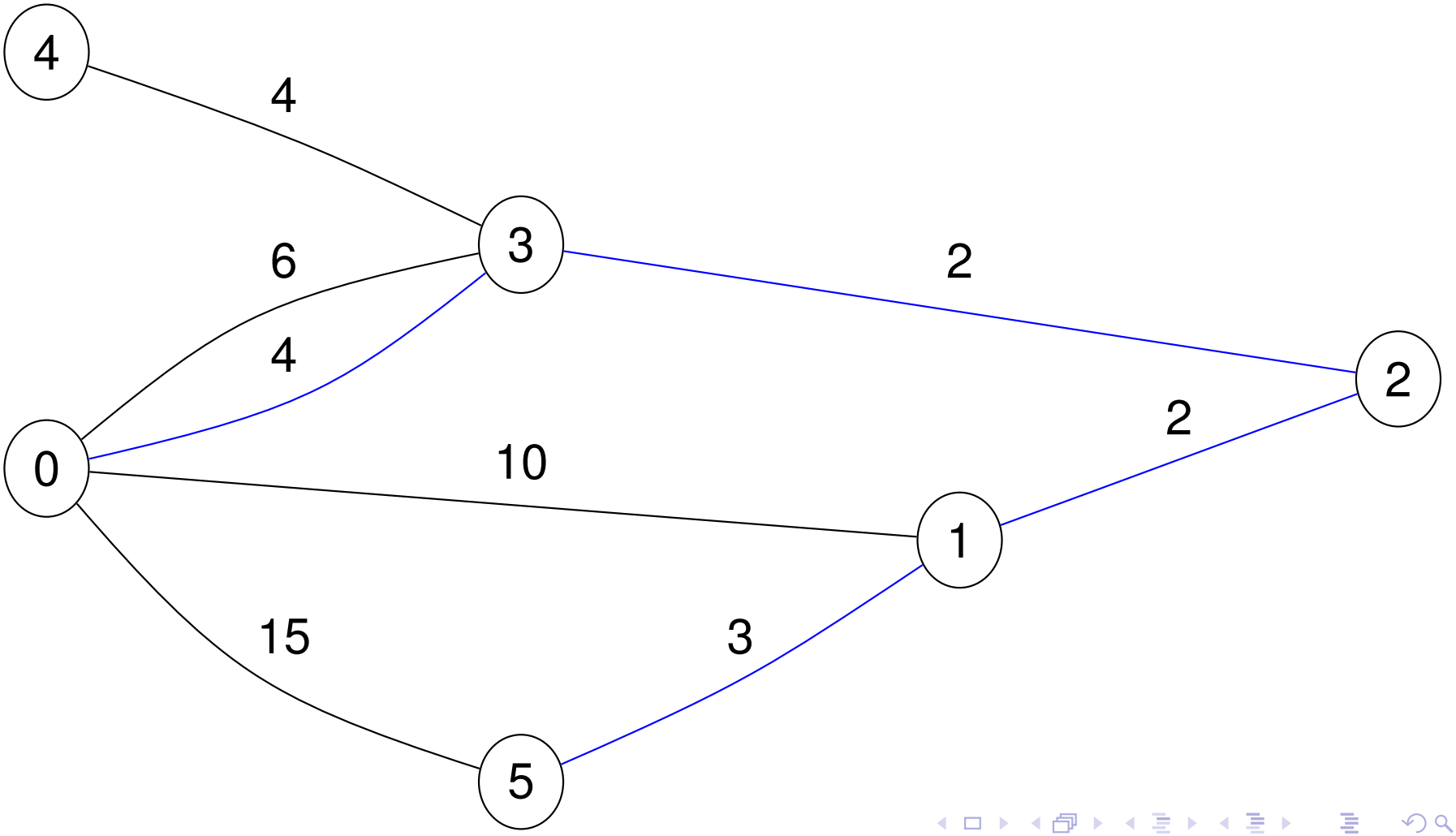
Cámino Mínimo

Dos de los problemas de grafos más estudiados son el la busqueda de cámino mínimo y el de cámino máximo.

El problema en general consiste en encontrar el camino $[v_0, v_1, \ldots, v_n]$ entre dos nodos i y j tal que $v_0 = i$ y $v_n = j$, tal que la distancia $d = \sum_{i=0}^{n} p_{v_i}$ sea lo menor (o lo mayor) posible.

Este problema se podía resolver fácilmente en un árbol o un grafo donde todas las aristas tengan el mismo peso, en tiempo lineal, usando BFS o DFS. Cuando usamos un grafo ponderado, es decir, donde las aristas pueden tener diferentes pesos, estos algoritmos no siempre dan una solución optima.

Camino mínimo $0 \longleftrightarrow 5 = 11$.



Contenidos

Algoritmo de Floyd-Warshall

Publicado por Robert Floyd y Stephen Warshall en 1962 (aunque ya se usaba desde bastante antes), el algoritmo sirve para encontrar el camino mínimo entre cada par de nodos en un grafo.

Como el algoritmo encuentra los N^2 caminos mínimos entre cada par de nodos y para la salida ya se usa $\mathcal{O}(N^2)$ de memoria, es ideal usar una matriz de adyacencia para representar el grafo.

Algoritmo de Floyd-Warshall

Publicado por Robert Floyd y Stephen Warshall en 1962 (aunque ya se usaba desde bastante antes), el algoritmo sirve para encontrar el camino mínimo entre cada par de nodos en un grafo.

Como el algoritmo encuentra los N^2 caminos mínimos entre cada par de nodos y para la salida ya se usa $\mathcal{O}(N^2)$ de memoria, es ideal usar una matriz de adyacencia para representar el grafo.

El algoritmo se basa en una técnica simple de programación dinámica: si se sabe el camino mínimo entre dos vértices i y j entre los caminos que solo pasen por los nodos $[i, k_1, k_2, \ldots, k_{n-1}, j]$, entonces en camino mínimo que pase por esos vértices y un vértice nuevo k_n puede ser:

- El mismo camino que ya teníamos, que pasaba solo por $[i, k_1, k_2, \ldots, k_{n-1}, j]$ y no pasaba por k_n.
- Un camino que sí o sí pase por k_n, cuya lóngitud va a ser la longitud del camino mínimo entre i y k_n más la longitud del camino mínimo entre k_n y j.

El algoritmo se basa en una técnica simple de programación dinámica: si se sabe el camino mínimo entre dos vértices i y j entre los caminos que solo pasen por los nodos $[i, k_1, k_2, \ldots, k_{n-1}, j]$, entonces en camino mínimo que pase por esos vértices y un vértice nuevo k_n puede ser:

- El mismo camino que ya teníamos, que pasaba solo por $[i, k_1, k_2, \ldots, k_{n-1}, j]$ y no pasaba por k_n.
- Un camino que sí o sí pase por k_n, cuya lóngitud va a ser la longitud del camino mínimo entre i y k_n más la longitud del camino mínimo entre k_n y j.

Para encontrar el camino mínimo entre cada par de nodos, simplemente tenemos que agregar todos los nodos en el grafo en la "lista" de nodos entre cada par de nodos. Como solo tenemos que guardar la longitud del camino mínimo y no importa el orden en que agreguemos cada k_i a la lista, podemos definir a $w_{i,j,k}$ como el camino mínimo entre i y j que pase por i, j, y todos los nodos en $[0, k)$. En ese caso,

$$w_{i,j,0} = \text{distancia entre i y j en el grafo}$$

$$w_{i,j,k} = min(w_{i,j,k-1}, w_{i,k,k-1} + w_{k,j,k-1})$$

$$w_{i,j,k} = min(w_{i,j,k-1}, w_{i,k,k-1} + w_{k,j,k-1})$$

Como la dinámica tiene $\mathcal{O}(|V|^3)$ estados, y cada estado tiene complejidad de tiempo $\mathcal{O}(1)$, la complejidad total del algoritmo es $\mathcal{O}(|V|^3)$.

Noten que no es necesario guardar los $\mathcal{O}(|V|^3)$ estados en la memoria, ya que una vez que cada parte del algoritmo solo accede a los elementos con k inmediatamente menor. Por lo tanto, solo hay que guardar una matriz por cada par de elementos conteniendo el camino mínimo entre estos dos elementos hasta ahora, con una complejidad de espacio igual a $\mathcal{O}(|V|^2)$

$$w_{i,j,k} = min(w_{i,j,k-1}, w_{i,k,k-1} + w_{k,j,k-1})$$

Como la dinámica tiene $\mathcal{O}(|V|^3)$ estados, y cada estado tiene complejidad de tiempo $\mathcal{O}(1)$, la complejidad total del algoritmo es $\mathcal{O}(|V|^3)$.

Noten que no es necesario guardar los $\mathcal{O}(|V|^3)$ estados en la memoria, ya que una vez que cada parte del algoritmo solo accede a los elementos con k inmediatamente menor. Por lo tanto, solo hay que guardar una matriz por cada par de elementos conteniendo el camino mínimo entre estos dos elementos hasta ahora, con una complejidad de espacio igual a $\mathcal{O}(|V|^2)$

El algoritmo se puede escribir de esta manera:

Data: w, la matriz de adyacencia del grafo
Result: $w_{i,j}$ es el camino mínimo entre i y j
for $k = 0 \rightarrow |V|$ **do**
  for $i = 0 \rightarrow |V|$ **do**
    for $j = 0 \rightarrow |V|$ **do**
      $w_{i,j} = min(w_{i,j}, w_{i,k} + w_{k,j})$
    end
  end
end

Aristas Negativas y camino máximo

Ya sabemos como encontrar el camino mínimo en un grafo.
¿Cómo encontramos el camino máximo?

¿No podemos multiplicar el largo todas las aristas por -1, y buscar el camino mínimo (o hacer las operaciones equivalentes sin cambiar los valores de las aristas?
Respuesta: Ni (que te da 0 puntos en la competencia).

Aristas Negativas y camino máximo

Ya sabemos como encontrar el camino mínimo en un grafo. ¿Cómo encontramos el camino máximo?

¿No podemos multiplicar el largo todas las aristas por -1, y buscar el camino mínimo (o hacer las operaciones equivalentes sin cambiar los valores de las aristas?

Respuesta: Ni (que te da 0 puntos en la competencia).

Aristas Negativas y camino máximo

Ya sabemos como encontrar el camino mínimo en un grafo. ¿Cómo encontramos el camino máximo?

¿No podemos multiplicar el largo todas las aristas por -1, y buscar el camino mínimo (o hacer las operaciones equivalentes sin cambiar los valores de las aristas?
Respuesta: Ni (que te da 0 puntos en la competencia).

Aristas Negativas y camino máximo

Ya sabemos como encontrar el camino mínimo en un grafo. ¿Cómo encontramos el camino máximo?

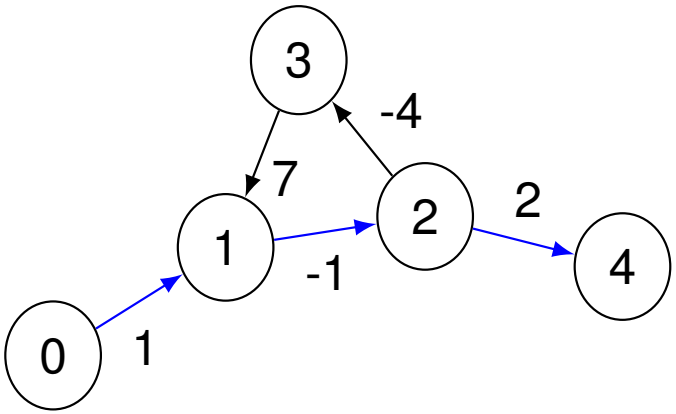
¿No podemos multiplicar el largo todas las aristas por -1, y buscar el camino mínimo (o hacer las operaciones equivalentes sin cambiar los valores de las aristas?
Respuesta: Ni (que te da 0 puntos en la competencia).

El problema es que, cuando buscar el camino mínimo en un grafo con aristas negativas (o el camino máximo en un grafo con aristas positivas), puede haber un ciclo negativo.

Camino mínimo = 2.

Camino mínimo = $-\infty$

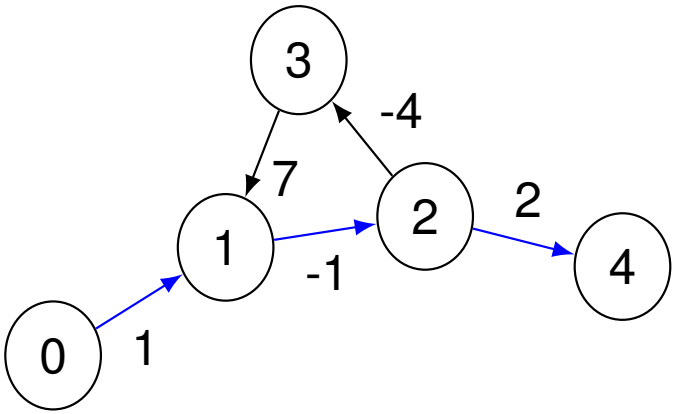
El problema es que, cuando buscar el camino mínimo en un grafo con aristas negativas (o el camino máximo en un grafo con aristas positivas), puede haber un ciclo negativo.



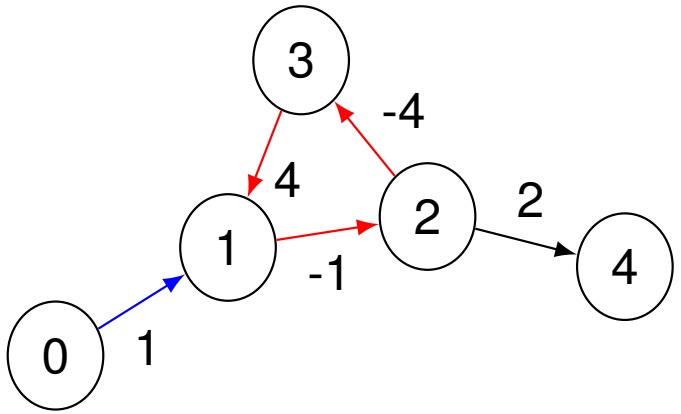
Camino mínimo = 2.

Camino mínimo = $-\infty$

El problema es que, cuando buscar el camino mínimo en un grafo con aristas negativas (o el camino máximo en un grafo con aristas positivas), puede haber un ciclo negativo.



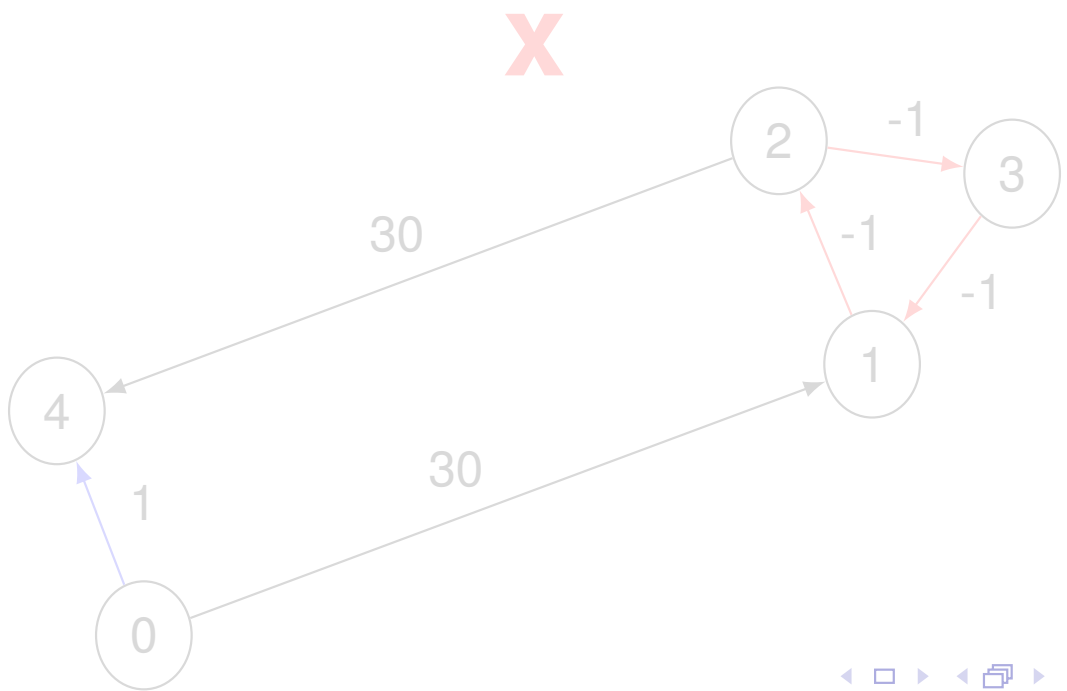
Camino mínimo = 2.



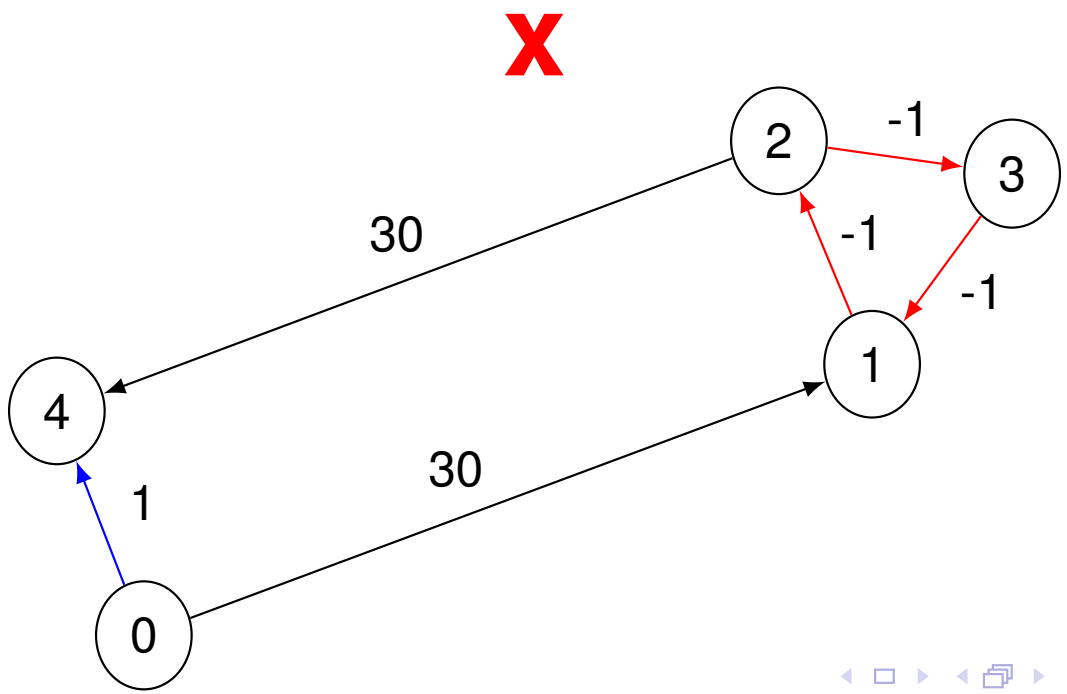
Camino mínimo = $-\infty$

Una manera de buscar el camino mínimo entre i y j es simple: primero buscar el camino mínimo entre todos los pares de nodos, después ver si hay un ciclo negativo que haha que no haya ningún camino mínimo finito entre i y j. Pero como encontrar lo segundo?

Proposición 1: Usar el algoritmo anterior para buscar el camino mínimo entre i y j, y después usarlo una vez más. Si hay un cambio la distancia entre i y j en la segunda vuelta eso significa que hay un camino entre i y j que pasa por más de $|V|$ nodos más corto que uno que pasa por solo $|V|$, por lo tanto hay un ciclo infinito.

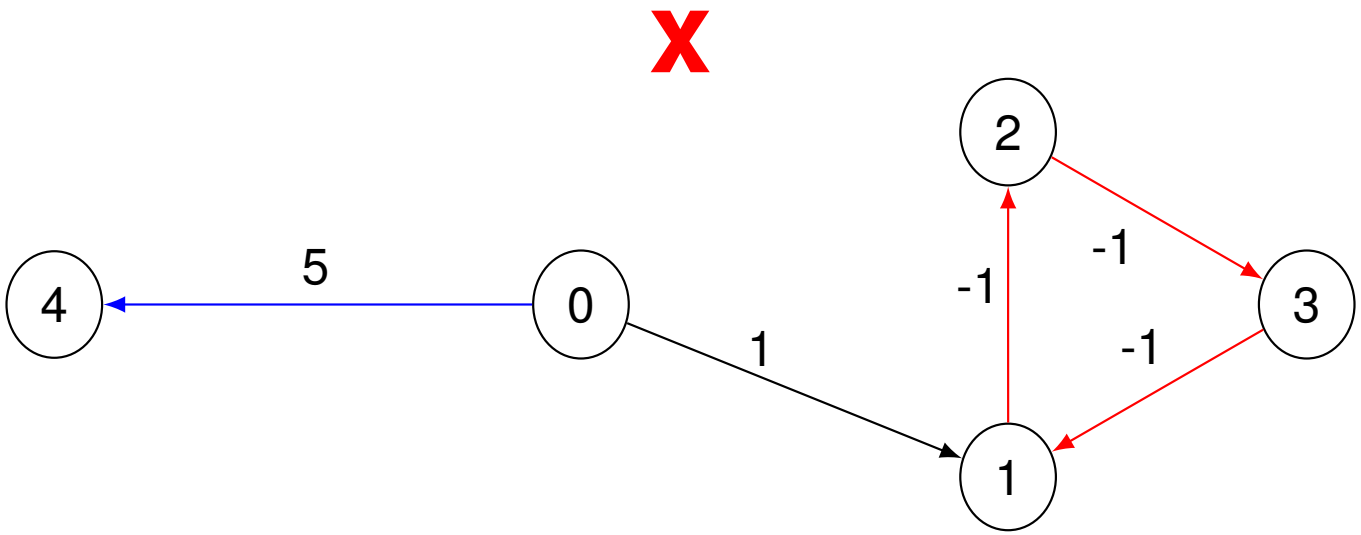


Proposición 1: Usar el algoritmo anterior para buscar el camino mínimo entre i y j, y después usarlo una vez más. Si hay un cambio la distancia entre i y j en la segunda vuelta eso significa que hay un camino entre i y j que pasa por más de $|V|$ nodos más corto que uno que pasa por solo $|V|$, por lo tanto hay un ciclo infinito.



Proposición 2: Bueno, entonces si hay algún ciclo negativo en el grafo no va a haber un camino mínimo finito entre cualquier par de nodos *i* y *j*.
El grafo tiene un ciclo infinito si y solo si, definiendo inicialmente la distancia de cualquier nodo *i* a si mismo como 0, después de usar Floyd-Warshall hay un nodo con distancia hacia si mismo menor que 0.

Proposición 2: Bueno, entonces si hay algún ciclo negativo en el grafo no va a haber un camino mínimo finito entre cualquier par de nodos *i* y *j*.
El grafo tiene un ciclo infinito si y solo si, definiendo inicialmente la distancia de cualquier nodo *i* a si mismo como 0, después de usar Floyd-Warshall hay un nodo con distancia hacia si mismo menor que 0.



Proposición 3: Hagamos lo mismo que en la proposición anterior, pero esta vez hagamos un DFS entre i y j, y decidamos que no hay ningún camino mínimo finito entre i y j si hay por lo menos un camino entre i y j que pase por un nodo en un ciclo negativo, eso es, que la distancia entre ese nodo y si mismo sea menor a 0.

Proposición 3: Hagamos lo mismo que en la proposición anterior, pero esta vez hagamos un DFS entre *i* y *j*, y decidamos que no hay ningún camino mínimo finito entre *i* y *j* si hay por lo menos un camino entre *i* y *j* que pase por un nodo en un ciclo negativo, eso es, que la distancia entre ese nodo y si mismo sea menor a 0.

Contenidos

Algoritmo de Dijkstra

Publicado por Edsger Dijkstra en 1959, el Algoritmo de Dijkstra calcula la distancia mínima desde un nodo hasta cualquier otro nodo en el grafo.

Para lograrlo, se usa un algoritmo similar a BFS o DFS, que se usan para encontrar el camino mínimo en un grafo con aristas sin pesos, llamado Priority First Search, o PFS.

Priority First Search

Recordemos el código que se usó para definir BFS y DFS:

```
visitados ← ∅;
proximo ← {X_0};
while proximo not empty do
    nodo ← proximo.pop() ; // Sacar un elemento de proximo
    if nodo ∈ visitados then
        continue ; // Descartar este nodo
    end
    visitados ← visitados ∪ {nodo};

    for siguiente ∈ adyacentes(nodo) do
        proximo ← proximo ∪ {siguiente};
    end
end
```

De la misma manera que BFS definía a proximo con una **cola**, donde *pop* saca el elemento que se puso hace más tiempo, y DFS lo definía con una **pila**, donde *pop* saca el último que se puso, PFS usa una **cola de prioridad**, o **priority queue**, donde *pop* saca al elemento que minimize una función.

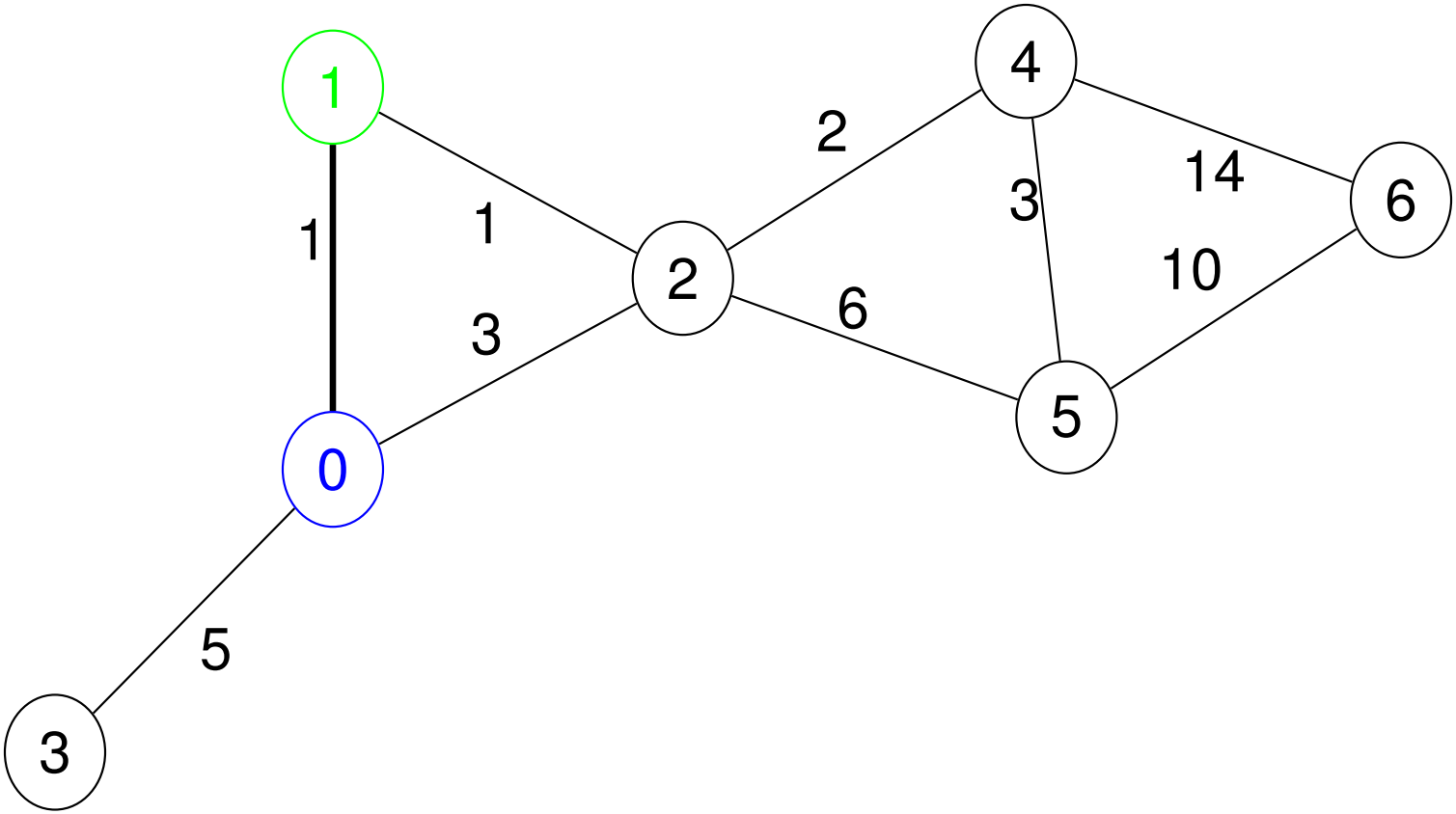
Hay que tener en cuenta que, a diferencia de BFS y DFS, yo puedo llegar a tener que "actualizar" el valor de un nodo dentro de *proximo*, ya que puedo llegar a encontrar un mejor camino para llegar a un nodo al que todavía no visité. Una posible modificación an algoritmo en ese caso es insertar aristas (o, más simplemente, pares $(distancia, proximo)$) a *proximo*, así no hay que preocuparse por actualizar los valores.

En el caso de Dijkstra, una función que nos conviene minimizar es distancia al nodo inicial.

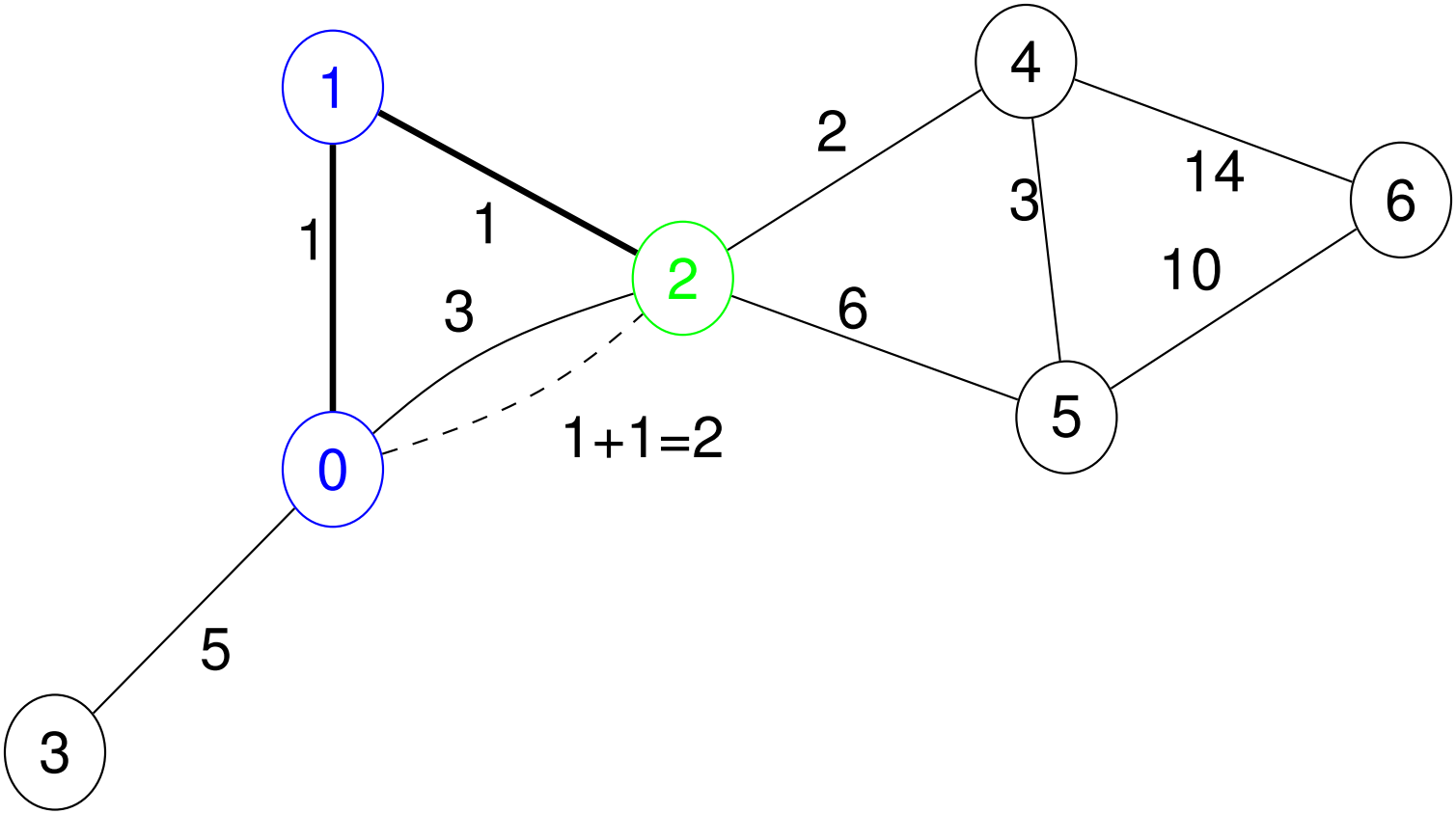
Algoritmo de Dijkstra

Una linda propiedad que sirve para que el algoritmo de Dijkstra funcione es que, dado tres nodos i, k, y j, el camino mínimo entre i y j que pase por k va a ser igual al camino mínimo entre i y k, seguido del camino mínimo entre k y j.

Por esta propiedad, en un grafo sin aristas negativas, el camino mínimo entre un nodo i y su nodo adyacente con distancia mínimal j (es decir, que no existe ninguna arista limitrofe a i con distancia menor a la arista que conecta i con j) no va a tener ninguna arista intermedia, osea que va a ser nada más la arista que conecta i con j.


1
4
2
14
6
1
1
3
10
2
6
3
5
0
5
3

Por la misma razón, si se conoce el camino mínimo de un nodo i a un grupo de nodos $[i, k_1, k_2, \ldots, k_n]$, el camino mínimo desde el nodo j adyacente al grupo con distancia a i minimal no va a pasar por ningún otro nodo.

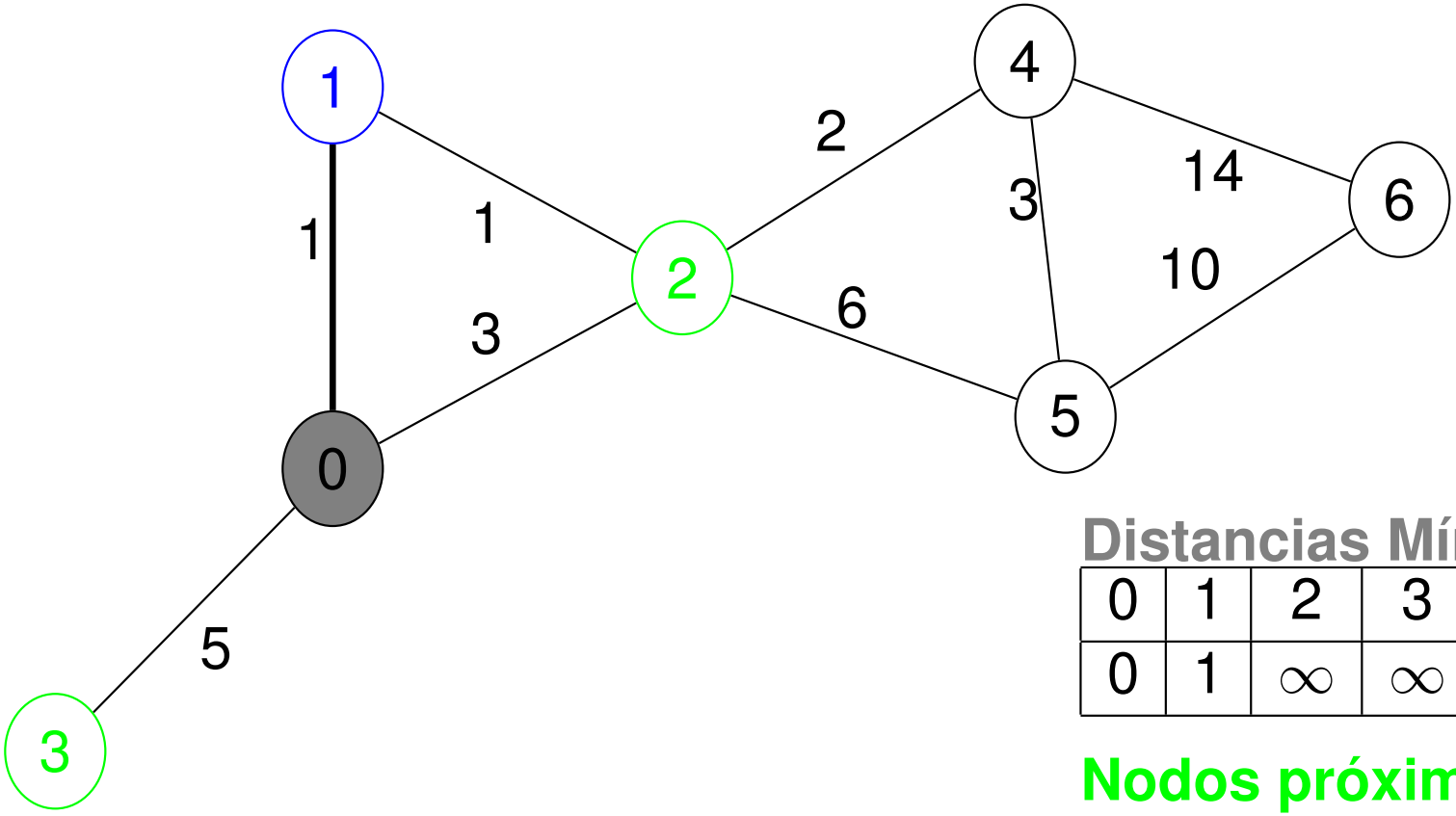

1
4
2
14
6
1
3
1
10
2
3
6
1+1=2
5
0
5
3

Esto se puede seguir hasta encontrar el camino mínimo desde *i* hasta cualquier otro nodo. Las preguntas obvias son, ¿Cómo guardar los valores de alguna manera que encontrar el menor sea efectivo? ¿Cómo manejar nodos “repetidos” bien?

La respuesta es simple: usar Priority First Search, usando la distancia hasta *i* como función que la priority queue usa para elegir el próximo nodo.

Esto se puede seguir hasta encontrar el camino mínimo desde *i* hasta cualquier otro nodo. Las preguntas obvias son, ¿Cómo guardar los valores de alguna manera que encontrar el menor sea efectivo? ¿Cómo manejar nodos “repetidos” bien?

La respuesta es simple: usar Priority First Search, usando la distancia hasta *i* como función que la priority queue usa para elegir el próximo nodo.

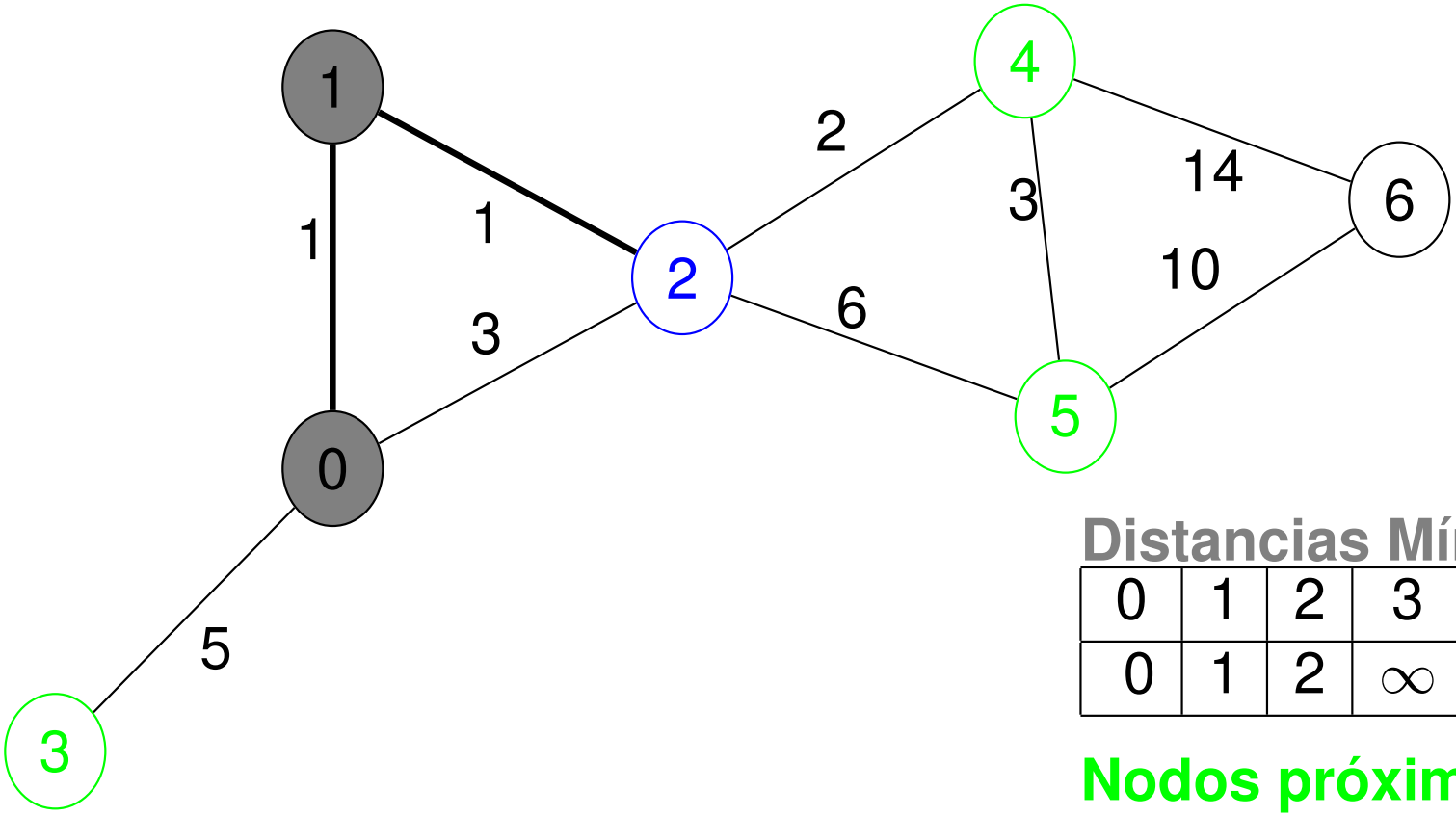


Distancias Mínimas

0	1	2	3	4	5	6
0	1	∞	∞	∞	∞	∞

Nodos próximos

2	3
2	5

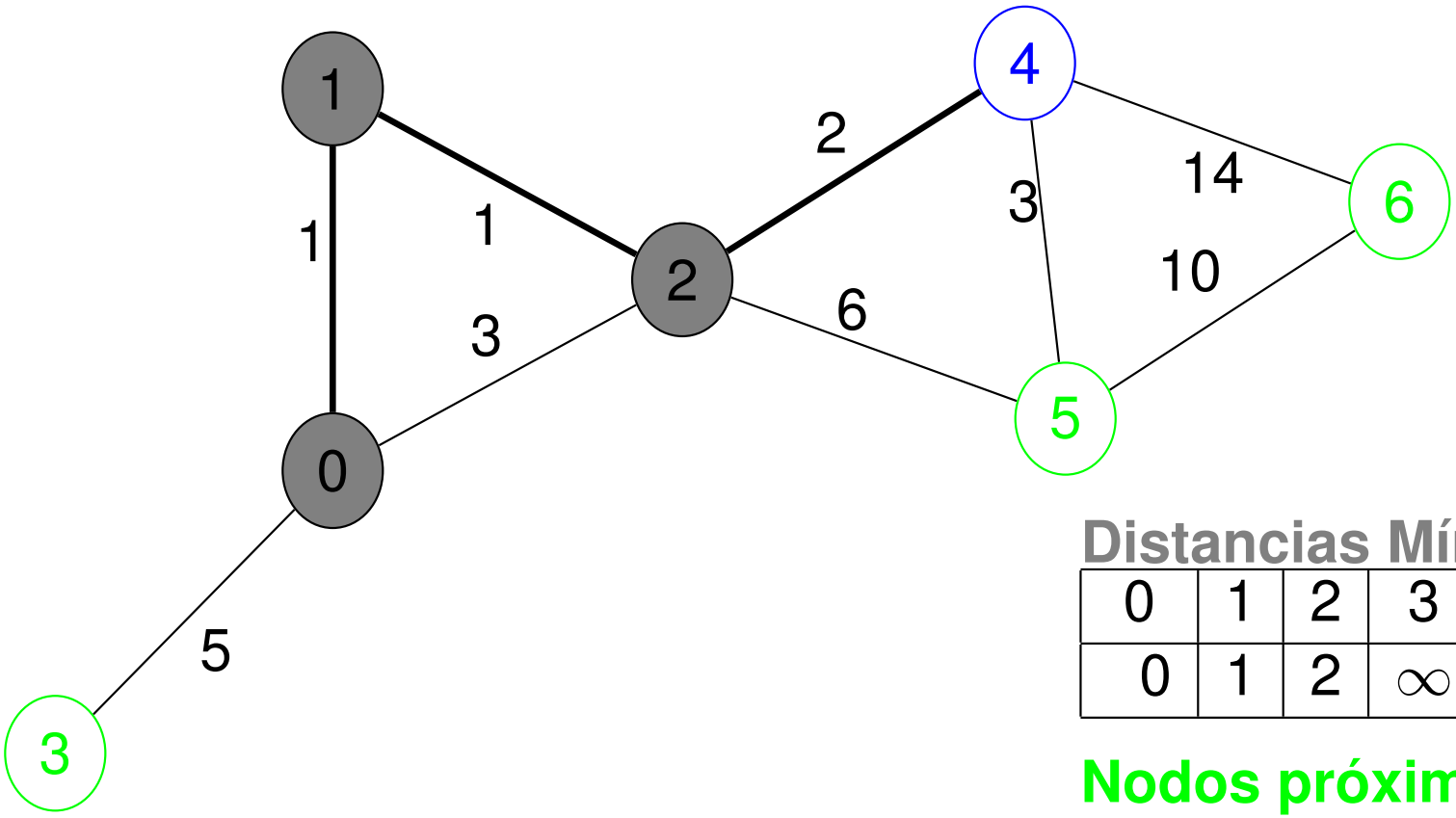


Distancias Mínimas

0	1	2	3	4	5	6
0	1	2	∞	∞	∞	∞

Nodos próximos

4	3	5
4	5	8

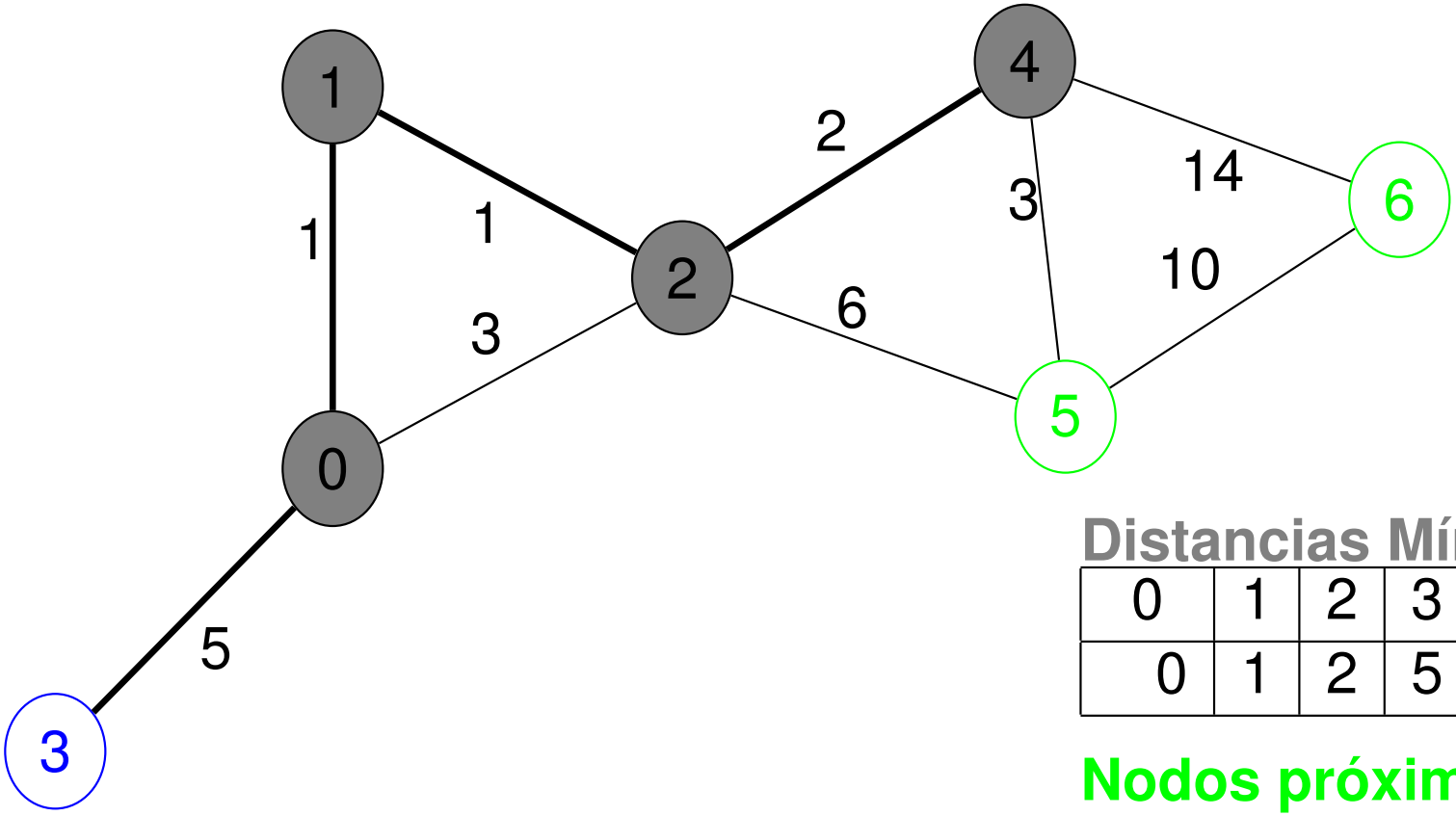


Distancias Mínimas

0	1	2	3	4	5	6
0	1	2	∞	4	∞	∞

Nodos próximos

3	5	6
5	7	18

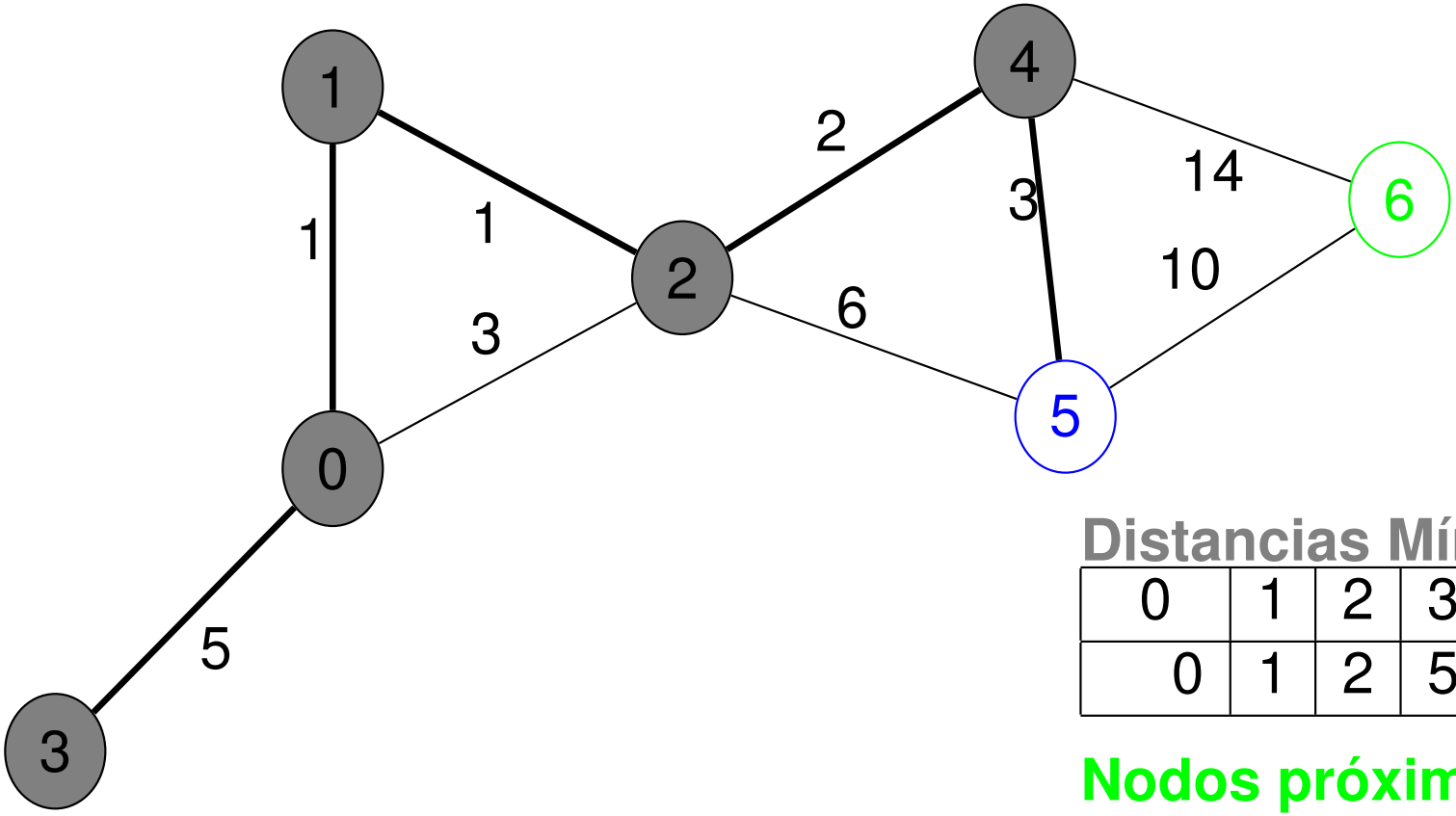


Distancias Mínimas

0	1	2	3	4	5	6
0	1	2	5	4	∞	∞

Nodos próximos

5	6
7	18

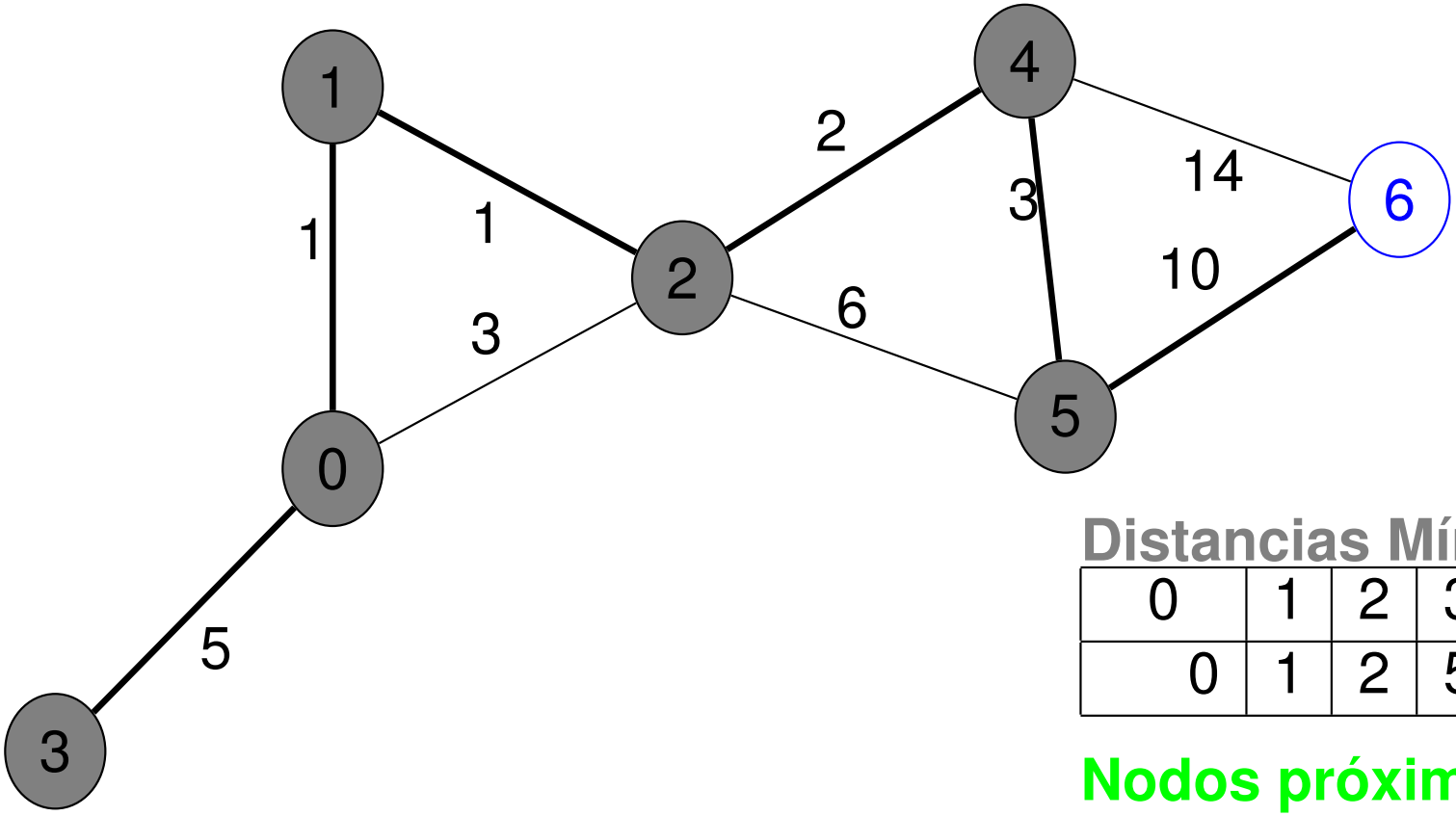


Distancias Mínimas

0	1	2	3	4	5	6
0	1	2	5	4	7	∞

Nodos próximos

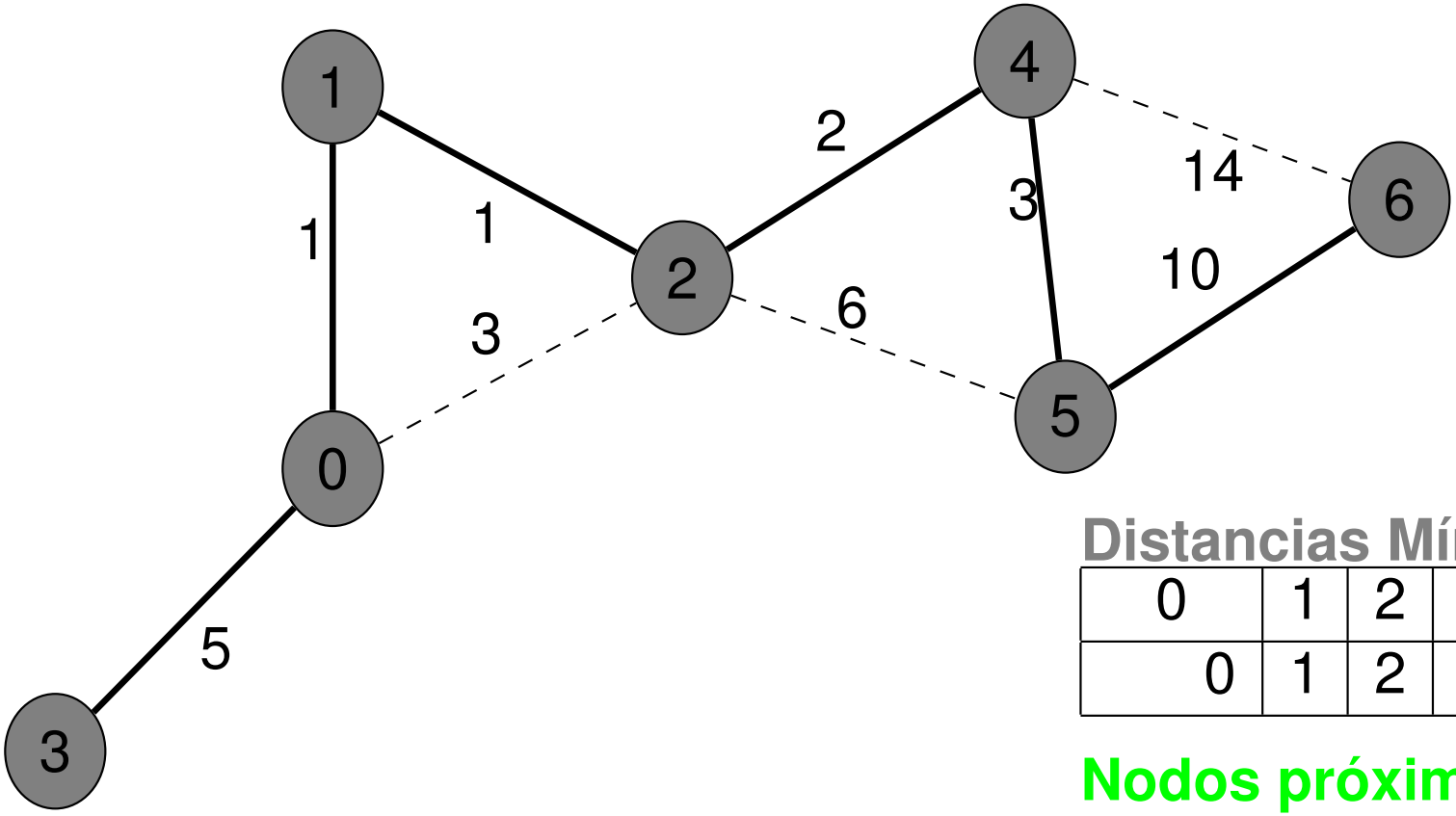
6
17



Distancias Mínimas

0	1	2	3	4	5	6
0	1	2	5	4	7	17

Nodos próximos



Distancias Mínimas

0	1	2	3	4	5	6
0	1	2	5	4	7	17

Nodos próximos

Análisis de complejidad

Como con BFS y DFS, el Priority First Search que usamos en el algoritmo de Dijkstra hace los siguientes pasos:

Mientras queden nodos,

1. Elegir un nodo, y descartarlo si ya fué visitado.
2. Buscar todos sus vecinos, e insertarlos a *proximo*.

A diferencia de BFS y DFS, elegis un nodo e insertar a un vecino a *proximo* pueden tener una complejidad mayor a $\mathcal{O}(1)$ dependiendo de la estructura que se use para guardar los próximos nodos.

Análisis de complejidad

Como con BFS y DFS, el Priority First Search que usamos en el algoritmo de Dijkstra hace los siguientes pasos:

Mientras queden nodos,

1. Elegir un nodo, y descartarlo si ya fué visitado.
2. Buscar todos sus vecinos, e insertarlos a *proximo*.

A diferencia de BFS y DFS, elegis un nodo e insertar a un vecino a *proximo* pueden tener una complejidad mayor a $\mathcal{O}(1)$ dependiendo de la estructura que se use para guardar los próximos nodos.

Usando un arreglo normal, y haciendo una busqueda lineal para encontrar el próximo nodo con distancia mínima, la complejidad en tiempo es

$$\mathcal{O}(\sum_{i \in V}(|V|+gr(i))) = \mathcal{O}(|V|^2+\sum_{i \in V} gr(i)) = \mathcal{O}(|V|^2+|E|) = \mathcal{O}(|V|^2)$$

Una manera de mejorar esta complejidad es usando una estructura de datos, llamada **heap** o parva, donde elegir y remover el nodo que minimize una función tiene una complejidad en tiempo de $\mathcal{O}(logn)$, a cambio de que agregar un nodo también sea $\mathcal{O}(logn)$ en el peor caso.
Usando un heap, la complejidad del algoritmo de Dijkstra termina siendo

$$\mathcal{O}(\sum_{i \in V}(log|V|+gr(i))) = \mathcal{O}(|V|log|V|+\sum_{i \in V} gr(i)) = \mathcal{O}(|V|log|V|+|E|)$$

El heap está implementado en la STL de C++ como *std::priority_queue<T>*, y en Java como *java.util.PriorityQueue<E>*.

Una manera de mejorar esta complejidad es usando una estructura de datos, llamada **heap** o parva, donde elegir y remover el nodo que minimize una función tiene una complejidad en tiempo de $\mathcal{O}(logn)$, a cambio de que agregar un nodo también sea $\mathcal{O}(logn)$ en el peor caso.
Usando un heap, la complejidad del algoritmo de Dijkstra termina siendo

$$\mathcal{O}(\sum_{i \in V}(log|V|+gr(i))) = \mathcal{O}(|V|log|V|+\sum_{i \in V} gr(i)) = \mathcal{O}(|V|log|V|+|E|)$$

El heap está implementado en la STL de C++ como *std::priority_queue<T>*, y en Java como *java.util.PriorityQueue<E>*.