Grafos

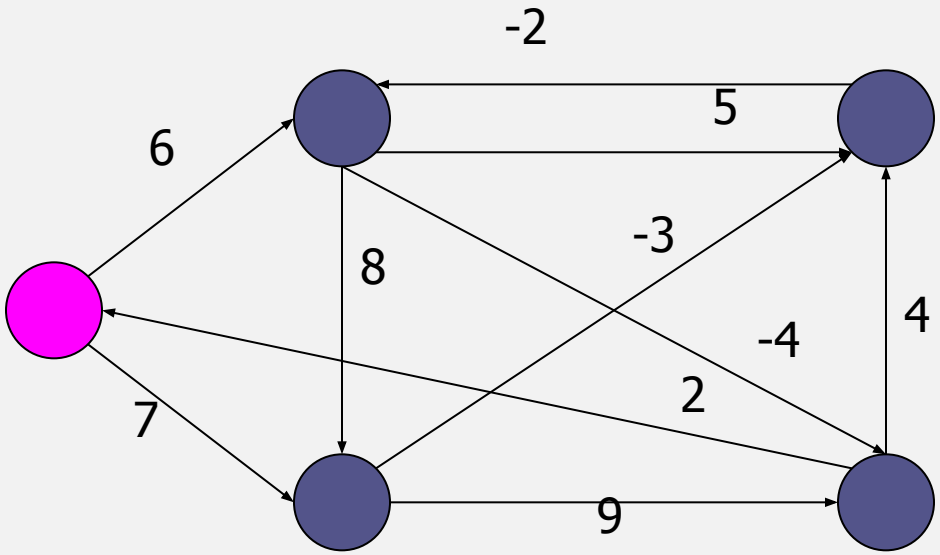
- Caminos mínimos
  - Algoritmo para grafos sin peso
  - Algoritmo de Dijkstra
  - Algoritmo de Bellman-Ford
  - Algoritmo de Floyd
- Árbol de expansión mínima
  - Algoritmo de Kruskal
  - Aplicaciones

Caminos mínimos

- Algoritmo para grafos sin peso
- Algoritmo de Dijkstra
- Algoritmo de Bellman-Ford
- Algoritmo de Floyd

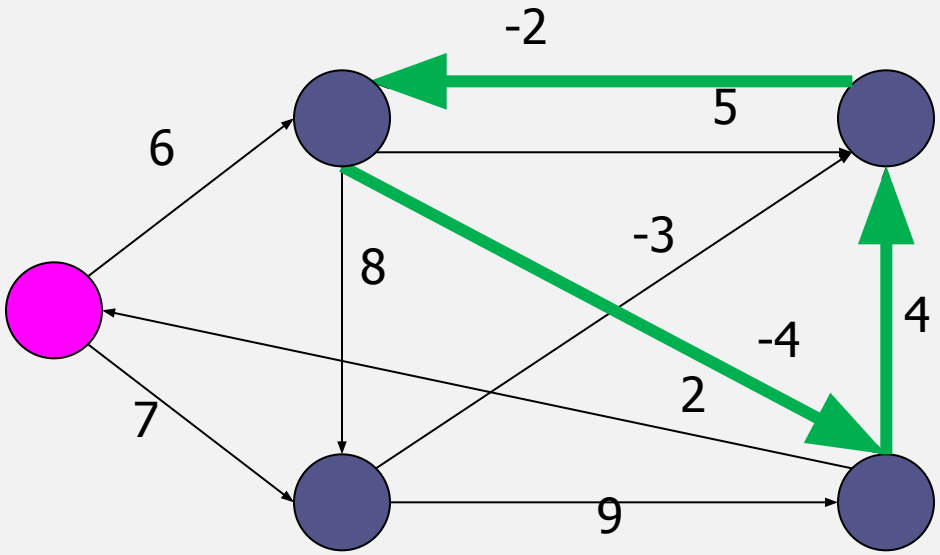
Camino mínimo

- $\delta(u,v) = \min \{w(p): p \text{ es un camino de } u \text{ a } v\}$
- $\delta(u,v) = \infty$ si no hay camino de u a v
- $\delta(u,v) = -\infty$ si hay un camino de u a v, que visita un ciclo negativo



Camino mínimo

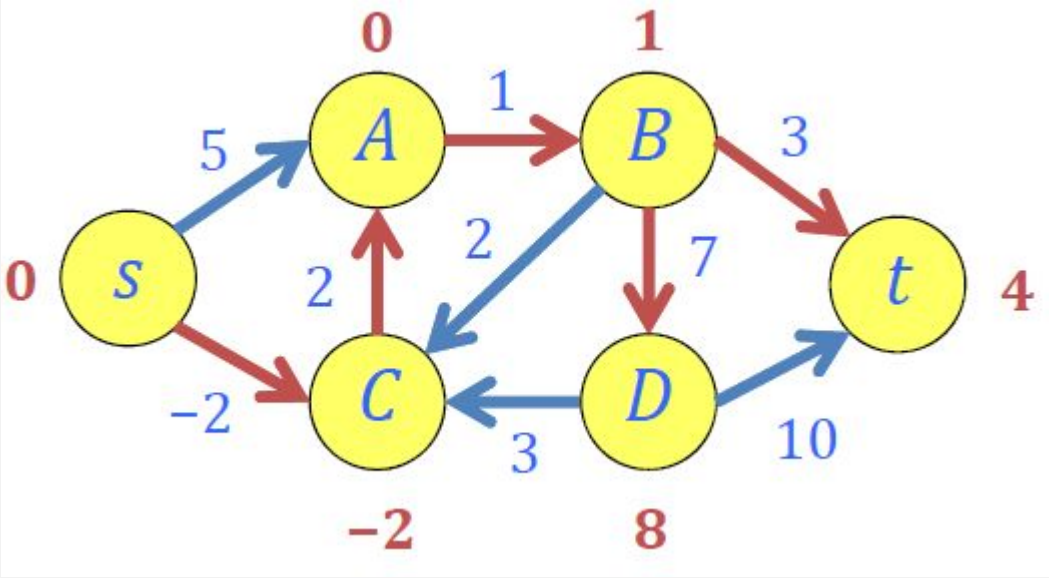
- $\delta(u,v) = \min \{w(p): p \text{ es un camino de } u \text{ a } v\}$
- $\delta(u,v) = \infty$ si no hay camino de u a v
- $\delta(u,v) = -\infty$ si hay un camino de u a v, que visita un ciclo negativo



Caminos mínimos desde un origen (SSSP)

Dado un grafo G=(V,E) con una función que representa el peso de las aristas w : E $\rightarrow \mathbb{R}$ y un vértice origen s, calcular δ(s,v) para todo v de V

Esto genera un árbol de caminos mínimos.



Algoritmo para grafos sin peso

- El algoritmo se basa en BFS $\Longrightarrow O(|V| + |E|)$
- 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

Algoritmo para grafos sin peso

- 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.

Algoritmo para grafos sin peso

```
BFS(nodoInicial) {
    queue cola;
    vector<int> distancias; //inicializado en ∞, puede ser n
    cola.push(nodoInicial) ;
    distancias[nodoInicial] = 0;
    mientras( !cola.empty() ) {
      t = cola. front () ;
      cola.pop() ;
      para cada w adyacente a t
            if (distancias[w]=n) {
                distancias[w] = distancias[t]+1;
                cola.push(w) ;
            }
    }
   return distancias;
}
```

Algoritmo de Dijkstra

Pasos:

- Dado un vértice origen s, elegir el vértice v que esté a la menor distancia de s, dentro de los vértices no procesados
- Marcar v como procesado
- Actualizar la distancia de w adyacente a v

  La actualización de la distancia de los adyacentes w se realiza con el siguiente criterio:

  Se compara D_w con $D_v + c(v,w)$

  Distancia de s a w (sin pasar por v)

  Distancia de s a w, pasando por v

  Se actualiza si $D_w > D_v + c(v,w)$

Algoritmo de Dijkstra

- Para cada vértice v mantiene la siguiente información:
  - Dv : distancia mínima desde el origen (inicialmente ∞ para todos lo vértices excepto el origen con valor 0)
  - Pv : vértice por donde paso para llegar
  - Conocido : dato booleano que me indica si está procesado (inicialmente todos en 0)

Algoritmo de Dijkstra

```
Dijkstra(G,w, s) {
    para cada vértice v Є V
        Dv = ∞;   Pv = 0;

    Ds = 0;
    para cada vértice v Є V {
        u = vérticeDesconocidoMenor Dist;
        Marcar u como conocido;
        para cada vértice w Є V adyacente a u
          si w no está conocido entonces
            si Dw > Du + c(u,w) entonces {
                Dw = Du + c(u,w);
                Pw = u;
            }
    }
}
```

Algoritmo de Dijkstra

- Hay dos versiones:
  - Dijkstra sin cola de prioridad → $O(|V|^2 + |E|)$

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 → $O(|E| \log |V|)$

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.

Algoritmo de Bellman-Ford

- Mecanismo de relajación
- Algoritmo de Bellman-Ford
- Detección de ciclos negativos
- Análisis del tiempo de ejecución

Algoritmo de Bellman-Ford: **Mecanismo de relajación**

para v en *V*:

    v.d = ∞

    v.p = nulo

s.d = 0

mientras alguna arista (u,v) tenga v.d > u.d + w(u,v)

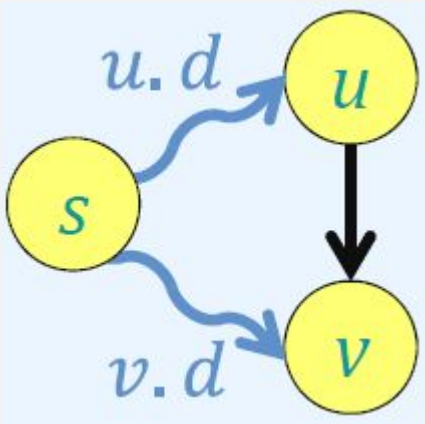
    tomar la arista (u,v)

    relajar (u,v):

        si v.d > u.d + w(u,v):

            v.d = u.d + w(u,v)

            v.p = u



Algoritmo de Bellman-Ford

- Mecanismo de relajación
- Aristas: $e_1, e_2, \ldots\ldots e_m$
- Relajar en este orden:

$e_1, e_2, \ldots\ldots e_m$; $e_1, e_2, \ldots\ldots e_m$; …. ….; $e_1, e_2, \ldots\ldots e_m$

$|V|$ - 1 repeticiones

Algoritmo de Bellman-Ford

para v en *V*:

    v.d = ∞

    v.p = nulo

s.d = 0

para *i* desde 1 hasta |V| -1:

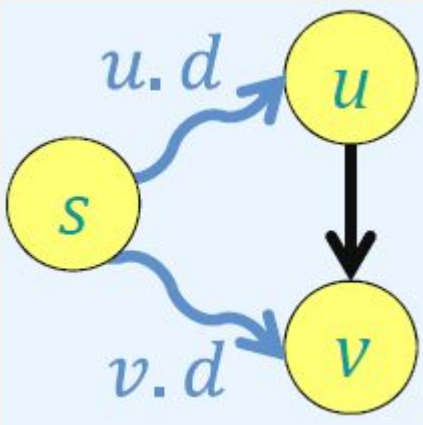
    para (u,v) en *E*:

    relajar (u,v):

        si v.d > u.d + w(u,v):

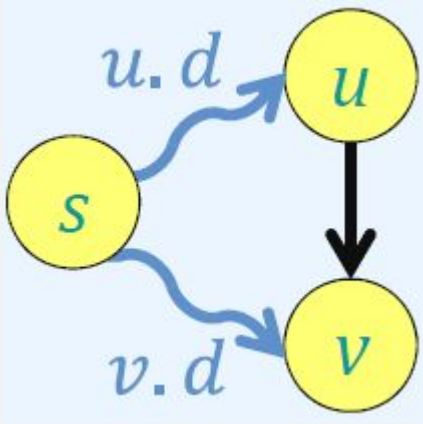
            v.d = u.d + w(u,v)

            v.p = u



Algoritmo de Bellman-Ford con detección de ciclos negativos

para v en *V*:
    v.d = ∞
    v.p = nulo
s.d = 0
para *i* desde 1 hasta |V| -1:
    para (u,v) en *E*:
    relajar (u,v)
para (u,v) en *E*:
    si v.d > u.d + w(u,v):
    se encontró un ciclo negativo



Si hay un ciclo negativo, alguna arista siempre puede relajarse

Algoritmo de Bellman-Ford: Análisis del tiempo de ejecución

```
para v en V:
    v.d = ∞
    v.p = nulo
s.d = 0
para i desde 1 hasta |V|-1:
    para (u,v) en E:
    relajar (u,v)
para (u,v) en E:
    si v.d > u.d + w(u,v):
    se encontró un ciclo negativo
```

O(|V|)

O(1)

O(|E|)

O(|V||E|)

O(|E|)

Tiempo total: O(|V||E|)

Determinar δ(s,v)

```
para v en V:
    v.d = ∞
    v.p = nulo
s.d = 0
para i desde 1 hasta |V|-1:
    para (u,v) en E:
    relajar (u,v)
para (u,v) en E:
        si v.d > u.d + w(u,v):
        v.d = -∞
        v.p = u
```

- Al setear v.d = -∞, se limita la relajación
- Los caminos desde un vértice u a cualquier vértice x con δ(s,x) = -∞ tiene a lo sumo |V| aristas

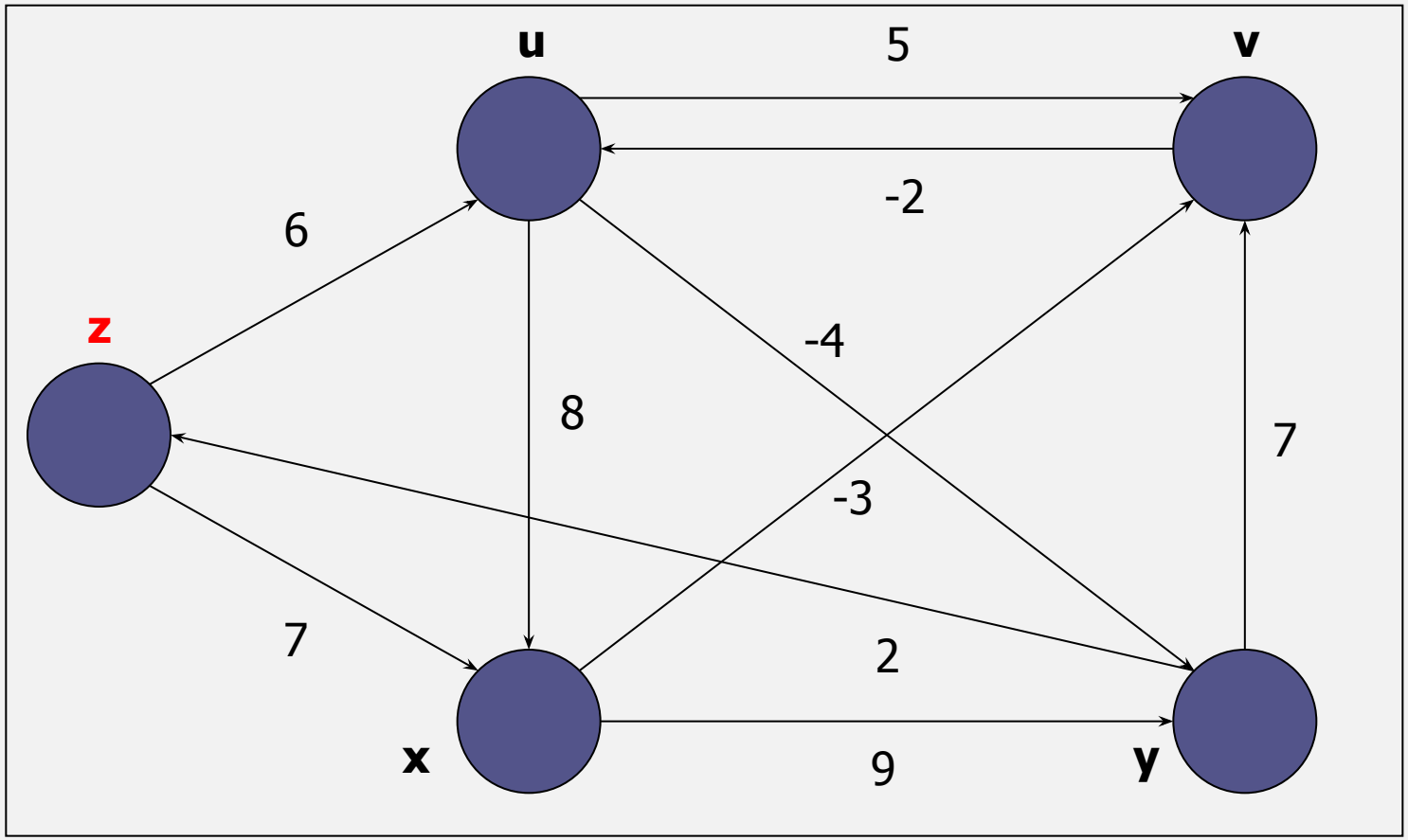
Algoritmo de Bellman-Ford

- El algoritmo de Bellman-Ford itera hasta V-1 veces sobre cada arista. Si moverse por esa arista disminuye la distancia desde el nodo inicial hasta el nodo hacia el cuál nos movemos a través de esa arista, entonces actualiza la distancia por la que obtenemos a través de la nueva arista.

- Esto funciona porque un camino mínimo tiene como máximo V-1 aristas, entonces en el i-ésimo paso actualizamos la distancia al (i + 1)-ésimo nodo del camino mínimo.

- Si luego de los V-1 pasos podemos seguir disminuyendo la distancia a un nodo quiere decir que hay un camino de longitud al menos V que asigna una menor distancia que todos los caminos más chicos, pero este camino contiene un ciclo, luego hay un ciclo negativo.

- Si hay un ciclo negativo entonces el camino mínimo a los nodos a los que se puede llegar desde un nodo del ciclo no existe porque hay caminos arbitrariamente chicos.



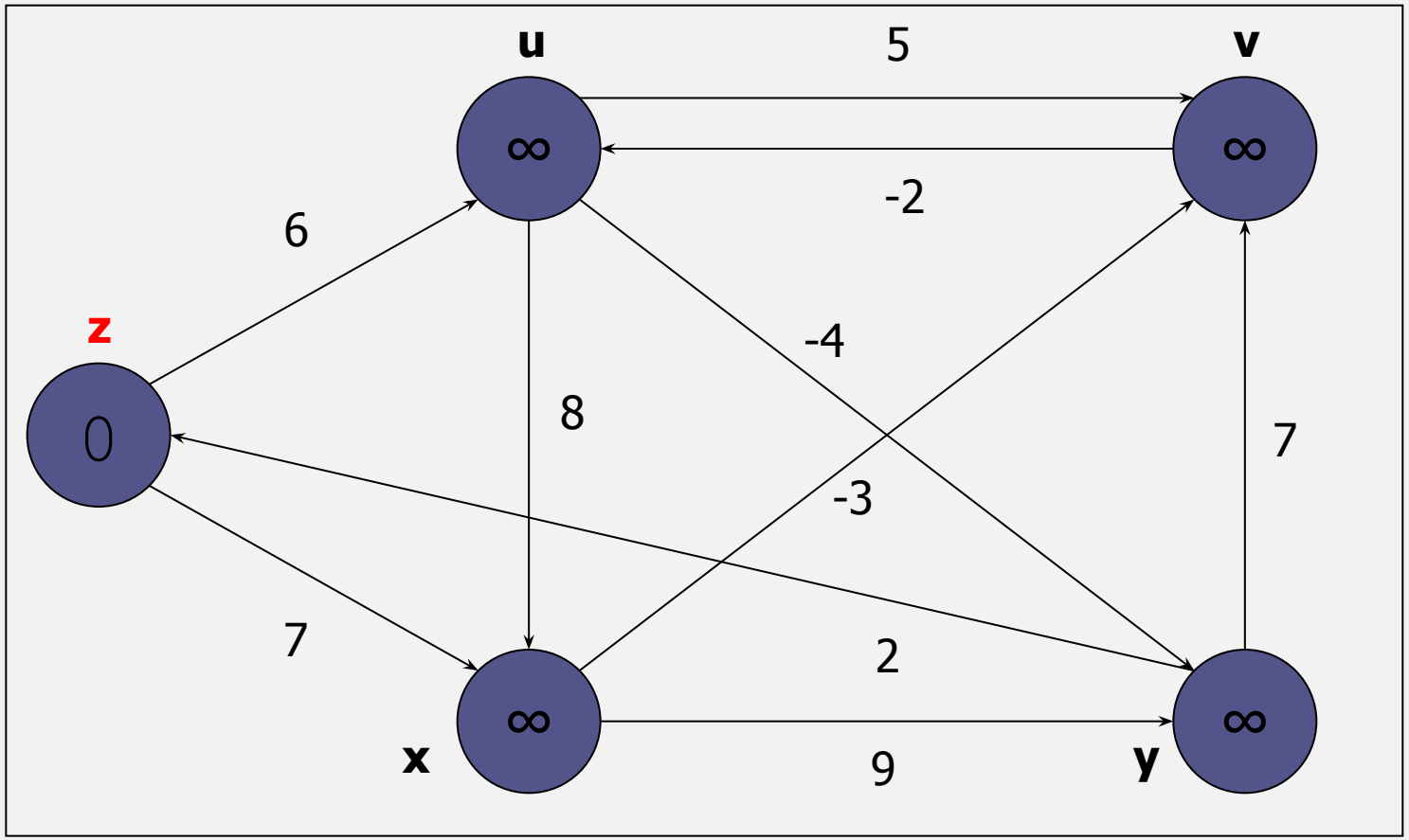
Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	_	_	_	_	_	**}**
Π	**[**	**]**	**=**	**{**	_	_	_	_	_	**}**

Paso 0.0

Encontrar el camino más corto del vértice **z** a cada uno de los otros vértices.



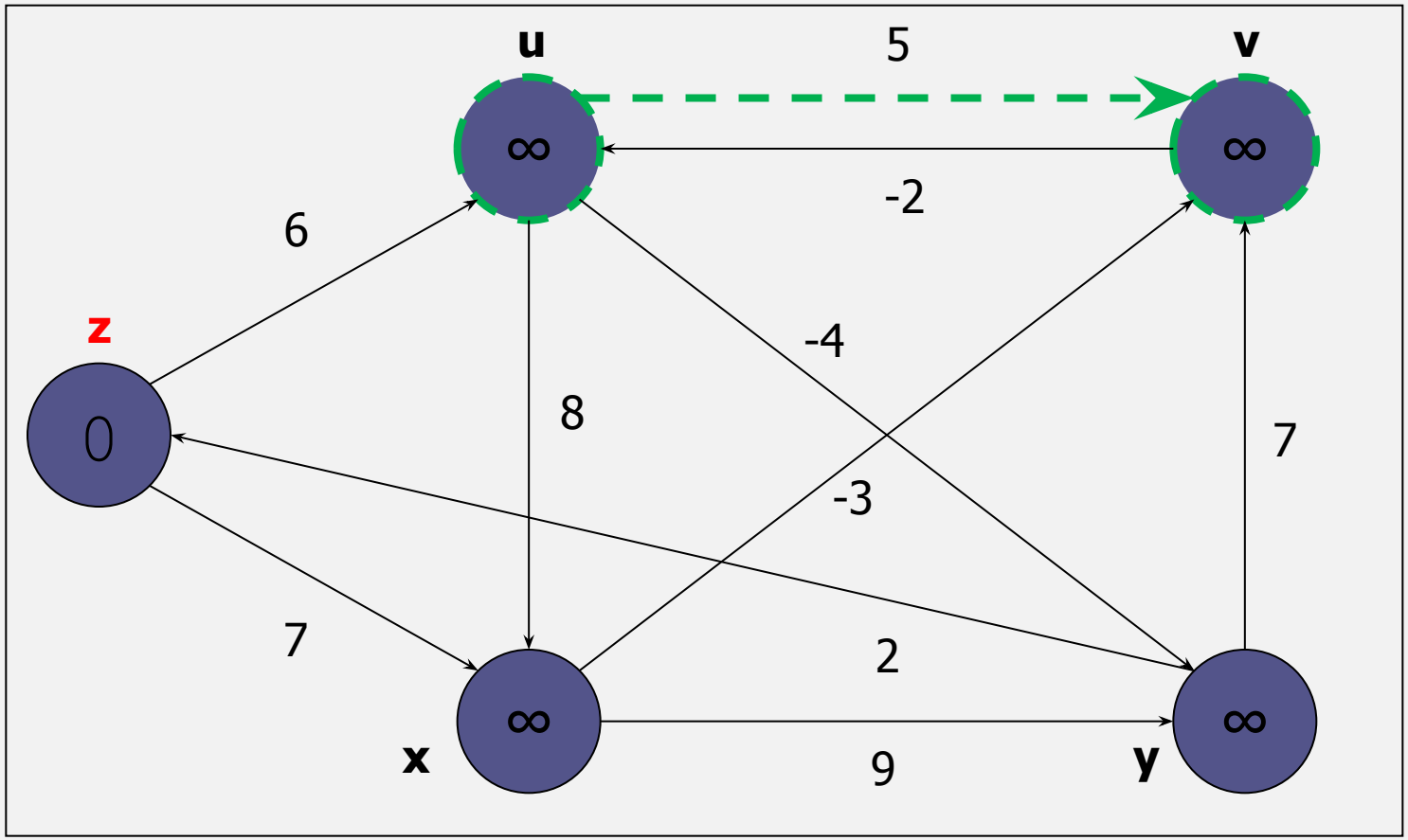
Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	**∞**	**∞**	**∞**	**∞**	**0**	**}**
Π	**[**	**]**	**=**	**{**						**}**

Paso 0.1

Inicializar los vectores d y Π.



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	[	]	=	{	u	v	x	y	z	}
d	[	]	=	{	∞	∞	∞	∞	0	}
Π	[	]	=	{						}

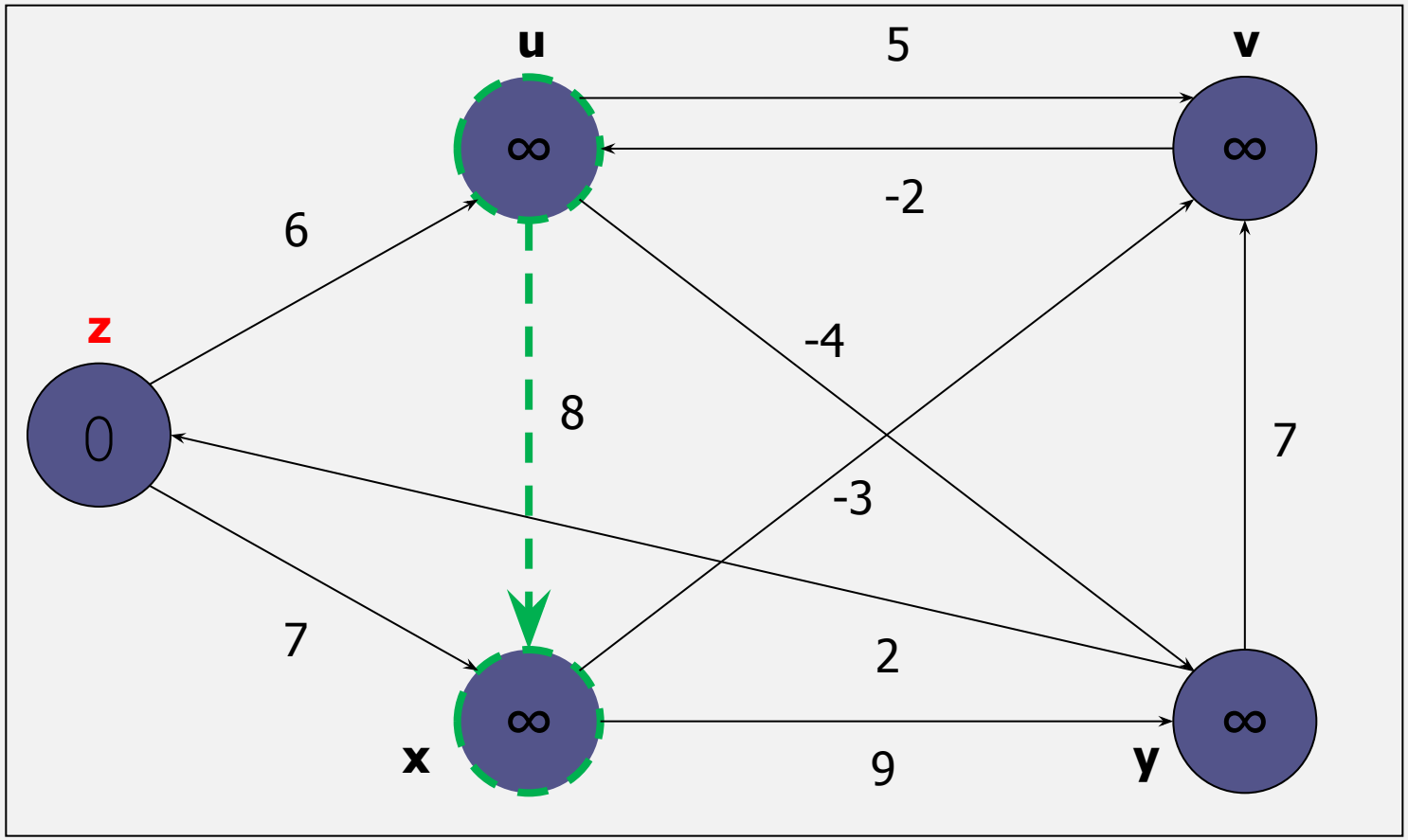
Paso 1.1

Aplicar Relax al Arco (u,v)

Pregunta: ¿ d[v] > d[u] + w(u , v) ?

Respuesta: NO

Proceso: No se hace nada.



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	**∞**	**∞**	**∞**	**∞**	**0**	**}**
Π	**[**	**]**	**=**	**{**						**}**

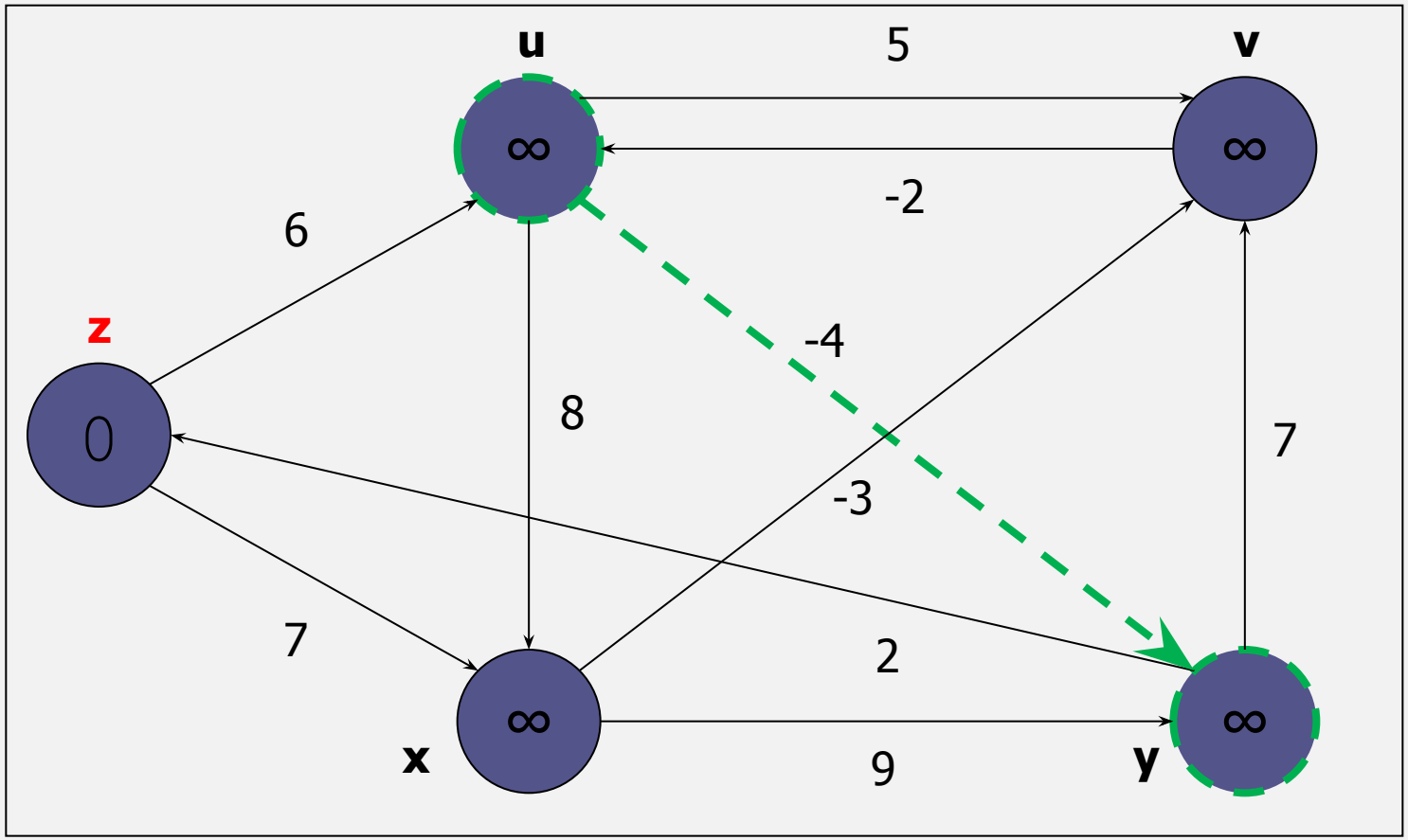
Paso 1.2

Aplicar Relax al Arco (u,x)

Pregunta: ¿ d[x] > d[u] + w(u , x) ?

Respuesta: NO

Proceso: No se hace nada.



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	[	]	=	{	u	v	x	y	z	}
d	[	]	=	{	∞	∞	∞	∞	0	}
Π	[	]	=	{						}

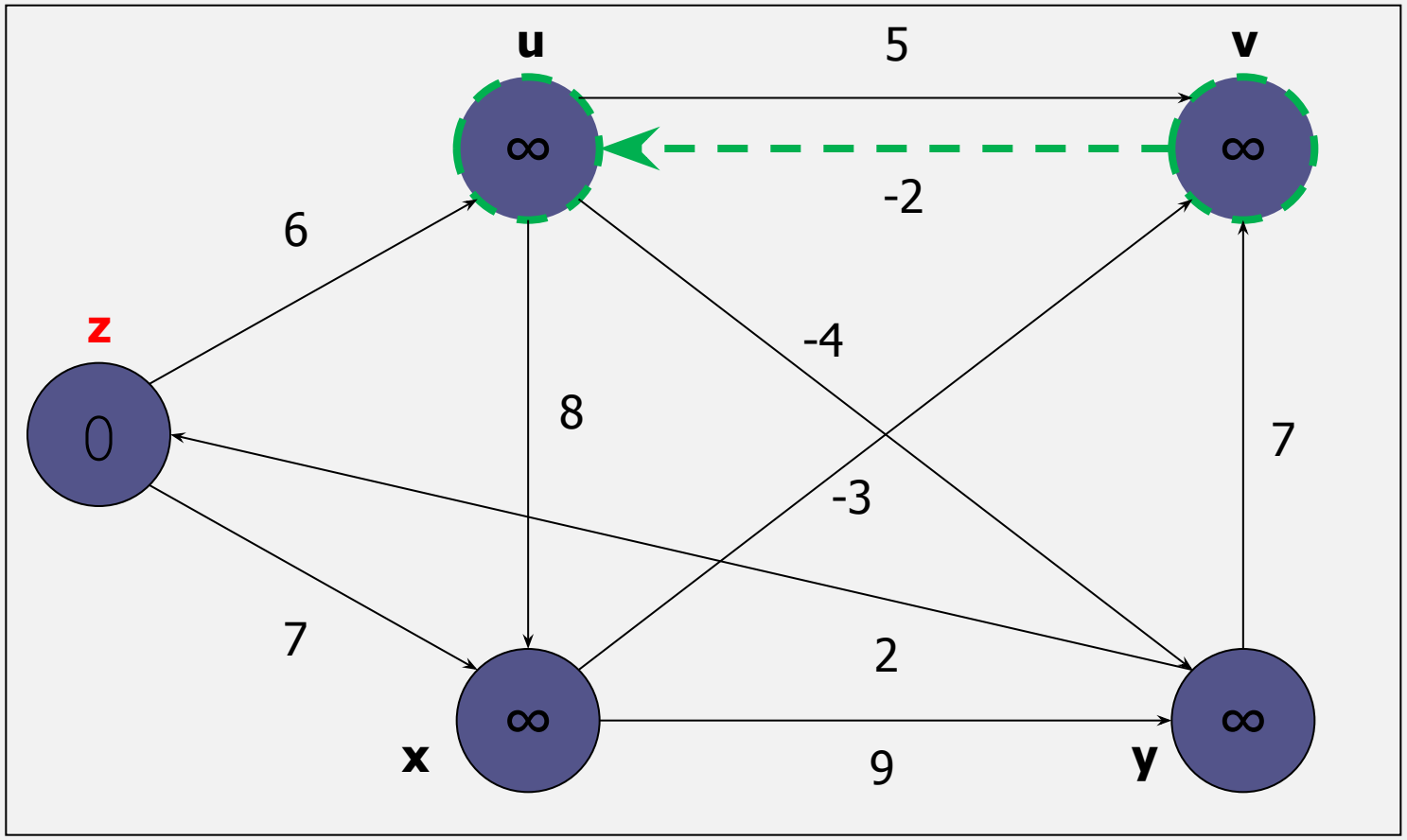
Paso 1.3

Aplicar Relax al Arco (u,y)

Pregunta: ¿ d[y] > d[u] + w(u , y) ?

Respuesta: NO

Proceso: No se hace nada.



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u) ←
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	[	]	=	{	u	v	x	y	z	}
d	[	]	=	{	∞	∞	∞	∞	0	}
Π	[	]	=	{						}

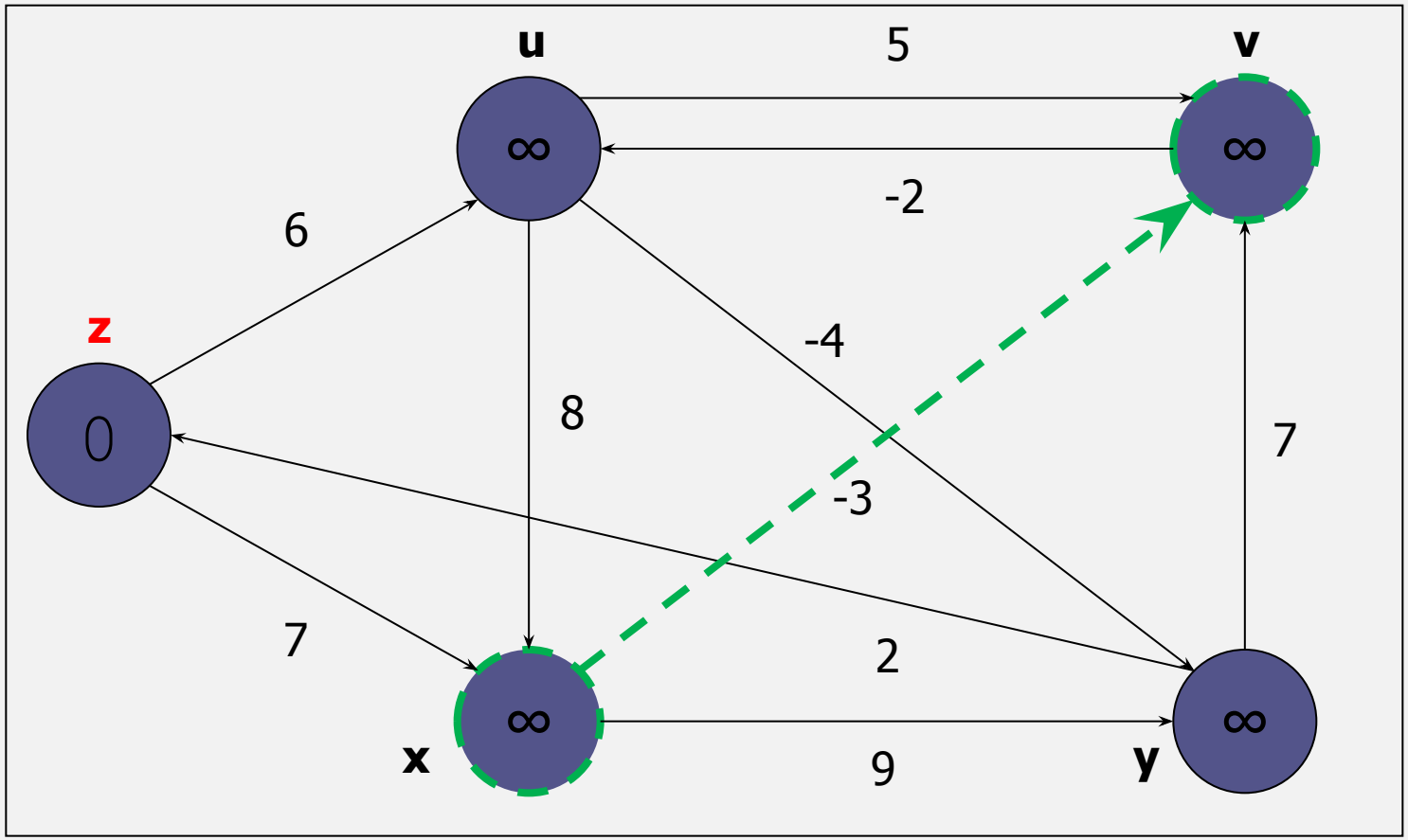
Paso 1.4

Aplicar Relax al Arco (v,u)

Pregunta: ¿ d[u] > d[v] + w(v , u) ?

Respuesta: NO

Proceso: No se hace nada.



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v) ←
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	**∞**	**∞**	**∞**	**∞**	**0**	**}**
Π	**[**	**]**	**=**	**{**						**}**

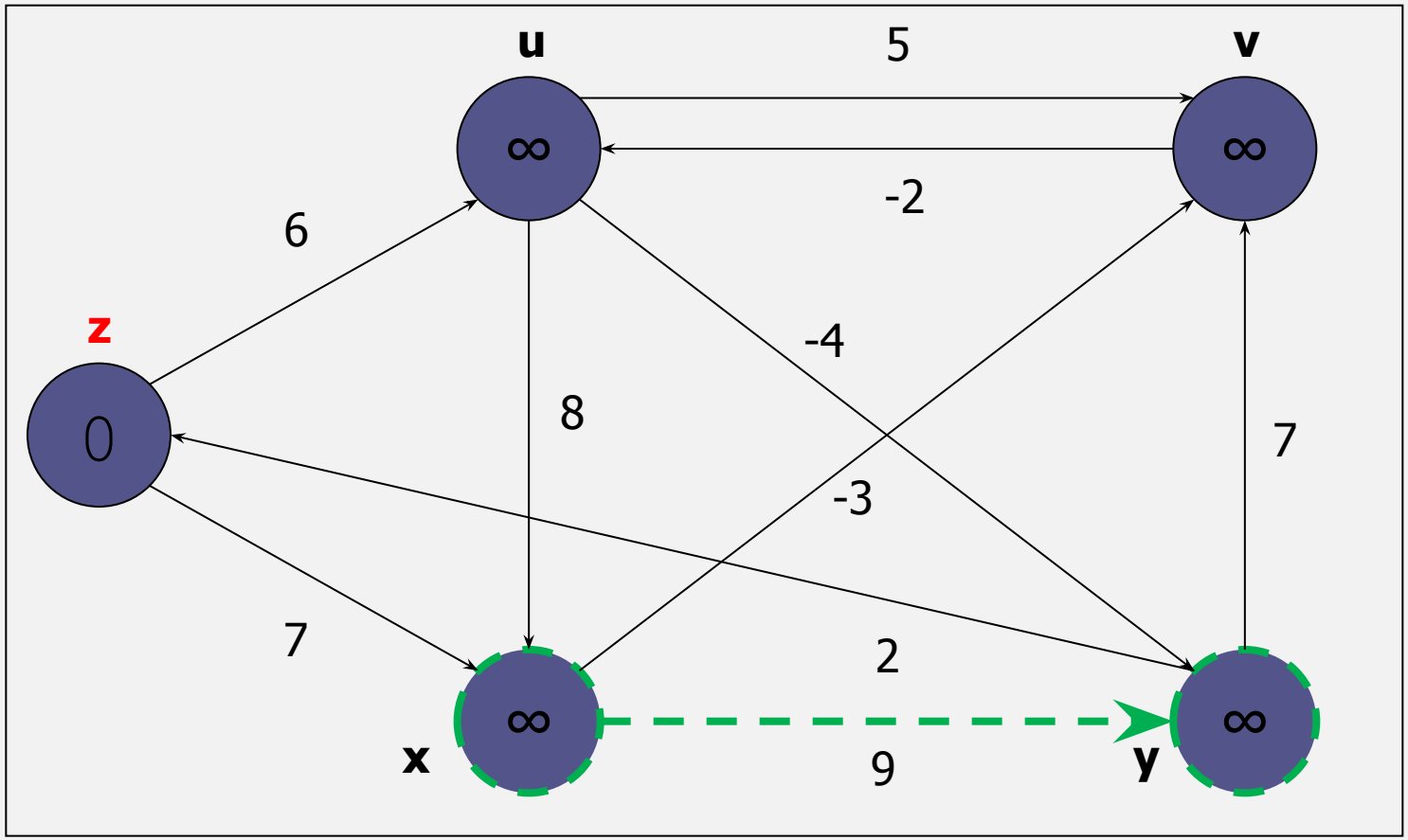
Paso 1.5

Aplicar Relax al Arco (x,v)

Pregunta: ¿ d[v] > d[x] + w(x , v) ?

Respuesta: NO

Proceso: No se hace nada.



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y) ←
(y,v)
(y,z)
(z,u)
(z,x)

V	[	]	=	{	u	v	x	y	z	}
d	[	]	=	{	∞	∞	∞	∞	0	}
Π	[	]	=	{						}

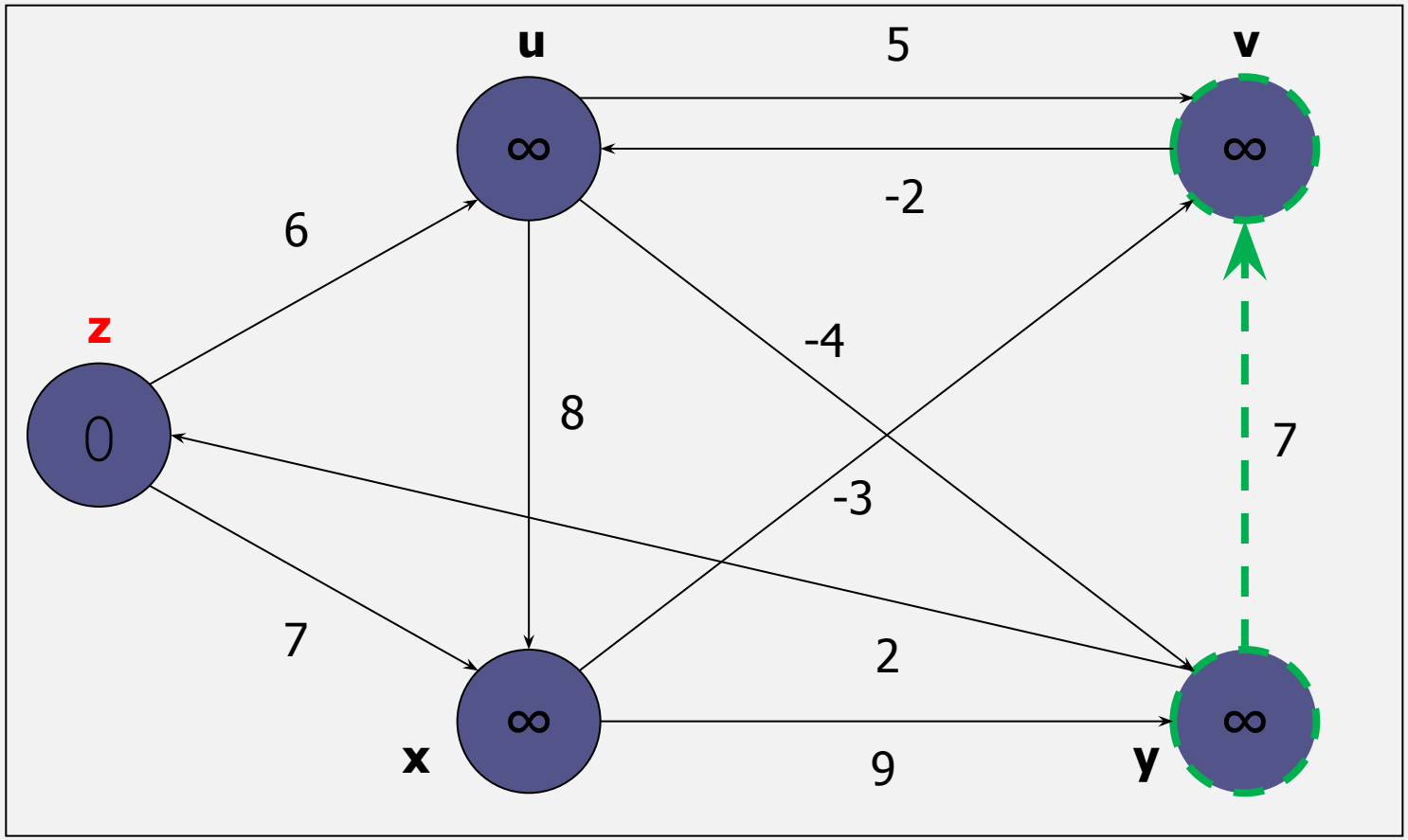
Paso 1.6

Aplicar Relax al Arco (x,y)

Pregunta: ¿ d[y] > d[x] + w(x , y) ?

Respuesta: NO

Proceso: No se hace nada.



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	**∞**	**∞**	**∞**	**∞**	**0**	**}**
Π	**[**	**]**	**=**	**{**						**}**

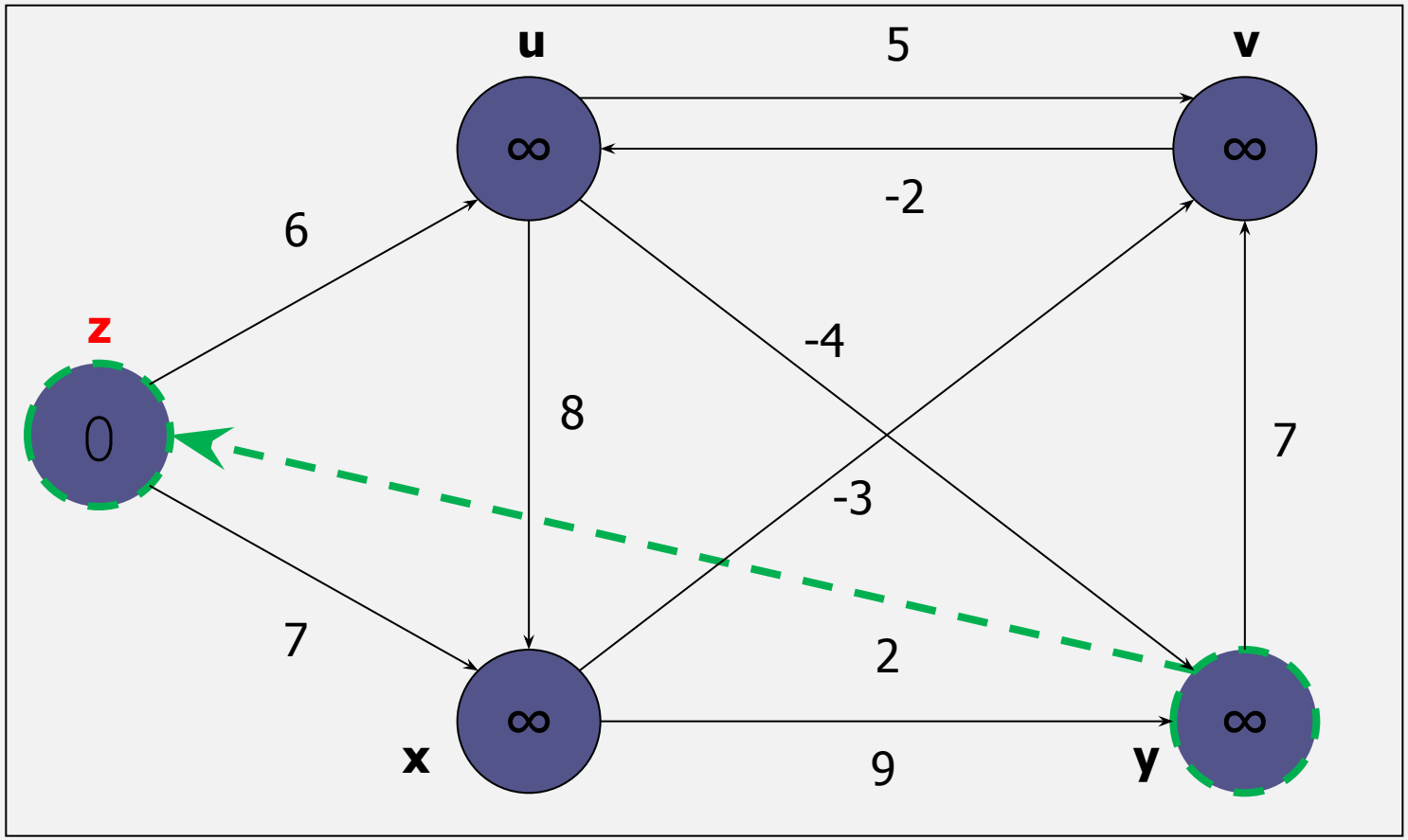
Paso 1.7

Aplicar Relax al Arco (y,v)

Pregunta: ¿ d[v] > d[y] + w(y , v) ?

Respuesta: NO

Proceso: No se hace nada.



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z) ←
(z,u)
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	**∞**	**∞**	**∞**	**∞**	**0**	**}**
Π	**[**	**]**	**=**	**{**						**}**

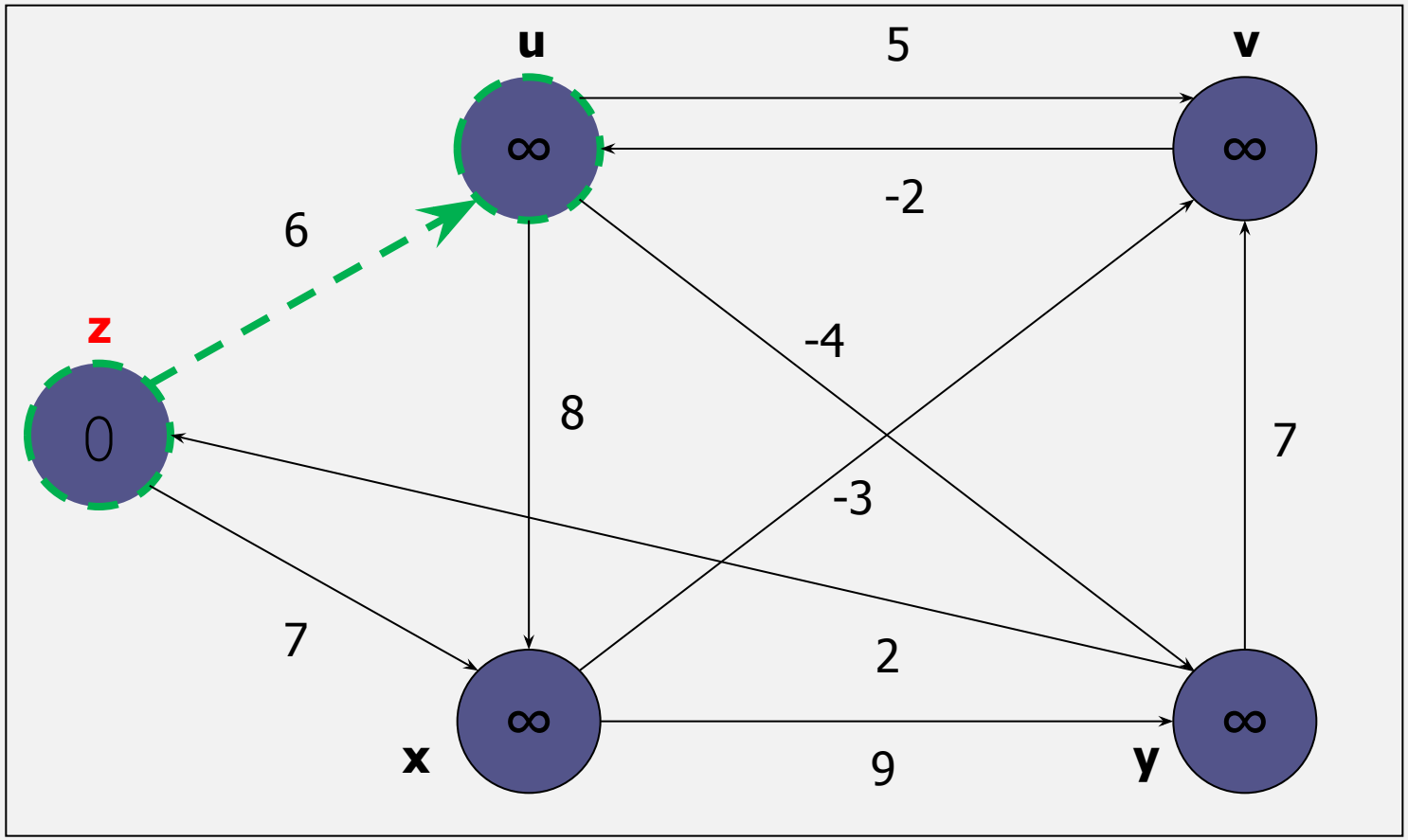
Paso 1.8

Aplicar Relax al Arco (y,z)

Pregunta: ¿ d[z] > d[y] + w(y , z) ?

Respuesta: NO

Proceso: No se hace nada.



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u) ←
(z,x)

V	[	]	=	{	u	v	x	y	z	}
d	[	]	=	{	∞	∞	∞	∞	0	}
Π	[	]	=	{						}

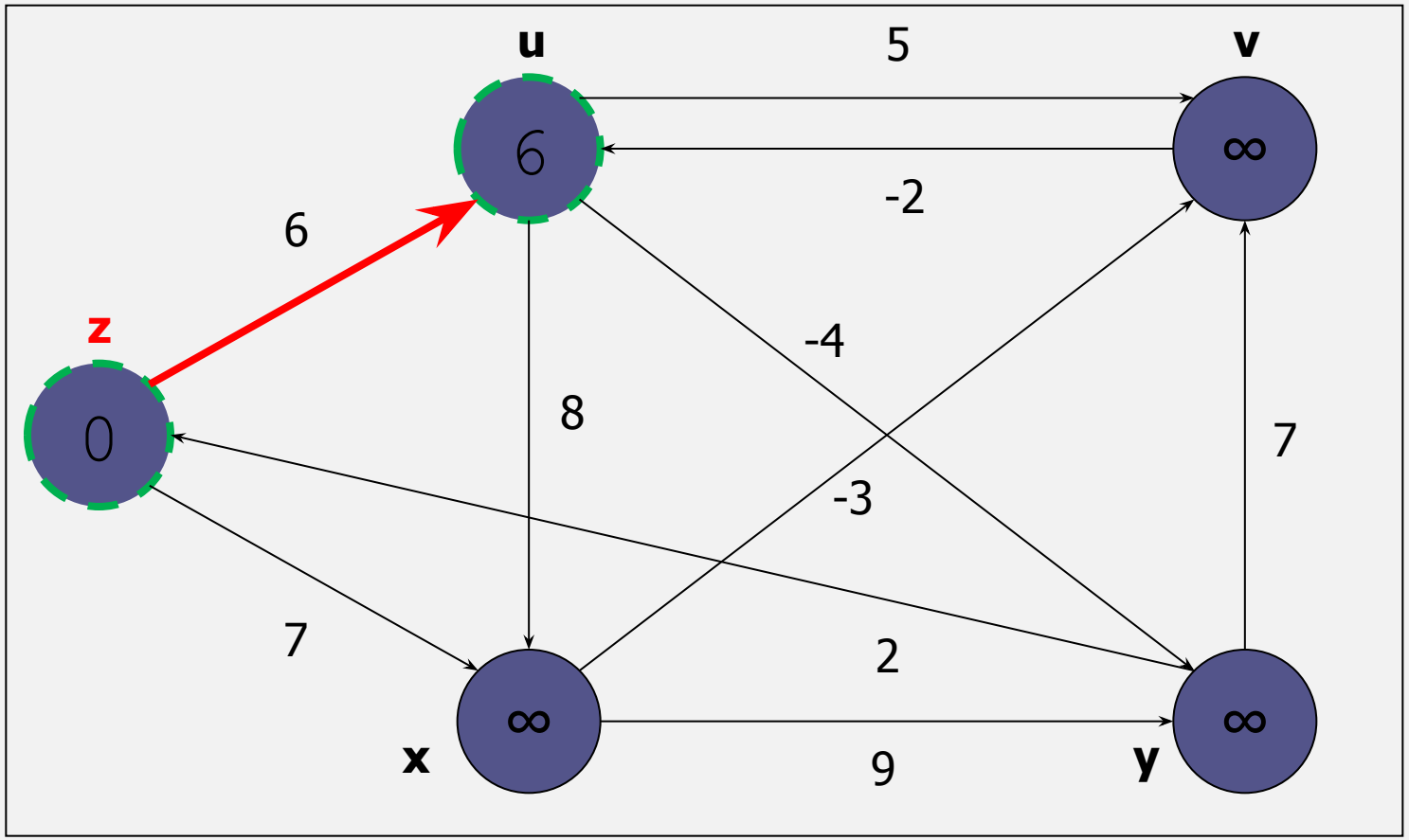
Paso 1.9

Aplicar Relax al Arco (z,u)

Pregunta: ¿ d[u] > d[z] + w(z , u) ?

Respuesta: SI

Proceso: d[u] = d[z] + w(z, u) y Π[u] = z



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u) ←
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	**6**	**∞**	**∞**	**∞**	**0**	**}**
Π	**[**	**]**	**=**	**{**	**z**					**}**

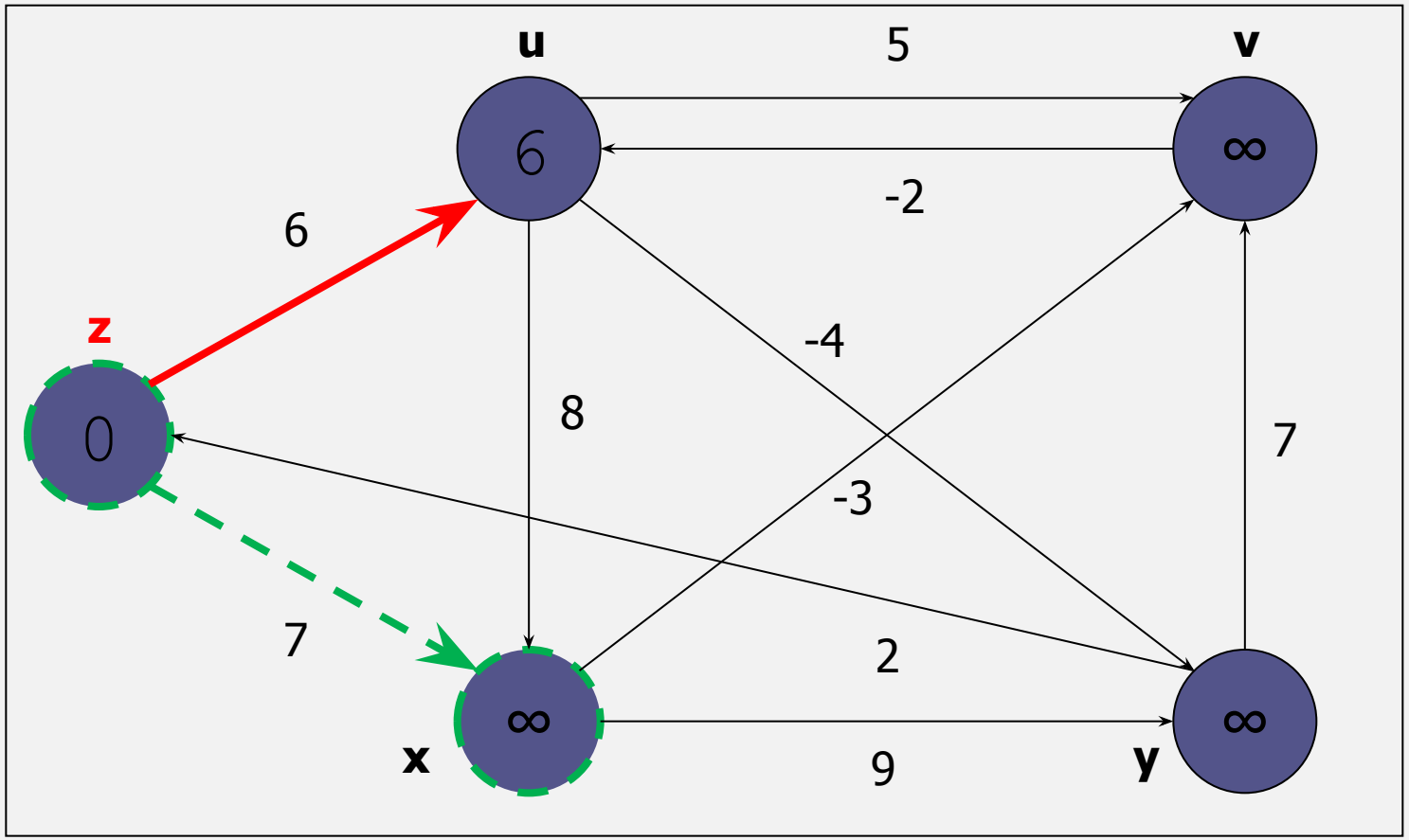
Paso 1.9

Aplicar Relax al Arco (z,u)

Pregunta: ¿ d[u] > d[z] + w(z , u) ?

Respuesta: SI

Proceso: d[u] = d[z] + w(z, u) y Π[u] = z



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	**6**	**∞**	**∞**	**∞**	**0**	**}**
Π	**[**	**]**	**=**	**{**	**z**					**}**

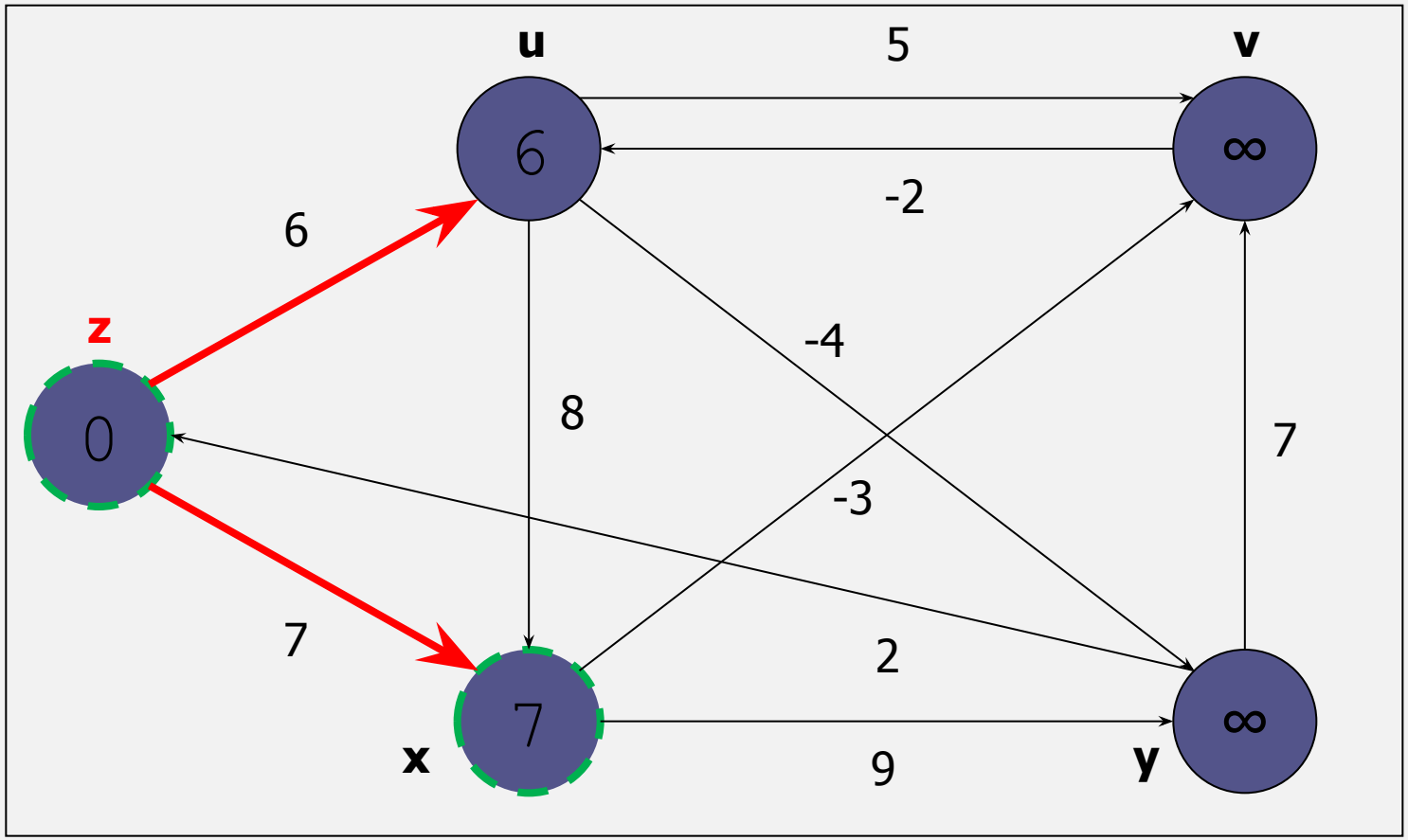
Paso 1.10

Aplicar Relax al Arco (z,x)

Pregunta: ¿ d[x] > d[z] + w(z , x) ?

Respuesta: SI

Proceso: d[x] = d[z] + w(z, x) y Π[x] = z



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	**6**	**∞**	**7**	**∞**	**0**	**}**
Π	**[**	**]**	**=**	**{**	**z**		**z**			**}**

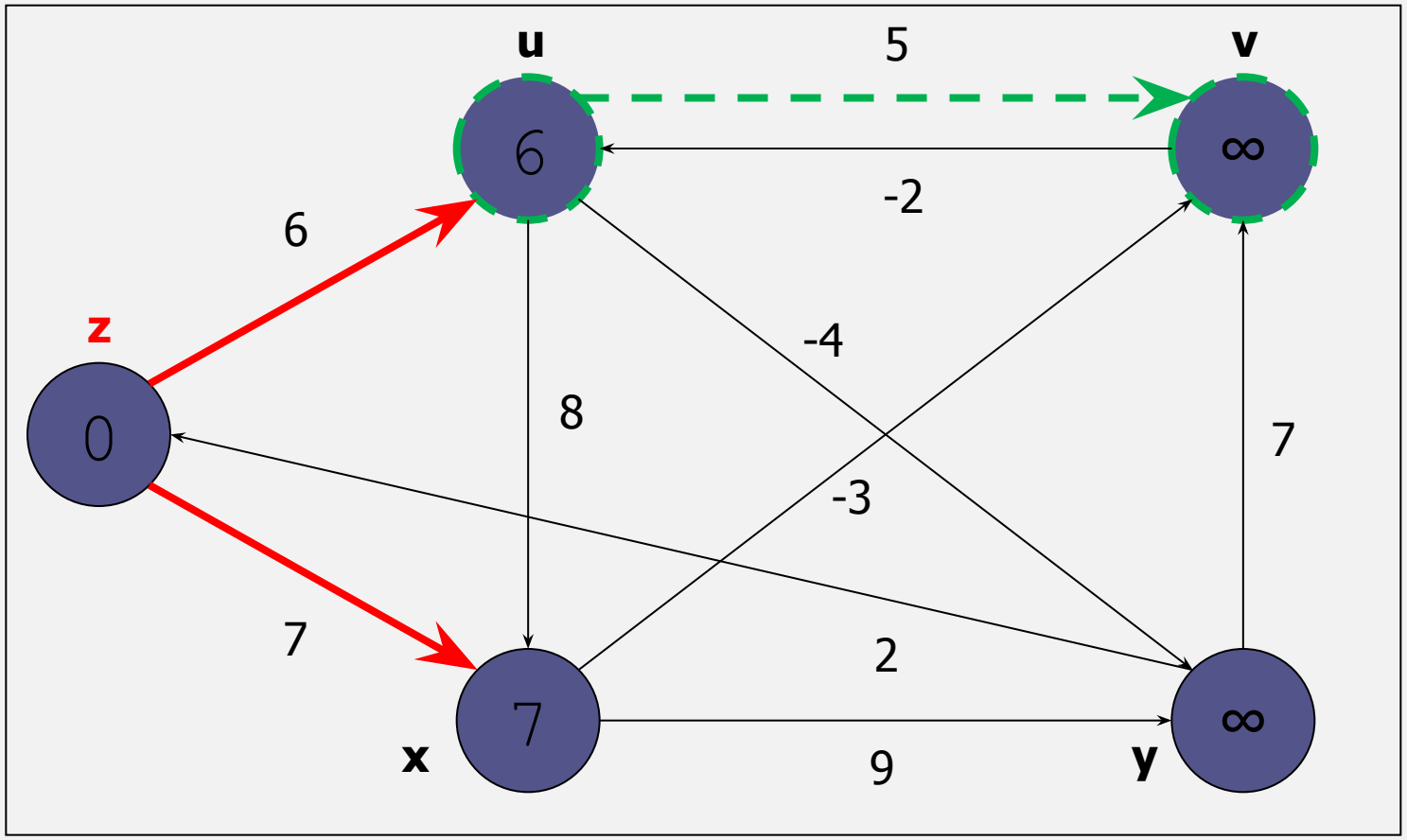
Paso 1.10

Aplicar Relax al Arco (z,x)

Pregunta: ¿ d[x] > d[z] + w(z , x) ?

Respuesta: SI

Proceso: d[x] = d[z] + w(z, x) y Π[x] = z



Lista de Arcos

(u,v) ←
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	**6**	**∞**	**7**	**∞**	**0**	**}**
Π	**[**	**]**	**=**	**{**	**z**		**z**			**}**

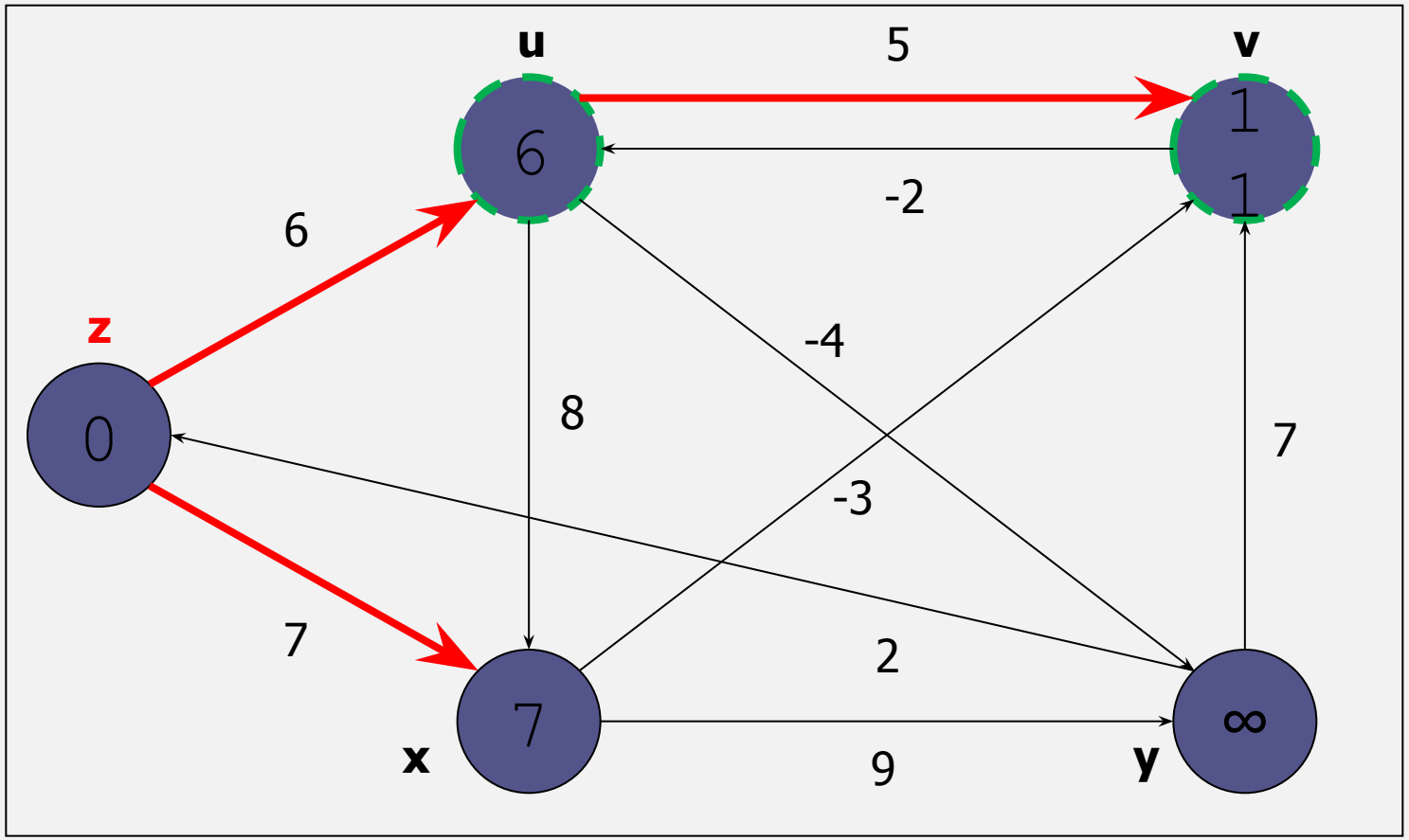
Paso 2.1

Aplicar Relax al Arco (u,v)

Pregunta: ¿ d[v] > d[u] + w(u , v) ?

Respuesta: SI

Proceso: d[v] = d[u] + w(u, v) y Π[v] = u



Lista de Arcos

(u,v) ←
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	**6**	**11**	**7**	**∞**	**0**	**}**
Π	**[**	**]**	**=**	**{**	**z**	**u**	**z**			**}**

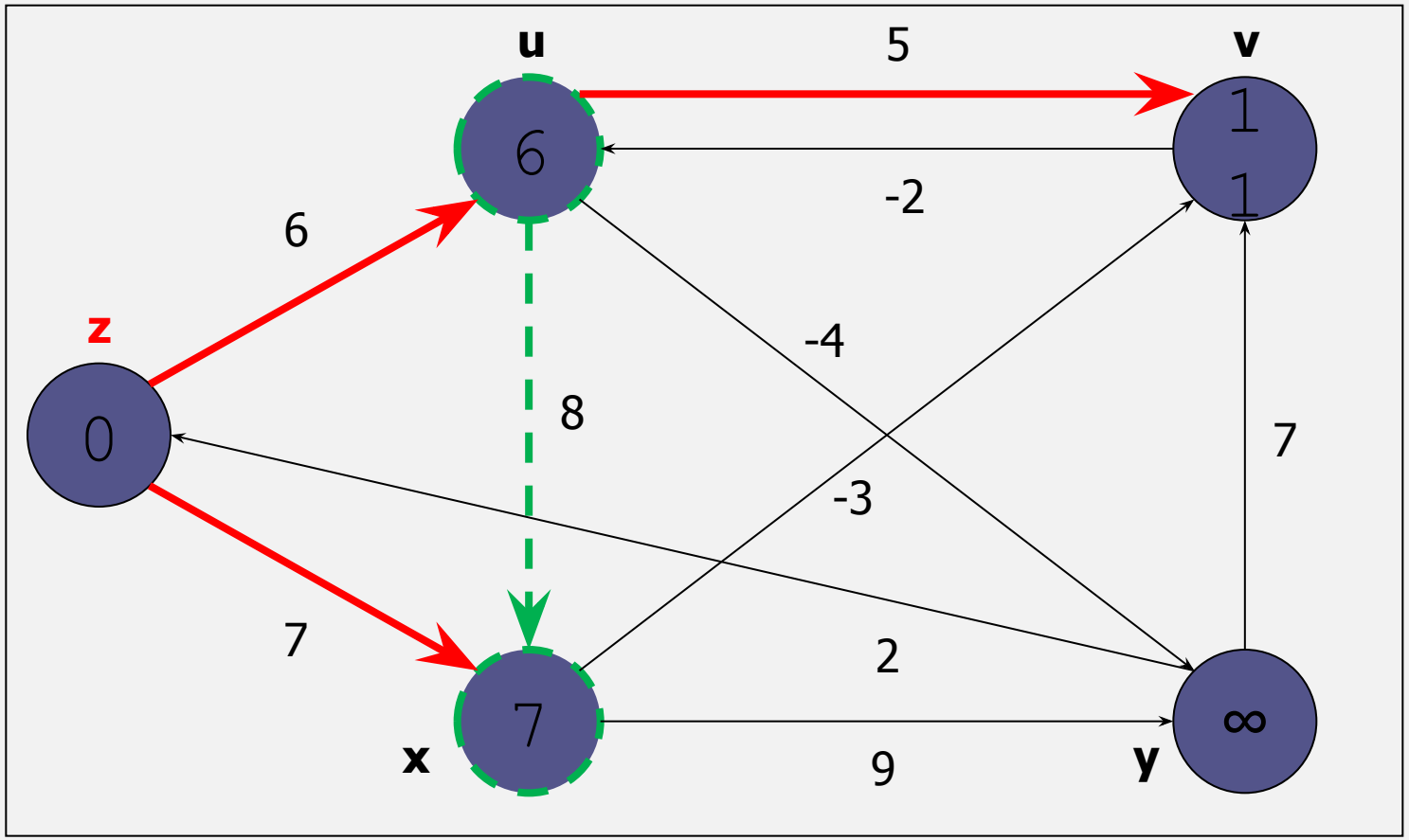
Paso 2.1

Aplicar Relax al Arco (u,v)

Pregunta: ¿ d[v] > d[u] + w(u , v) ?

Respuesta: SI

Proceso: d[v] = d[u] + w(u, v) y Π[v] = u



Lista de Arcos

(u,v)
(u,x) ←
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	**6**	**11**	**7**	**∞**	**0**	**}**
Π	**[**	**]**	**=**	**{**	**z**	**u**	**z**			**}**

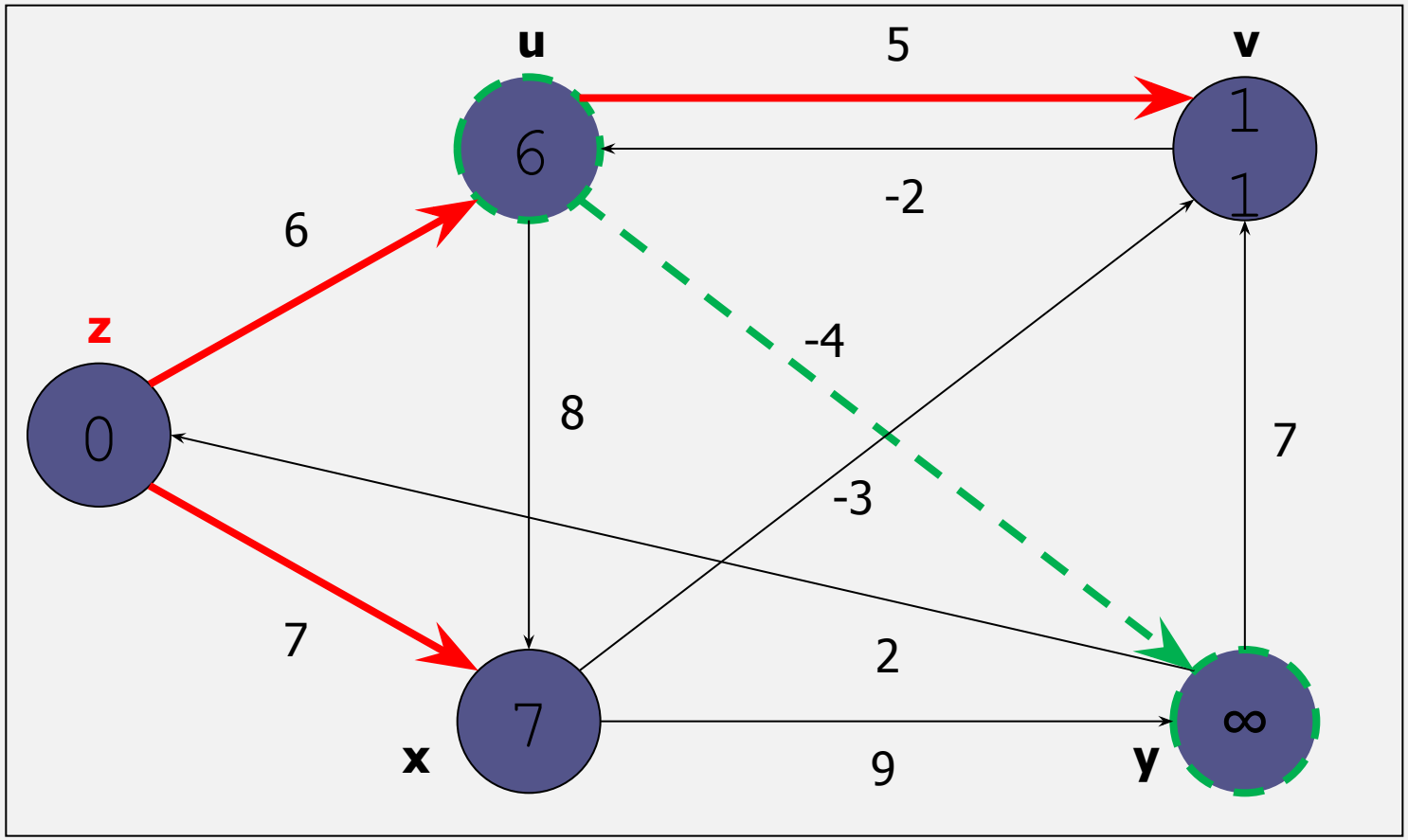
Paso 2.2

Aplicar Relax al Arco (u,x)

Pregunta: ¿ d[x] > d[u] + w(u , x) ?

Respuesta: NO

Proceso: No se hace nada.



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	[	]	=	{	u	v	x	y	z	}
d	[	]	=	{	6	11	7	∞	0	}
Π	[	]	=	{	z	u	z			}

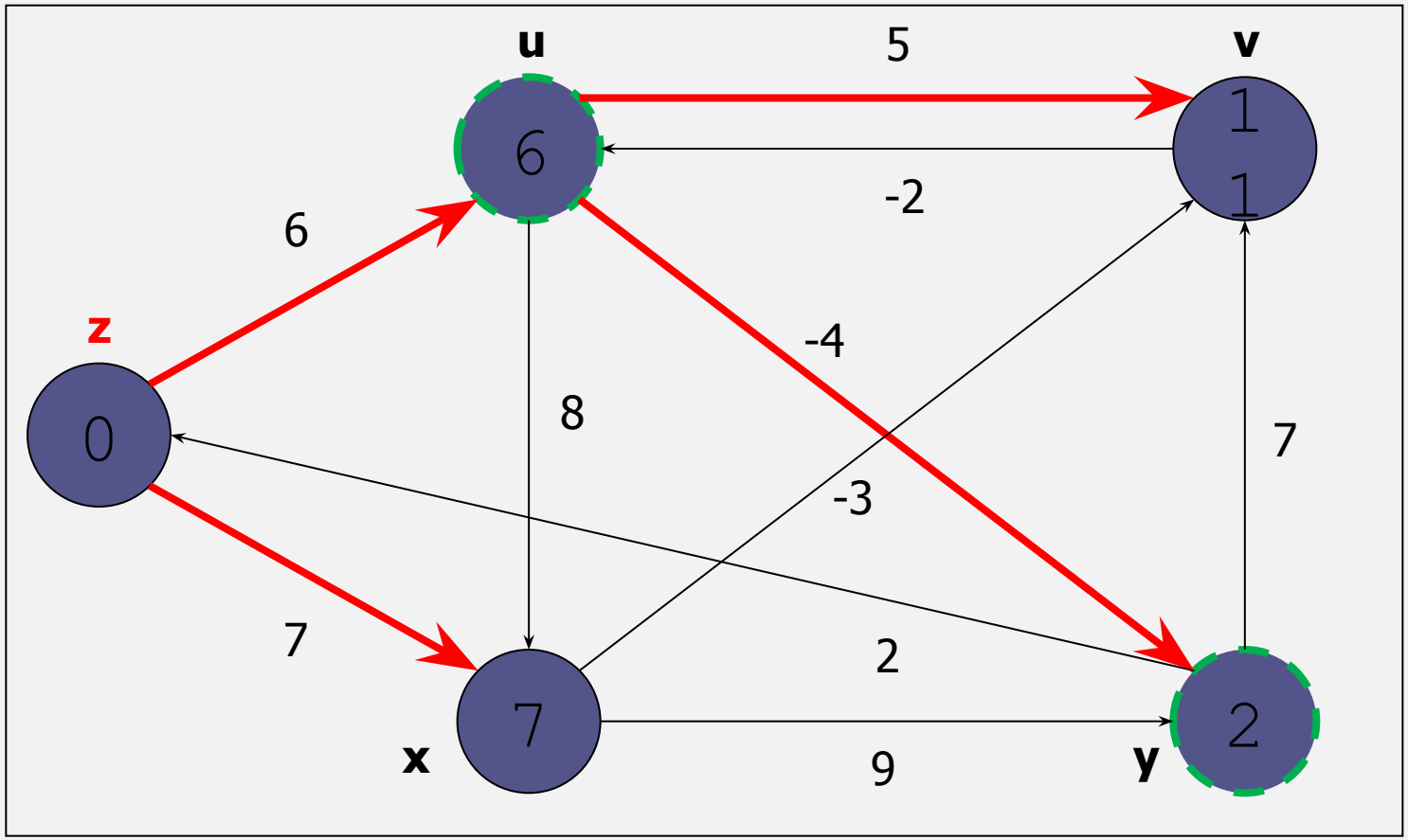
Paso 2.3

Aplicar Relax al Arco (u,y)

Pregunta: ¿ d[y] > d[u] + w(u , y) ?

Respuesta: SI

Proceso: d[y] = d[u] + w(u, y) y Π[y] = u



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	[	]	=	{	u	v	x	y	z	}
d	[	]	=	{	6	11	7	2	0	}
Π	[	]	=	{	z	u	z	u		}

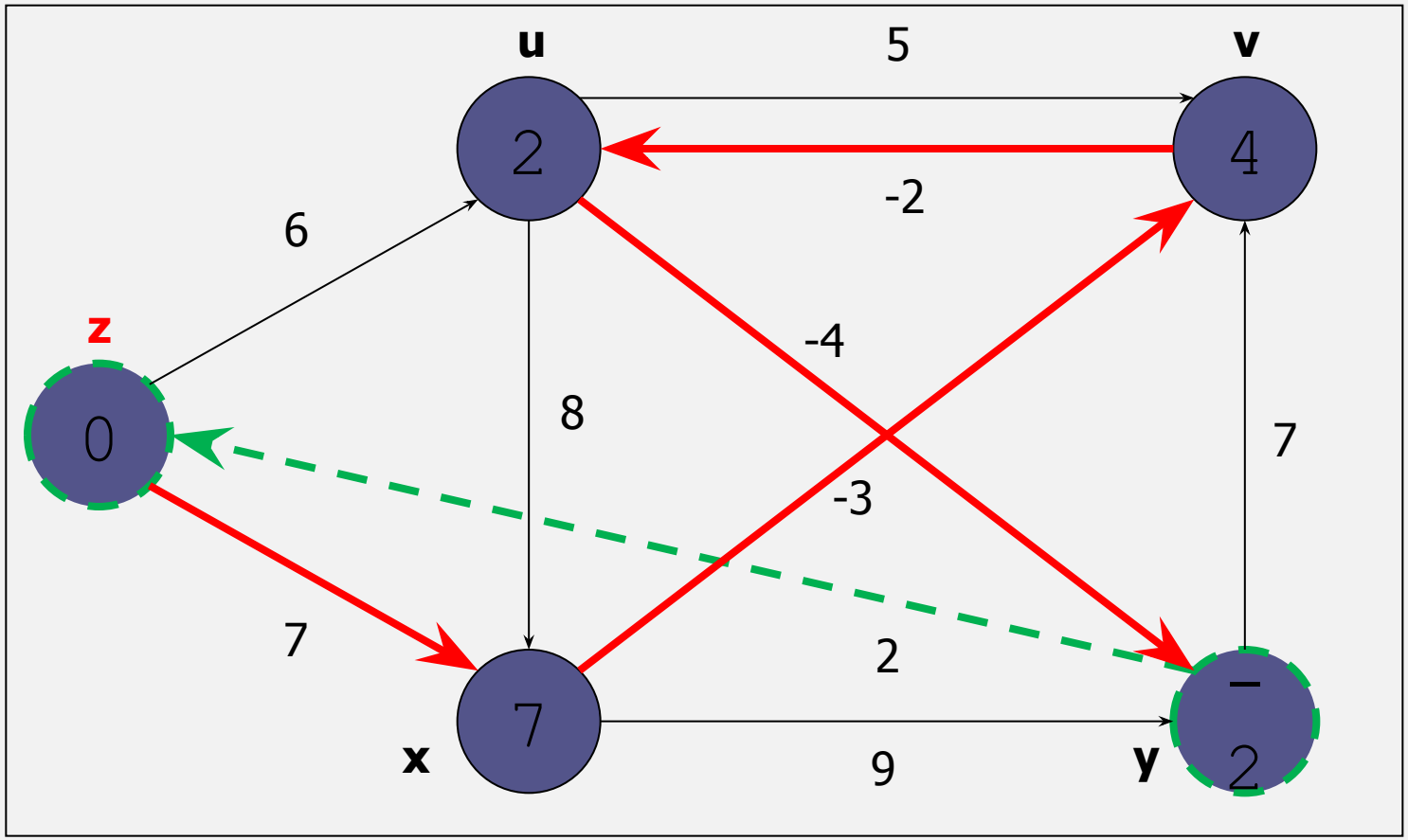
Paso 2.3

Aplicar Relax al Arco (u,y)

Pregunta: ¿ d[y] > d[u] + w(u , y) ?

Respuesta: SI

Proceso: d[y] = d[u] + w(u, y) y Π[y] = u



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z) ←
(z,u)
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	**2**	**4**	**7**	**−2**	**0**	**}**
Π	**[**	**]**	**=**	**{**	**v**	**x**	**z**	**u**		**}**

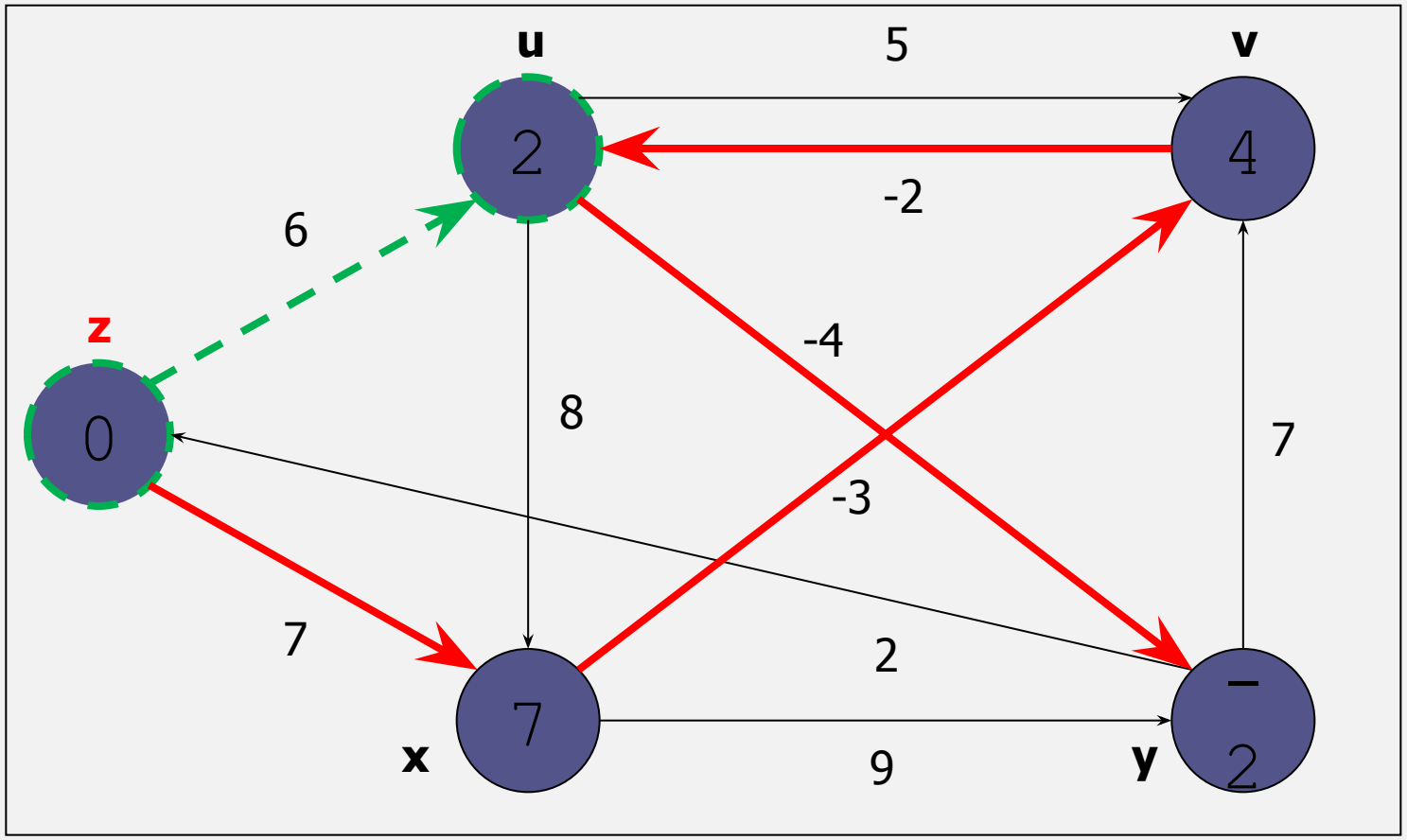
Paso 4.8

Aplicar Relax al Arco (y, z)

Pregunta: ¿ d[z] > d[y] + w(y , z) ?

Respuesta: NO

Proceso: No se hace nada.



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u) ←
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	**2**	**4**	**7**	**−2**	**0**	**}**
Π	**[**	**]**	**=**	**{**	**v**	**x**	**z**	**u**		**}**

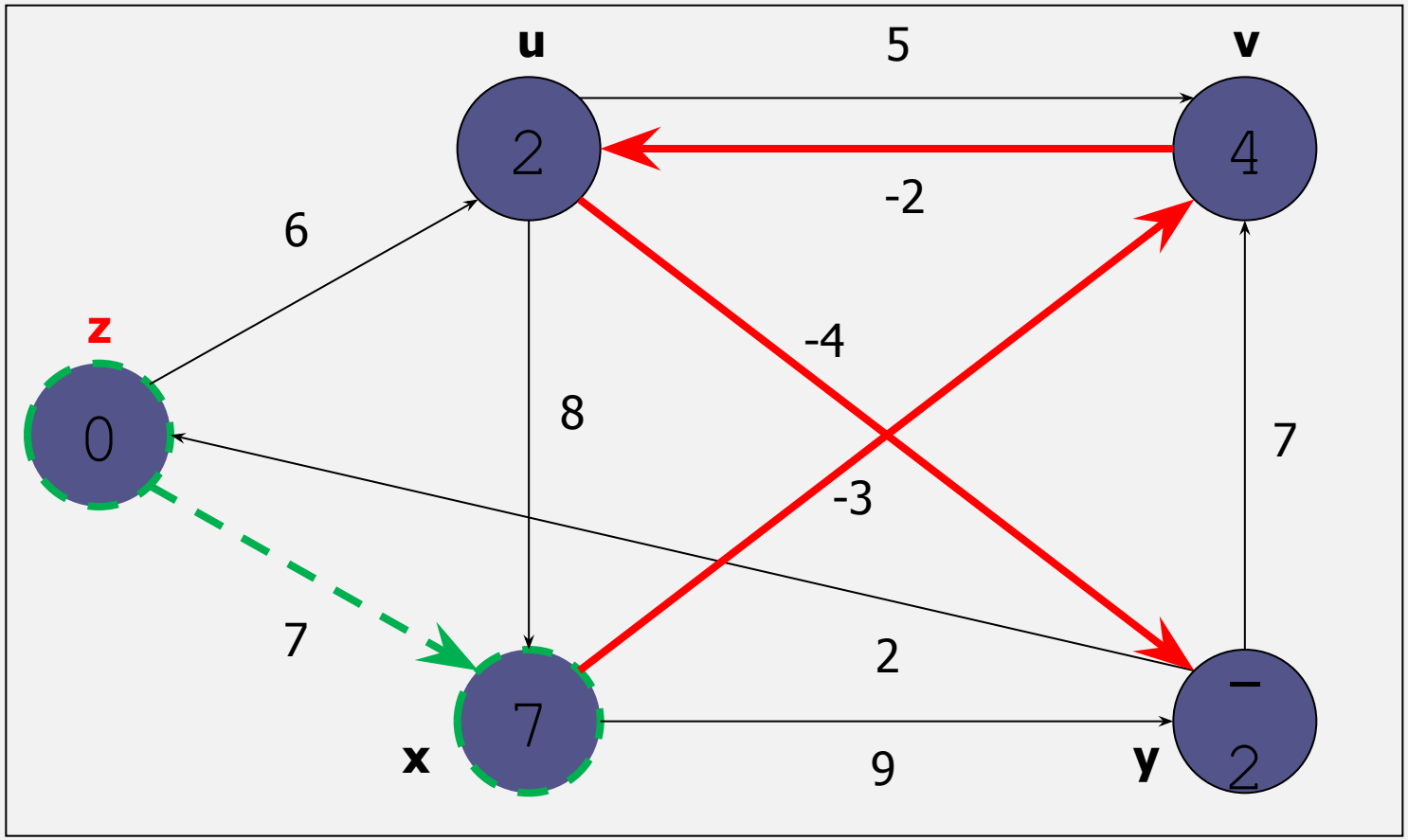
Paso 4.9

Aplicar Relax al Arco (z, u)

Pregunta: ¿ d[u] > d[z] + w(z , u) ?

Respuesta: NO

Proceso: No se hace nada.



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	[	]	=	{	u	v	x	y	z	}
d	[	]	=	{	2	4	7	−2	0	}
Π	[	]	=	{	v	x	z	u		}

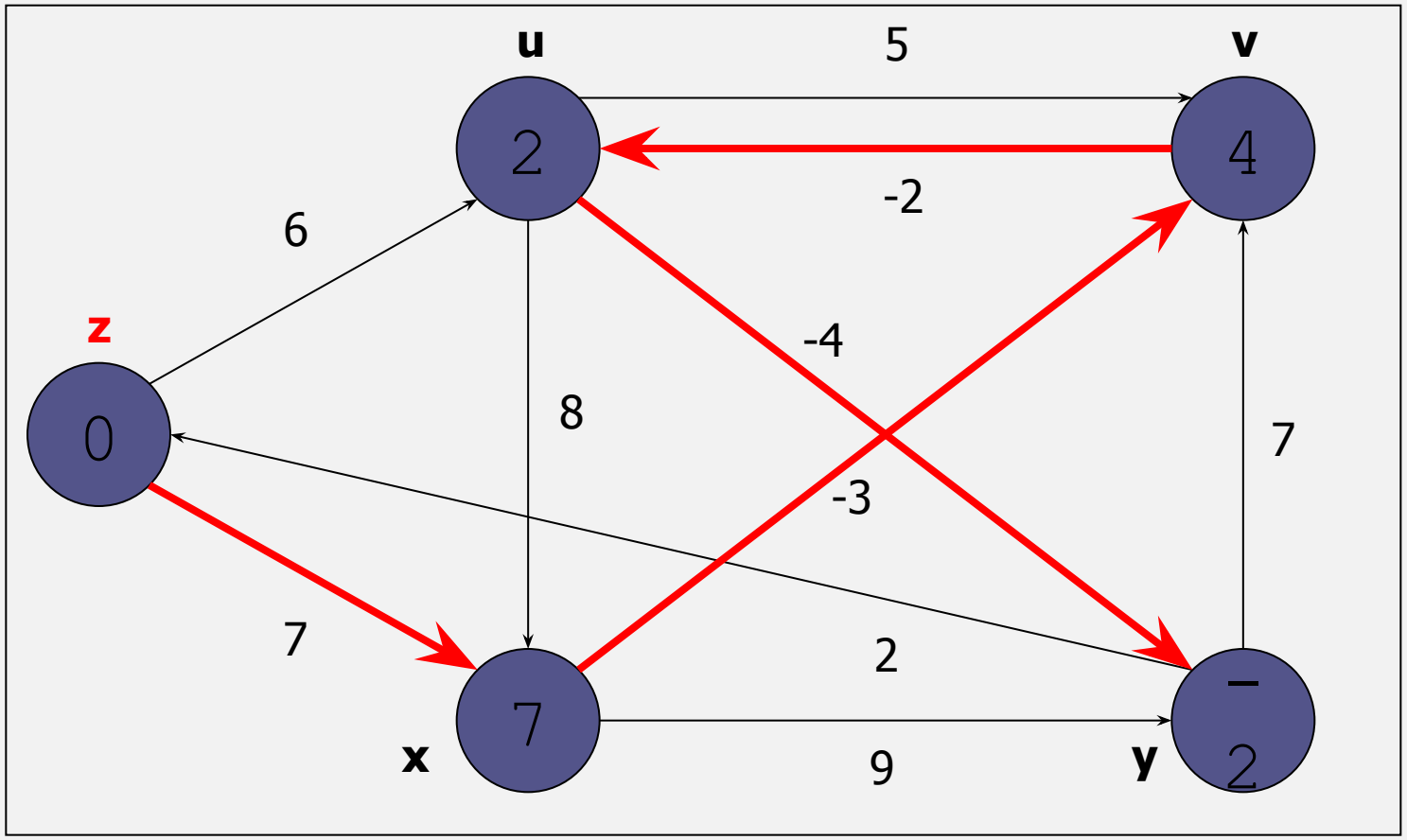
Paso 4.10

Aplicar Relax al Arco (z, x)

Pregunta: ¿ d[x] > d[z] + w(z , x) ?

Respuesta: NO

Proceso: No se hace nada.



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	**2**	**4**	**7**	**−2**	**0**	**}**
Π	**[**	**]**	**=**	**{**	**v**	**x**	**z**	**u**		**}**

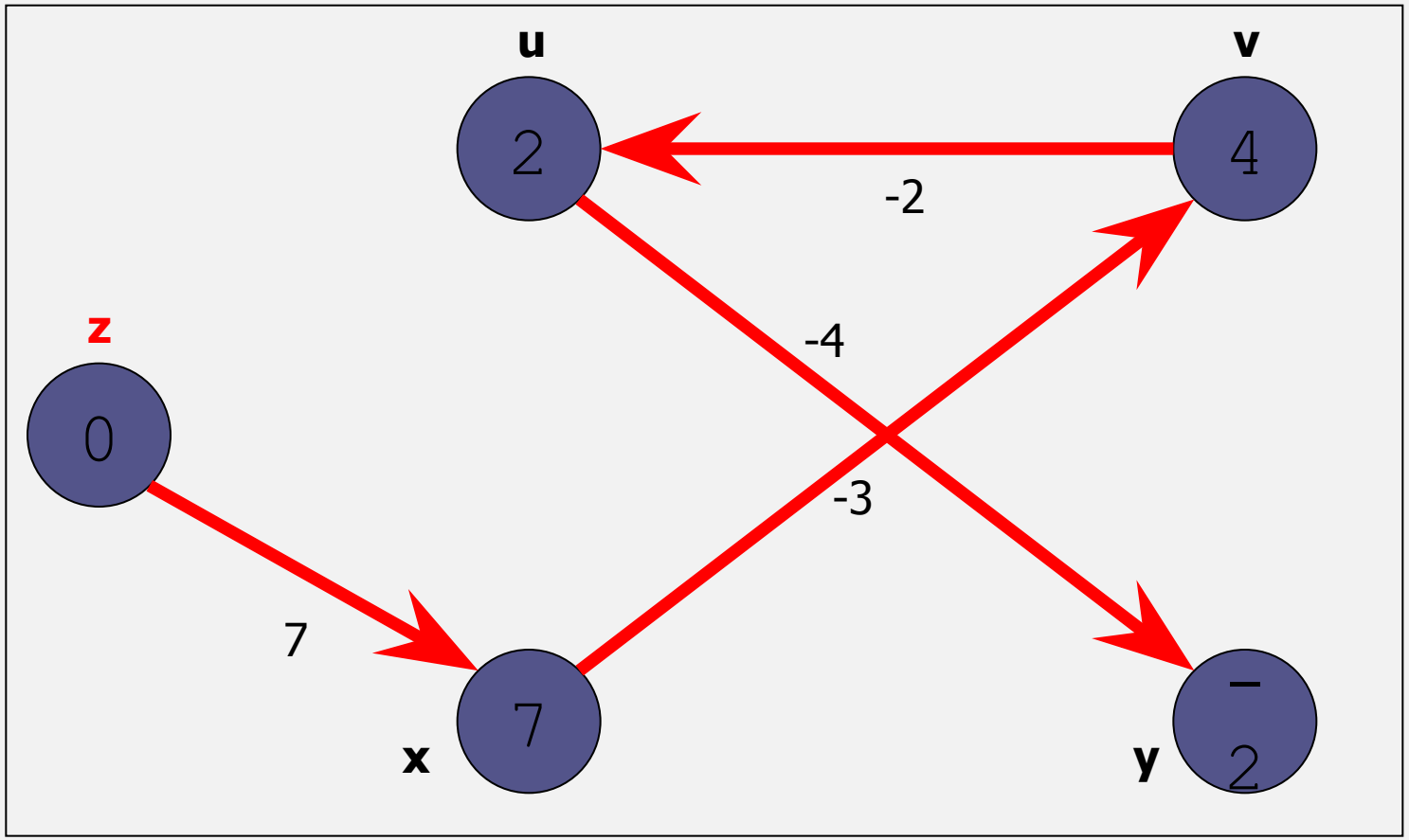
Paso 5.0

Verificar en cada arco que se cumple la condición:

d[Vf] <= d[Vi] + w(Vi , Vf)

Si no se cumple:

=> **NO EXISTE SOLUCIÓN.**



Lista de Arcos

(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

V	**[**	**]**	**=**	**{**	**u**	**v**	**x**	**y**	**z**	**}**
d	**[**	**]**	**=**	**{**	**2**	**4**	**7**	**−2**	**0**	**}**
Π	**[**	**]**	**=**	**{**	**v**	**x**	**z**	**u**		**}**

SOLUCIÓN

Algoritmo de Floyd (APSP)

```
// Precondición: AdjMat [i, j] contiene el peso de la arista (i, j)
// o INF si no hay arista

 para k desde 1 hasta |V|
    para i desde 1 hasta |V|
      para j desde 1 hasta |V|
         si  AdjMat [i, j] > AdjMat [i, k] + AdjMat [k, j]
               AdjMat [i, j] = AdjMat [i, k] + AdjMat [k, j]
               P[i, j] = P[k, j]  //la matriz para armar el camino
```

El orden es $O(|V|^3)$, se usa para grafos pequeños, en este contexto, con $|V| \leq 400$

Grafos **Tamaño máximo**	**BFS** **O(V+E)** V, E ≤ 10 M	**Dijkstra** **O(E log V)** V, E ≤ 300 K	**Bellman Ford** **O(V*E)** V*E ≤ 10 M	**Floyd** **O(V^3)** V ≤ 400
No pesados	Óptimo	Correcto	Malo	En general malo
Pesados (positivos)	WA	Óptimo	Correcto	En general malo
Pesos negativos	WA	WA (Variante OK)	Correcto	En general malo
Ciclos negativos	No los puede detectar	No los puede detectar	Los detecta	Los detecta
Grafos pequeños	WA si es pesado	Ok pero innecesario	Ok pero innecesario	Óptimo

Correcto → adecuado pero no es el mejor
Malo → una solución muy lenta
WA → algoritmo incorrecto

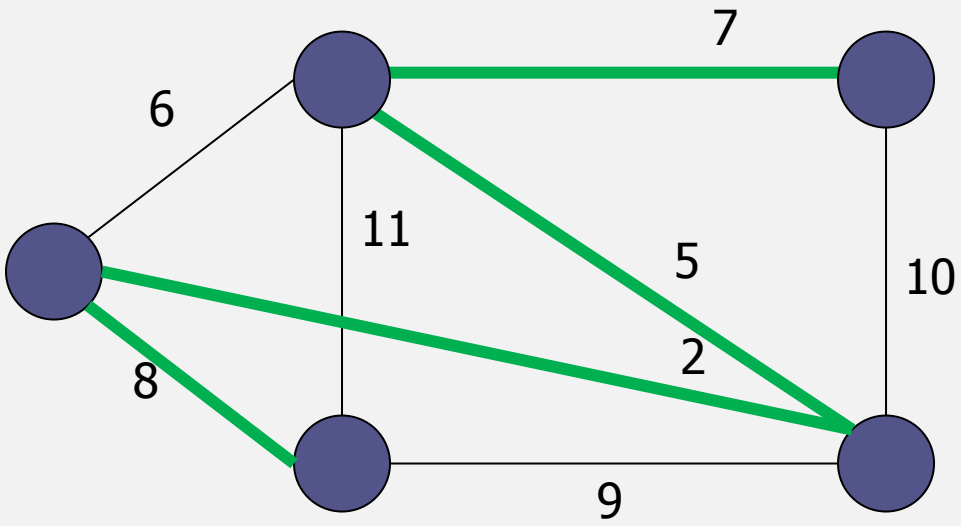
Árbol de expansión mínima

- Algoritmo Kruskal
- Aplicaciones

Árbol de expansión mínima

Dado un grafo G=(V, E) no dirigido, pesado y conexo, se debe seleccionar el subconjunto de aristas E' incluido en E, de manera que el grafo G siga estando conectado y el peso total de las aristas seleccionadas sea **mínimo**.
Para que se cumpla el criterio de conectividad se necesitan **V-1 aristas que abarquen todos los vértices de G**

Árbol de expansión o abarcador mínimo



Árbol de expansión mínima: Algoritmo de Kruskal

- Ordena las E aristas en orden no decreciente según el peso
- Trata de agregar cada arista en el MST en orden, si no forma un ciclo
  - Usando UFDS

Orden de ejecución:

O(ordenar las aristas + tratar de agregar cada arista x costo de las operaciones de UFDS)

Árbol de expansión mínima: Algoritmo de Kruskal

Orden de ejecución:

O(ordenar las aristas + tratar de agregar cada arista x costo de las operaciones de UFDS) =

$O(E \log E + E \times (\approx 1)) = O(E \log E) = O(E \log V^2) =$

$O(2 \times E \log V) = O(E \log V)$

Árbol de expansión mínima: Algoritmo de Kruskal

para cada arista ***e*** (v_i, v_j)
    leer la arista;
    guardarla en la ListaAristas (vector);
ordenar ListaAristas;
costo_mst =0;
inicializar los conjuntos con cada vértice;
para cada arista ***e*** (v_i, v_j) {
    si conjunto(v_i) != conjunto(v_j)
        costo_mst = costo_mst + peso de (v_i,v_j);
        hacer la union del conjunto(v_i) y conjunto(v_j)
}
devolver costo_mst;

Árbol de expansión mínima: Aplicaciones

- Árbol de expansión máxima

  Modificar el algoritmo de Kruskal, ordenando las aristas en orden *no creciente* según el peso.

- Segundo mejor árbol de expansión

  En caso en que no funcione el MST, podemos buscar el segundo mejor
  - Ordenar las aristas → O(E log V)
  - Buscar el MST usando Kruskal → O(E)
  - Para cada arista (hay V -1 aristas), excluirla y volver a calcular el MST → O(E)

  El orden final es:
  O(E log V + E + V*E) = O(V*E)

Árbol de expansión mínima: Aplicaciones

- Minimax (y Maximin)

  El problema del camino minimax es encontrar el mínimo de los pesos máximos de las aristas entre todos los posibles caminos entre dos vértices i y j.

  Se puede modelizar como un problema de MST, ya que éste elige un camino con los mínimos pesos individuales.

  En el MST hay un camino entre todo par de vértices.

  La solución al problema minimax es el mayor peso en el único camino entre los vértices i y j del MST.

  El orden es:

  O(construir MST + un recorrido en el árbol)

  O(E log V + V) = O(E log V)

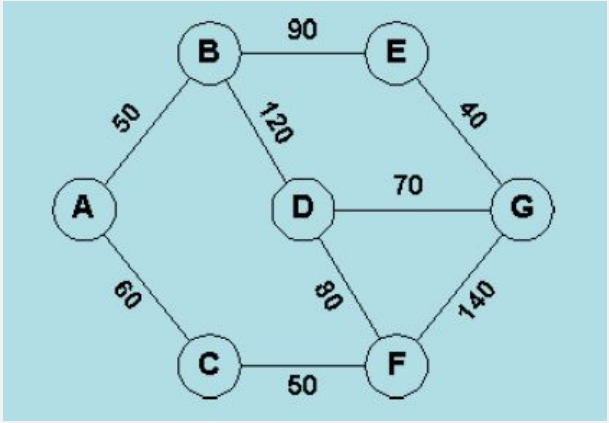
Árbol de expansión mínima: Aplicaciones

Problema: 10048 - Audiophobia

Consider yourself lucky! Consider yourself lucky to be still breathing and having fun participating in this contest. But we apprehend that many of your descendants may not have this luxury. For, as you know, we are the dwellers of one of the most polluted cities on earth. Pollution is everywhere, both in the environment and in society and our lack of consciousness is simply aggravating the situation.

However, for the time being, we will consider only one type of pollution - the sound pollution. The loudness or intensity level of sound is usually measured in *decibels* and sound having intensity level 130 decibels or higher is considered painful. The intensity level of normal conversation is 6065 decibels and that of heavy traffic is 7080 decibels.

Consider the following city map where the edges refer to streets and the nodes refer to crossings. The integer on each edge is the average intensity level of sound (in decibels) in the corresponding street.

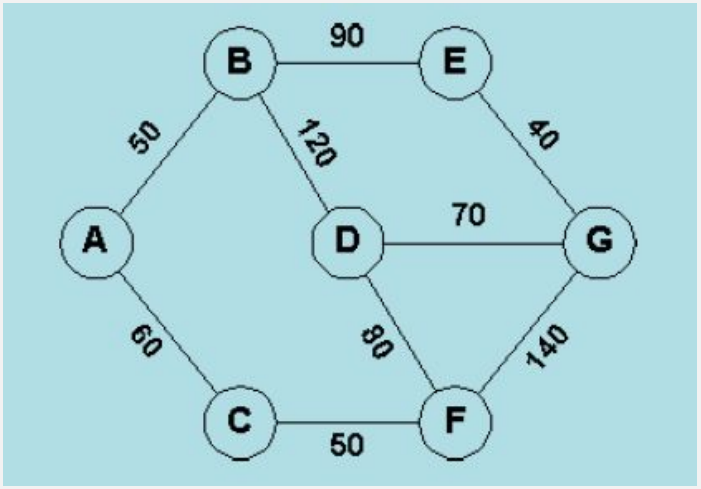


Árbol de expansión mínima: Aplicaciones

Problema: 10048 - Audiophobia

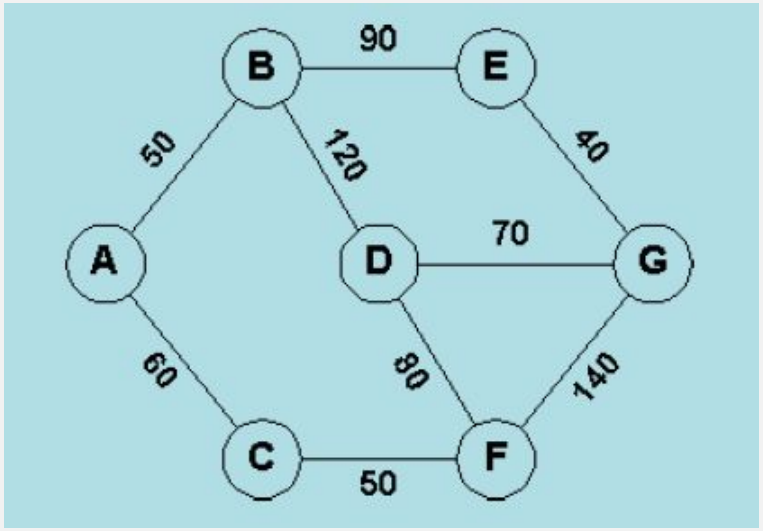
To get from crossing **A** to crossing **G** you may follow the following path: **A-C-F-G**. In that case you must be capable of tolerating sound intensity as high as 140 decibels. For the paths **A-B-E-G**, **A-B-D-G** and **A-C-F-D-G** you must tolerate respectively 90, 120 and 80 decibels of sound intensity. There are other paths, too. However, it is clear that **A-C-F-D-G** is the most comfortable path since it does not demand you to tolerate more than 80 decibels.

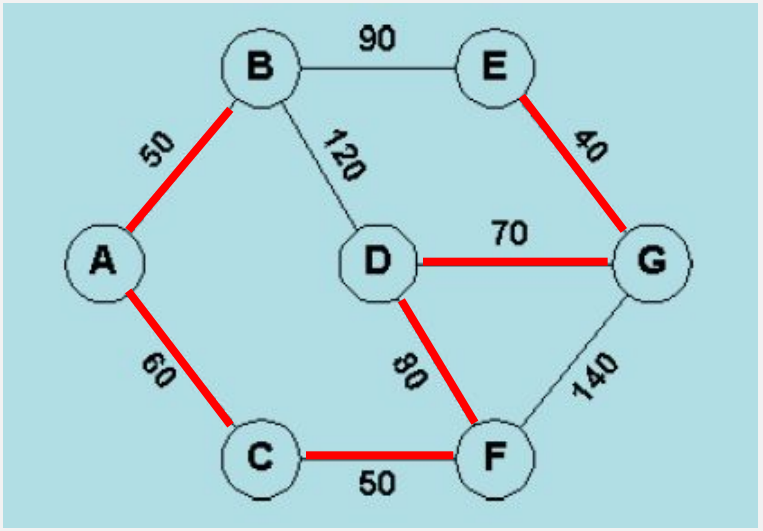
In this problem, given a city map you are required to determine the minimum sound intensity level you must be able to tolerate in order to get from a given crossing to another.



Árbol de expansión mínima: Aplicaciones

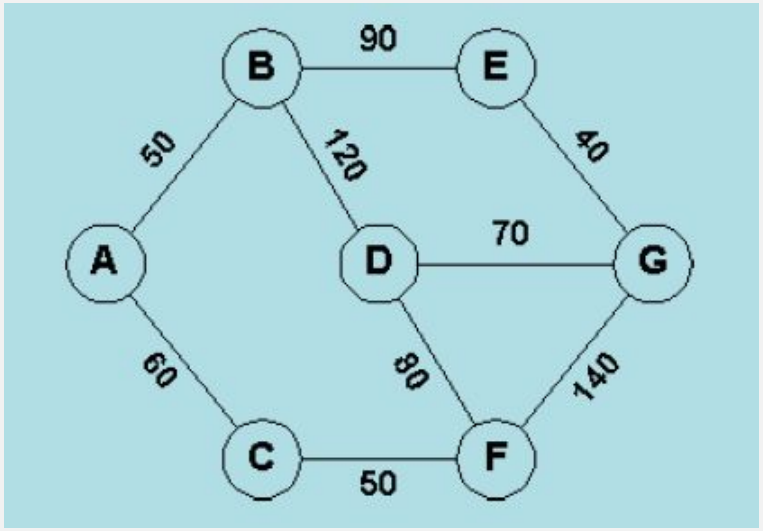
Problema: 10048 - Audiophobia

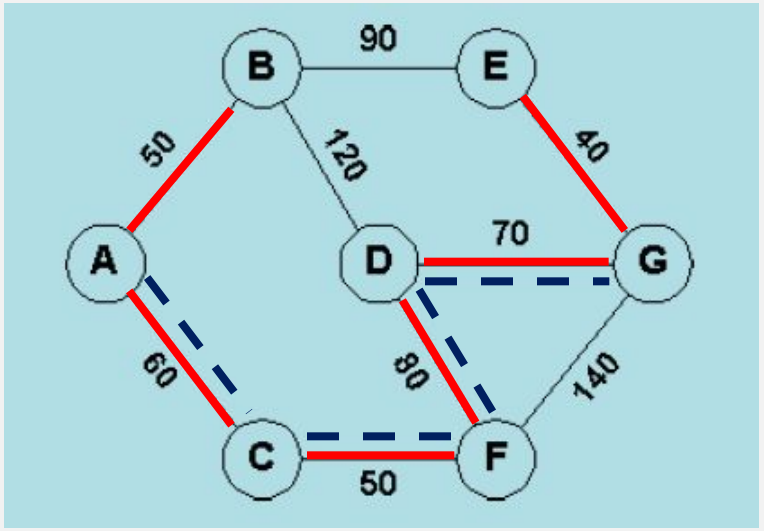




Árbol de expansión mínima: Aplicaciones

Problema: 10048 - Audiophobia





Si queremos encontrar el camino minimax entre los vértices A y G, simplemente recorremos el MST desde A hasta G.
Hay un solo camino: A-C-F-D-G
El peso máximo de una arista en ese camino es el costo minimax: 80