Índice

- **EL ENFOQUE GREEDY**
- **ALGORITMOS GREEDY EN GRAFOS**
  - **Arboles de Recubrimiento Mínimo (Generadores Minimales)**
    - **Algoritmo de Kruskal**
    - **Algoritmo de Prim**
  - **Caminos Mínimos**
    - **Algoritmo de Dijkstra**
- **HEURÍSTICA GREEDY**

Árbol generador minimal

- Sea G = (V, A) un grafo conexo no dirigido, ponderado con pesos positivos. Calcular un subgrafo conexo tal que la suma de las aristas seleccionadas sea mínima.

- Este subgrafo es necesariamente un árbol: árbol generador minimal (AGM) o árbol de recubrimiento mínimo (ARM) (en inglés, minimum spanning tree)

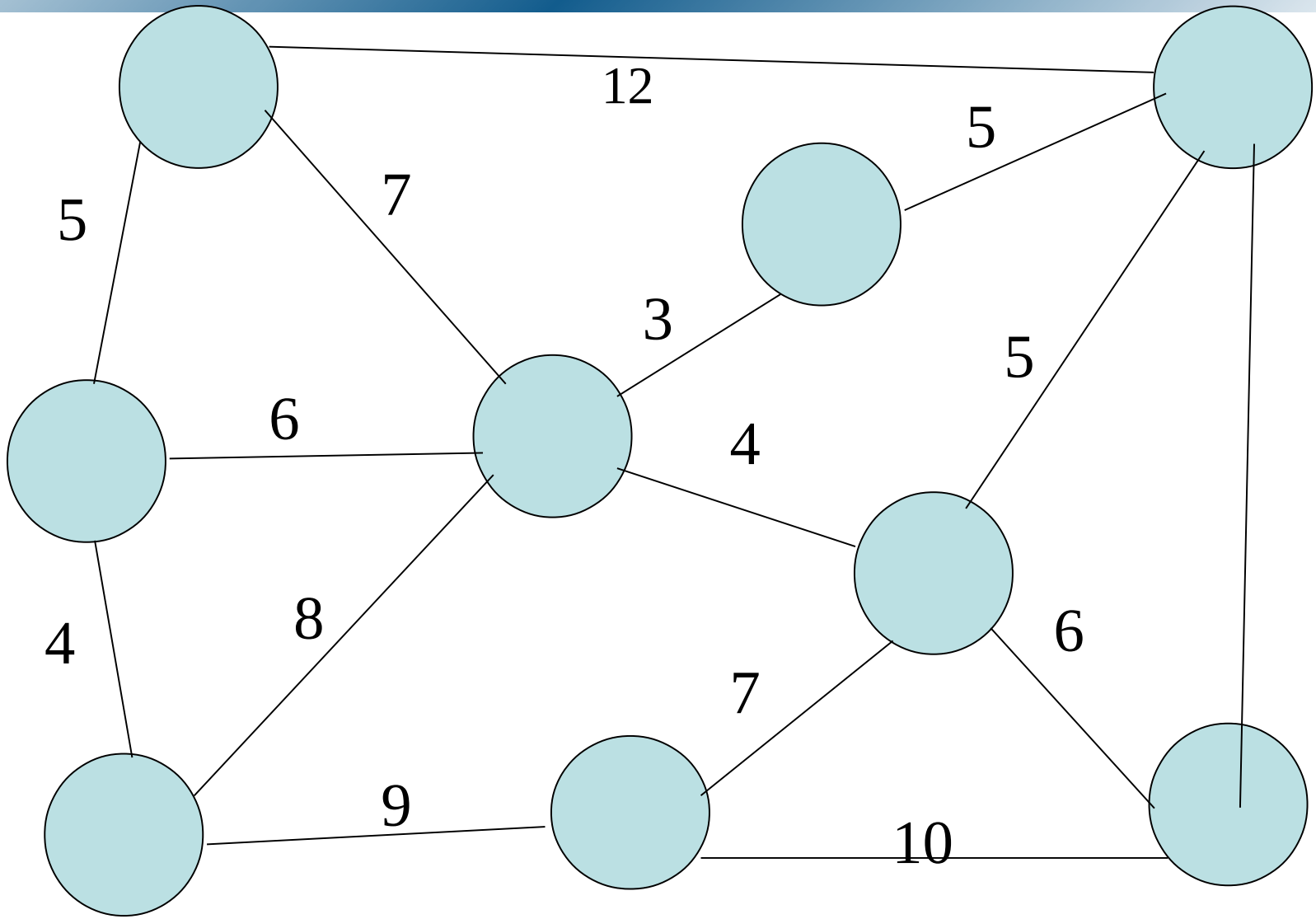
- Dos enfoques para la solución:
  - Basado en aristas: algoritmo de Kruskal
  - Basado en vértices: algoritmo de Prim

Árbol Generador Minimal

Joseph B. Kruskal (investigador del Math Center Bell-Labs), que en 1956 descubrió su algoritmo para la resolución del problema del Árbol Generador Minímal. Este problema es un problema típico de optimización combinatoria, que fue considerado originalmente por Otakar Boruvka (1926) mientras estudiaba la necesidad de electrificación rural en el sur de Moravia en Checoslovaquia.

- Las aplicaciones de este problema lo hacen muy importante.
  - Diseño de redes físicas.
    - teléfonos, eléctricas, hidráulicas, TV por cable, computadores, carreteras, ...
  - Análisis de Clusters.
    - Eliminación de aristas largas entre vértices irrelevantes
    - Búsqueda de cúmulos de quasars y estrellas
  - Solución aproximada de problemas NP.
    - PVC, árboles de Steiner, ...
  - Distribución de mensajes entre agentes
  - Aplicaciones indirectas.
    - Plegamiento de proteinas, Reconocimiento de células cancerosas, ...

Arbol generador minimal



Algoritmo de Kruskal

- Conjunto de candidatos: aristas

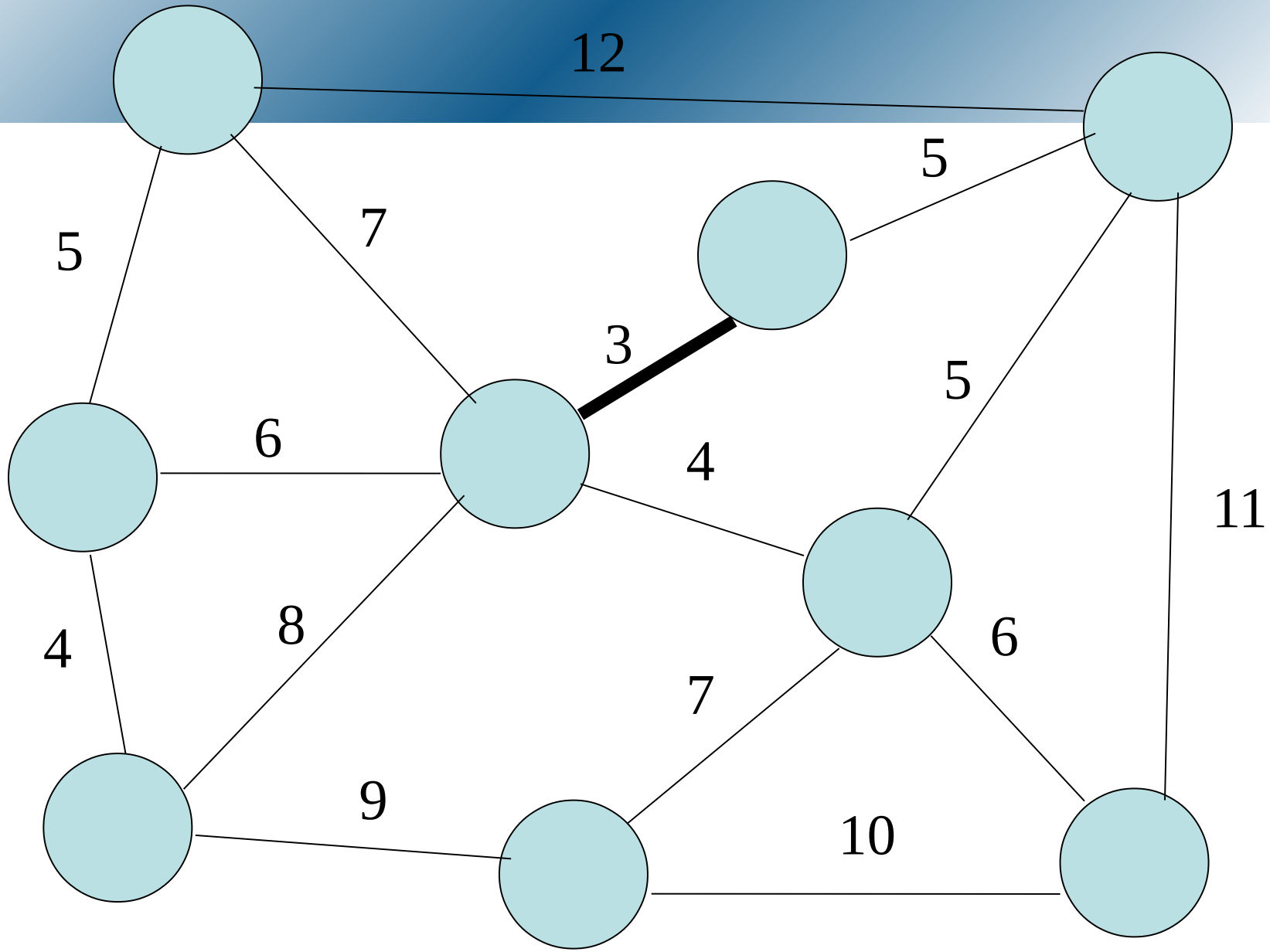
- Función Solución: un conjunto de aristas que conecta todos los vértices

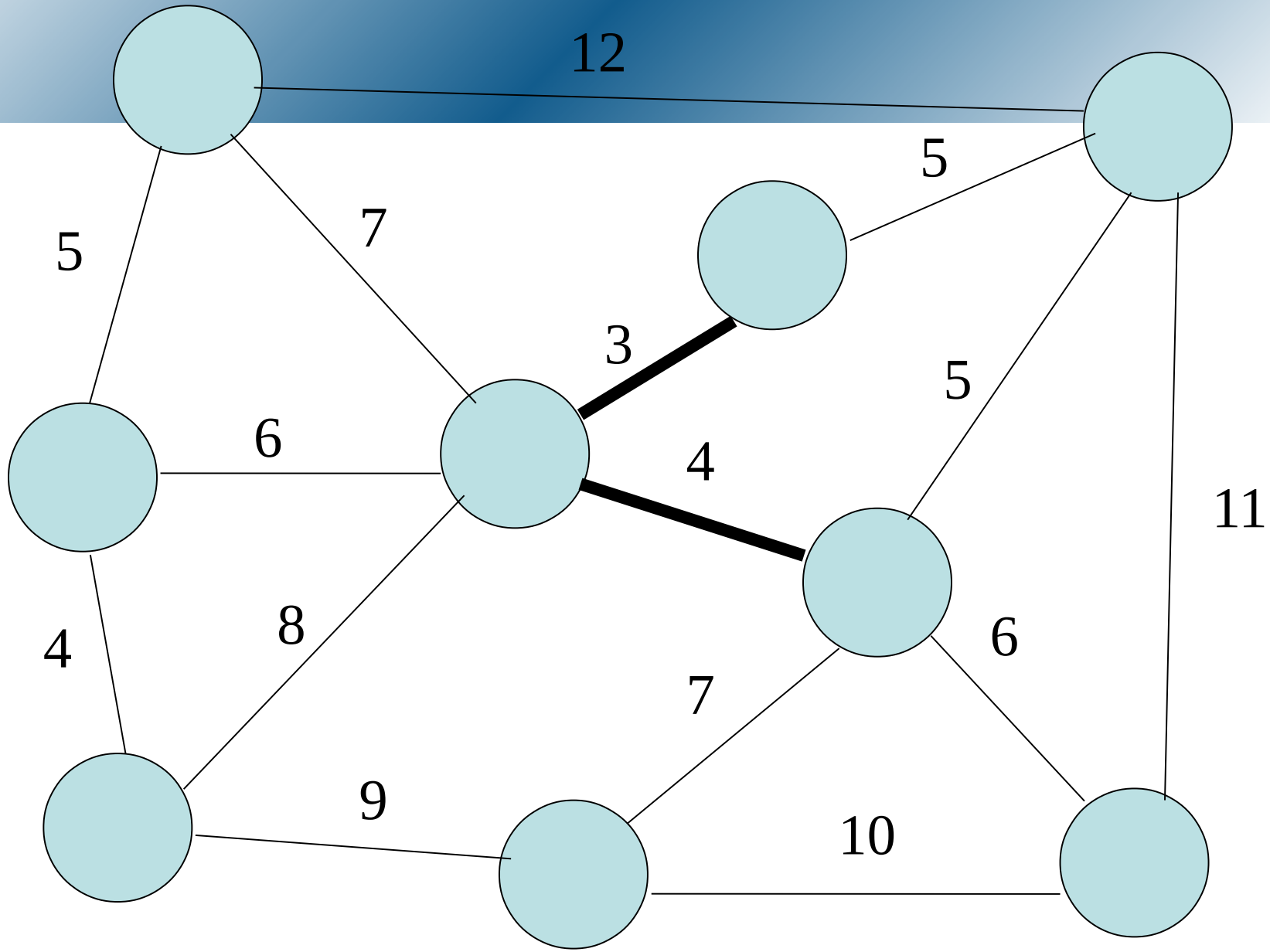
  Se ha construido un árbol de recubrimiento (n-1 aristas seleccionadas).

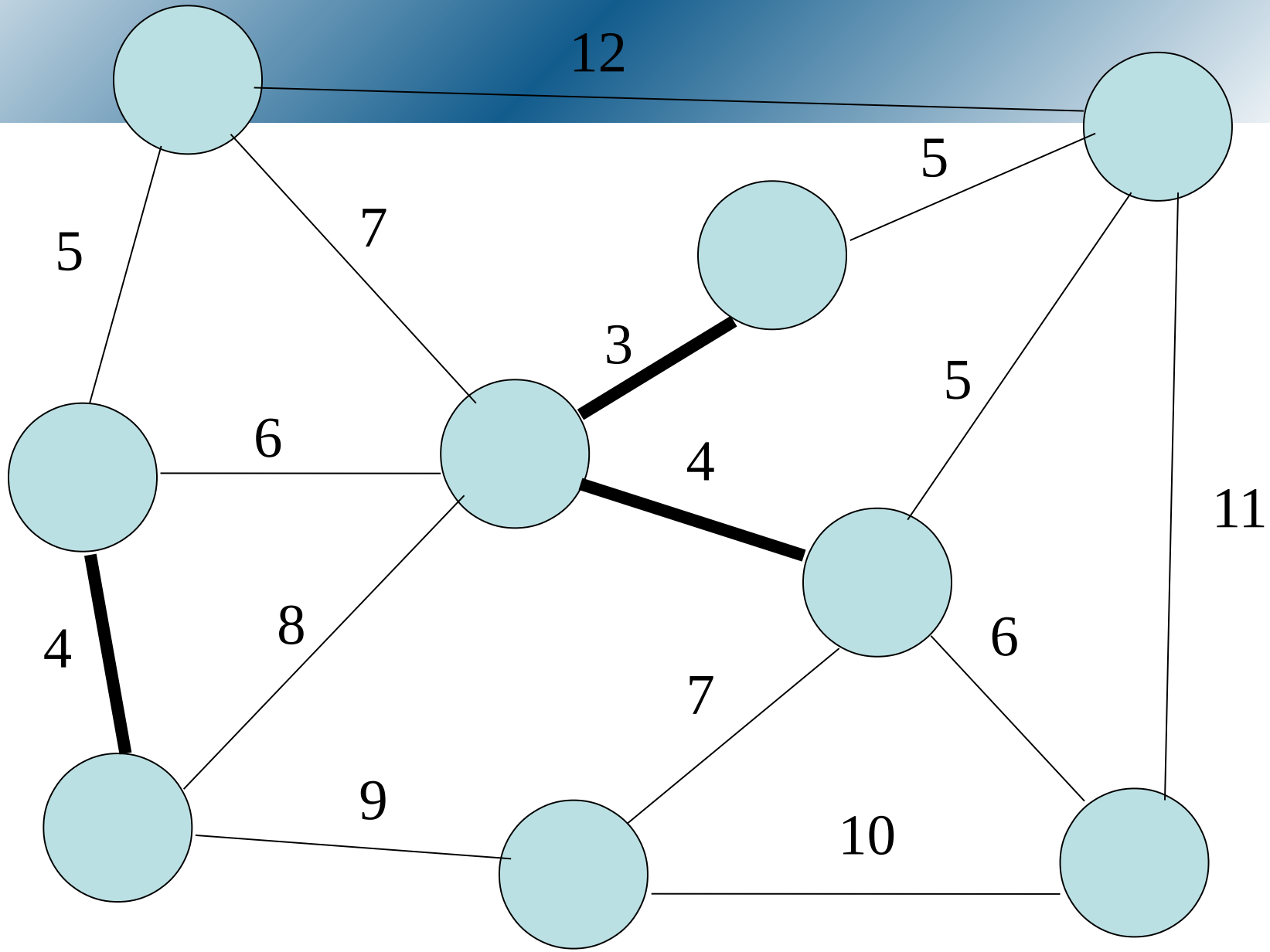
- Función factible: el conjunto de aristas no forma ciclos
- Función selección: la arista de menor coste
- Función objetivo: minimizar la suma de los costes de las aristas

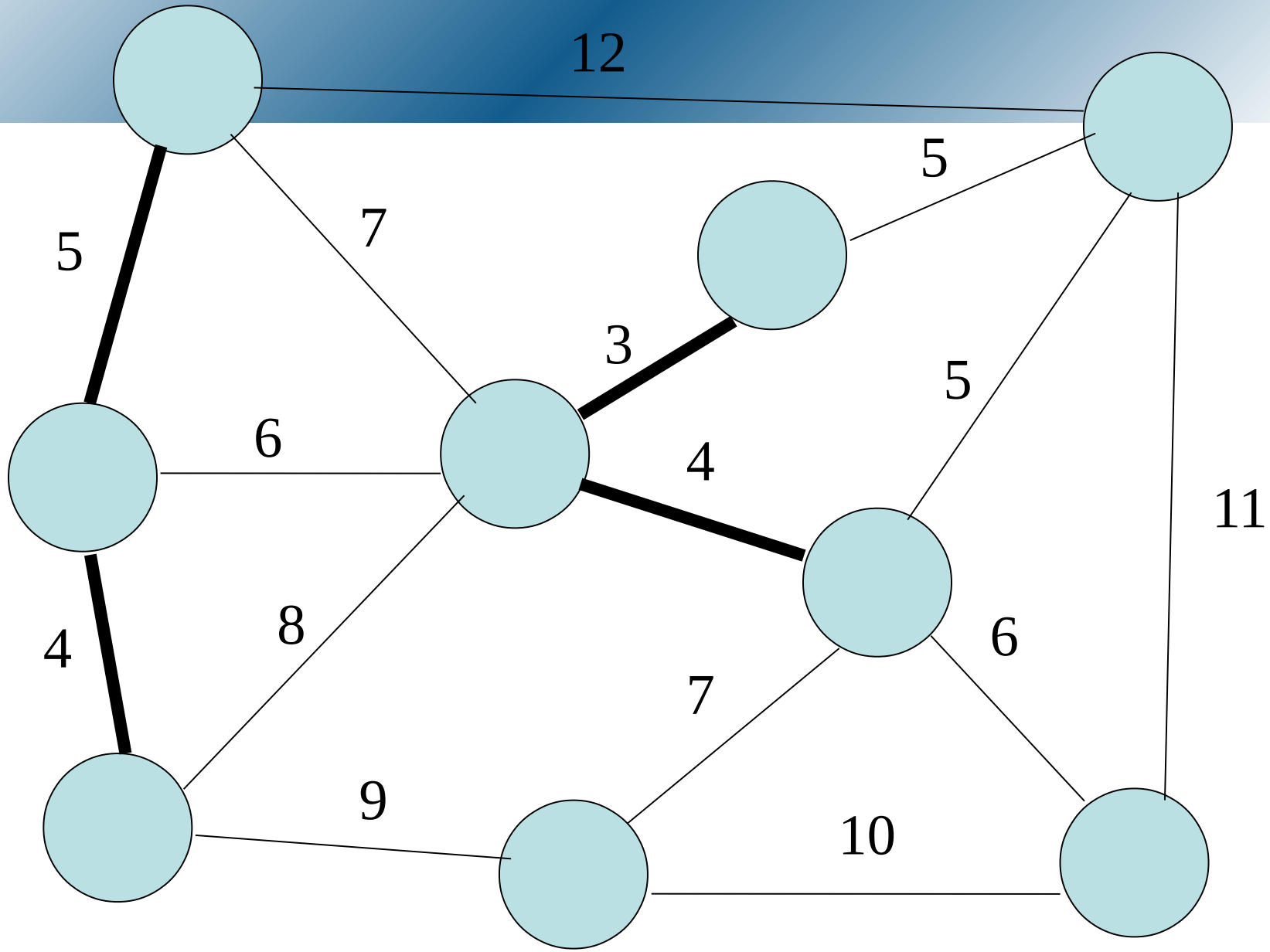
Algoritmo de Kruskal

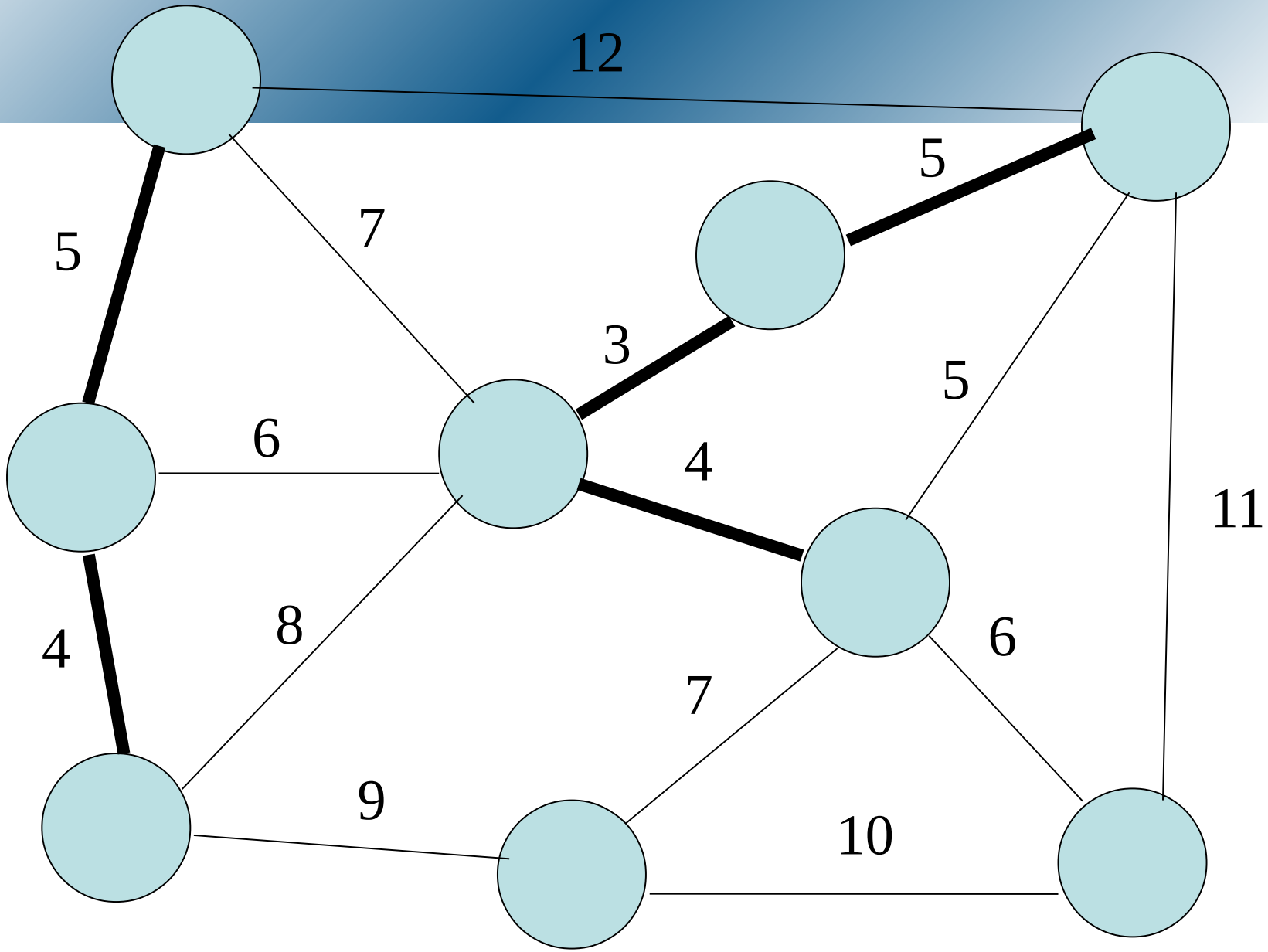
```
 Kruskal_I (Grafo G(V,A))
{ set<arcos> C(A);
  set<arcos> S;                  // Solución inic. Vacía
  Ordenar(C);    // de menor a mayor costo
  while (!C.empty() && S.size()!=V.size()-1) { //No solución
     x = C.first(); //seleccionar el menor
     C.erase(x);
      if (!HayCiclo(S,x))  //Factible
         S.insert(x);
      }
   if (S.size()==V.size()-1) return S;      // Hay solución
   else return “No_hay_solucion”;
}
```

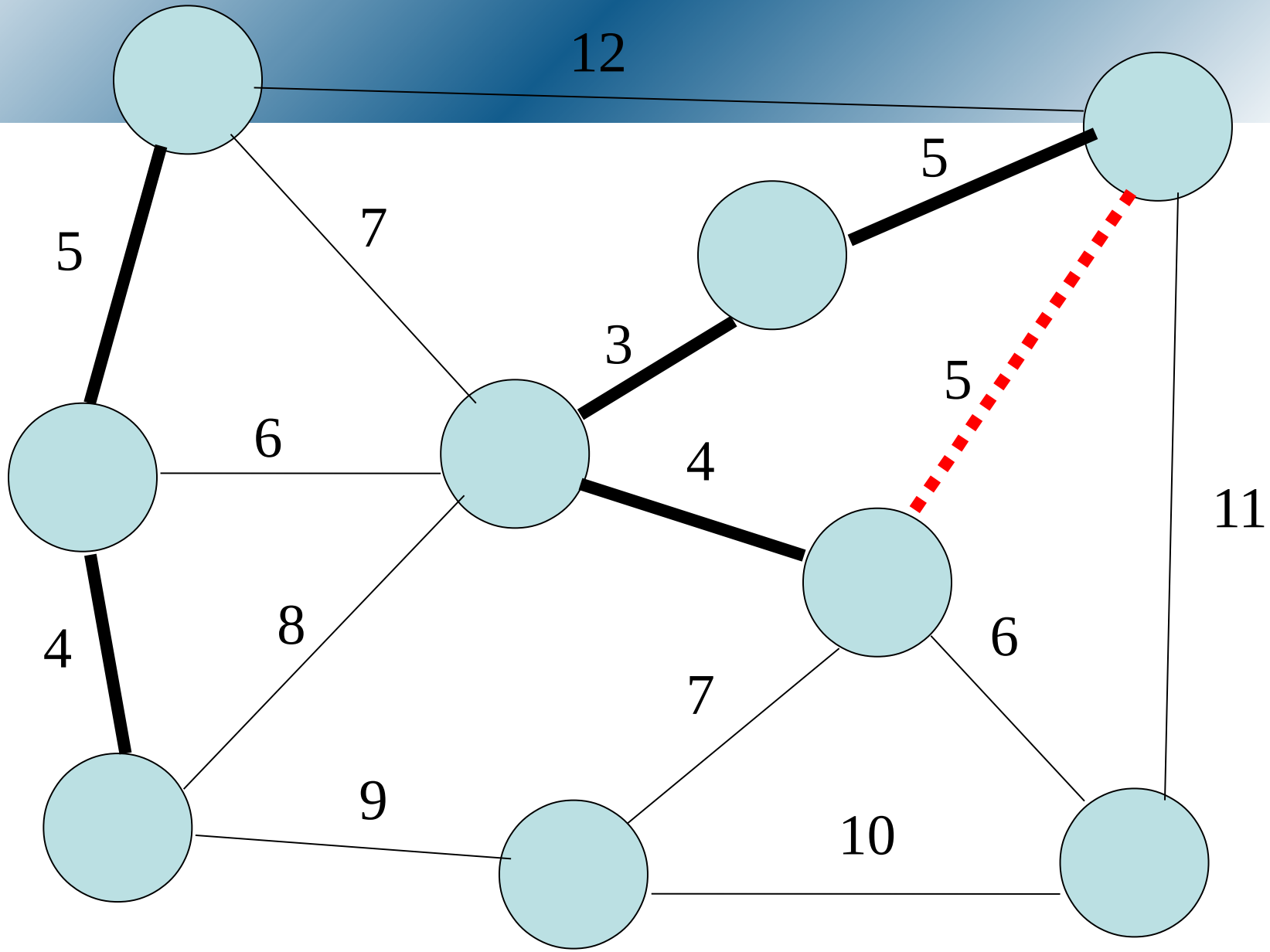

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

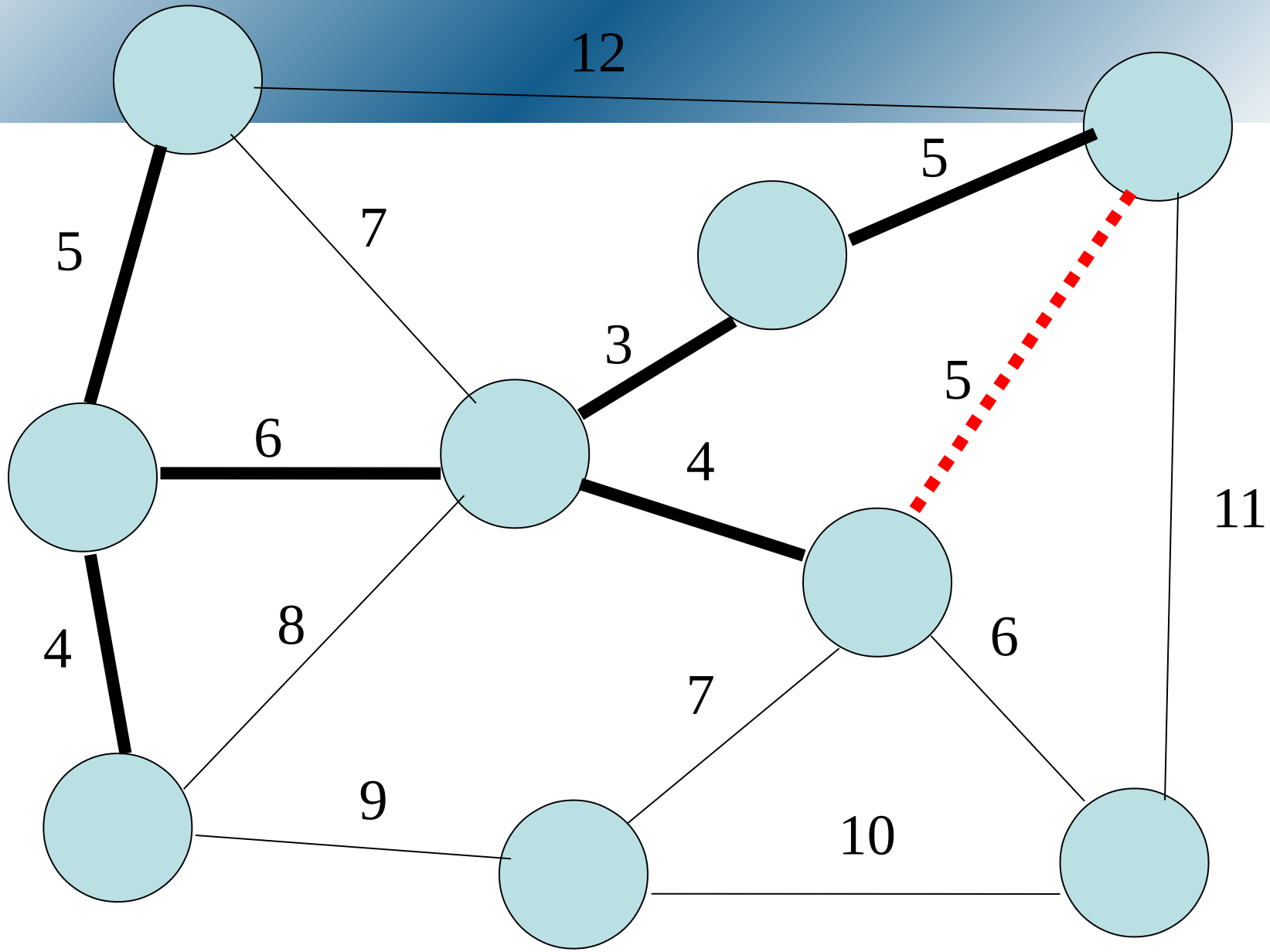

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

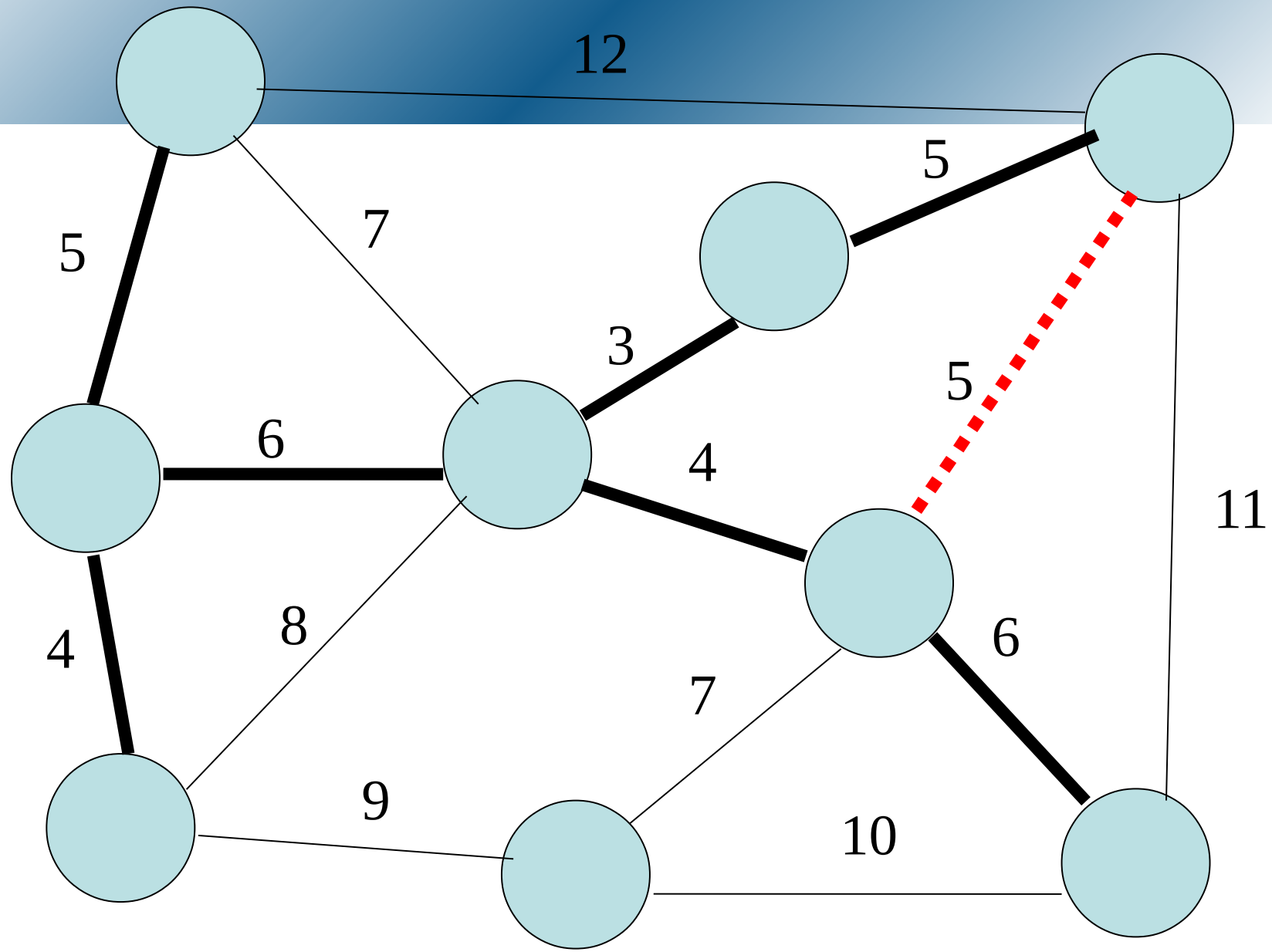

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

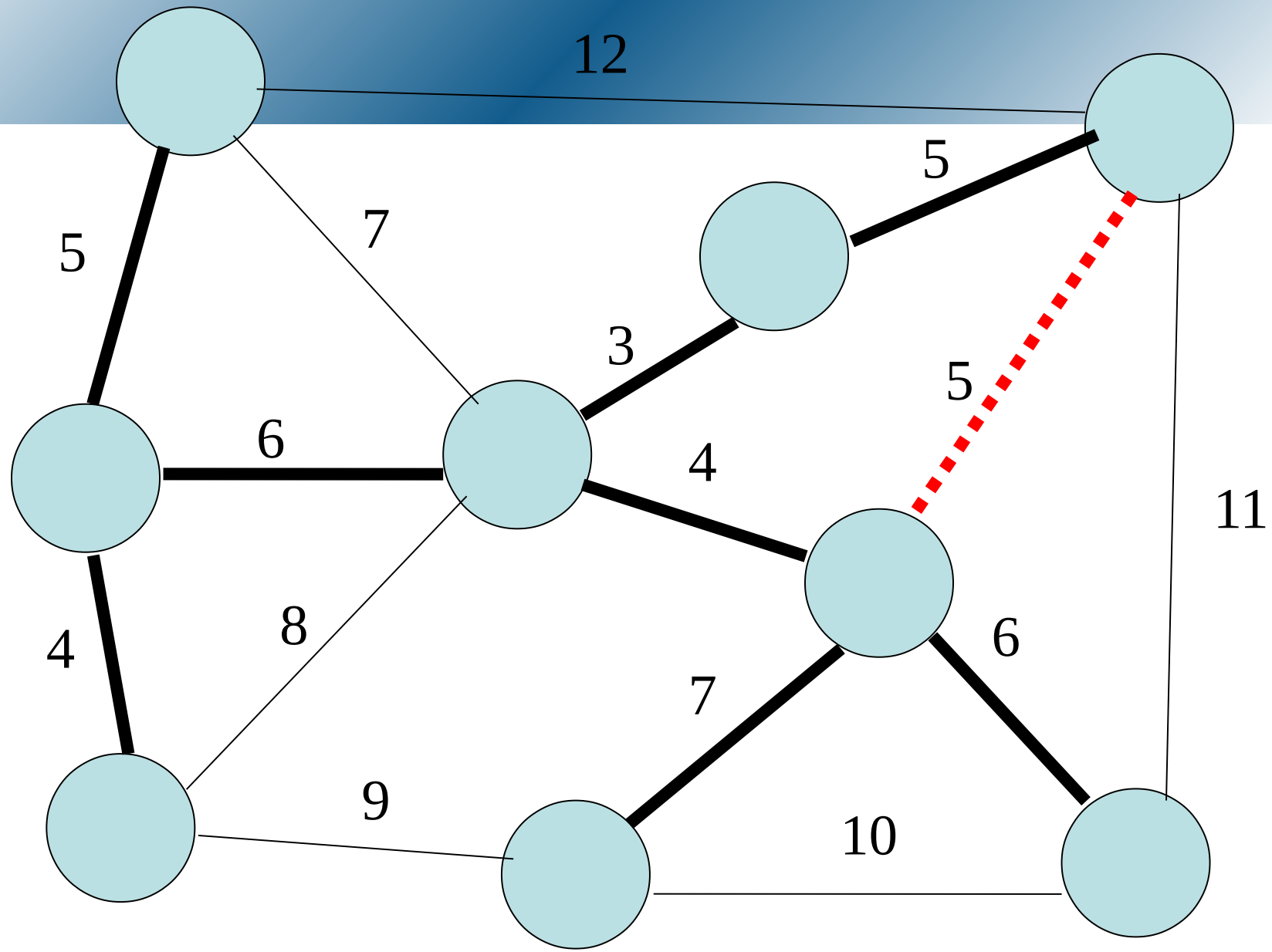

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

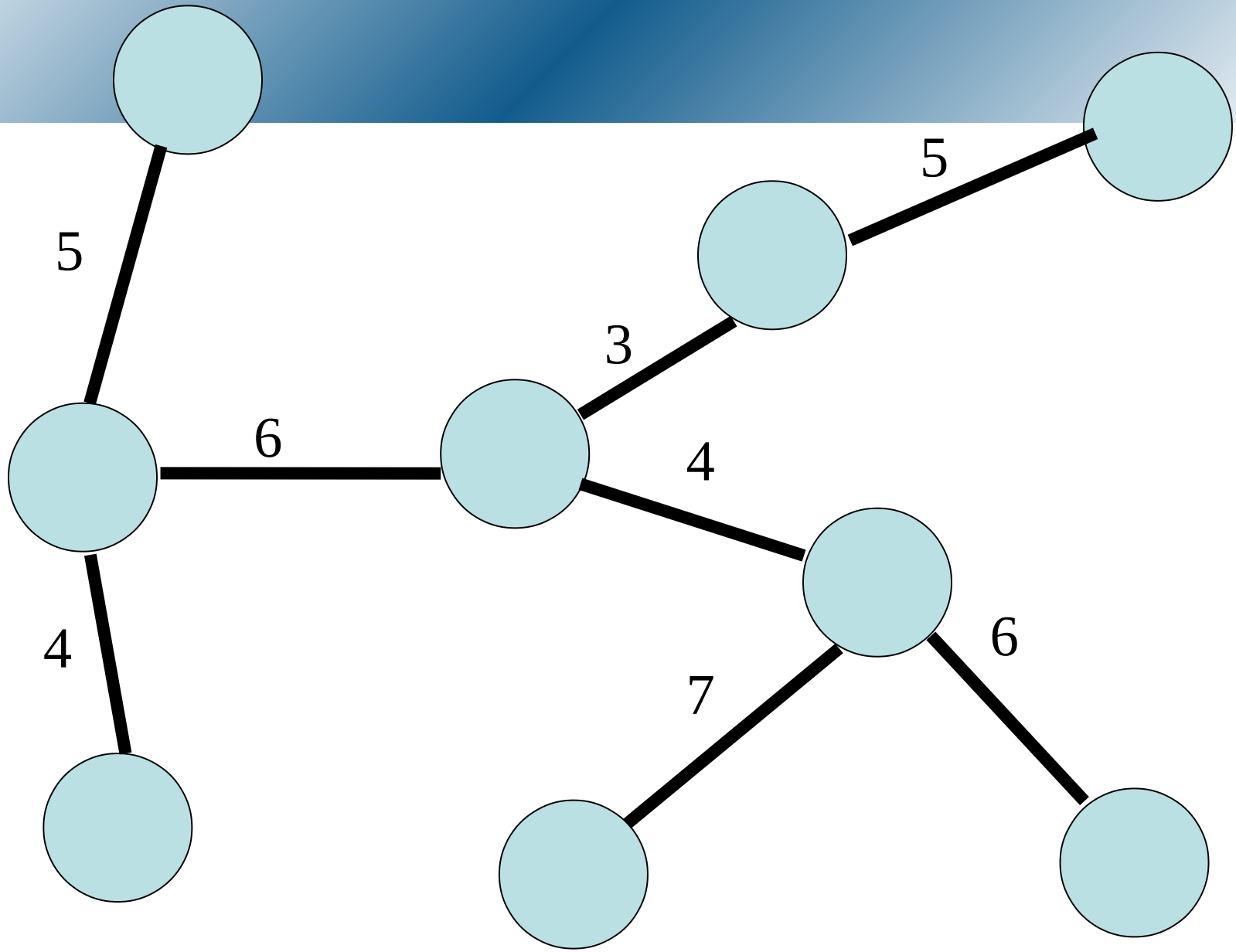

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10


12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

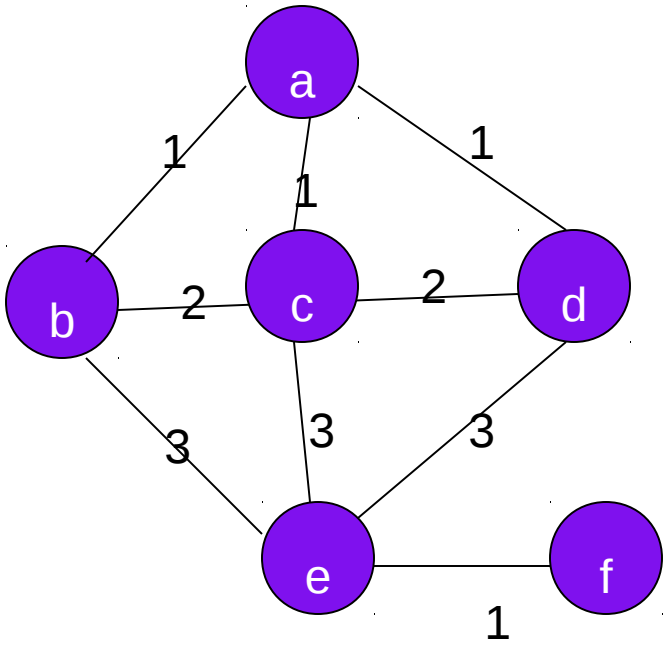

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

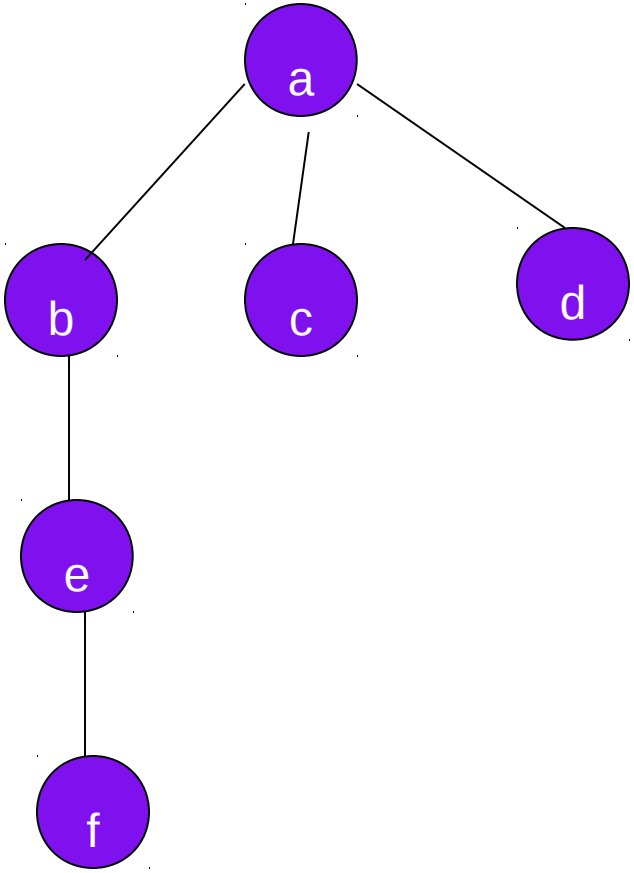

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10


12
5
7
5
3
5
6
4
11
4
8
6
7
9
10


5
5
3
6
4
4
7
6

Ejemplo





(a,b), (a,c), (a,d), (e,f), (b,c), (c,d), (b,e), (c,e), (d,e)

Optimalidad de Kruskal

Teorema:

El algoritmo de Kruskal halla un árbol de recubrimiento mínimo.

Demostración: Inducción sobre el número de aristas que se han incluido en el ARM.

Optimalidad de Kruskal

Base: Sea k1 la arista de menor peso en A, entonces existe un ARM optimal *T* tal que {k1} *pertenece a T.*

Suponemos cierto para m-1.

La arista (m-1)-esima en incluirse por el algoritmo de Kruskal pertenece a un ARM T optimal.

Demostramos que es cierto para m

La arista m-esima incluida por el algoritmo de Kruskal pertenece a un ARM T optimal.

Optimalidad de Kruskal

Demostración *Caso Base:* *(Red. Absurdo)*

Supongamos un ARM óptimo T' que no incluye a k1.
Consideremos T'∪k1 con

peso(T'∪k1) = peso(T') + peso(k1).

En este caso aparece un ciclo (¿por qué?). Eliminemos cualquier arista del ciclo, (x), distinta de k1. Al eliminar la arista obtenemos un árbol T*=T'+ {k1} - {x} con peso

peso(T*) = peso(T') + peso(k1) – peso(x).

Si tenemos peso(k1) < peso(x) entonces deducimos que

peso(T*) < peso(T'). !!! Contr.

Optimalidad de Kruskal

Paso Inducción: Red. Absurdo.

Supongamos un ARM óptimo T' que incluyendo a {k1,..., k(m-1)} no incluye a km. Consideremos T'∪km con

$$peso(T' \cup km) = peso(T') + peso(km).$$

En este caso aparece un ciclo, que incluirá al menos una arista x que NO pertenece al conjunto de aristas seleccionadas {k1, ..., km} (¿por qué?). Eliminando dicha arista del ciclo, obtenemos un árbol T=T'+km-x con peso*

$$peso(T^*) = peso(T') + peso(km) - peso(x).$$

Si tenemos peso(km) < peso(x) (porqué?) entonces deducimos que peso(T) < peso(T'). !!! Contr.*

Eficiencia Kruskal (Directa)

```
 Kruskal_I(Grafo G(V,A))
{ vector<arcos> C(A);
  Arbol<arcos> S;                    // Solución inic. Vacía
  Ordenar(C);                        O(A log A) = O(A log V) xq?
  while (!C.empty() && S.size()!=V.size()-1) {   O(1)
     x = C.first(); //seleccionar el menor       O(1)
     C.erase(x);                                 O(1)      } O(AV)
      if (!HayCiclo(S,x))  //Factible            O(V)
         S.insert(x);                            O(1)
      }
   if (S.size()==V.size()-1) return S;      // Hay solución
   else return "No_hay_solucion";
}
```

Ciclos

¿Cómo determino que una arista no forma un ciclo?

Comienzo con un conjunto de componentes conexas de tamaño n (cada nodo en una componente conexa).

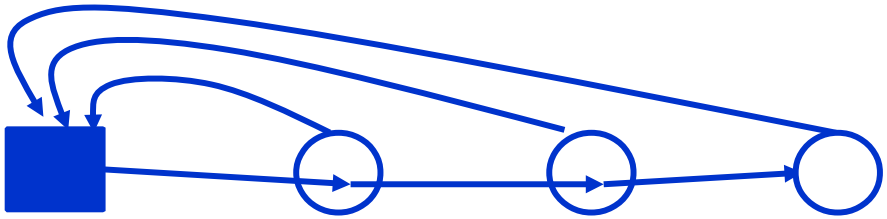
La función factible me acepta una arista de menor costo que una dos componentes conexas, así garantizamos que no hay ciclos.

En cada paso hay una componente conexa menos, finalmente termino con una única componente conexa que forma el árbol generador minimal.

```
Kruskal( G(V,A) )
{  S = 0;  // Inicializamos S con el conjunto vacio
   for (i=0; i< V.size()-1; i++)
        MakeSet(V[i]);    // Conjuntos vertice v[i]
   Ordenar(A) //Orden creciente de pesos
 while (!A.empty() && S.size()!=V.size()-1) {//No
    solución
    (u,v) =A.first(); //Sel. el menor arco (y eliminar)
    compu=FindSet(u); compv=FindSet(v);
    if (compu != compv)
        S = S U {{u,v}};
        Union(compu,compv); // Unimos los dos conj.
    }
}
```

Eficiencia del Algoritmo de Kruskal

- Representación para conjuntos disjuntos
  - Listas enlazadas de elementos con punteros hacia el conjunto al que pertenecen



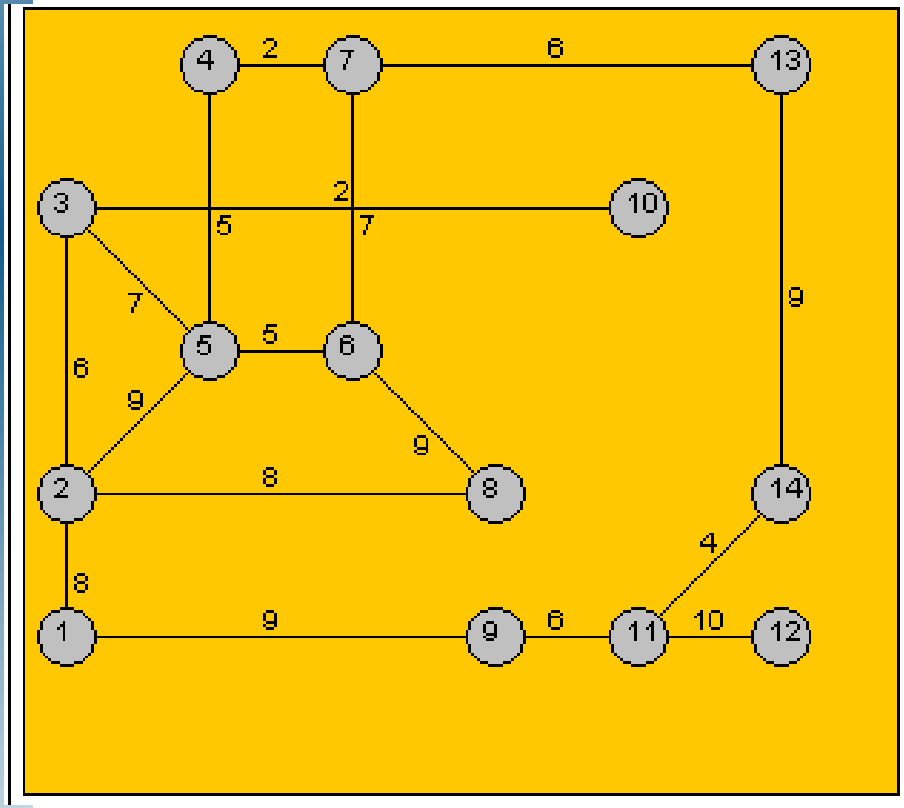
- MakeSet(): O(1)
- FindSet(): O(1)
- Union(A,B): O(log V)

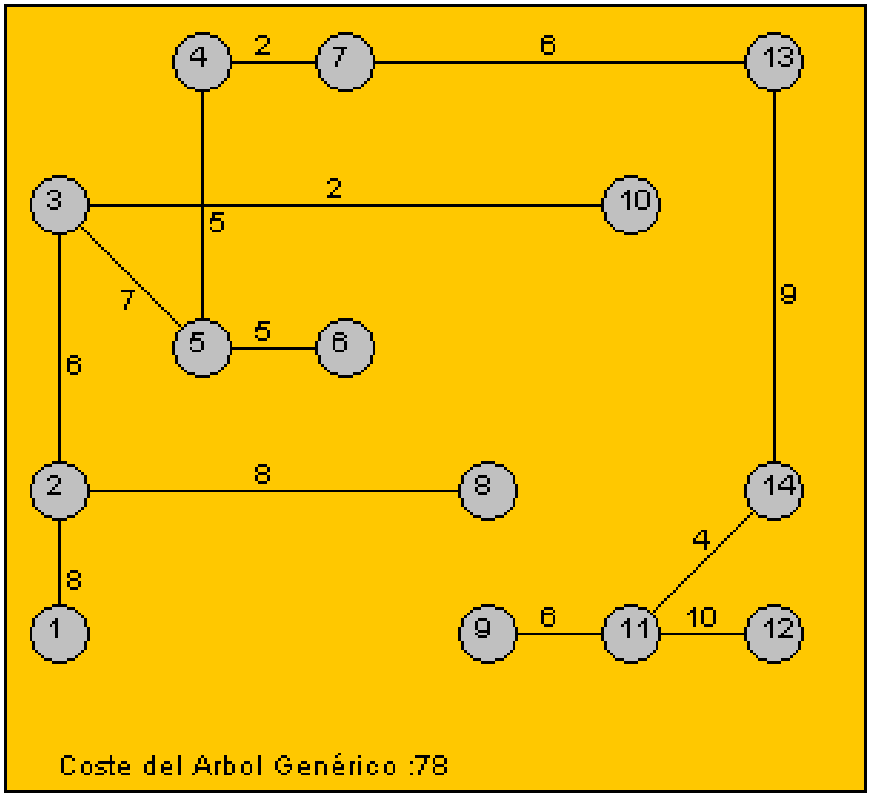
Análisis de Eficiencia

```
Funcion Kruskal(G(V,A)){
   S = 0;  // Inicializamos S con el conjunto vacio
   for (i=0; i< V.size()-1; i++)
     MakeSet(V[i]);        O(1)
  Ordenar(A) //Orden creciente de pesos   O(A log A)
  while (!A.empty() && S.size()!=V.size()-1) { //No solución
     (u,v) =A.first(); O(1)
     compu=FindSet(u); O(1)
     compv=FindSet(v); O(1)
     if (compu != compv) O(1)
        S = S U {{u,v}}; O(1)
        Union(compu,compv); // O(log V)
     }
}
```

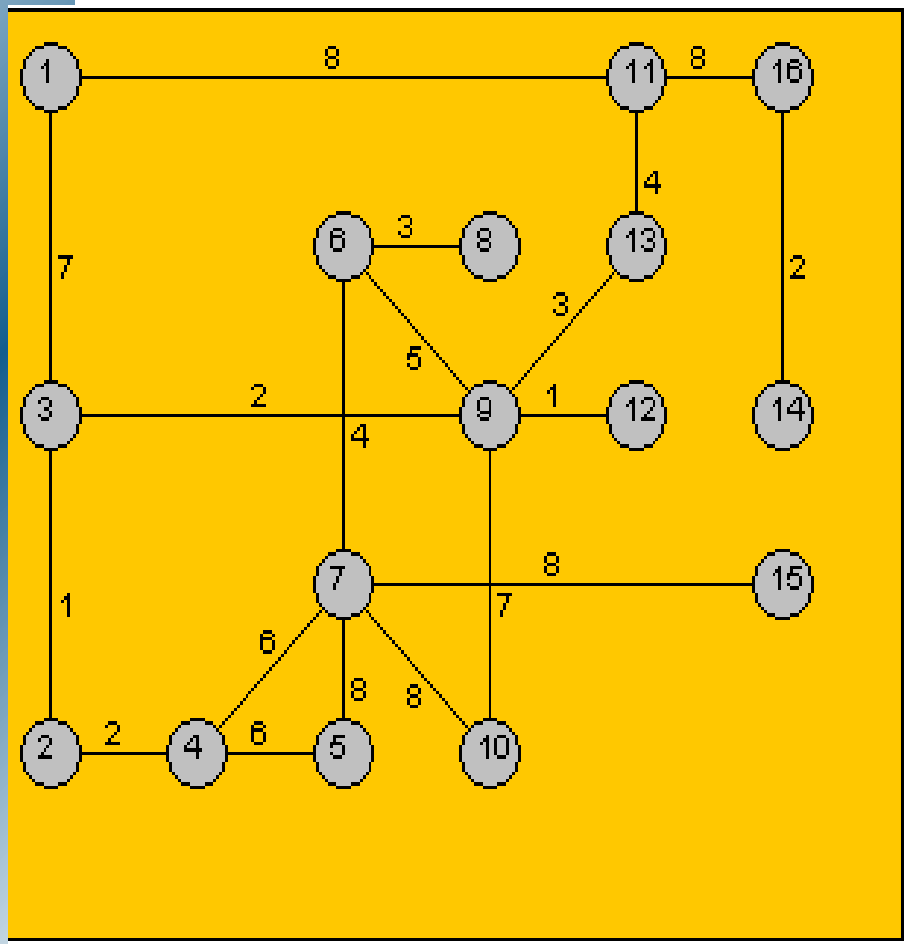
Kruskal es de O(A log V)

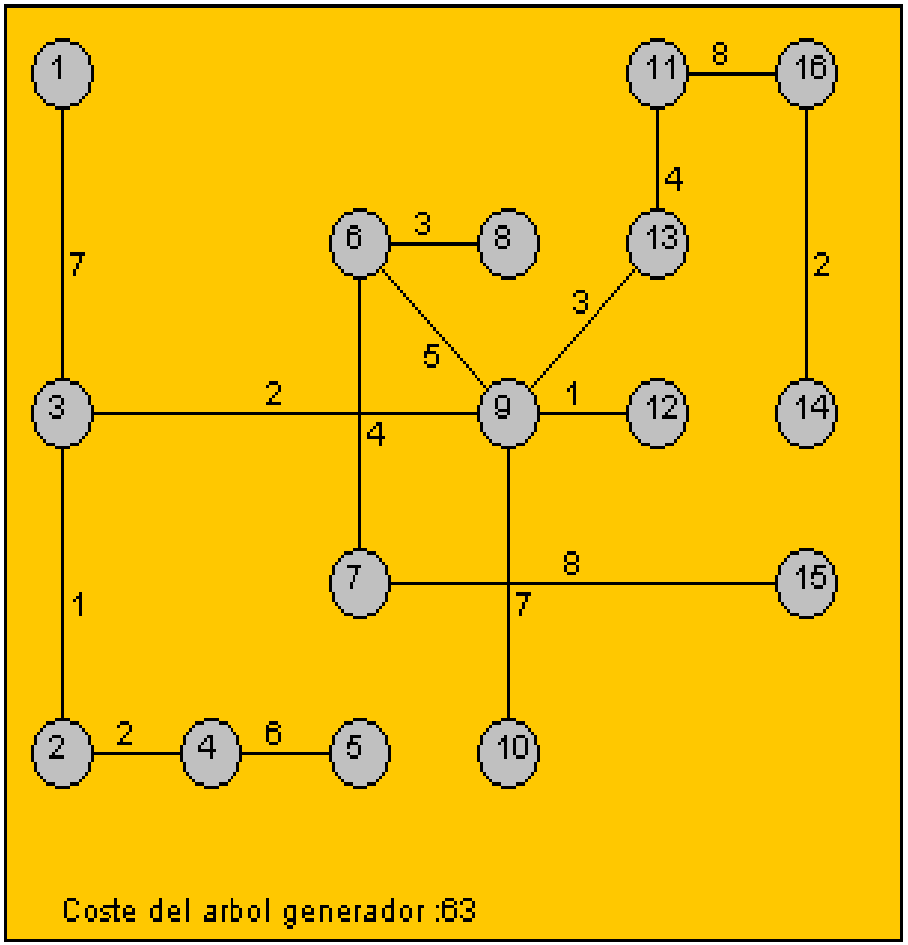
Otro ejemplo de Kruskal





Y otro más


1
8
11
8
16
4
6
3
8
13
7
3
2
5
3
2
9
1
12
14
4
7
8
15
1
7
6
8
8
2
2
4
6
5
10

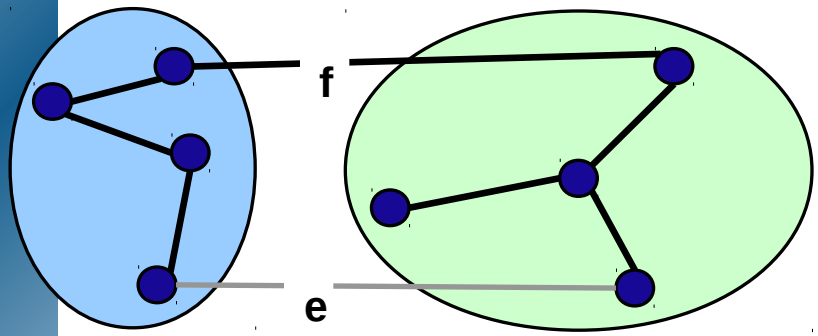

1
11
8
16
4
6
3
8
13
7
3
2
5
3
2
9
1
12
14
4
7
8
15
1
7
2
2
4
6
5
10
Coste del arbol generador :63

Algoritmo de Prim

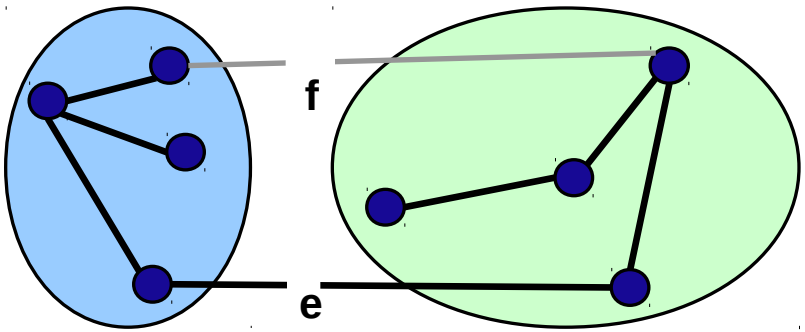
- Se verifica la **Propiedad del AGM**:
- Sea G = (V,A) un grafo no dirigido y conexo donde cada arista tiene una longitud conocida. Sea $U \subseteq V$ un subconjunto propio (lo que significa que U no puede coincidir con V) de los nodos de G. Si (u,v) es una arista tal que $u \in U$ y $v \in V - U$ y, además, es la arista del grafo que verifica esa condición con el menor peso, entonces existe un AGM T que incluye a (u,v).
- Esta propiedad garantizará que el nuevo algoritmo greedy que diseñemos proporcione la solución óptima del problema.

Demostración de la propiedad del AGM

- Demostración por contradicción:
  Supongamos que T es un AGM que no contiene a e = (u,v)



T



T*

- Si añadimos e a T se crea un ciclo C.
- Si rompemos ese ciclo (eliminando una arista f conectando U y V-U), obtenemos un nuevo AG T* que (por incluir a e) tendra menor longitud que T. Eso es una contradicción

Algoritmo de Prim

- El **Algoritmo de Prim**, es otro método de

resolver el problema del AGM que se basa en:

  - Construcción del algoritmo en función de la propiedad del AGM.
  - En el algoritmo de Kruskal partíamos de la selección de la arista más corta que hubiera en la lista de aristas, lo que implica un crecimiento desordenado del AGM.
  - Para evitarlo, el algoritmo de Prim propone que el crecimiento del AGM sea ordenado.
  - Para ello aplica el algoritmo a partir de una raíz, lo que no implica restricción alguna.

Algoritmo de Prim

Candidatos: *Vértices*

Función Solución: Se ha construido un árbol de recubrimiento (n vértices seleccionadas).

Función Selección: Seleccionar el vértice u del conjunto de no seleccionados que se conecte mediante la arista de menor peso a un vértice v del conjunto de vértices seleccionados. La arista (u,v) está en T.

Función de Factibilidad: *El conjunto de aristas entre los vértices seleccionados no contiene ningún ciclo.*

Está implícita en el proceso

Función Objetivo: determina la longitud total de las aristas seleccionadas.

Implementación del Algoritmo de Prim

FUNCION PRIM (G = (V, A)) conjunto de aristas.

(Inicialización)

T = ∅ (Contendrá las aristas del AGM que buscamos).

U = {un miembro arbitrario de V}

MIENTRAS |U| != n HACER

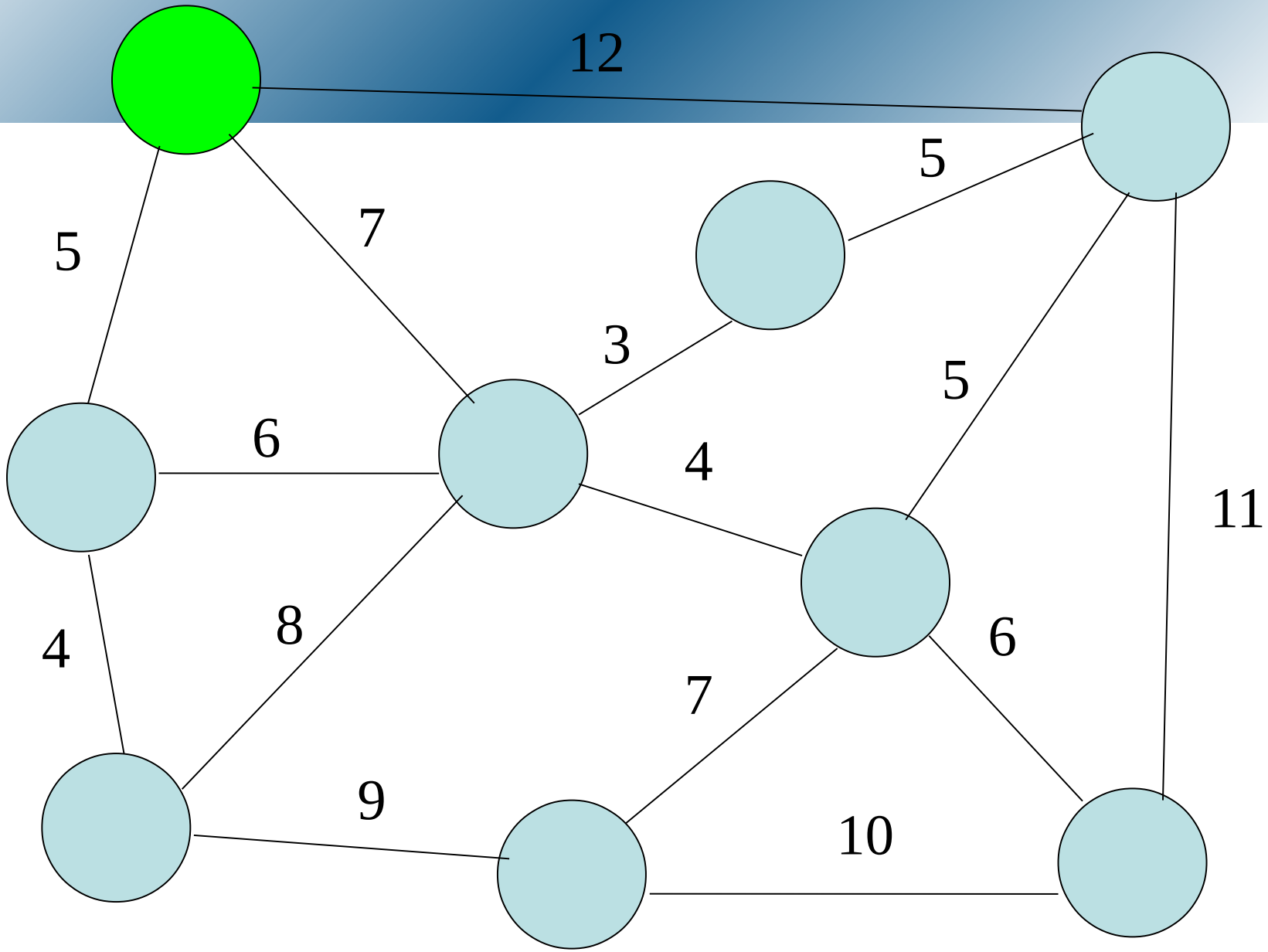
BUSCAR e = (u,v) de longitud mínima tal que

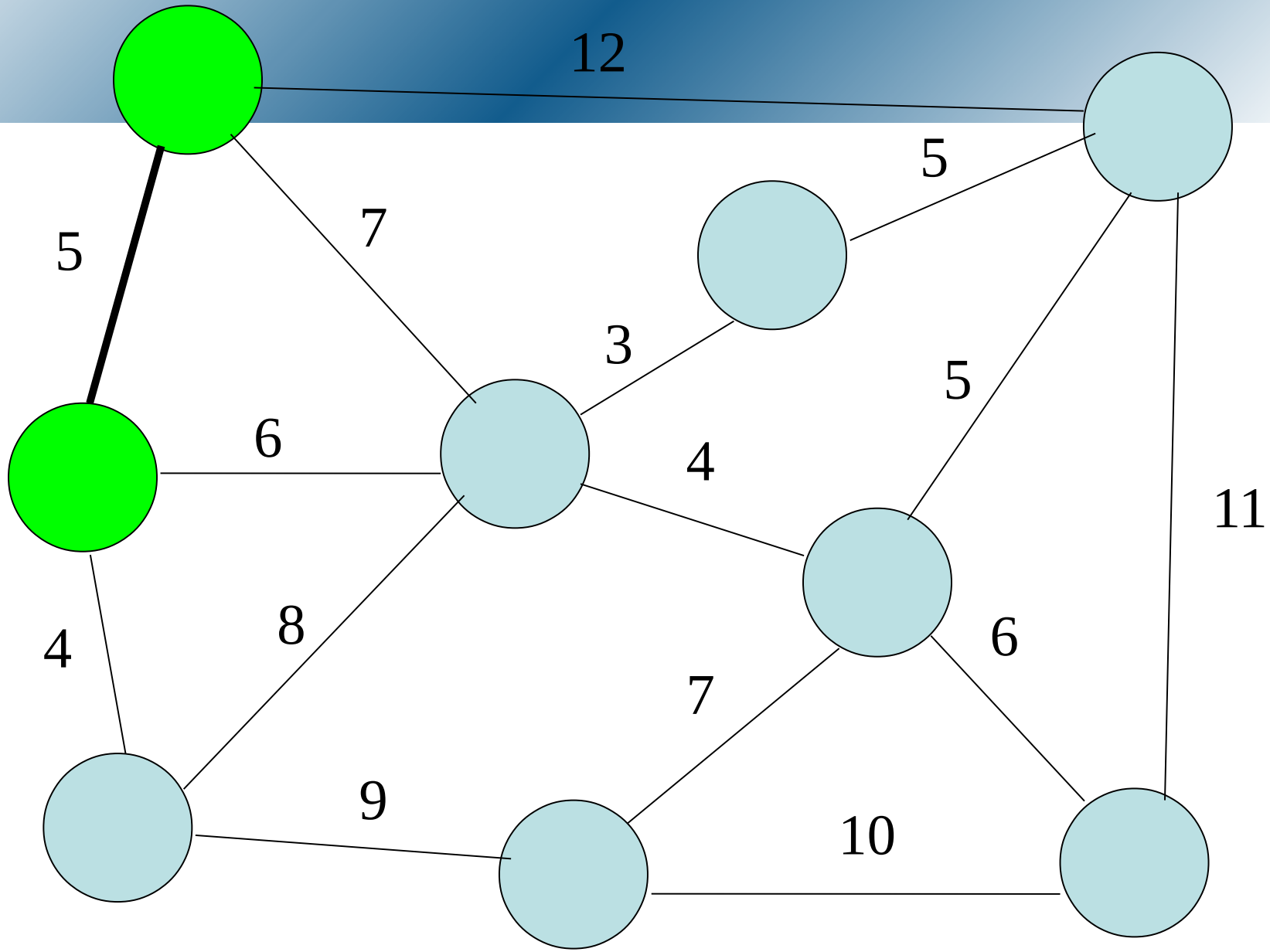
$u \in U$ y $v \in V - U$

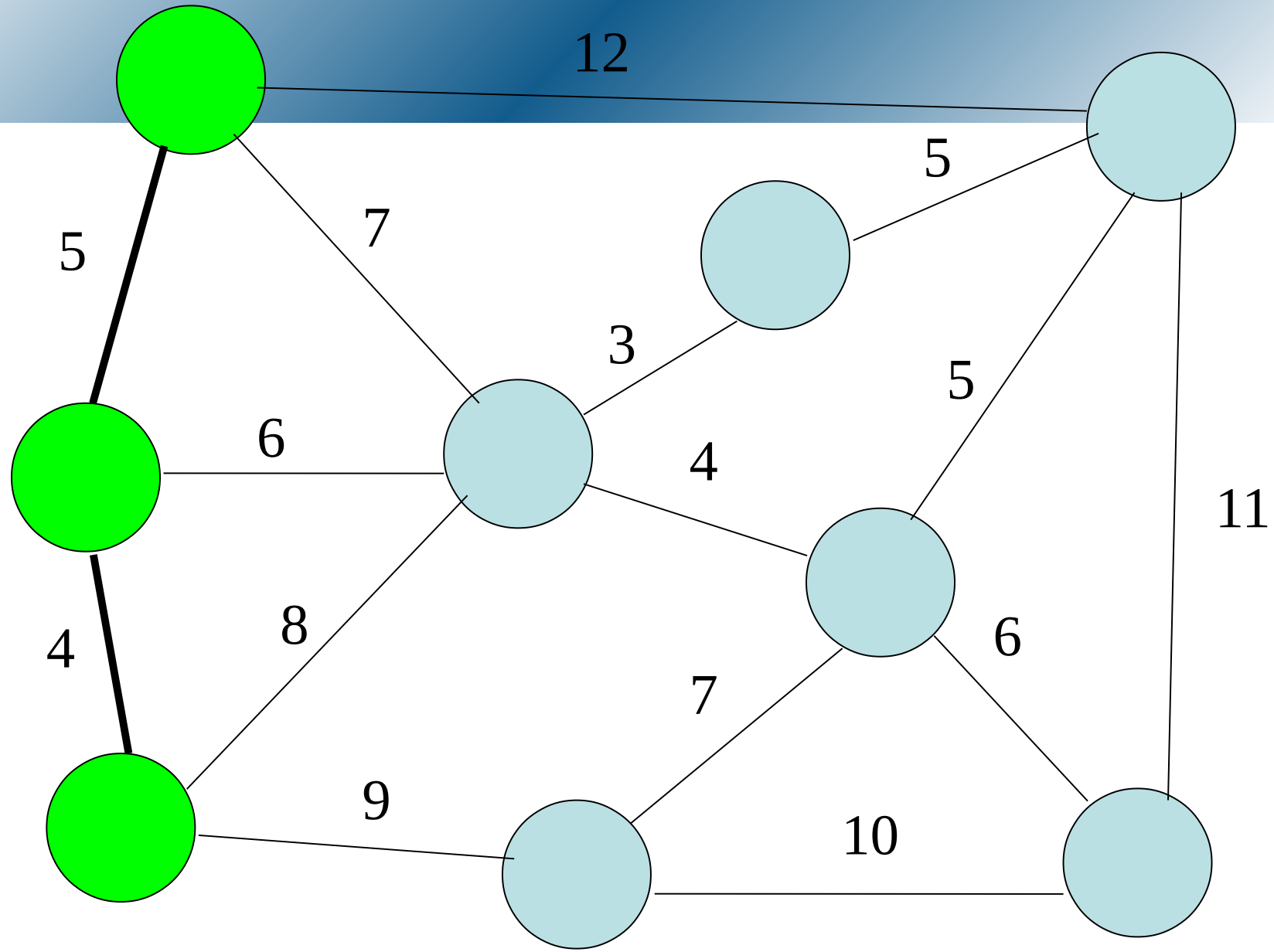
T = T + e

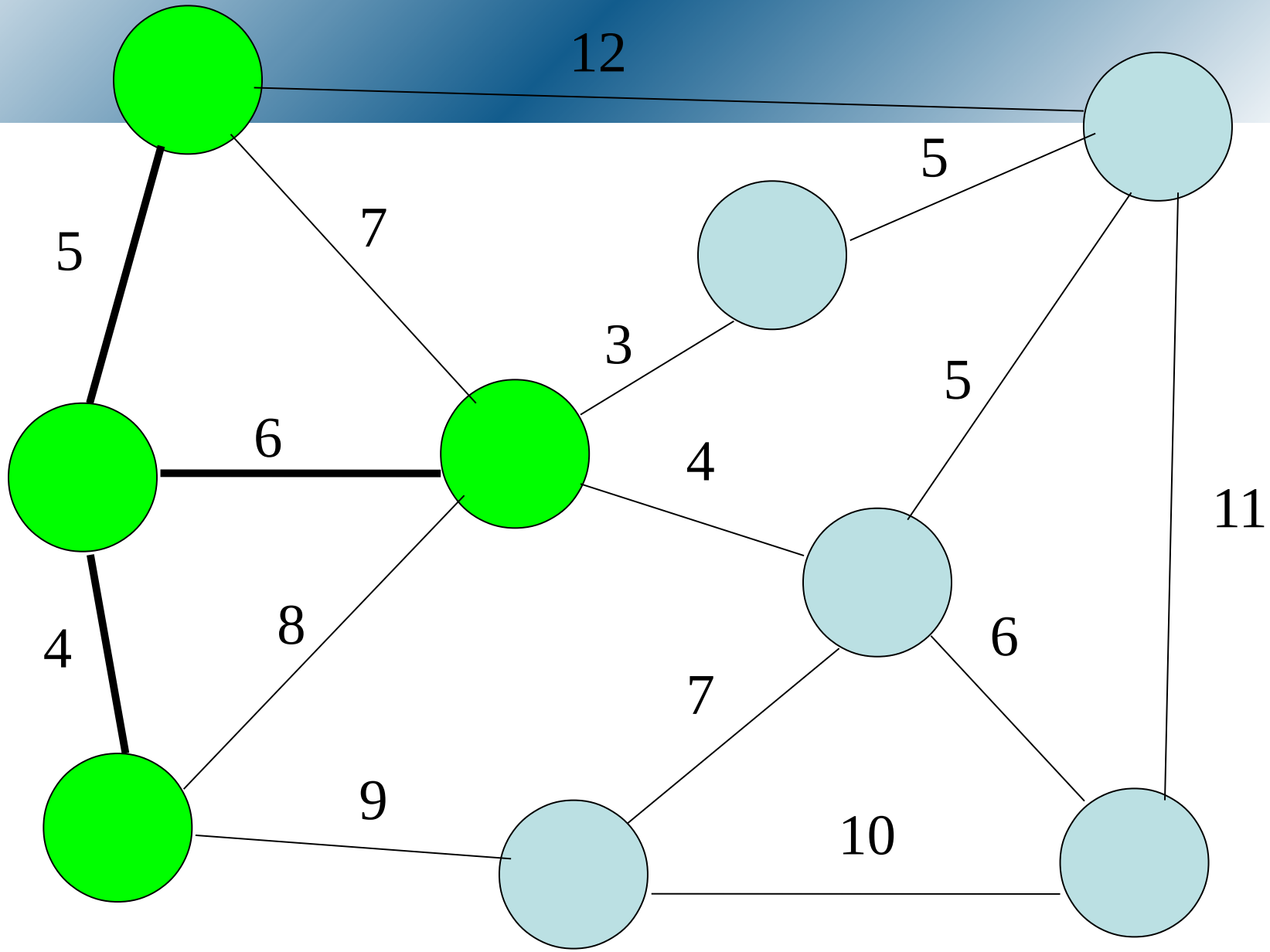
U = U + v

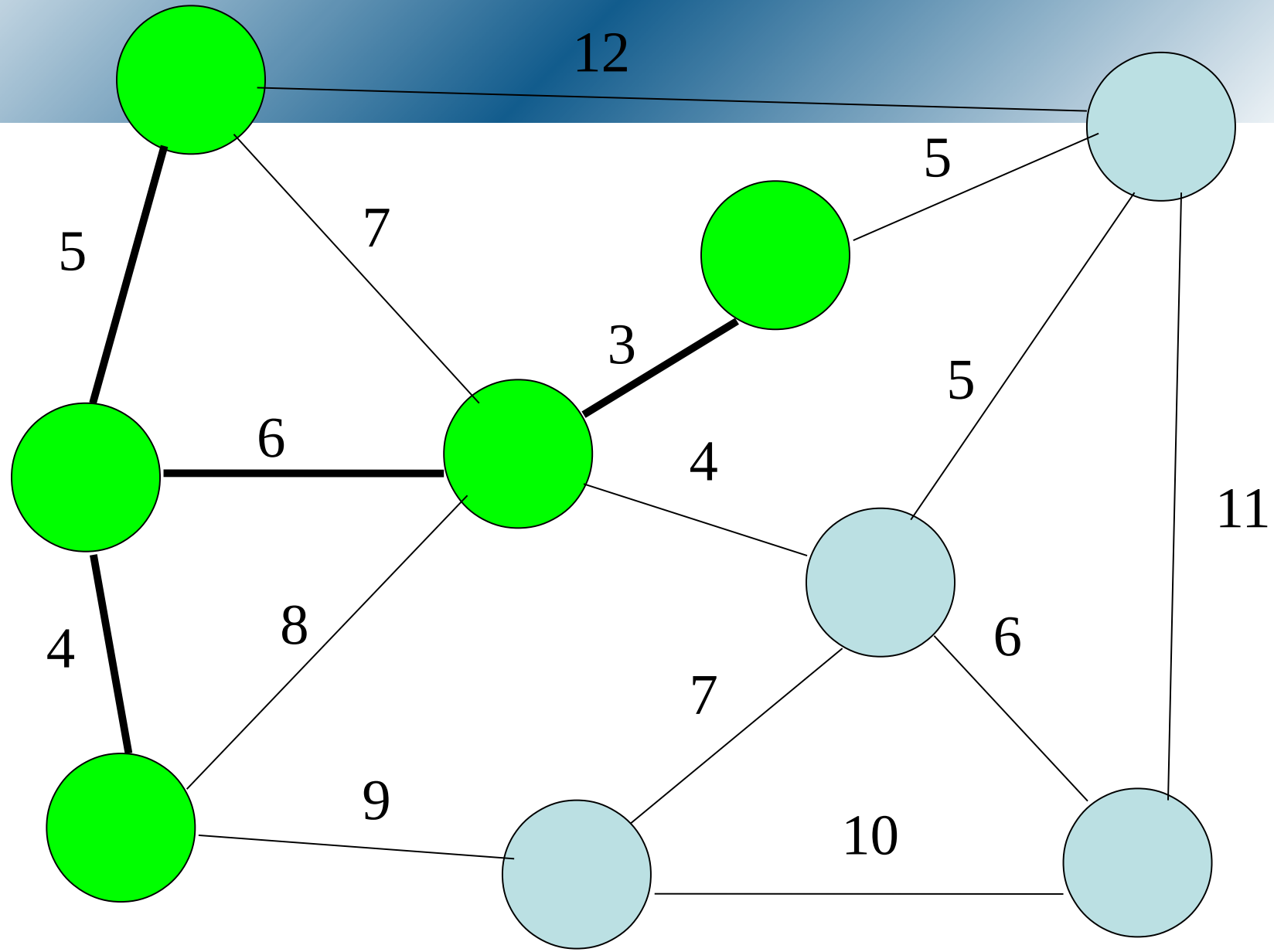
DEVOLVER (T)

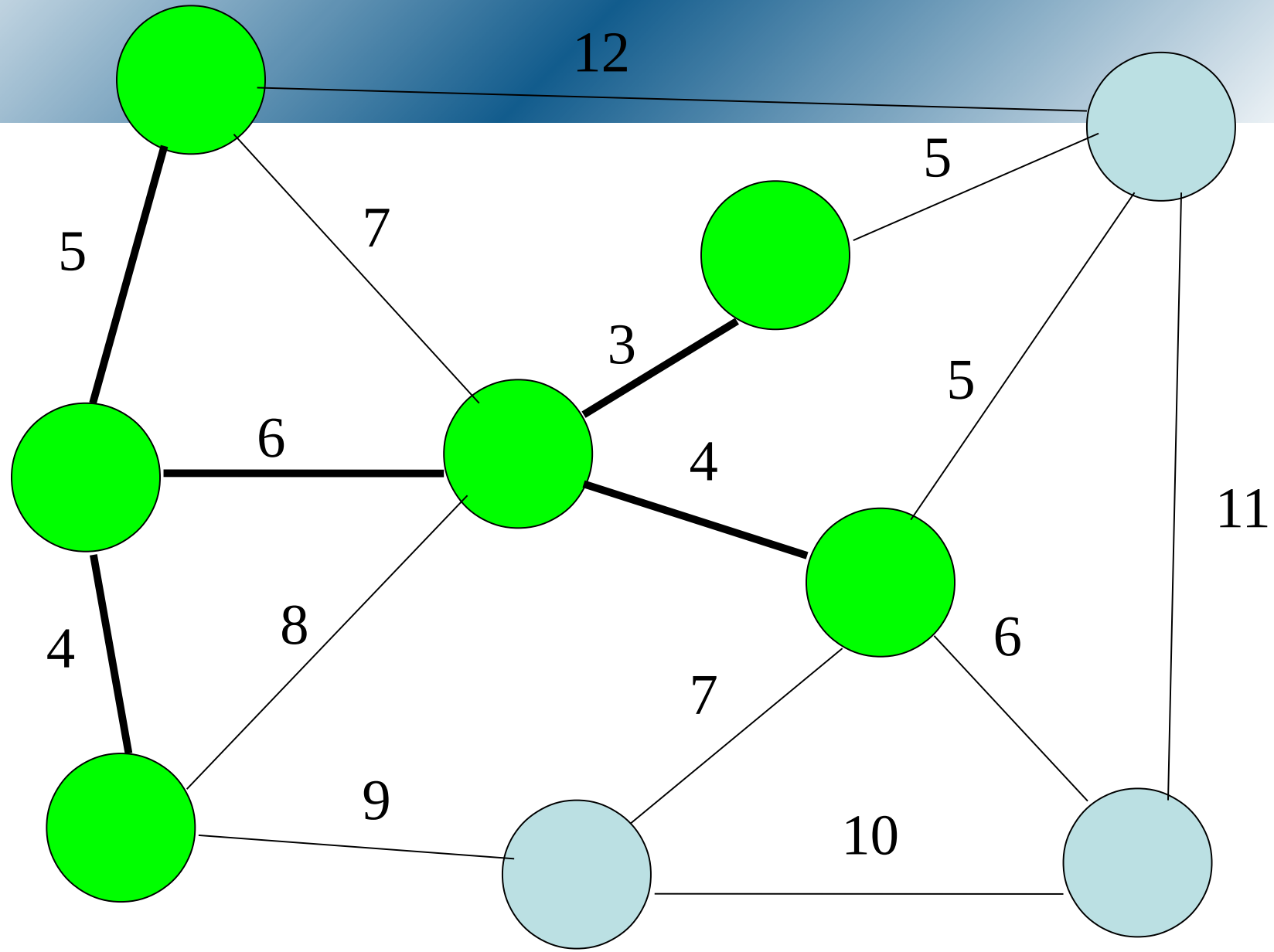

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

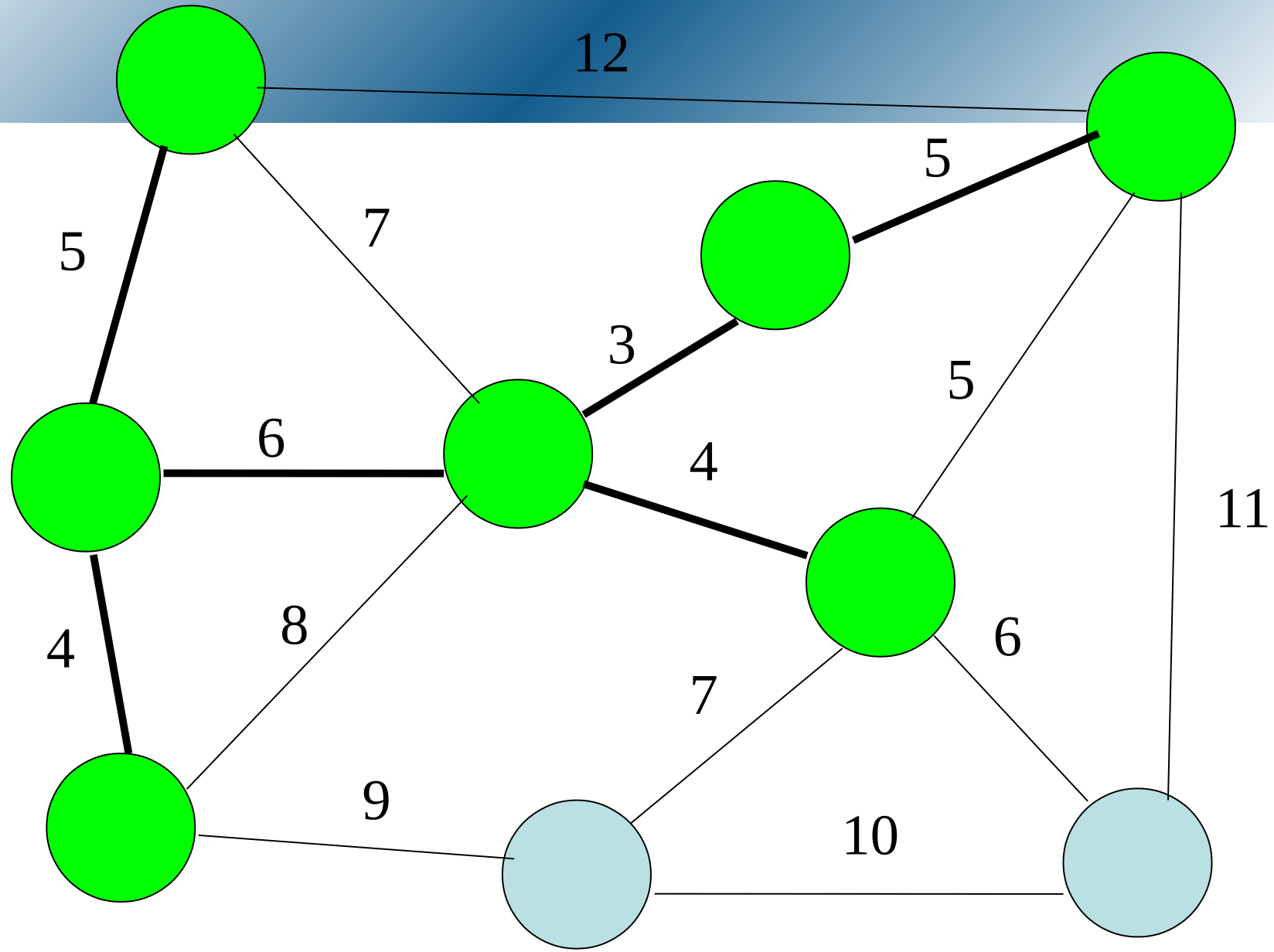

12
5
7
5
5
3
6
4
5
11
4
8
6
7
9
10

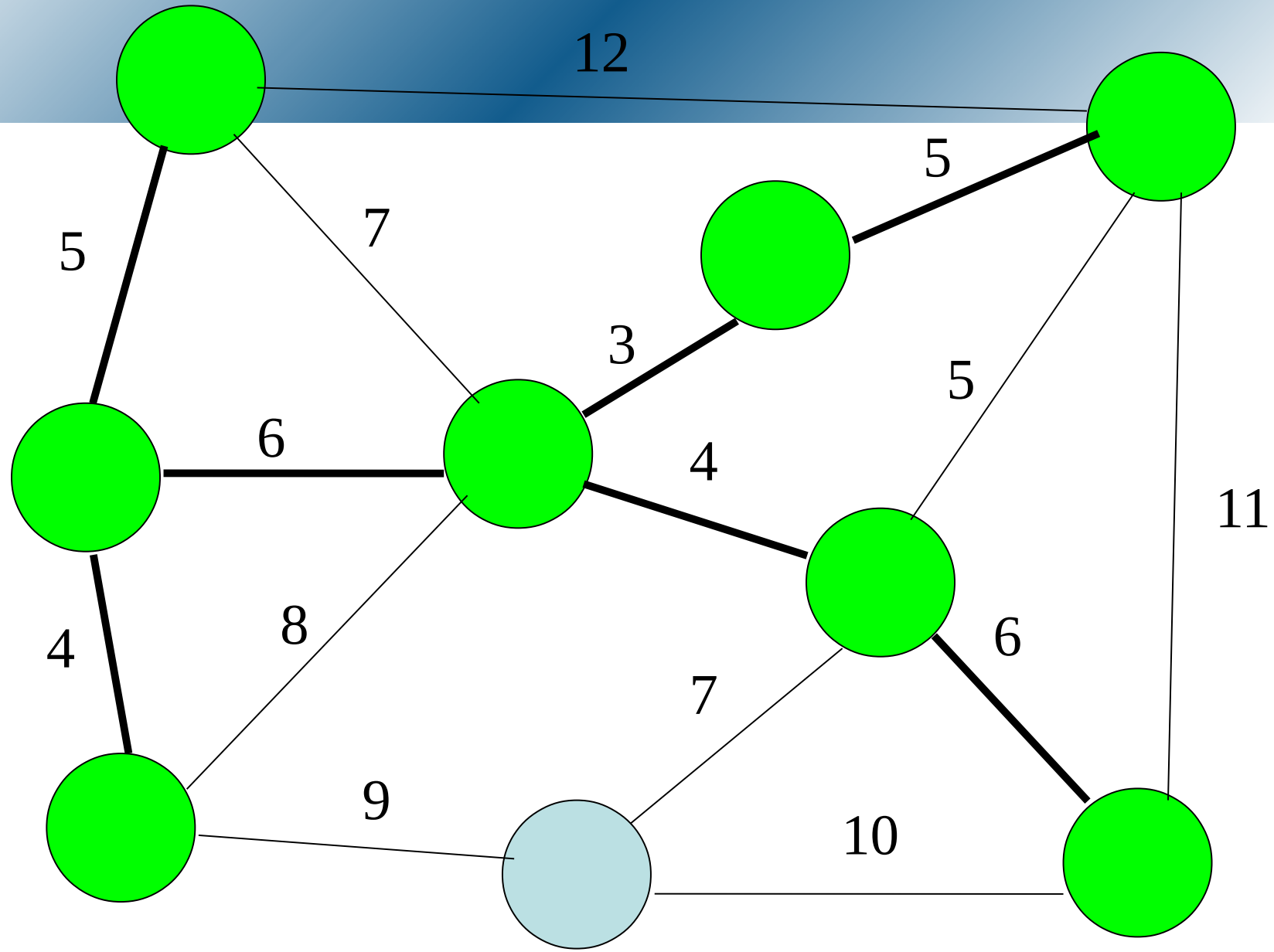

12
5
7
5
5
3
6
4
5
11
4
8
6
7
9
10

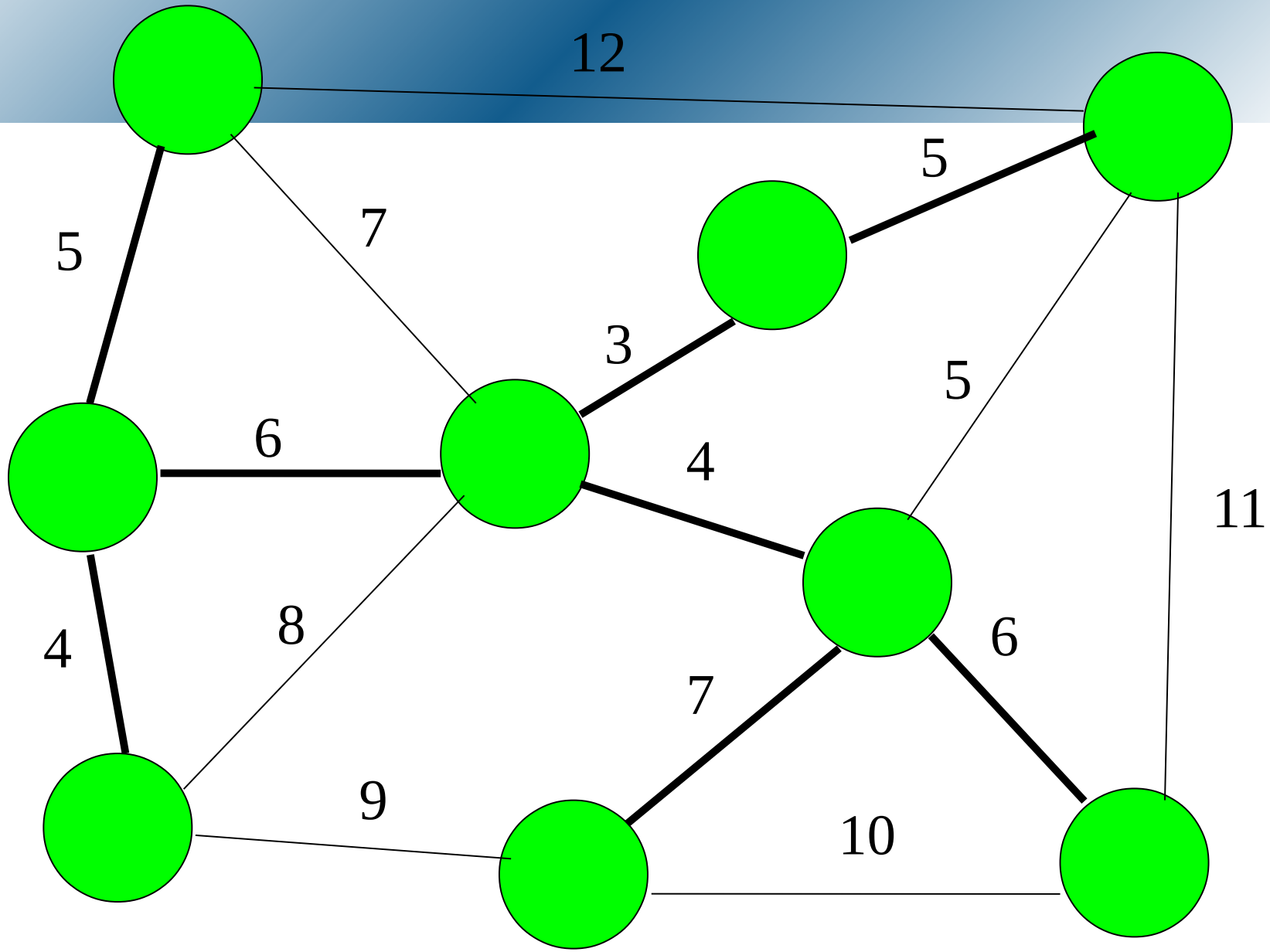

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10


12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

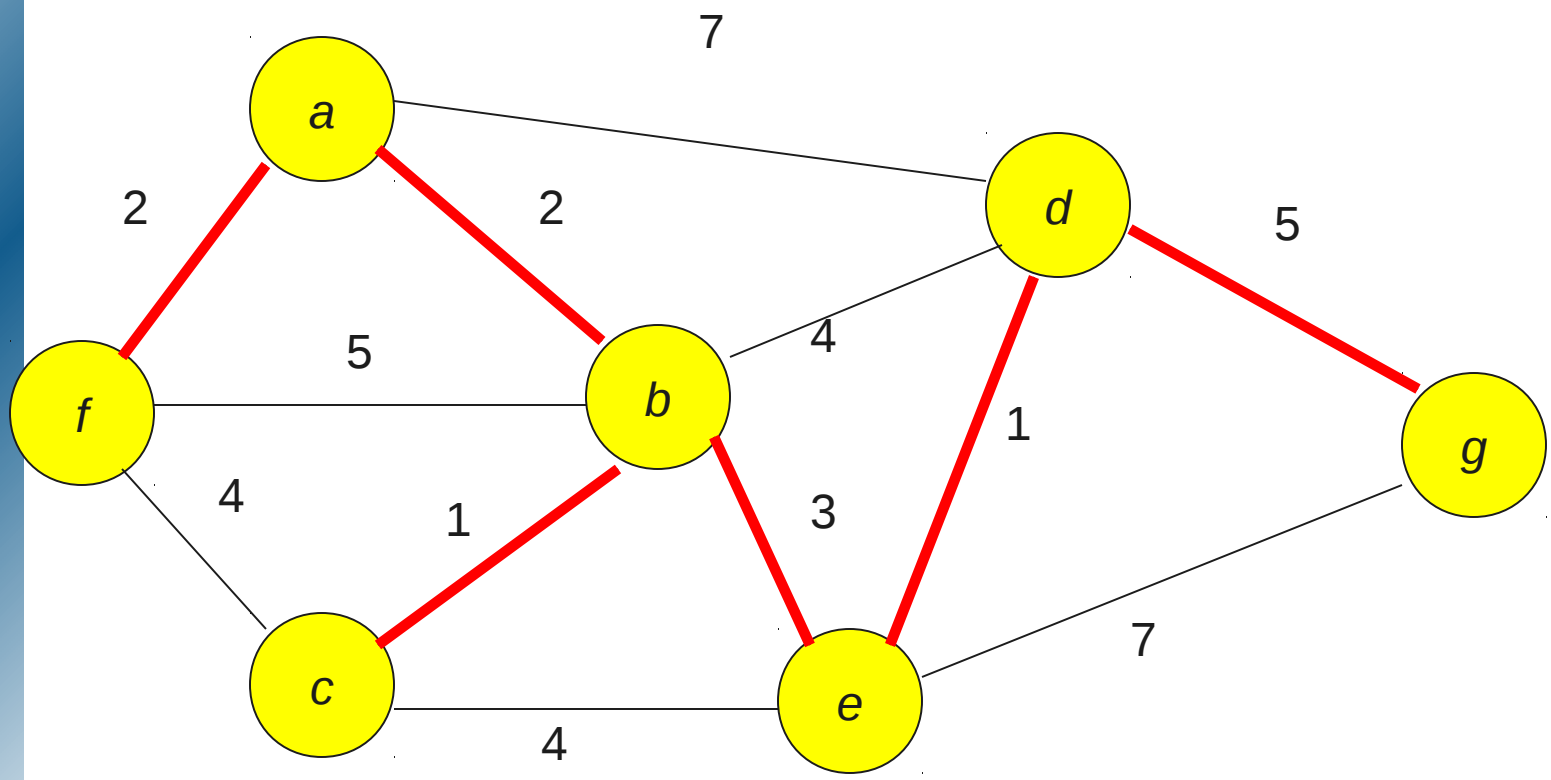

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10


12
5
7
5
3
5
6
4
11
4
8
6
7
9
10


12
5
7
5
3
5
6
4
11
4
8
6
7
9
10


12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

Otro ejemplo del Algoritmo de Prim



Implementación del algoritmo de Prim

- Para estudiar la eficiencia de Prim, es necesario elaborar un poco más su implementación, por lo que suponemos:
- L [i , j] ≥ 0 una matriz de distancias.
- MasProximo [x] es un vector que nos da el nodo de U que está más cercano al vértice x.
- DistMin [x] es un vector que nos da la distancia entre x y MasProximo [x].

Implementación del algoritmo de Prim

```
Funcion Prim (L[1...n, 1...n]: conjunto de aristas
{al comienzo solo el nodo 1 se encuentra en U}
    T = ∅ (contendrá las aristas del AGM)
    Para i = 2 hasta n hacer
        MasProximo [i] = 1; DistMin [i] = L[i, 1];
    Repetir n – 1 veces
        min = ∞;
        Para j = 2 hasta n hacer
            Si 0 ≤ DistMin [j] < min entonces min = DistMin [j];
                                              k = j;
        T = T + (MasProximo [k], k);
        DistMin [k] = -1; (estamos añadiendo k a U)
        para j = 2 hasta n hacer
            si L[j, k] < DistMin [j] entonces
                DistMin [j] = L[j, k];
                MasProximo [j] = k;
    Devolver T.
```

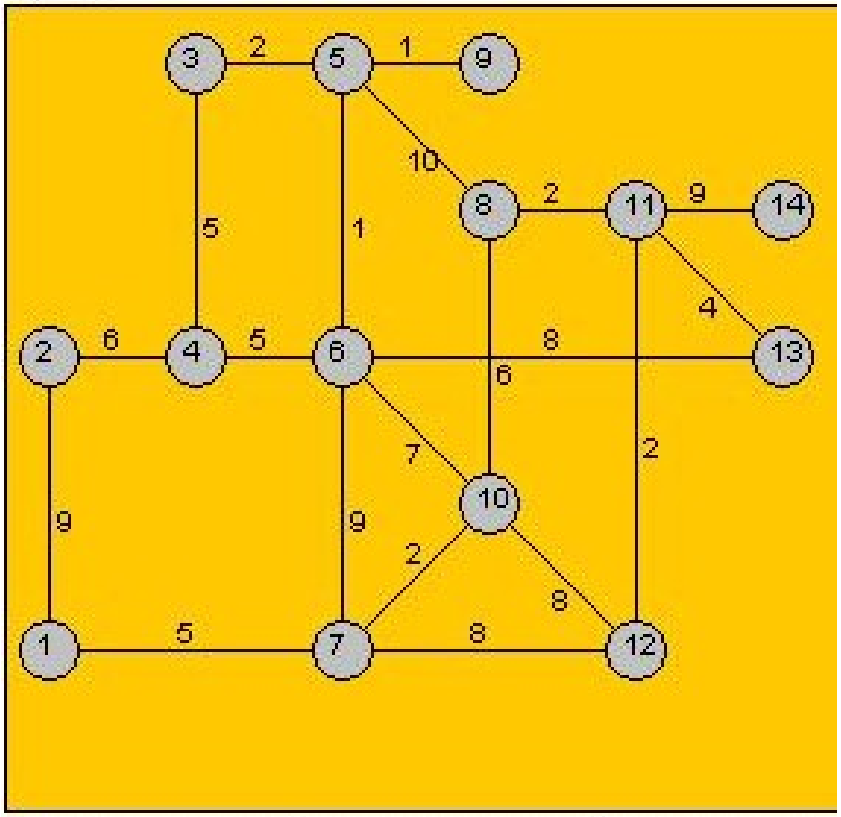
Análisis del algoritmo de Prim

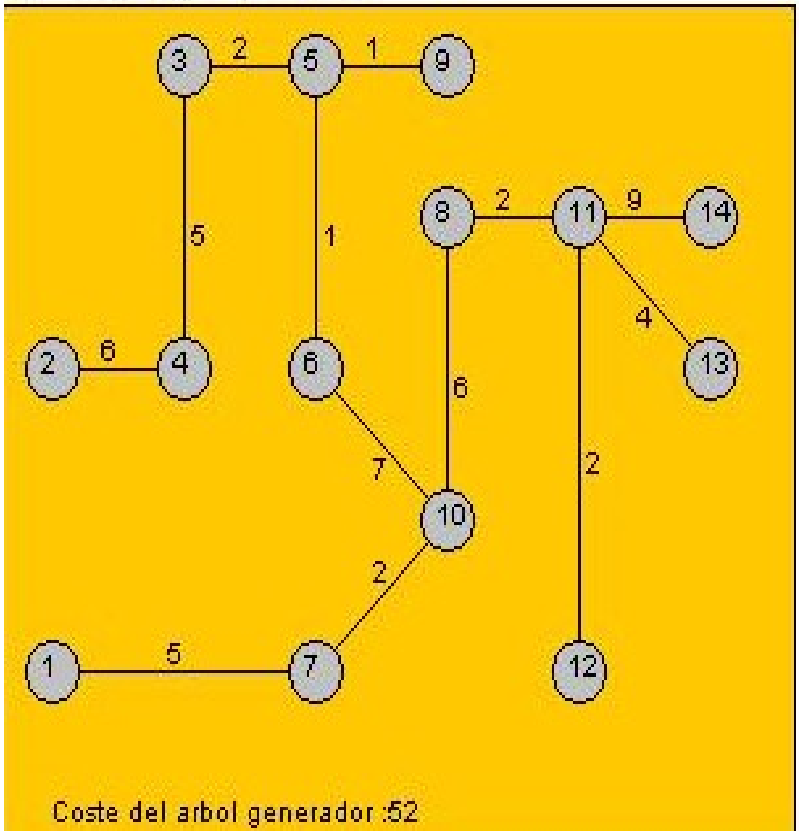
- El bucle principal del algoritmo se ejecuta n – 1 veces.
- En cada iteración, el bucle “para” anidado requiere un tiempo O(n). Por tanto, el algoritmo de Prim requiere un tiempo O (n^2).
- Como el Algoritmo de Kruskal era O(a log n), siendo a el número de aristas del grafo,
- ¿Qué algoritmo usar Prim o Kruskal?
- Sabemos que

$$n-1 \leq a < \frac{n(n-1)}{2}$$

- Luego en grafos poco densos, lo mejor seria emplear Kruskal, y si el grafo es muy denso, Prim.
- Se puede conseguir que Prim también sea O(a log n).

Ejemplo de Prim





Problema de Caminos Mínimos

Dado un grafo ponderado se quiere calcular el camino con menor peso entre un vértice *v* y otro *w*.

Problema de Caminos Minimos

Supongamos que tenemos un mapa de carreteras de España y estamos interesados en conocer el camino más corto que hay para ir desde Granada a Güevéjar.

Problema de Caminos Minimos

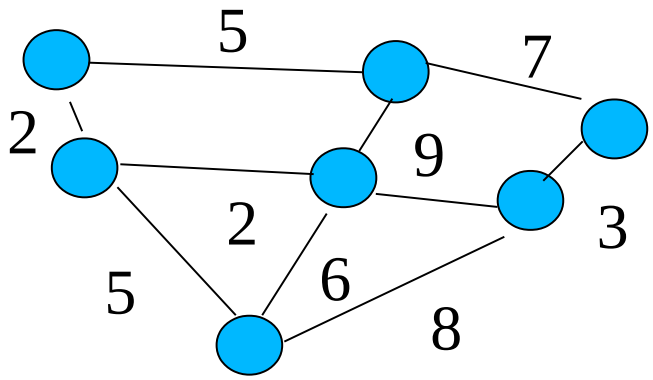
Modelamos el mapa de carreteras como un grafo: los vértices representan las intersecciones y los arcos representan las carreteras. El peso de un arco = distancia entre sus extremos.

Aplicaciones

- En comunicaciones y telecomunicaciones
- En inteligencia artificial (p.e. robótica)
- En investigación operativa (p.e distribución de instalaciones)
- Diseño VLSI

Problema de Caminos Minimos

- Dado un grafo G(V,A) y dado un vértice s



Encontrar el camino de costo mínimo para llegar desde s al resto de los vértices en el grafo.

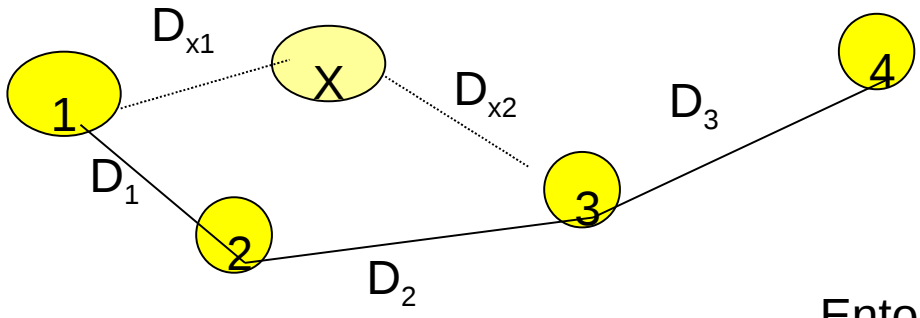
El costo del camino se define como la suma de los pesos de los arcos.

Se puede aplicar a grafos dirigidos y no dirigidos.

Propiedades de Caminos Mínimos

Asumimos que no hay arcos con costo negativo (es imprescindible para que funcione el algoritmo).

P1.- Tiene subestructuras optimales. Dado un camino óptimo, todos los subcaminos son óptimos. (xq?)



$M(1,4) = D_1+D_2+D_3$

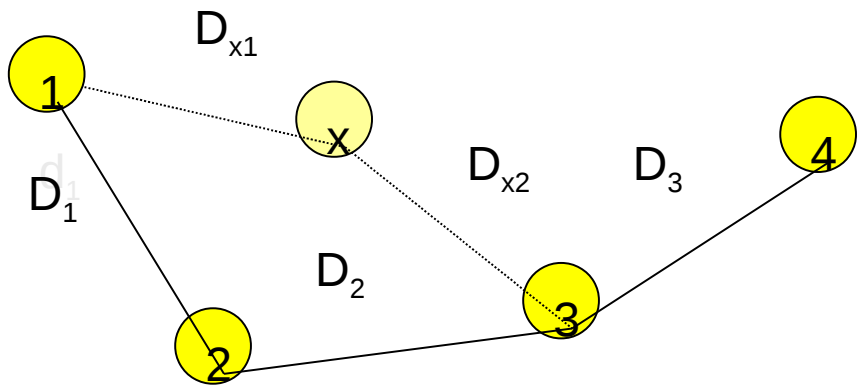
Si $M(1,3) = Dx_1+Dx_2$

Entonces $M(1,4) = Dx_1+Dx_2+D_3$

Propiedades de Caminos Mínimos

P2.- Si M(s,v) es la longitud del camino mínimo para ir de s a v, entonces se satisface que

$$M(s,v) <= M(s,u) + M(u,v)$$ (xq?)



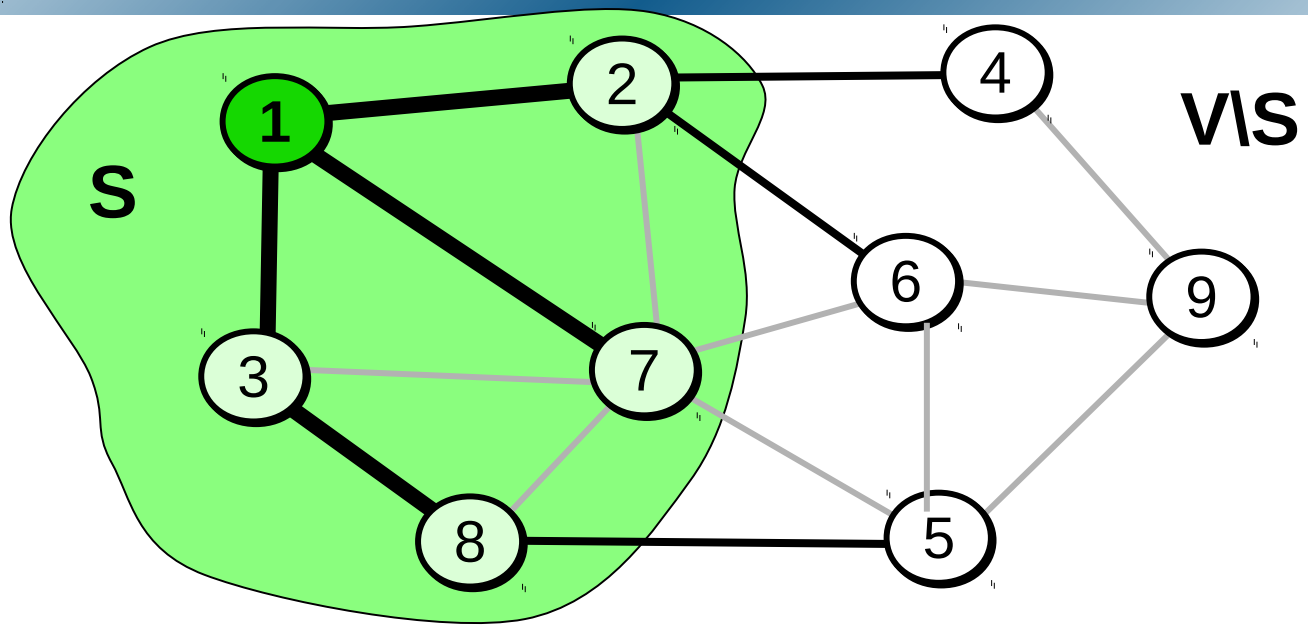
Si M(s,v)>M(s,u)+M(u,v), entonces encuentro un camino más corto de s a v pasando por u.

Algoritmo de Dijkstra

- Supongamos un grafo **G**, con pesos positivos y un nodo origen **s**.

- El algoritmo usa dos conjuntos de nodos:
  - **Escogidos: S**. Nodos para los cuales se conoce ya el camino mínimo desde el origen.
  - **Candidatos: V\S**. Nodos pendientes de calcular el camino mínimo, aunque conocemos los caminos mínimos desde el origen pasando por nodos de **S**.

Dijkstra



- **Camino especial:** camino desde el origen hasta un nodo, que pasa sólo por nodos escogidos, **S.**
- **Idea:** en cada paso, coger el nodo de **V\S** con menor distancia al origen. Añadirlo a **S**.

Algoritmo de Dijkstra

Candidatos: *Vértices*

Función Selección: Seleccionar el vértice u del conjunto de no seleccionados (V\S) que tenga menor distancia estimada al vértice origen (s).

Para mantener la información necesaria usamos:

- d[v] = longitud del camino (especial) de menor distancia del vértice s al vértice v pasando por vértices en S. Toma el valor infinito si no existe dicho camino
- p[v] = padre de v en el camino. Toma Null si no existe dicho padre.

Alg. Dijkstra

- Al incluirse un vértice x en S puede ocurrir que sea necesario actualizar d[.], esto es, recalcular los caminos mínimos de los demás candidatos, pudiendo pasar por el nodo escogido.
  - Xq?
  - Qué vértices son suceptibles de sufrir dicha actualización?
  - Cómo se ven afectados d[.] y p[.]?

Algoritmo de Dijkstra

```
C = {2, 3, ..., n } // el nodo 1 es el origen; implicitamente S={1}
PARA i = 2 HASTA n HACER d[i] = c[1, i]
                                 p[i] = 1
REPETIR n – 2 VECES
        v = algún elemento de C que minimice d[v]
        C = C – {v} // implicitamente se añade v a S
        PARA CADA w ∈ C HACER
                SI d[w] > d[v] + c[v, w] ENTONCES
                    d[w] = d[v] + c[v ,w]
                    p[w] = v
DEVOLVER d
```

Eficiencia $O(n^2) = O(V^2)$

Implementación: Alg. Dijkstra.

Para mejorar usamos una cola con prioridad de vértices con campos d y p

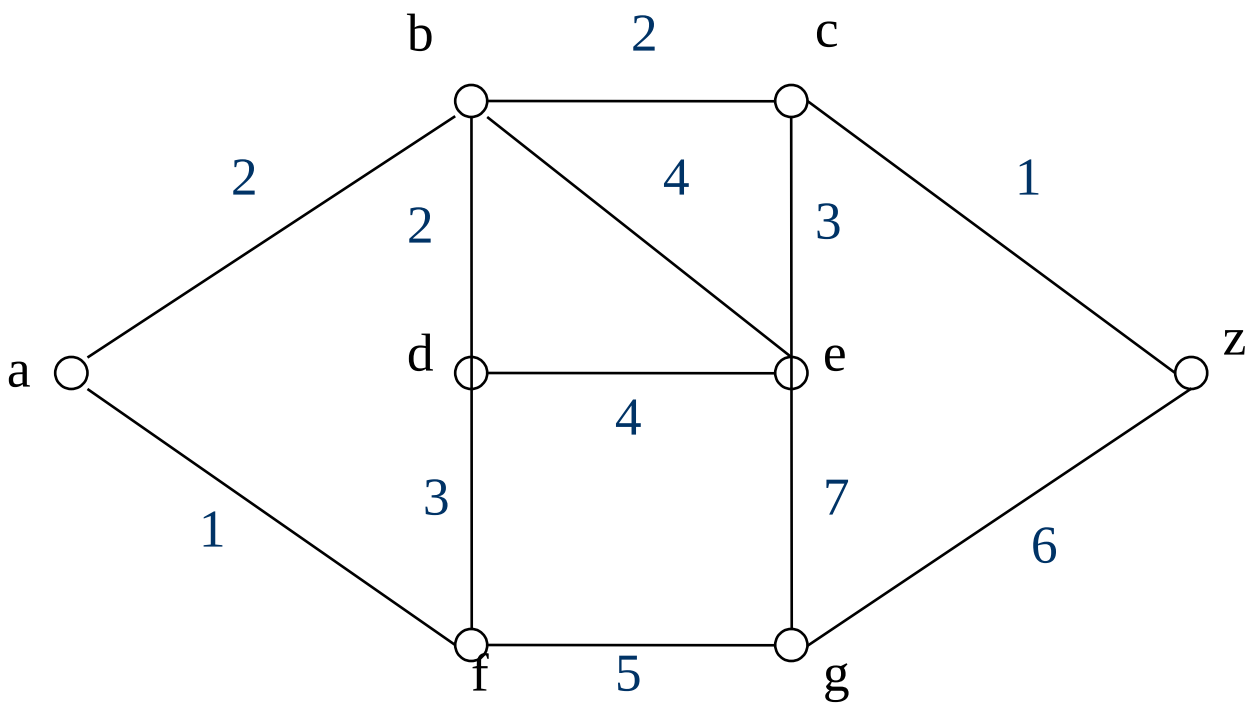
ALGORITMO DE DIJKSTRA

```
Para cada v en V
   d[v]= infinito
   p[v] =  null
d[s]=0
set<vertices> S; // Vértices seleccionados está vacio
priorityqueue Q;
Para cada  v en V
   Q.insert(v);
while (!Q.empty())
   v = Q.delete-min()
   S.insert(v)     // incluimos v en vértices seleccionados
   Para cada  w en Adj[v]
      if d[w] > d[v] + c(v,w)
         d[w] = d[v] + c(v,w)
         p[w] = v
```

Análisis de eficiencia

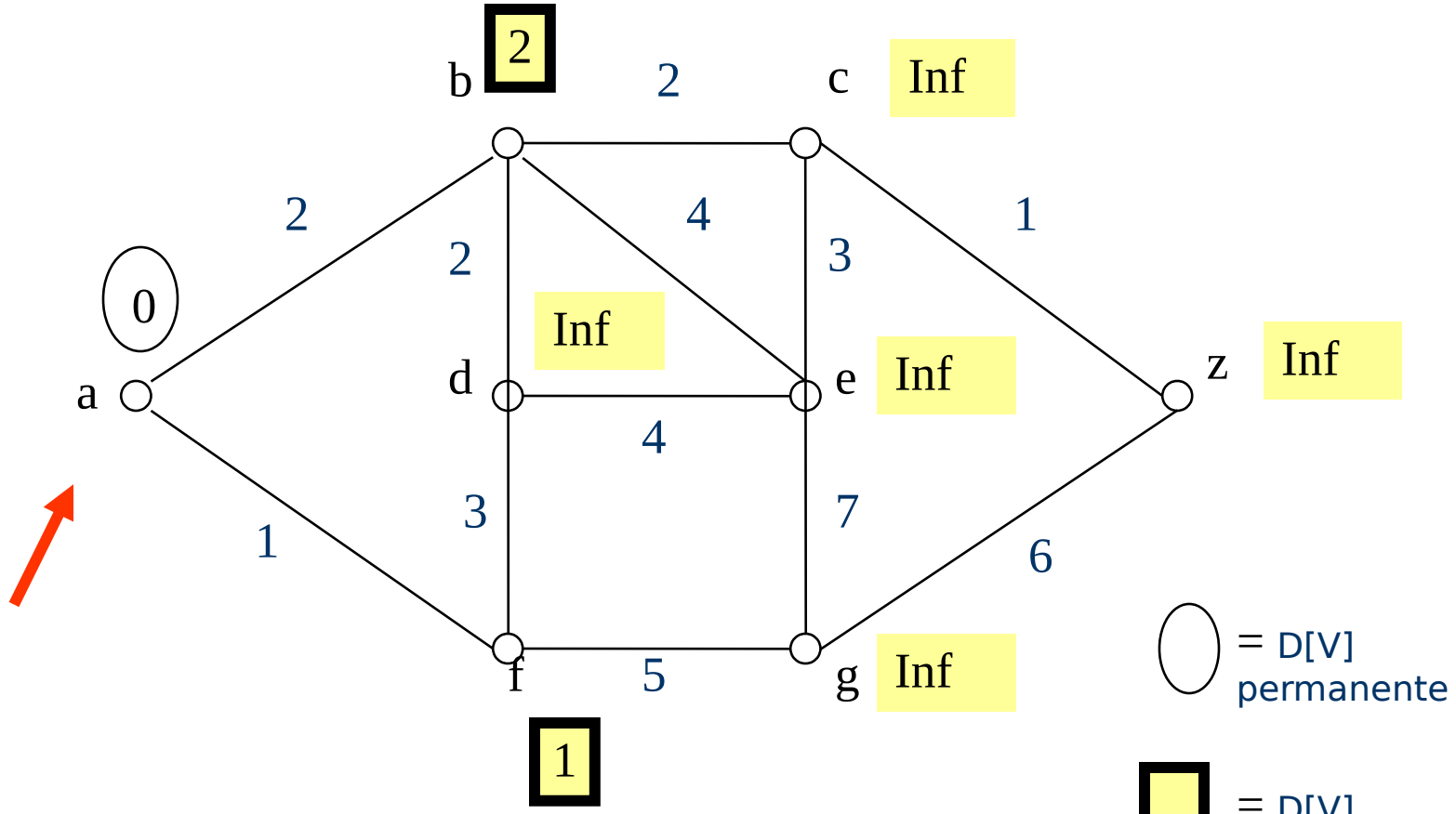
- La implementación sencilla es $O(V^2)$
- La implementación con cola con prioridad es $O(A \log V)$ (ver Brassard)
- La sencilla es preferible para grafos densos
- La de cola con prioridad es mejor para grafos dispersos

Ejemplo: Algoritmo Dijkstra



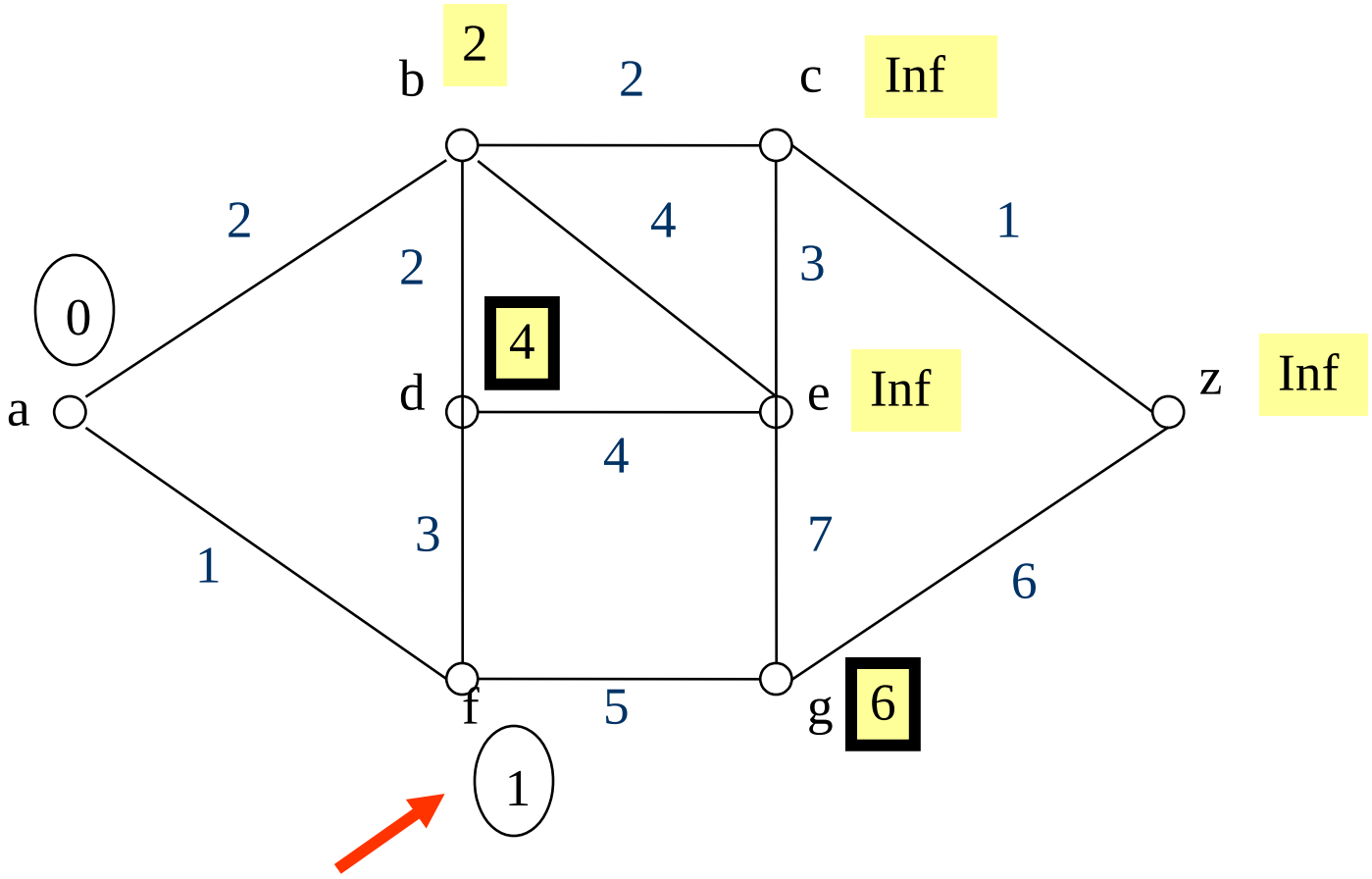
Ejemplo (Solo entre a y z)

Ejemplo: Algoritmo Dijkstra



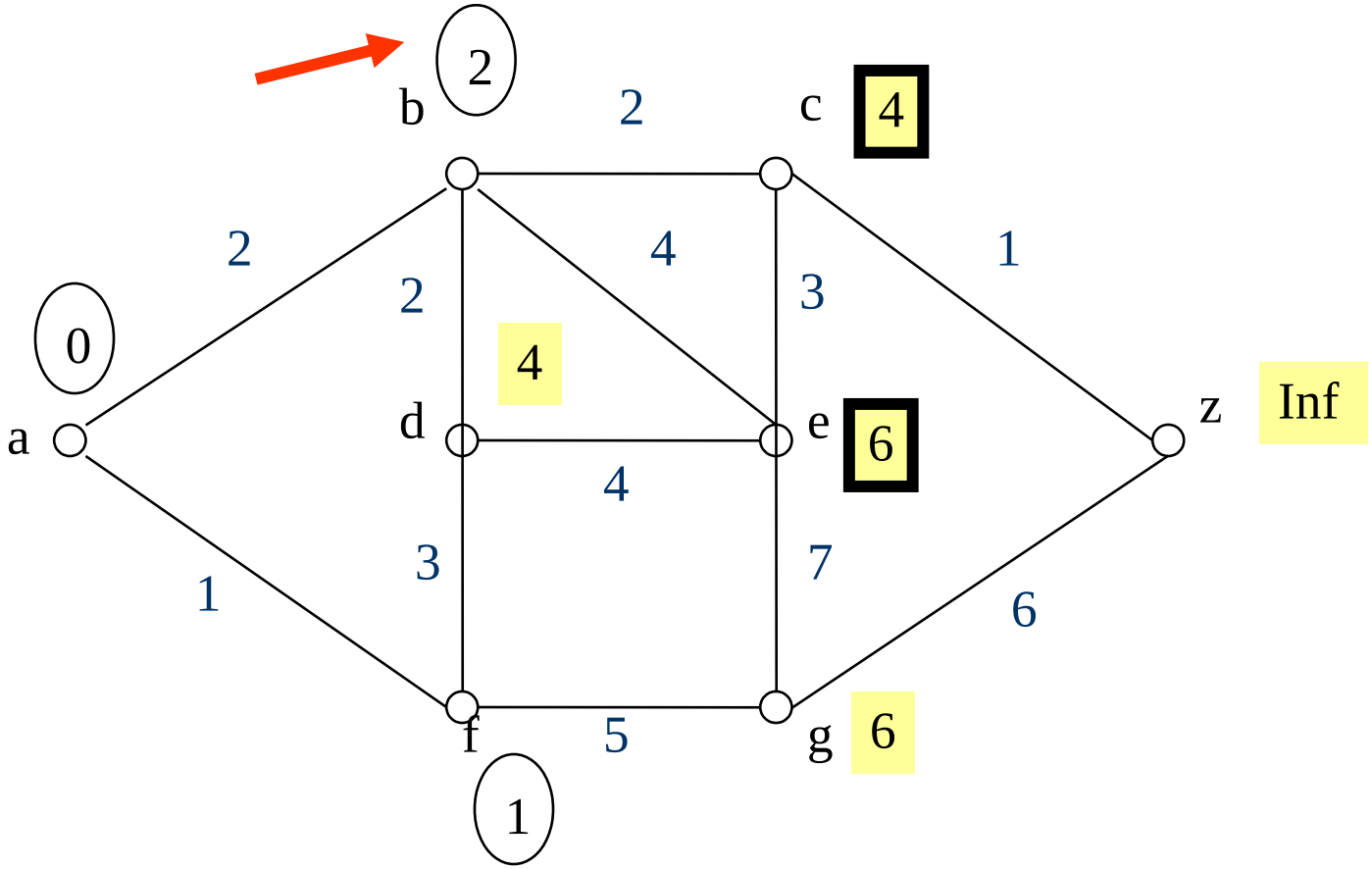
Inicialización

Ejemplo: Algoritmo Dijkstra



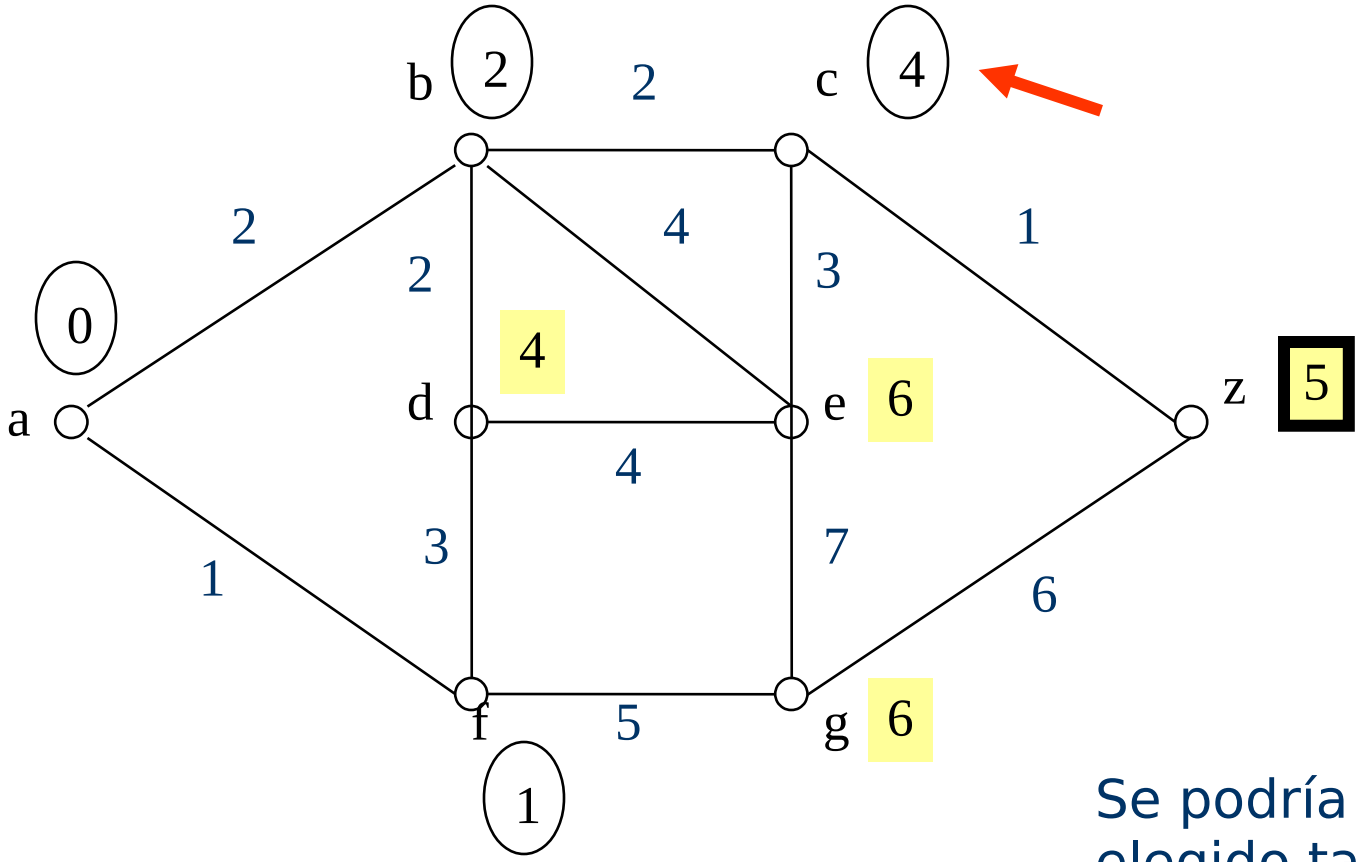
Primera Iteración

Ejemplo: Algoritmo Dijkstra



Segunda Iteración

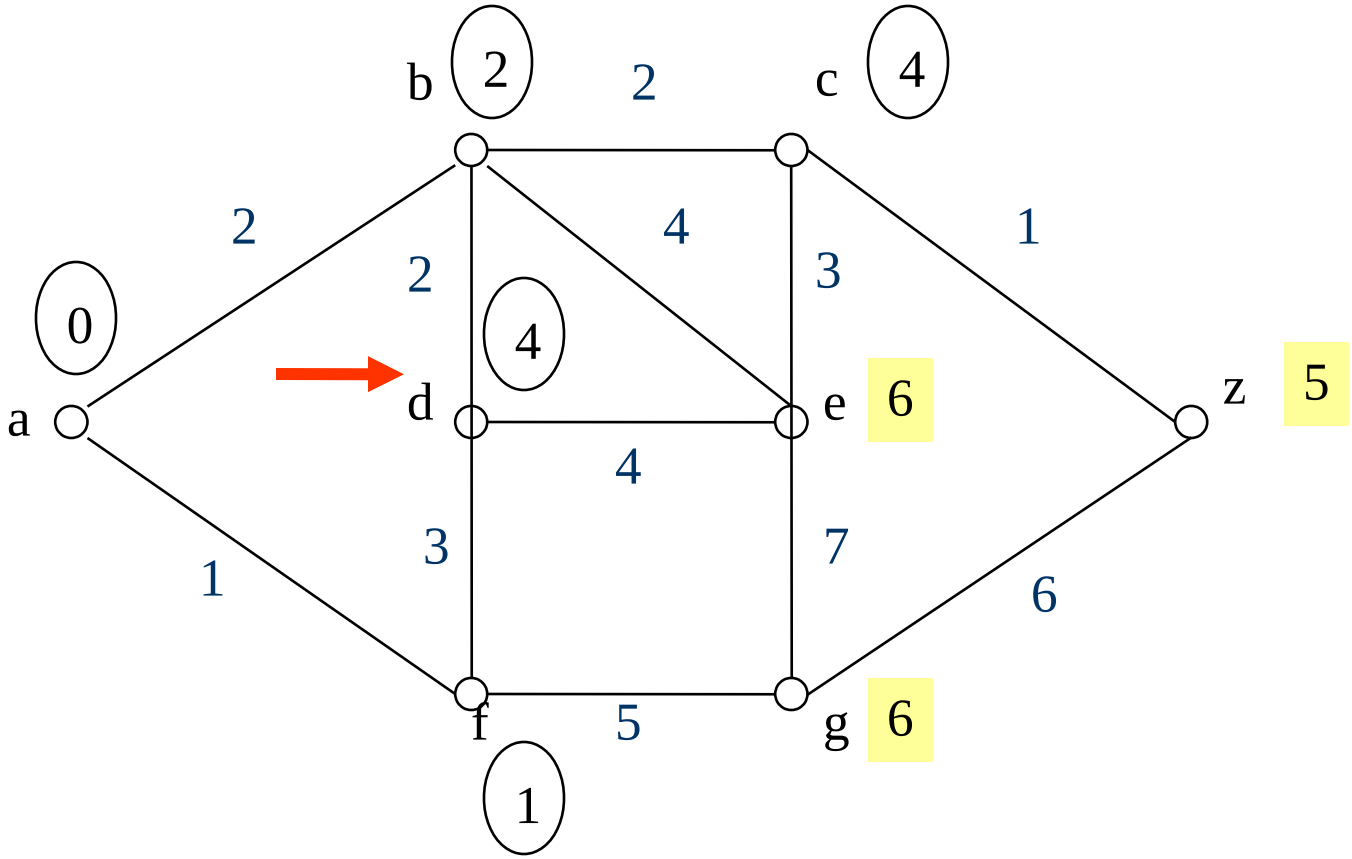
Ejemplo: Algoritmo Dijkstra



Se podría haber elegido también ‘d’

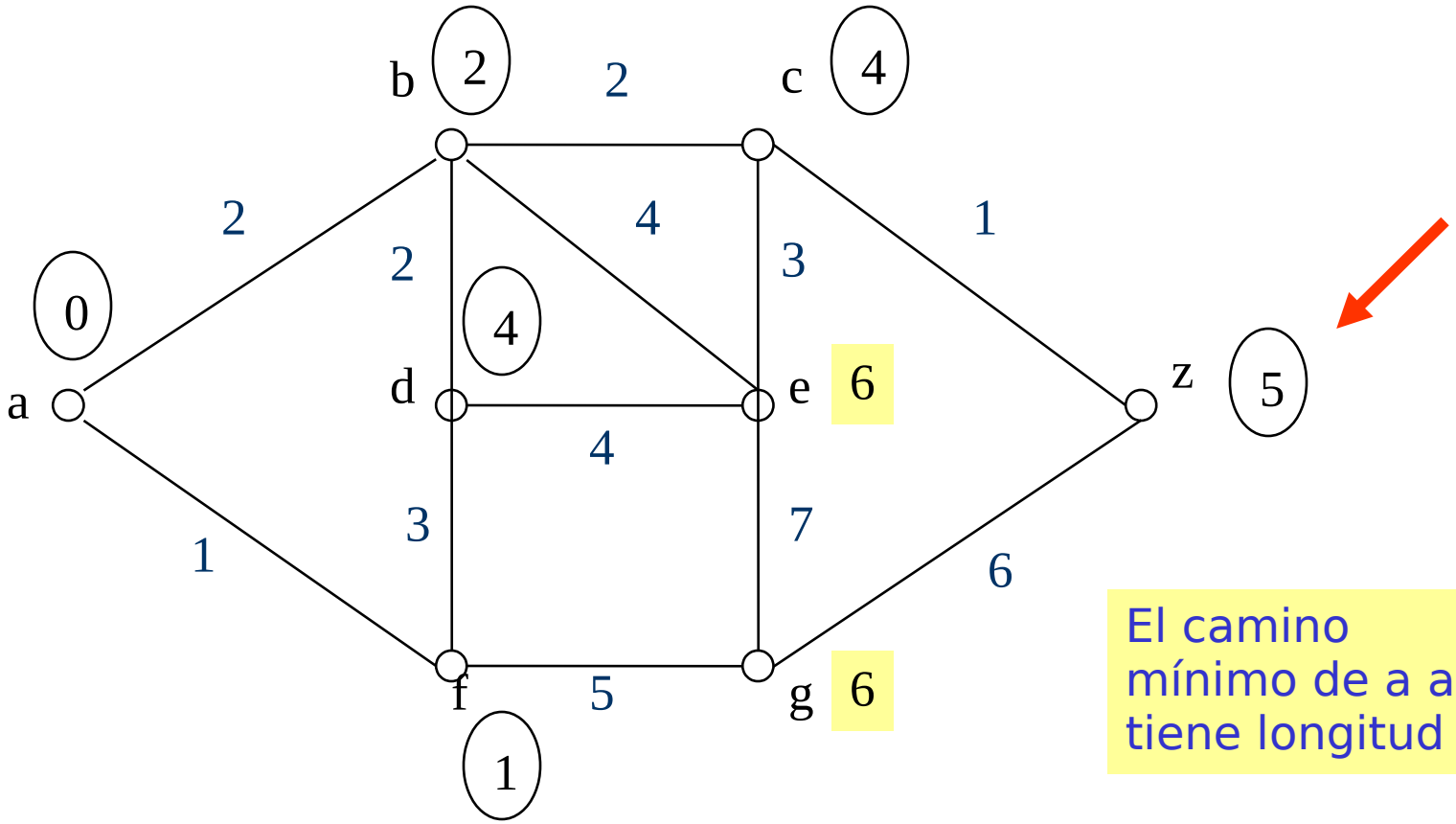
Tercera Iteración

Ejemplo: Algoritmo Dijkstra



Cuarta Iteración

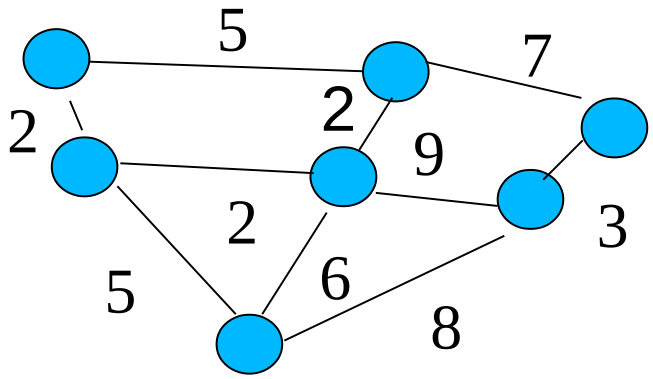
Ejemplo: Algoritmo Dijkstra



El camino mínimo de a a z tiene longitud 5

Quinta (y última) Iteración

Ejemplo: Algoritmo Dijkstra

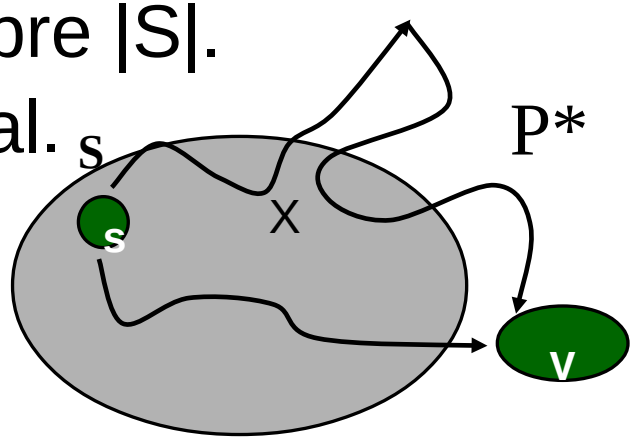


Queda como ejercicio.

Algoritmo Dijkstra: Demostración

Invariante. Para cada $v \in S$, $d(v) = M(s, v)$.

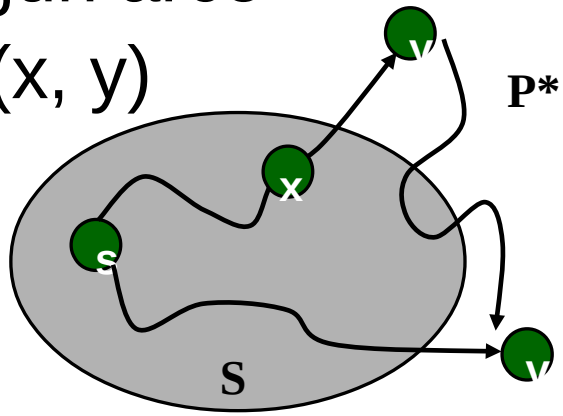
- Demostr. Por inducción sobre |S|.
- Caso base: |S| = 0 es trivial.



- Paso de inducción:
  - Supongamos que el algoritmo de Dijkstra añade el vértice v a S. d(v) representa la longitud de algún camino de s a v
  - Si d(v) no es la longitud del camino mínimo de s a v, entonces sea P* el camino mínimo de s a v
  - Sea x el último vértice en S en dicho camino (todos los anteriores son de S y el siguiente no)

Algoritmo Dijkstra: Demostración

En este caso P* debe utilizar algún arco que parta de x, por ejemplo (x, y)



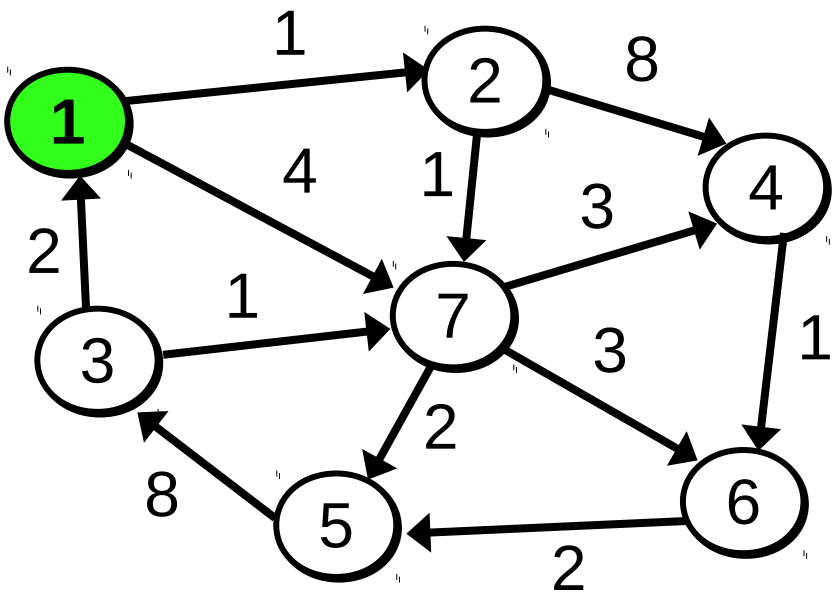
- Entonces tenemos que

d(v)>	M(s, v)	asumimos
=	M(s, x) + c(x, y) + M(y, v)	subestr. optimal
≥	M(s, x) + c(x, y)	arcos positivos
=	d(x) + c(x, y)	hipótesis inducción
≥	d(y)	el camino de s a y pasando por x es especial y d(y) es la longitud del menor de ellos

Por tanto el algoritmo de Dijkstra hubiese seleccionado y en lugar de v.

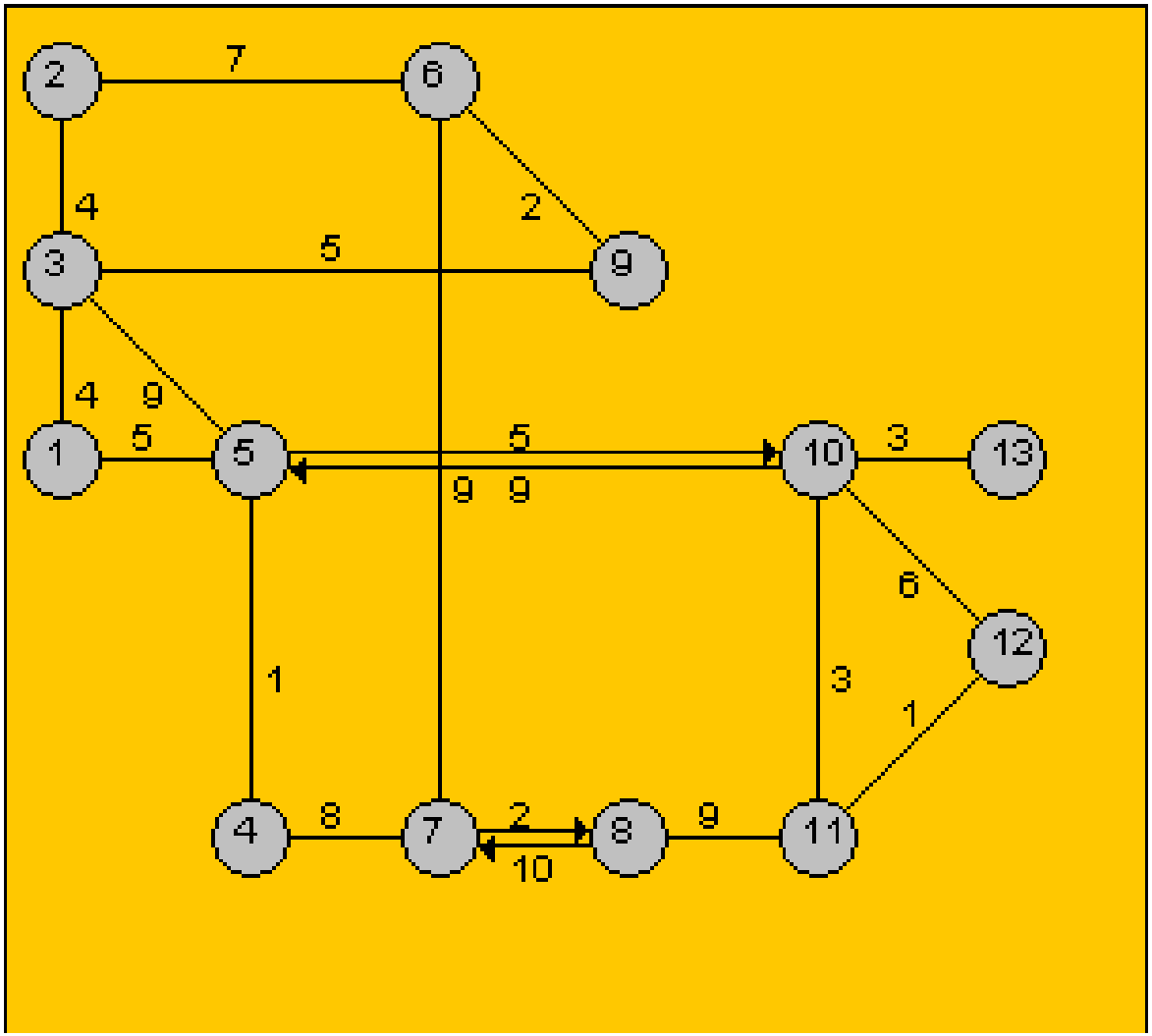
Ejemplo

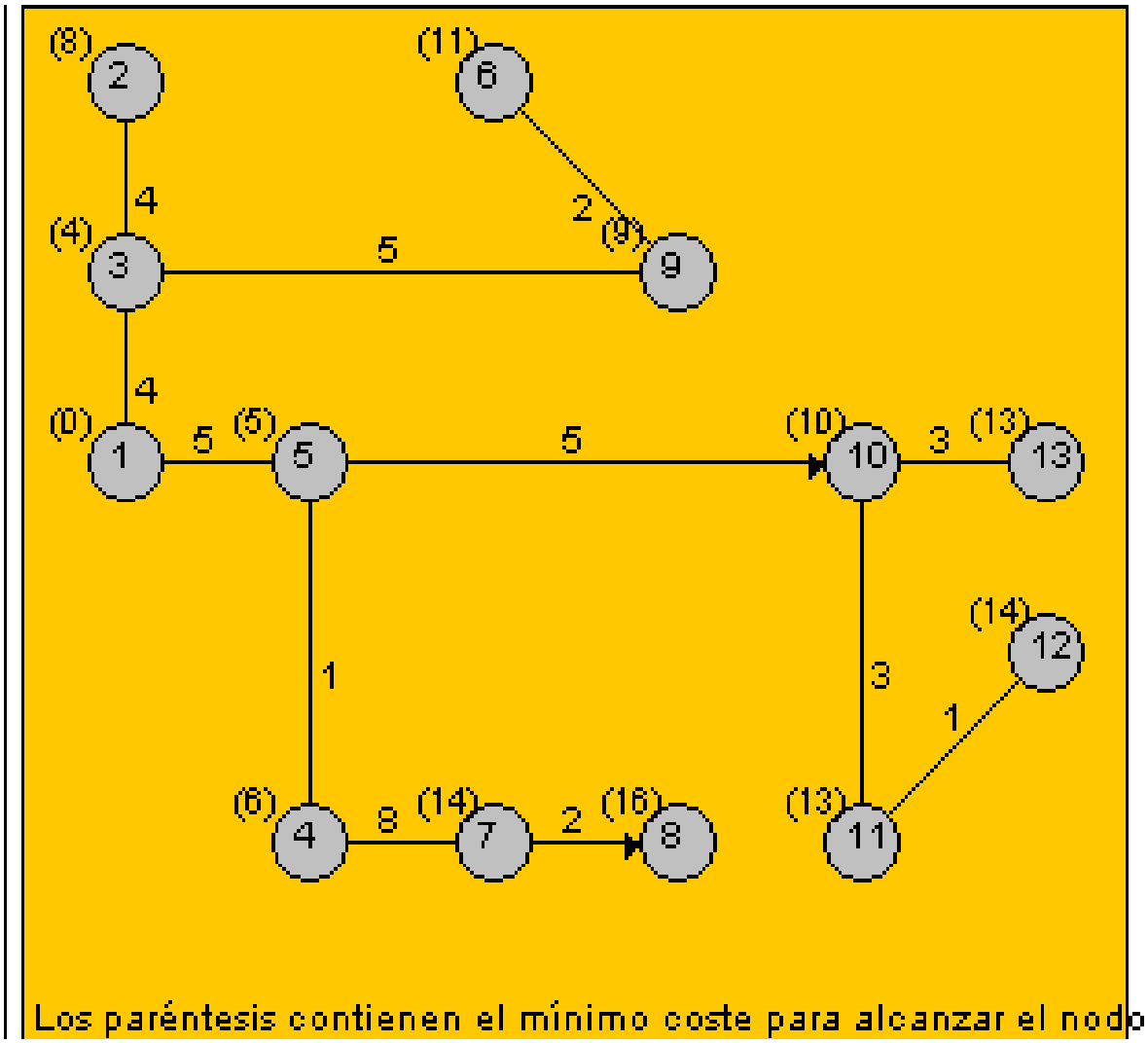
- **Ejemplo:** mostrar la ejecución del algoritmo de Dijkstra sobre el siguiente grafo.



Nodo	S	D	P
2	F	1	1
3	F	∞	1
4	F	∞	1
5	F	∞	1
6	F	∞	1
7	F	4	1

- A partir de las tablas, ¿cómo calcular cuál es el camino mínimo para un nodo **v**?


2
7
6
4
2
3
5
9
4
9
1
5
5
10
3
13
9
9
6
12
1
3
1
4
8
7
2
8
9
11
10


(8)
2
(11)
6
4
2
(4)
3
5
(9)
9
4
(0)
1
5
(5)
5
5
(10)
10
3
(13)
13
1
3
(14)
12
1
(6)
4
8
(14)
7
2
(16)
8
(13)
11
Los paréntesis contienen el mínimo coste para alcanzar el nodo