Algorítmica

Tema 1. La Eficiencia de los Algoritmos

Tema 2. Algoritmos “Divide y Vencerás”

Tema 3. Algoritmos Voraces (“Greedy”)

Tema 4. Algoritmos para la Exploración de Grafos (“Backtraking”, “Branch and Bound”)

Tema 5. Algoritmos basados en Programación Dinámica

Tema 6. Otras Técnicas Algorítmicas de Resolución de Problemas

Objetivos

- Comprender la filosofía de diseño de los algoritmos voraces
- Conocer las características de un problema resoluble mediante un algoritmo voraz
- Resolución de diversos problemas
- Heurísticas voraces: Soluciones aproximadas a problemas

Índice

- EL ENFOQUE GREEDY
- ALGORITMOS GREEDY EN GRAFOS
- HEURÍSTICA GREEDY

Índice

La filosofía Greedy (voraz)

- Buscan siempre la mejor opción en cada momento
- La decisión se toma en base a criterios locales

*El termino **greedy es sinónimo de voraz, ávido, glotón, ...***

La filosofía Greedy (voraz)

- Buscan siempre la mejor opción en cada momento
- La decisión se toma en base a criterios locales

Son algoritmos **no previsores** (ya se verá). Refranero:

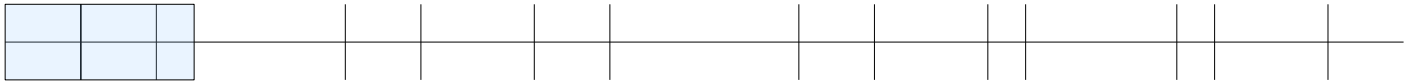
-Pan para hoy y hambre para mañana

-Dios proveerá

-Reventar antes que sobre

Selección de puntos de Parada

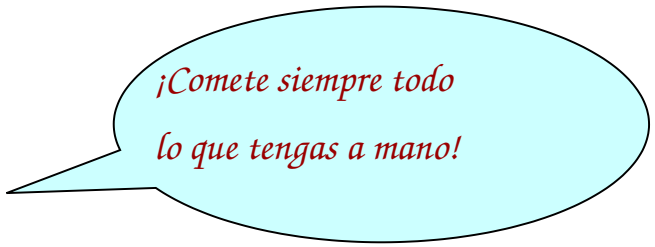
- Un camión va desde Granada a Moscú siguiendo una ruta predeterminada. Se asume que conocemos las gasolineras que se pueden encontrar en la ruta.
- La capacidad del depósito es = C.



- Problema: Minimizar el número de paradas que hace el conductor

¿Cómo se aplicaría la idea anterior para resolver este problema?

Algoritmos Greedy (voraz)



En este problema

Avanza lo más que puedas antes de rellenar el depósito.

Selección de puntos de parada

```
Ordenar las gasolineras en orden creciente
0 = g0 < g1 < g2 < ... < gn.
set<gasolineras> S;  // Seleccionamos gasolineras
x = g0
while (x != gn)
  gp = mayor gasolinera t.q. gp <= (x + C)
  if (gp = x) return "no hay solución"
  else {x = gp
        S.push_back(gp)
  }
return S, S.size();
```

Características generales de los algoritmos voraces

- Se utilizan generalmente para resolver problemas de optimización: máximo o mínimo.
- Un algoritmo “greedy” toma las decisiones en función de la información local que está disponible en cada momento (“miopes”).
- Una vez tomada la decisión no se la vuelve a replantear en el futuro.
- Suelen ser rápidos y fáciles de implementar.
- No garantizan alcanzar la solución óptima.

Elementos de un algoritmo voraz

Conjunto de Candidatos (C) : representa al conjunto de posibles decisiones que se pueden tomar en cada momento.

Conjunto de Seleccionados (S): representa al conjunto de decisiones tomadas hasta este momento.

Función Solución: determina si se ha alcanzado una solución (no necesariamente óptima).

Función de Factibilidad: determina si es posible completar el conjunto de candidatos seleccionados para alcanzar una solución al problema (no necesariamente óptima).

Función Selección: determina el candidato más prometedor del conjunto a seleccionar.

Función Objetivo: da el valor de la solución alcanzada.

Selección de puntos de parada

```
Ordenar las gasolineras en orden creciente
0 = g0 < g1 < g2 < ... < gn.
set<gasolineras> S;  // Seleccionamos gasolineras
x = g0
while (x != gn)
   gp = mayor gasolinera t.q. gp <= (x + C)
   if (gp = x) return "no hay solución"
   else { x = gp
          S.push_back(gp)
   }
return S, S.size()
```

Candidatos

Seleccionados

F. Solución

F. Selección

F. Factibilidad

F. Objetivo

Esquema de un algoritmo voraz

Un algoritmo Greedy procede siempre de la siguiente manera:

- Se parte de un conjunto de candidatos a solución vacío: S = ⊘
- De la lista de candidatos que hemos podido identificar, con la función de selección, se coge el mejor candidato posible
- Vemos si con ese elemento podríamos llegar a constituir una solución: Si se verifican las condiciones de factibilidad en S
- Si el candidato anterior no es válido, lo borramos de la lista de candidatos posibles, y nunca más es considerado
- Evaluamos la función solución. Si no hemos terminado seleccionamos con la función de selección otro candidato y repetimos el proceso anterior hasta alcanzar la solución.

Esquema de un algoritmo voraz

```
Voraz(C : conjunto de candidatos) : conjunto solución
    S = Ø
    mientras C ≠ Ø y no Solución(S) hacer
        x = Seleccion(C)
        C = C – {x}
        si factible(S U {x}) entonces
            S = S U {x}
        fin si
    fin mientras
    si Solución(S) entonces
        Devolver S
    en otro caso
        Devolver "No se encontró una solución"
    fin si
```

El enfoque Greedy suele proporcionar soluciones óptimas, pero no hay garantía de ello. Por tanto, siempre habrá que estudiar la **corrección del algoritmo** para verificar esas soluciones

Esquema de un algoritmo voraz

```
Ordenar las gasolineras en orden creciente
0 = g0 < g1 < g2 < ... < gn.
set<gasolineras> S;  // Seleccionamos gasolineras
x = g0
while (x != gn)
  gp = mayor gasolinera t.q. gp <= (x + C)
  if (gp = x) return "no hay solución"
  x = gp
  S.push_back(gp)
return S
```

Demostraciones

- ¿Por qué? Debemos garantizar que el algoritmo alcanza la solución óptima.
- ¿Cómo?, entre otras utilizaremos la reducción al absurdo (y la inducción)

Para probar que una proposición es verdadera, se supone que es falsa y se llega a un absurdo o a una contradicción, concluyéndose entonces que la proposición debe ser verdadera, pues no puede ser falsa.

Demostrar que una proposición es verdadera demostrando que no puede ser falsa.

Ejemplo Sudoku

1	2	3						
4	5	6						
7	X							
		9						

Proposición:

S[3,2]!=8

Demostración:

Asumir que S[3,2] = 8 y razonar

Hasta alcanzar un absurdo.

Si S[3,2] = 8, entonces, por

Regla del Cuadrante S[3,3]=9

Contradicción con que S[3,3]!=9 por regla Columna

Optimalidad Selec. paradas

- Demostración (red. absurdo):

Sean 0 = g0 < g1 < . . . < gp = L las gasol. seleccionadas por el alg. Greedy. Asumamos que L no es óptimo

Sobre todas las soluciones óptimas 0 = f0 < f1 < . . . < fq (q < p), llamemos r al máximo valor posible donde f0 = g0, f1= g1 , . . . , fr = gr. Sea Lop una de estas soluciones

Entonces, tenemos que

1) g(r+1) > f(r+1) (por como el algoritmo greedy selecciona las gasol. g).

2) 0 = g0 < . . < gr < g(r+1) < f(r+2) < ...< fq es otra solución al problema.

3) Además es óptima (tiene el mismo tamaño que Lop).

Luego Alcanzamos una contradicción: r NO es el máximo valor posible donde se alcanza la igualdad entre L y Lop

Problema de dar cambio

Se desea dar cambio usando el menor número posible de monedas de 1, 5, 10 y 25

¿Es greedy el problema?:

Candidatos: 1, 5, 10, 25 (con una moneda de cada tipo por lo menos).

Seleccionados: Podremos definirlo.

Solución: Lista de candidatos tal que la suma de los mismos coincida exactamente con el cambio pedido.

Criterio de factibilidad: Que no se supere el cambio.

Criterio de selección: Se escoge la moneda de mayor valor entre las disponibles.

Objetivo: El número de monedas ha de ser mínimo.

Problema de dar cambio

Si tenemos, por ejemplo, una moneda de 100 que queremos cambiar y la máquina dispone de 3 monedas de 25, 1 de 10, 2 de 5 y 25 de 1, entonces la primera solución que alcanza el algoritmo greedy directo es justamente la Solución óptima: 3 de 25, 1 de 10, 2 de 5 y 5 de 1.

Pero si tenemos 10 monedas de 1, 5 de 5, 3 de 10, 3 de 12 y 2 de 25, la solución del algoritmo para una moneda de 100 sería 2 de 25, 3 de 12, 1 de 10 y 4 de 1, en total diez monedas.

La solución no es óptima, ya que existe otra mejor que utiliza nueve monedas en vez de diez, que es 2 de 25, 3 de 10 y 4 de 5.

Para demostrar la no optimalidad basta un contraejemplo.

Problema Selección Programas

- Dado un conjunto T de n programas, cada uno con tamaño t1, ..., tn y un dispositivo de capacidad máxima C

- Objetivo:
  - Seleccionar el mayor número de programas que se pueden almacenar en C.

Problema: SelPro(T,C)

Problema Selección Programas: Solución Greedy

- *Candidatos a seleccionar:*
  - Programas
- *Candidatos seleccionados*
- *Función Solución*:
  - No entran más candidatos en el dispositivo
- *Función de Factibilidad*:
  - Es posible incluir el candidato actual en el dispositivo
- *Función Selección*: determina el mejor candidato del conjunto a seleccionar.
  - Seleccionar el candidato de menor tamaño
- *Función Objetivo*:
  - Número de programas en el dispositivo

Demostración técnica greedy alcanza el óptimo:

Tenemos que demostrar que

Optimo Local ⇒ Solución global optimal

- Paso 1.
  - Demostrar que la primera decisión es correcta.
- Paso 2.
  - Demostrar que existen subestructuras optimales.
    - Es decir, cuando se ha tomado la decisión #1, el problema se reduce a encontrar una solución optimal para el sub-problema que tiene como candidatos los compatibles con #1

Dem.: Primera decisión es correcta

- Suponemos los programas ordenados en tamaño

$$t1 <= t2 <= t3 <= \ldots <= tn$$

- Teorema: Si T es un conjunto de programas (ordenado), entonces $\exists$ una solución optimal $A \subseteq T$ tal que $\{1\} \in A$
  - Idea: Usar reducción al absurdo
  - Si ninguna solución optimal contiene {1}, elegimos una, B, y siempre podremos reemplazar la primera actividad en B por {1} (xq?). Obteniendo el mismo número de programas, y por tanto una sol. optimal.

Dem.: subestructuras optimales

- Teorema: Sea *A* una solución óptima al problema *SelPro(T,C) y sea t1 el primer programa en A* . Entonces *A* - {*t1*} es solución óptima para SelPro(T*,C-t1), con T* = {t*i en T*: i>1}
  - Demostración: (Red. Absurdo)
  - Partimos de que *A* – {*t1*} NO ES solución óptima para SelPro(T*,C-t1) . Esto es, podemos encontrar una solución optimal *B* al problema SelPro(T*,C-t1) donde se verifica que |*B*| > |*A* - {*t1*}|,
    - Entonces:
    - ?? *B* U {*t1*} es solución a SelPro(T,C)
    - Pero |*B* U {*t1*}| > |A| !!! Contradicción con el hecho de que A es solución optimal al problema SelPro(T,C)

Problema Selección de Actividades

Tenemos la entrada de una Exposición que organiza un conjunto de actividades

- Para cada actividad conocemos su horario de comienzo y fin.
- Con la entrada podemos asistir a todas las actividades.
- Hay actividades que se solapan en el tiempo.

Objetivo: Asistir al mayor número de actividades posible =>
Problema de selección de actividades

Otra alternativa: Minimizar el tiempo que estamos ociosos.

Problema Selección de Actividades

Dado un conjunto *S* de *n* actividades

si = tiempo de comienzo de la actividad *i*

fi = tiempo de finalización de la actividad *i*

– Encontrar el subconjunto de actividades compatibles A de tamaño máximo

Problema Selección de Actividades

Fijemos los elementos de la técnica

- *Candidatos a seleccionar: Conj. Actividades, S*
- *Candidatos seleccionados: Conjunto A, inic. A={Ø }*
- *Función Solución: S={Ø }.*
- *Función Selección: determina el mejor candidato, x*
  - *– Menor duración.*
  - *– Menor solapamiento.*
  - *– Comienza antes.*
  - *– Termina antes.*
- *Función de Factibilidad: x es factible si es compatible con las actividades en A (no hay actividades solapadas).*
- *Función Objetivo: Tamaño de A.*

Selección de actividades: contraejemplos

- Comienza antes

- Menor duración

- Menor solapamiento

Problema Selección de Actividades

Algoritmo Greedy

- *SelecciónActividades(Activ S,A)*
  - *Ordenar S en orden creciente de tiempo de finalización*
  - *Seleccionar la primera actividad.*
  - *Repetir*

    Seleccionar la siguiente actividad en el orden que comience después de que la actividad previa termine.
  - *Hasta que S esté vacio.*

Selección de actividades

```
SelecciónActividadesGreedy(S, A){
qsort(S,n); // según tiempo de finalización
A[0]= S[0] // Seleccionar primera actividad
i=1; prev = 0;

while (!S.empty()) { // es solucion(S)
    x = S[i] // seleccionar;
    if (x.inicio > A[prev].fin) // factible x
        A[prev++] = x; // insertamos en solucion
    else i++; // rechazamos
    }
 }
```

Problema Selección de Actividades

- ***¿Optimalidad?***

T.H. CORMEN, C.E. LEISERSON, R.L. RIVEST.
Introduction to Algorithms. The MIT Press (1992)

Se puede probar de forma similar a la del problema de las gasolineras.

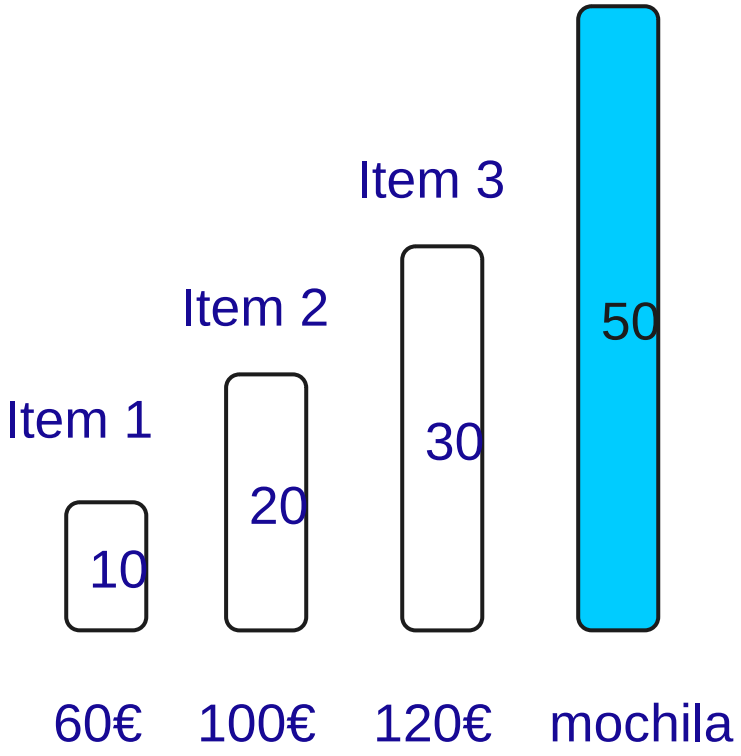
Problema de la mochila fraccional

- Consiste en llenar una mochila:
  - Puede llevar como máximo un peso *P*
  - *Hay n distintos posibles* objetos i fraccionables cuyos pesos son p_i y el beneficio por cada uno de esos objetos es de b_i. Si incluimos una fracción x_i ($0<=x_i<=1$) del objeto i se genera un beneficio de $x_i b_i$ y un peso de $x_i p_i$.

- El *objetivo* es maximizar el beneficio de los objetos transportados.

$$\max \sum_{i=1}^{n} x_i b_i$$
$$s.a. \sum_{i=1}^{n} x_i p_i \leq P$$

Ejemplo



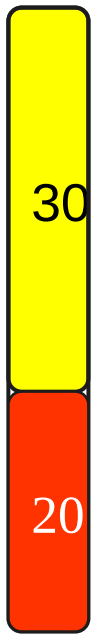
Es un claro problema de tipo greedy

Sus aplicaciones son inumerables

Es un banco de pruebas algorítmico

La técnica greedy produce soluciones optimales para este tipo de problemas cuando se permite fraccionar los objetos

Ejemplo: Mochila 0/1

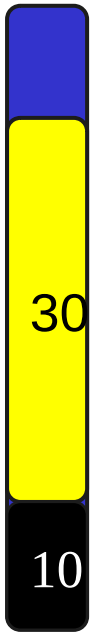


120€

100€

100€

60 €



120€

60 €

Total = 220 € = 160 € = 180 €

¿Cómo seleccionamos los items?

Ejemplo: Problema de la mochila fraccional

- Supongamos 5 objetos de peso y precios dados por la tabla, la capacidad de la mochila es 100.

Precio (euros)	20	30	65	40	60
Peso (kilos)	10	20	30	40	50

- **Método 1 elegir primero el menos pesado**
  - Peso total = 1*10 + 1*20 + 1*30 + 1*40 = 100
  - Beneficio total = 20 + 30 + 65 + 40 = 155
- **Método 2 elegir primero el de más beneficio**
  - Peso Total = 1*30 + 1*50 + 0.5*40 = 100
  - Beneficio Total = 65 + 60 + 0.5*40 = 145

Ejemplo: Problema de la mochila fraccional

- Metodo 3 elegir primero el que tenga mayor valor por unidad de peso (razón beneficio / peso)

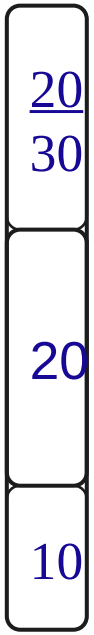
Precio (euros)	20	30	65	40	60
Peso (Kilos)	10	20	30	40	50
Precio/Peso	2	1,5	2,1	1	1,2

  - Peso Total = 1*30 + 1*10 + 1*20 + 0.8*50 = 100
  - Benef. Total = 65 + 20 + 30 + 0.8*60 = 163

Mochila fraccional

Tomando los items en orden de mayor valor por unidad d e peso, se obtiene una solución optimal

$\{60/10, 100/20, 120/30\}$



80 €

100€

60 €

Total = 240 €

Solución Greedy

- Definimos la densidad del objeto A_i por b_i/p_i.
- Se usan objetos de tan alta densidad como sea posible, es decir, los seleccionaremos en orden decreciente de densidad.
- Si es posible se coge todo lo que se pueda de A_i, pero si no se rellena el espacio disponible de la mochila con una fracción del objeto en curso, hasta completar la capacidad, y se desprecia el resto.
- Se ordenan los objetos por densidad no creciente, i.e.:

$$b_i/p_i \geq b_{i+1}/p_{i+1} \text{ para } 1 \leq i < n.$$

Pseudocódigo mochila

Procedimiento MOCHILA_GREEDY(b,p,M,X,n)

//b(1:n) y p(1:n) contienen los beneficios y pesos respectivos de los n objeto s ordenados como b(i)/p(i) ≥ b(i+1)/p(i+1). M es la capacidad de la mochila y X(1:n) es el vector solución

X = 0; //inicializa la solucion en cero

cr = M; // cr = capacidad restante de la mochila

i=1;

While (p(i) <= cr and i <= n)

    X(i) = 1;

    cr = cr – p(i);

    i = i+1;

if i<= n Entonces X(i) = cr/p(i)

Demostración de optimalidad

Vamos a demostrar que el algoritmo siempre encuentra la solución optimal del problema

Sea $b_1/p_1 \geq b_2/p_2 \geq \ldots \geq b_n/p_n$

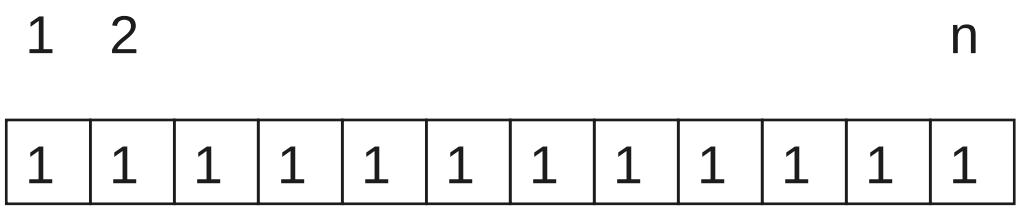
Sea $X = (x_1, x_2, \ldots, x_n)$ la solución generada por MOCHILA_GREEDY, para una capacidad M

Sea $Y = (y_1, y_2, \ldots, y_n)$ una solución factible cualquiera

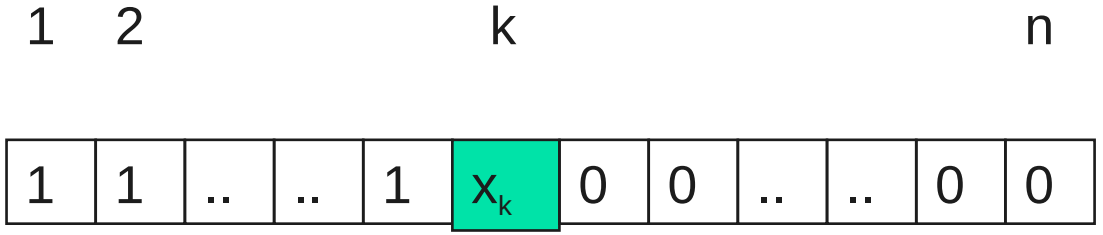
Queremos demostrar que

$$\sum_{i=1}^{n} (x_i - y_i) b_i \geq 0$$

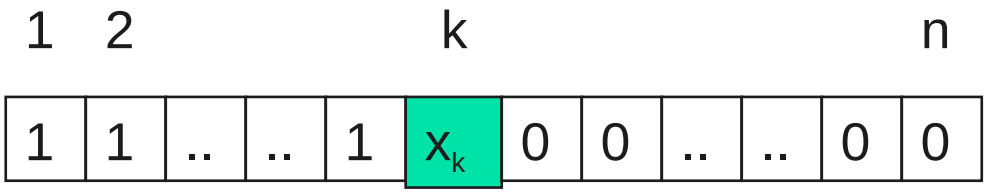
Demostración de optimalidad



Si todos los x_i son 1, la solución es claramente optimal. En otro caso, sea k el menor número tal que $x_k < 1$.



Demostración de optimalidad



$$\sum_{i=1}^{n}(x_i-y_i)b_i = \sum_{i=1}^{k-1}(x_i-y_i)\,p_i\frac{b_i}{p_i} + (x_k-y_k)\,p_k\frac{b_k}{p_k}$$

$$+\sum_{i=k+1}^{n}(x_i-y_i)\,p_i\frac{b_i}{p_i}$$

Demostración de optimalidad

- $x_i - y_i >= 0$ y $b_i/p_i >= b_k/p_k$

$$\sum_{i=1}^{k-1} (x_i - y_i) p_i \frac{b_i}{p_i} \geq \sum_{i=1}^{k-1} (x_i - y_i) p_i \frac{b_k}{p_k}$$

$$(x_k - y_k) p_k \frac{b_k}{p_k} = (x_k - y_k) p_k \frac{b_k}{p_k}$$

- $x_i - y_i <= 0$ y $b_i/p_i <= b_k/p_k$

$$\sum_{i=k+1}^{n} (x_i - y_i) p_i \frac{b_i}{p_i} \geq \sum_{i=k+1}^{n} (x_i - y_i) p_i \frac{b_k}{p_k}$$

Demostración de optimalidad

$$\sum_{i=1}^{n}(x_i - y_i)b_i \geq \sum_{i=1}^{n}(x_i - y_i)p_i \frac{b_k}{p_k}$$

$$\frac{b_k}{p_k}\sum_{i=1}^{n}(x_i - y_i)p_i$$

$\Sigma_i x_i p_i = \mathcal{M}$ por hipótesis, pero $\Sigma_i y_i p_i \leq \mathcal{M}$. Luego

$$\sum_{i=1}^{n}(x_i - y_i)b_i \geq 0$$

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)

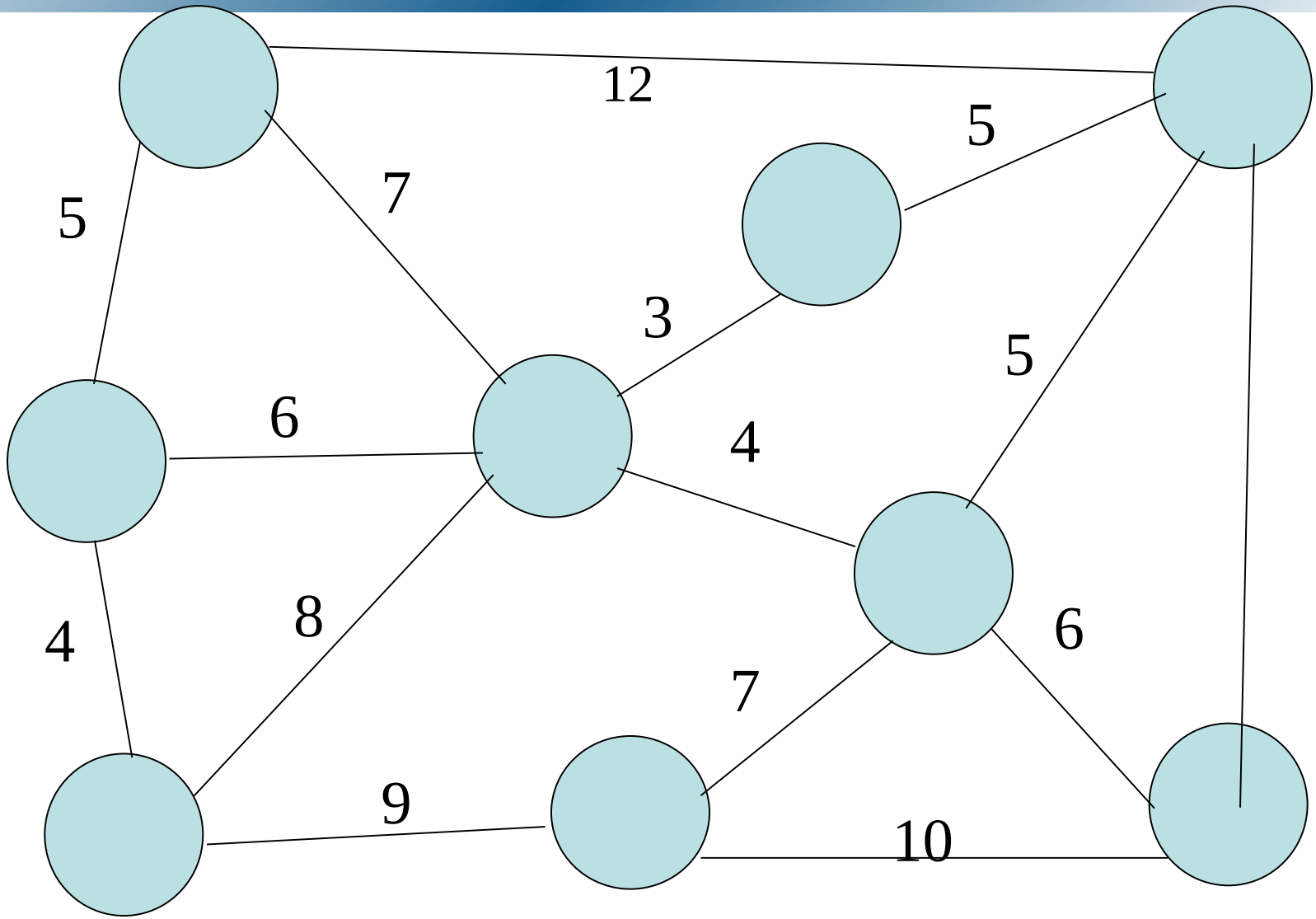
- Dos enfoques para la solución:
  - Basado en aristas: algoritmo de Kruskal
  - Basado en vértices: algoritmo de Prim

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 Mínimal. 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

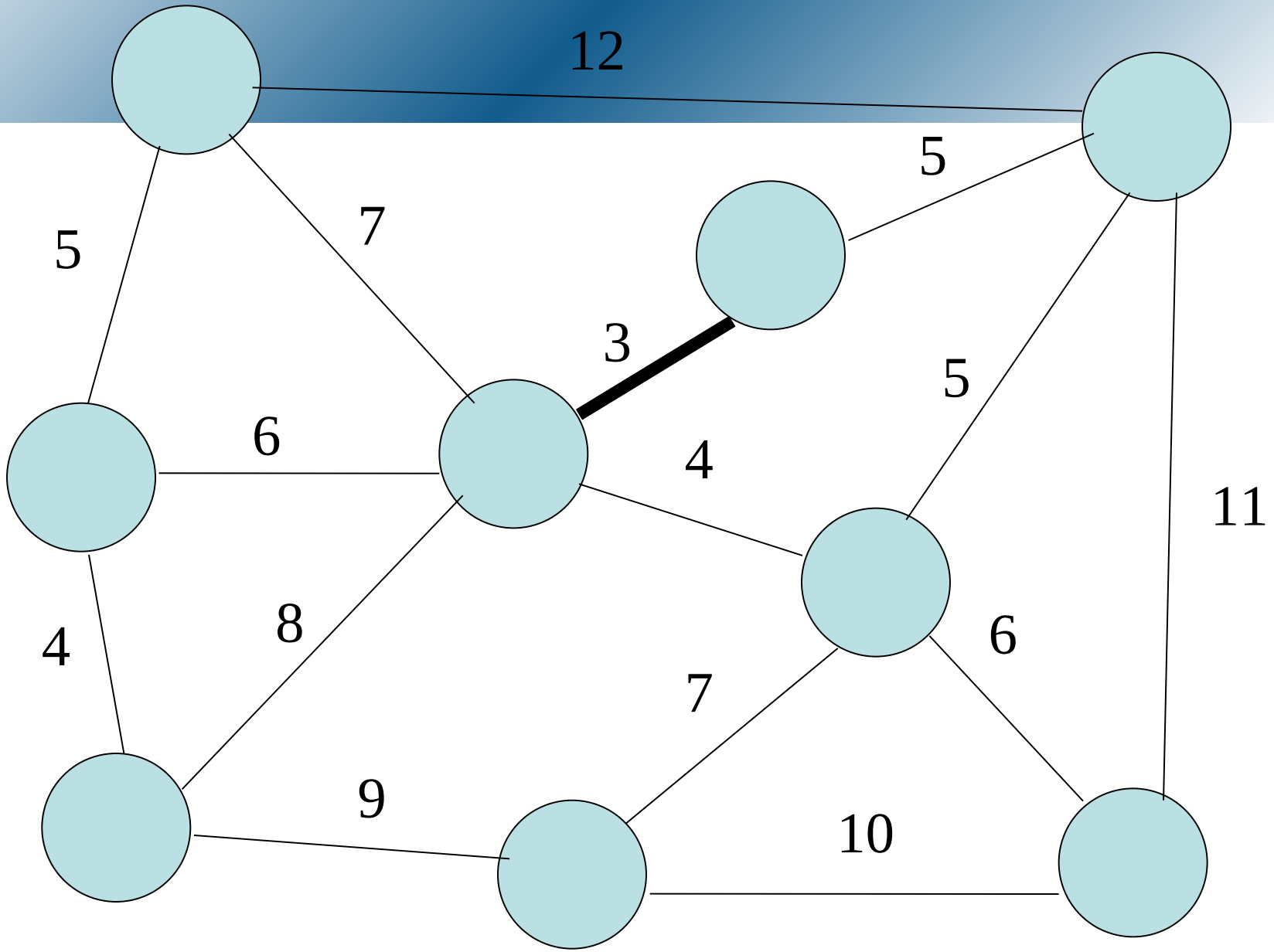
- Función Solución: un conjunto de aristas que conecta todos los vértices

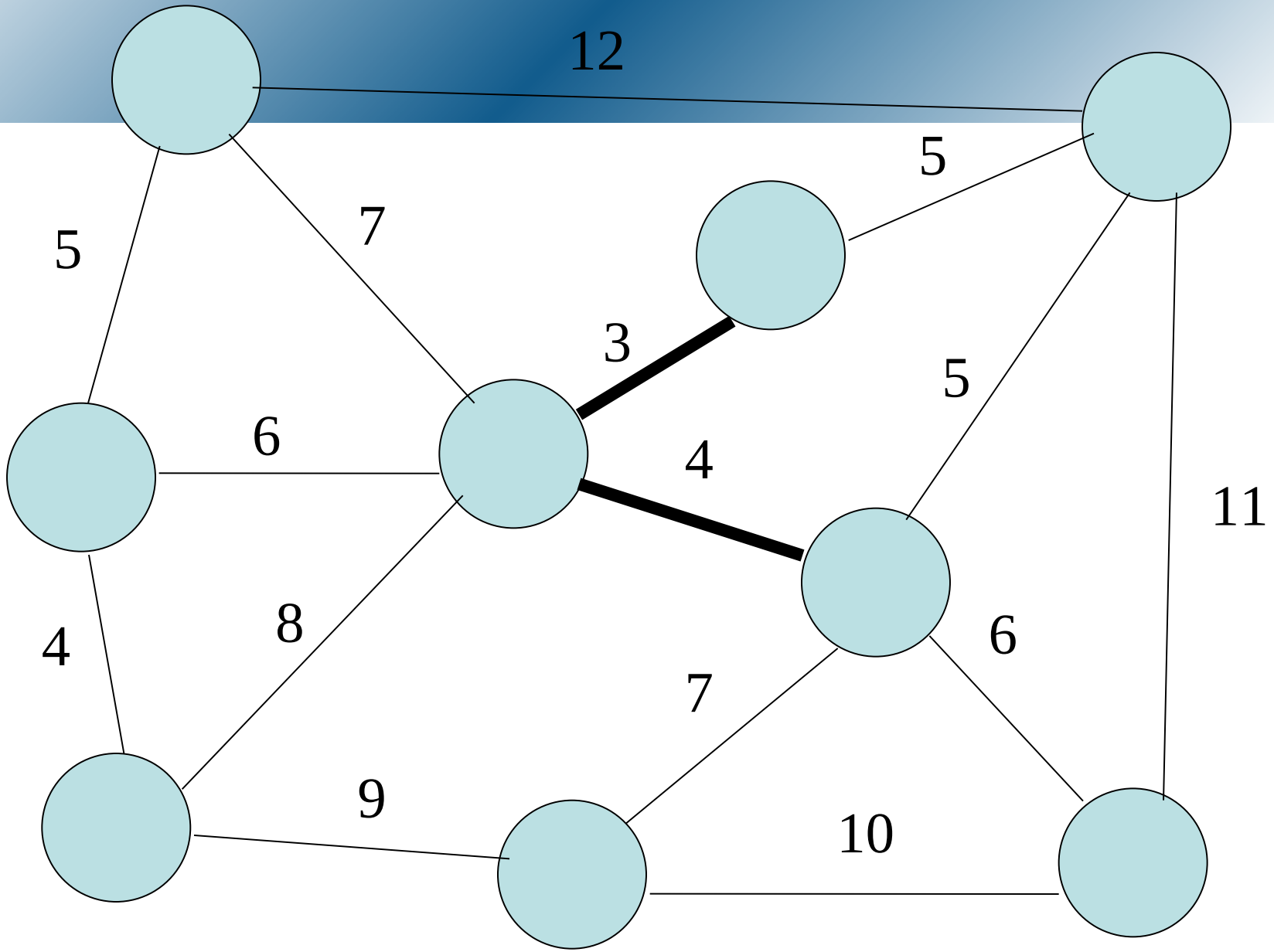
  Se ha construido un árbol de recubrimiento (n-1 aristas seleccionadas).

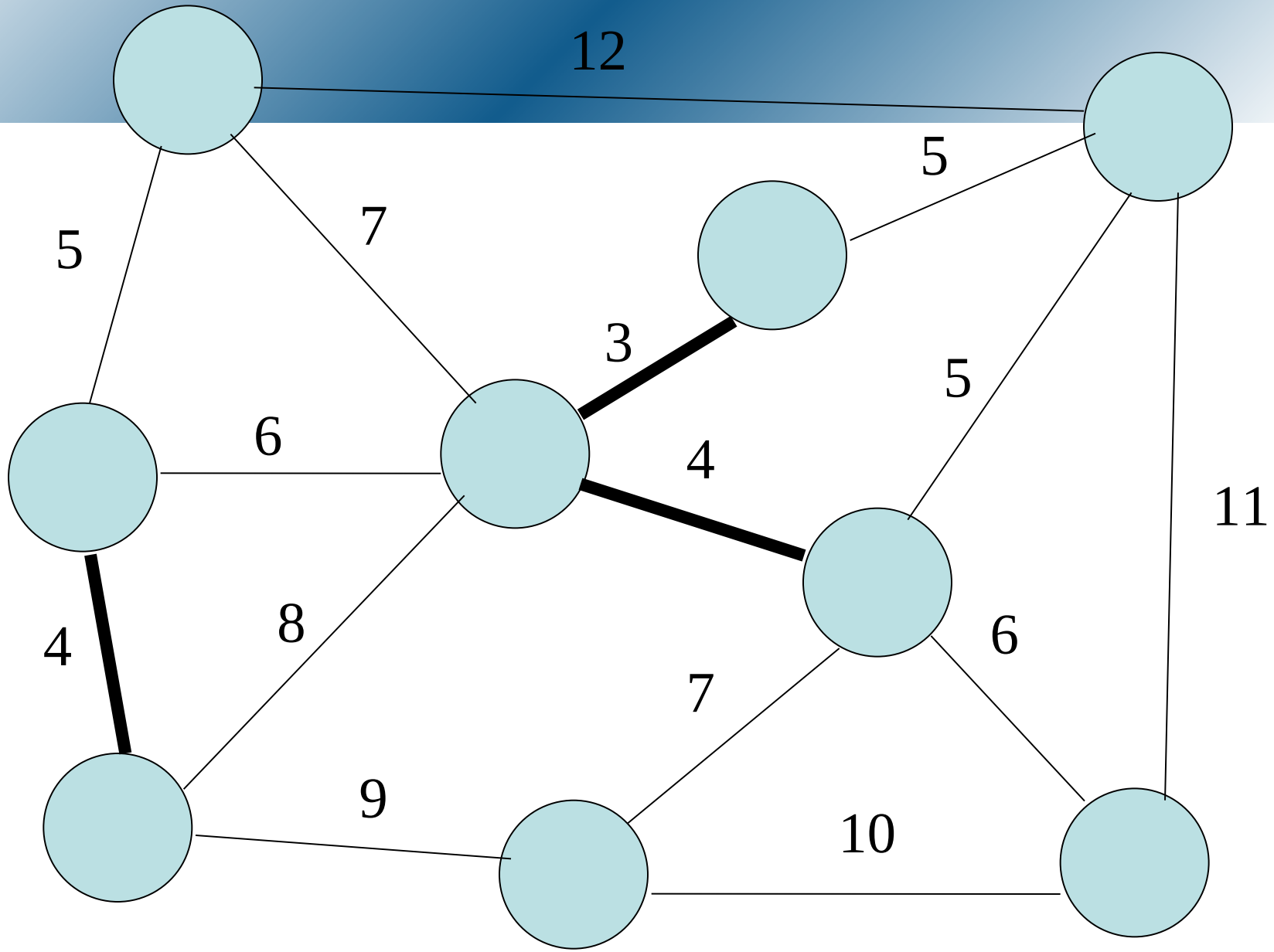
- Función factible: el conjunto de aristas no forma ciclos
- Función selección: la arista de menor coste
- Función objetivo: minimizar la suma de los costes de las aristas

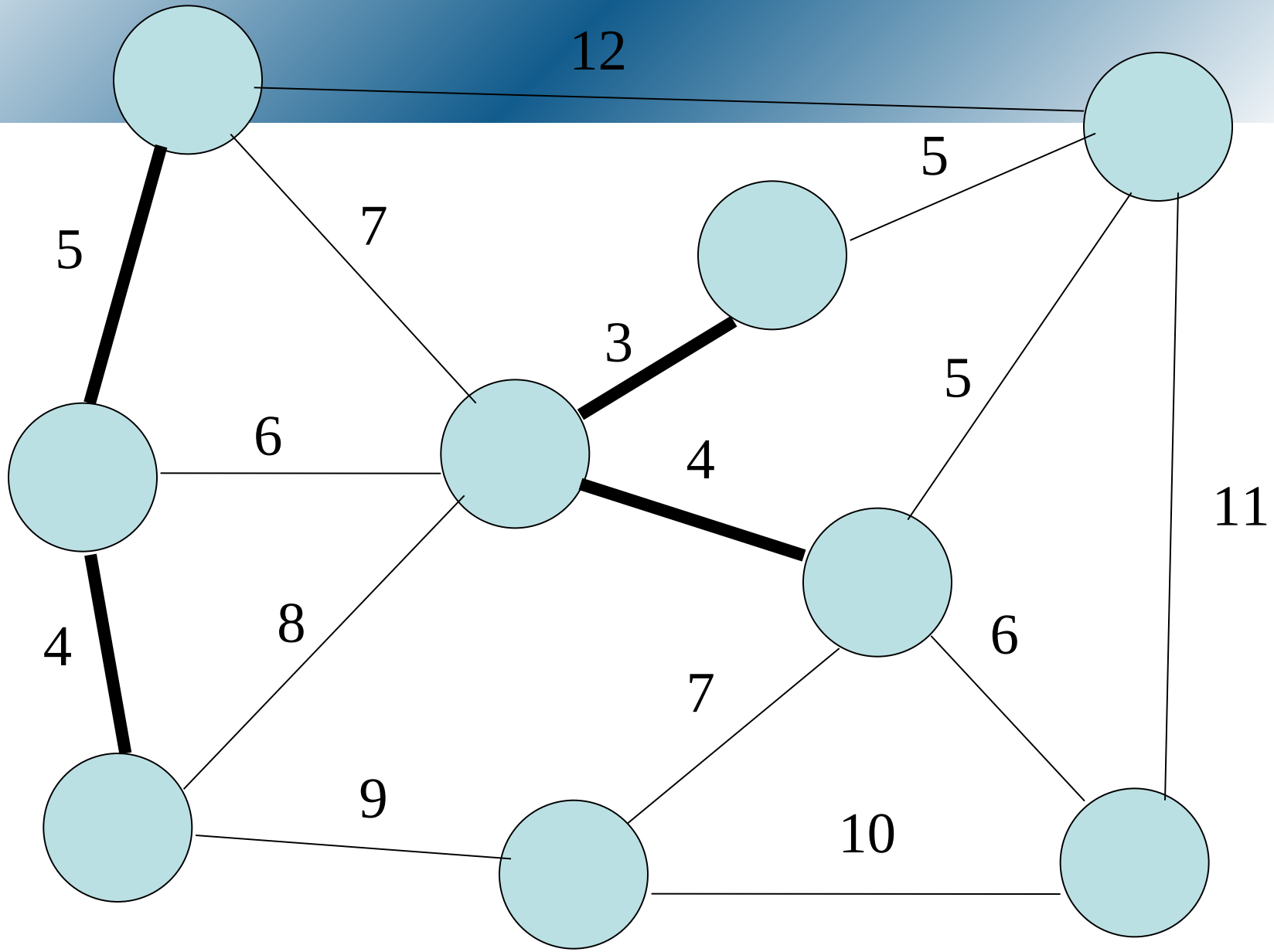
Algoritmo de Kruskal

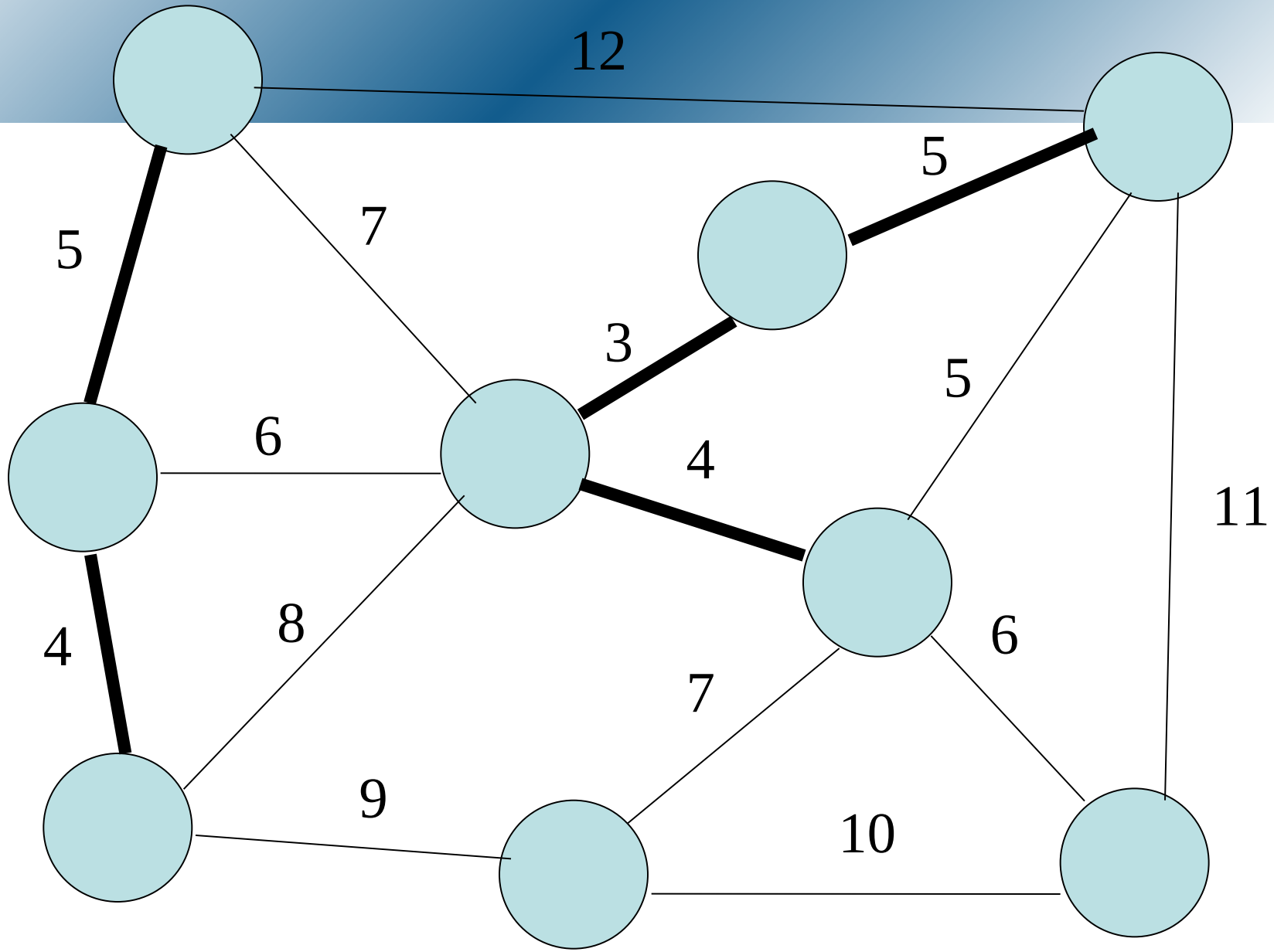
```
 Kruskal_I (Grafo G(V,A))
{ set<arcos> C(A);
  set<arcos> S;                  // Solución inic. Vacía
  Ordenar(C);    // de menor a mayor costo
  while (!C.empty() && S.size()!=V.size()-1) { //No solución
     x = C.first(); //seleccionar el menor
     C.erase(x);
      if (!HayCiclo(S,x))  //Factible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

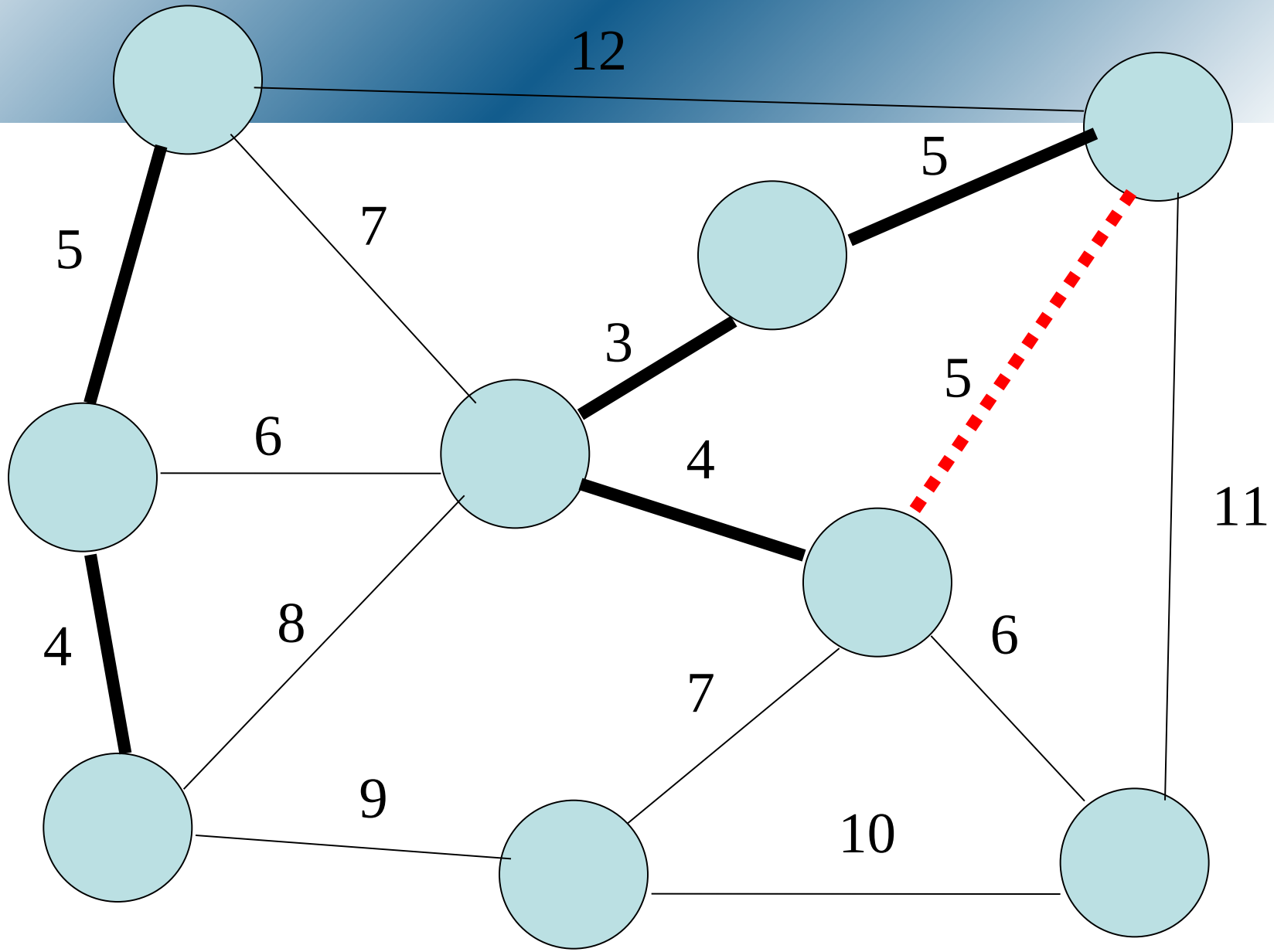

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

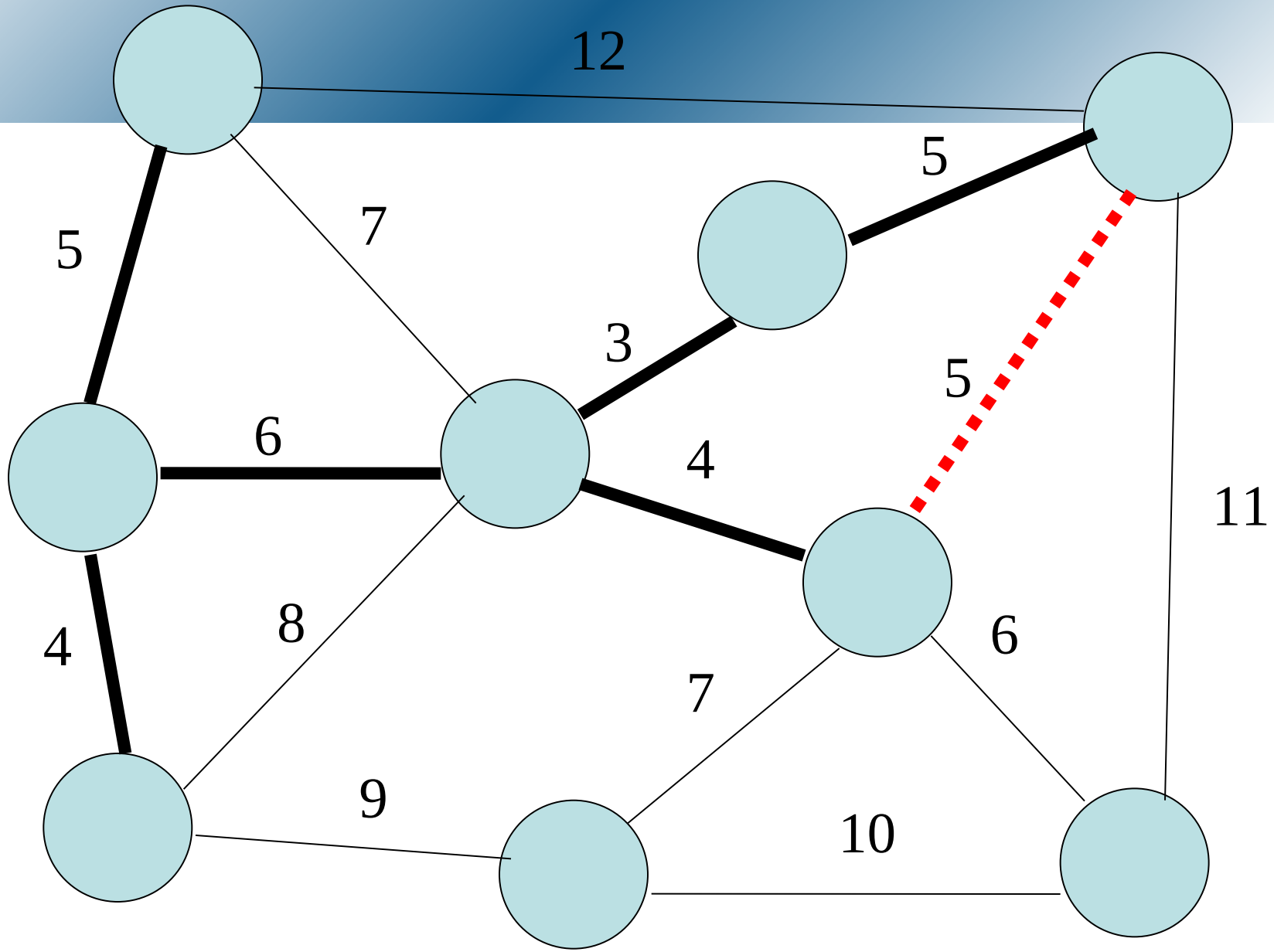

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

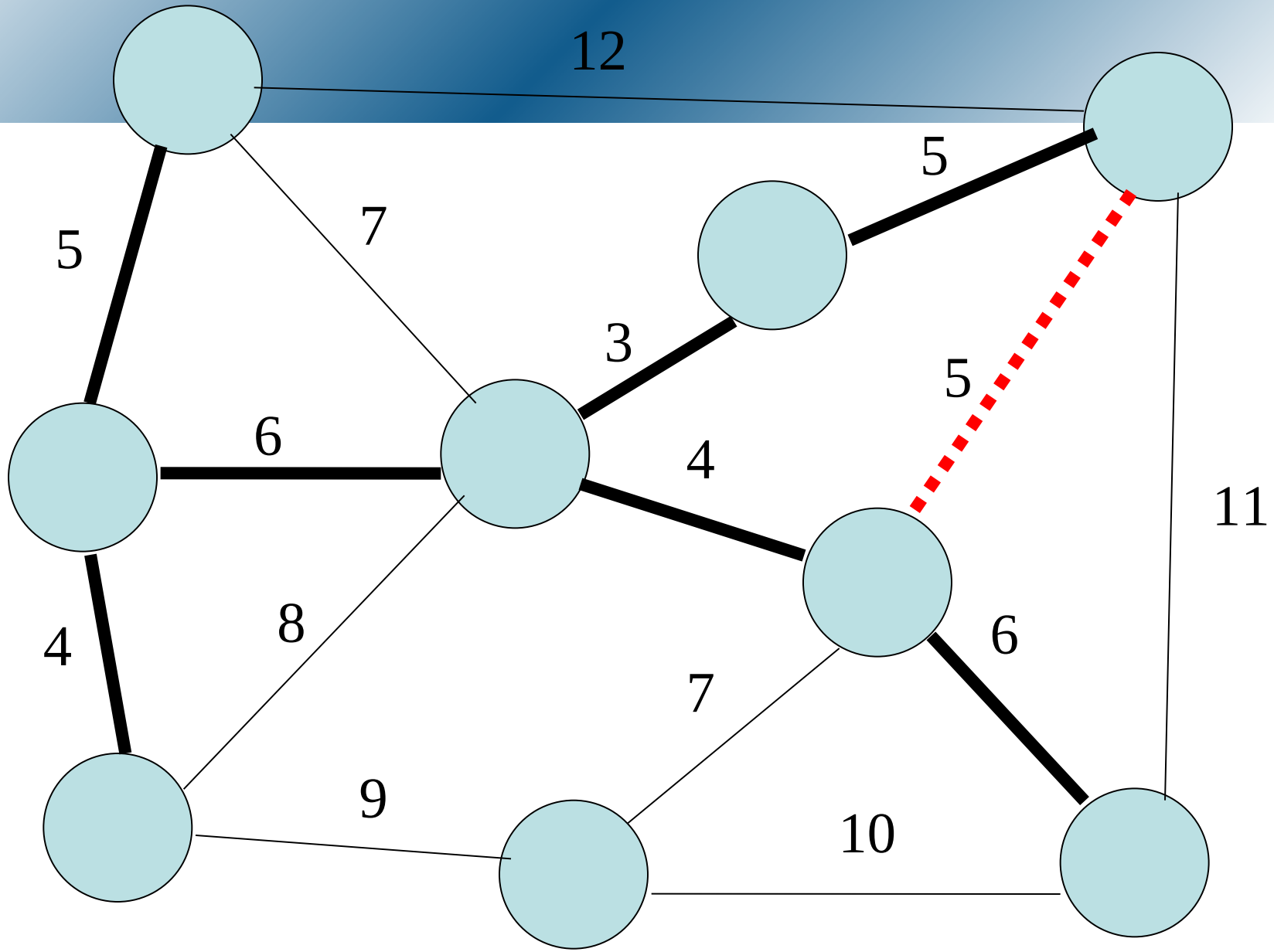

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

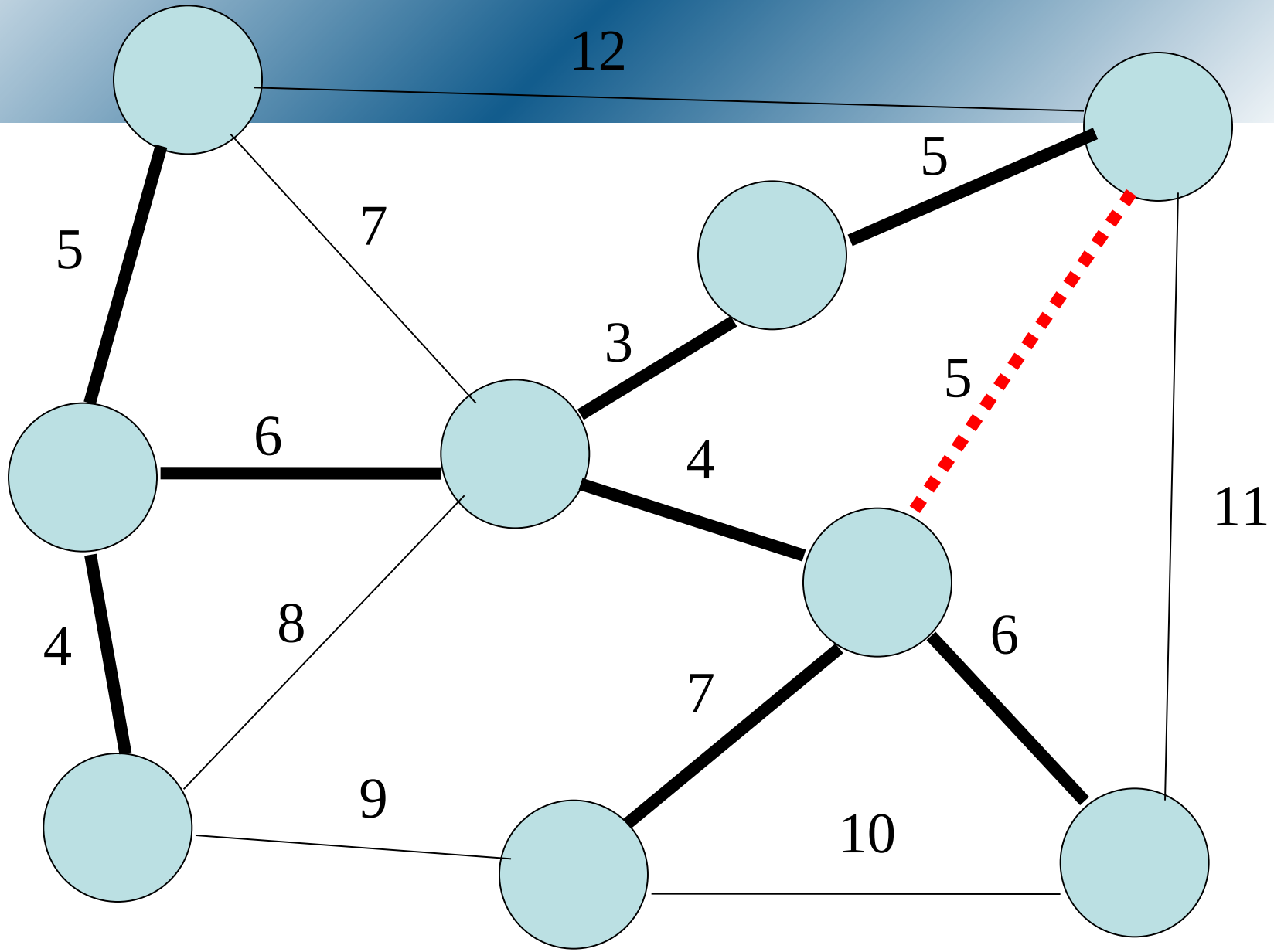

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

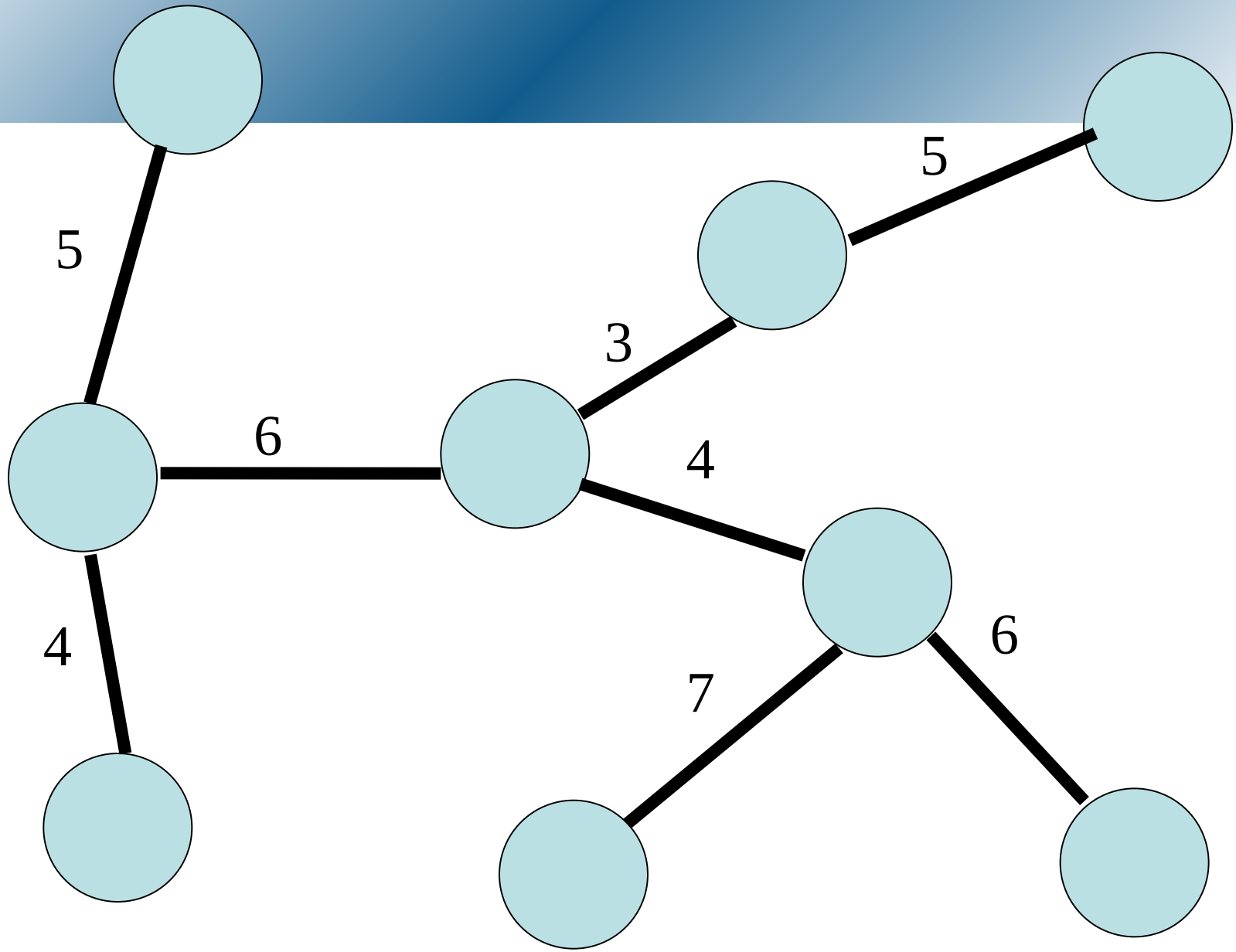

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10


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

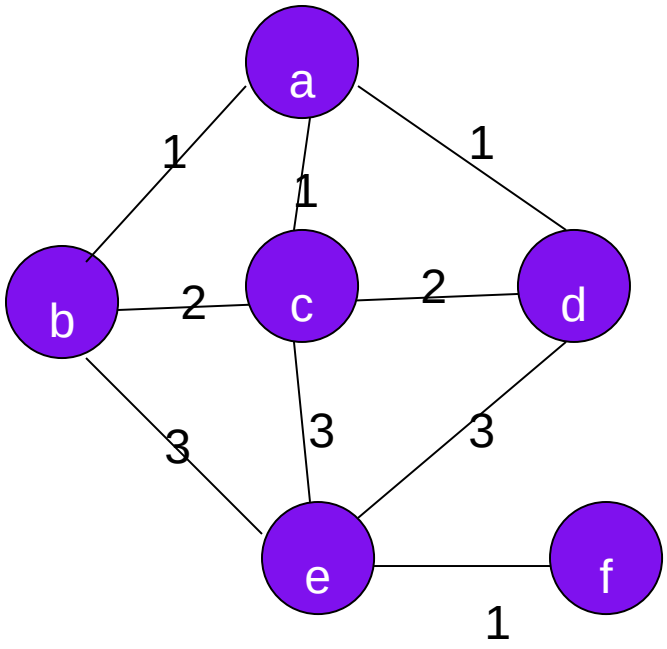

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10


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

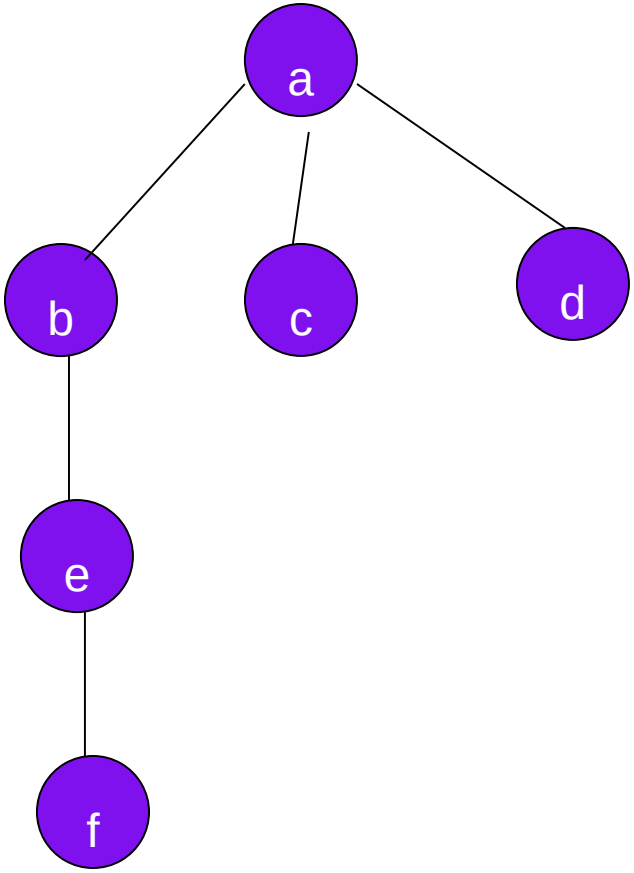

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10


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).

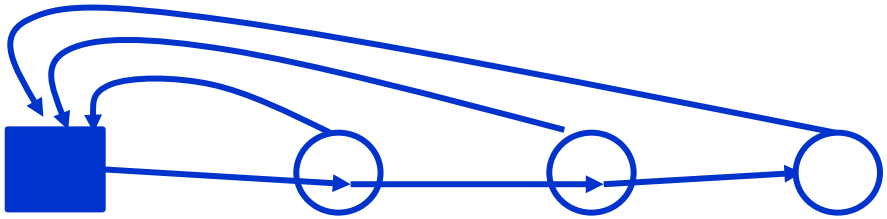
La función factible me acepta una arista de menor costo que una dos componentes conexas, así garantizamos que no hay ciclos.

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



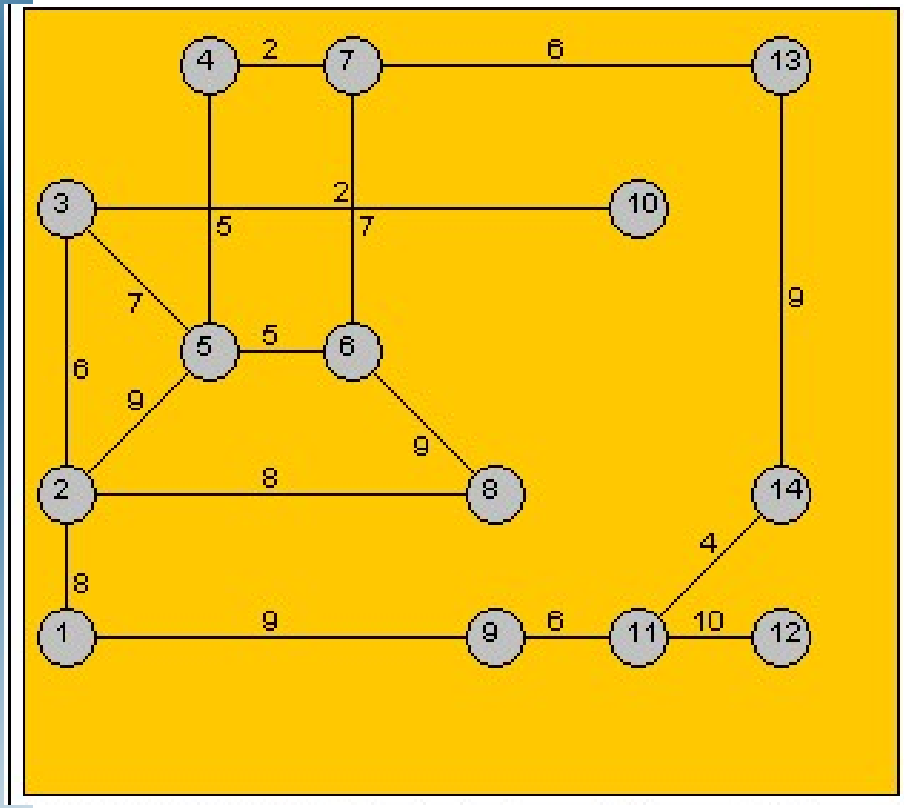
- MakeSet(): O(1)
- FindSet(): O(1)
- Union(A,B): O(log V)

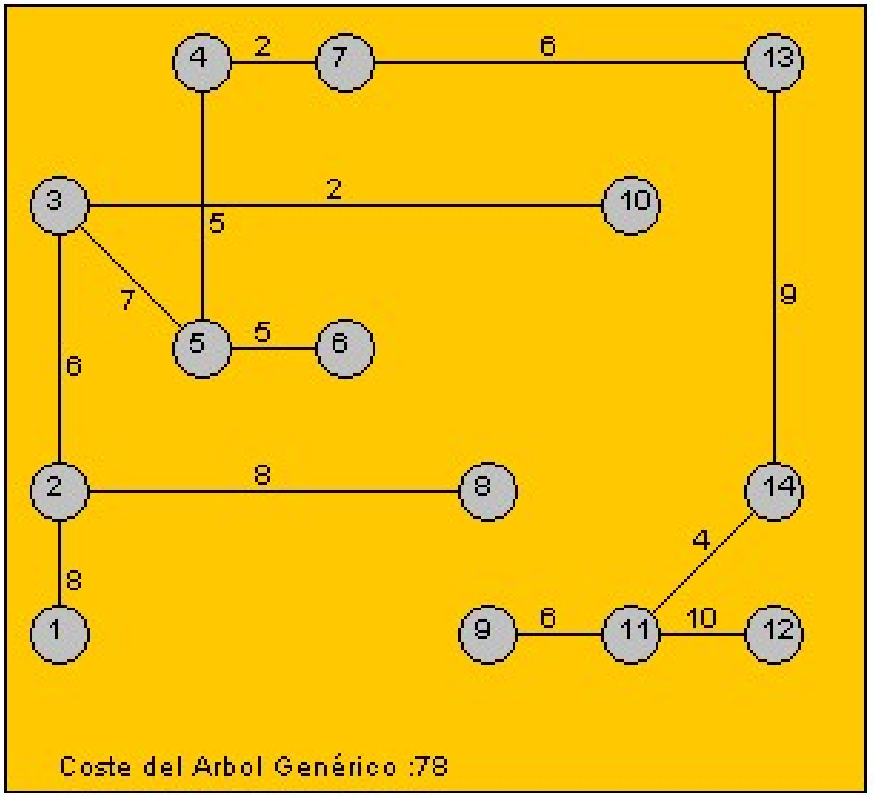
Análisis de Eficiencia

```
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)
     }
}
```

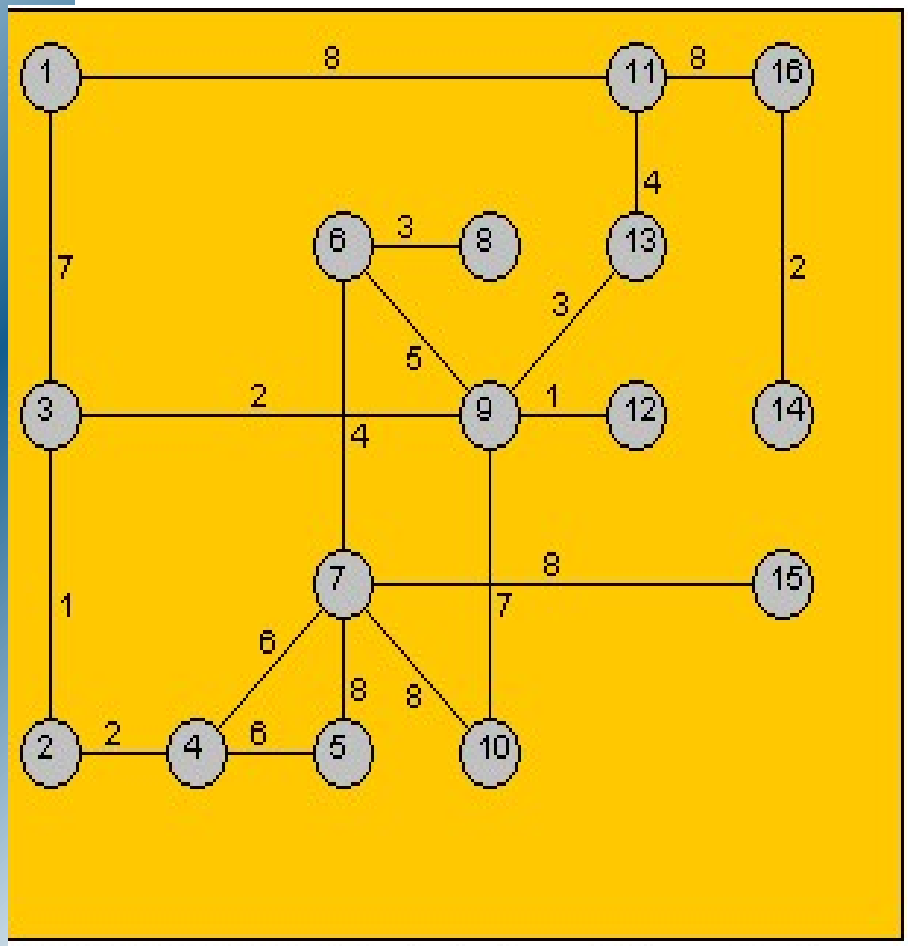
Kruskal es de O(A log V)

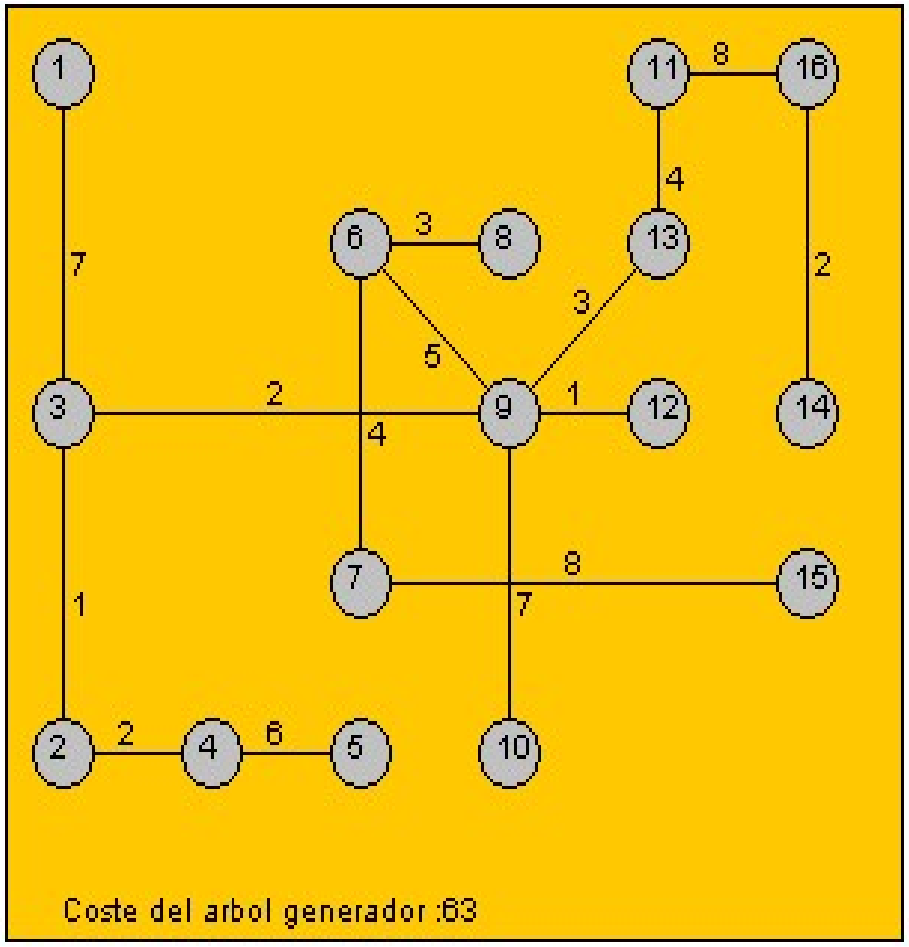
Otro ejemplo de Kruskal





Y otro más


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

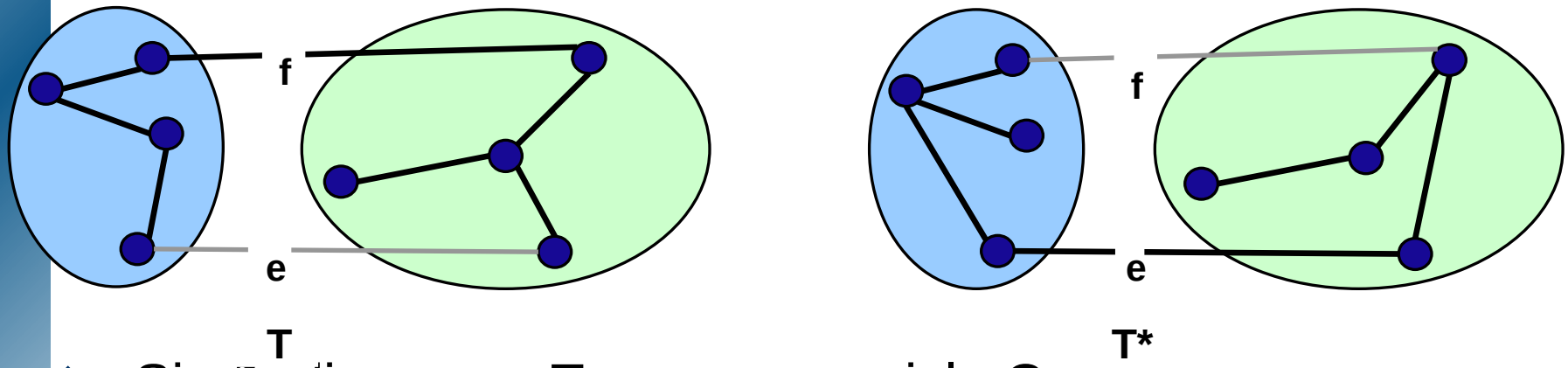

1
11
16
6
8
13
3
9
12
14
7
15
2
4
5
10
Coste del arbol generador :63

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)



- 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 = $\oslash$ (Contendrá las aristas del AGM que buscamos).

U = {un miembro arbitrario de V}

MIENTRAS ıUı != n HACER

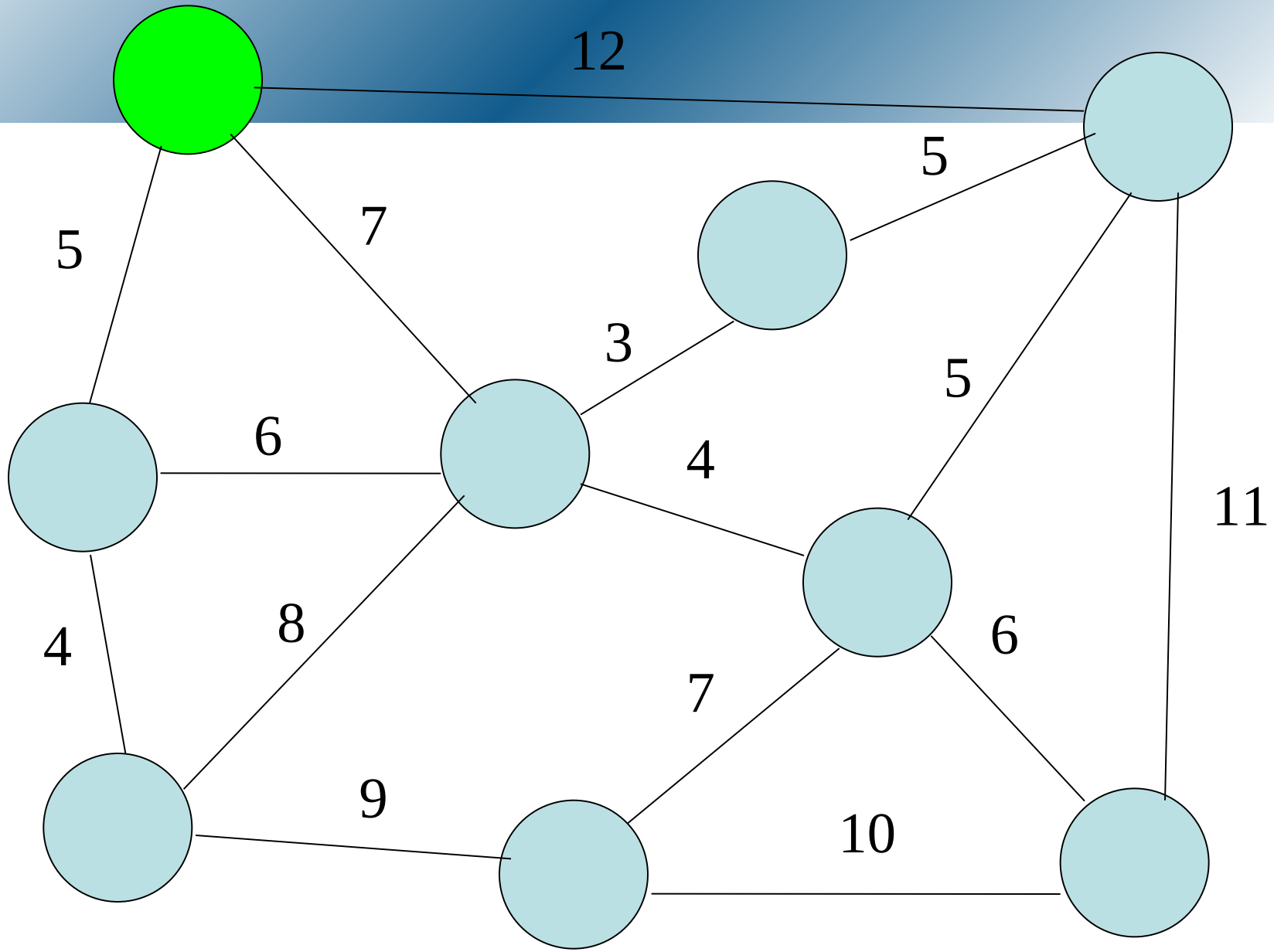
BUSCAR e = (u,v) de longitud mínima tal que

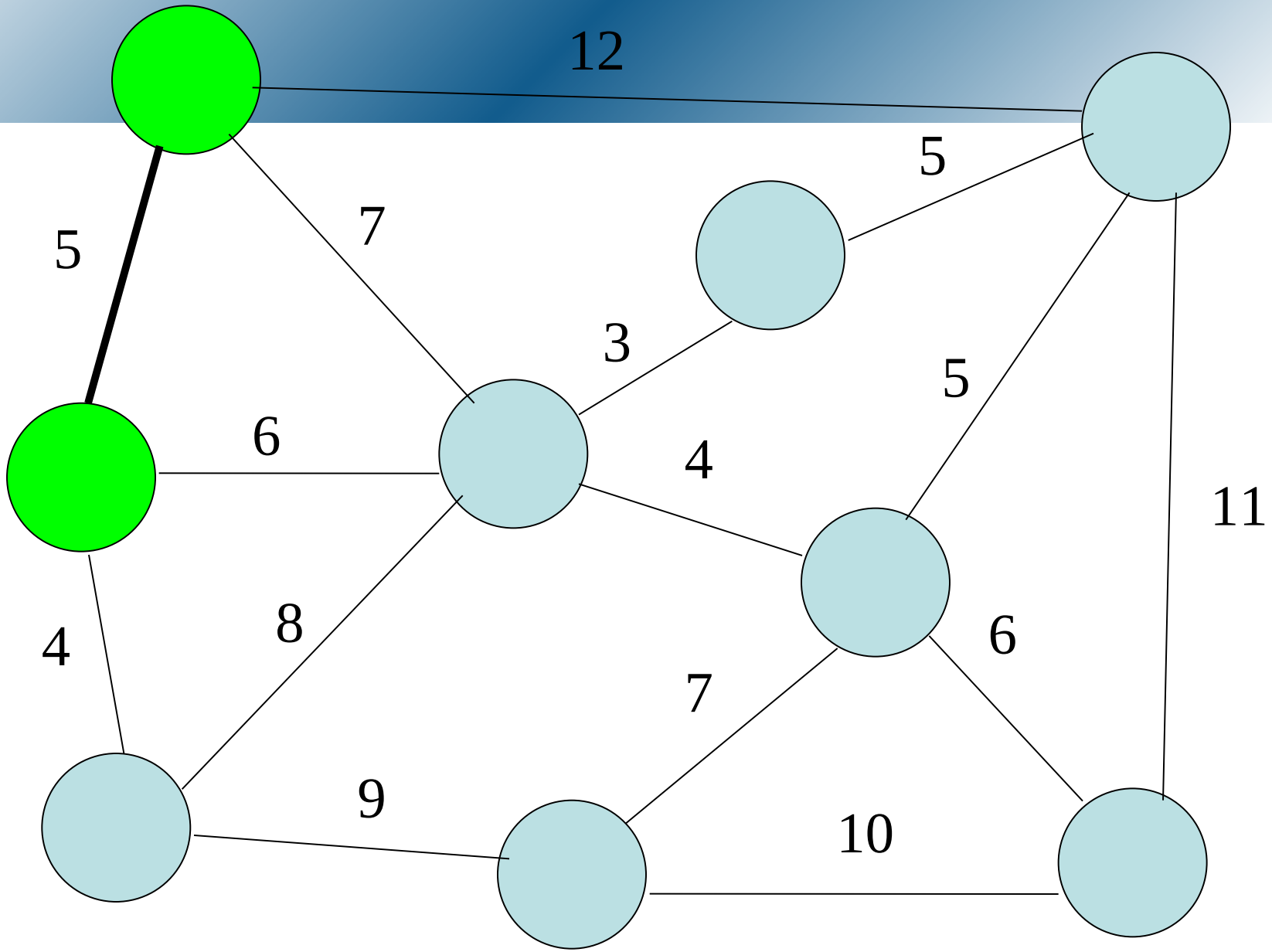
$u \in U$ y $v \in V - U$

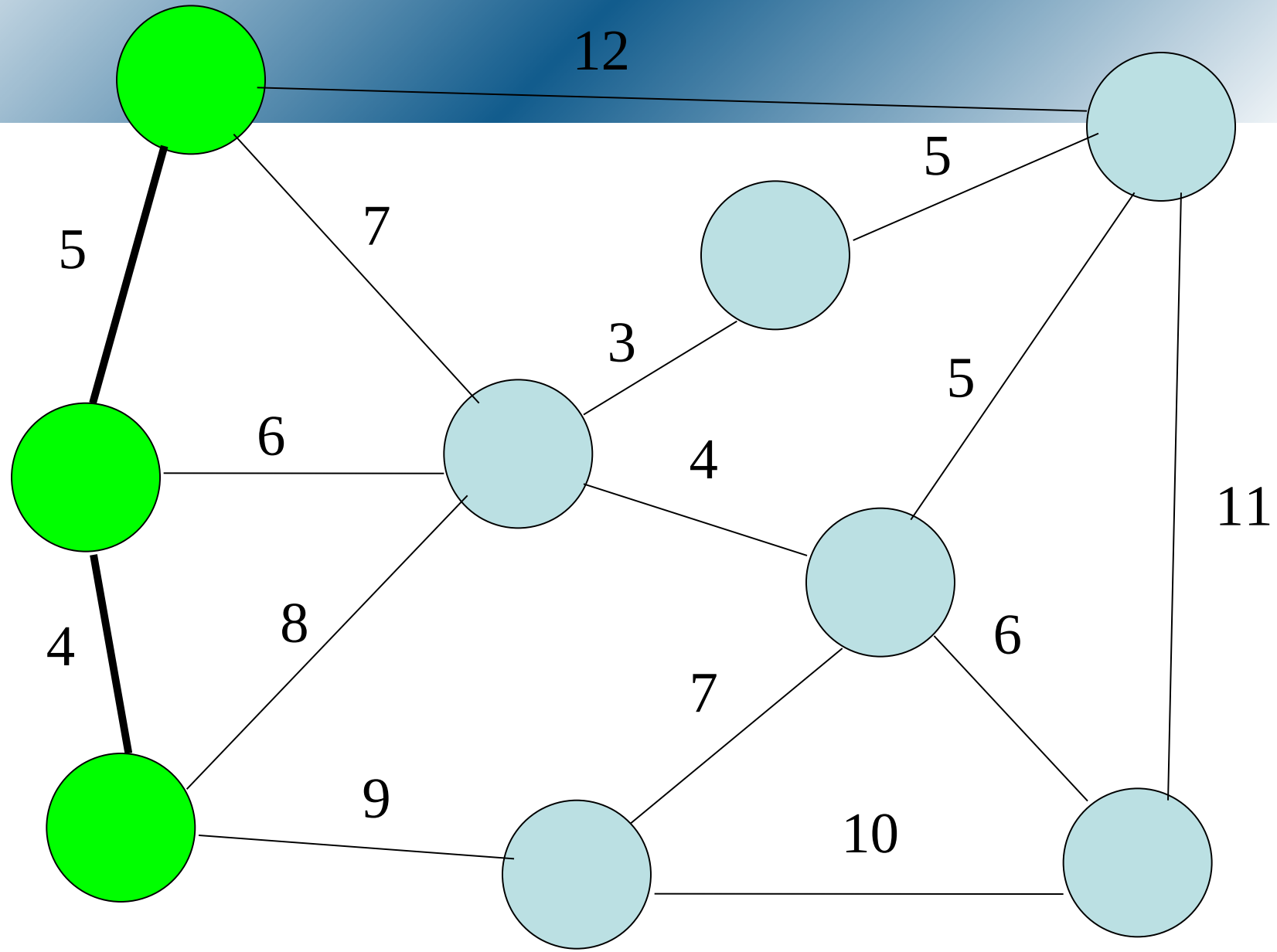
T = T + e

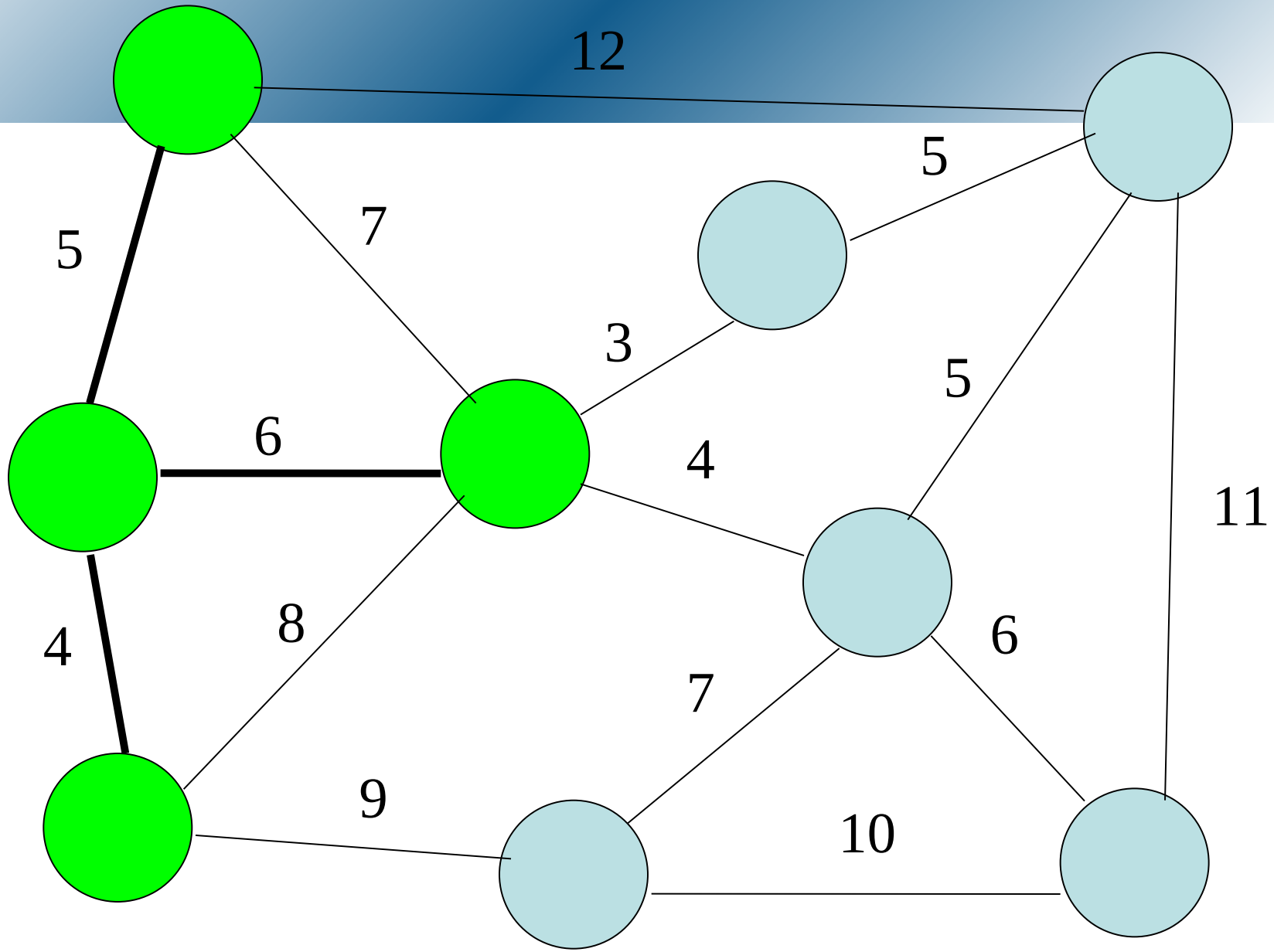
U = U + v

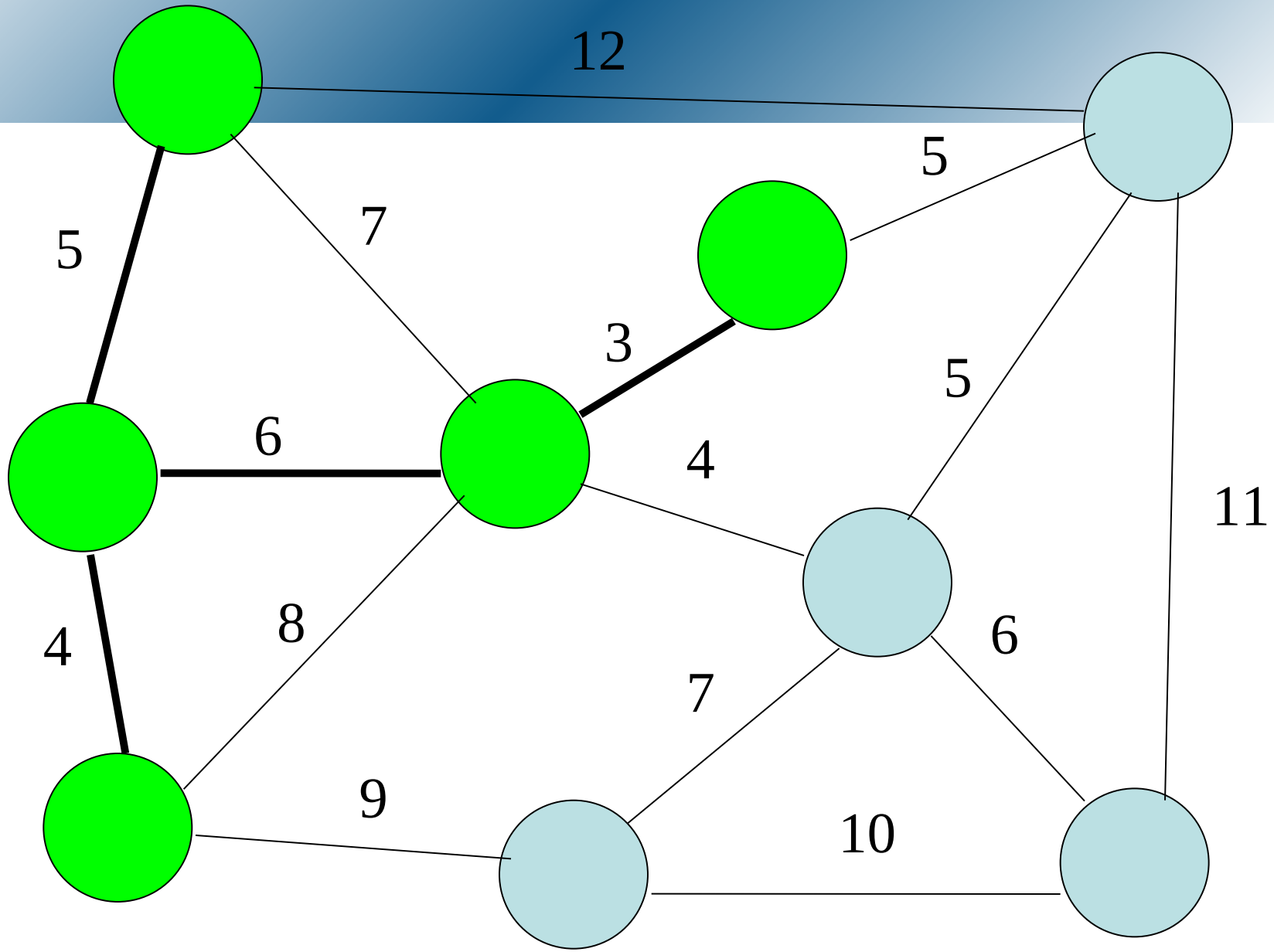
DEVOLVER (T)

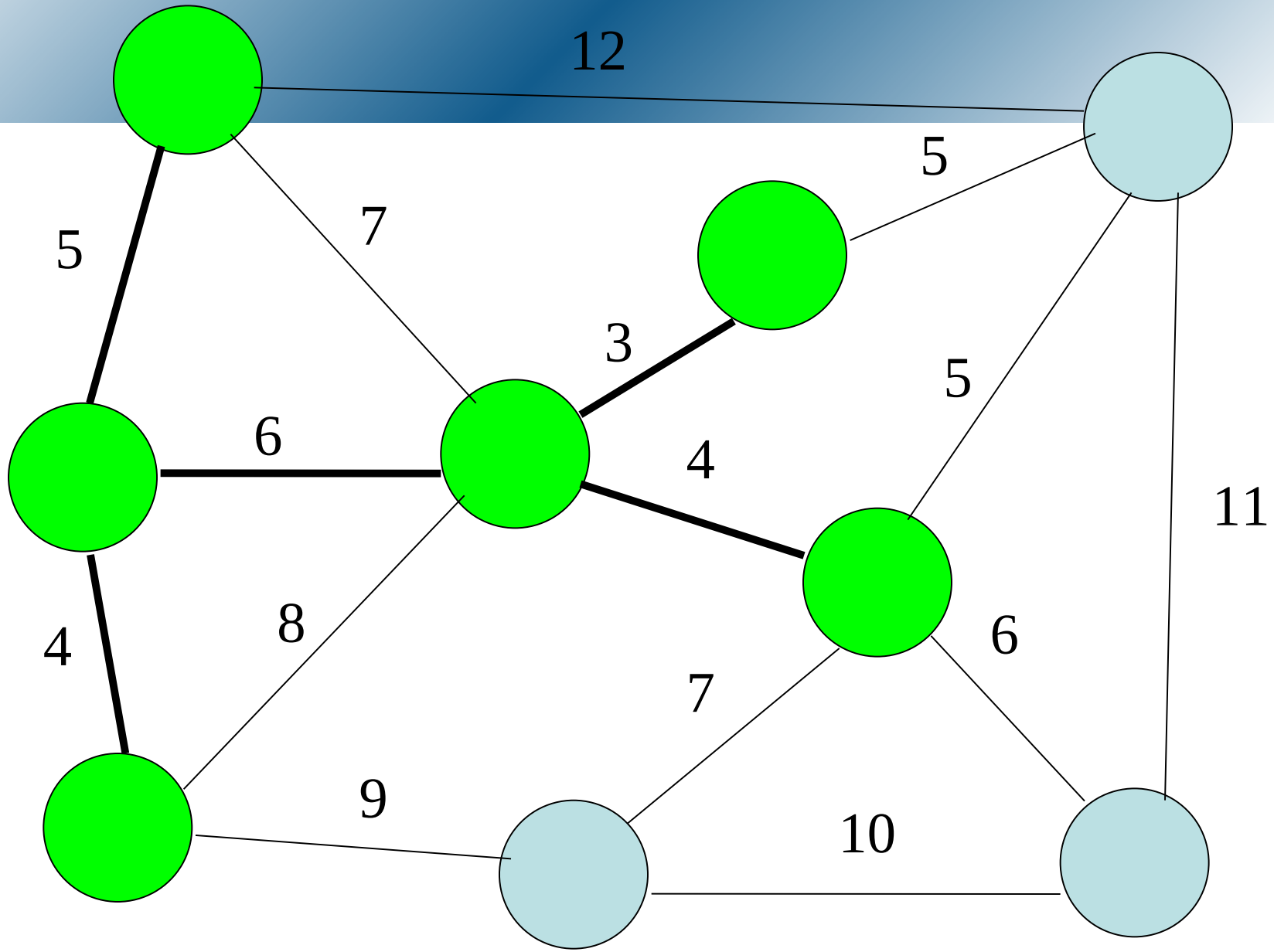

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

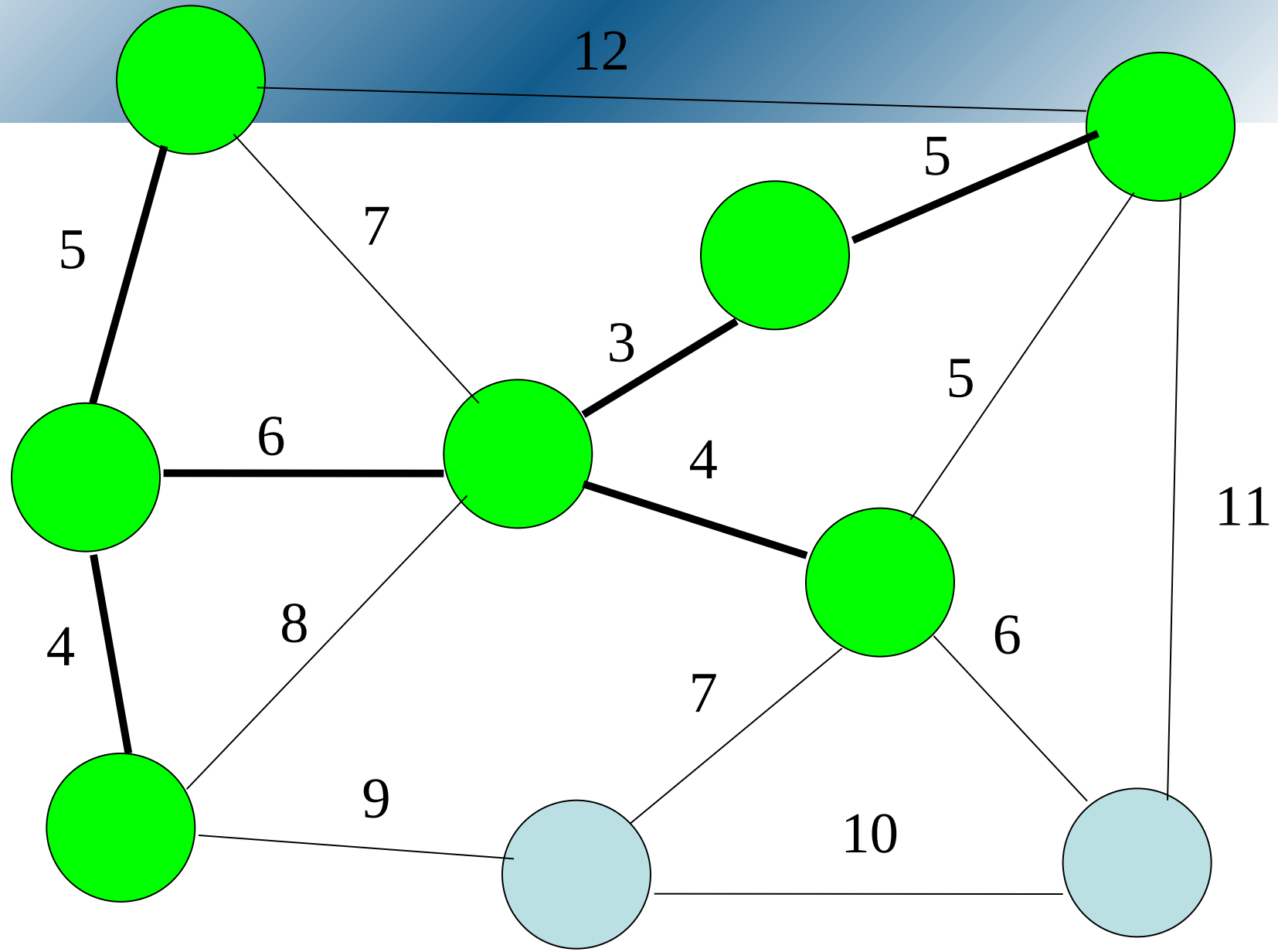

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

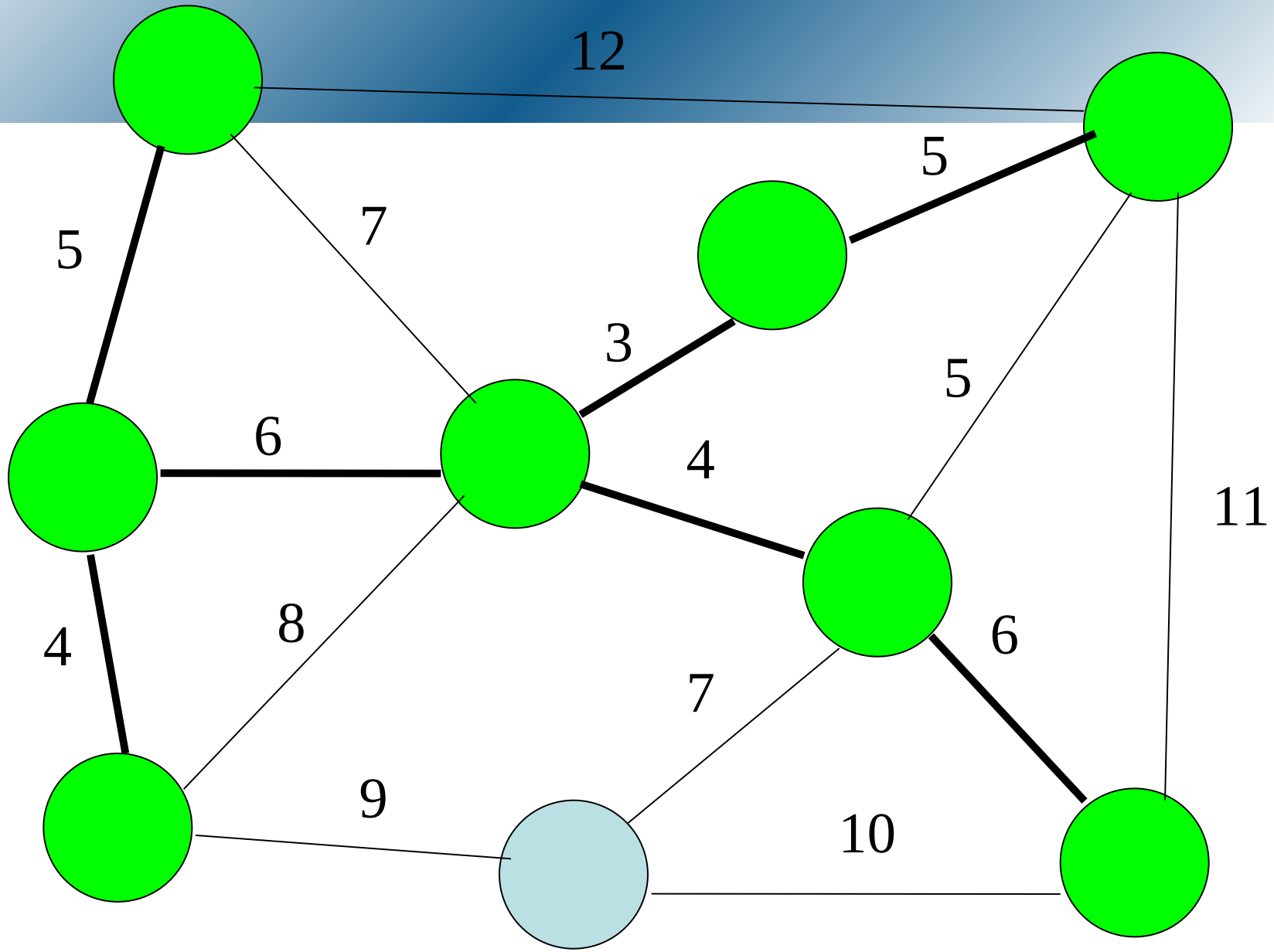

12
5
7
5
5
3
6
4
5
11
4
8
6
7
9
10

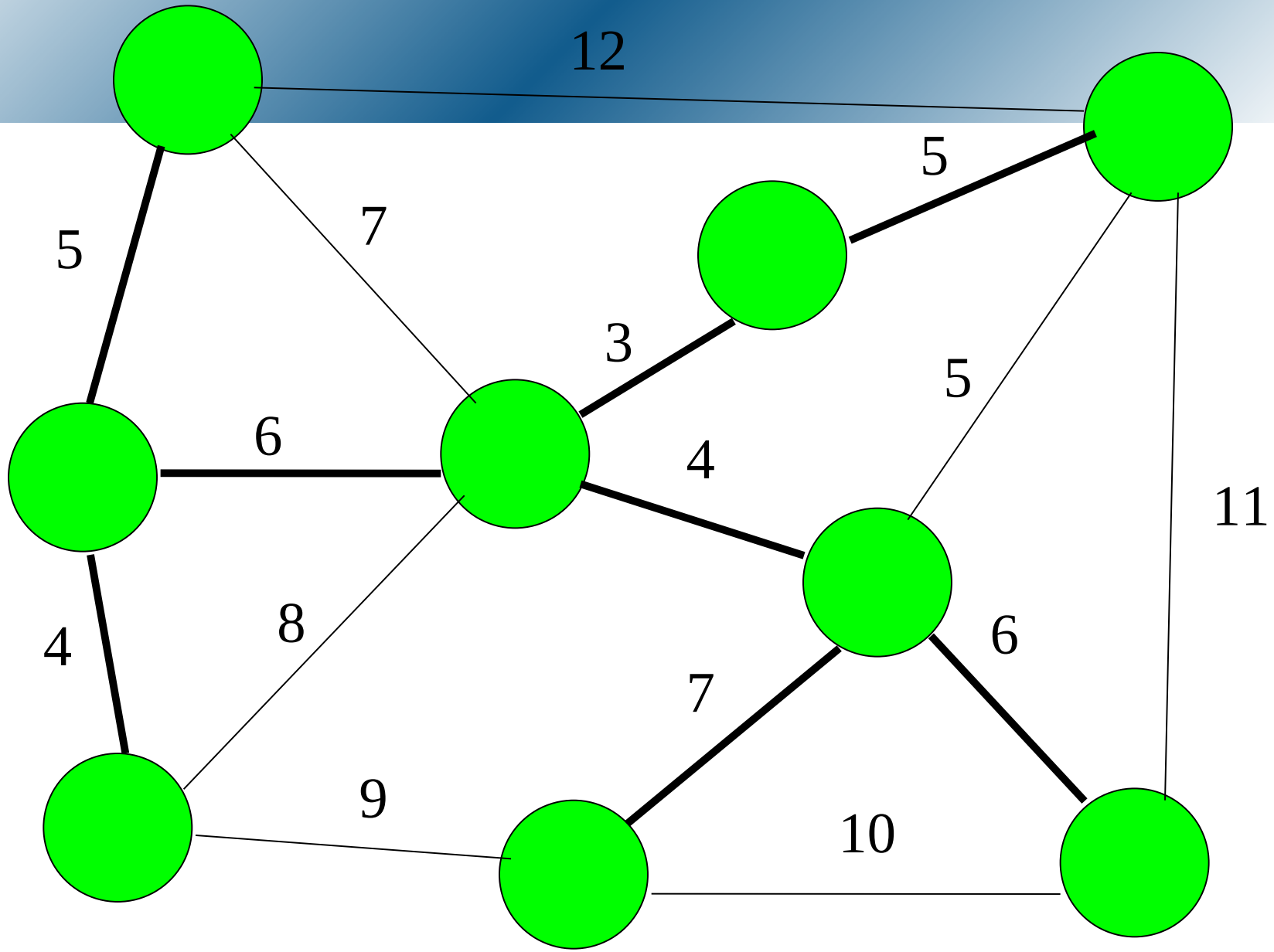

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10


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

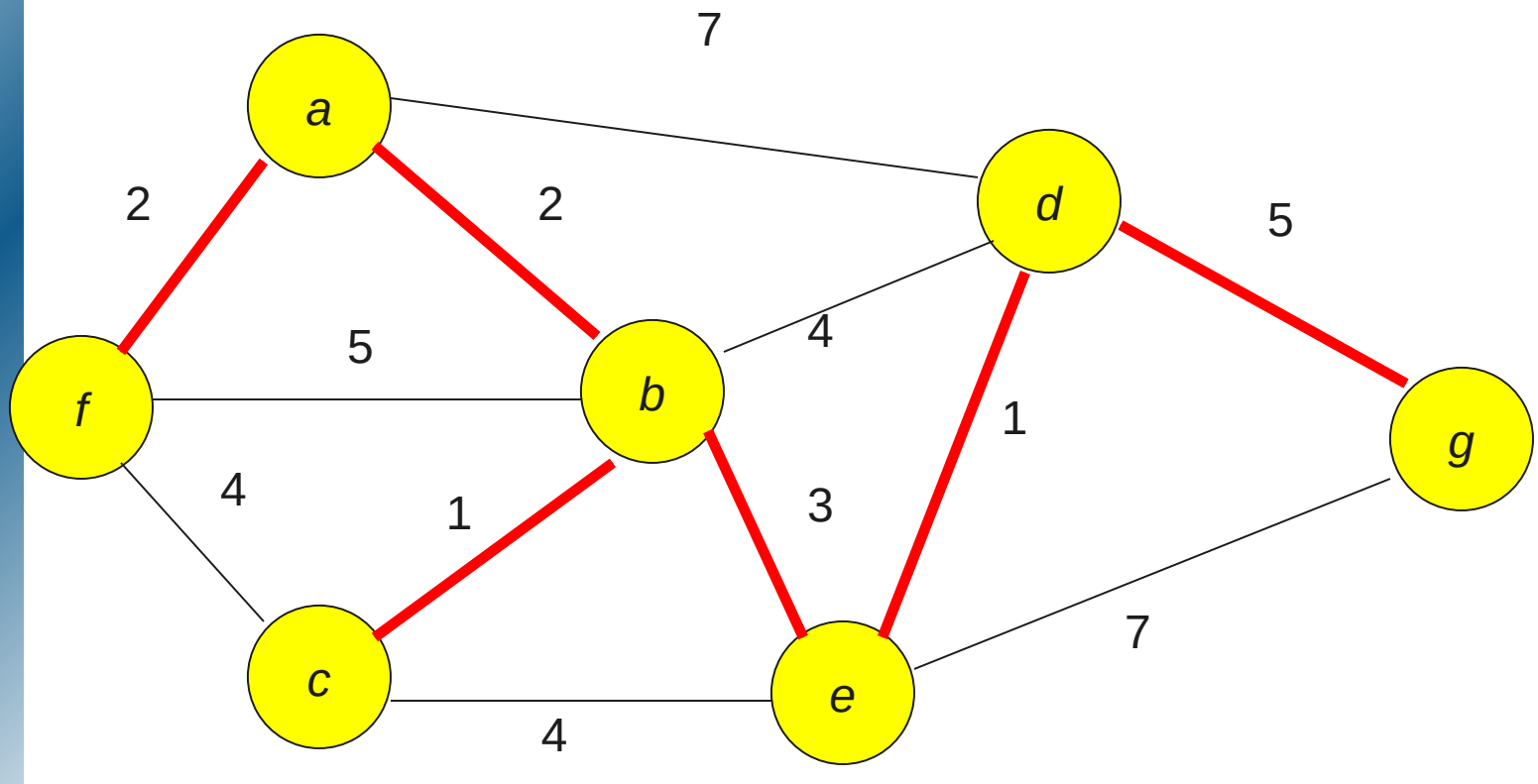

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10


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] $\geq$ 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.
```

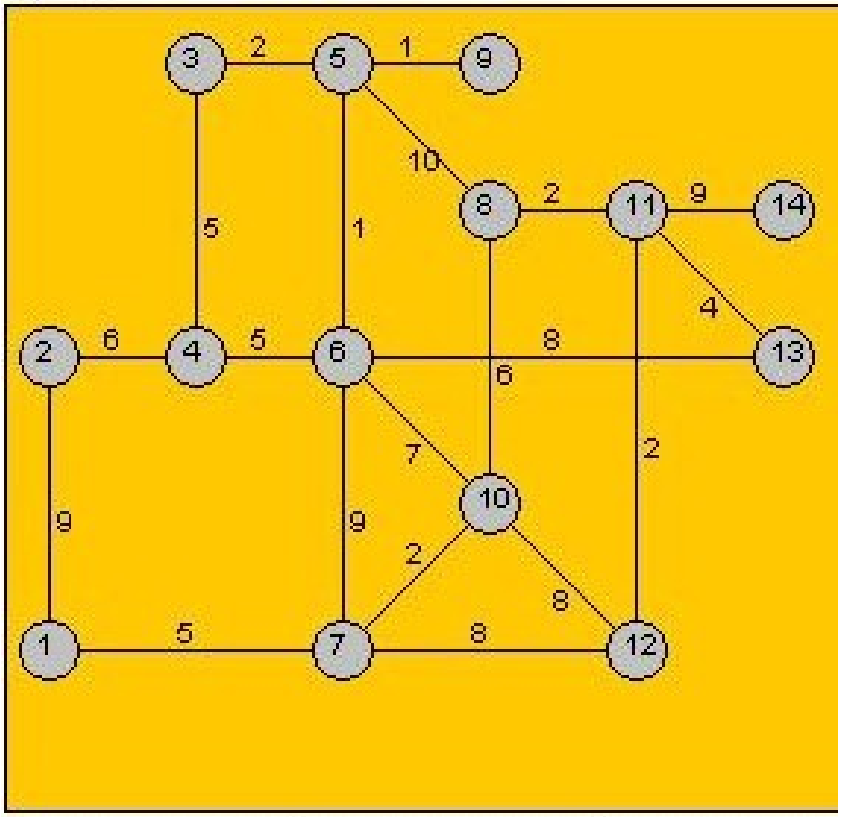
Análisis del algoritmo de Prim

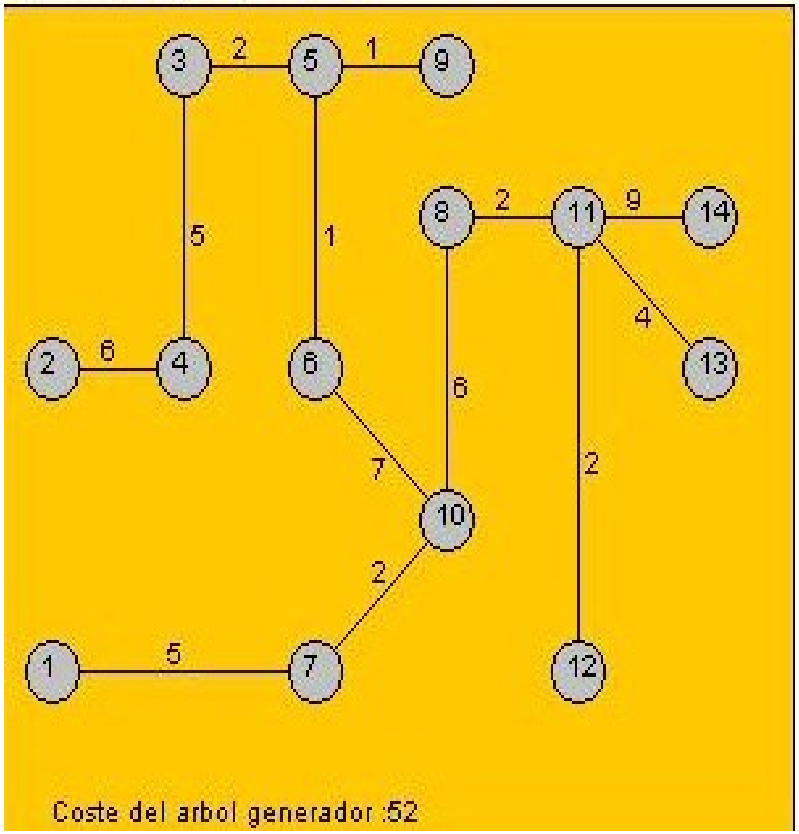
- El bucle principal del algoritmo se ejecuta n – 1 veces.
- En cada iteración, el bucle “para” anidado requiere un tiempo O(n). Por tanto, el algoritmo de Prim requiere un tiempo O (n^2).
- Como el Algoritmo de Kruskal era O(a log n), siendo a el número de aristas del grafo,
- ¿Qué algoritmo usar Prim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

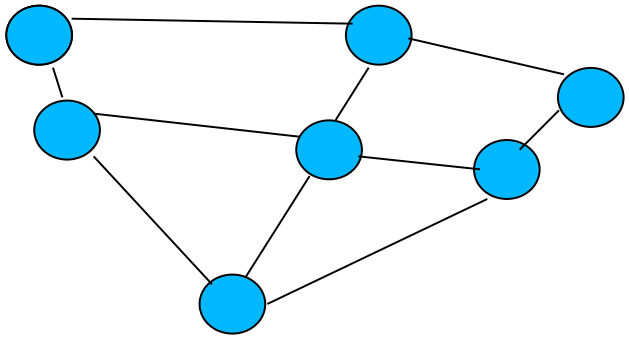
Dado un grafo ponderado se quiere calcular el camino con menor peso entre un vértice *v* y otro *w*.

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.

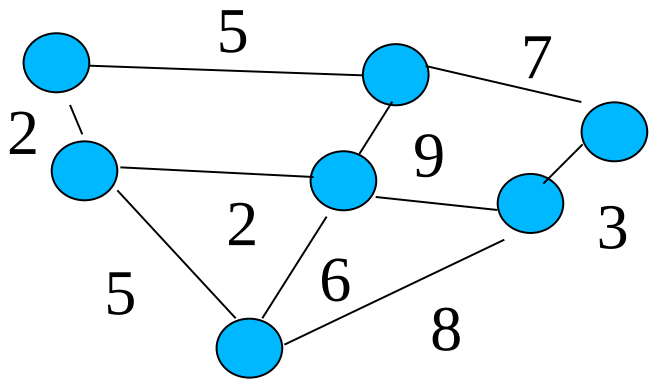


Aplicaciones

- 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.

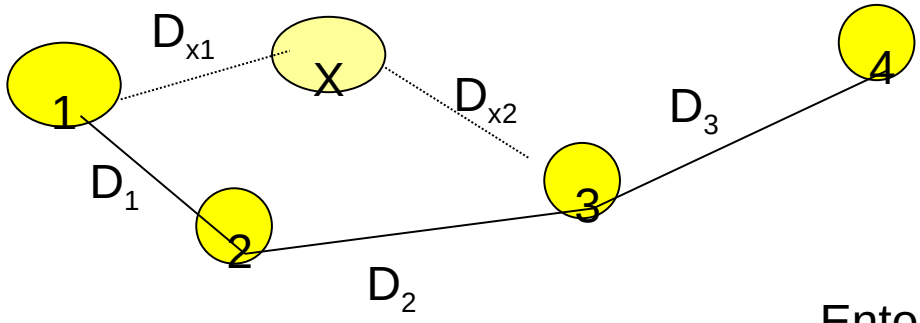
El costo del camino se define como la suma de los pesos de los arcos.

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?)



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

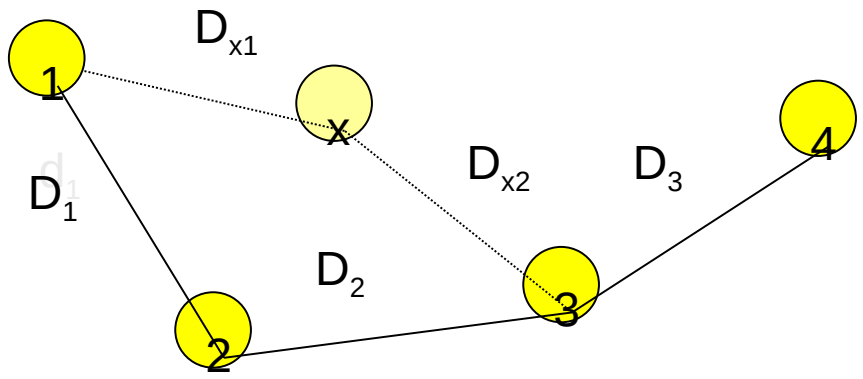
Si $M(1,3) = Dx_1+Dx_2$

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?)



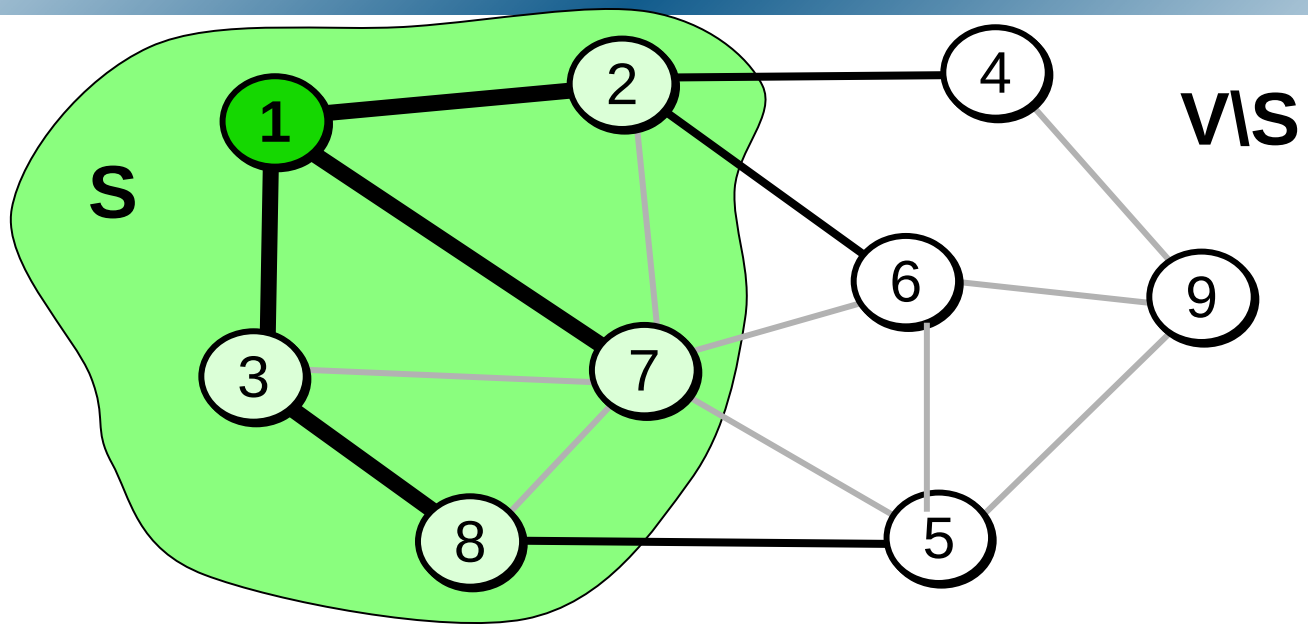
Si M(s,v)>M(s,u)+M(u,v), entonces encuentro un camino más corto de s a v pasando por u.

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

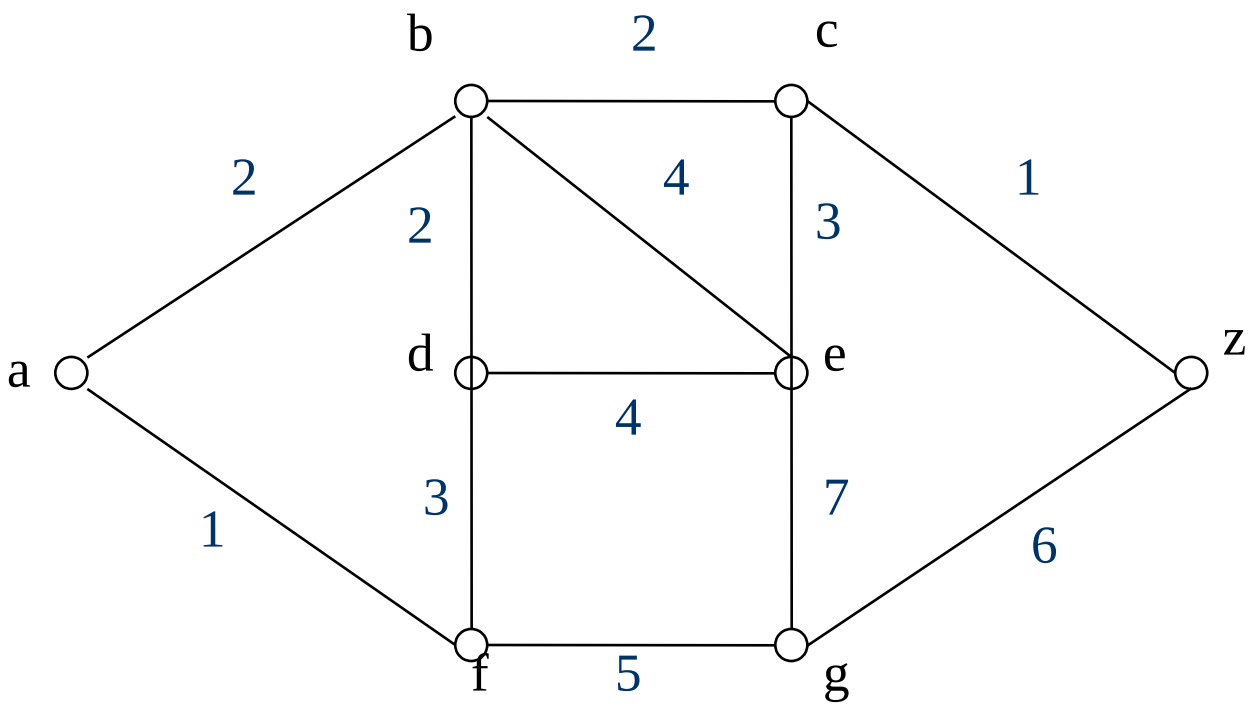
ALGORITMO DE DIJKSTRA

```
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

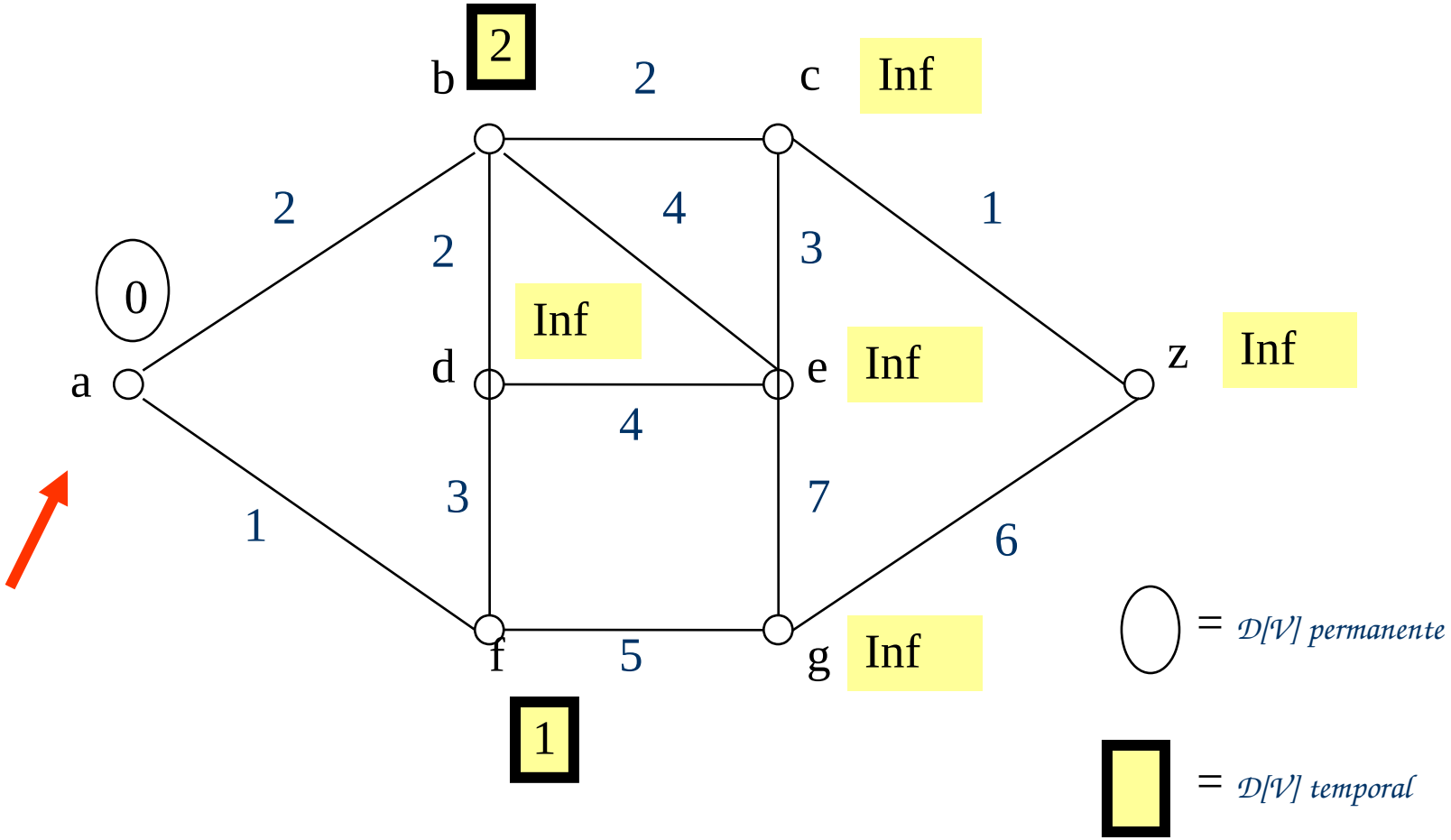
- La implementación sencilla es $O(V^2)$
- La implementación con cola con prioridad es $O(A \log V)$ (ver Brassard)
- La sencilla es preferible para grafos densos
- La de cola con prioridad es mejor para grafos dispersos

Ejemplo: Algoritmo Dijkstra



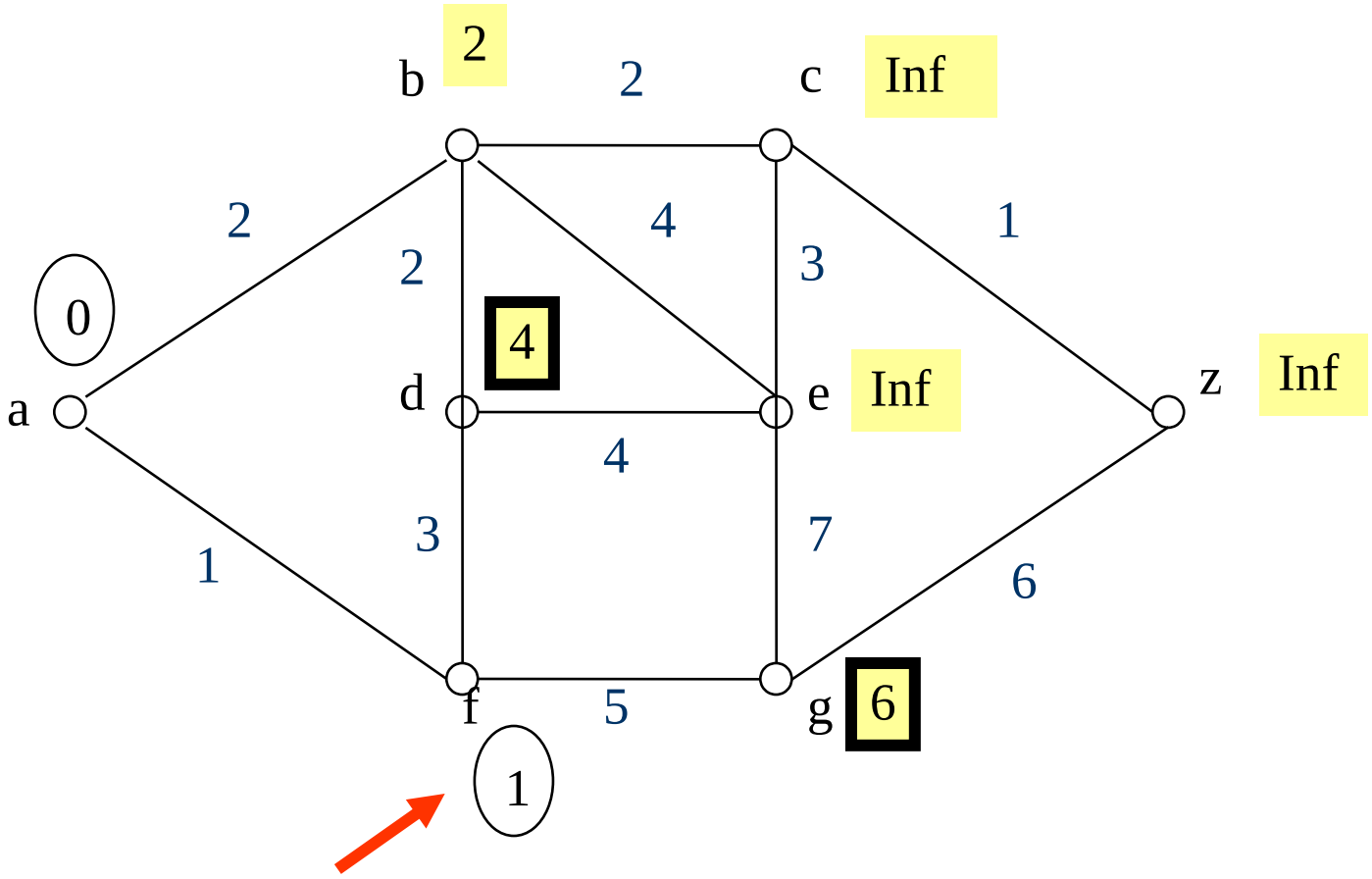
Ejemplo (Solo entre a y z)

Ejemplo: Algoritmo Dijkstra



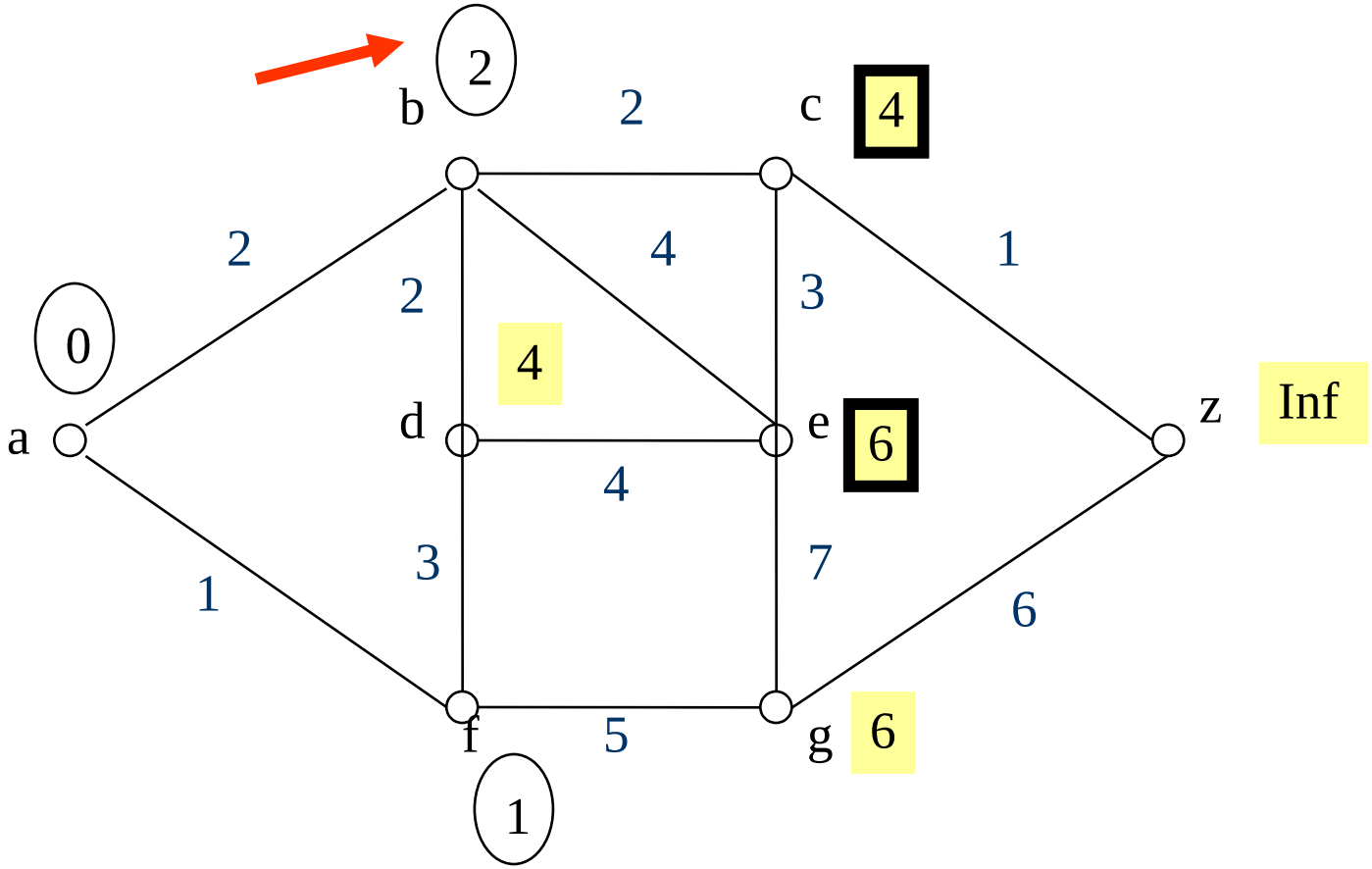
Inicialización

Ejemplo: Algoritmo Dijkstra



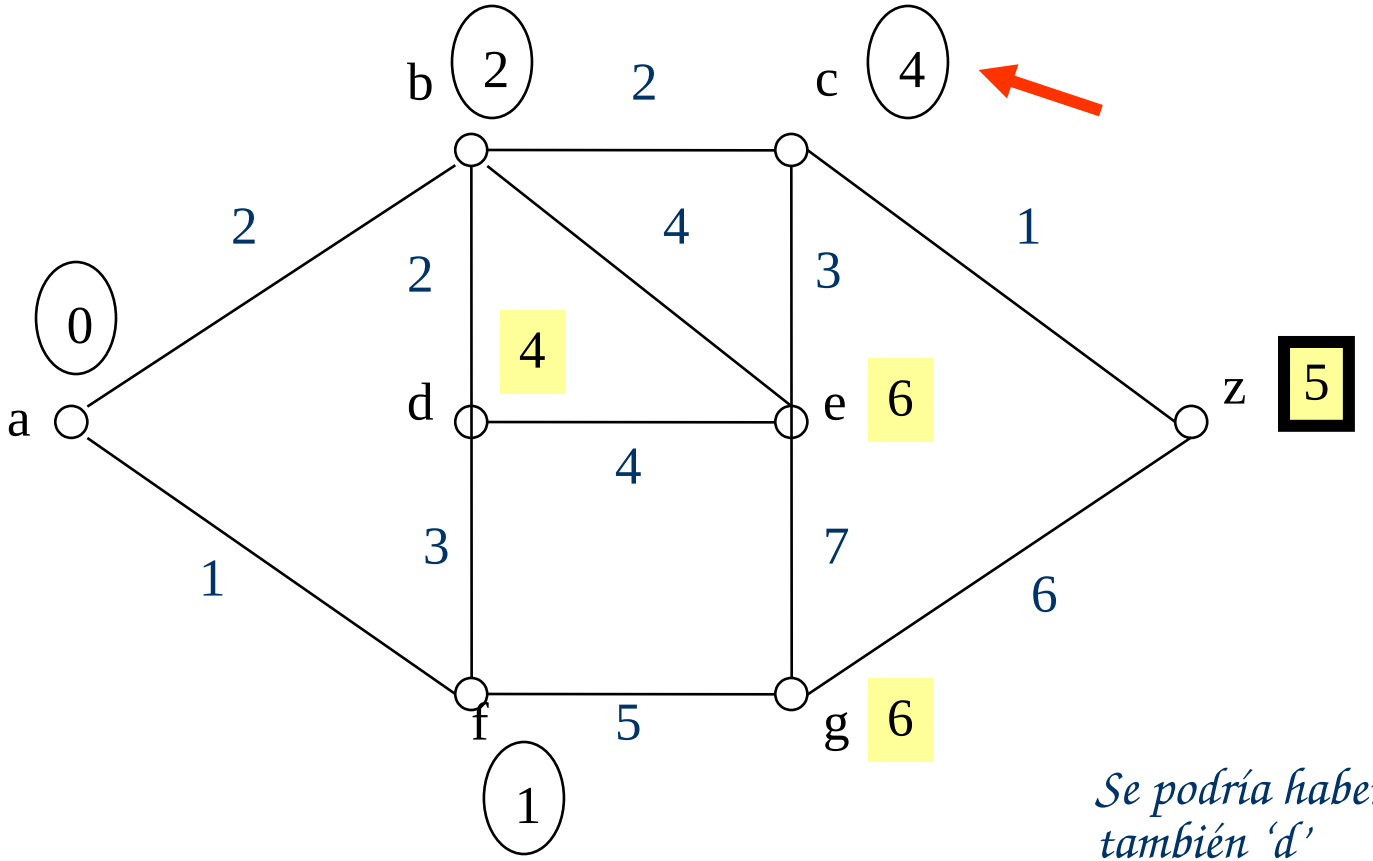
Primera Iteración

Ejemplo: Algoritmo Dijkstra



Segunda Iteración

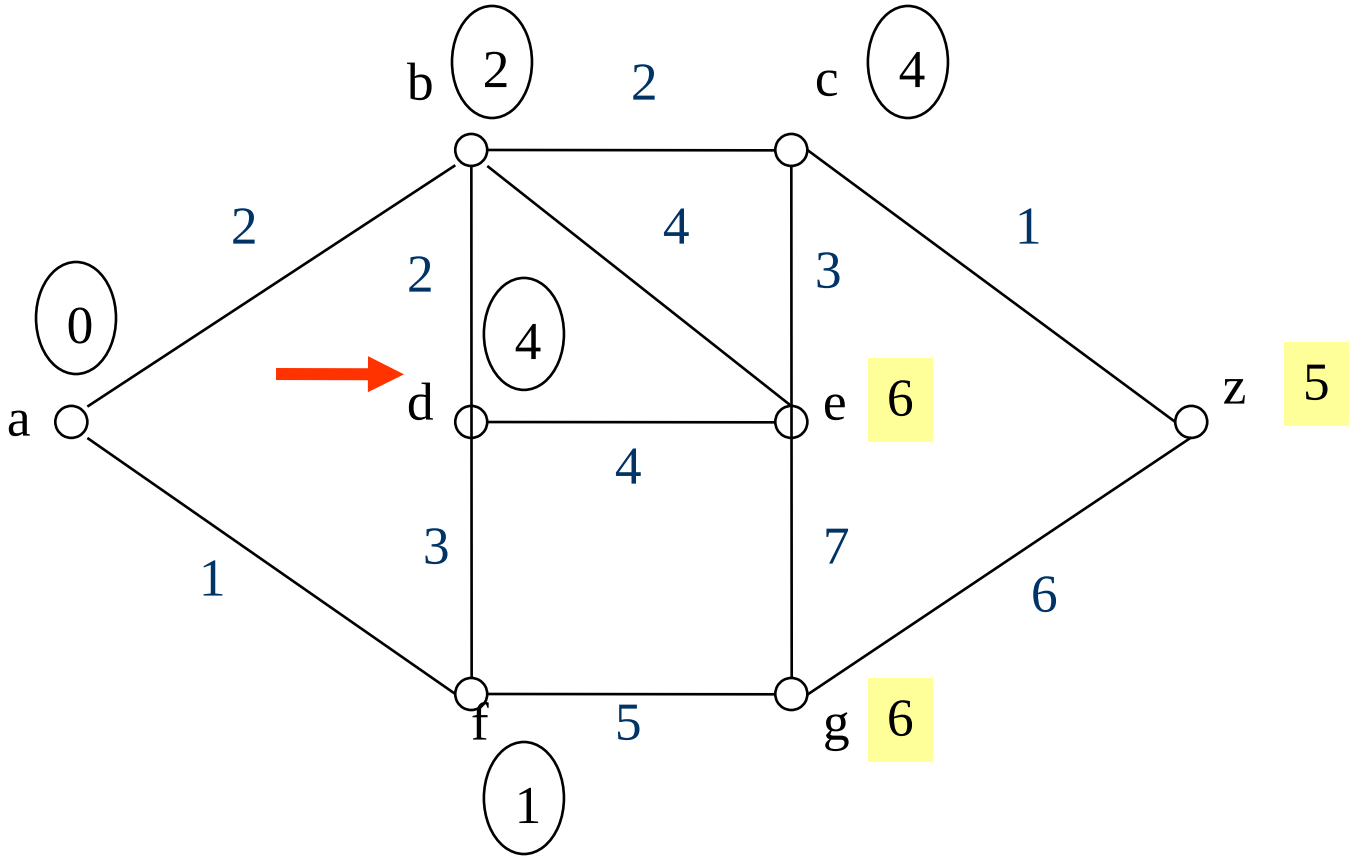
Ejemplo: Algoritmo Dijkstra



Se podría haber elegido también 'd'

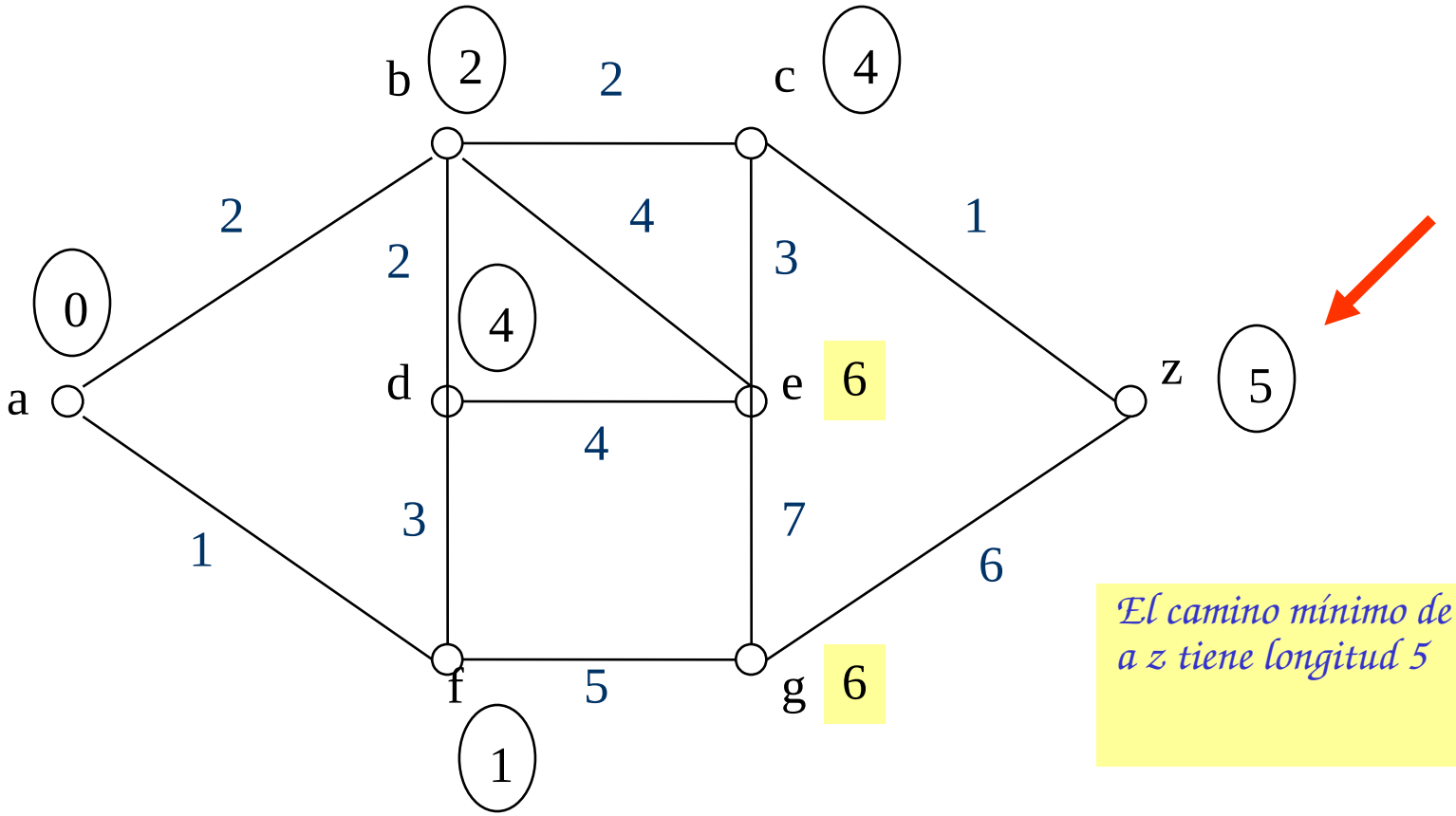
Tercera Iteración

Ejemplo: Algoritmo Dijkstra



Cuarta Iteración

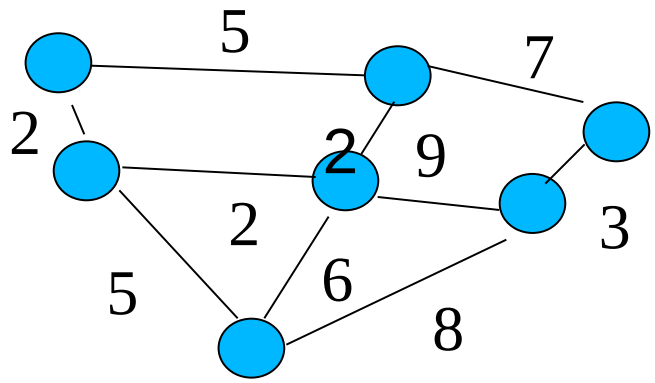
Ejemplo: Algoritmo Dijkstra



El camino mínimo de a a z tiene longitud 5

Quinta (y última) Iteración

Ejemplo: Algoritmo Dijkstra

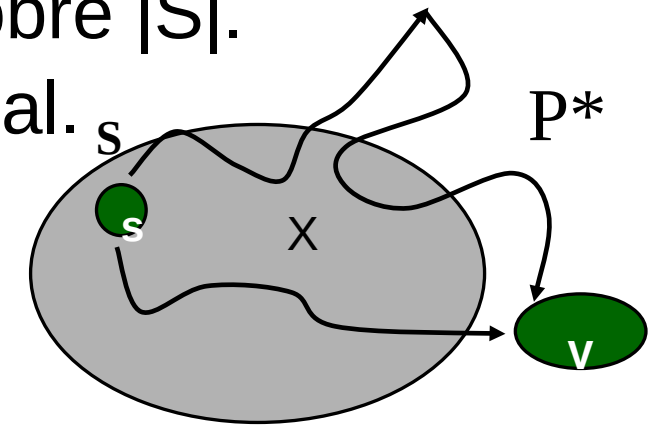


Queda como ejercicio.

Algoritmo Dijkstra: Demostración

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

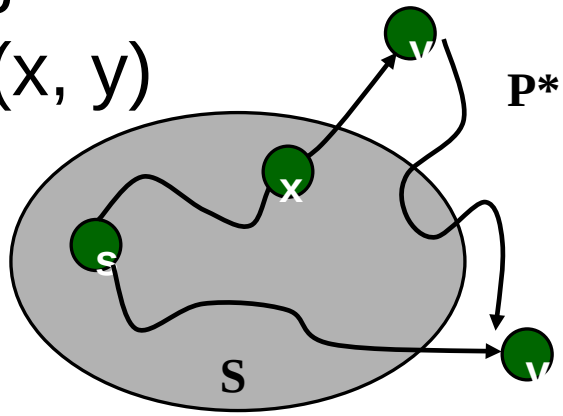
- Demostr. Por inducción sobre |S|.
- Caso base: |S| = 0 es trivial.



- 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

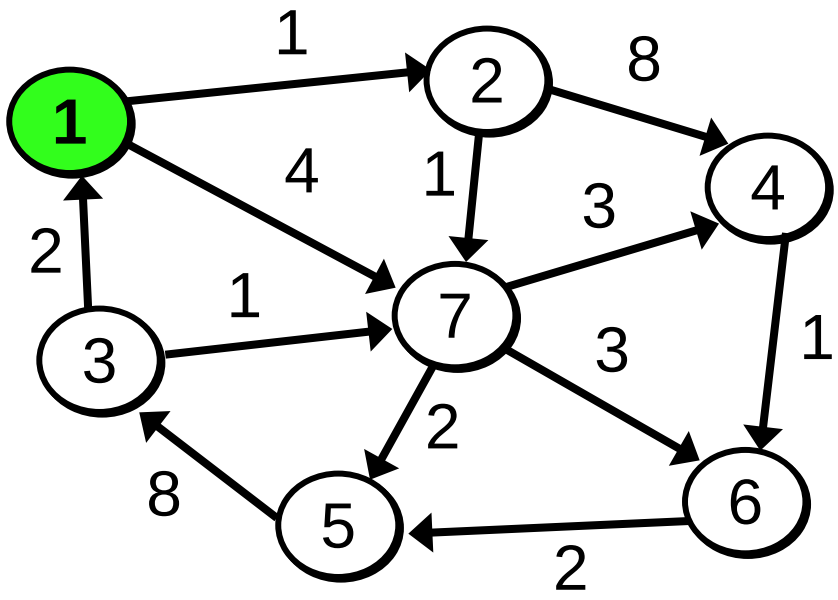
  = d(x) + c(x, y) hipótesis inducción

  ≥ 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.

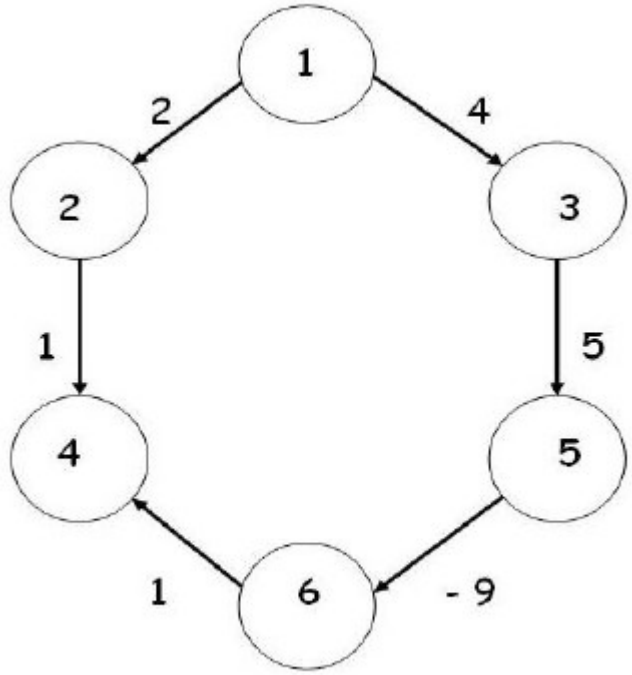


Nodo	S	D	P
2	F	1	1
3	F	∞	1
4	F	∞	1
5	F	∞	1
6	F	∞	1
7	F	4	1

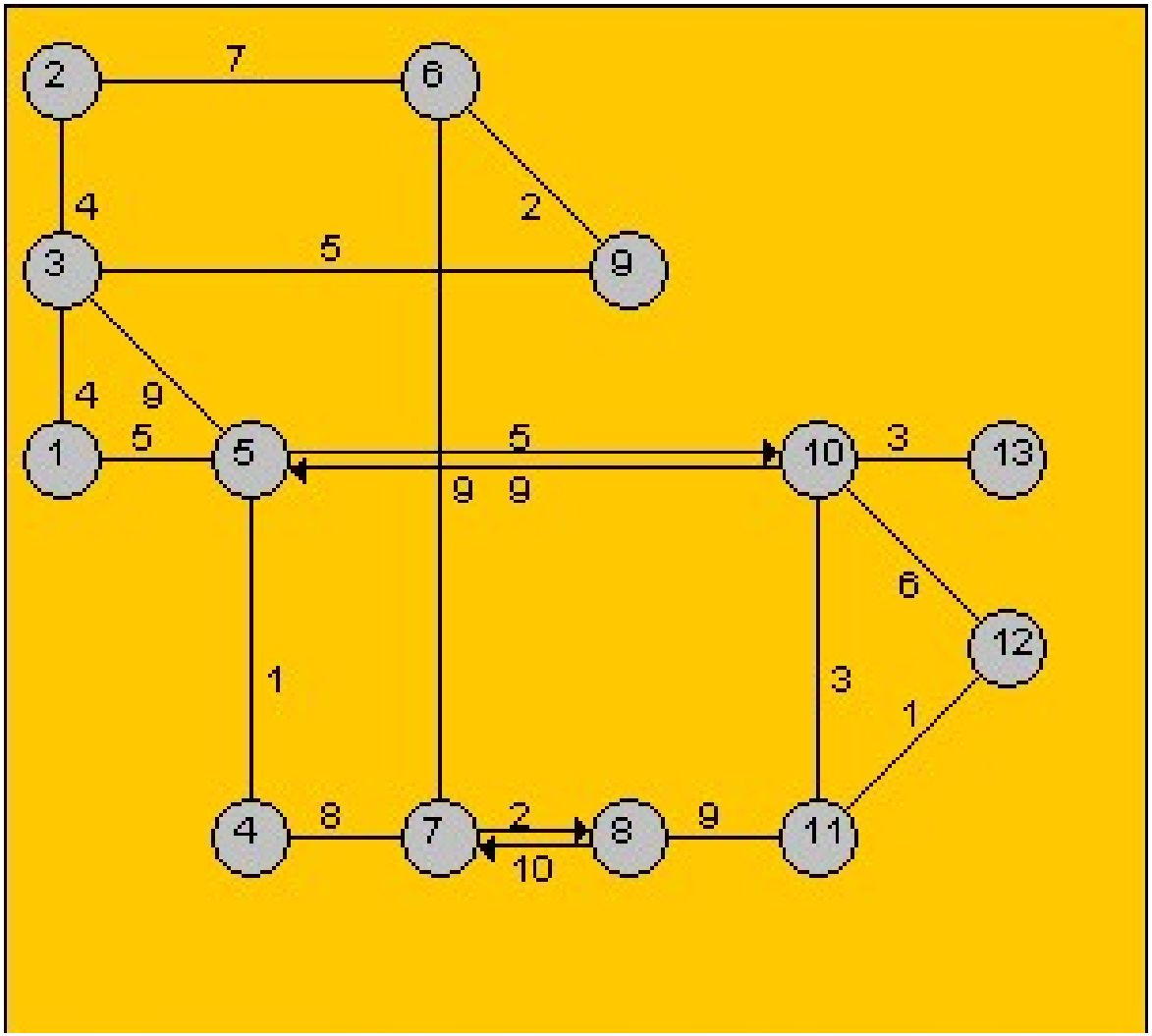
- A partir de las tablas, ¿cómo calcular cuál es el camino mínimo para un nodo **v**?

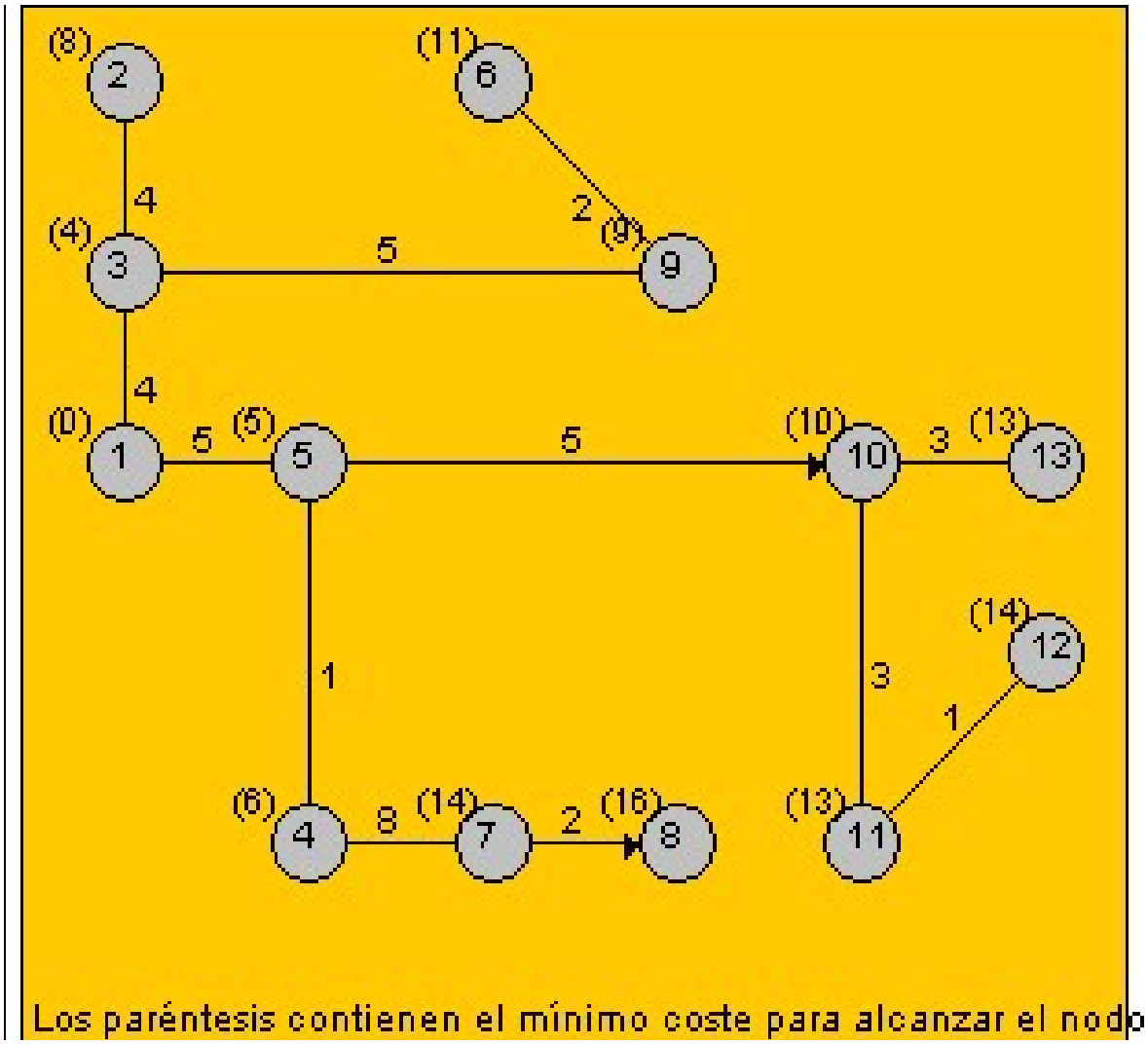
Si hay pesos negativos...

- Ejemplo de grafo donde el algoritmo de Dijkstra no funciona correctamente cuando existen pesos negativos



Dijkstra calcula que el camino mínimo de 1 a 4 es 1-2-4, con distancia 3, cuando realmente el camino mínimo es 1-3-5-6-4 con distancia 1


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

Índice

Heurísticas Greedy

SITUACIÓN QUE NOS PODEMOS ENCONTRAR

- Hay casos en los cuales no se puede conseguir un algoritmo voraz para el que se pueda demostrar que encuentra la solución óptima
- Existen para distintos problemas NP-completos

Heurísticas

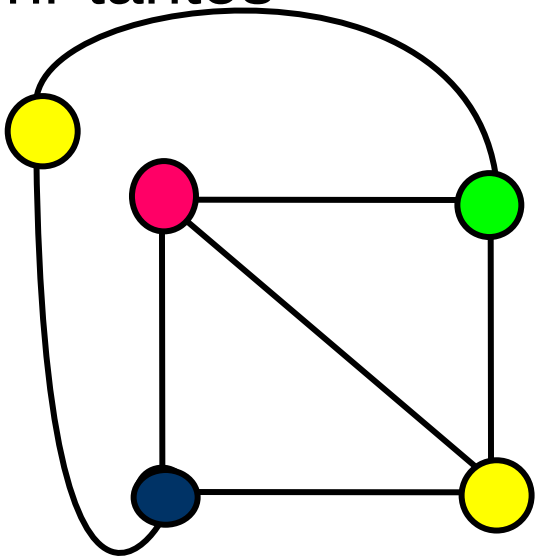
- Heurística: Son procedimientos que, basados en la experiencia, proporcionan buenas soluciones a problemas concretos
- Metaheurísticas de propósito general:
  - Enfriamiento Simulado, Búsqueda Tabu,
  - GRASP (Greedy Randomized Adaptive Search Procedures), Búsqueda Dispersa, Búsqueda por Entornos Variables, Búsqueda Local Guiada,
  - Computación Evolutiva (Algoritmos Genéticos, …), Algoritmos Meméticos, Colonias de Hormigas, Redes de Neuronas

Heurísticas Greedy

- **¿Satisfacer / optimizar?**
- El tiempo efectivo que se tarda en resolver un problema es un factor clave
- Los algoritmos greedy pueden actuar como heurísticas
  - El problema del coloreo de un grafo
  - El problema del Viajante de Comercio
  - El problema de la Mochila
  - ...
- Suelen usarse también para encontrar una primera solución (como inicio de otra heurística)

El Problema del Coloreo de un Grafo

- Planteamiento
  - Dado un grafo plano G=(V, E), determinar el mínimo numero de colores que se necesitan para colorear todos sus vértices, y que no haya dos de ellos adyacentes pintados con el mismo color
- Si el grafo no es plano puede requerir tantos colores como vértices haya
- Las aplicaciones son muchas
  - Representación de mapas
  - Diseño de páginas webs
  - Diseño de carreteras



El Problema del Coloreo de un Grafo

- El problema es NP y por ello se necesitan heurísticas para resolverlo.
- El problema reune todos los requisitos para ser resuelto con un algoritmo greedy.
- Del esquema general greedy se deduce un algoritmo inmediatamente.
- Teorema de Appel-Hanke (1976): Un grafo plano requiere a lo sumo 4 colores para pintar sus nodos de modo que no haya vértices adyacentes con el mismo color.

El Problema del Coloreo de un Grafo

- Suponemos que tenemos una paleta de colores (con más colores que vértices).
- Elegimos un vértice no coloreado y un color. Pintamos ese vértice de ese color.
- Lazo greedy: Seleccionamos un vértice no coloreado v. Si no es adyacente (por medio de una arista) a un vértice ya coloreado con el nuevo color, entonces coloreamos v con el nuevo color.
- Se itera hasta pintar todos los vértices.

Implementacion del algoritmo

Funcion **COLOREO**

{COLOREO pone en NuevoColor los vertices de G que pueden tener el mismo color}

Begin

NuevoColor = ⊘

Para cada vertice no coloreado v de G Hacer

Si v no es adyacente a ningun vertice en NuevoColor Entonces

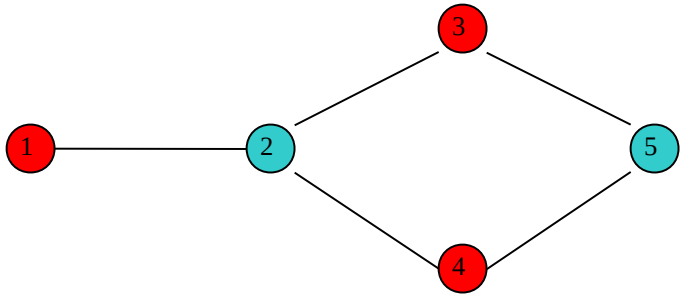
Marcar v como coloreado

Añadir v a NuevoColor

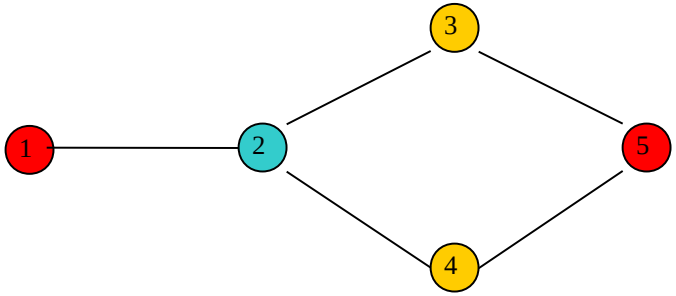
End

Se trata de un algoritmo que funciona en **O(n),** pero que no siempre da la solución óptima.

Ejemplo



El orden en el que se escogen los vértices para colorearlos puede ser decisivo: el algoritmo da la solución optimal en el grafo de arriba, pero no en el de abajo



Ejemplo: Diseño de cruces de semáforos

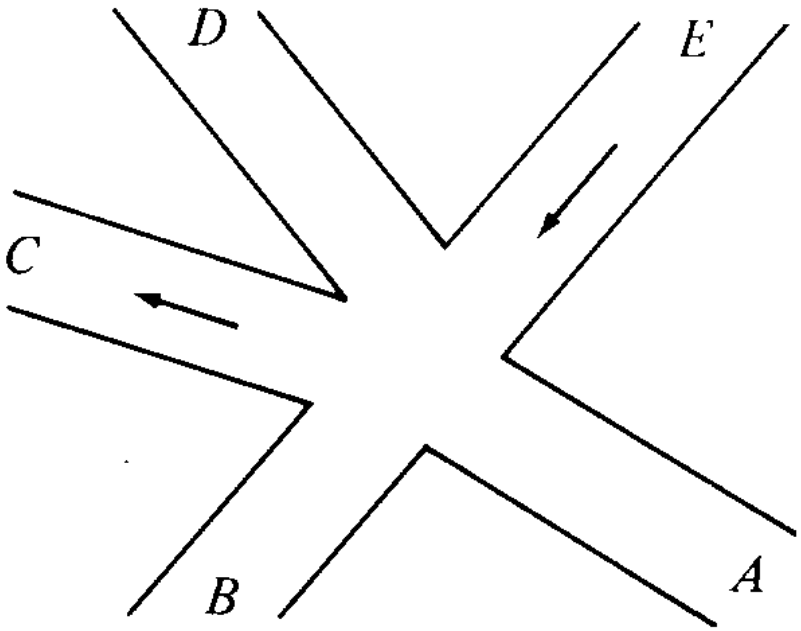
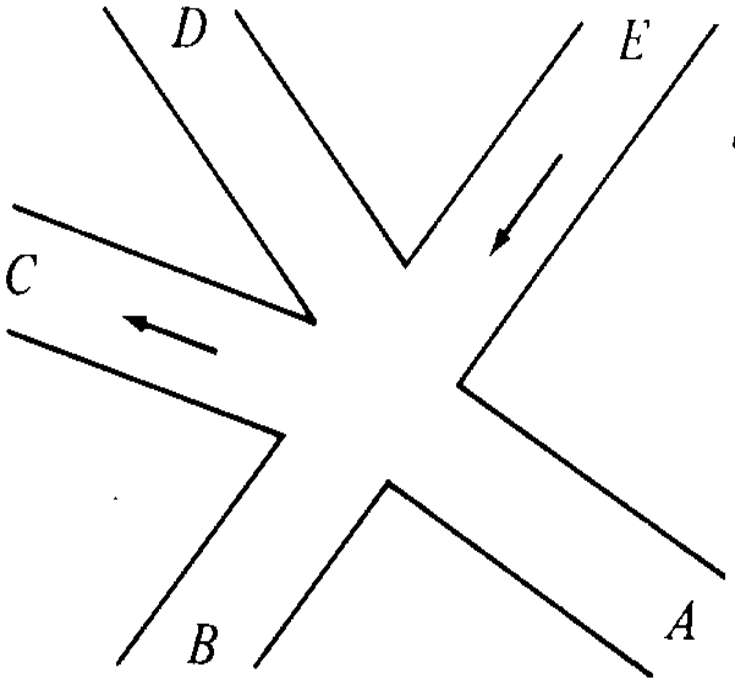


Fig. 1.1. An intersection.

- *A la izquierda tenemos un cruce de calles.*
- *Se señalan los sentidos de circulación.*
- *La falta de flechas significa que podemos ir en las dos direcciones.*
- *Queremos diseñar un patrón de semáforos con el mínimo número de semáforos, lo que*
- *Ahorrara tiempo (de espera) y dinero.*
- *Suponemos un grafo cuyos vértices representan turnos, y cuyas aristas*

unen esos turnos que no pueden realizarse simultáneamente sin que haya colisiones. El problema del cruce con semáforos se convierte en un problema de coloreo de los vértices de un grafo.

Cruce con semáforos: vértices



Los turnos (vértices del grafo) serían 13:

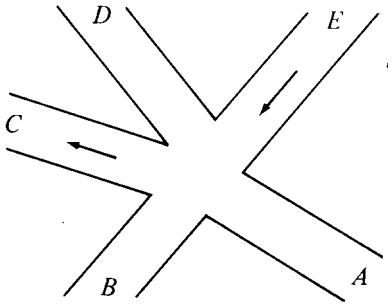
{AB, AC, AD,

BA, BC, BD,

DA, DB, DC,

EA, EB, EC, ED}

Cruce con semáforos: aristas



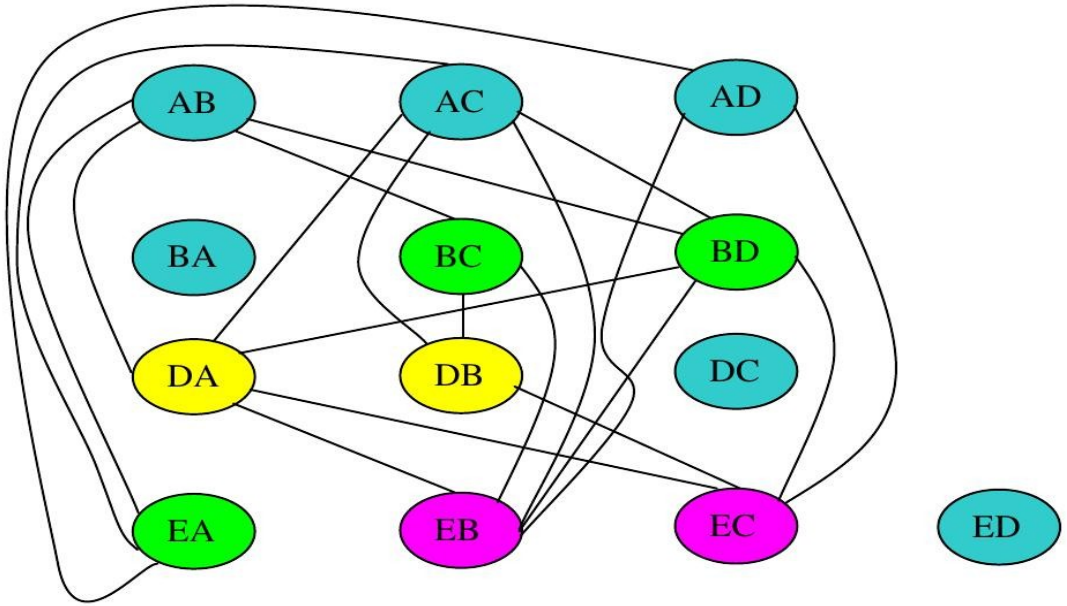
Los turnos (vértices del grafo) serían 13:

{AB, AC, AD,

BA, BC, BD,

DA, DB, DC,

EA, EB, EC, ED}



El Problema del Viajante de Comercio

- Un viajante de comercio que reside en una ciudad, tiene que trazar una ruta que, partiendo de su ciudad, visite todas las ciudades a las que tiene que ir una y sólo una vez, volviendo al origen y con un recorrido mínimo

- Es un problema NP, no existen algoritmos en tiempo polinomial, aunque si los hay exactos que lo resuelven para grafos con 40 vértices aproximadamente.

- Para más de 40, es necesario utilizar heurísticas, ya que el problema se hace intratable en el tiempo.

El Problema del Viajante de Comercio

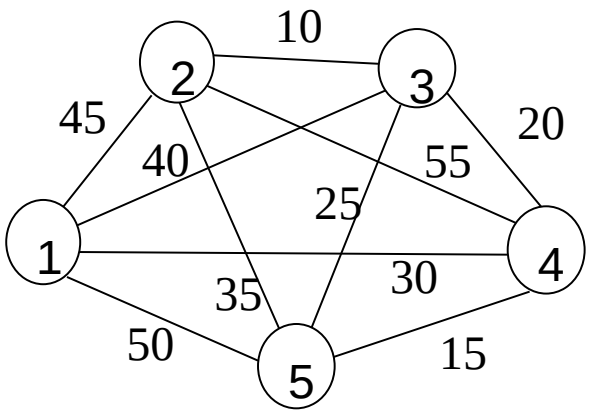
- Supongamos un grafo no dirigido y completo G = (N, A) y L una matriz de distancias no negativas referida a G. Se quiere encontrar un **Circuito Hamiltoniano Minimal**.

- Este es un problema Greedy típico, que presenta las 6 condiciones para poder ser enfocado con un algoritmo greedy.
- Puede plantearse con nodos como candidatos o con aristas como candidatos.

- Destaca de esas 6 características **la condición de factibilidad** (para el caso de aristas):
  - que al seleccionar una arista no se formen ciclos,
  - que las aristas que se escojan cumplan la condición de no ser incidentes en tercera posición al nodo escogido.

El Problema del Viajante de Comercio

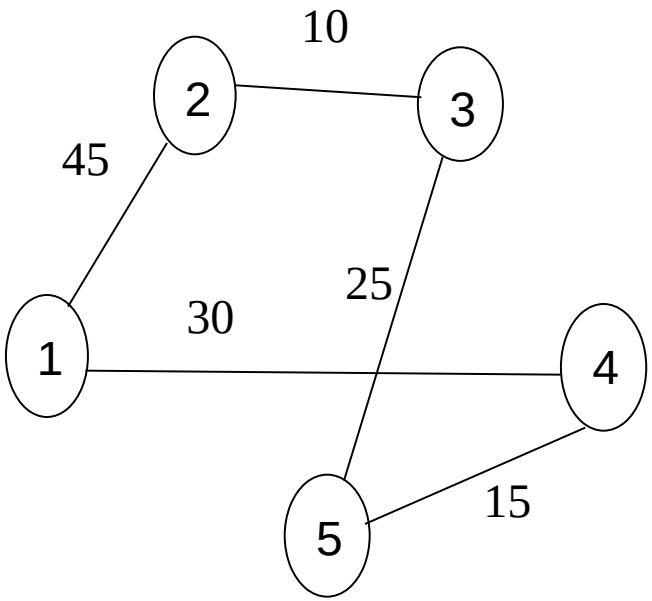
- Consideremos el siguiente grafo



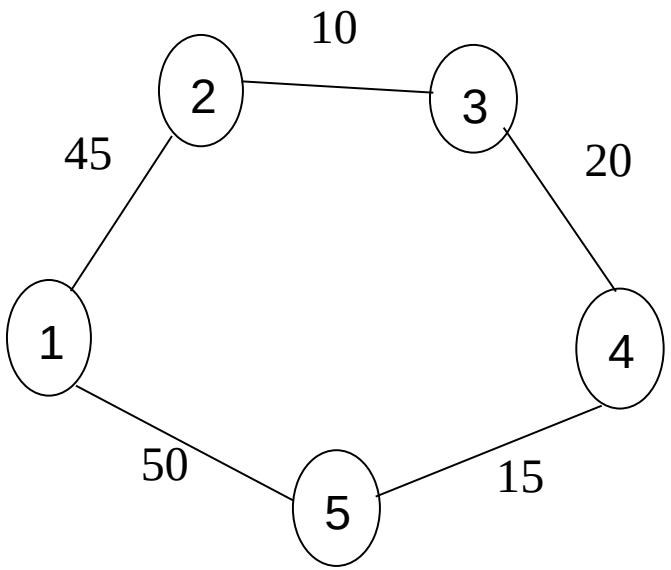
- Posibilidades:
  - **Los nodos son los candidatos. Empezar en un nodo cualquiera y en cada paso moverse al nodo no visitado más próximo al último nodo seleccionado.**
  - **Las aristas son los candidatos. Hacer igual que en el algoritmo de Kruskal, pero garantizando que se forme un ciclo al final del proceso. Y de cada nodo no salgan más de dos aristas.**

El Problema del Viajante de Comercio

- Solución con la primera heurística
- Solución empezando en el nodo 1 — Solución empezando en el nodo 5



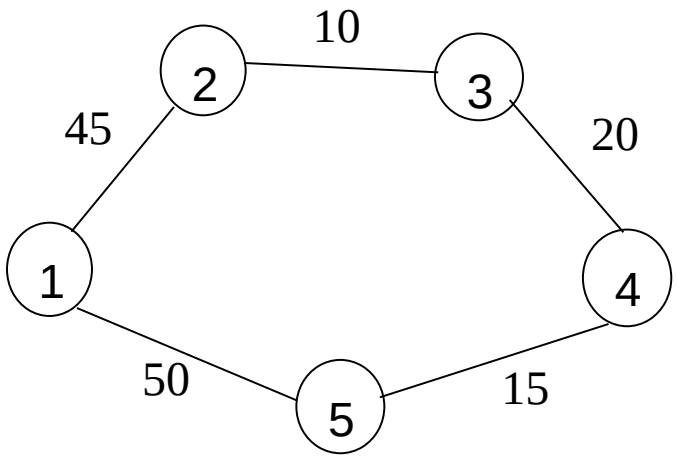
Solución: (1, 4, 5 ,3, 2), ***125***



Solución: (5, 4, 3, 2, 1), ***140***

El Problema del Viajante de Comercio

- Solución con la segunda heurística



- Solución: ((2, 3), (4, 5), (3, 4), (1, 2), (1, 5))

  Coste = 10+15+20+45+50 = 140

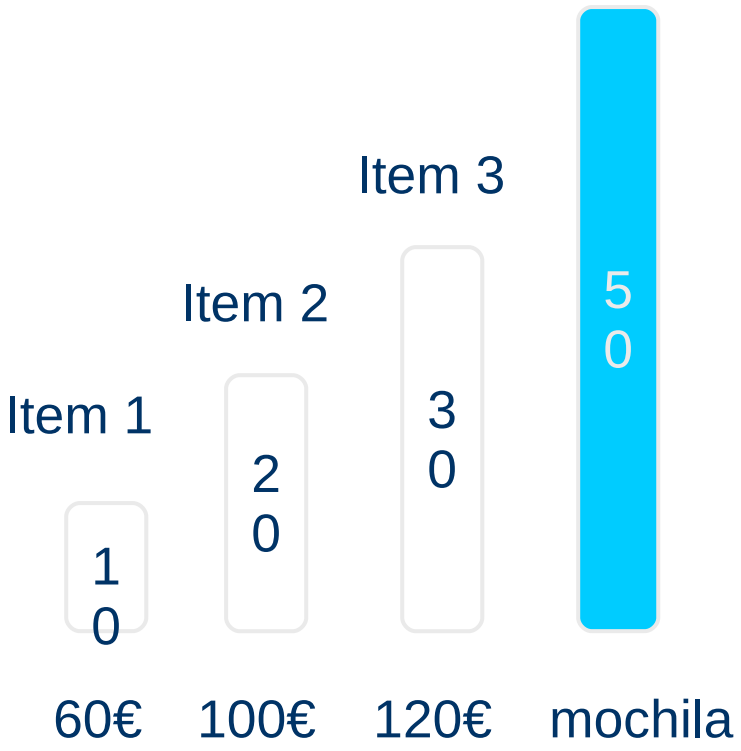
- En todos los casos la eficiencia es la del algoritmo de ordenación que se use

El problema de la Mochila

- Tenemos n objetos y una mochila. El objeto i tiene un peso w_i y la mochila tiene una capacidad M.
- Si metemos en la mochila el objeto i, generamos un beneficio de valor p_i
- El objetivo es rellenar la mochila de tal manera que se maximice el beneficio que producen los objetos que se transportan, con la limitación de la capacidad de valor M.

- Ya hemos visto que si los objetos se pueden fraccionar, el algoritmo greedy da la solución óptima.
- Pero si los objetos no se pueden fraccionar la estrategia greedy no es óptima, es solo una heurística.

Ejemplo: Mochila 0/1



En este ejemplo la solución óptima es incluir los items 2 y 3, con valor 220.

La estrategia greedy elegiría primero el item 1 (densidad 6) y luego el item 2 (densidad 5). El item 3 (densidad 4) ya no cabe. Valor: 160.

La asignación de tareas

- Supongamos que disponemos de n trabajadores y n tareas. Sea $b_{ij} > 0$ el coste de asignarle el trabajo j al trabajador i.
- Una asignación de tareas puede ser expresada como una asignación de los valores 0 ó 1 a las variables x_{ij}, donde
  - $x_{ij} = 0$ significa que al trabajador i no le han asignado la tarea j, y
  - $x_{ij} = 1$ indica que sí.
- Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador.

La asignación de tareas

- Dada una asignación válida, definimos el coste de dicha asignación como:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij} b_{ij}$$

- Diremos que una asignación es óptima si es de mínimo coste.

- Cara a diseñar un algoritmo ávido para resolver este problema podemos pensar en dos estrategias distintas: asignar cada trabajador a la mejor tarea posible, o bien asignar cada tarea al mejor trabajador disponible.
- Sin embargo, ninguna de las dos estrategias tiene por qué encontrar siempre soluciones óptimas. ¿Es alguna mejor que la otra?

La asignación de tareas

		Tarea		
		1	2	3
Trabajador	1	16	11	17
	2	20	15	1
	3	18	17	20

- La estrategia greedy para trabajadores le asignaría la tarea 2 al trabajador 1, la tarea 3 al trabajador 2, y la tarea 1 al trabajador 3, con un coste de 11+1+18=30. En este caso ese es el óptimo.
- La estrategia greedy para tareas asignaría el trabaj. 1 a la tarea 1, el trabaj. 2 a la tarea 2 y el trabaj. 3 a la tarea 3, con un coste de 16+15+20=51.

La asignación de tareas

- Esas estrategias dependen del orden en que se procesan los trabajadores o las tareas.
- Si en vez de usar el orden 123 como antes usamos el 321 obtenemos:
- Para la estrategia para trabajadores: tarea 2 a trabaj. 3, tarea 3 a trabaj. 2 y tarea 1 a trabaj. 1, con coste 17+1+16=34.
- Para la estrategia para tareas: trabaj. 2 a tarea 3, trabaj. 1 a tarea 2 y trabaj. 3 a tarea 1, con coste 1+11+18=30

La asignación de tareas

- Otra estrategia (que no depende de ningún orden) es elegir en cada paso el par trabajador-tarea con menor coste, de entre los compatibles con las decisiones ya tomadas.
- En el ejemplo esa estrategia escogería los pares (trabajador,tarea) (2,3), (1,2) y (3,1), con costo 1+11+18=30, que en este caso es la solución óptima.
- Esta estrategia tampoco es óptima en general.

Asignación de tareas: no optimalidad

Cada uno de cuatro laboratorios, A,B,C y D tienen que ser equipados con uno de cuatro equipos informáticos. El costo de instalación de cada equipo en cada laboratorio lo da la tabla. Queremos encontrar la asignación menos costosa.

	1	2	3	4
A	48	48	50	44
B	56	60	60	68
C	96	94	90	85
D	42	44	54	46

No optimalidad del algoritmo greedy para asig. tareas

	1	2	3	4
A	48	48	50	44
B	56	60	60	68
C	96	94	90	85
D	42	44	54	46

La estrategia greedy asignaría el equipo 1 al laboratorio D (42), el equipo 4 al laboratorio A (44), el equipo 2 (o el 3) al laboratorio B (60), y el equipo 3 al laboratorio C (90), con un costo de 236.

Pero la estrategia D-2 (44), A-4 (44), B-1 (56), C-3 (90) tiene un costo menor, 234.

Algoritmica

Tema 1. La Eficiencia de los Algoritmos

Tema 2. Algoritmos “Divide y Vencerás”

Tema 3. Algoritmos Voraces (“Greedy”)

Tema 4. Algoritmos para la Exploración de Grafos (“Backtraking”, “Branch and Bound”)

Tema 5. Algoritmos basados en Programación Dinámica

Tema 6. Otras Técnicas Algorítmicas de Resolución de Problemas