Algorítmica

Tema 1. Planteamiento General

Tema 2. La Eficiencia de los Algoritmos

Tema 3. Algoritmos “Divide y Vencerás”

Tema 4. Algoritmos Voraces (“Greedy”)

Tema 5. Algoritmos para la Exploración de Grafos (“Backtraking”, “Branch and Bound”)

Tema 6. Algoritmos basados en Programación Dinámica

Tema 7. Otras Técnicas Algorítmicas de Resolución de Problemas

Objetivos

- Comprender la filosofía de diseño de 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

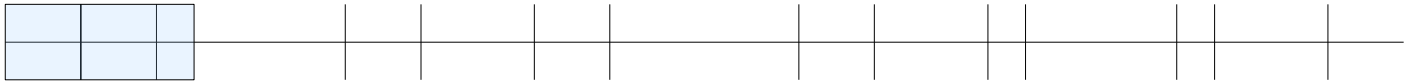
Algoritmos Greedy (voraz)

- 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,

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)

¡Comete siempre todo lo que tengas a mano!

En este problema

Avanza lo más que puedas antes de rellenar el depósito.

Selección gasolinera 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 que está disponible en cada momento.
- Una vez tomada la decisión no vuelve a replantearse en el futuro.
- Suelen ser rápidos y fáciles de implementar.
- No garantizan alcanzar la solución óptima.

Elementos de un algoritmo voraz

1. *Función de Factibilidad*: determina si es posible completar el conjunto de candidatos seleccionas para alcanzar una solución al problema.
2. *Función Selección*: determina el mejor candidatos del conjunto a seleccionar.
3. *Función Objetivo*: da el valor de la solución alcanzada.
4. *Función Solución:* determina si se ha alcanzado una solución
5. Conjunto de Candidatos (C) : representa al conjunto de posibles decisiones que se pueden tomar en cada momento.
6. Conjunto de Seleccionados (S): representa al conjunto de decisiones tomadas hata este momento.

Selección gasolinera 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: $S = \varnothing$
- 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 mas es considerado
- Evaluamos la función objetivo. Si no estamos en el optimo seleccionamos con la función de selección otro candidato y repetimos el proceso anterior hasta alcanzar la solución.

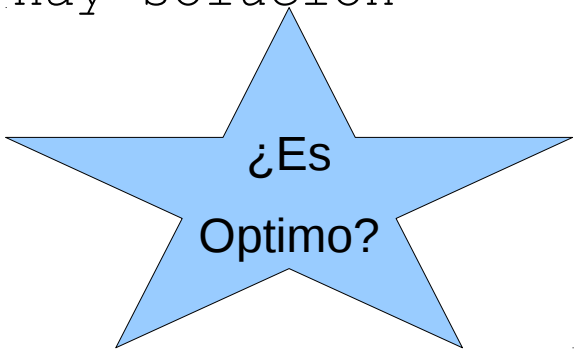
Esquema de un algoritmo voraz

```
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

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+1) < f(r+2) < ...< fq es 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 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 candidatos 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 $\Rightarrow$ Solución global optimal

- Paso 1.
  - Demostrar que la primera decisión es correcta.
- Paso 2.
  - Demostrar que existen subestructuras optimales.
    - Es decir, cuando la 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 tiempo

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

- 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 existe una solución optimal B que no contiene $\{1\}$, siempre podremos reemplazar la primera actividad en B por $\{1\}$ (xq?). Obteniendo el mismo numero de actividades, y por tanto 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 S.,

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 =>
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*
  - *– Mayor, menor duración.*
  - *– Menor solapamiento.*
  - *– Termina antes..*
- *Función de Factibilidad: x es factible si es compatible con las actividades en A.*
- *Función Objetivo: Tamaño de A.*

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 despues de que la actividad previa termine.
  - *Hasta que S este vacio.;*

```
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)

Ejemplo: Almacenamiento Optimal en Cintas

- Almacenar n programas en una cinta de longitud L.
- Cada programa i tiene una longitud l_i , $1 \leq i \leq n$
- Todos los programas se recuperan del mismo modo, siendo el tiempo medio de recuperación (TMR),

$$1/n \sum_{i=1}^{n} \sum_{k=1}^{i} l_k$$

- Nos piden encontrar una permutación de los n programas tal que cuando esten almacenados en la cinta el TMR sea mínimo.
- Minimizar el TMR es equivalente a minimizar

$$D(I) = 1/n \sum_{i=1}^{n} \sum_{k=1}^{i} l_k$$

Ejemplo: Almacenamiento Optimal en Cintas

El problema se puede resolver mediante la técnica greedy

Sea n = 3 y (l1,l2,l3) = (5,10,3)

Orden I	D(I)
1,2,3	5 + 5 + 10 + 5 + 10 + 3 = 38
1,3,2	5 + 5 + 3 + 5 + 3 + 10 = 31
2,1,3	10 + 10 + 5 + 10 + 5 + 3 = 43
2,3,1	10 + 10 + 3 + 10 + 3 + 5 = 41
3,1,2	3 + 3 + 5 + 3 + 5 + 10 = 29
3,2,1	3 + 3 + 10 + 3 + 10 + 5 = 34

Ejemplo: Almacenamiento Optimal en Cintas

Partiendo de la cinta vacia

for i := 1 to n do

grabar el siguiente programa mas corto

ponerlo a continuacion en la cinta

El algoritmo escoge lo más inmediato y mejor sin tener en cuenta si esa decisión será la mejor a largo plazo.

Ejemplo: Almacenamiento Optimal en Cintas

Teorema

Si l1 ≤ l2 ≤ … ≤ ln entonces el orden de colocación $i_j = j$, $1 \leq j \leq n$ minimiza

$$\sum_{i=1}^{n} \sum_{k=1}^{i} l_k$$

para todas las posibles permutaciones de i_j

- ***Optimalidad?***

Ver la demostración en Horowitz-Sahni

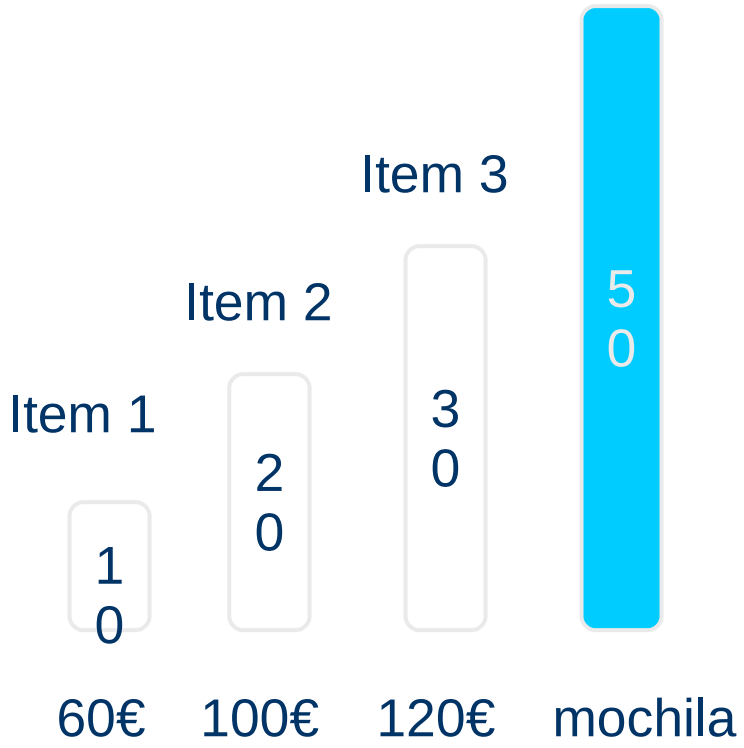
Ejemplo: Problema de la mochila fraccional

- 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

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

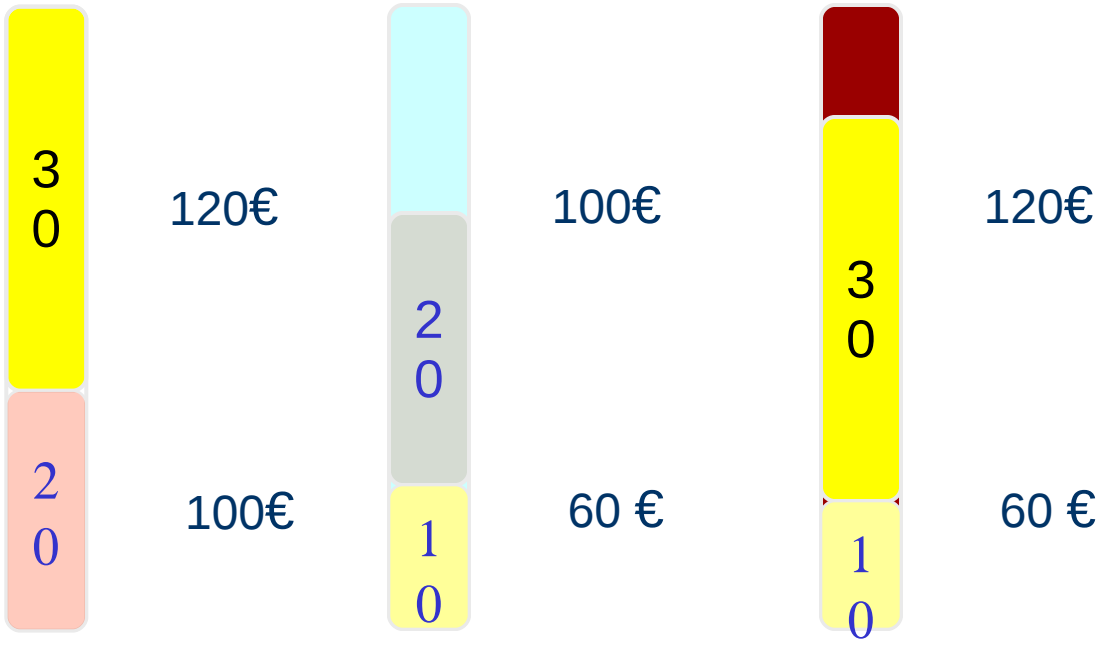
$$\max \sum_{i=1}^{n} x_i b_i$$
$$s.a. \sum_{i=1}^{n} x_i p_i \leq P$$

Ejemplo: Mochila 0/1



- Es un claro problema de tipo greedy
- Sus aplicaciones son inumerables
- La tecnica greedy produce soluciones optimales para este tipo de problemas cuando se permite fraccionar los objetos

Ejemplo: Mochila 0/1



Total = 220 € = 160 € = 180 €

¿Cómo seleccionamos los items?

Ejemplo: Problema de la mochila fraccional

Ejemplo

p_i	10	20	30	40	50
b_i	20	30	66	40	60

n= 5, P=100

Opciones: ¿Poner primero los más pesados? ¿Primero los que tienen más valor?

QUEDA COMO EJERCICIO

Solucion 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 fraccion del objeto en curso, hasta completar la capacidad, y se desprecia el resto.
- Se ordenan los objetos por densidad no creciente, i.e.:

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

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

- **Metodo 1 elegir primero el menos pesado**
  - Peso total = 1*10 + 1*20 + 1*30 + 1*40 = 100
  - Beneficio total = 20 + 30 + 65 + 40 = 155
- **Metodo 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 (razon 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*40 = 100
  - Benef. Total = 65 + 20 + 30 + 0.8*48 = 163

¿Qué ocurre si tenemos 45/45 en lugar de 40/40?

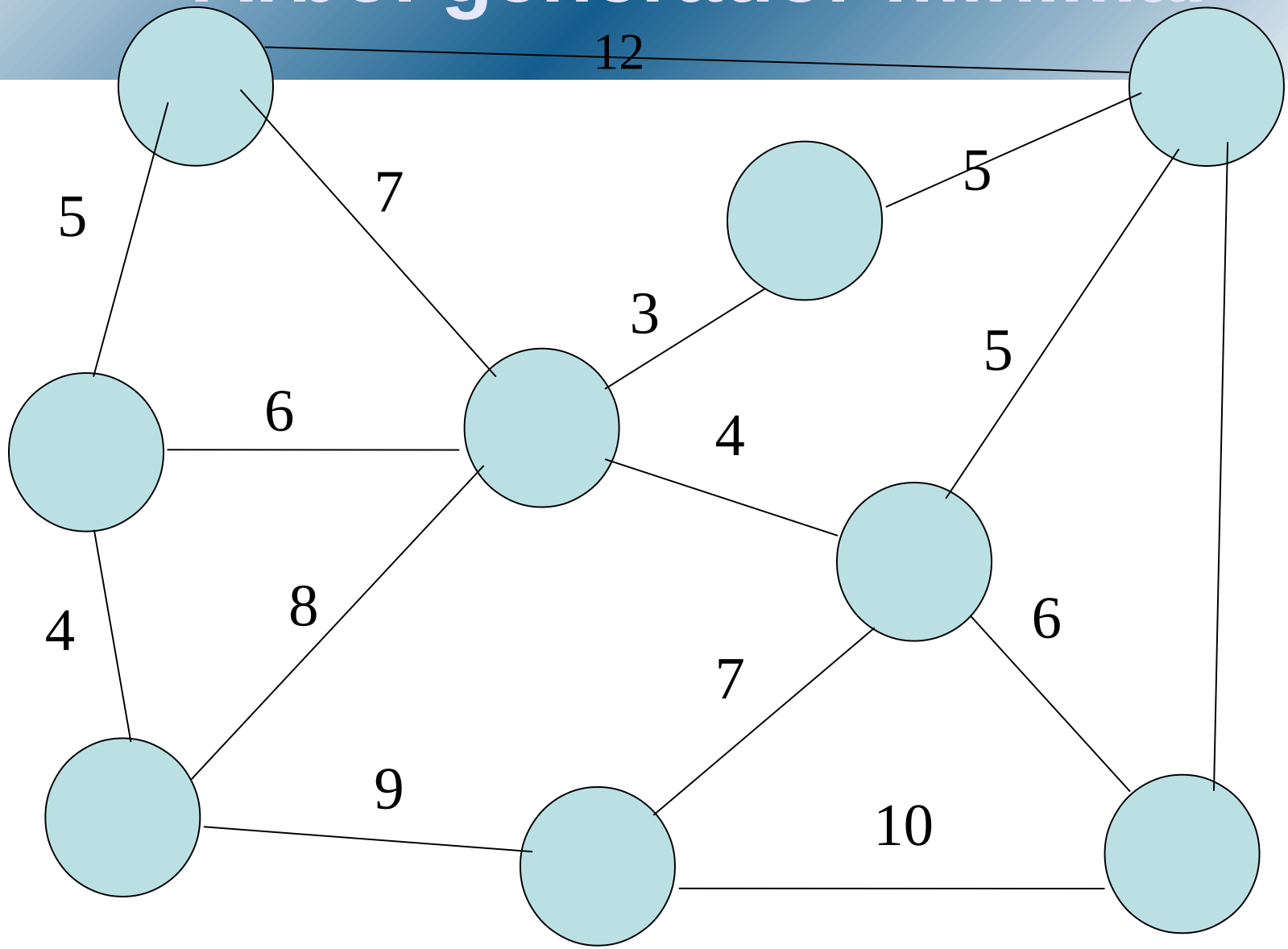
Índice

- 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

Arbol 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 o árbol de recubrimiento mínimo (ARM) (en inglés, minimum spanning tree)
- Aplicaciones:
  - Red de comunicaciones de mínimo coste
  - Refuerzo de líneas críticas con mínimo coste
- Dos enfoques para la solución:
  - Basado en aristas: algoritmo de Kruskal
  - Basado en vértices: algoritmo de Prim

Arbol generador minimal



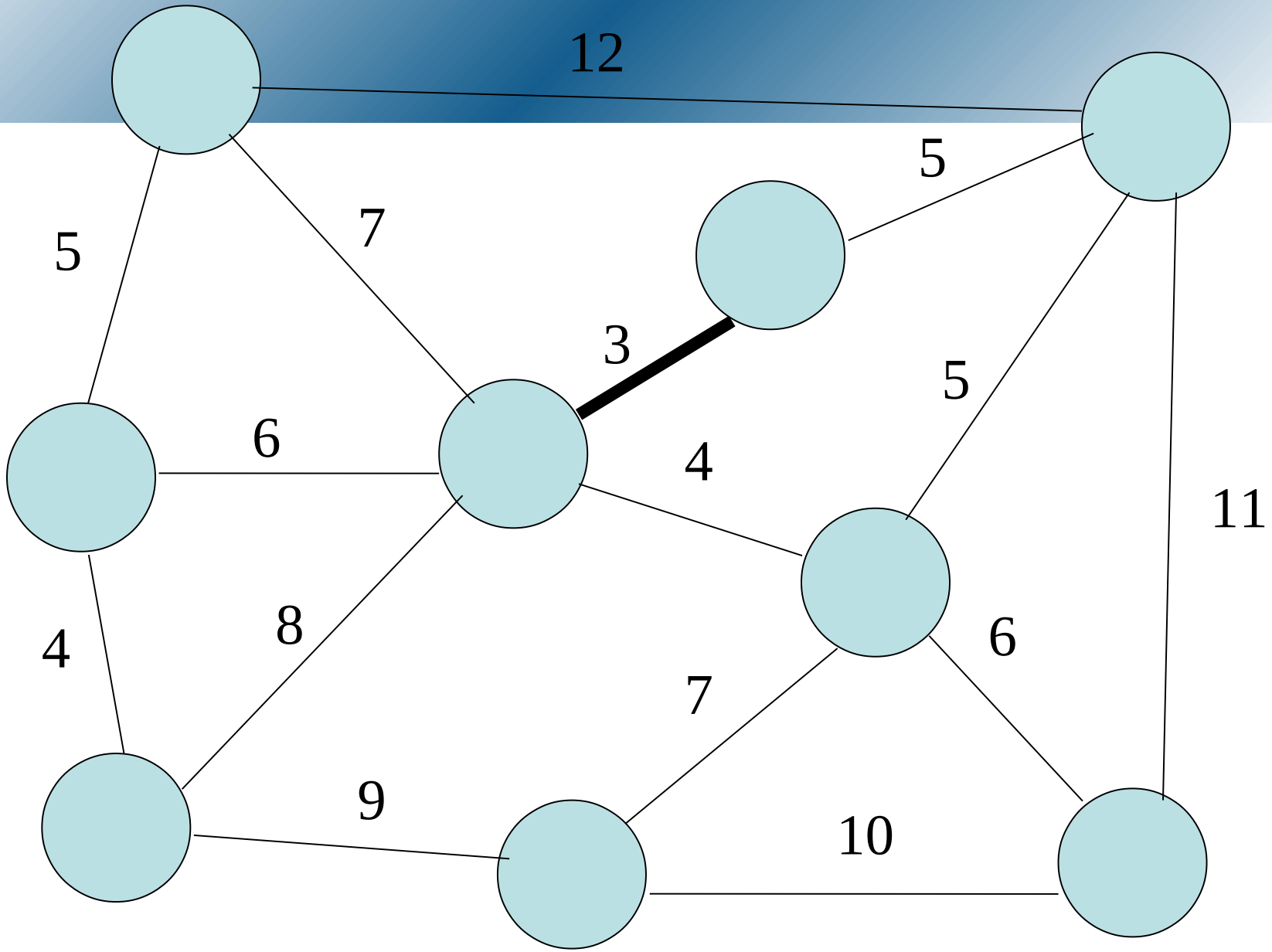
Algoritmo de Kruskal

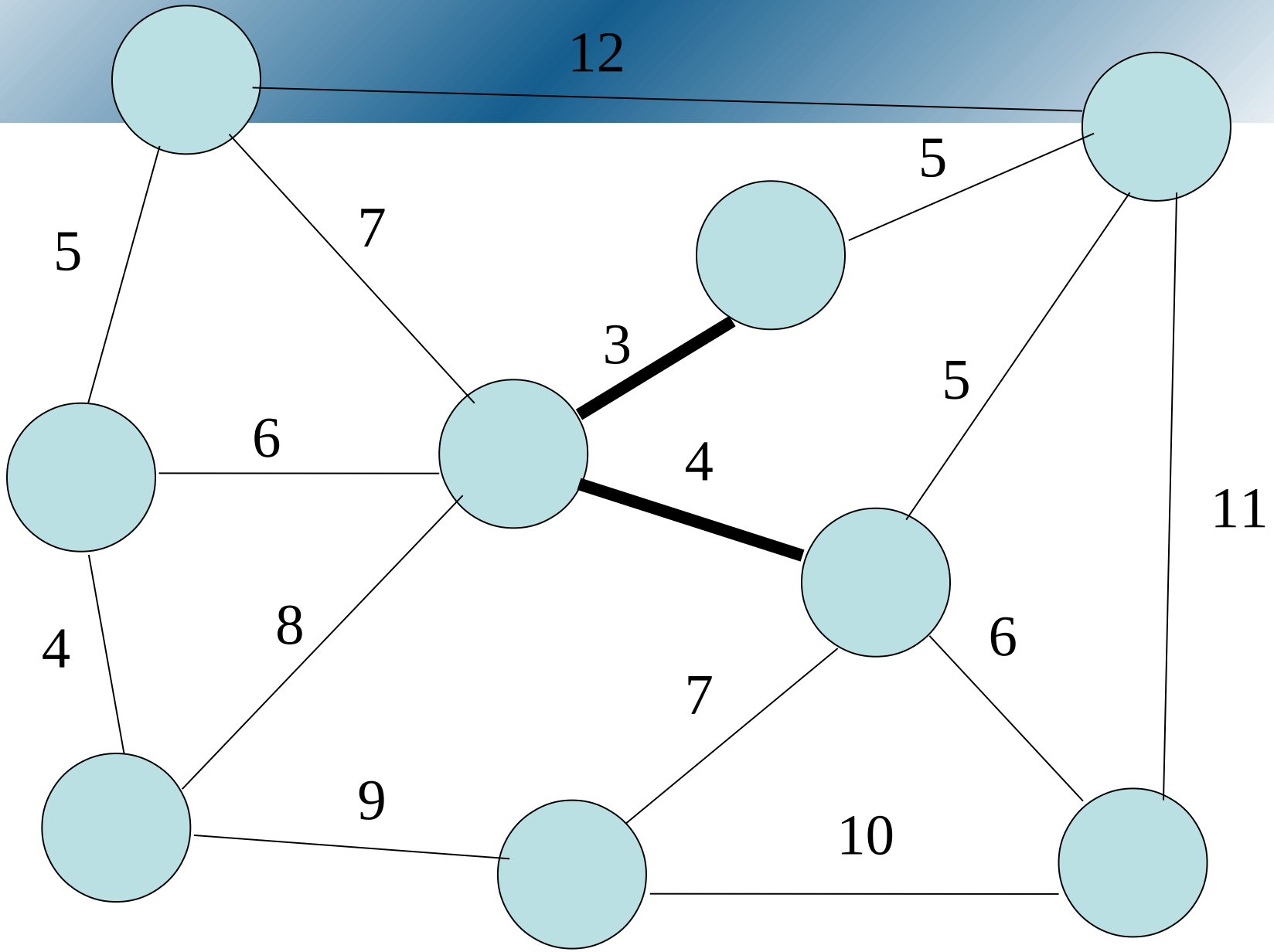
- Conjunto de candidatos: aristas
- Función Solución: un conjunto de aristas que conecta todos los vértices

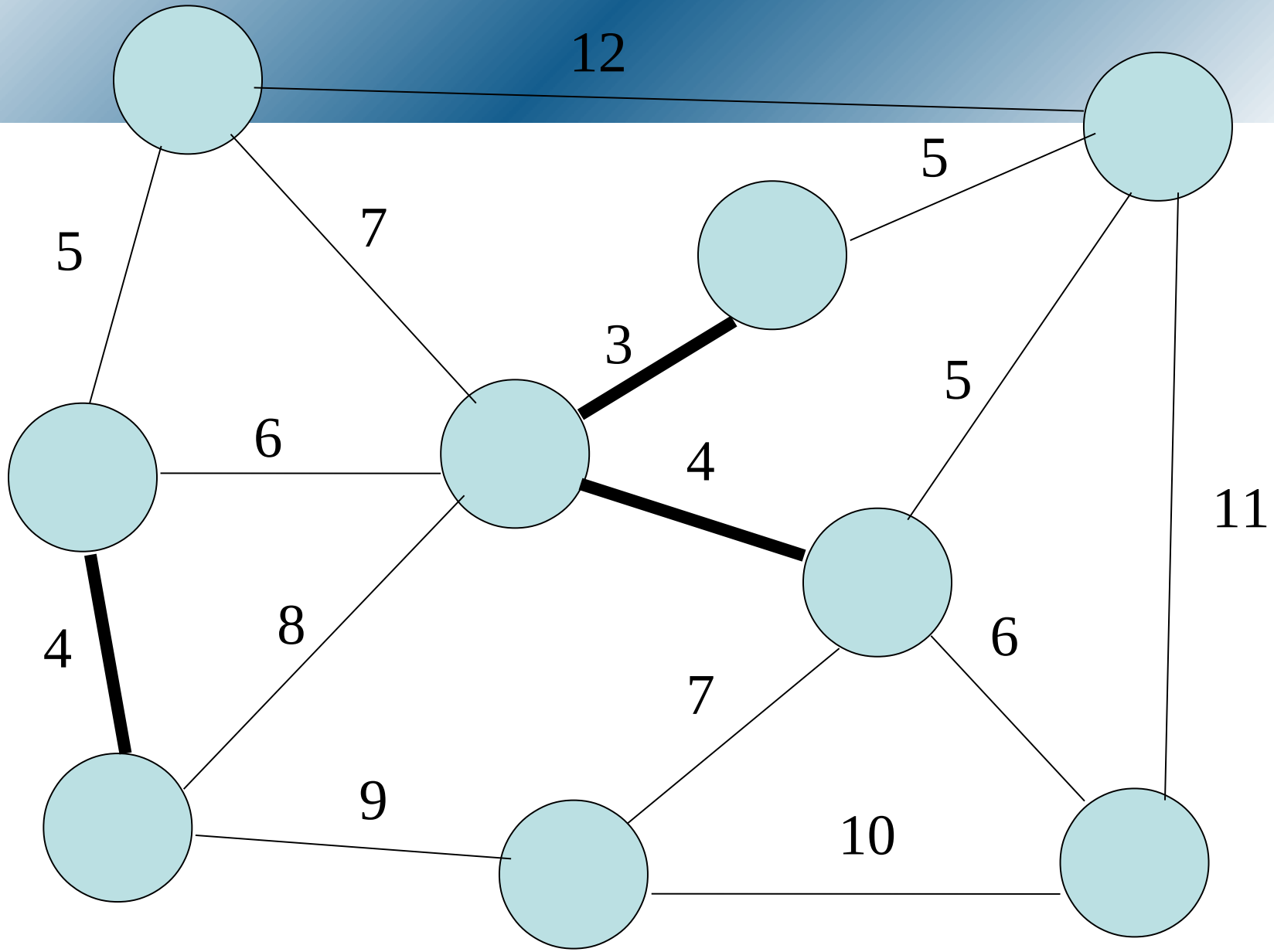
  Se ha construido un árbol de recubrimiento (n-1 aristas seleccionadas).
- Función factible: el conjunto de aristas no forma ciclos
- Función selección: la arista de menor coste
- Función objetivo: minimizar la suma de los costes de las aristas
- *Conjunto prometedor*: se puede extender para producir una solución óptima

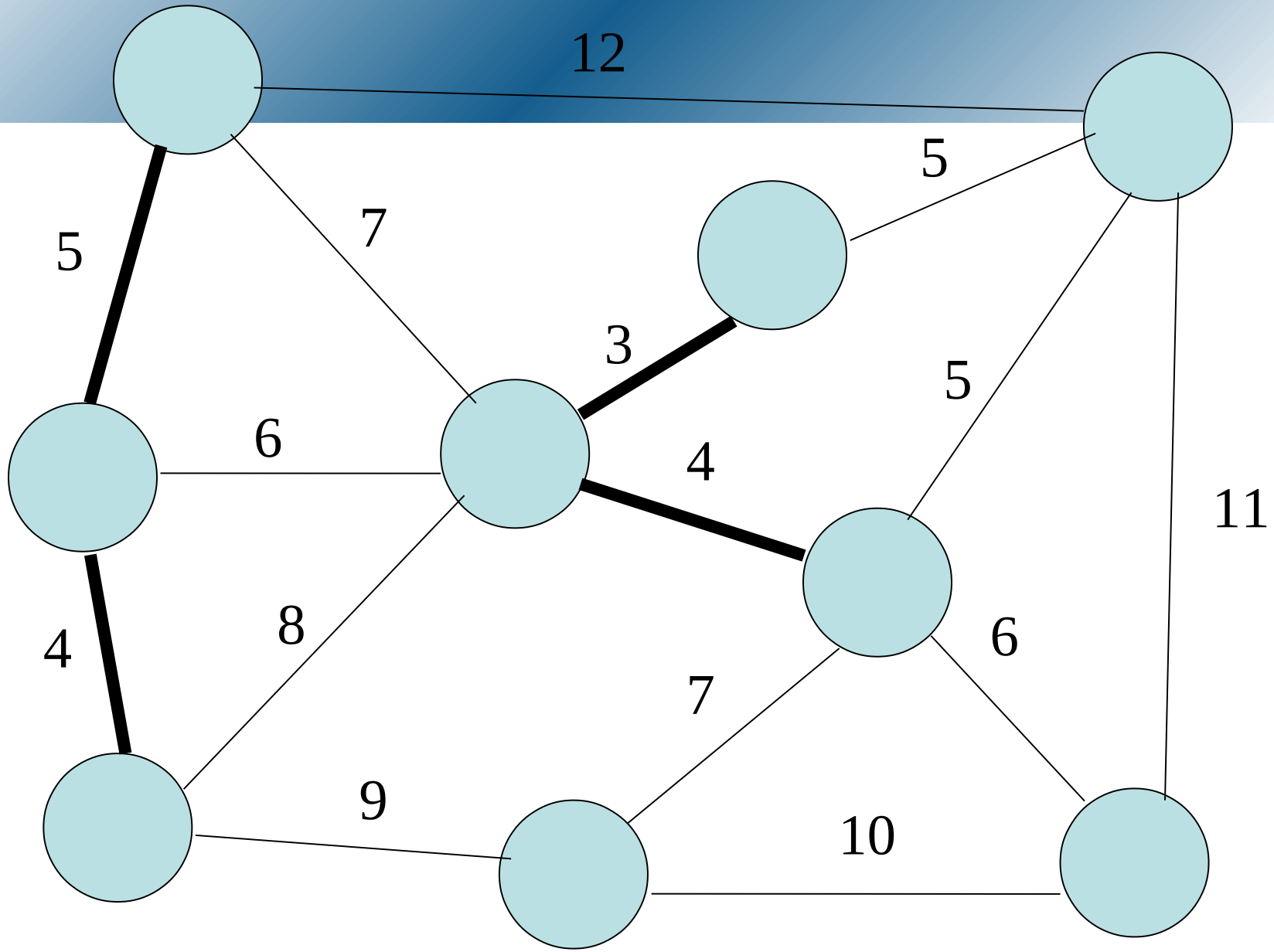
Algoritmo de Kruskal

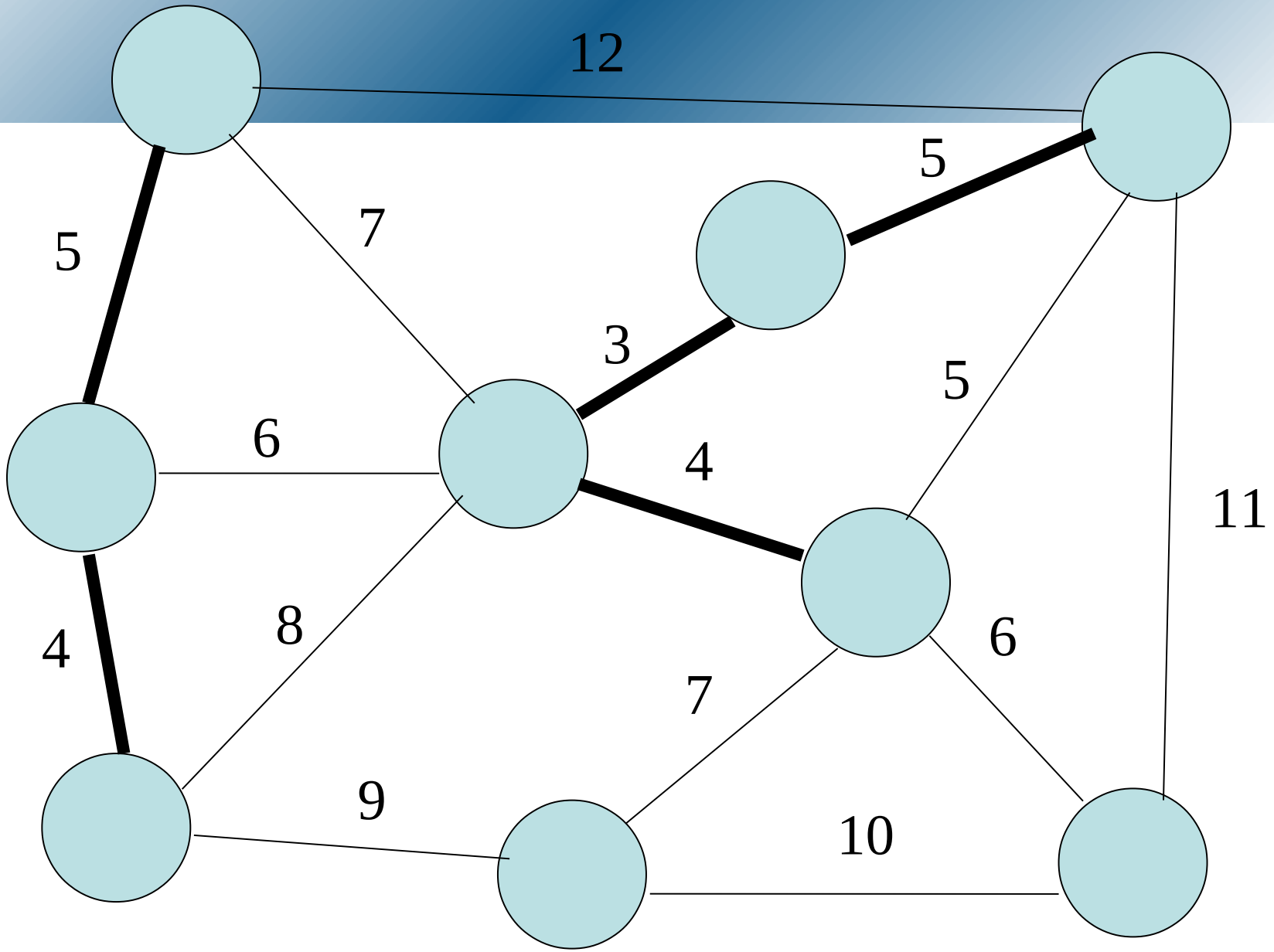
```
 Kruskal_I (Grafo G(V,A))
{ set<arcos> C(A);
  set<arcos> S;                  // Solución inic. Vacía
  Ordenar(C);
  while (!C.empty() && S.size()!=V.size()-1) { //No solución
     x = C.first(); //seleccionar el menor
     C.erase(x);
      if (!HayCiclo(S,x))  //Factible
         S.insert(x);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

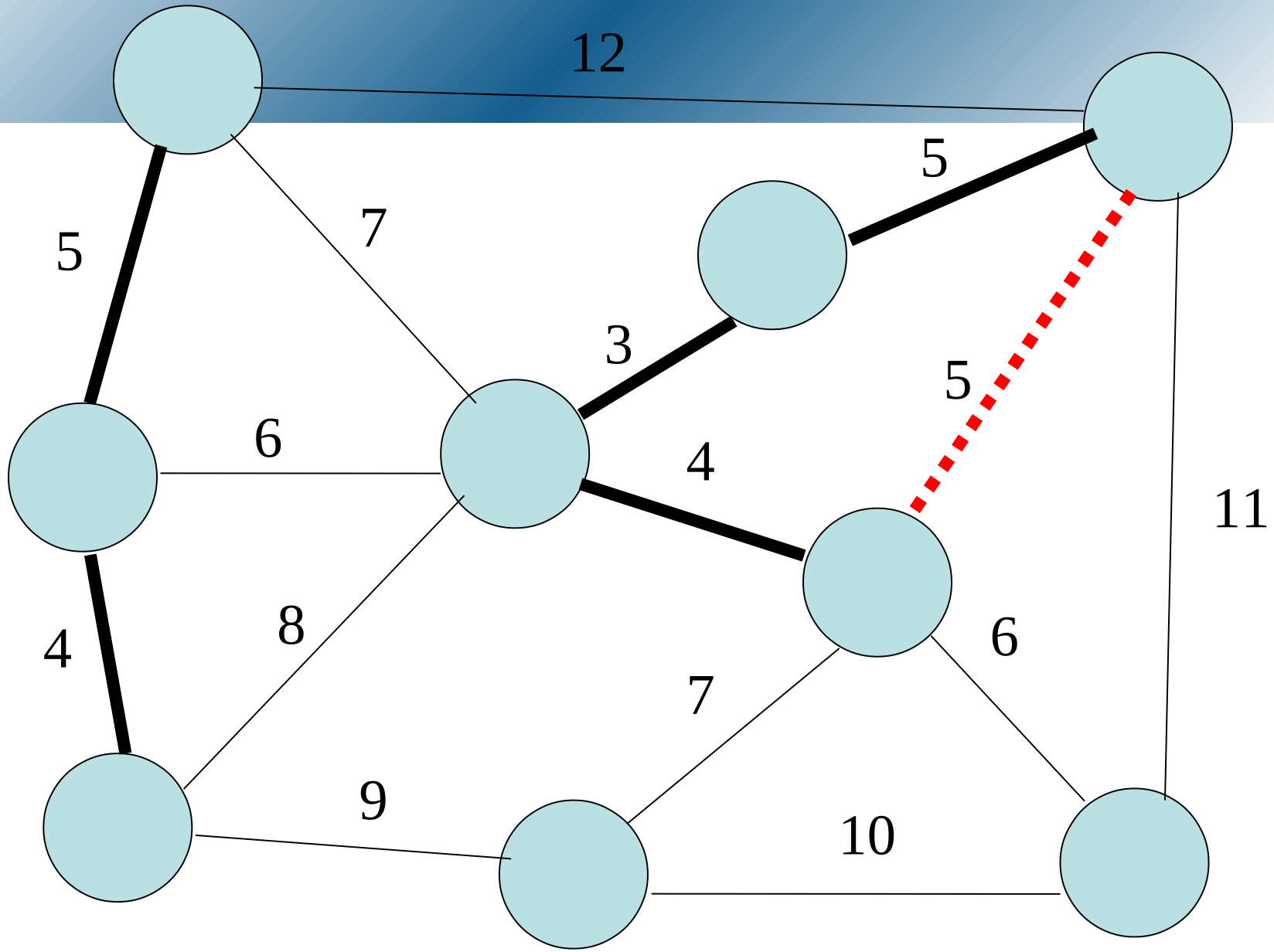

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

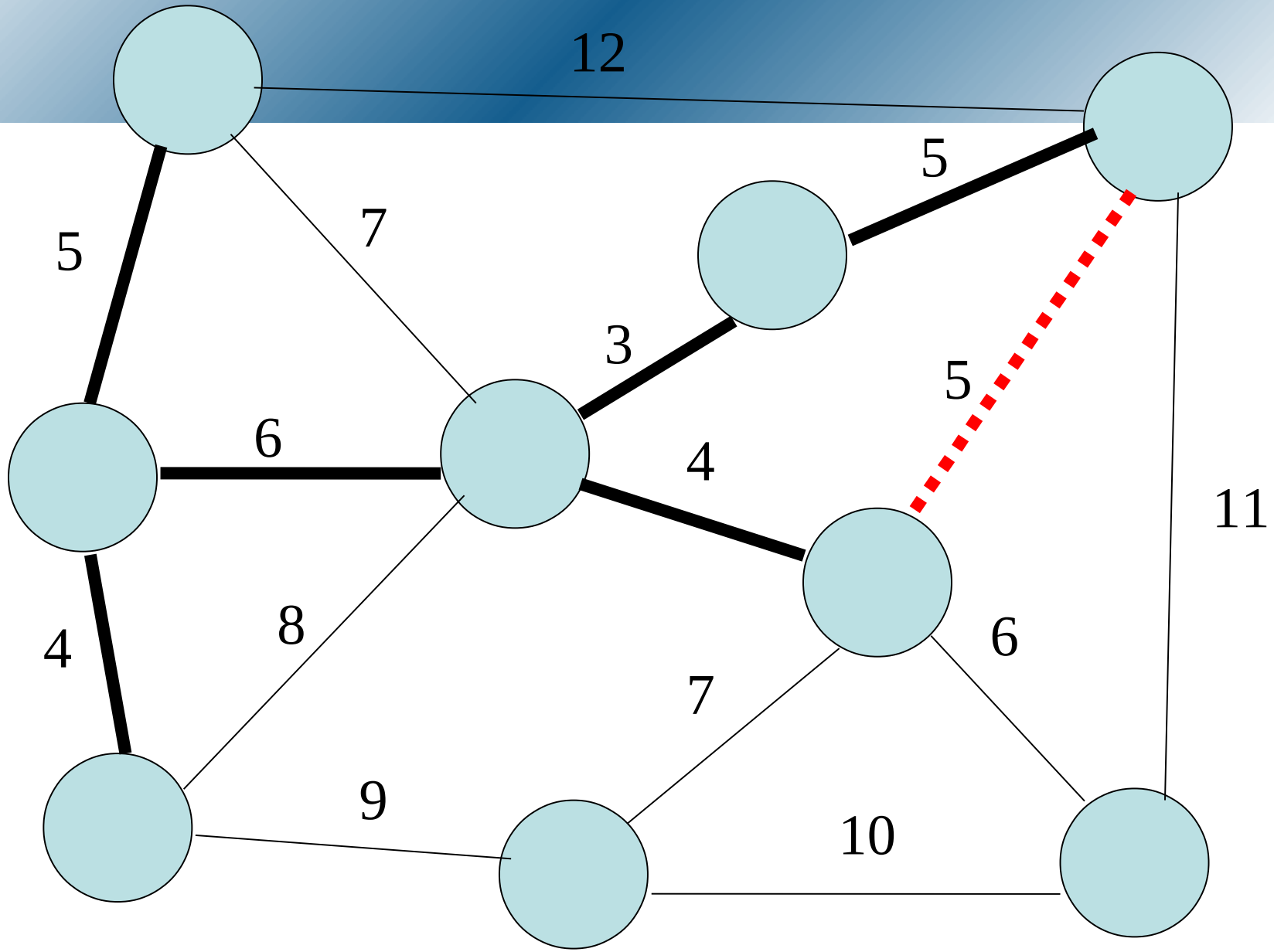

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

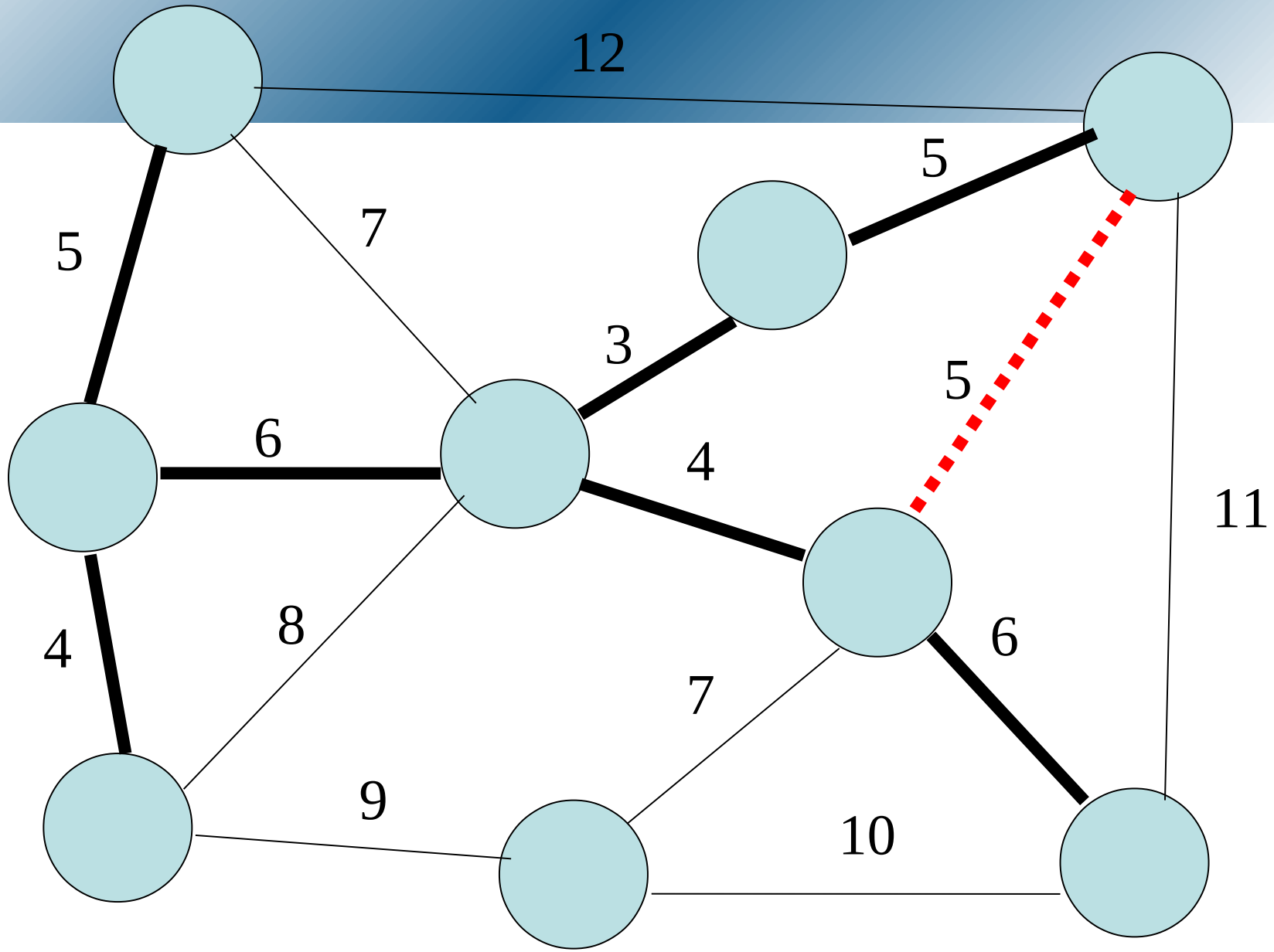

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

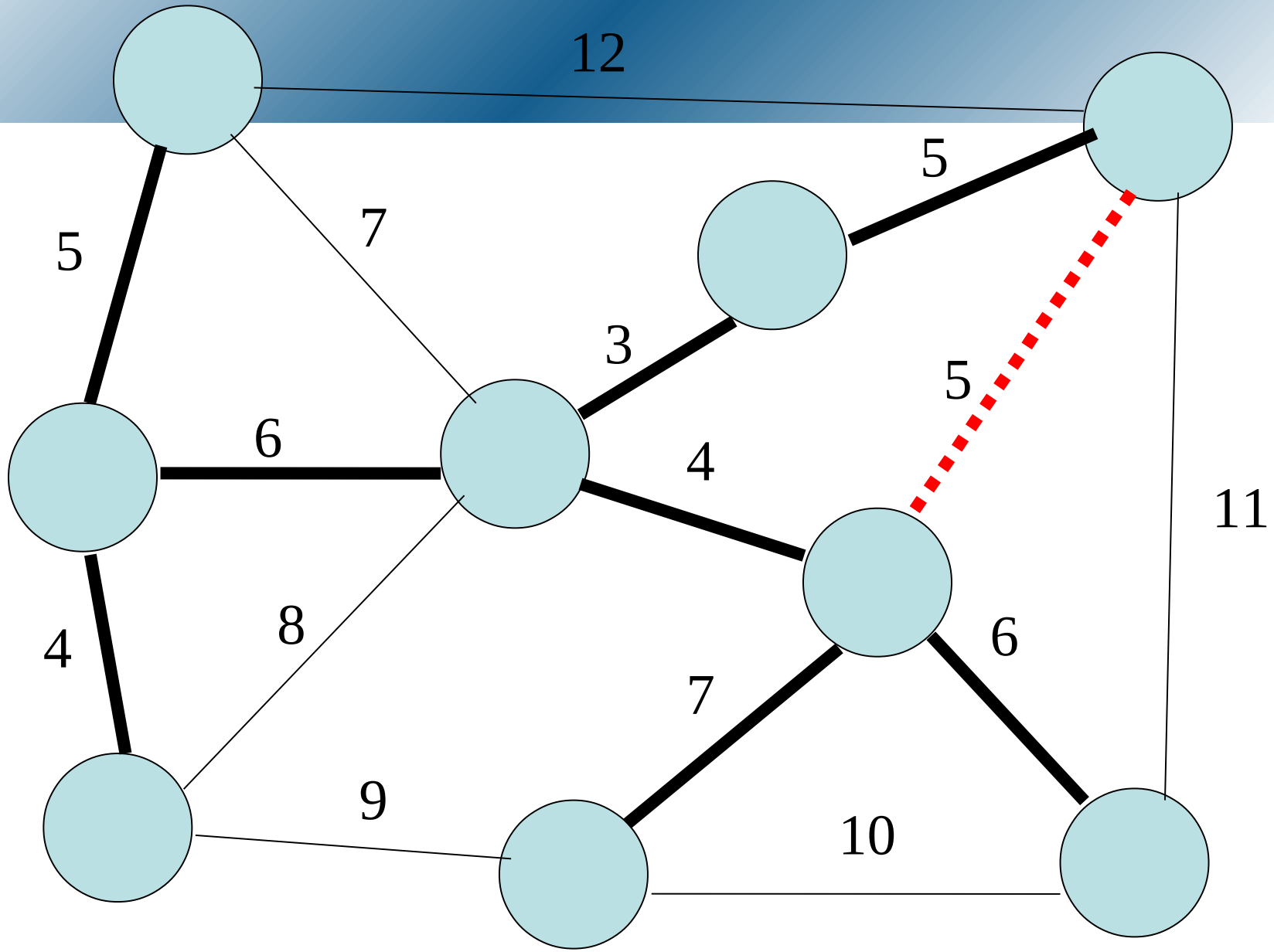

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

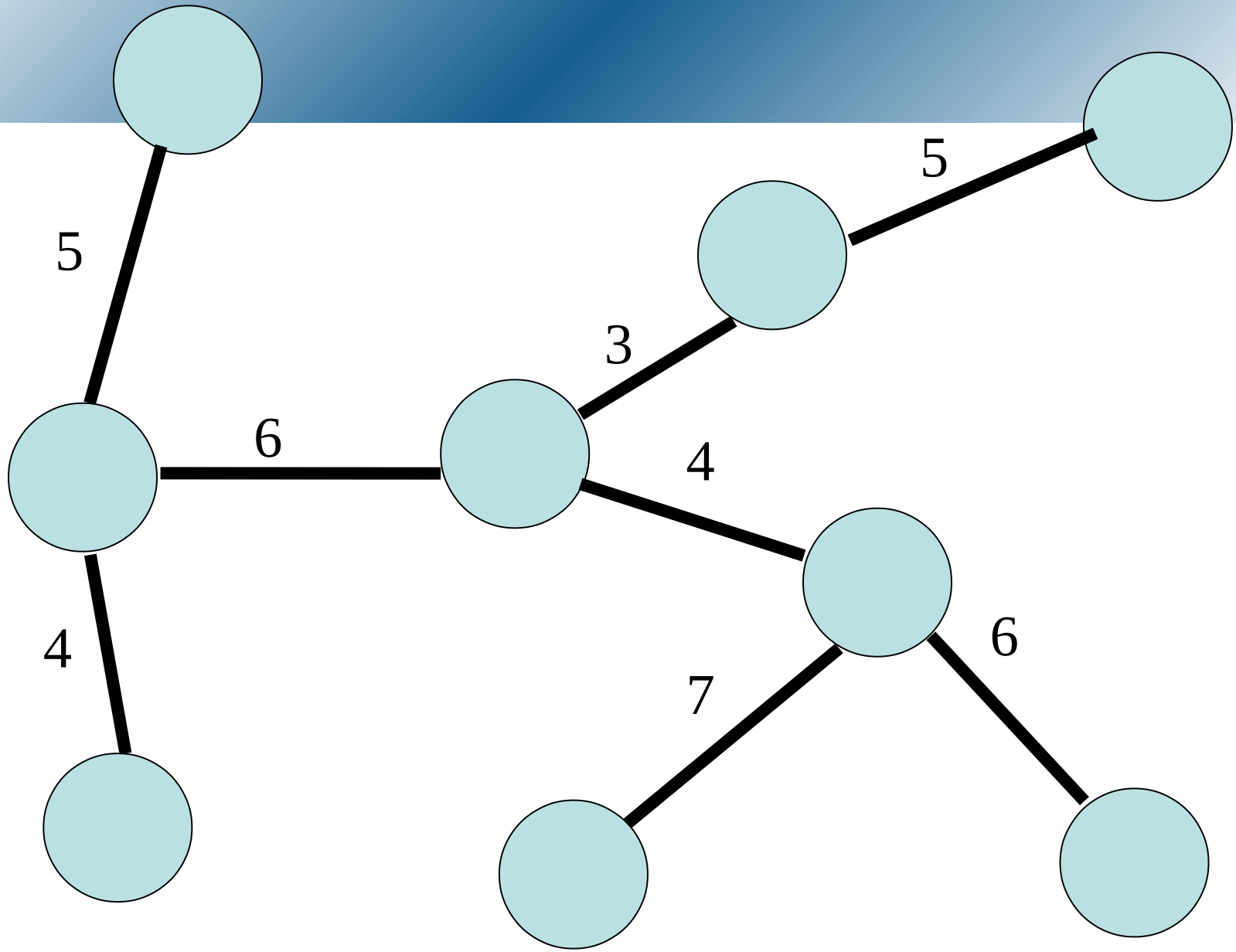

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10


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
6
7

Optimalidad de Kruskal (Directa)

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 (Directa)

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 n-1.

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

Demostramos que es cierto para n

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

Optimalidad de Kruskal (Directa)

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 (Directa)

Paso Inducción: Red. Absurdo.

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

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

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

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

Si tenemos peso(kn) < 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(V)
      }
   if (S.size()==V.size()-1) return S;     // Hay solución
   else return "No_hay_solucion";
}
```

Algoritmo de Kruskal

¿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 arbol 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)
     if (FindSet(u) != FindSet(v)
        S = S U {{u,v}};
        Union(u,v); // Unimos los dos conjuntos
     }
}
```

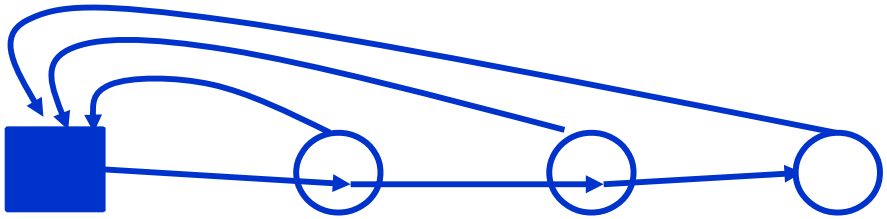
```
Kruskal( G(V,A) ) {
  S = 0≡;  // Inicializamos S con el conjunto
  vacio
  for (i=0; i< V.size()-1; i++)
    MakeSet(V[i]);          O(1)
  Ordenar(A) //Orden creciente   O(A log A)

while (!A.empty() && S.size()!=V.size()-1) {
  //No solución
    (u,v) =A.first(); O(1)
    if (FindSet(u) !=≠ FindSet(v)) O(1)
       S = S U {{u,v}}; O(1)
       Union(u,v); // O(V) cuello debotella, !!!!
    }
}
```

Análisis de Eficiencia

Eficiencia del Algoritmo de Kruskal

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



    - MakeSet(): O(1) FindSet(): O(1)
    - Union(A,B): "Copia" elementos de A a B haciendo que los elementos de A tambien apunten a B: O(A)
  - *¿Cuanto tardan en realizarse las n*

Unión de conjuntos disjuntos

- Análisis del peor caso: $O(n^2)$

  Union(S_1, S_2) "copia" 1 elemento

  Union(S_2, S_3) "copia" 2 elementos

  Union(S_{n-1}, S_n) "copia" n-1 elementos

  $O(n^2)$

- Mejora: Siempre copiar el menor en el mayor
  - *Peor caso* un elemento es copiado como máximo log(n) veces => n Uniones es O(n lg n)
  - Por tanto, el análisis amortizado dice que una unión es de O(log n)

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)
    if (FindSet(u) !=ᵡᵡ  FindSet(v)) O(1)
       S = S U {{u,v}}; O(1)
       Union(u,v); // O(log V)
    }
}
```

Kruskal es de O(A log V)

Algoritmo de Prim

- Primero veremos que el problema ARM tiene subestructuras optimales
- Teorema : Sea T *un ARM y sea (u,v) una arista de T. Sean* T_1 y T_2 los dos árboles que se obtienen al eliminar la arista (*u,v*) de T. Entonces T_1 es un ARM de $G_1 = (V_1,E_1)$, y T_2 es un ARM de $G_2 = (V_2,E_2)$
  - Demostración: Simple, por red. absurdo

Idea, partir de que

$$peso(T) = peso(u,v) + peso(T_1) + peso(T_2)$$

Por tanto, no puede haber arboles de recubrimiento mejores que T_1 o T_2, pues si los hubiese T no sería solución óptima.

Algoritmo de Prim

Se basa en el siguiente

Teorema

- Sea T un ARM optimal de G, con A $\subseteq$ T un subárbol de T y sea (u,v) la arista de menor peso conectando los vértices de A con los de V-A. Entonces, $(u,v) \in$ T

  Demostración: Se deja como ejercicio.

Algoritmo de Prim

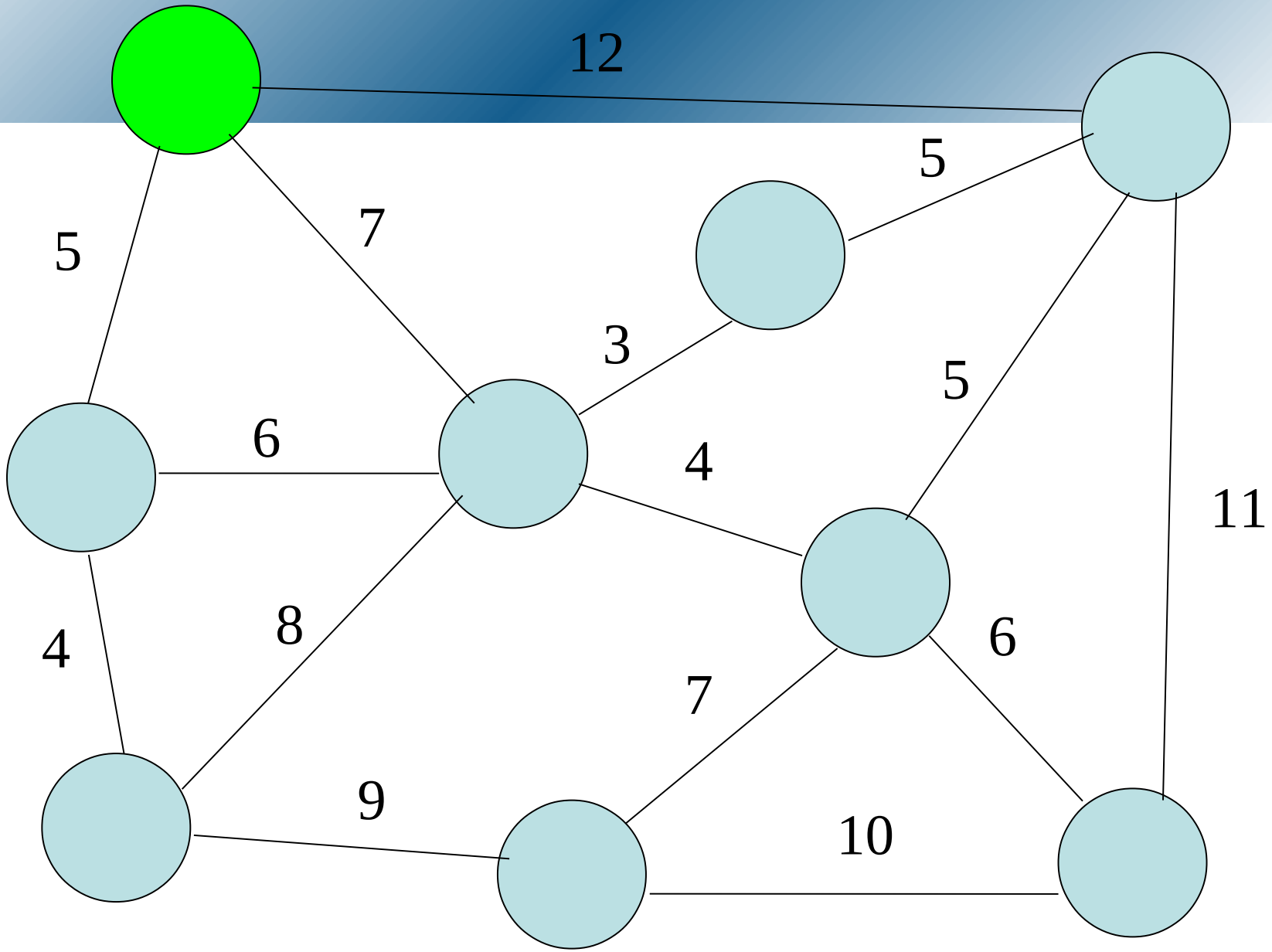
Candidatos: *Vértices*

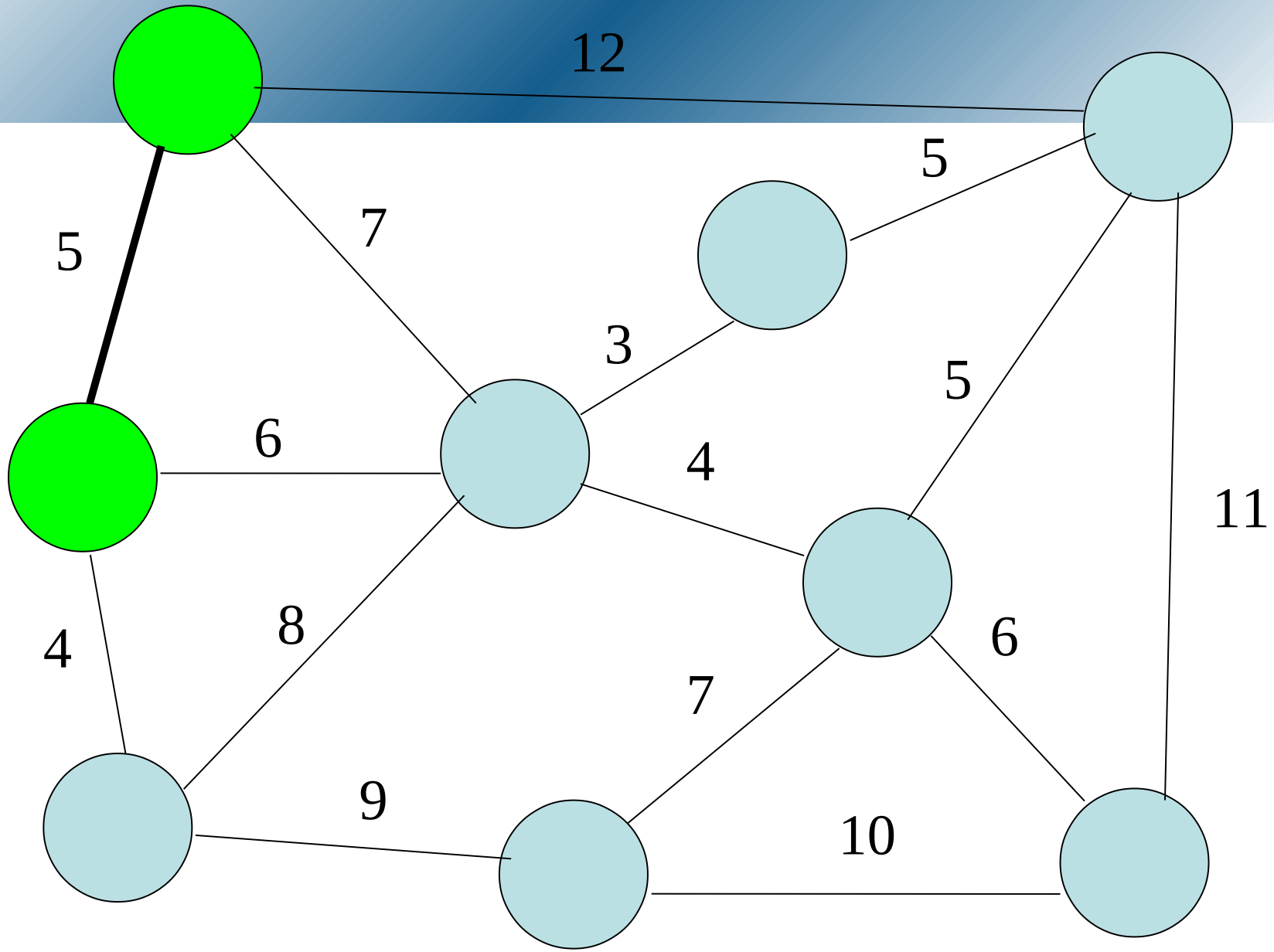
Función Solución: Se ha construido un árbol de recubrimiento (n vértices seleccionadas).

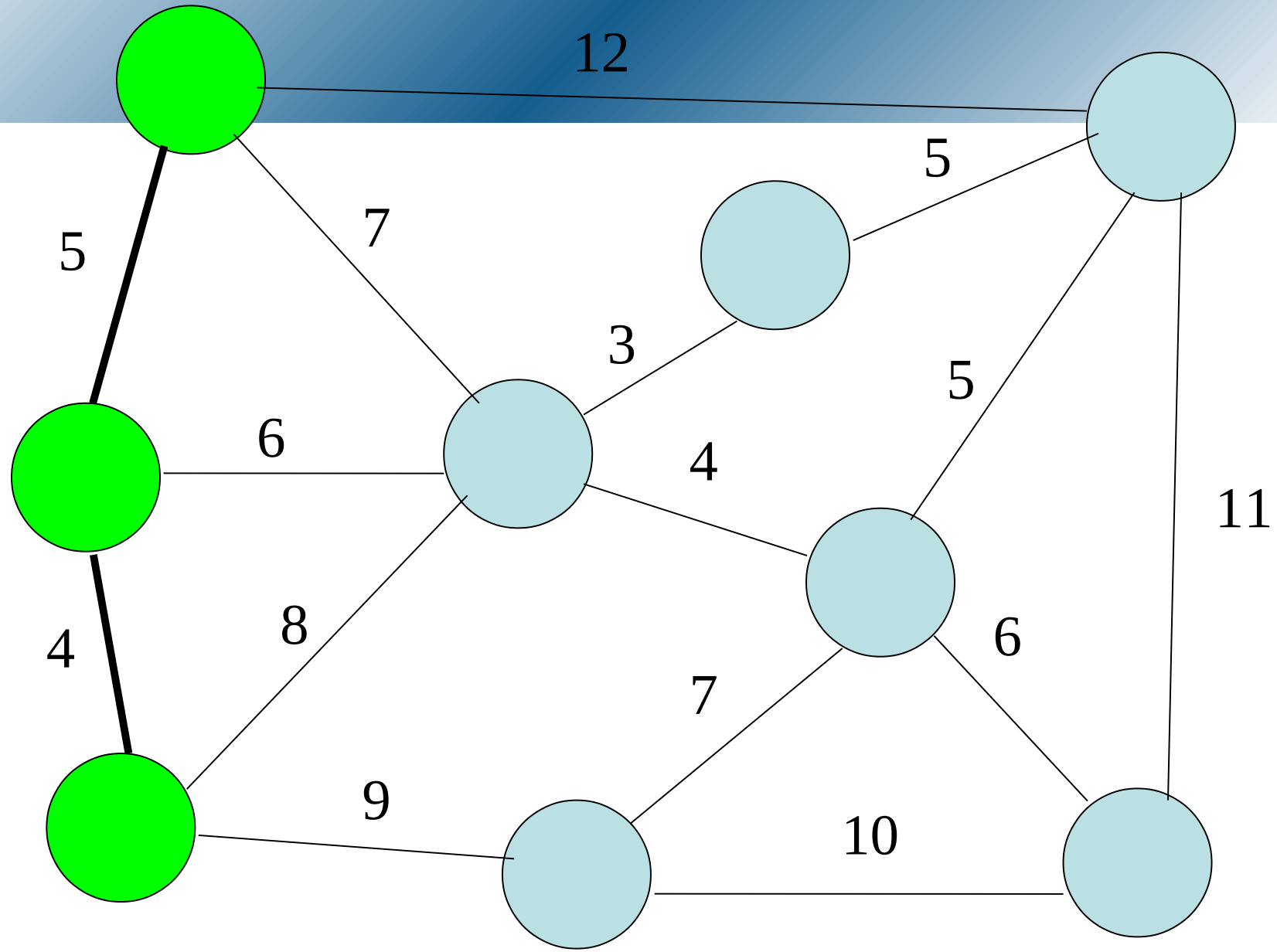
Función Selección: Seleccionar el vertice u del conjunto de no seleccionados que se conecte mediante la arista de menor peso a un vértice v del conjunto de vértices seleccionados. La arista (u,v) está en T.

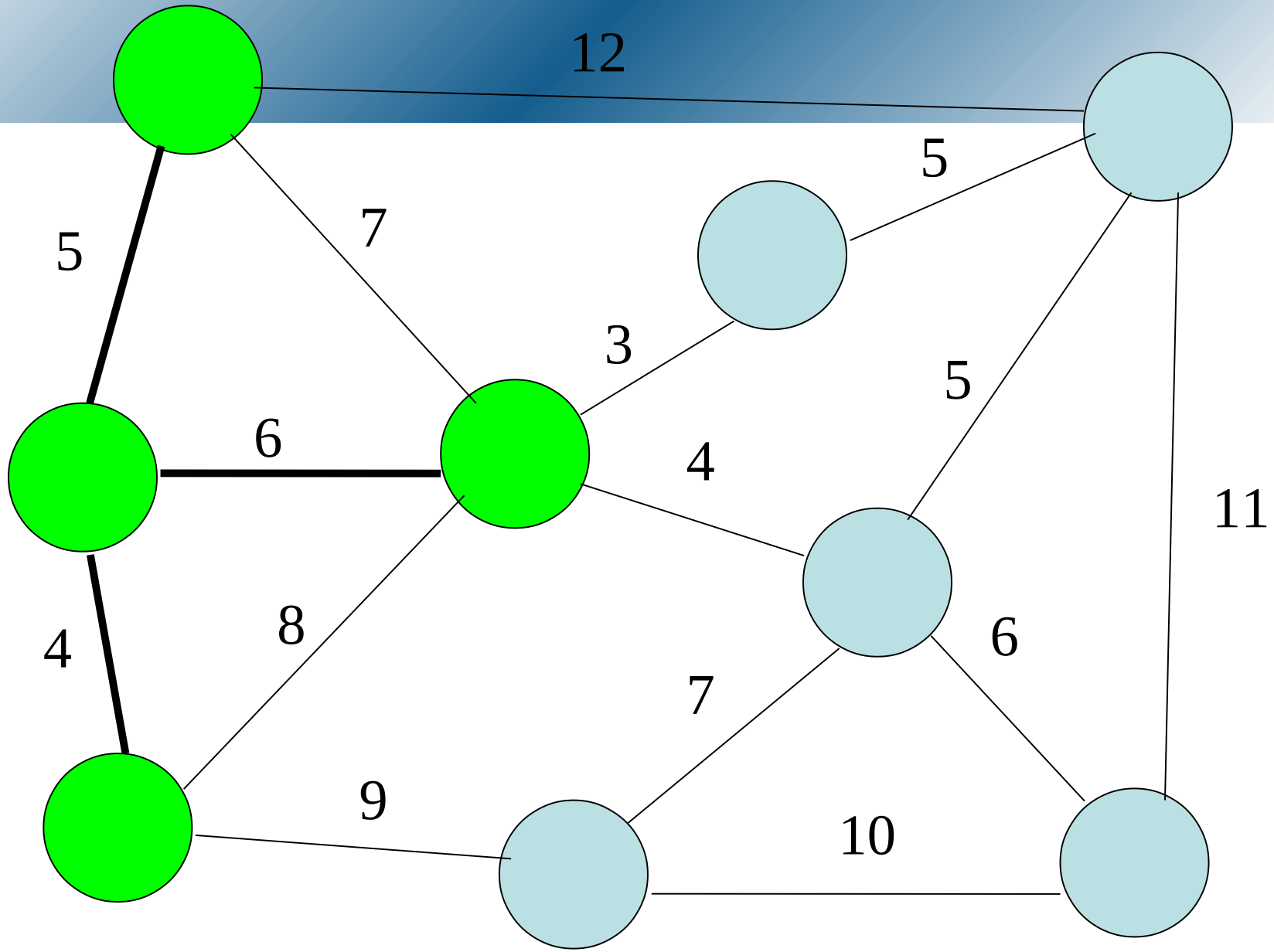
Función de Factibilidad: *El conjunto de aristas no contiene ningún ciclo. Está implícita en el proceso*

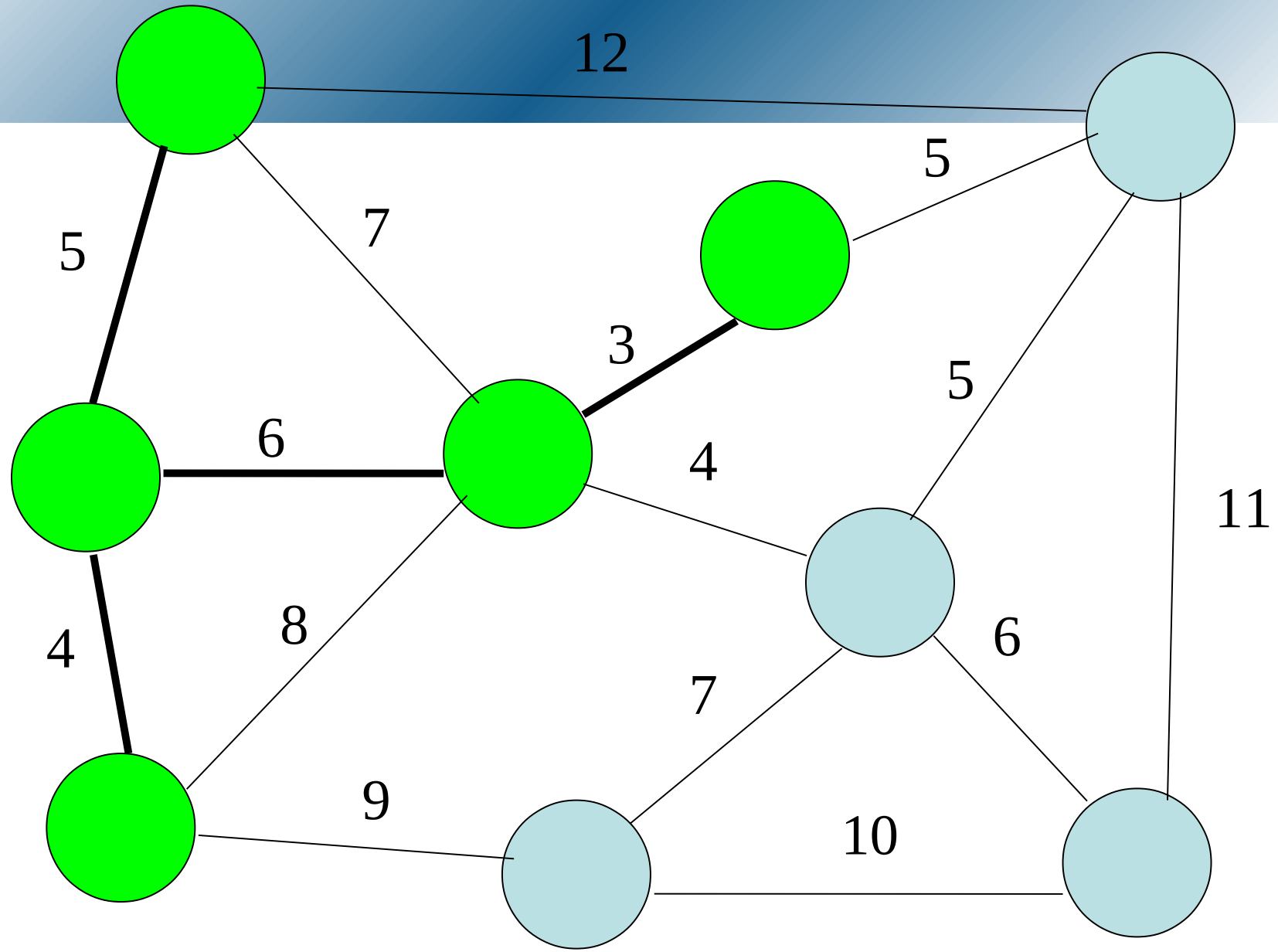
Función Objetivo: determina la longitud total de las aristas seleccionadas.

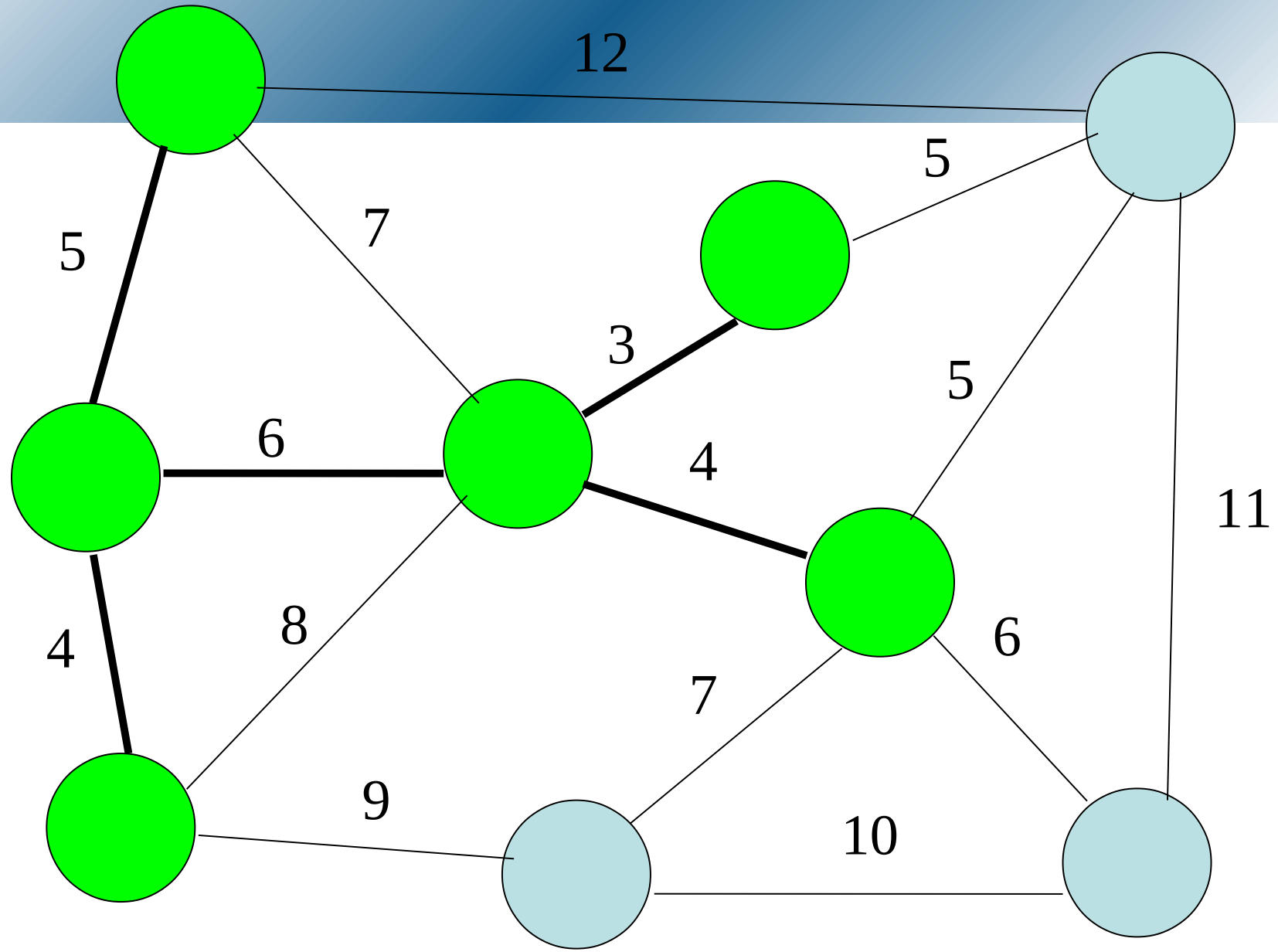

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

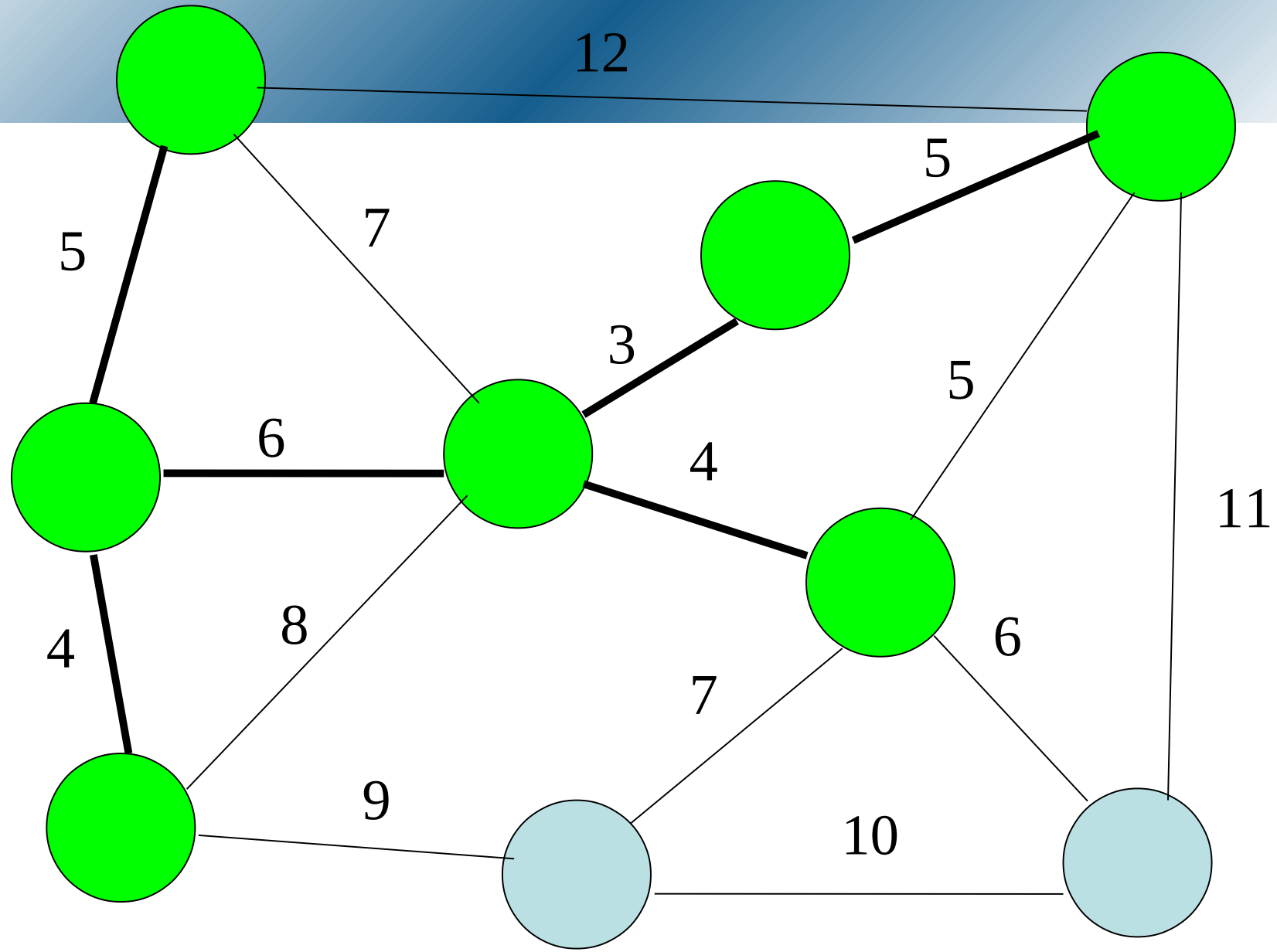

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10

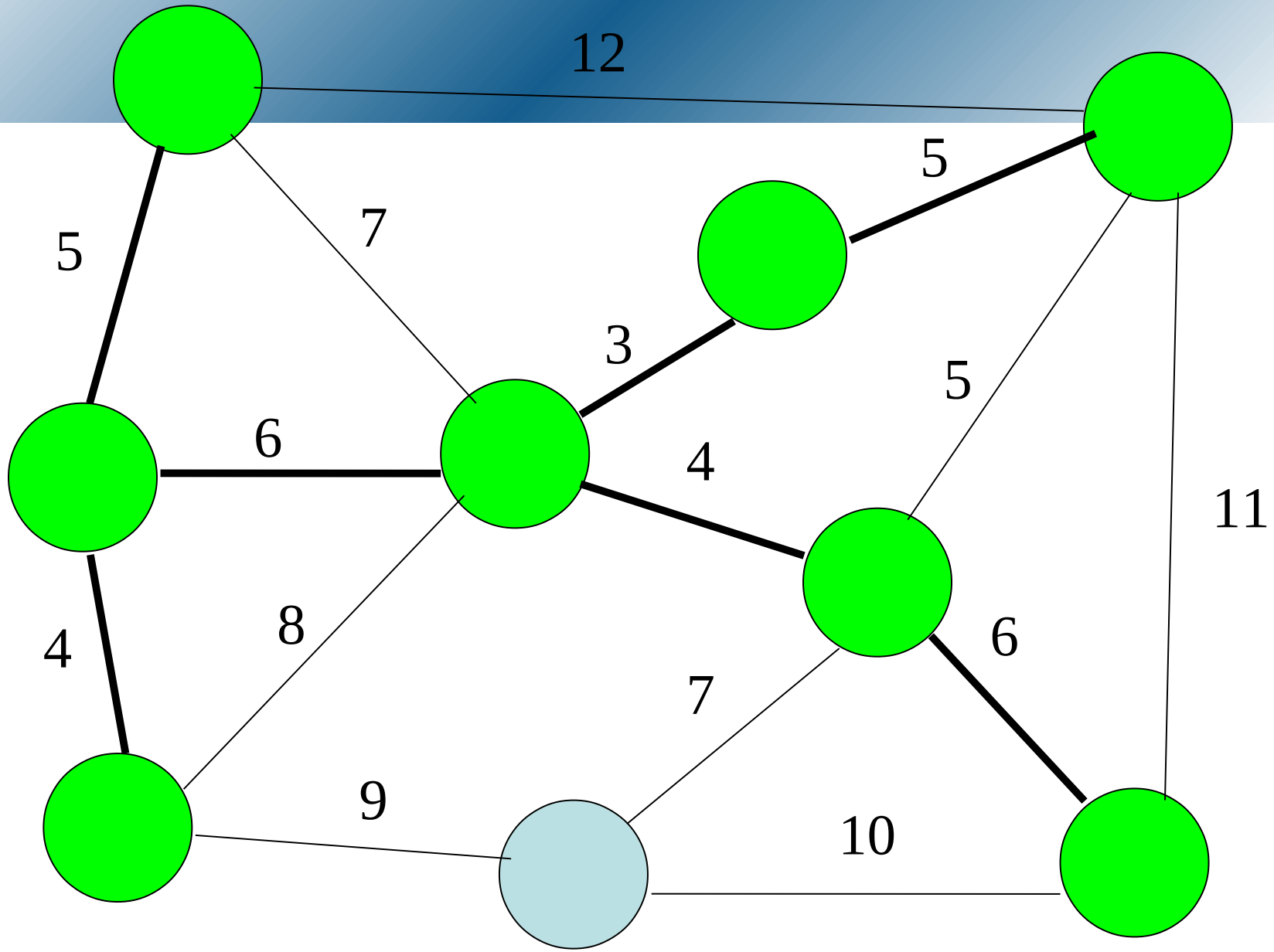

12
5
7
5
5
3
6
4
5
11
4
8
6
7
9
10

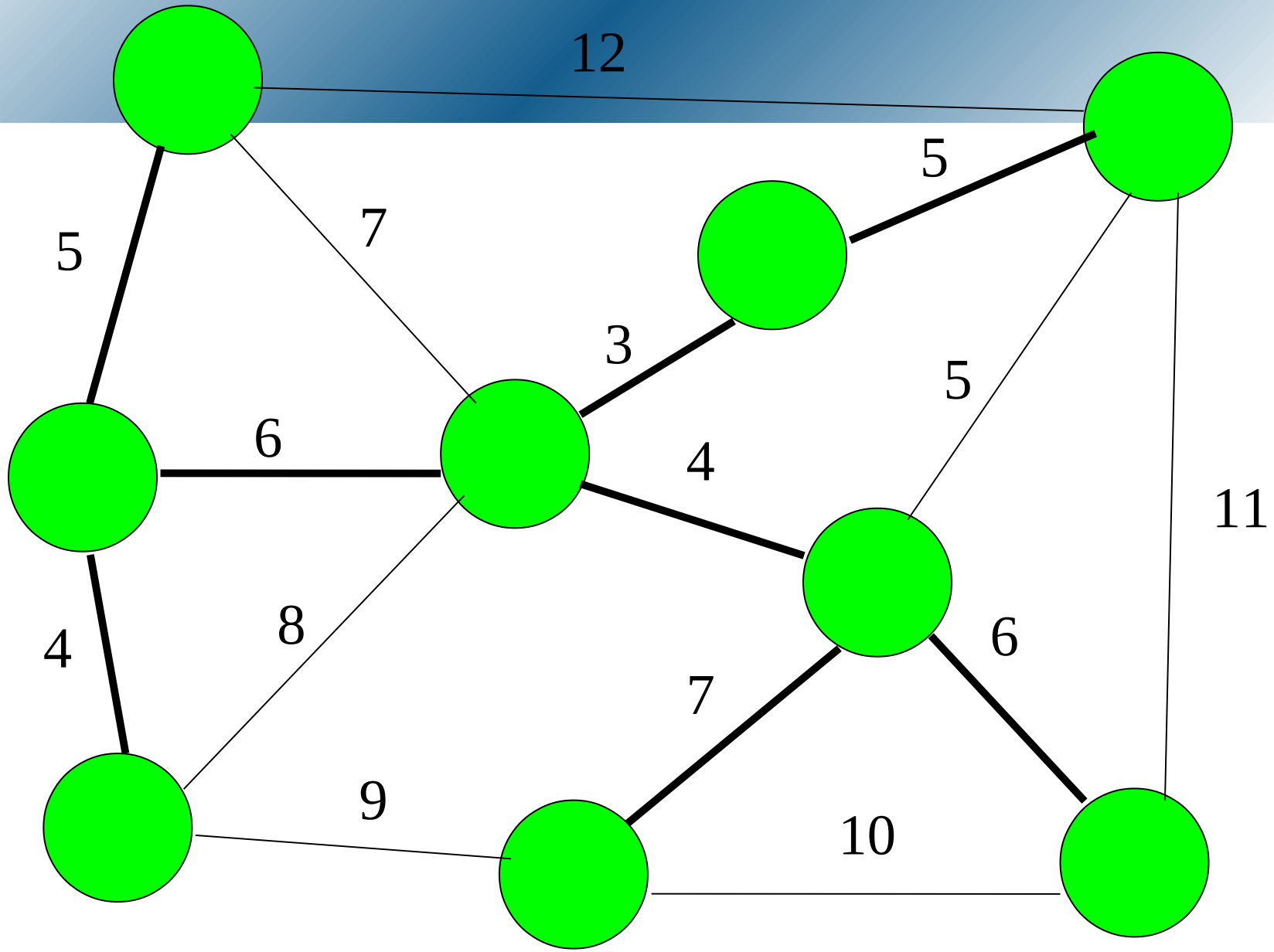

12
5
7
5
3
5
6
4
11
4
8
6
7
9
10


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

Implementación

- Clave: Seleccionar eficientemente el nuevo arco para añadir al ARM, T.

- Utilizaremos una cola con prioridad Q de vértices con dos campos:
  - key[v] = menor peso de un arco que conecte v con un vértice en el conjunto de arcos seleccionados. Toma el valor infinito si no existe dicho arco
  - p[v] = padre del v en el arbol.

Implementación

```
Función Prim(G(V,A), r ) // r es el vertice_inicio.
  for each u ∈ en V
      key[u] = infinito;
      p[u] = NULL
   key[r] = 0;
  PriorityQueue Q(V-r) // con todos los vértices de V, menos r
   while (!Q.empty())
      u = Q.ExtractMin();
      for each v  en Adj[u]
         if (v en ∈  Q and peso(u,v) < key[v])
            p[v] = u;
            key[v] = w(u,v);
```

El ARM está almacenado en la matriz de padres P

Eficiencia algoritmo Prim

```
Función Prim(G(V,A), r ) // .
  for each u ∈ en V
      key[u] = infinito⊣;
      p[u] = NULL
   key[r] = 0;
  PriorityQueue Q(V-r) // O(V log V), se puede bajar a O(V).
   while (!Q.empty())
      u = Q.ExtractMin();  O(log V), en total O(V log V)
      for each v  en Adj[u]
         if (v en∈  Q and peso(u,v) < key[v])
            p[v] = u;
            key[v] = w(u,v);
```

Eficiencia algoritmo Prim

```
Función Prim(G(V,A), r ) // .
  for each u ∈ en V
      key[u] = infinito⊣;
      p[u] = NULL
   key[r] = 0;
  PriorityQueue Q(V-r) // O(V log V), se puede bajar a O(V).
   while (!Q.empty())
      u = Q.ExtractMin();  O(log V), en total O(V log V)
      for each v  en Adj[u]
         if (v en∈  Q and peso(u,v) < key[v])          O(1)
            p[v] = u;
            key[v] = w(u,v); => Modificar Q => O(log V)
```

Este cómputo se realiza una vez para cada arco del grafo, por tanto en total tenemos un tiempo O(A log V)

Eficiencia algoritmo Prim

- Finalmente, podemos concluir que la eficiencia del algoritmo de Prim es del orden

$$O(A \log V + V \log V) = O(A \log V)$$

Problema de Caminos Minimos

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

Problema de Caminos Minimos

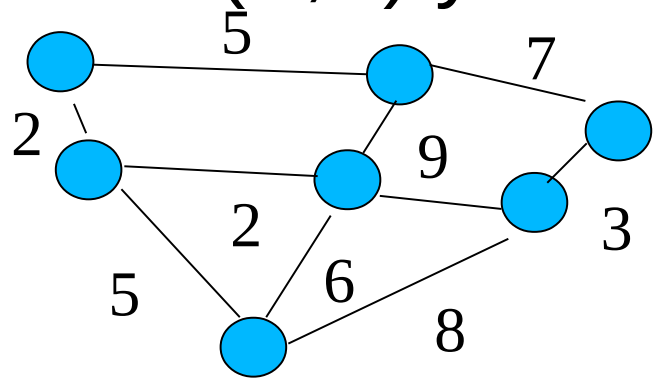
Supongamos que tenemos un mapa de carreteras de España y estamos interesados en conocer el camino más corto que hay para ir desde Granada a Guevejar.

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

Problema de Caminos Minimos

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



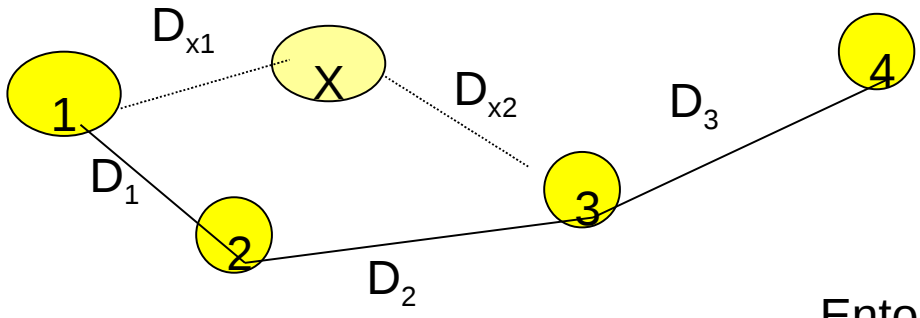
Encontrar el
Camino de costo mínimo para llegar
desde s al resto de los vértices en el grafo

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

Propiedades de Caminos Mínimos

Asumimos que no hay arcos con costo negativo.

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



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

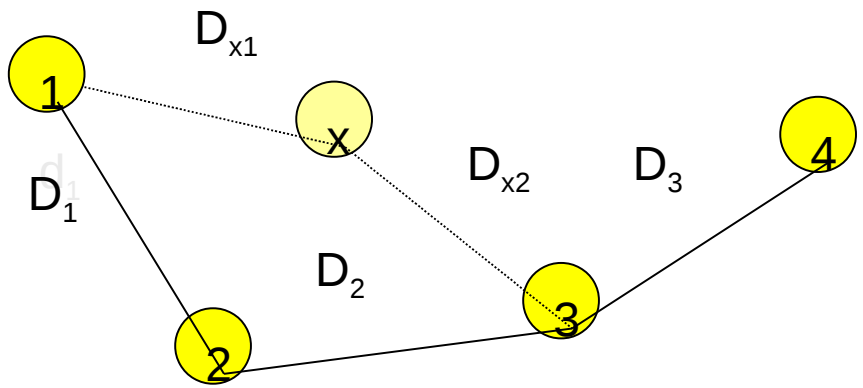
Si $M(1,3) = Dx_1+Dx_2$

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?)

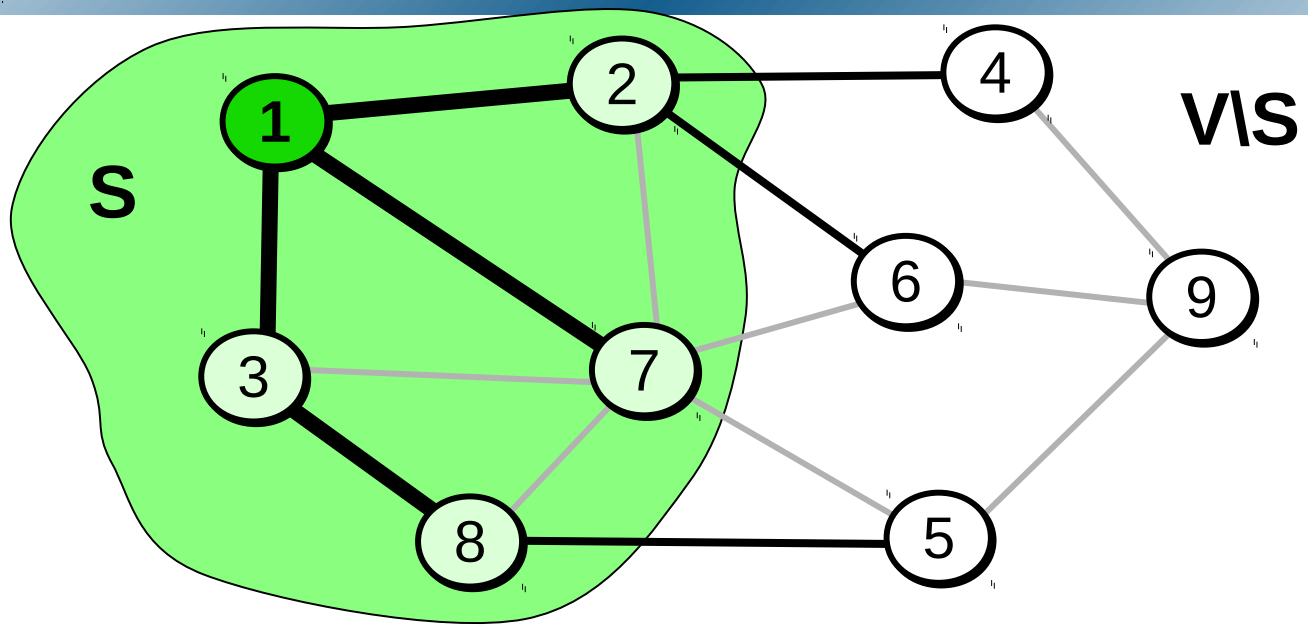


Algoritmo Dijkstra

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

- El algoritmo usa con 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**.

Implementación Dijkstra

Candidatos: *Vértices*

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

Uso de cola con prioridad Q de vértices con dos campos:

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

Implementación: Alg. Dijstra.

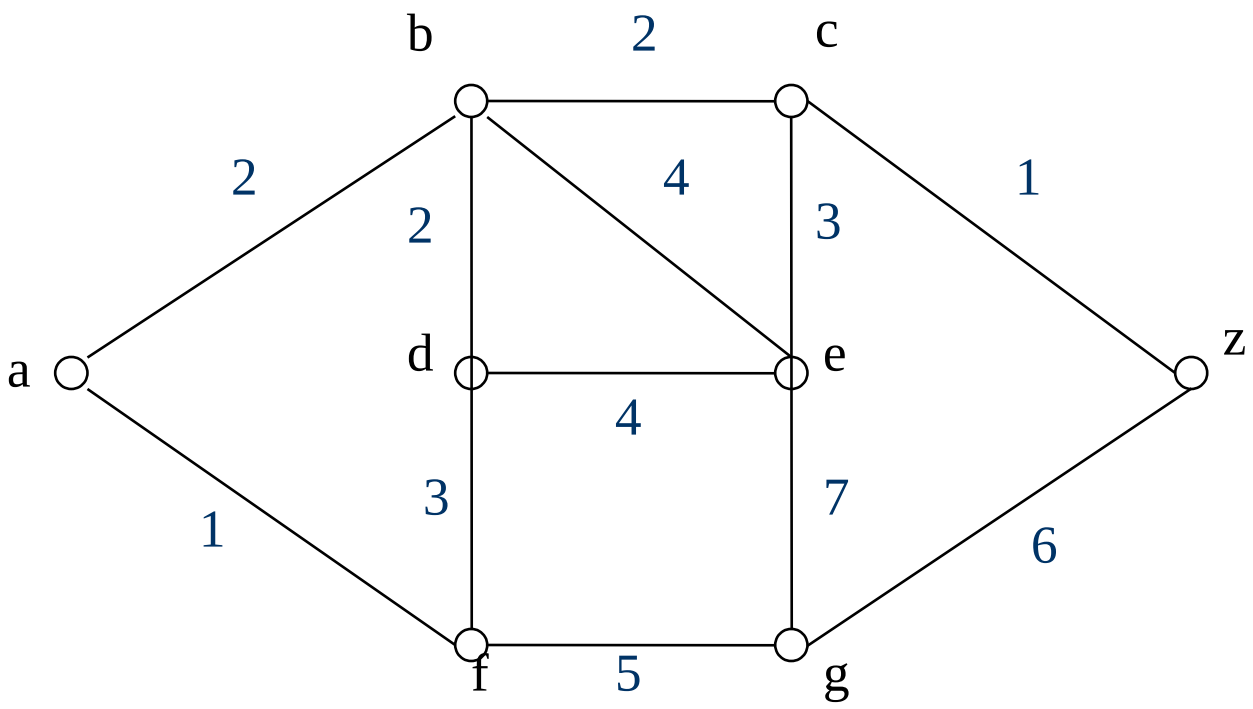
- Al incluirse un vértice x en S puede ocurrir que sea necesario actualizar d[x], esto es, Recalcular los caminos mínimos de los demás candidatos, pudiendo pasar por el nodo cogido.
  - Xq?
  - Qué vértices son suceptibles de sufrir dicha actualización?
  - Como se ve afectado d[x] y p[x]?

Implementación: Alg. Dijstra.

```
Para cada v en V
   d[v] = infinito
   pred(v) = null
d[s]=0
set<vertices> S; // Vèrtices selecciondos 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)
         pred(w) = v
```

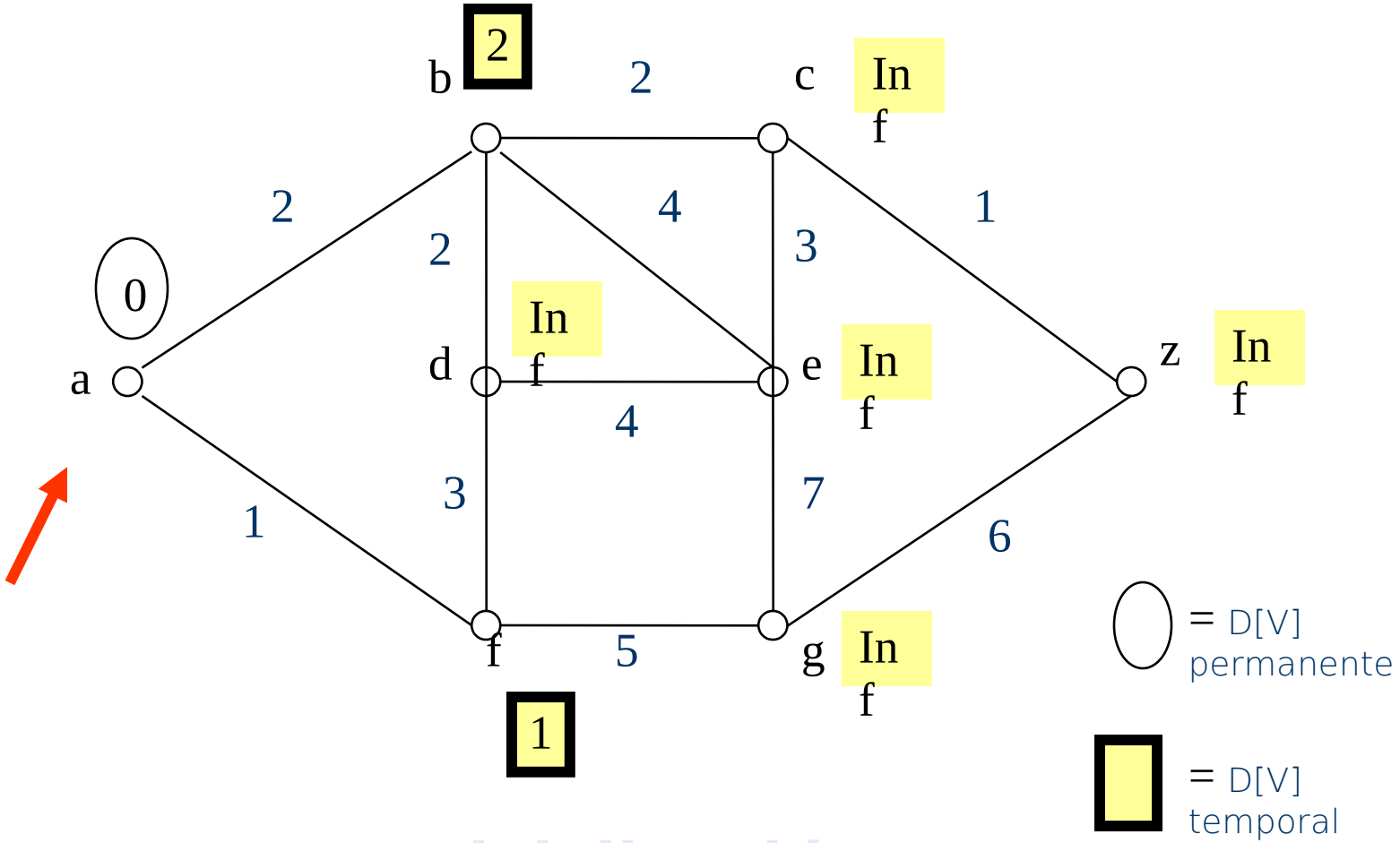
ALGORITMO DE DIJKSTRA

Ejemplo: Algoritmo Dijkstra



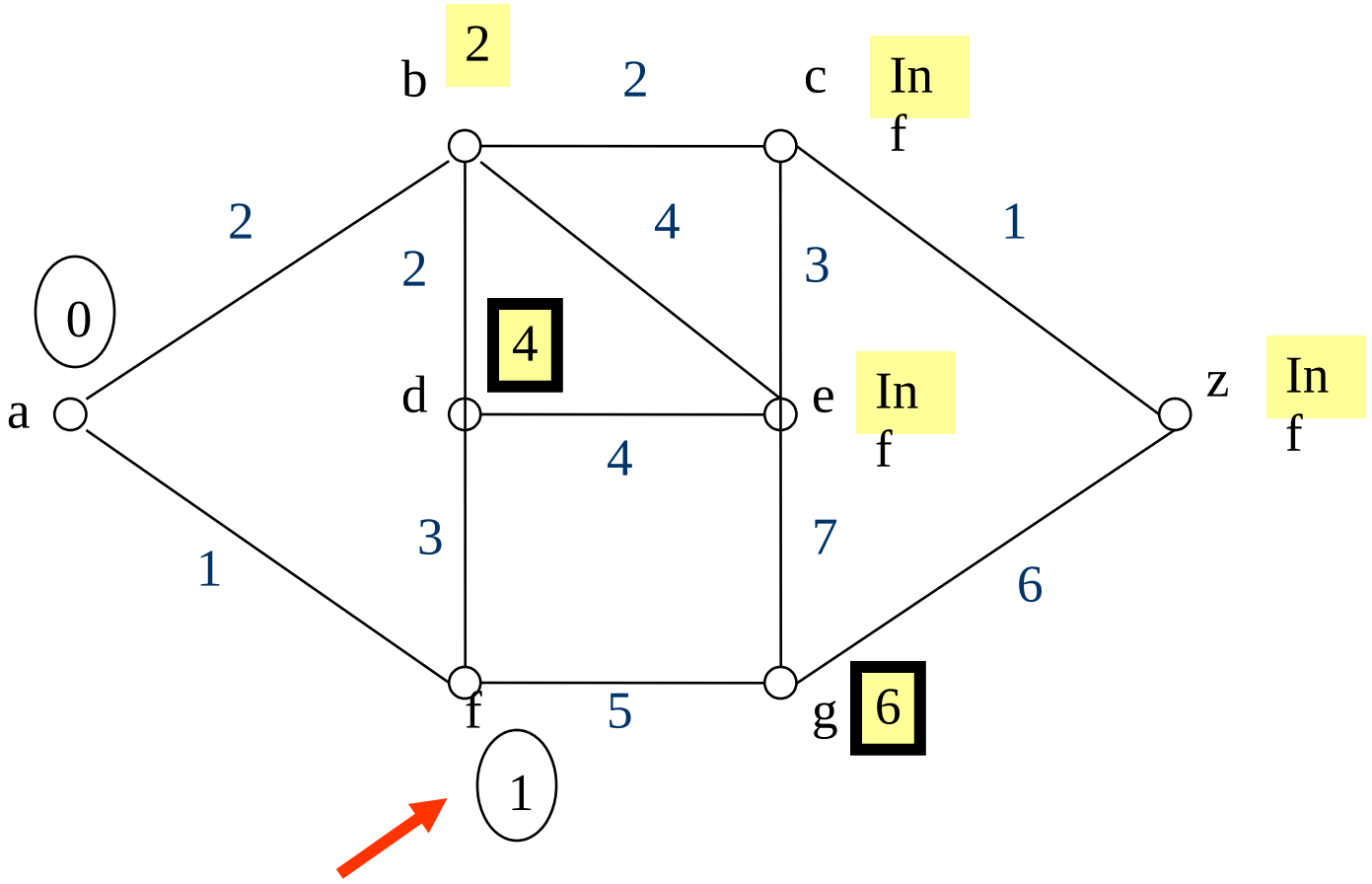
Ejemplo (Solo entre a y z)

Ejemplo: Algoritmo Dijkstra



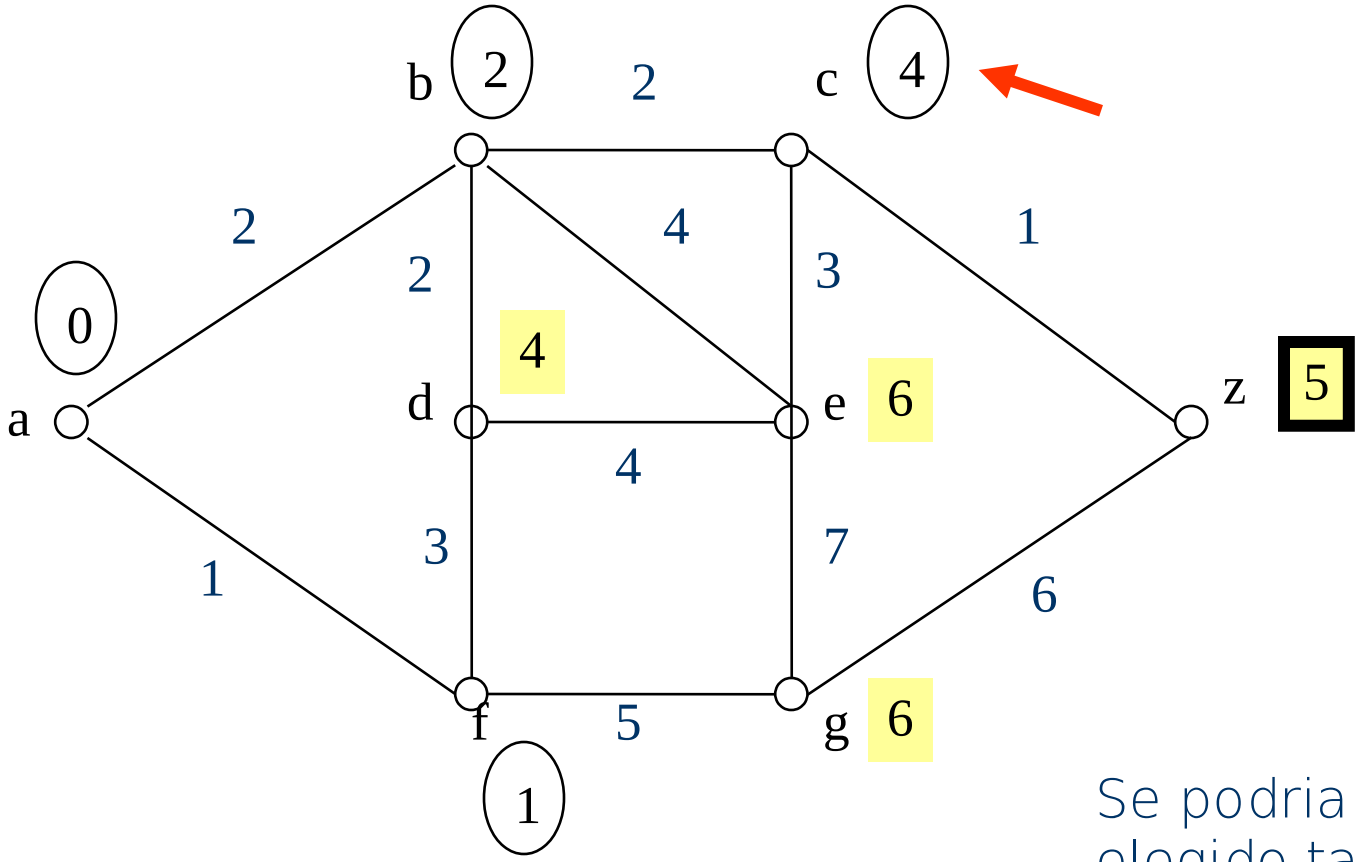
Inicialización

Ejemplo: Algoritmo Dijkstra



Primera Iteración

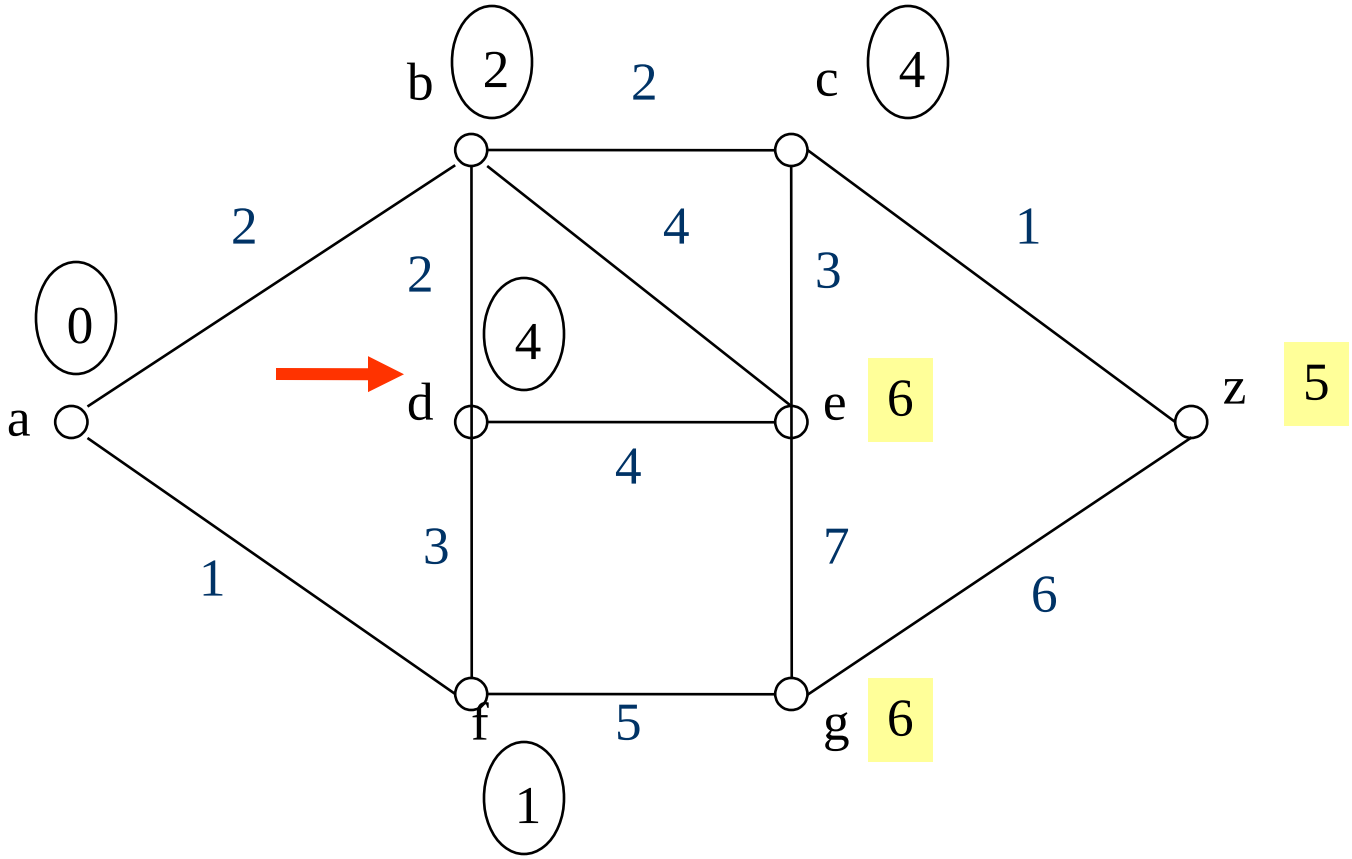
Ejemplo: Algoritmo Dijkstra



Se podria haber elegido tambien 'd'

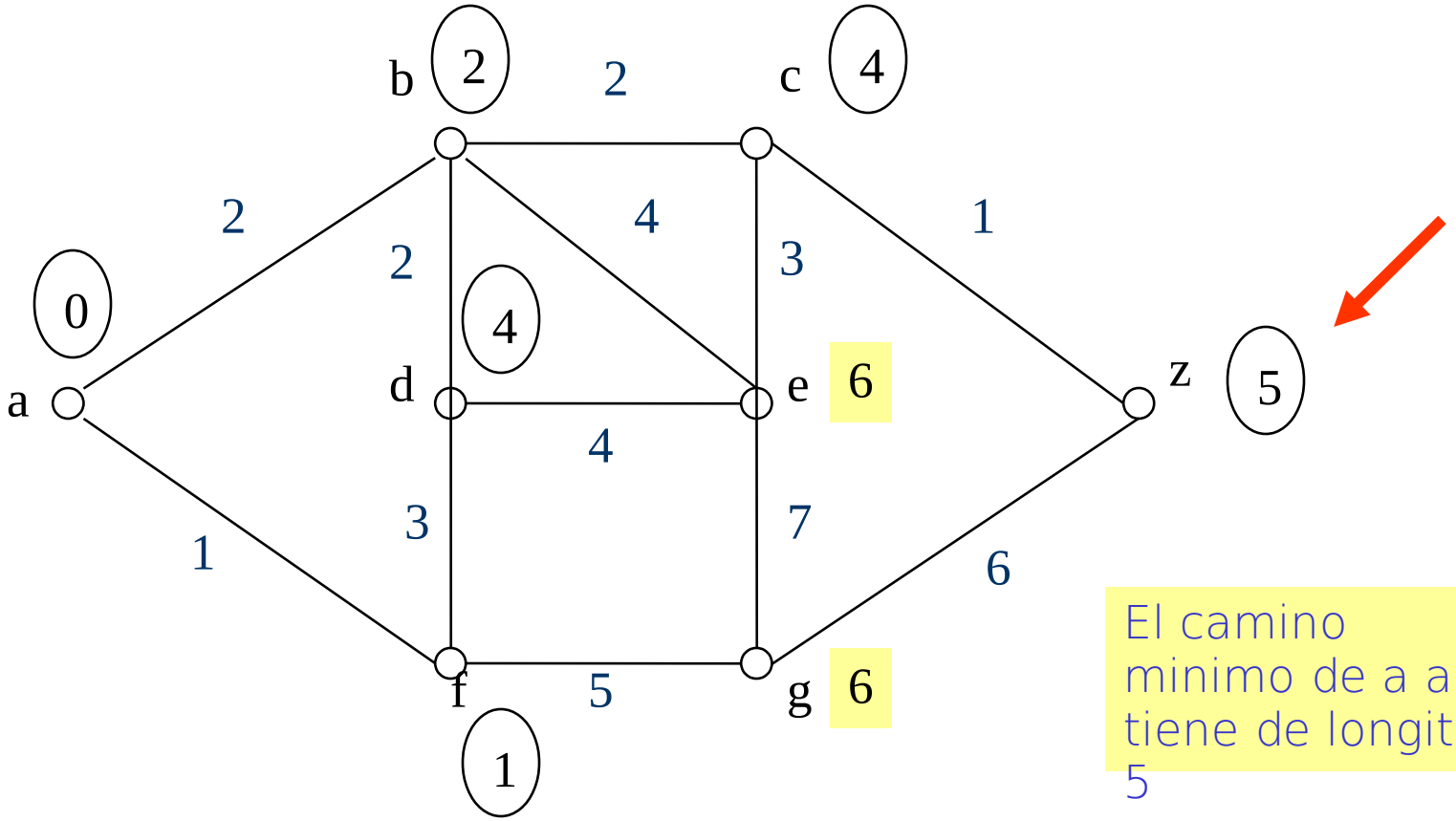
Segunda Iteración

Ejemplo: Algoritmo Dijkstra



Tercera Iteración

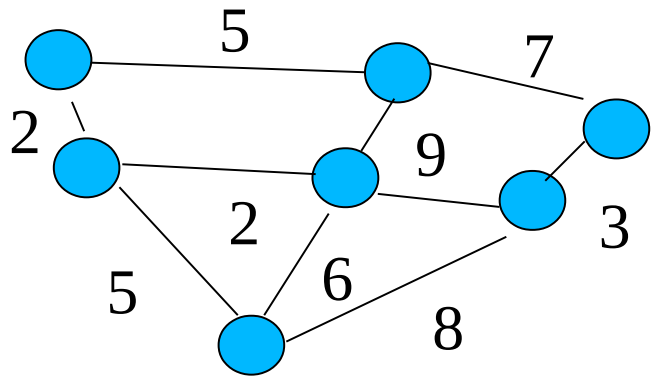
Ejemplo: Algoritmo Dijkstra



El camino minimo de a a z tiene de longitud 5

Cuarta (y ultima) Iteración

Ejemplo: Algoritmo Dijkstra

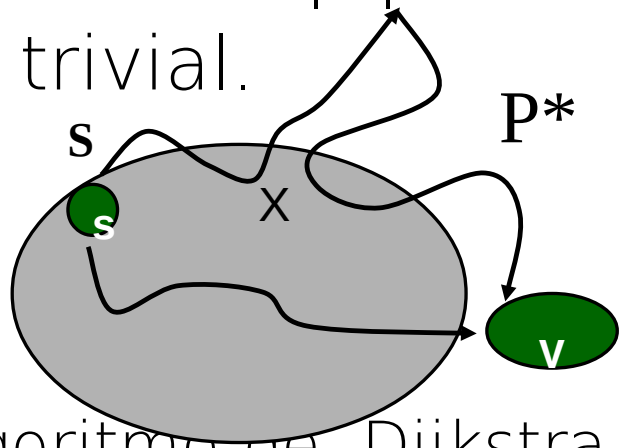


Queda como ejercicio.

Algoritmo Dijkstra: Demostración

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

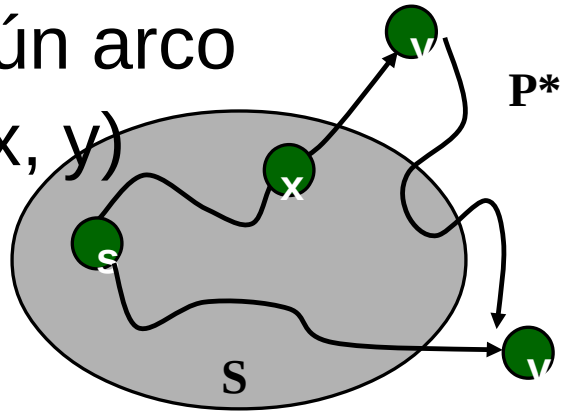
- Demostr. Por inducción sobre $|S|$.
- Caso base: $|S| = 0$ es trivial.



- Paso induccion:
  - Supongamos que el algoritmo de Dijkstra añade el vértice v a S. d(v) rerpresenta 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-v
  - Sea x el último vertice en S en dicho

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)

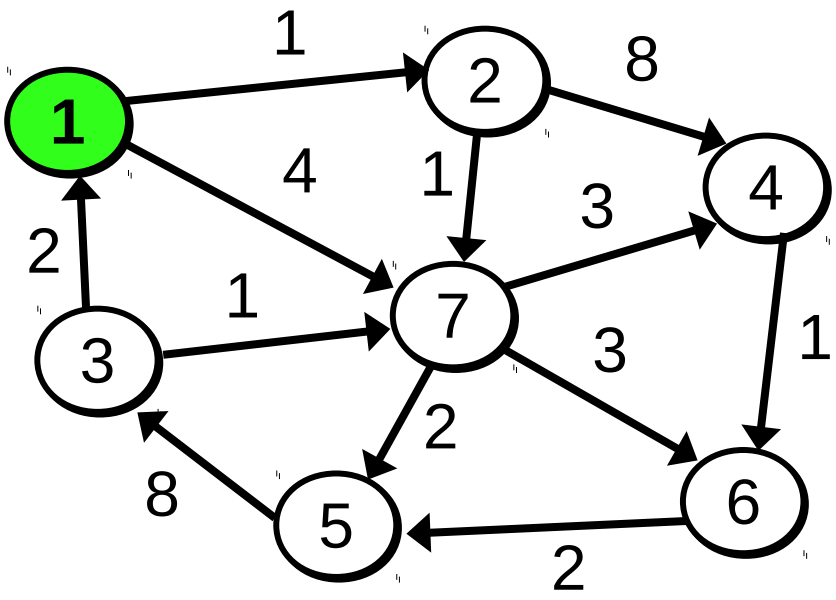
Por tanto el algoritmos de Dijkstra hubiese seleccionado y en lugar de v.

Análisis de Eficiencia

- EL análisis del algoritmo es parecido al realizado para el algoritmo de Prim, deduciendo que el algoritmo es del orden O(A logV).
  - Demostrarlo queda como ejercicio.

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**?

Índice

- EL ENFOQUE GREEDY
- ALGORITMOS GREEDY EN GRAFOS
-
- HEURÍSTICA GREEDY
  - Introducción a la Heurística Greedy
  - El Problema de Coloreo de un Grafo
  - Problema del Viajante de Comercio
  - Problema de la Mochila

Heurísticas Greedy

SITUACIÓN QUE NOS PODEMOS ENCONTRAR

- Hay casos en los cuales no se puede conseguir una algoritmo voraz para que el que se pueda demostrar que encuentra la solución óptima

- Existen para distintos problemas NP-completos

Heurísticas

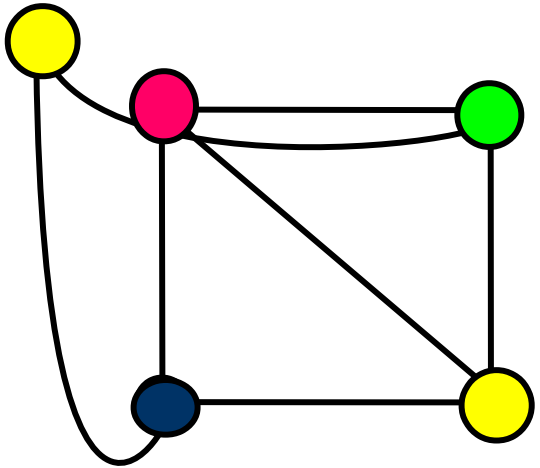
- Heurística: Son procedimientos que, basados en la experiencia, proporcionan buenas soluciones a problemas concretos
- Metaheurísticas de propósito general:
  - Enfriamiento Simulado, Busqueda Tabu,
  - GRASP (Greedy Randomized Adaptive Search Procedures), Busqueda Dispersa, Busqueda por Entornos Variables, Busqueda Local Guiada,
  - Computación Evolutiva (Algoritmos Genéticos, ...), Algoritmos Memeticos, 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 minimo numero de colores que se necesitan para colorear todos sus vertices, y que no haya dos de ellos adyacentes pintados con el mismo color
- Si el grafo no es plano puede requerir tantos colores como vertices haya
- Las aplicaciones son muchas
  - Representación de mapas
  - Diseño de paginas 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 vertices adyacentes con el mismo color

El Problema del Coloreo de un Grafo

- Suponemos que tenemos una paleta de colores (con mas 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

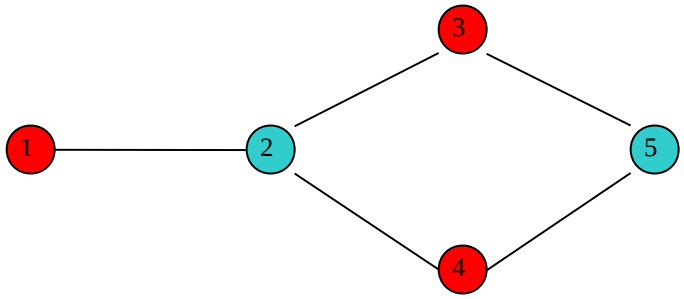
Marcar v como coloreado

Añadir v a NuevoColor

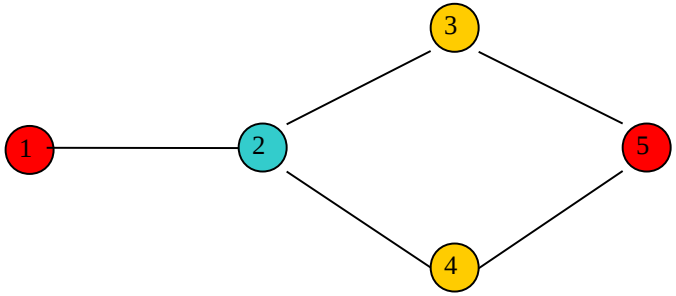
End

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

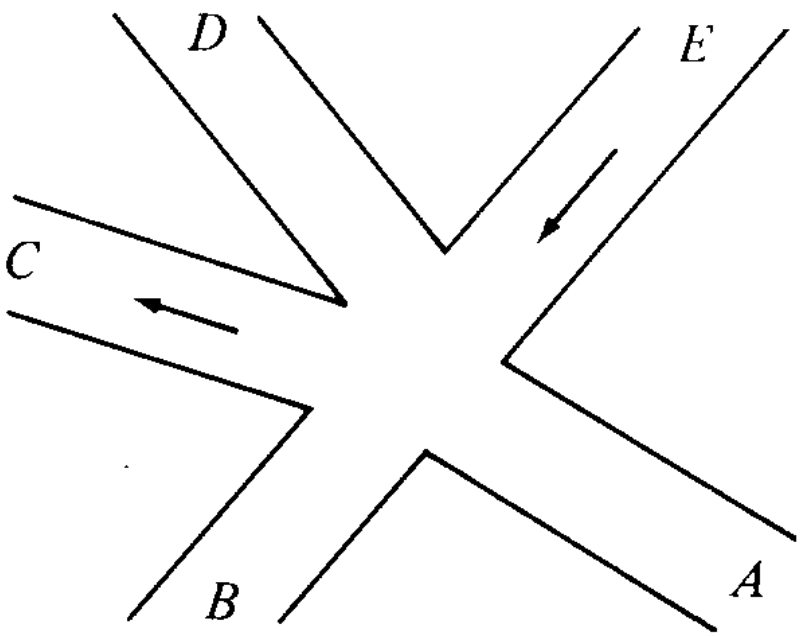
Ejemplo



El orden en el que se escogen los vertices 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 semaforos



- A la izquierda tenemos un cruce de calles
  - Se señalan los sentidos de circulacion.
  - La falta de flechas, significa que podamos ir en las dos direcciones.
  - Queremos diseñar un patron de semaforos con el minimo numero de semaforos, lo que
  - Ahorrara tiempo (de espera) y dinero
  - Suponemos un grafo cuyos vertices representan turnos, y unen esos turnos que no pueden realizarse simultaneamente sin que haya colisiones, y el problema del cruce con semaforos se convierte en un problema de coloreo de los vertices de un grafo

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

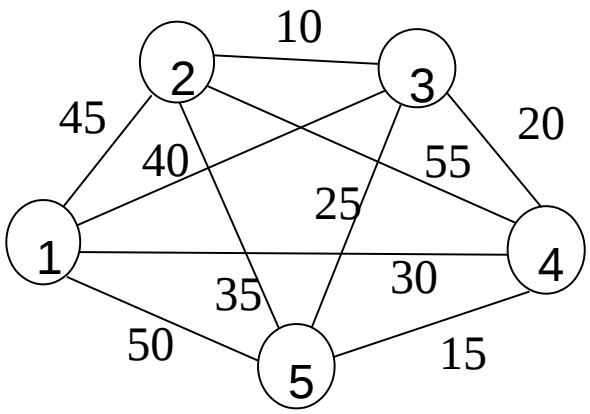
- Supongamos un grafo no dirigido y completo G = (N, A) y L una matriz de distancias no negativas referida a G. Se quiere encontrar un **Circuito Hamiltoniano Minimal**.

- Este es un problema Greedy típico, que presenta las 6 condiciones para poder ser enfocado con un algoritmo greedy

- Destaca de esas 6 caracteristicas **la condición de factibilidad**:
  - 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

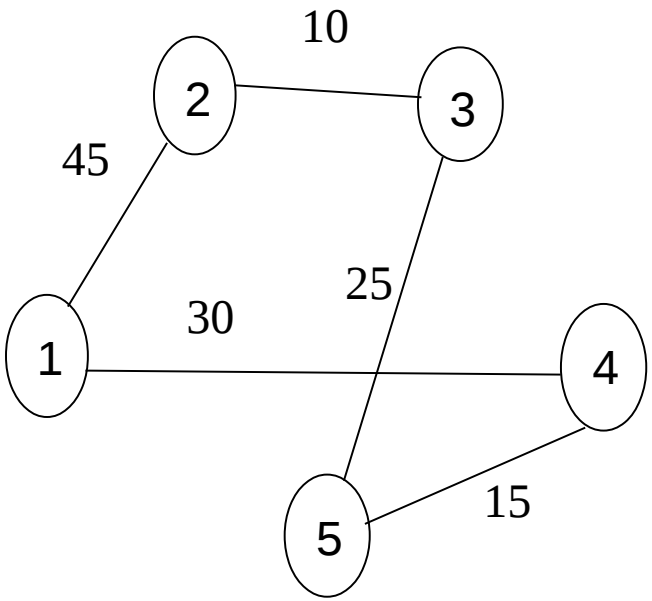
- Consideremos el siguiente grafo



- 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 al final del proceso ciclo.

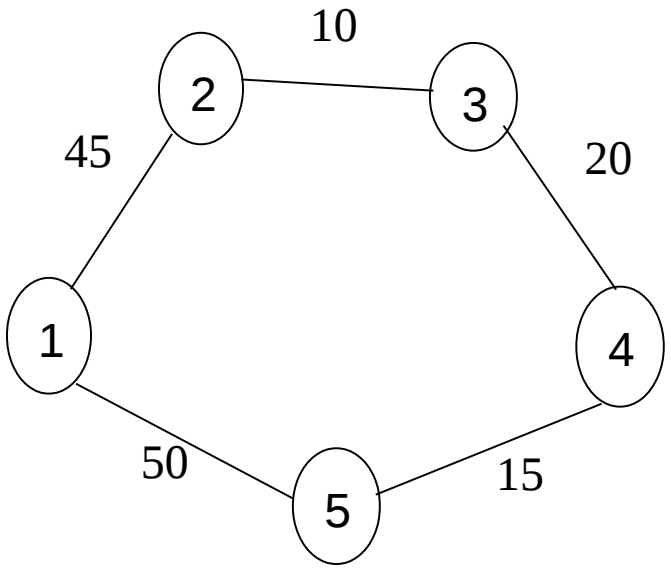
El Problema del Viajante de Comercio

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



Solución: (1, 4, 5 ,3, 2),
125

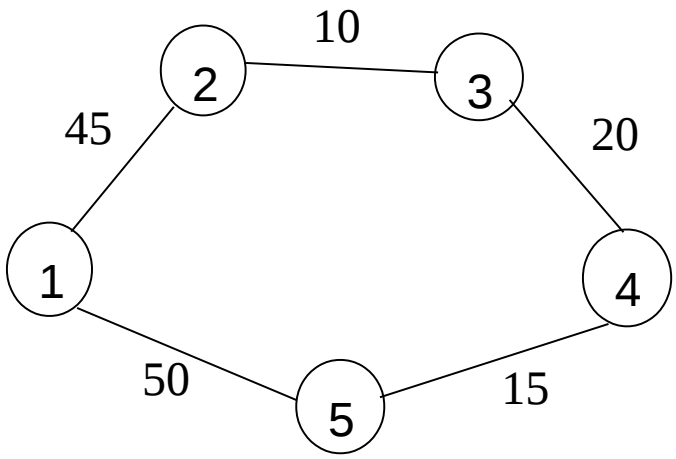
Solución empezando en el nodo 5



Solución: (5,4,3, 2,1),
140

El Problema del Viajante de Comercio

- Solucion 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 la fraccion x_i, $0 \leq x_i \leq 1$, del objeto i, generamos un beneficio de valor $p_i x_i$
- El objetivo es rellenar la mochila de tal manera que se maximice el beneficio que produce el peso total de los objetos que se transportan, con la limitación de la capacidad de valor M

maximizar 

sujeto a 

con $0 \leq x_i \leq 1$, $1 \leq i \leq n$

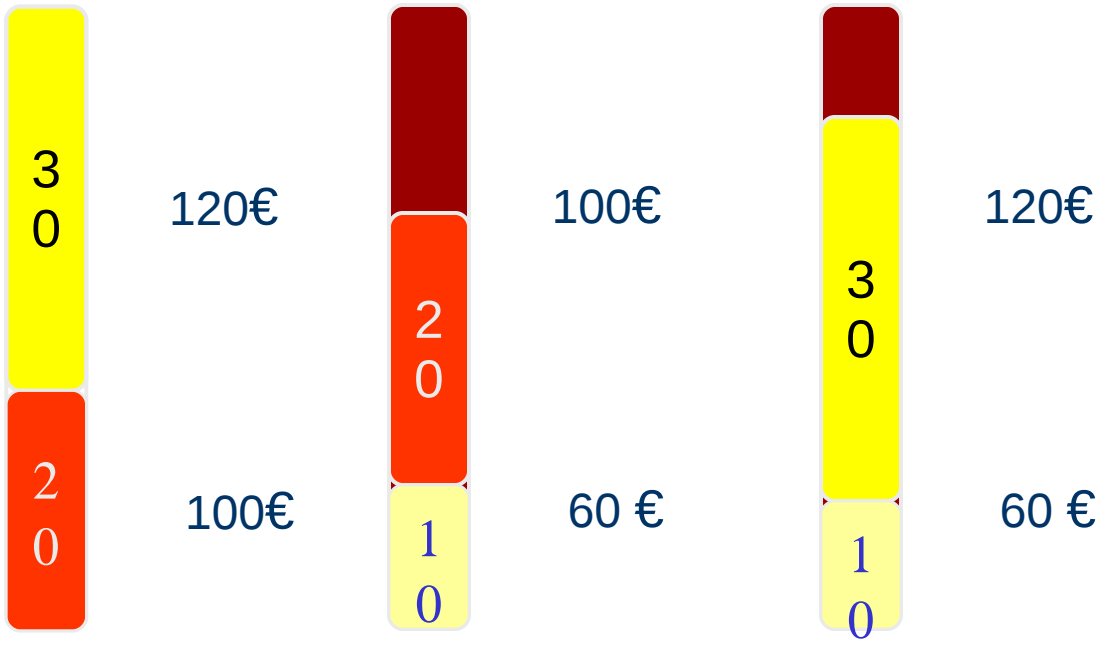
Ejemplo: Mochila 0/1

Es un claro problema de tipo greedy

Sus aplicaciones son inumerables

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

Ejemplo: Mochila 0/1



Total = 220 € = 160 € = 180 €

¿Cómo seleccionamos los items?

Solucion Greedy

- Definimos la densidad del objeto A_i por p_i/w_i.
- Se usan objetos de tan alta densidad como sea posible, es decir, los seleccionaremos en orden decreciente de densidad.
- Primero, se ordenan los objetos por densidad no creciente, i.e.:

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

PseudoCodigo

```
Procedimiento MOCHILA_GREEDY(P,W,M,X,n)
//P(1:n) y W(1:n) contienen los costos y pesos respectivos de los n
    objetos ordenados como P(I)/W(I) ≥  P(I+1)/W(I+1). M es la
    capacidad de la mochila y X(1:n) es el vector solucion//
real P(1:n), W(1:n), X(1:n), M, cr;
integer I,n;
    x = 0; //inicializa la solucion en cero //
    cr = M;  // cr = capacidad restante de la mochila //
    Para i = 1 hasta n Hacer
        Si W(i) > cr Entonces exit endif
        X(I) = 1;
        cr =  c - W(i);
    repetir
        End MOCHILA_GREEDY
```

Teoría de Algoritmos

Tema 1. Planteamiento General

Tema 2. La Eficiencia de los Algoritmos

Tema 3. Algoritmos "Divide y Vencerás"

Tema 4. Algoritmos Voraces ("Greedy")

Tema 5. Algoritmos para la Exploración de Grafos ("Backtraking", "Branch and Bound")

Tema 6. Algoritmos basados en Programación Dinámica

Tema 7. Otras Técnicas Algorítmicas de Resolución de Problemas

Teoría de Algoritmos

Objetivos

- Comprender la filosofía de diseño de 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

Algoritmos Greedy (voraz)

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