

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 basados en Programación Dinámica

Tema 6. Algoritmos para la Exploración de Grafos
 (“Backtracking”, “Branch and Bound”)

Tema 7. Otras metodologías algorítmicas

Tema 4: Algoritmos Voraces ("Greedy")

Bibliografía:

G. BRASSARD, P. BRATLEY. Fundamentos de Algoritmia. Prentice Hall (1997).

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

E. HOROWITZ, S. SAHNI, S. RAJASEKARAN. Computer Algorithms. Computer Science Press (1998).

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

- **EL ENFOQUE GREEDY**

- Características Generales
- Elementos de un Algoritmo Voraz
- Esquema Voraz
- Ejemplo: Problema de Selección de Actividades
- Ejemplo: Almacenamiento Optimal en Cintas
- Ejemplo: Problema de la Mochila Fraccional

- **ALGORITMOS GREEDY EN GRAFOS**

- **HEURÍSTICA GREEDY**

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 siempre garantizan alcanzar la solución óptima.

Características generales de los algoritmos voraces



¡Comete siempre todo lo que tengas a mano!

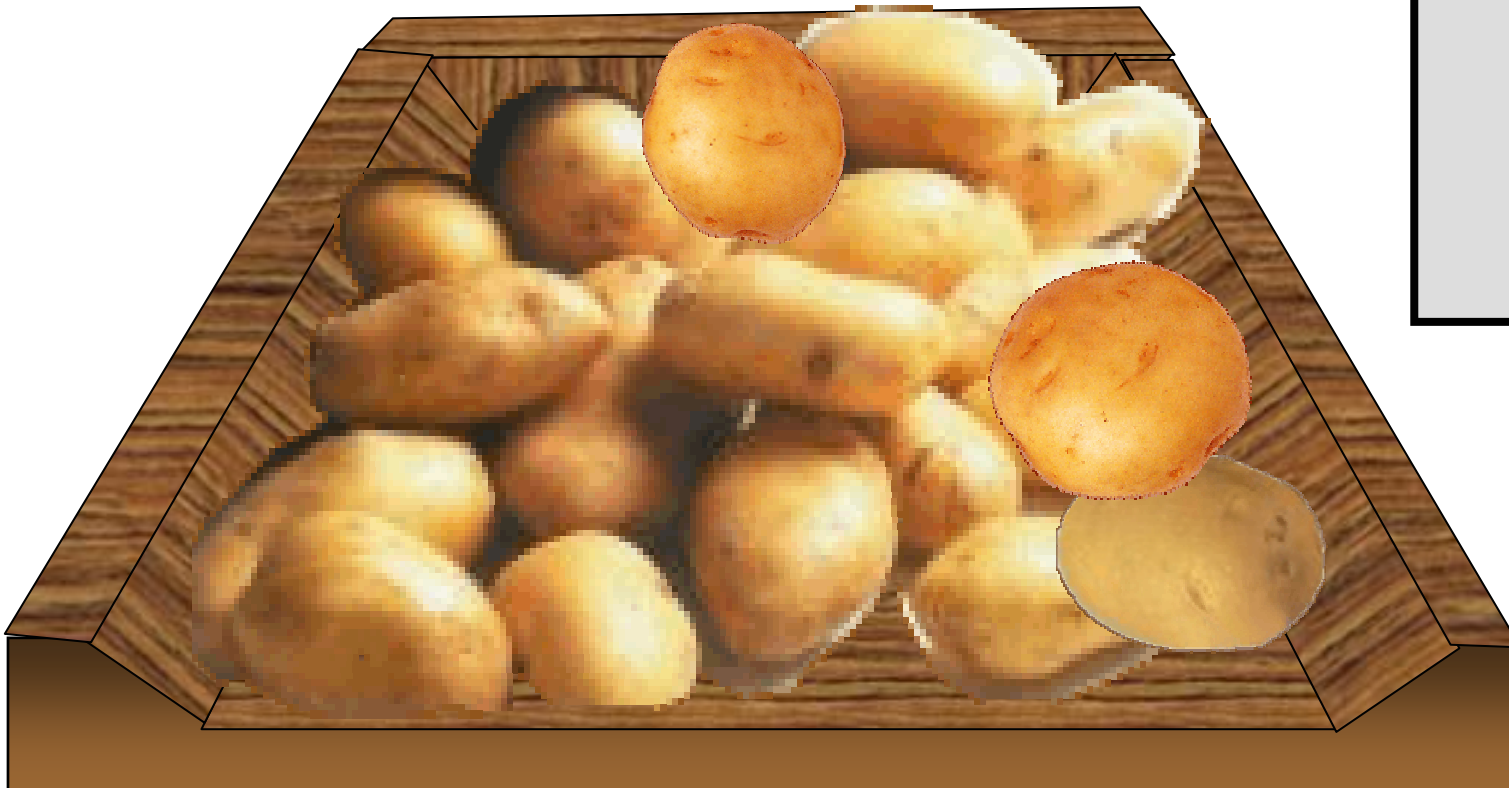
El termino **greedy** es sinónimo de voraz, ávido, glotón, ...

Características generales

- Como decíamos, los algoritmos **voraces**, **ávidos** o de **avance rápido** (en inglés **greedy**) se utilizan normalmente en problemas de optimización.
 - El problema se ***interpreta*** como: “tomar algunos elementos de entre un conjunto de candidatos”.
 - El **orden** el que se cogen puede ser importante o no.
- Un **algoritmo voraz** funciona por pasos:
 - Inicialmente partimos de una solución vacía.
 - En cada paso se escoge el siguiente elemento para añadir a la solución, entre los candidatos.
 - Una vez tomada esta decisión no se podrá deshacer.
 - El algoritmo acabará cuando el conjunto de elementos seleccionados constituya una solución.

Características generales

- **Ejemplo:** “el viejo algoritmo de comprar patatas en el mercado”.
¿Sí o no?



2 Kg.

Características generales

Características básicas del algoritmo:

- Inicialmente empezamos con una solución “vacía”, sin patatas.
- Función de selección: seleccionar la mejor patata del montón o la que “parezca” que es la mejor.
- Examinar la patata detenidamente y decidir si se coge o no.
- Si no se coge, se aparta del montón.
- Si se coge, se mete a la bolsa (y ya no se saca).
- Una vez que tenemos 2 kilos paramos.

Características generales

- Se puede generalizar el proceso intuitivo a un esquema algorítmico general.
- El esquema trabaja con los siguientes conjuntos de elementos:
 - **C**: Conjunto de elementos **candidatos**, **pendientes** de seleccionar (inicialmente todos).
 - **S**: Candidatos seleccionados para la **solución**.
 - **R**: Candidatos seleccionados pero **rechazados** después.
- ¿Qué o cuáles son los candidatos? Depende de cada problema.

Características generales

- **Esquema general de un algoritmo voraz:**
voraz (C: CjtoCandidatos; var S: CjtoSolución)
 $S := \emptyset$
 mientras $(C \neq \emptyset)$ **Y NO solución**(S) **hacer**
 $x := \text{seleccionar}(C)$
 $C := C - \{x\}$
 si **factible**(S, x) **entonces**
 insertar(S, x)
 finsi
 finmientras
 si **NO solución**(S) **entonces**
 devolver “No se puede encontrar solución”
 finsi

Características generales

Funciones genéricas

- **solución (S)**. Comprueba si un conjunto de candidatos es una solución (independientemente de que sea óptima o no).
- **seleccionar (C)**. Devuelve el elemento más “prometedor” del conjunto de candidatos pendientes (no seleccionados ni rechazados).
- **factible (S, x)**. Indica si a partir del conjunto **S** y añadiendo **x**, es posible construir una solución (posiblemente añadiendo otros elementos).
- **insertar (S, x)**. Añade el elemento **x** al conjunto solución. Además, puede ser necesario hacer otras cosas.
- Función **objetivo (S)**. Dada una solución devuelve el coste asociado a la misma (resultado del problema de optimización).

Características generales

- El orden de complejidad depende del número de candidatos, de las funciones básicas a utilizar, del número de elementos de la solución.
- n : número de elementos de una solución.
- Repetir, como mínimo m :
 - Comprobar si el valor es correcto: $f(m)$. Normalmente $O(1)$.
 - Selección de un elemento entre los candidatos: $g(n)$. Entre $O(1)$ y $O(n)$.
 - La función **factible** es parecida a **solucion**, pero con una solución parcial $h(n)$.
 - La unión de un nuevo elemento a la solución puede requerir otras operaciones de cálculo, $j(n, m)$.

**¡EH! ¿QUE SE VA
SIN PAGAR LAS
PATATAS!!**

¡....!

Características generales

SON 1,11 EUROS

AHÍ VAN 5 EUROS

TOMA EL CAMBIO,
3,89 EUROS

- **Problema del cambio de monedas.**

Construir un algoritmo que dada una cantidad **P** devuelva esa cantidad usando el menor número posible de monedas.

Disponemos de monedas con valores de 1, 2, 5, 10, 20 y 50 céntimos de euro, 1 y 2 euros (€).



Características generales

- **Caso 1. Devolver 3,89 Euros.**

1 moneda de 2€, 1 moneda de 1€, 1 moneda de 50 c€, 1 moneda de 20 c€, 1 moneda de 10 c€, 1 moneda de 5 c€ y 2 monedas de 2 c€. Total: 8 monedas.



- El método intuitivo se puede entender como un **algoritmo voraz**: en cada paso añadir una moneda nueva a la solución actual, hasta llegar a **P**.

Características generales

Problema del cambio de monedas

- **Conjunto de candidatos:** todos los tipos de monedas disponibles. Supondremos una cantidad ilimitada de cada tipo.
- **Solución:** conjunto de monedas que sumen **P**.
- **Función objetivo:** minimizar el número de monedas.

Representación de la solución:

- $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$, donde x_i es el número de monedas usadas de tipo i .
- Suponemos que la moneda i vale c_i .
- **Formulación:** Minimizar $\sum_{i=1..8} x_i$, sujeto a $\sum_{i=1..8} x_i \cdot c_i = P$, $x_i \geq 0$

Características generales

Funciones del esquema:

- **inicialización.** Inicialmente $x_i = 0$, para todo $i = 1..8$
- **solución.** El valor actual es solución si $\sum x_i \cdot c_i = P$
- **seleccionar.** ¿Qué moneda se elige en cada paso de entre los candidatos?
- **Respuesta:** elegir en cada paso la moneda de valor más alto posible, pero sin sobrepasar la cantidad que queda por devolver.
- **factible.** Valdrá siempre verdad.
- En lugar de seleccionar monedas de una en una, usamos la división entera y cogemos todas las monedas posibles de mayor valor.

Características generales

- **Implementación.** Usamos una variable local **act** para acumular la cantidad devuelta hasta este punto.
- Suponemos que las monedas están ordenadas de menor a mayor valor.

**DevolverCambio (P: entero; C: array [1..n] de entero;
var X: array [1..n] de entero)**

act:= 0

j:= n

para i:= 1,...,n **hacer**

X[i]:= 0

mientras act ≠ P **hacer**

mientras (C[j] > (P - act)) AND (j > 0) **hacer** j:= j - 1

si j==0 **entonces devolver** "No existe solución"

X[j]:= ⌊(P - act) / C[j]⌋

act:= act + C[j]*X[j]

finmientras

inicialización

no solución(X)

seleccionar(C,P,X)

no factible(j)

insertar(X,j)

Características generales

- ¿Cuál es el orden de complejidad del algoritmo?
- ¿Garantiza siempre la solución óptima?
- Para este sistema monetario sí. Pero no siempre...

- **Ejemplo.** Supongamos que tenemos monedas de 100, 90 y 1. Queremos devolver 180.



- **Algoritmo voraz.** 1 moneda de 100 y 80 monedas de 1: total 81 monedas.
- **Solución óptima.** 2 monedas de 90: total 2 monedas.

Características generales

- El orden de complejidad depende de:
 - El número de candidatos existentes.
 - Los tiempos de las funciones básicas utilizadas.
 - El número de elementos de la solución.
 - ...
- **Ejemplo.** **n**: número de elementos de C. **m**: número de elementos de una solución.
- Repetir, como máximo **n** veces y como mínimo **m**:
 - Función solución: **f(m)**. Normalmente $O(1)$ ó $O(m)$.
 - Función de selección: **g(n)**. Entre $O(1)$ y $O(n)$.
 - Función **factible** (parecida a **solución**, pero con una solución parcial): **h(m)**.
 - Inserción de un elemento: **j(n, m)**.

Características generales

- Tiempo de ejecución **genérico**:
 $t(n,m) \in O(n^*(f(m)+g(n)+h(m)) + m*j(n, m))$
- Ejemplos:
 - Algoritmo de Dijkstra: **n** candidatos, la función de selección e inserción son $O(n)$: $O(n^2)$.
 - Devolución de monedas: podemos encontrar el siguiente elemento en un tiempo constante (ordenando las monedas): $O(n)$.
- El análisis depende de cada algoritmo concreto.
- En la práctica los algoritmos voraces **suelen ser bastante rápidos**, encontrándose dentro de órdenes de complejidad **polinomiales**.

Elementos de un algoritmo voraz

Para poder resolver un problema con un enfoque greedy, ha de reunir las 6 siguientes características, que no son necesarias, pero si suficientes:

1. Conjunto de *candidatos a seleccionar*.
2. Conjunto de *candidatos seleccionados*
3. ***Función Solución***: determina si los candidatos seleccionados han alcanzado una solución.

Elementos de un algoritmo voraz

4. ***Función de Factibilidad***: determina si es posible completar el conjunto de candidatos seleccionados con el siguiente elemento seleccionado para alcanzar una solución al problema.
5. ***Función Selección***: determina el mejor candidato del conjunto a seleccionar.
6. ***Función Objetivo***: da el valor de la solución alcanzada.

Esquema de un algoritmo voraz

Un algoritmo Greedy procede siempre de la siguiente manera:

- **Se parte de un conjunto de candidatos seleccionados vacío: $S = \emptyset$**
- **De la lista de candidatos C 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 se añade y se borra de C**
- **Si el candidato anterior no es válido, lo borramos de la lista de candidatos posibles, y nunca mas es considerado**
- **Utilizamos la función solución. Si todavía no tenemos una solución 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

$S = \emptyset$

mientras $C \neq \emptyset$ y no Solución(S) **hacer**

$x = \text{Seleccion}(C)$

$C = C - \{x\}$

si factible($S \cup \{x\}$) **entonces**

$S = S \cup \{x\}$

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

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 C de n actividades

s_i = tiempo de comienzo de la actividad i

f_i = 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, C*
- *Candidatos seleccionados: Conjunto S , inic. $S = \{\emptyset\}$*
- *Función Solución: $C = \{\emptyset\}$.*
- *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 S .*
- *Función Objetivo: Tamaño de S .*

Problema Selección de Actividades

Algoritmo Greedy

- *SelecciónActividades(Activ C , S)*
 - *Ordenar C 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 C este vacio.;*

Problema Selección de Actividades

SelecciónActividadesGreedy(C, S)

```
{ qsort(C,n); // según tiempo de finalización
S[0]= C[0] // Seleccionar primera actividad
i=1; prev = 1;
while (!C.empty()) { // es solucion(C)
    x = C[i] // seleccionar;
    if (x.inicio > S[prev-1].fin) //factible x
        S[prev++] = x; // insertamos en solucion
    i++; // miramos siguiente
}
}
```

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

- Tenemos n programas que hay que almacenar 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),

$$(1/n) \sum_{1 \leq j \leq n} t_j \quad t_j = \sum_{1 \leq k \leq j} l_{i_k}$$

- 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

$$D(I) = \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq j} l_{i_k}$$

Ejemplo: Almacenamiento Optimal en Cintas

El problema se puede resolver mediante la técnica greedy

Sea $n = 3$ y $(l_1, l_2, l_3) = (5, 10, 3)$

$((\text{espera1}) + (\text{espera1} + \text{espera2}) + (\text{espera1} + \text{espera2} + \text{espera3}))$

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 $l_1 \leq l_2 \leq \dots \leq l_n$ entonces el orden de colocación $i_j = j$, $1 \leq j \leq n$ minimiza

$$\sum_{k=1}^n \sum_{j=1}^k l_{i_j}$$

sobre todas las posibles permutaciones de i_j

■ *Optimalidad?*

Ver la demostración en Horowitz-Sahni

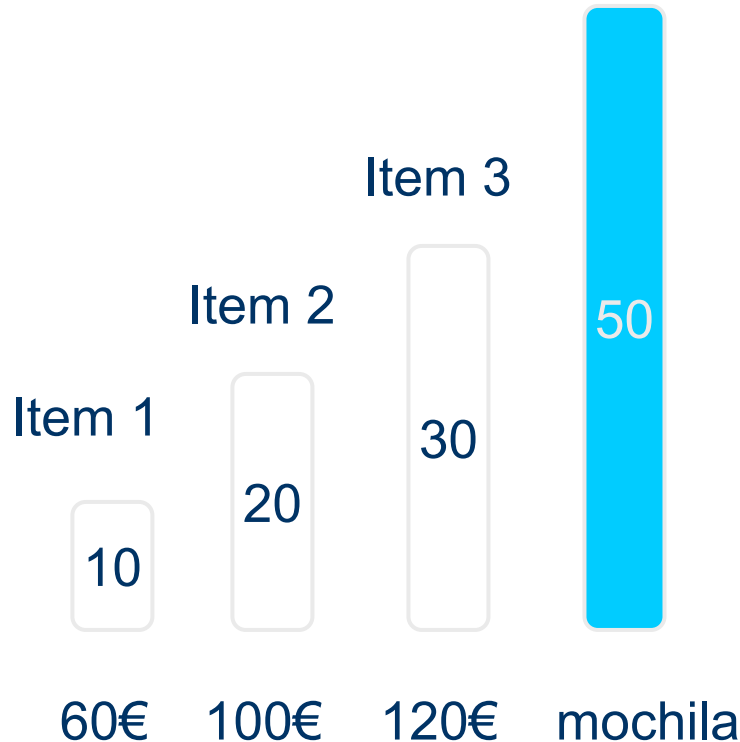
Ejemplo: Problema de la mochila fraccional

- 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.

$$\text{maximizar } \sum_{1 \leq i \leq n} x_i b_i \quad \text{sujeto a } \sum_{1 \leq i \leq n} x_i p_i \leq P$$

$x_i \in [0,1]$ representa la porción del objeto incluida en la mochila

Ejemplo: Mochila 0/1



Es un claro problema de tipo greedy

Sus aplicaciones son innumerables

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

Ejemplo: Mochila 0/1



¿Cómo seleccionamos los items?

Ejemplo: Problema de la mochila fraccional

Ejemplo

p_i	10	20	30	40	50
b_i	20	30	65	40	60

$n= 5, P=100$

Opciones:

¿Poner primero los más pesados?

¿Primero los que tienen más valor?

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**
 - $\text{Peso total} = 10 + 20 + 30 + 40 = 100$
 - $\text{Beneficio total} = 20 + 30 + 65 + 40 = 155$
- **Metodo 2 elegir primero el mas caro**
 - $\text{Peso Total} = 30 + 50 + 20 = 100$
 - $\text{Beneficio Total} = 65 + 60 + 20 = 145$

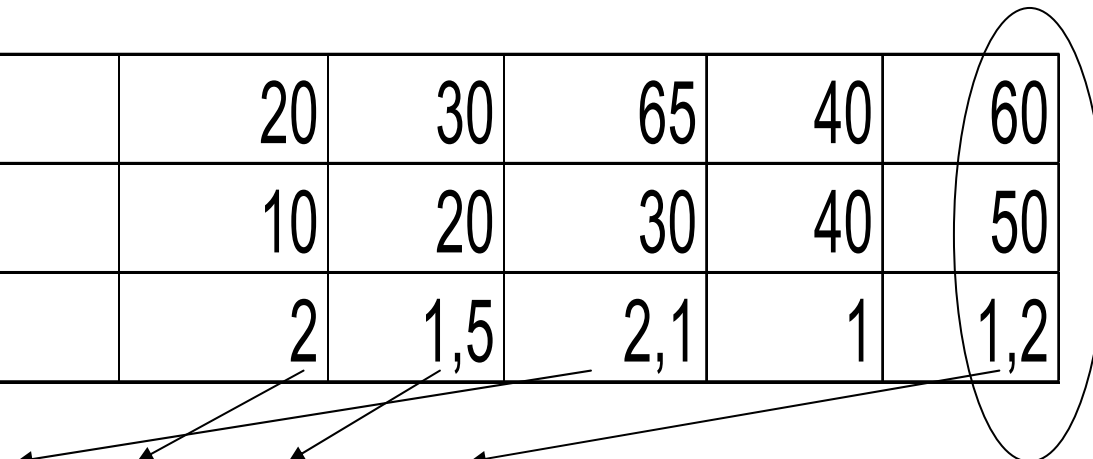
Solucion Greedy

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

Ejemplo: Problema de la mochila fraccional

- Metodo 3 elegir primero el que tenga mayor valor por unidad de peso (razon costo/ 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

A diagram with arrows pointing from the 'Precio/Peso' row of the table to the weights in the 'Peso Total' calculation: 30 (from 2,1), 10 (from 2), 20 (from 1,5), and 40 (from 1). A circle is drawn around the last column of the table, which contains the values 60, 50, and 1,2.

- $\text{Peso Total} = 30 + 10 + 20 + 40 = 100$
- $\text{Costo Total} = 65 + 20 + 30 + 48 = 163$

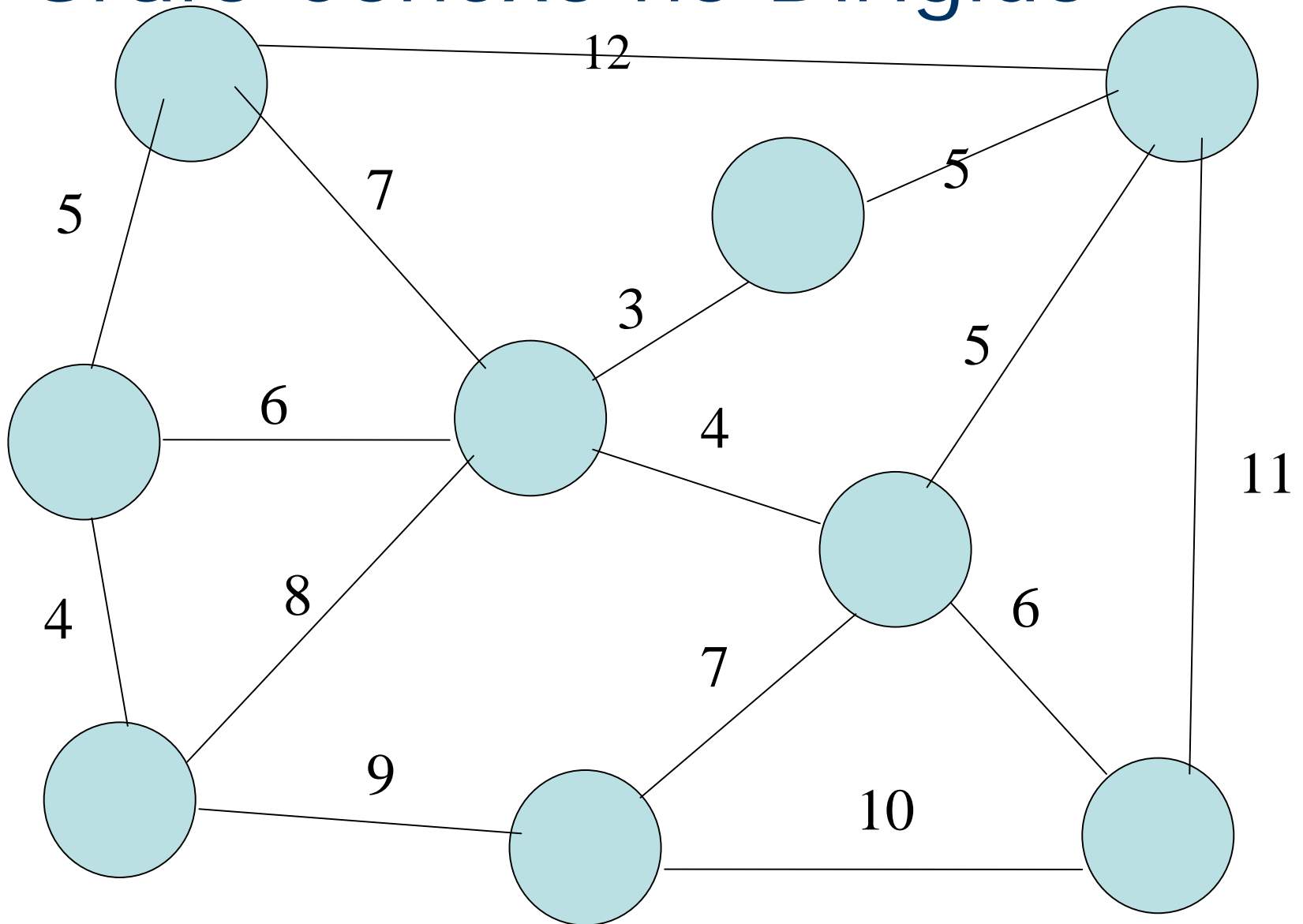
Índice

- **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

Grafo Conexo no Dirigido



Algoritmo de Kruskal

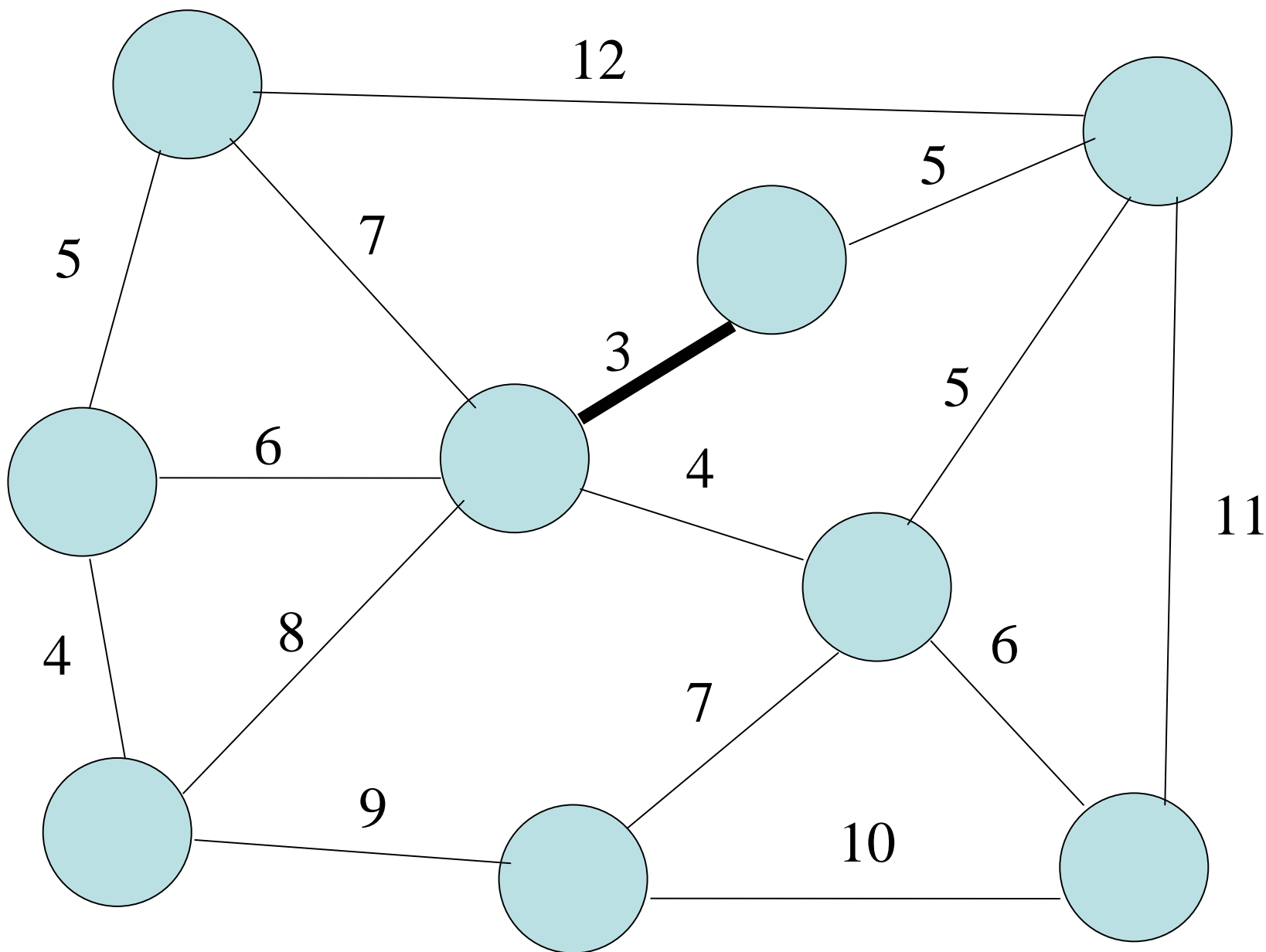
- Conjunto de candidatos: aristas
- **Función Solución:** un conjunto de aristas que conecta todos los vértices

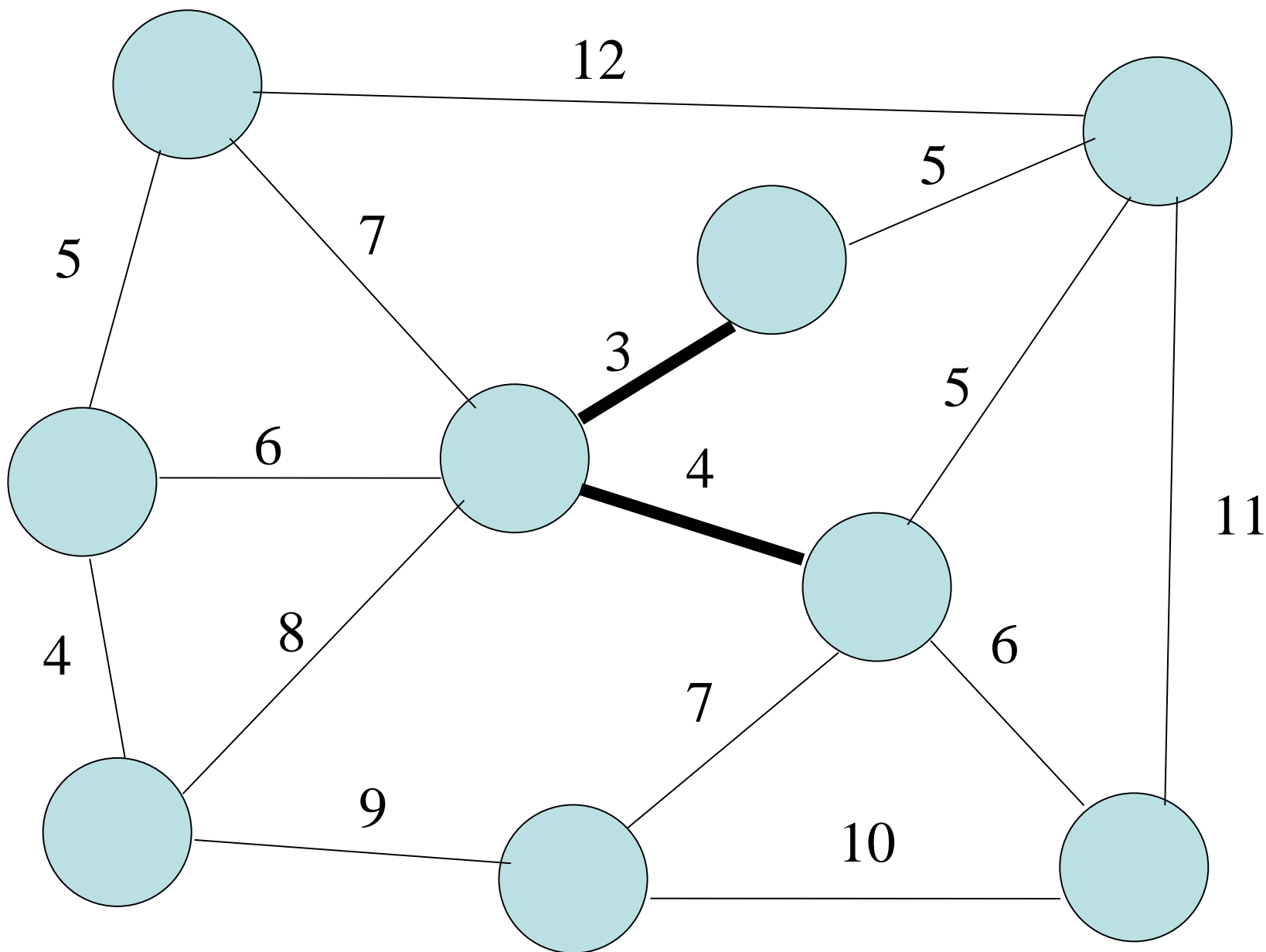
Se ha construido un árbol de recubrimiento ($n-1$ aristas seleccionadas).

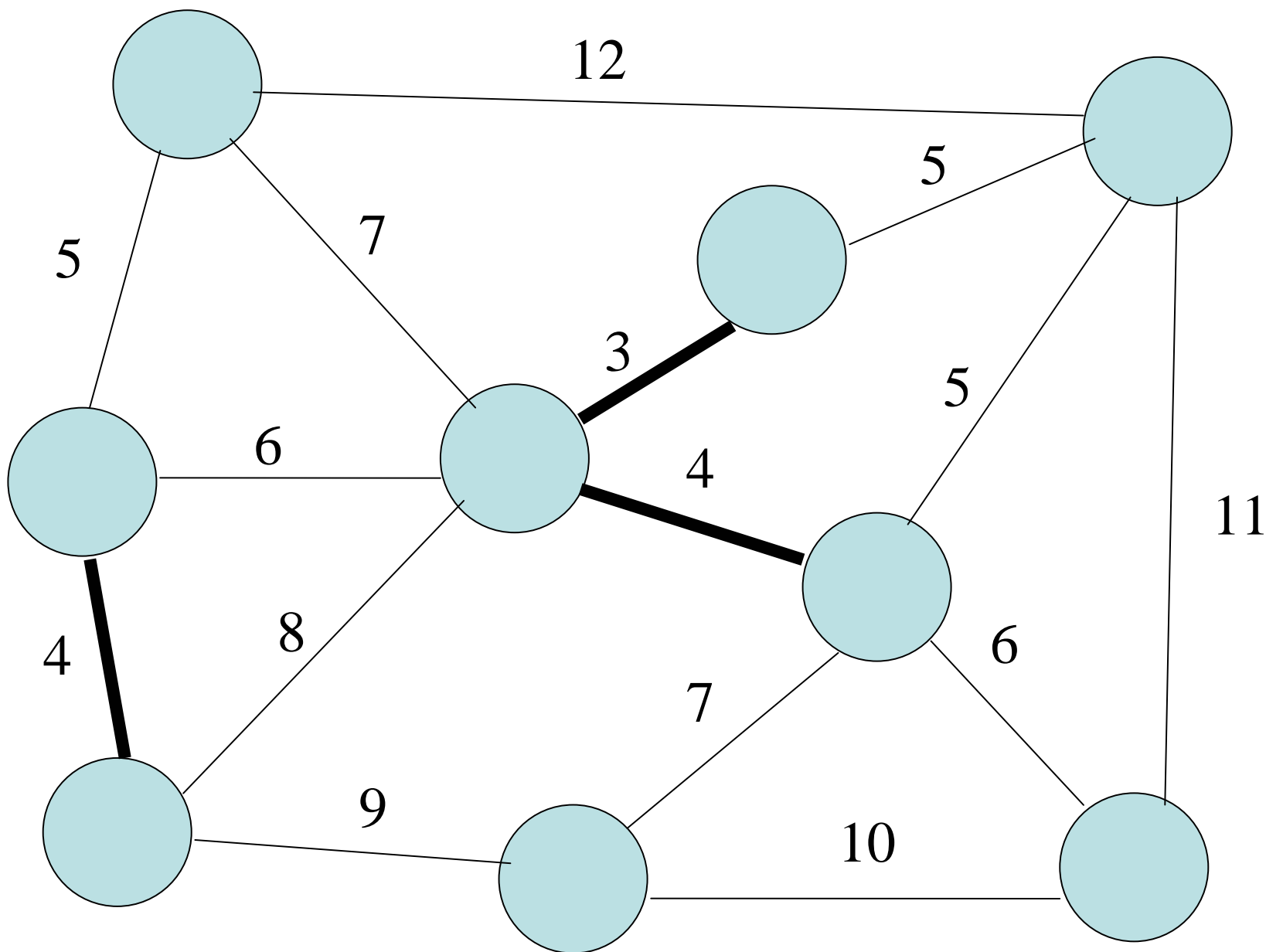
- **Función factible:** el conjunto de aristas no forma ciclos
- **Función selección:** la arista de menor coste
- **Función objetivo:** suma de los costes de las aristas
- ***Conjunto prometededor:*** se puede extender para producir una solución óptima

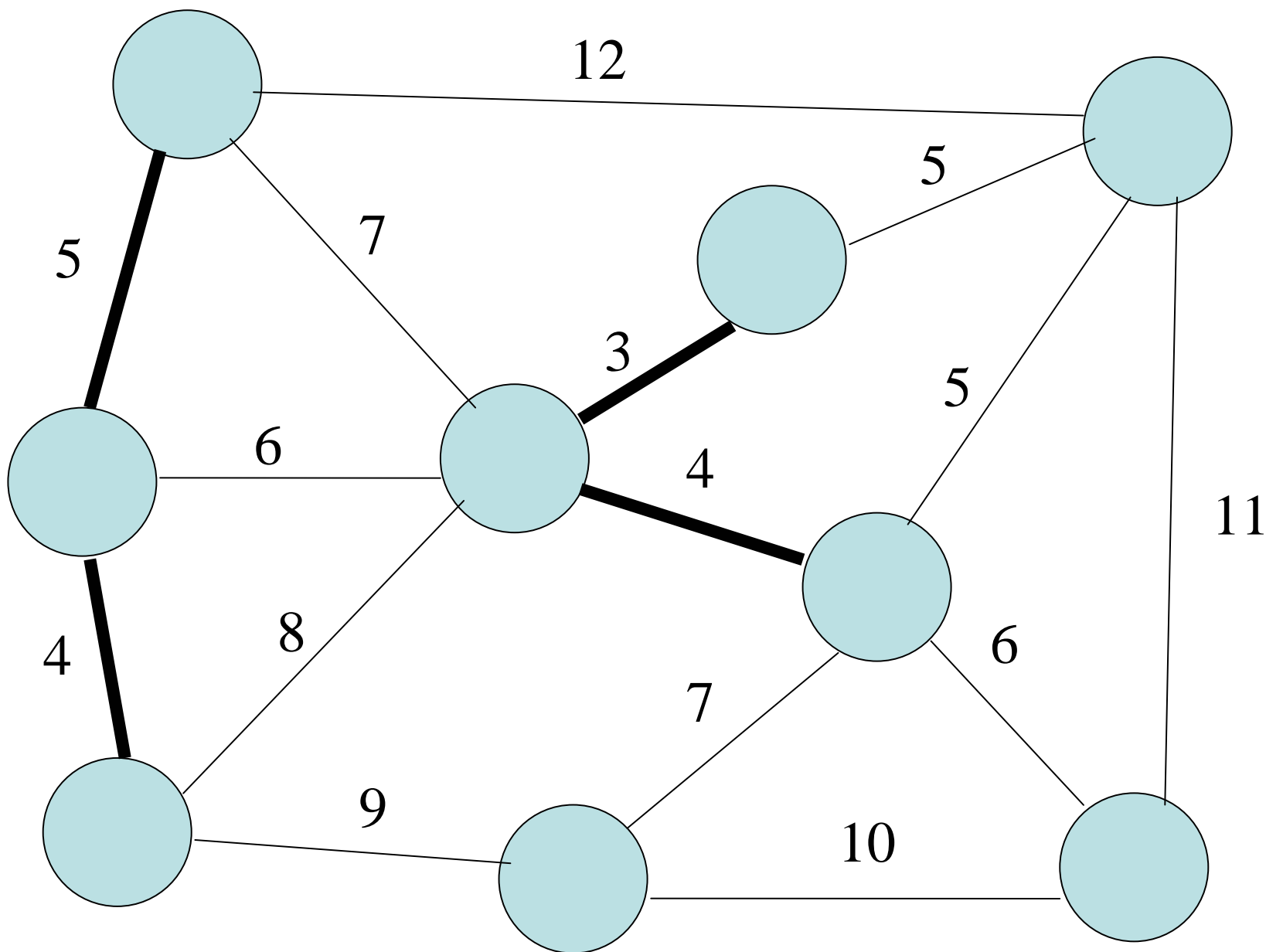
Algoritmo de Kruskal

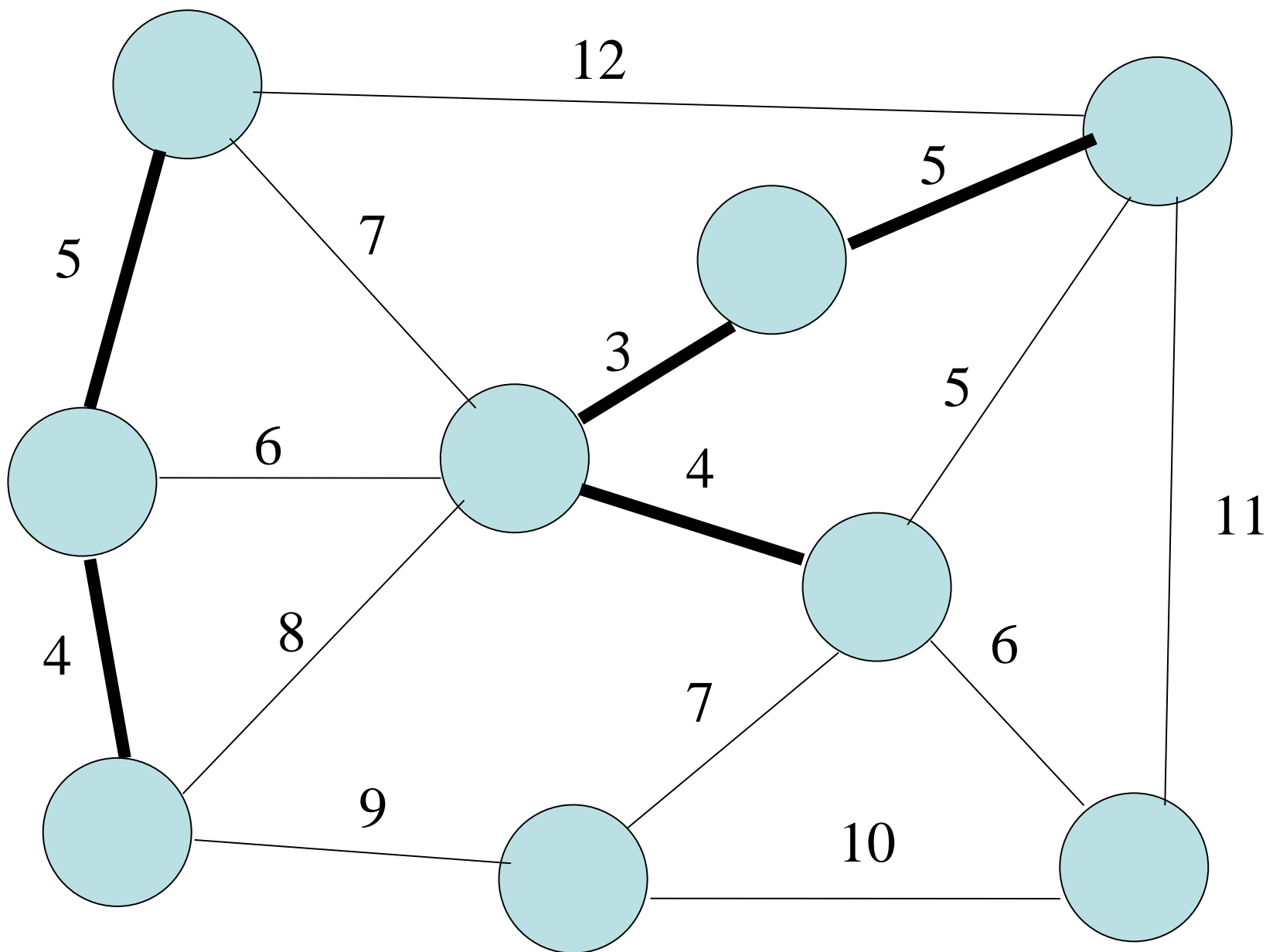
```
Funcion Kruskal_I(Grafo G(V,A))
{set<arcos> C(A);
  set<arcos> S;           // Solución inic. Vacía
  Ordenar(C);
  while (!C.empty() && S.size()!=V.size()-1) { //No solución
    x = C.first(); //seleccionar el menor
    C.erase(x);
    if (!HayCiclo(S,x)) //Factible
      S.insert(x);
  }
  if (S.size()==V.size()-1)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

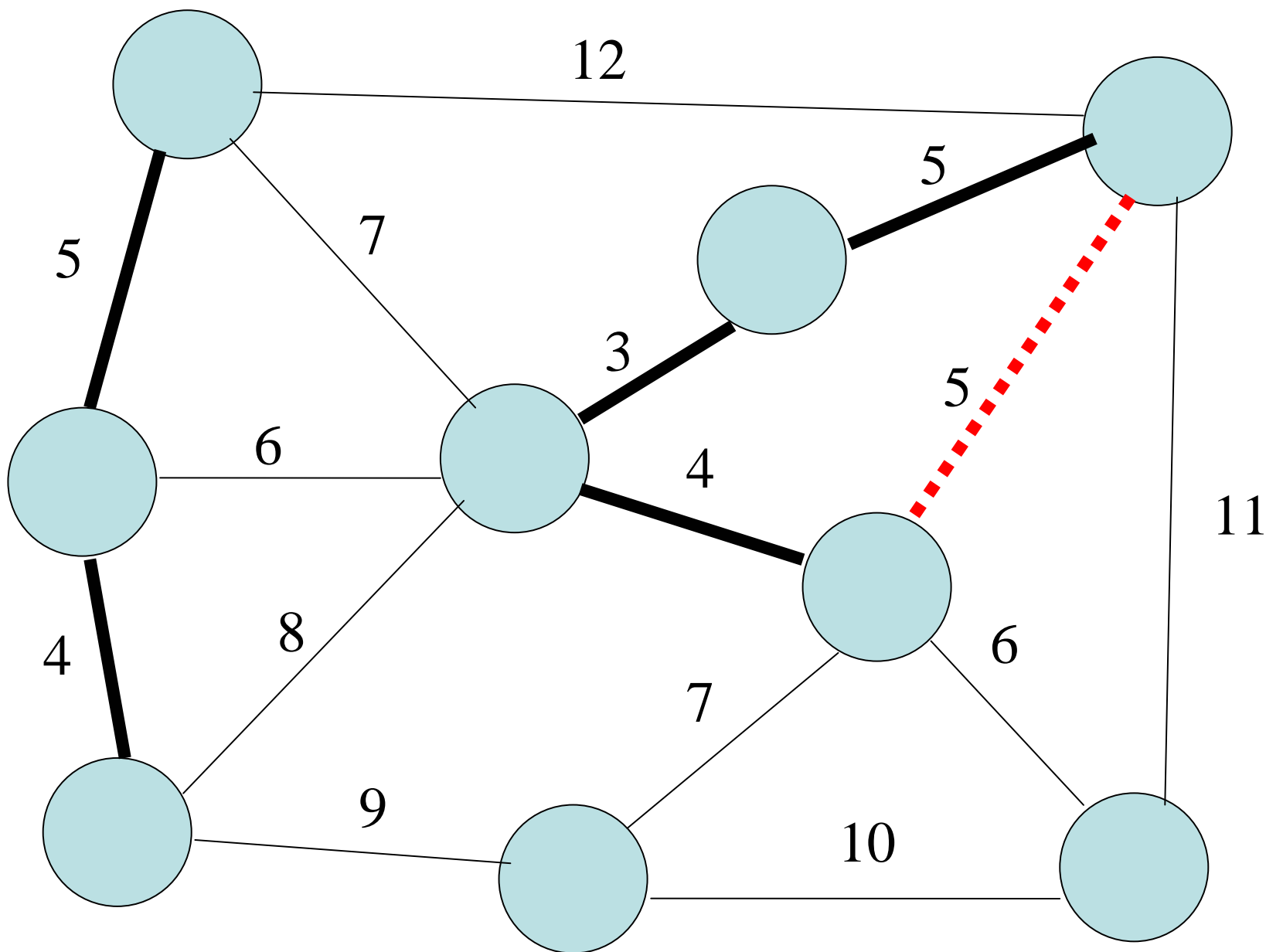



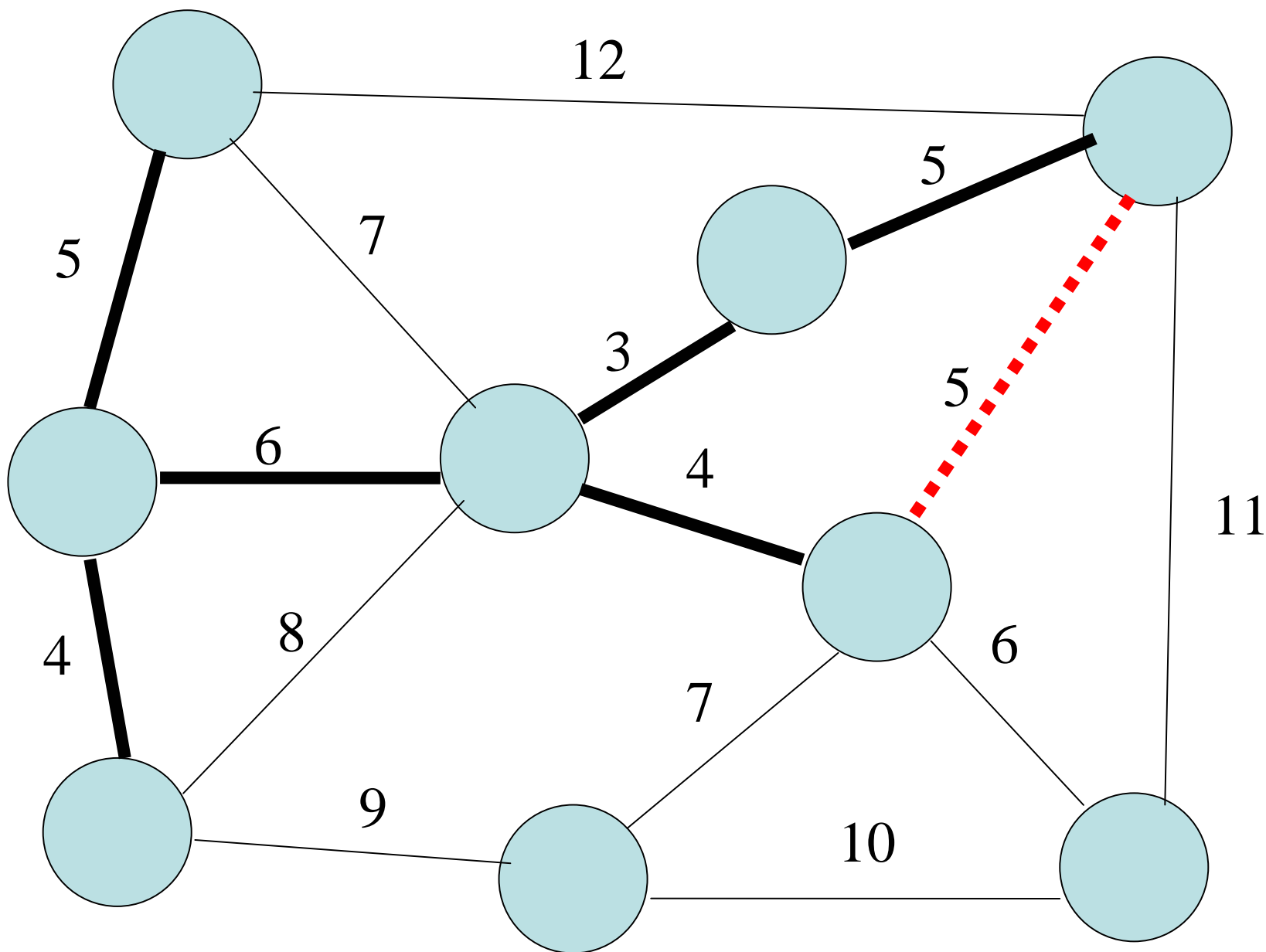


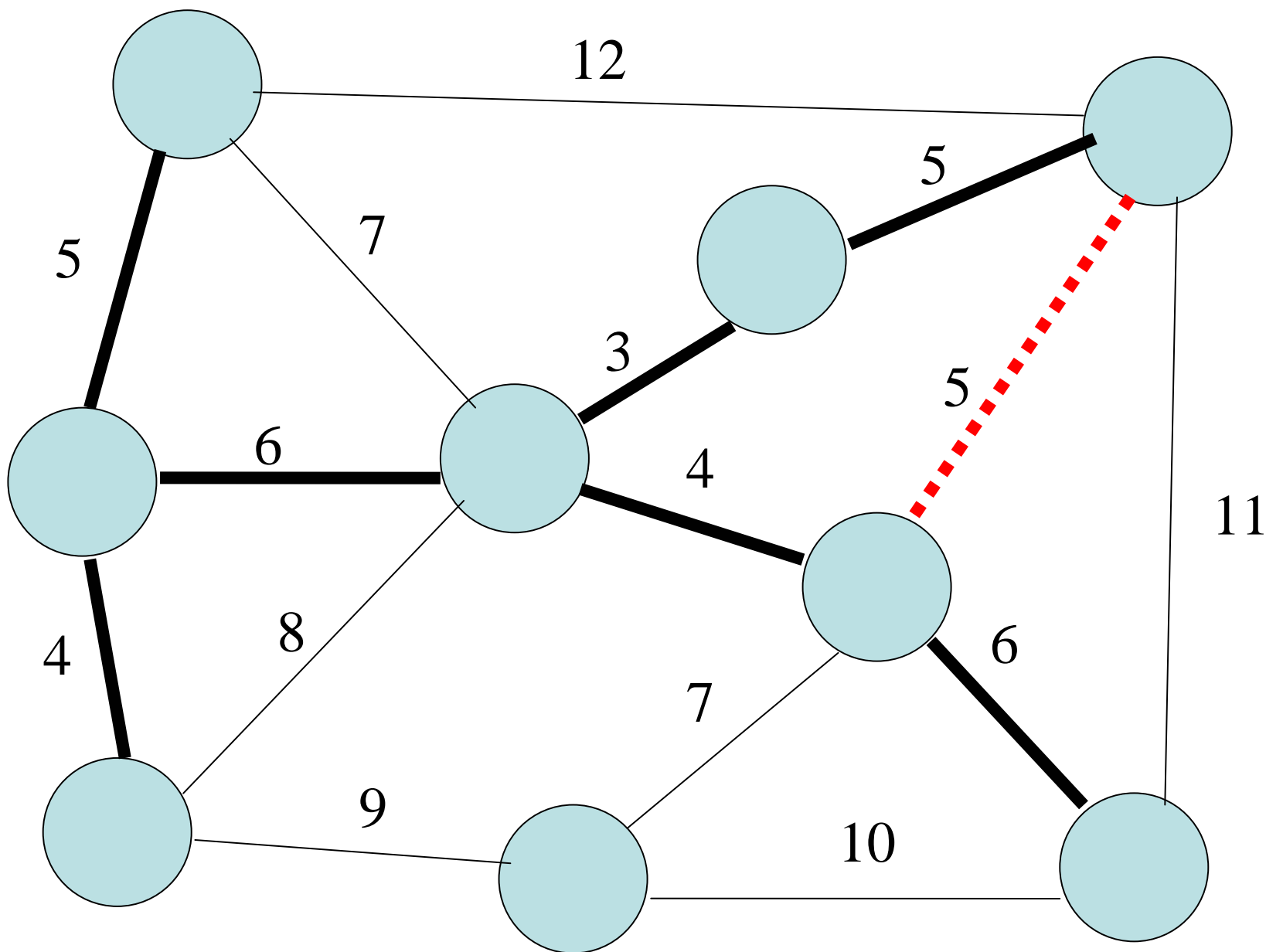


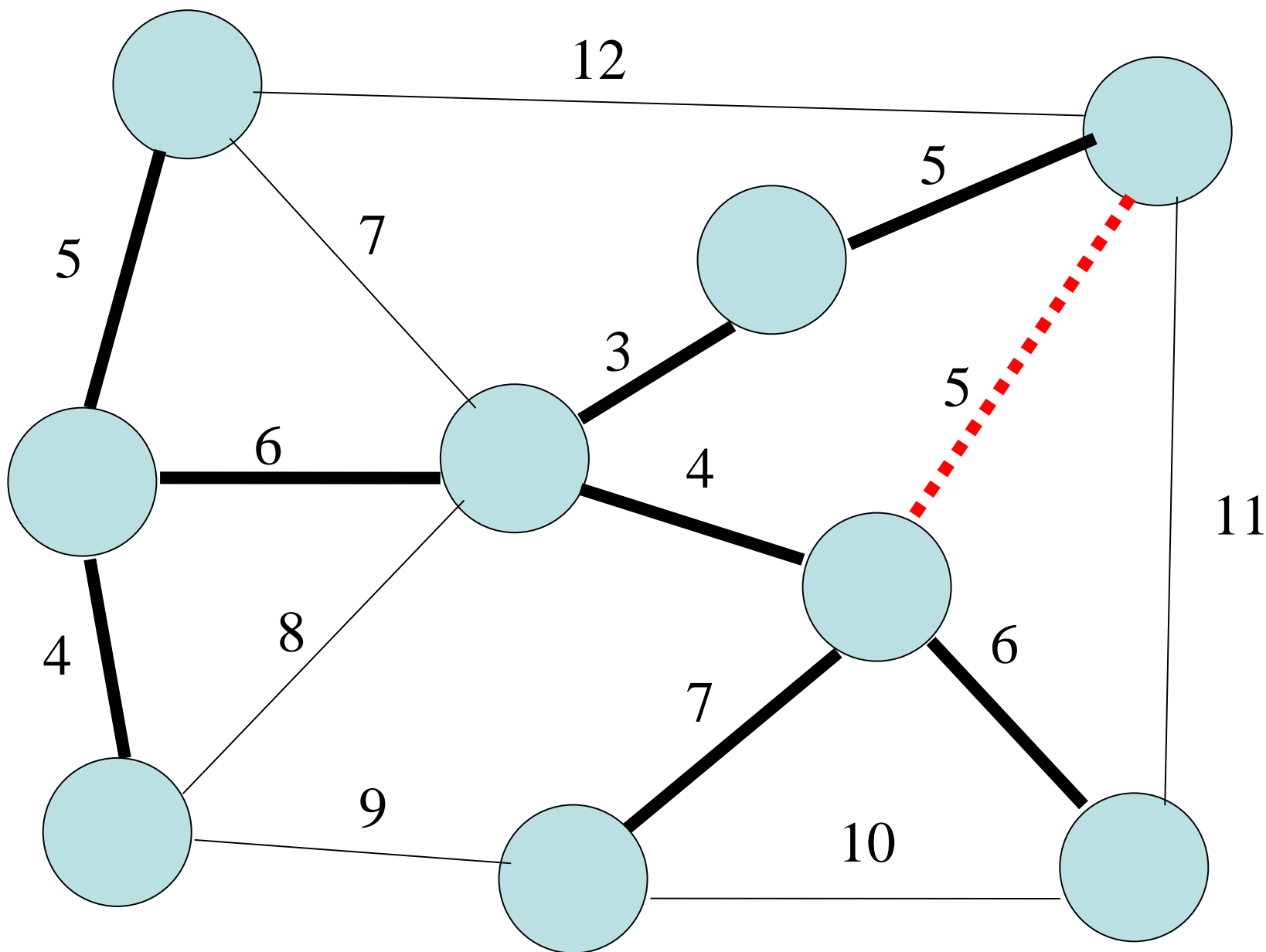


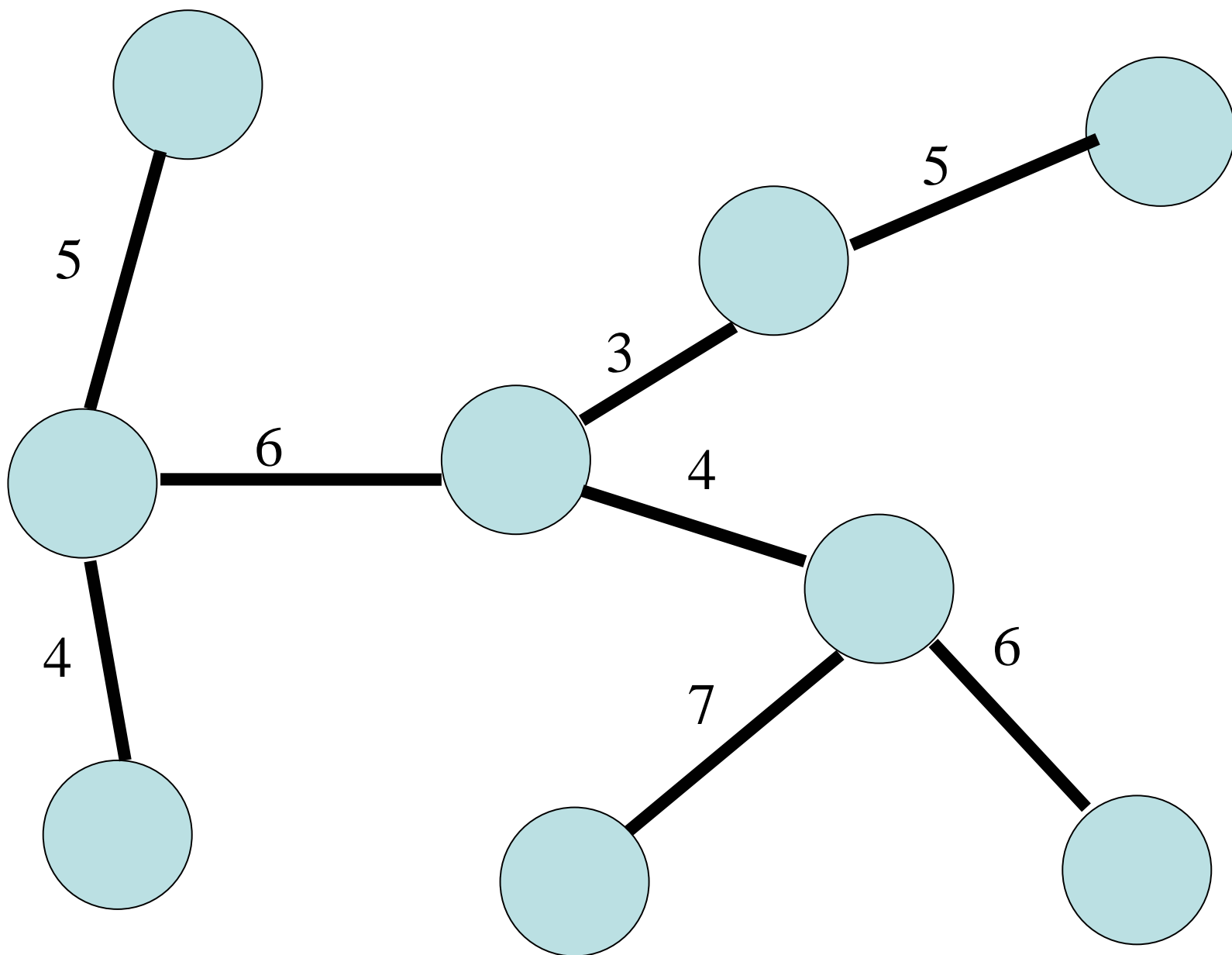












Eficiencia Kruskal (Directa)

Funcion 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);            $O(1)$ 
```

```
    if (!HayCiclo(S,x)) //Factible  $O(1)$ 
```

```
      S.insert(x);          $O(V)$ 
```

```
  }
```

```
  if (S.size() == V.size()-1) return S;    // Hay solución
```

```
  else return "No_hay_solucion";
```

```
}
```

$O(AV)$

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.

Base. Sea k la arista de menor peso en A (la primera propuesta por Kruskal), entonces \exists un ARM optimal T tal que $\{k\} \in T$.

Suponemos cierto para $i-1$.

La arista $(i-1)$ -ésima en incluirse por el algoritmo de Kruskal pertenece a un ARM T optimal. T' con las $i-1$ aristas es un conjunto prometedor

Demostramos que es cierto para i

La arista (i) -ésima incluida por el algoritmo de Kruskal tb. pertenece al ARM T optimal. Llamemos a esta arista i . Según Kruskal es la arista de menor peso que une dos componentes conexas

Optimalidad de Kruskal (Directa)

Demostración

Caso base: Red. Absurdo, supongamos un ARM T que no incluye a k . Consideremos $T \cup k$ con

$$\text{peso}(T \cup k) = \text{peso}(T) + \text{peso}(k).$$

En este caso aparece un ciclo (¿por qué?). Eliminemos cualquier arista del ciclo, (x) , distinta de k . Al eliminar la arista obtenemos un árbol $T^ = T \cup k - x$ con peso*

$$\text{peso}(T^*) = \text{peso}(T) + \text{peso}(k) - \text{peso}(x).$$

Si tenemos $\text{peso}(k) \leq \text{peso}(x)$ entonces deducimos que $\text{peso}(T^) \leq \text{peso}(T)$. !!! Contr. o T^* tb. es óptimo*

Optimalidad de Kruskal (Directa)

Paso Inducción:

Supongamos un ARM T tal que $T' \subseteq T$ y no incluye a i .

Consideremos $T \cup i$ con

$$\text{peso}(T \cup i) = \text{peso}(T) + \text{peso}(i).$$

En este caso aparece un ciclo, que incluirá al menos una arista x que NO pertenece al conjunto de aristas T' seleccionadas por Kruskal (¿por qué?*). Eliminando dicha arista del ciclo.*

obtenemos un nuevo árbol $T^ = T + i - x$ con peso*

$$\text{peso}(T^*) = \text{peso}(T) + \text{peso}(i) - \text{peso}(x).$$

*Si sabemos $\text{peso}(i) \leq \text{peso}(x)$ (*por qué?*) entonces deducimos que $\text{peso}(T^*) \leq \text{peso}(T)$. Puesto que $T' \subseteq T$ y $x \notin T'$ entonces $T' \subseteq T^*$, lo que implica que $T' \cup i \subseteq T^*$ (que es óptimo)*

Algoritmo de Kruskal

¿Cómo determino que una arista no forma un ciclo?

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

La función factible me acepta una arista (la de menor costo) si ésta permite unir 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.

Algoritmo de Kruskal

Voraz($G = (V, A)$; $L: A \rightarrow R$) : conjunto aristas

Ordenar A por longitudes crecientes

$n = |V|$; $T = \emptyset$

Iniciar n conjuntos $comp_i$, cada uno con un elemento v de V

repite

$e = \{u, v\}$

$comp_u, comp_v$ los conjuntos en los que están u, v

si $comp_u \neq comp_v$ **entonces**

fusionar($comp_u, comp_v$)

$T = T \cup \{e\}$

fin si

hasta $|T| = n - 1$

Devolver T

Implementación Kruskal (Eficiente)

```
Funcion Kruskal(G(V,A)){
```

```
    S = 0; // Inicializamos S con el conjunto vacio
```

```
    for (i=0; i< V.size()-1; i++)
```

```
        MakeSet(V[i]); // Conjuntos con el vértice v[i]
```

```
    Ordenar(A) //Orden creciente de pesos
```

```
    while (!A.empty() && S.size()!=V.size()-1) { //No solución
```

```
        (u,v) =A.first(); //seleccionar el menor arco (y eliminar)
```

```
        if (FindSet(u) != FindSet(v))
```

```
            S = S U {{u,v}};
```

```
            Union(u, v); // Unimos los dos conjuntos
```

```
        }
```

```
    }
```

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 V)
```

```
    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)
```

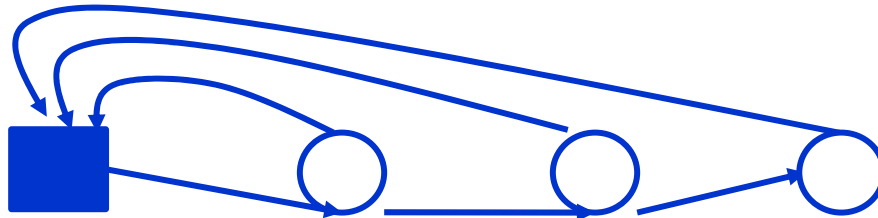
```
            Union(u, v); // O(V) Es el cuello de botella, !!!!
```

```
    }
```

```
}
```

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): "Copia" elementos de A a B haciendo que los elementos de A también apunten a B: $O(A)$
- *¿Cuanto tardan en realizarse las n uniones?*

Unión de conjuntos disjuntos

- Análisis del peor caso: $O(n^2)$

$\text{Union}(S_1, S_2)$	"copia" 1 elemento
$\text{Union}(S_2, S_3)$	"copia" 2 elementos
$\text{Union}(S_{n-1}, S_n)$	"copia" <u> </u> 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 \Rightarrow 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 V)
```

```
    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(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
$$\text{peso}(T) = \text{peso}(u,v) + \text{peso}(T_1) + \text{peso}(T_2)$$
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

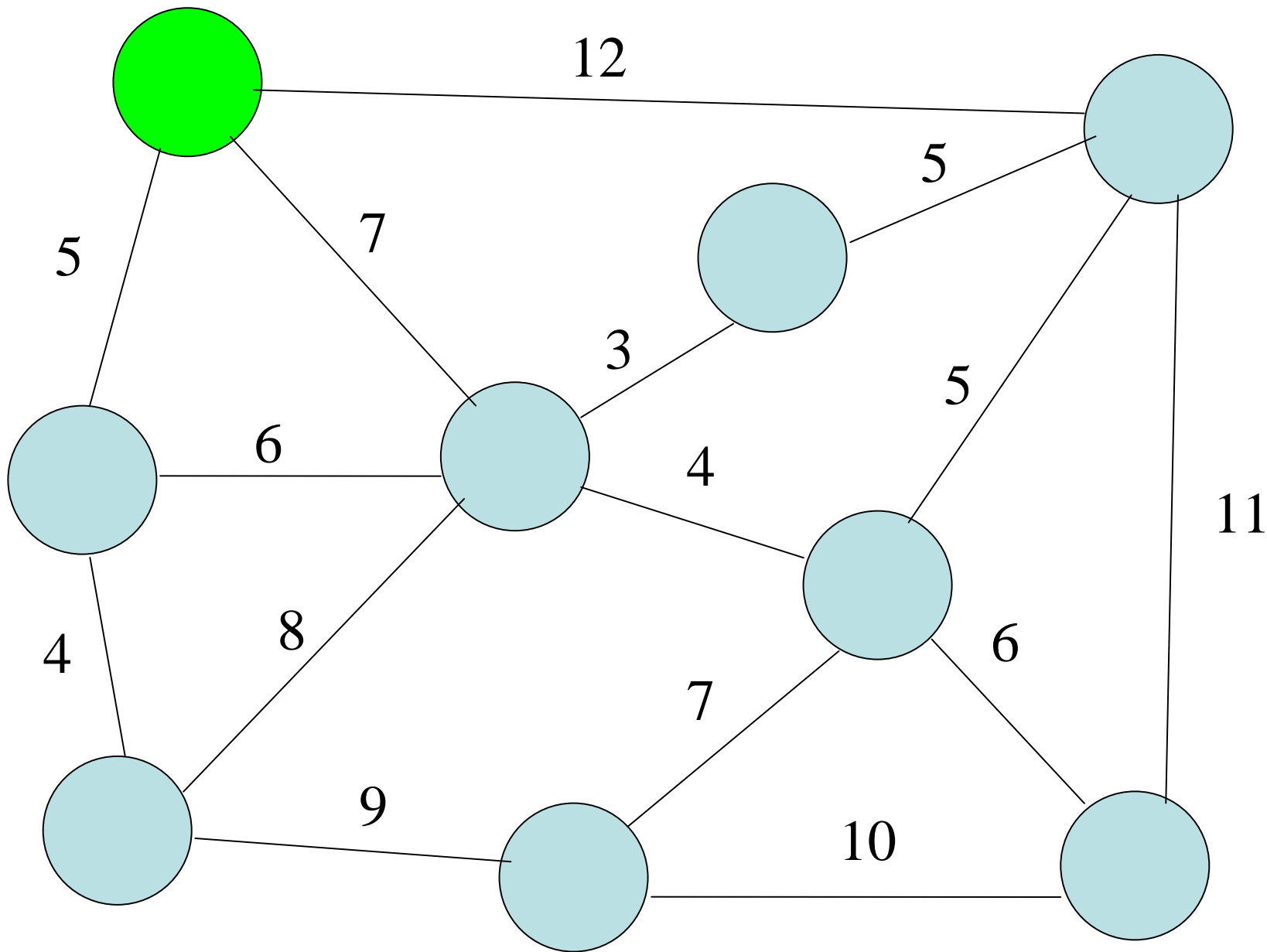
Candidatos: Vértices

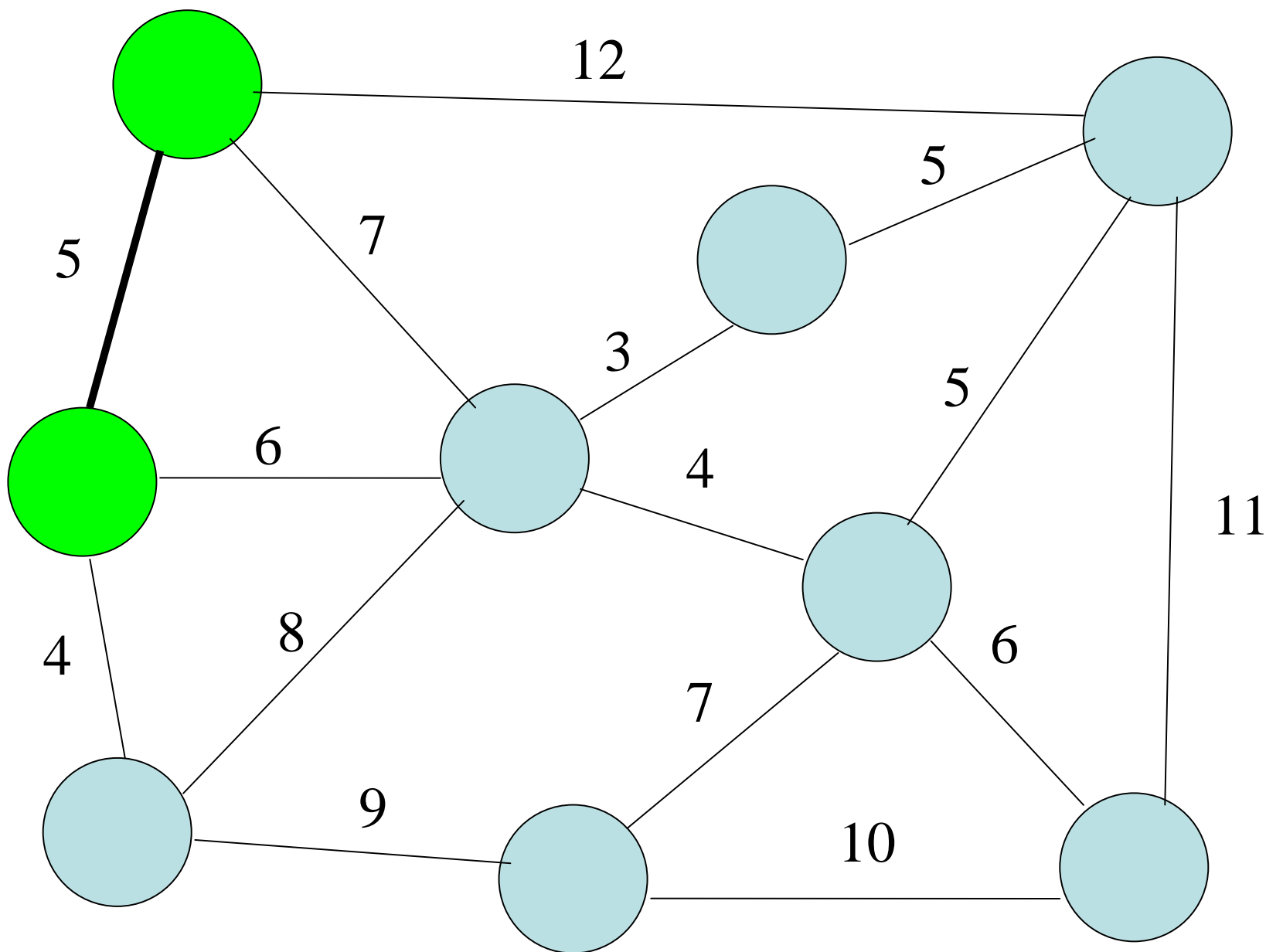
Función Solución: Se ha construido un árbol de recubrimiento (n vértices seleccionadas).

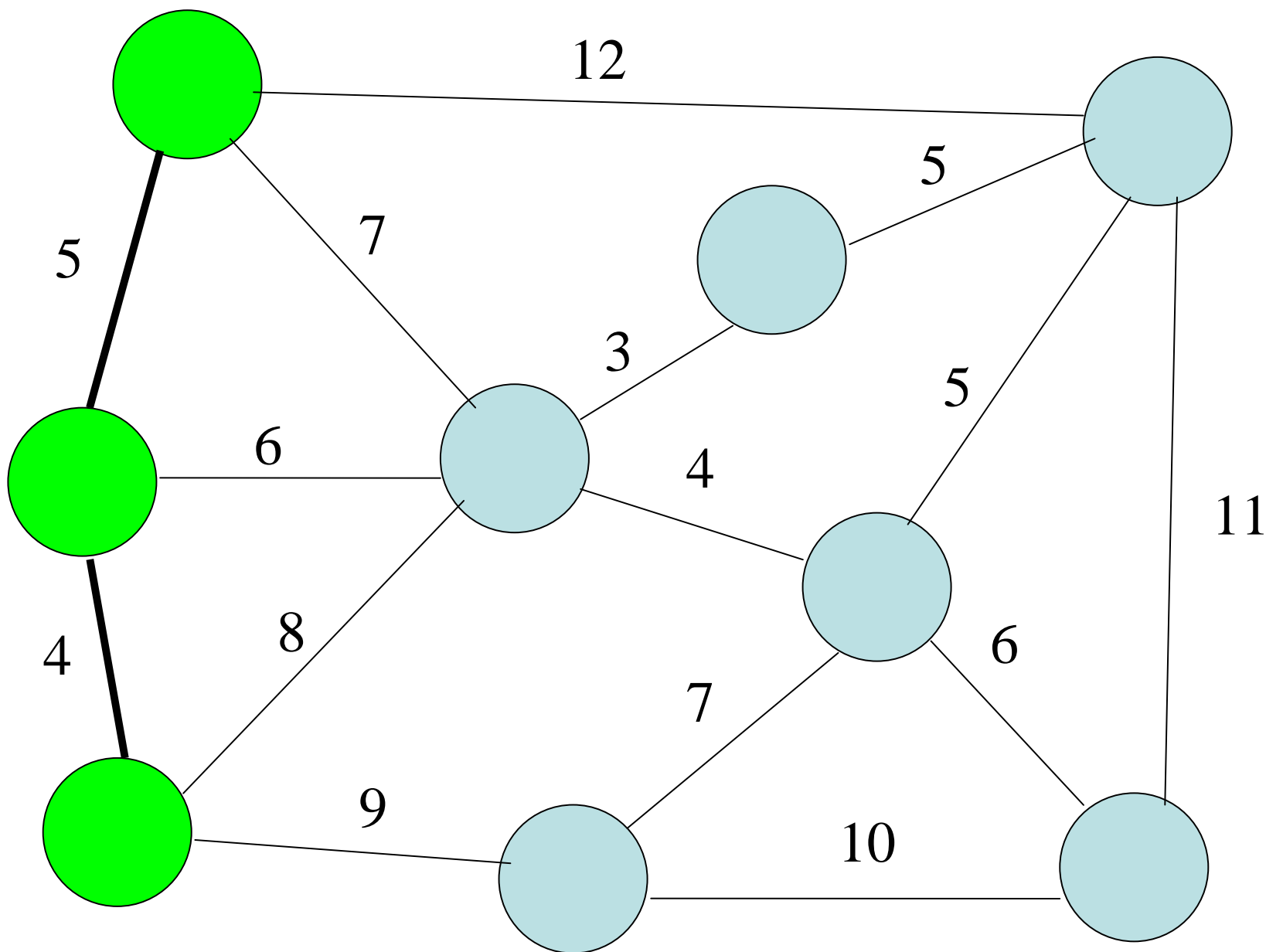
Función Selección: Seleccionar el vertice u del conjunto de no seleccionados que se conecte mediante la arista de menor peso a un vértice v del conjunto de vértices seleccionados. La arista (u,v) está en T .

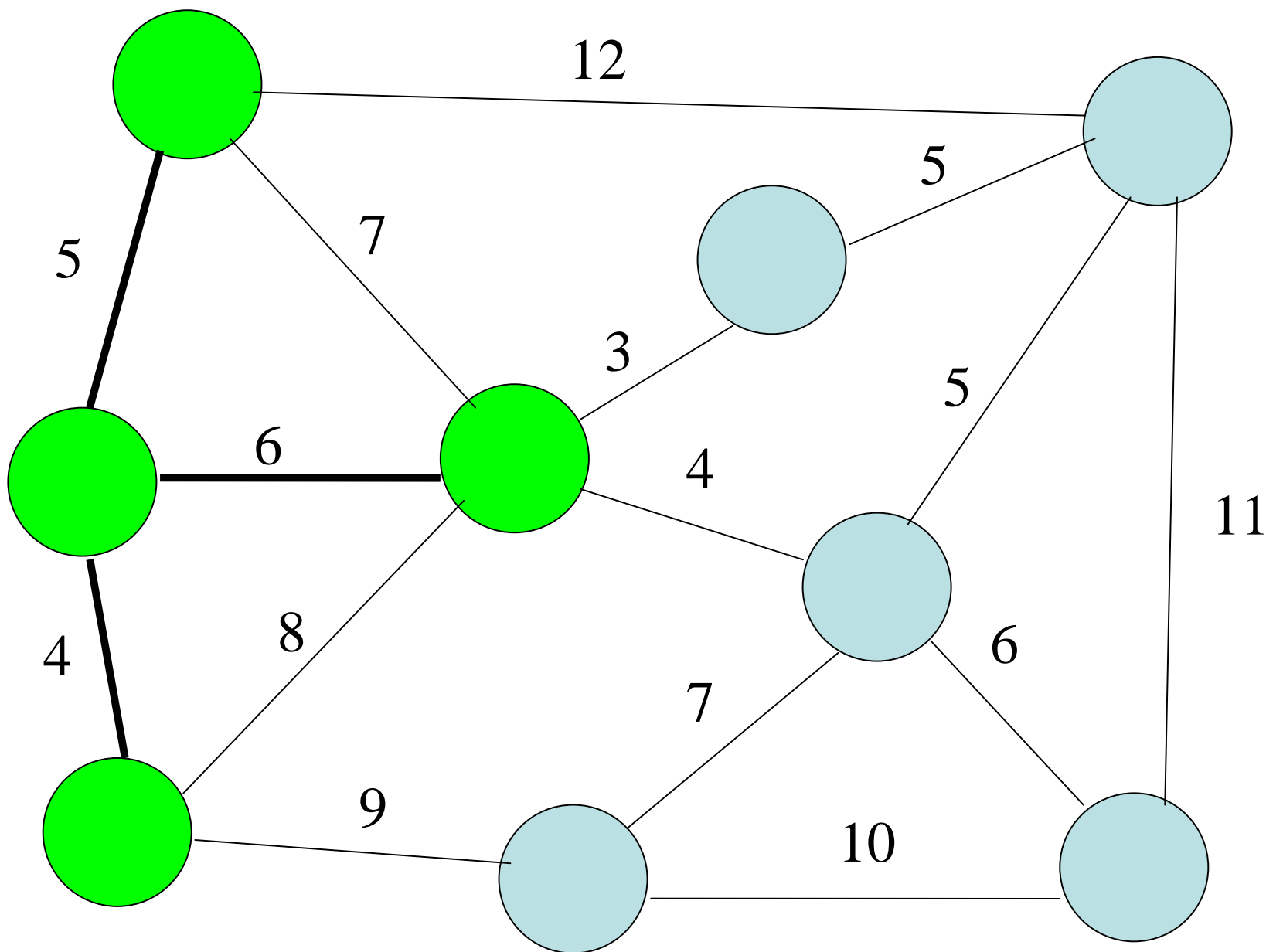
Función de Factibilidad: El conjunto de aristas no contiene ningún ciclo. Está implícita en el proceso

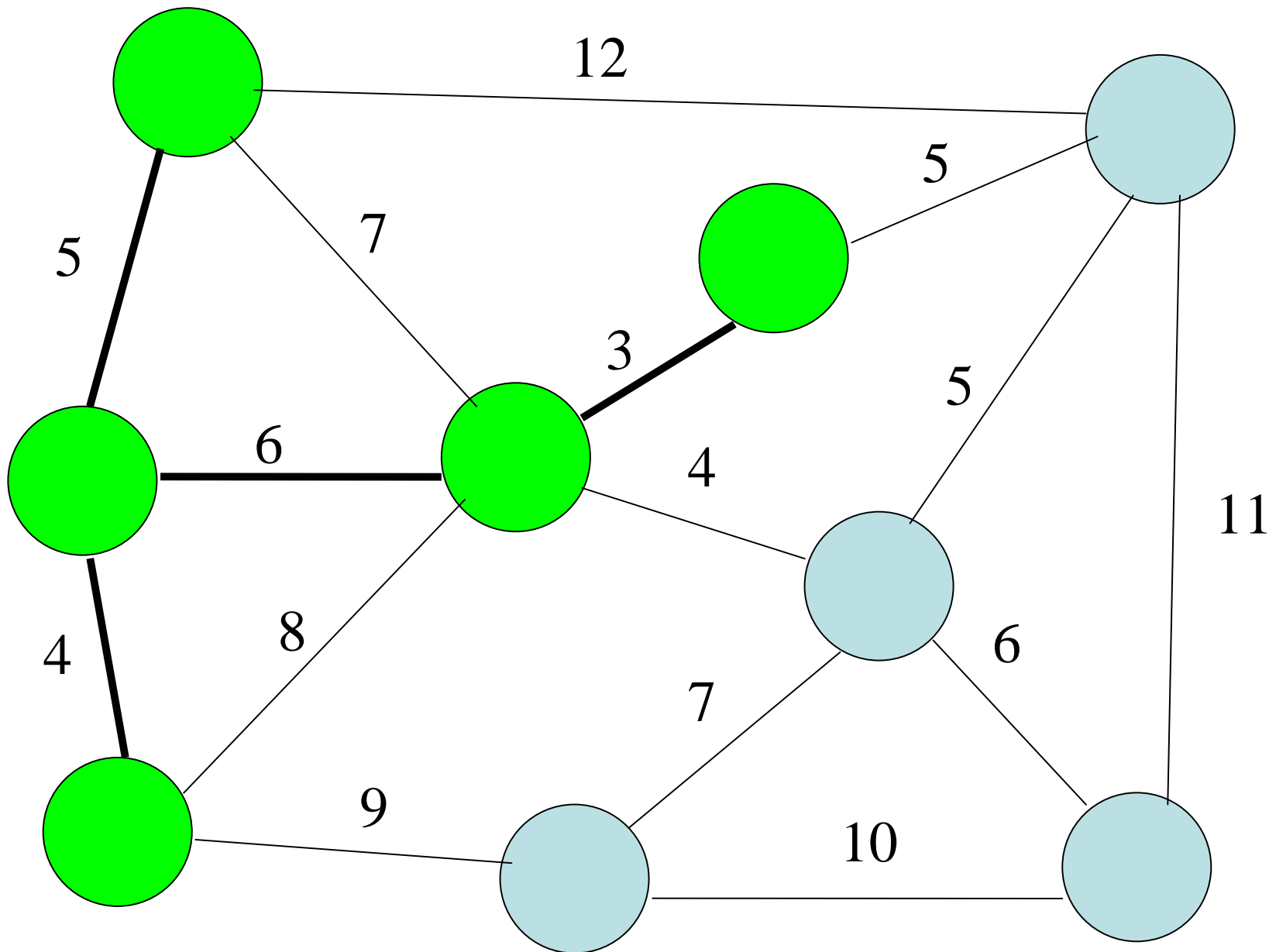
Función Objetivo: determina la longitud total de las aristas seleccionadas.

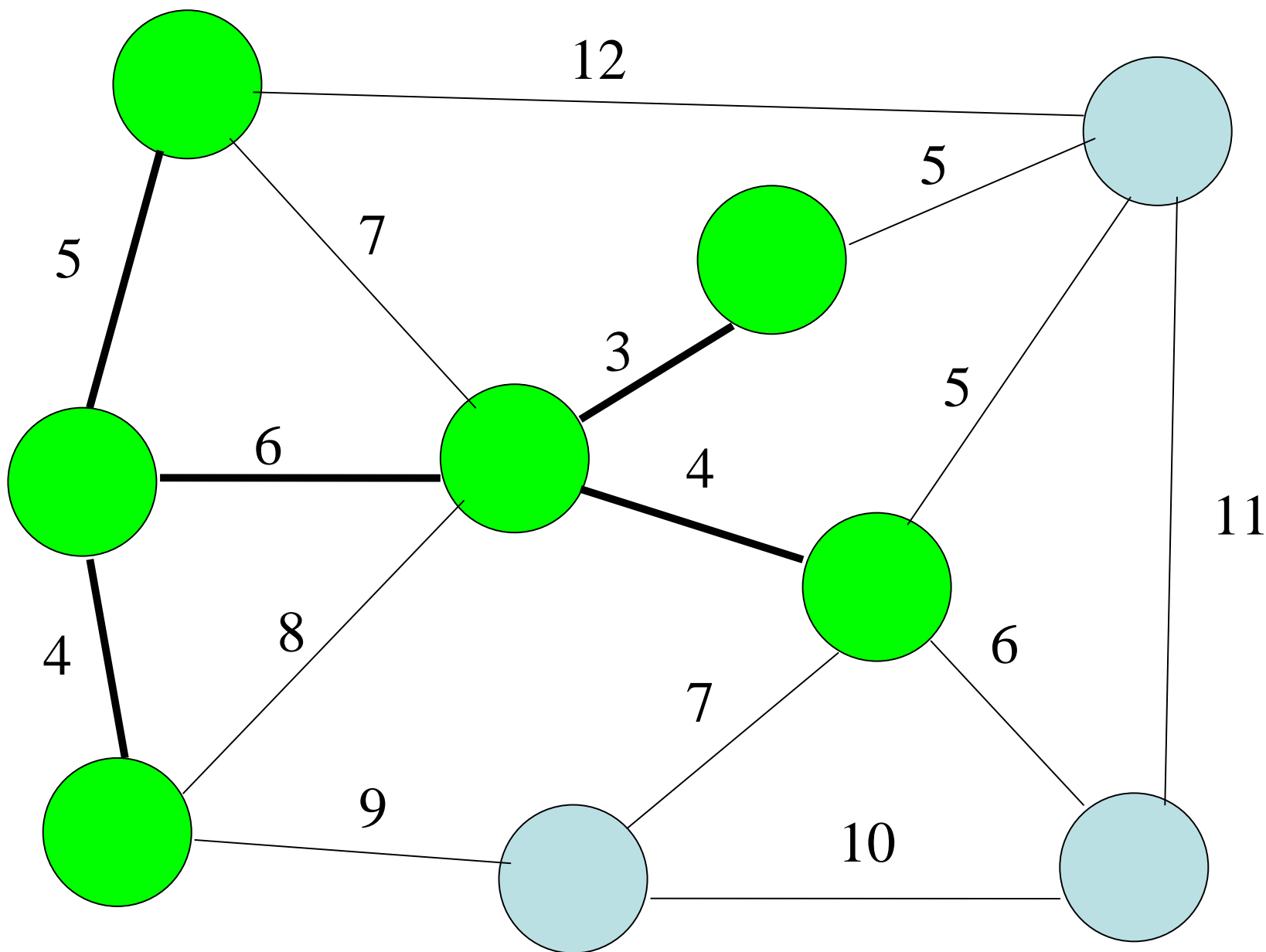


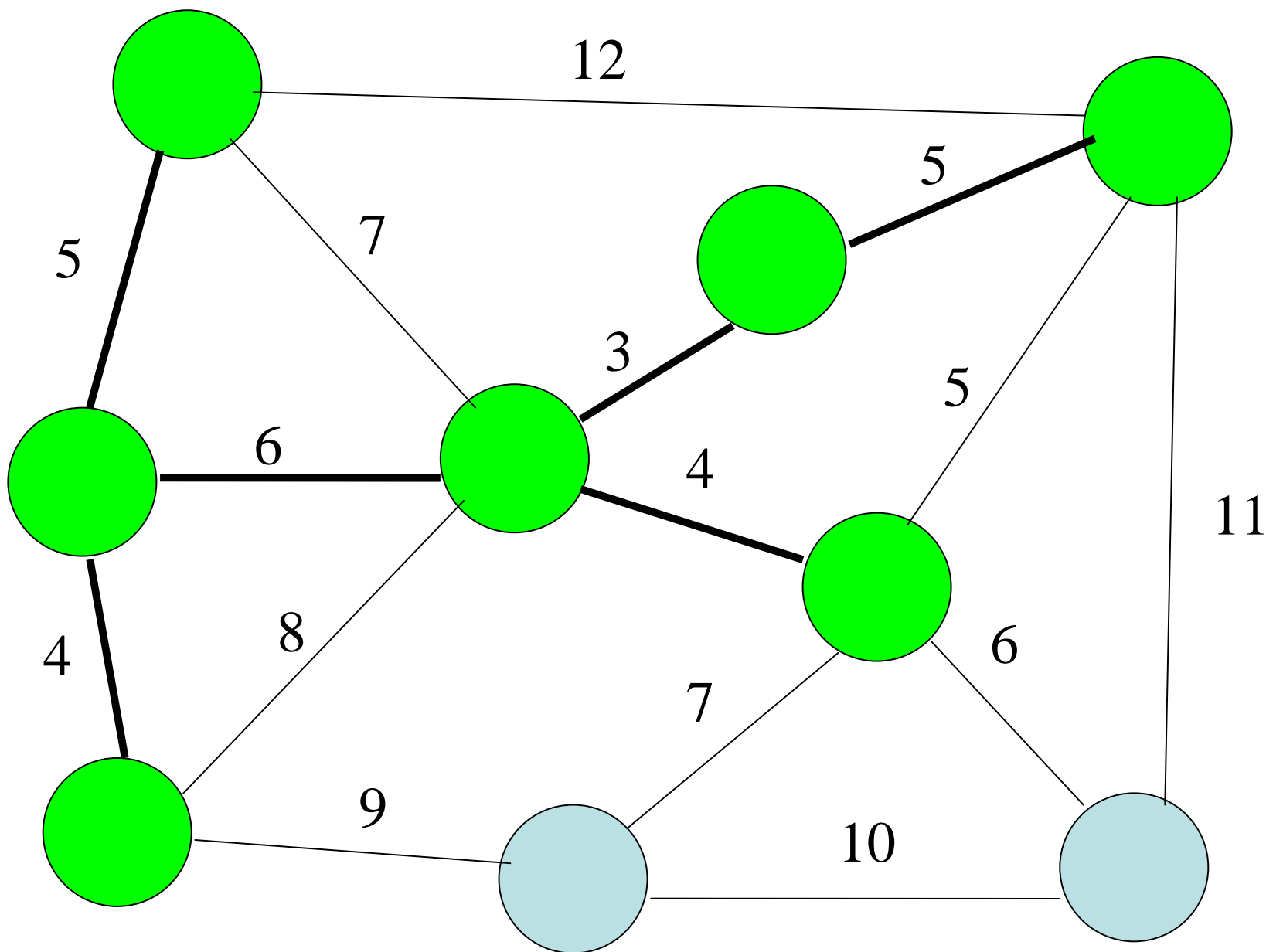


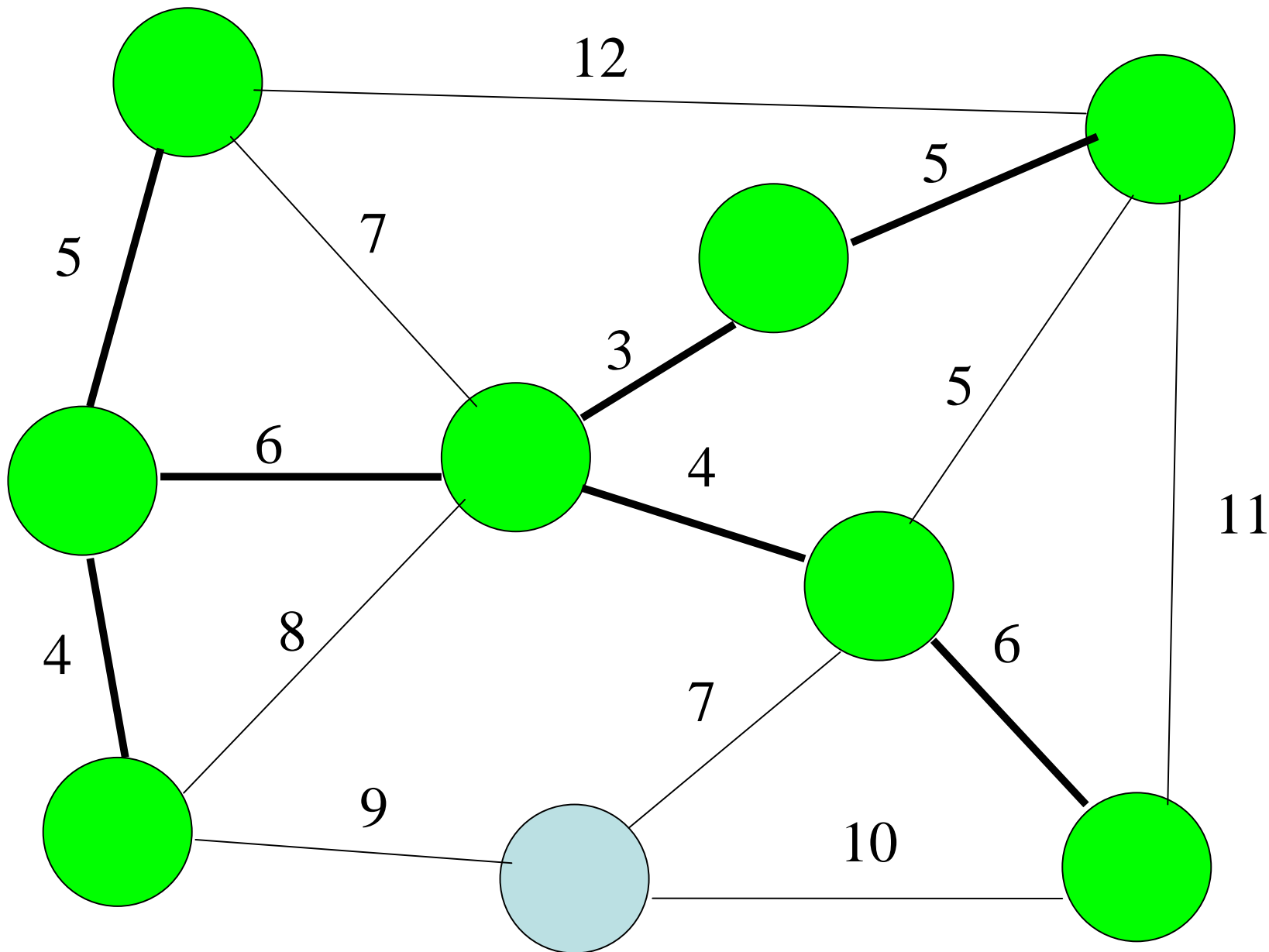


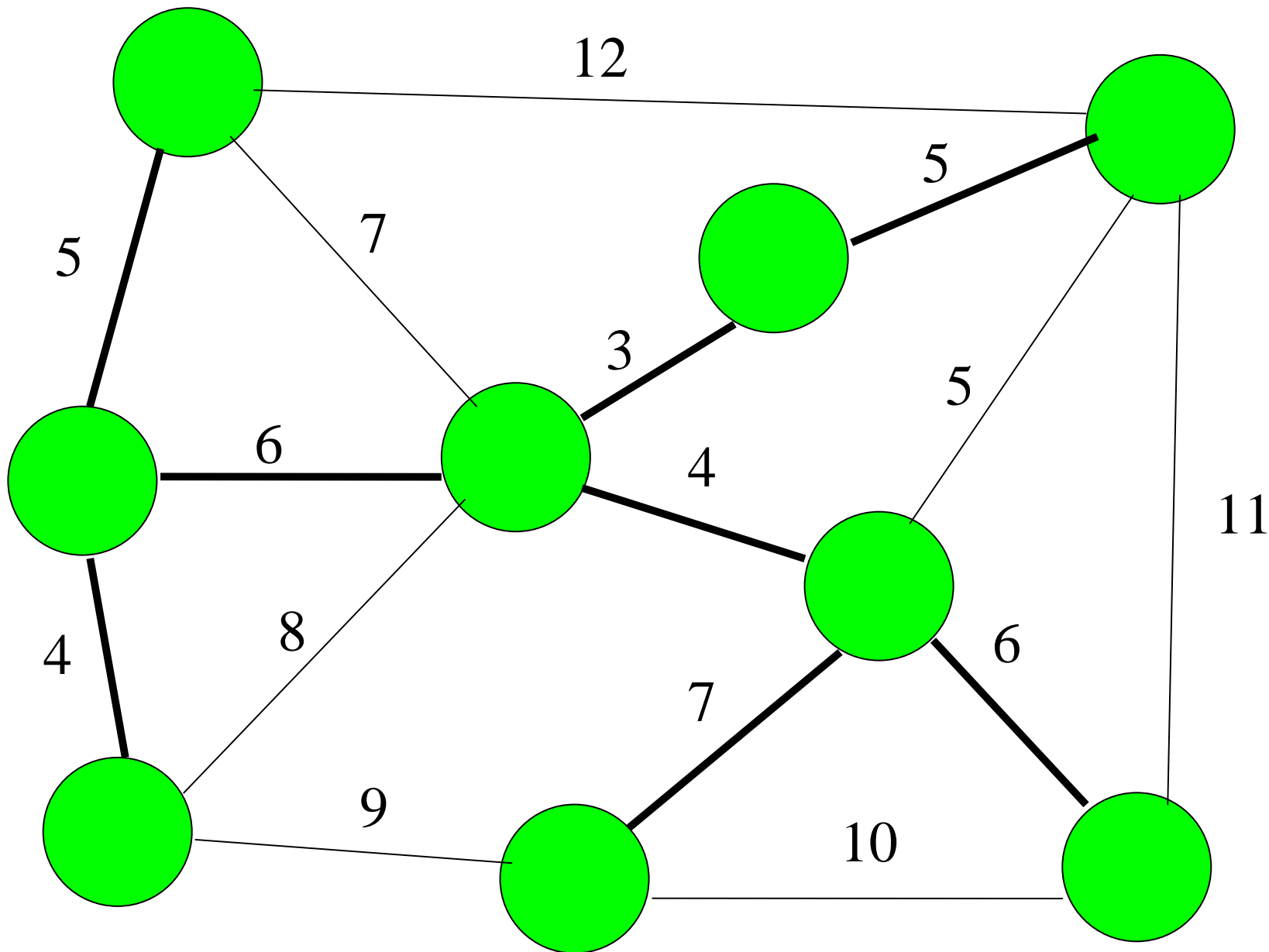












Implementación

- Clave: Seleccionar eficientemente el nuevo arco para añadir al ARM, T.
- Utilizaremos una cola con prioridad Q de vértices con dos campos:
 - $\text{key}[v]$ = 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 árbol.

Implementación

Función Prim($G(V,A)$, r) // r es el `vertice_inicio`.

for each u en V

$key[u] = \text{infinito}$;

$p[u] = \text{NULL}$

$key[r] = 0$;

PriorityQueue $Q(V)$ // con todos los vértices de V ,

while ($!Q.empty()$)

$u = Q.ExtractMin()$;

 for each v en $Adj[u]$

 if (v en Q and $\text{peso}(u,v) < key[v]$)

$p[v] = u$;

$key[v] = \text{peso}(u,v)$;

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 $Q(V)$ // $O(V \log V)$, 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] = peso(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 $Q(V)$ // $O(V \log V)$, 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] = peso(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 Minimios

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

Problema de Caminos Minimios

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.



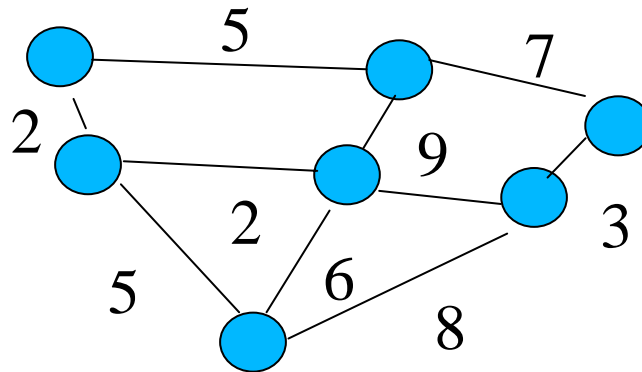
Problema de Caminos Minimios

Modelamos el mapa de carreteras como un grafo: los vértices representan las intersecciones y los arcos representan las carreteras. El peso de un arco \leq distancia



Problema de Caminos Minimios

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



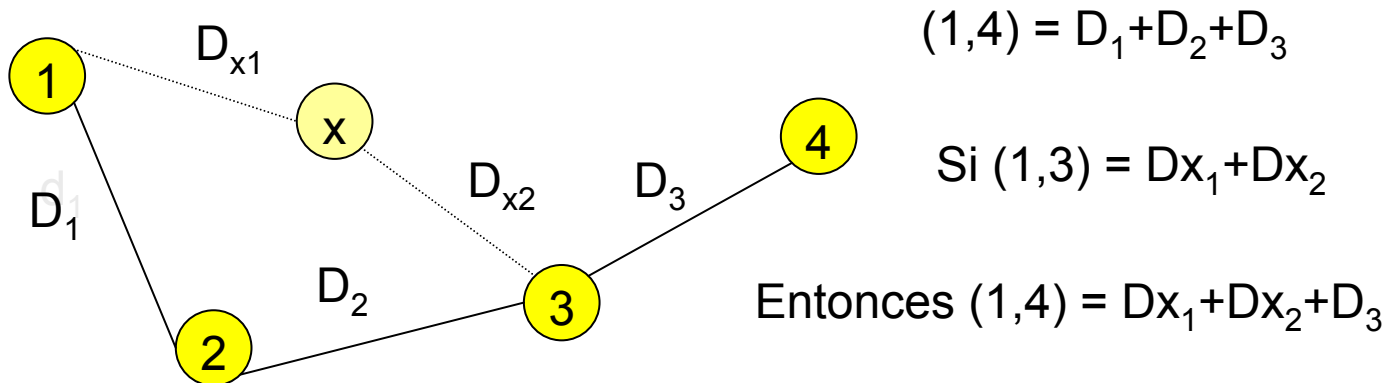
Encontrar el
Camino de costo mínimo para llegar
desde s al resto de los vértices en el grafo

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?)

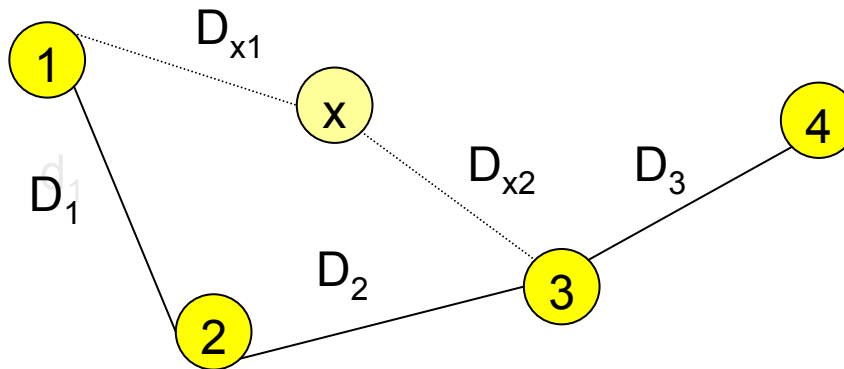


Propiedades de Caminos Mínimos

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

$$M(s,v) \leq M(s,u) + M(u,v)$$

(xq?)



Implementación: Algoritmo Dijkstra

Candidatos: Vértices

Función Selección: Seleccionar el vertice u del conjunto de no seleccionados ($V \setminus S$) 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 valor infinito si no existe dicho camino
- $p[v]$ = padre del v en el camino. Toma Null si no existe dicho padre.

Implementación: Alg. Dijkstra.

- Al incluirse un vértice v en S puede ocurrir que sea necesario actualizar $d[x]$.
 - $Xq?$
 - Qué vértices son susceptibles de sufrir dicha actualización?
 - Como se ve afectado $d[x]$ y $p[x]$?

Implementación: Alg. Dijkstra.

ALGORITMO DE DIJKSTRA

Para cada v en V

$d[v] = \text{infinito}$

$\text{pred}(v) = \text{null}$

$d[s] = 0$

Priorityqueue_ $Q(V)$; // se crea la cola respecto a $d[x]$

while ($!Q.\text{empty}()$)

$v = Q.\text{ExtractMin}()$

Para cada w en $\text{Adj}[v]$

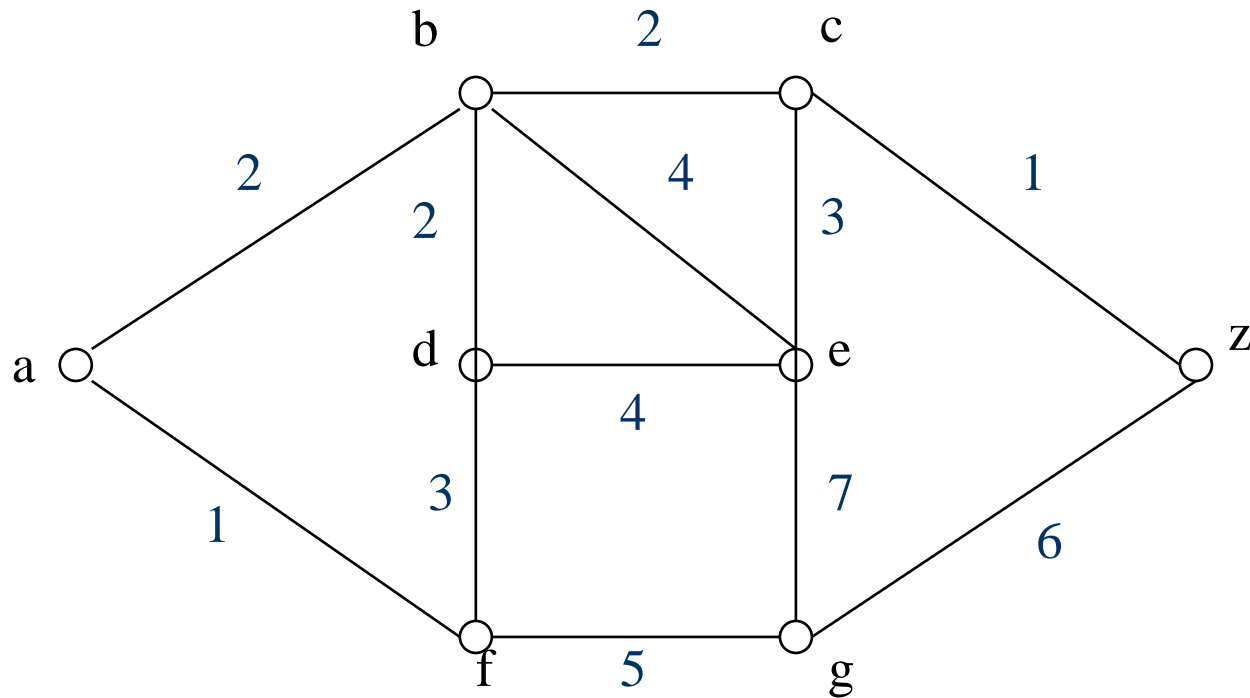
if $d[w] > d[v] + c(v, w)$

$d[w] = d[v] + c(v, w)$ // se reubica en la cola

$\text{pred}(w) = v$ // se van almacenando los caminos

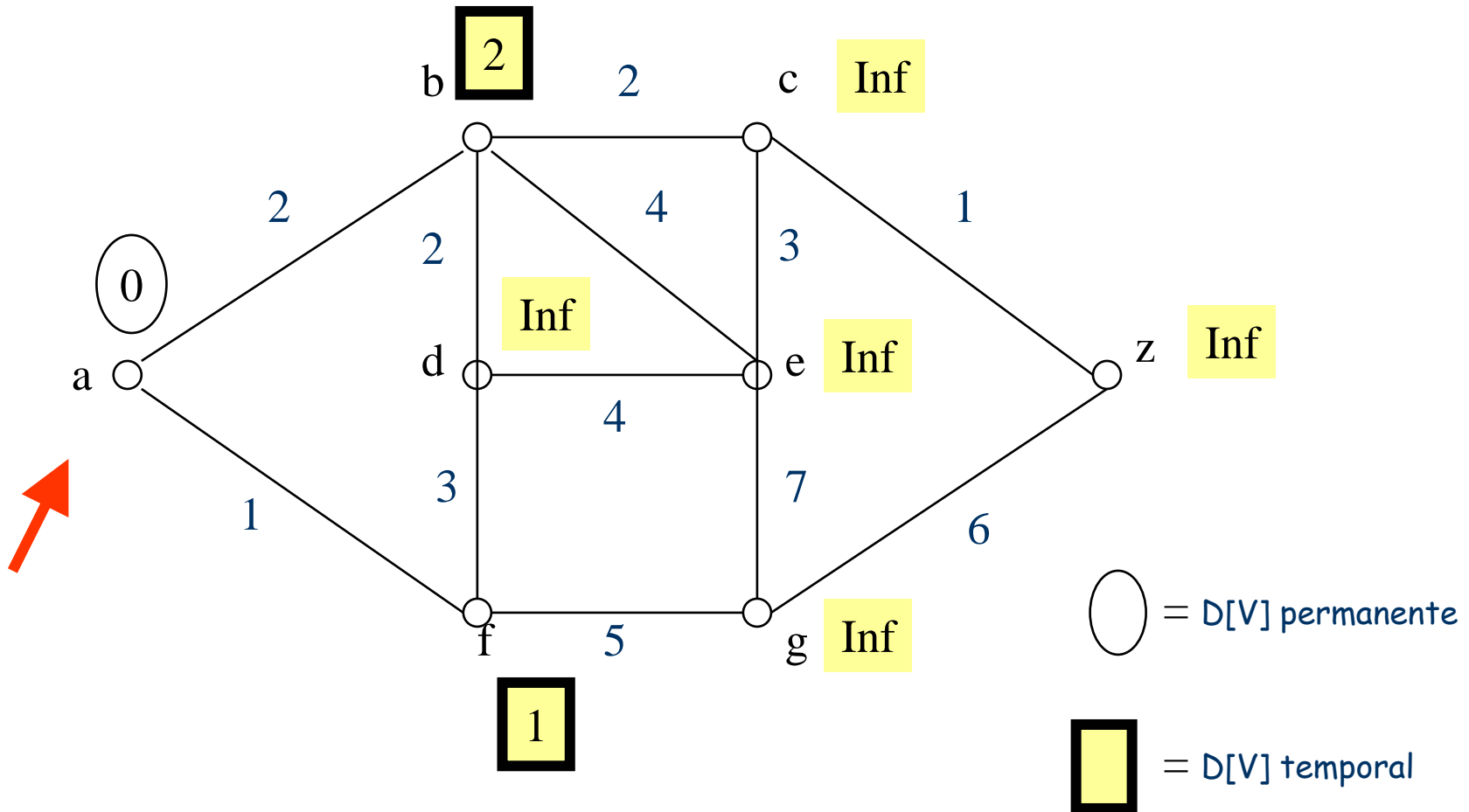
(Sol.: Se recorren los padres hacia atrás desde el vértice destino hasta el origen)

Ejemplo: Algoritmo Dijkstra



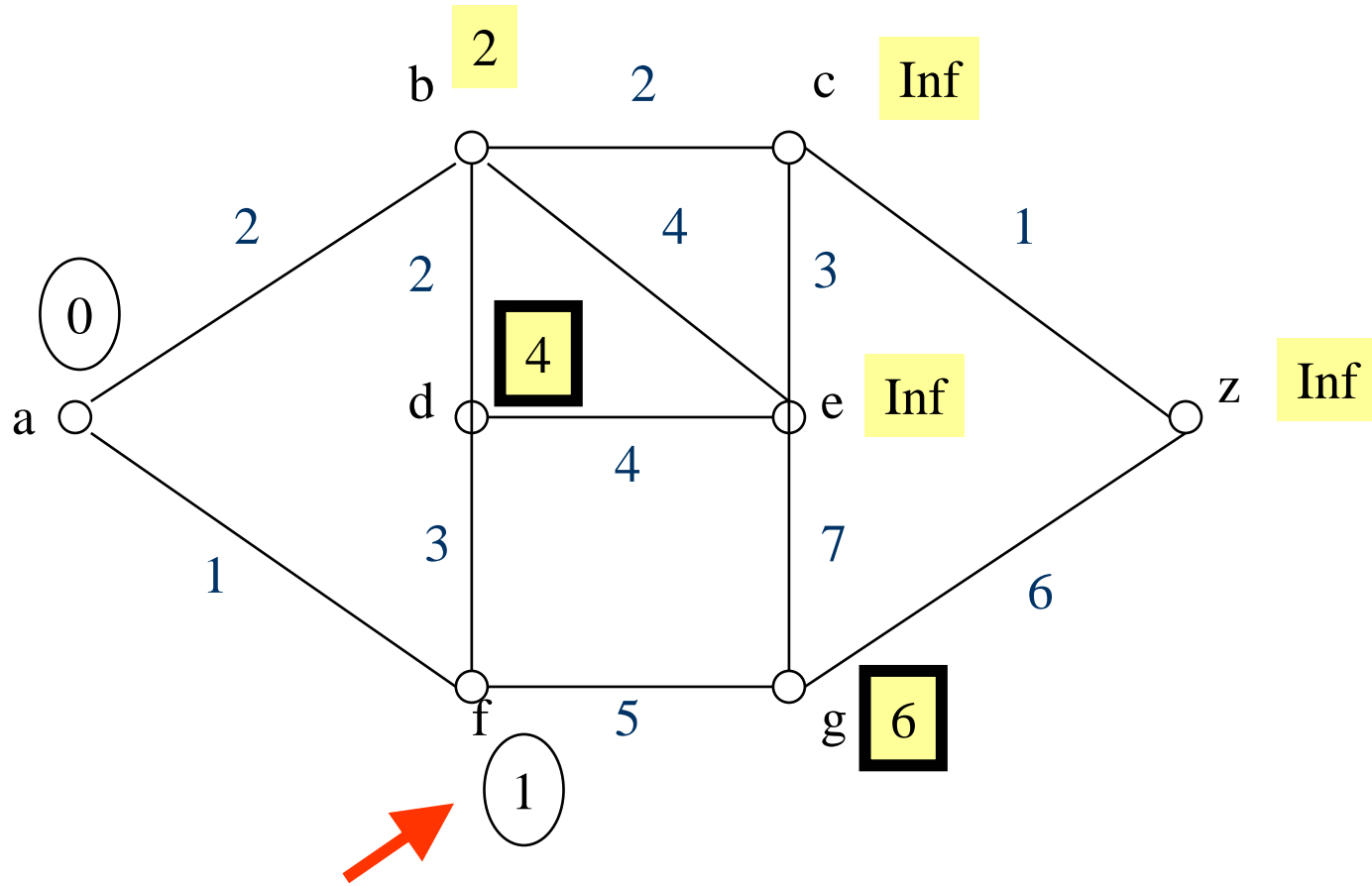
Ejemplo (Solo entre a y z)

Ejemplo: Algoritmo Dijkstra



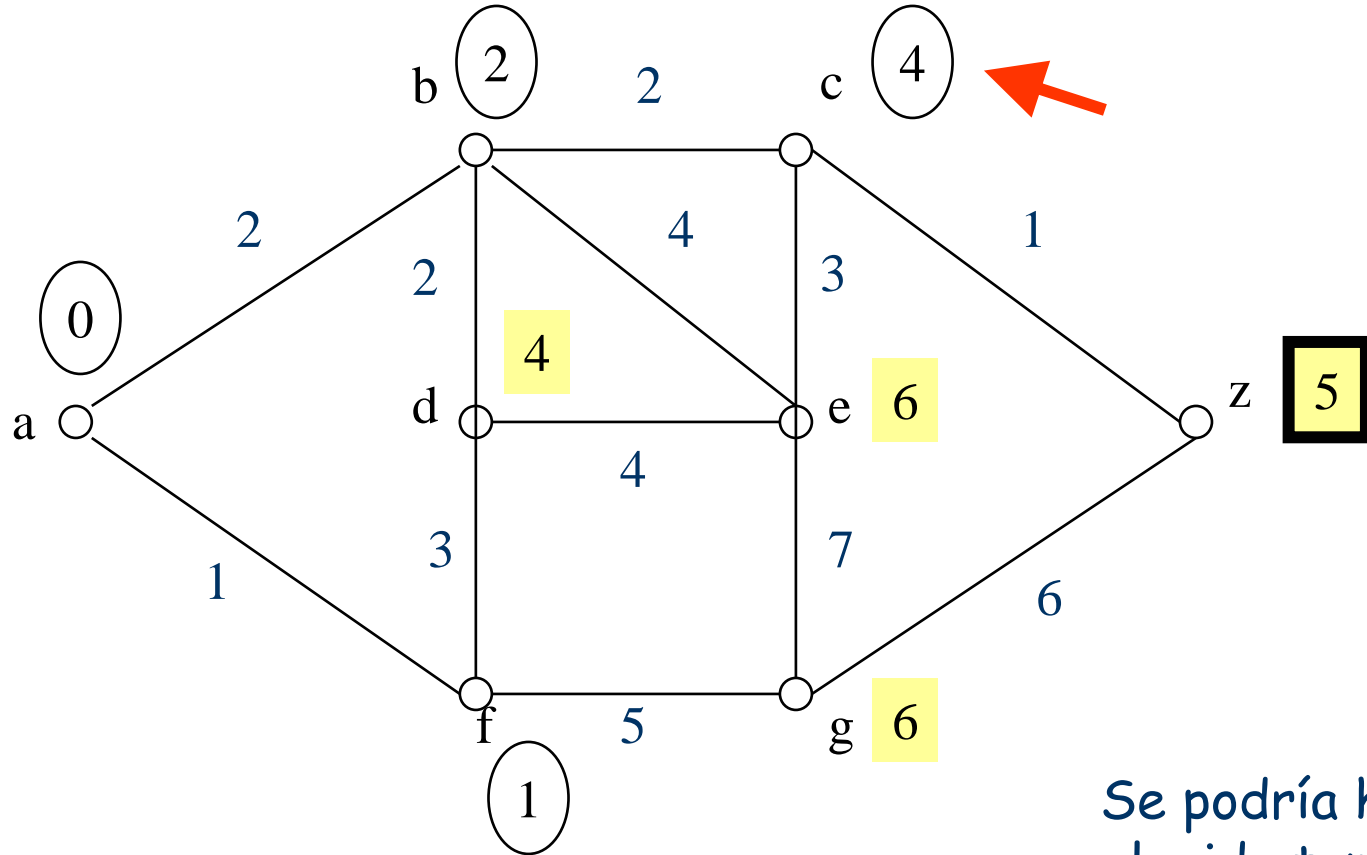
Primera iteración

Ejemplo: Algoritmo Dijkstra



Segunda Iteración

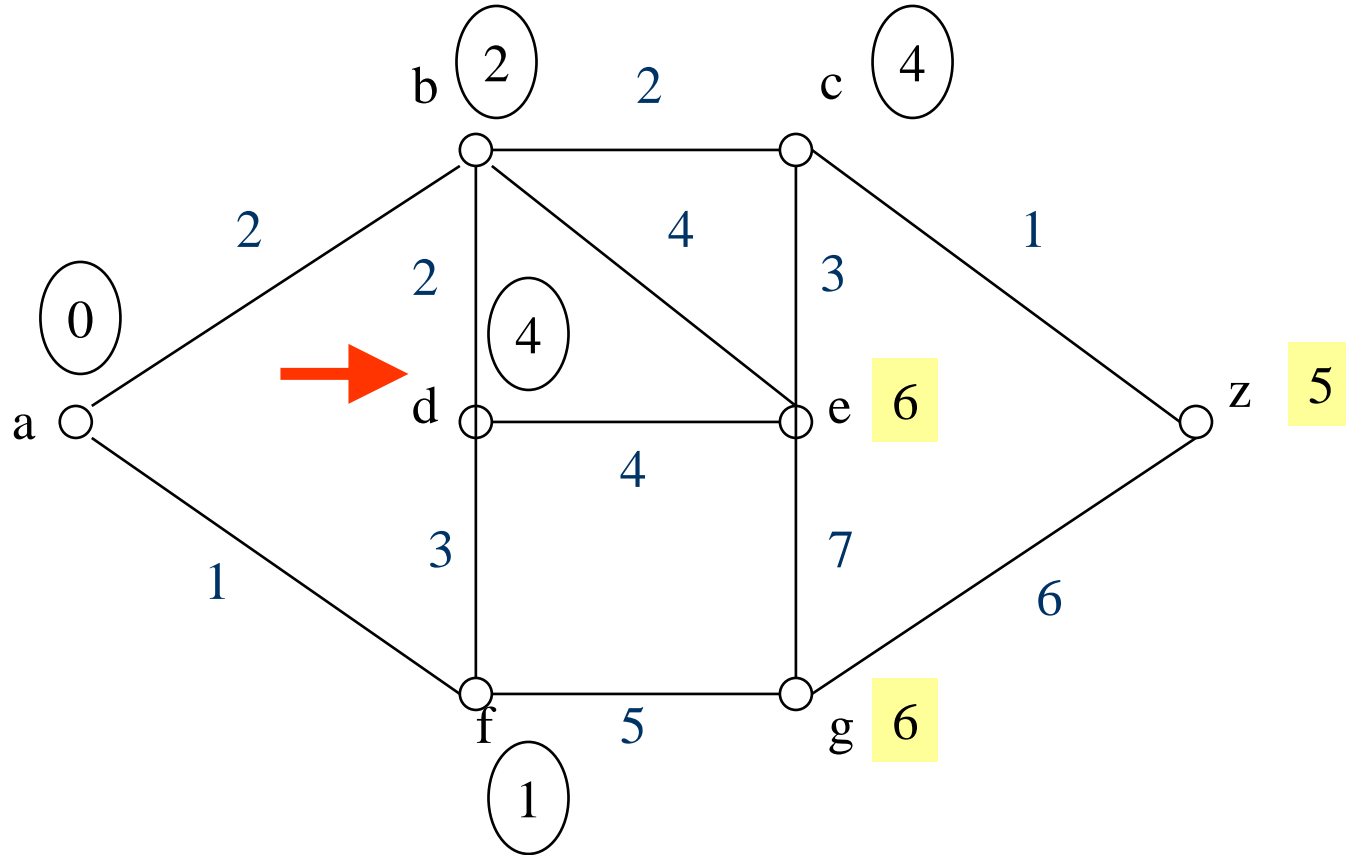
Ejemplo: Algoritmo Dijkstra



Se podría haber
elegido también 'd'

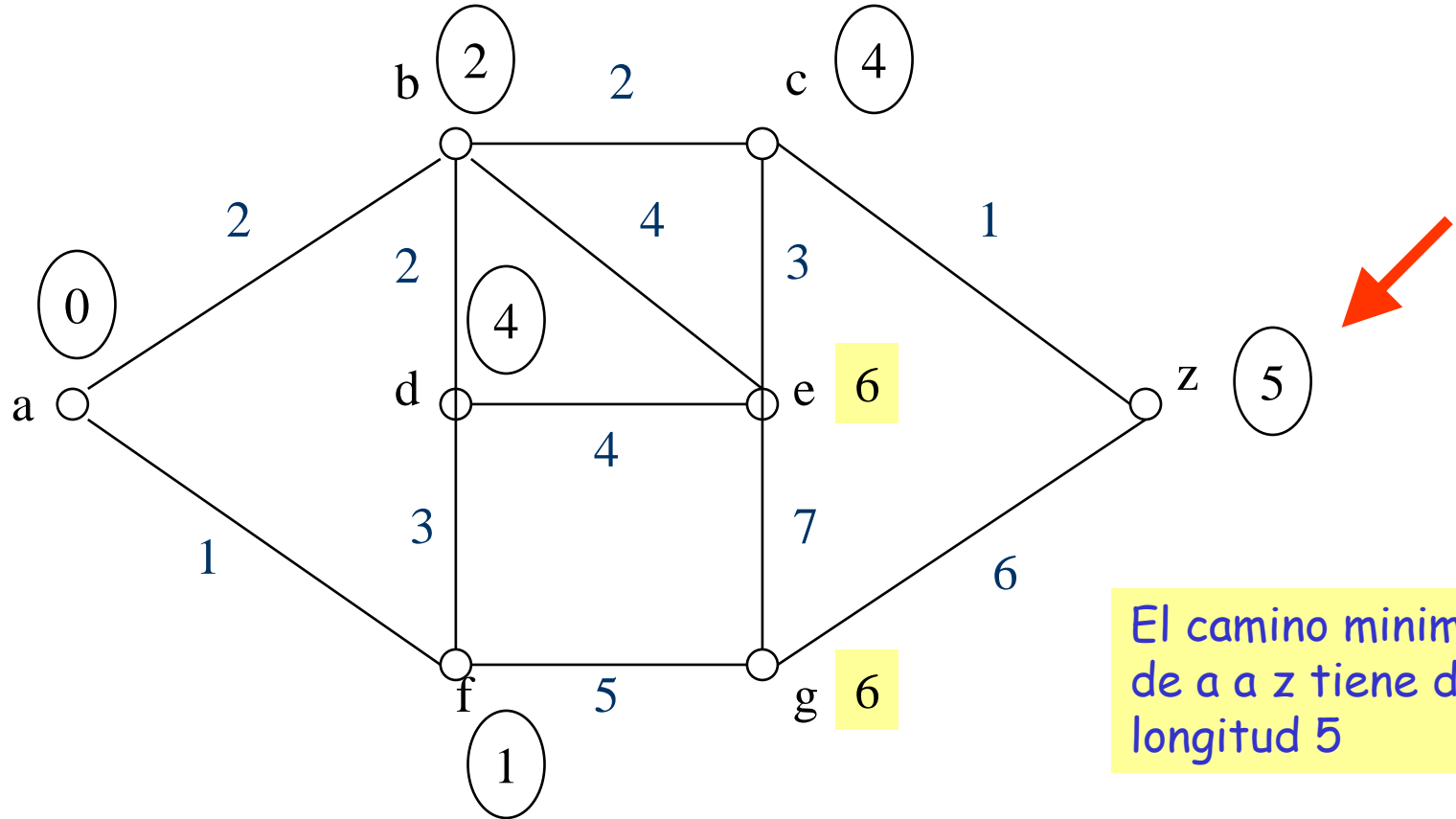
Cuarta Iteración

Ejemplo: Algoritmo Dijkstra



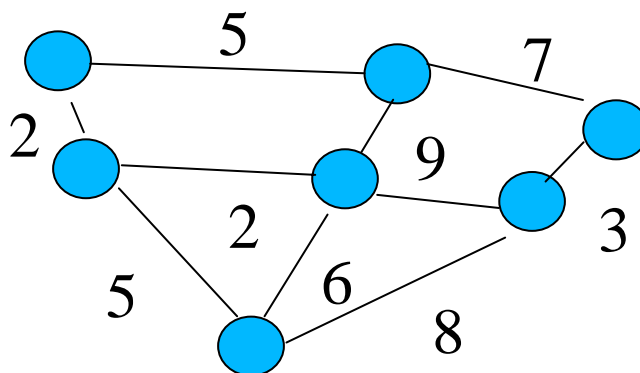
Quinta Iteración

Ejemplo: Algoritmo Dijkstra



Sexta (y ultima) Iteración

Ejemplo: Algoritmo Dijkstra

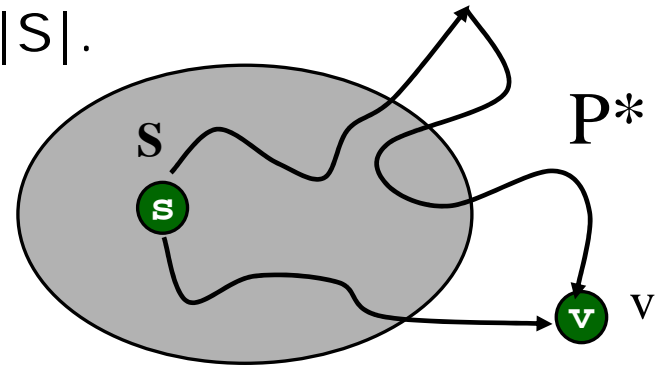


Queda como ejercicio.

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 . $d(v)$ 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 - v
 - Sea x el último vértice en S en dicho camino

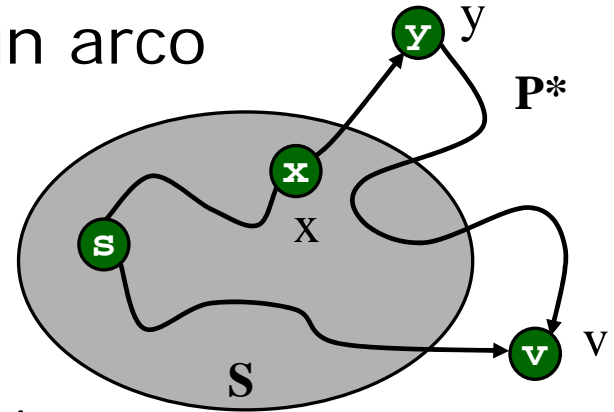
Algoritmo Dijkstra: Demostración

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

■ Entonces tenemos que

$$\begin{aligned} d(v) &> M(s, v) \\ &= M(s, x) + c(x, y) + M(y, v) \\ &\geq M(s, x) + c(x, y) \\ &= d[x] + c(x, y) \\ &\geq d(y) \end{aligned}$$

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



asumimos

subestr. optimal

arcos positivos

hipótesis inducción

Análisis de Eficiencia

- EL análisis del algoritmo es parecido al realizado para el algoritmo de Prim, deduciendo que el algoritmo es del orden $O(A \log V)$.
- Realmente es $O((A+V)\log V)$, pero se supone que A es bastante mayor que V . Además, si el grafo es muy denso $A \approx V^2$.
 - Demostrarlo queda como ejercicio.

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 un algoritmo voraz para el que se pueda demostrar que encuentra la solución óptima
- Sin embargo, en muchos casos se pueden llegar plantear 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 (Recocido) Simulado, Búsqueda Tabu,
 - GRASP (Greedy Randomized Adaptive Search Procedures), Búsqueda Dispersa, Búsqueda por Entornos Variables, Búsqueda Local Guiada, Búsqueda Local Iterativa
 - Computación Evolutiva (Algoritmos Genéticos, ...), Algoritmos Meméticos, Colonias de Hormigas, Redes de Neuronas, Algoritmos Basados en Sistemas Inmunológicos, ...

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)

Heurísticas greedy: Resumen

- **Problemas NP-completos:** la solución exacta puede requerir órdenes factoriales o exponenciales (el problema de la *explosión combinatoria*).
- **Objetivo:** obtener “buenas” soluciones en un tiempo de ejecución corto (*razonable*).
- **Algoritmos de aproximación:** garantizan una solución más o menos buena (o una cierta aproximación respecto al óptimo).
- Un tipo son los **algoritmos heurísticos**¹: algoritmo basado en el conocimiento “intuitivo” o “experto” del programador sobre determinado problema.

¹DRAE. *Heurística*: Arte de inventar.

Heurísticas greedy: Resumen

PROBLEMA COMPLEJO



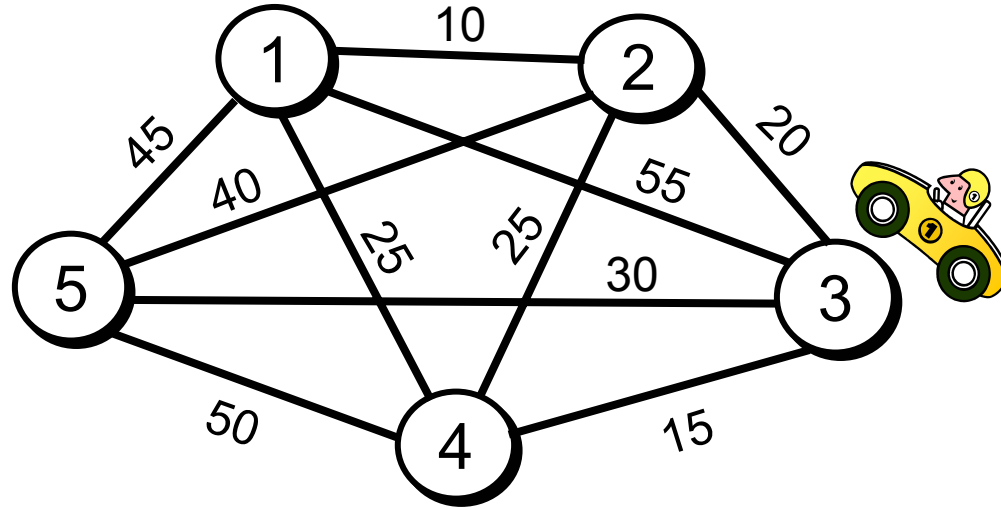
- La estructura de algoritmos voraces se puede utilizar para construir procedimientos heurísticos: hablamos de **heurísticas voraces**.
- **La clave:** diseñar buenas funciones de selección.

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.

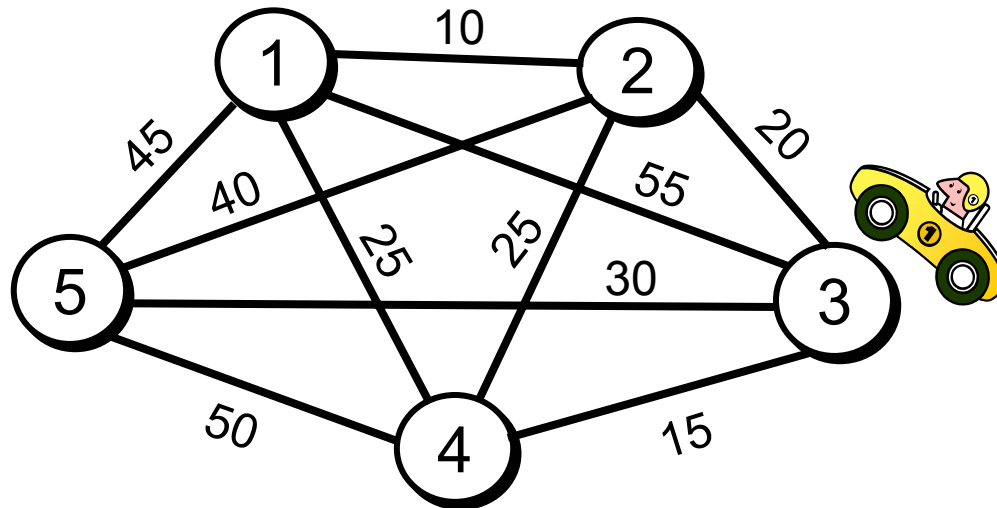
- **Problema:** Dado un grafo no dirigido, completo y ponderado $G = (V, A)$, encontrar un ciclo de coste mínimo que pase por todos los nodos.



- Es un problema NP-completo, pero necesitamos una solución eficiente.
- Problema de optimización, la solución está formada por un conjunto de elementos en cierto orden: podemos aplicar el esquema voraz.

El problema del viajante.

- Primera cuestión: ¿Cuáles son los candidatos?
- **Dos posibilidades:**
 - 1) Los nodos son los candidatos. Empezar en un nodo cualquiera. En cada paso moverse al nodo no visitado más próximo al último nodo seleccionado.
 - 2) Las aristas son los candidatos. Hacer igual que en el algoritmo de Kruskal, pero garantizando que se forme un ciclo.

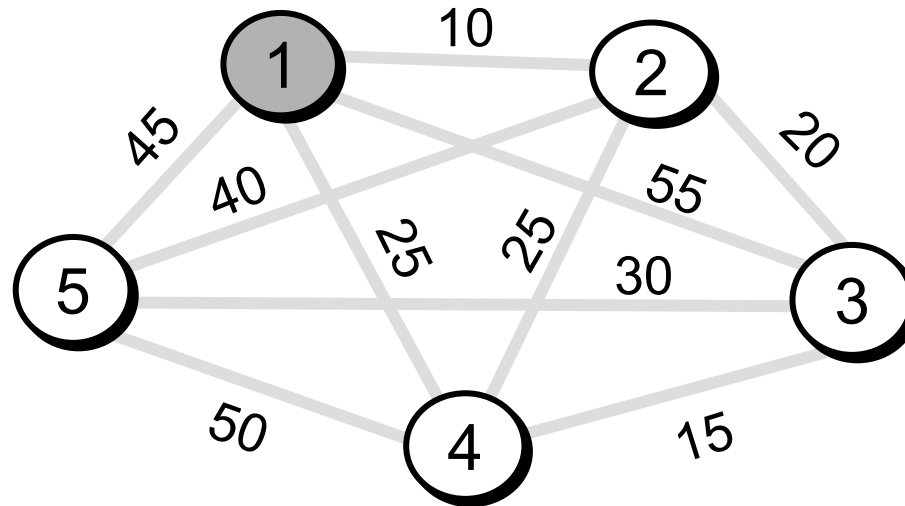


El problema del viajante.

- **Heurística voraz 1) Candidatos = V**
 - Una solución será un cierto orden en el conjunto de nodos.
 - **Representación de la solución:** $s = (c_1, c_2, \dots, c_n)$, donde c_i es el nodo visitado en el lugar i -ésimo.
 - **Inicialización:** empezar en un nodo cualquiera.
 - Función de **selección:** de los nodos candidatos seleccionar el más próximo al último (o al primero) de la secuencia actual (c_1, c_2, \dots, c_a) .
 - Acabamos cuando tengamos n nodos.

El problema del viajante.

- **Ejemplo 1.** Empezando en el nodo 1.

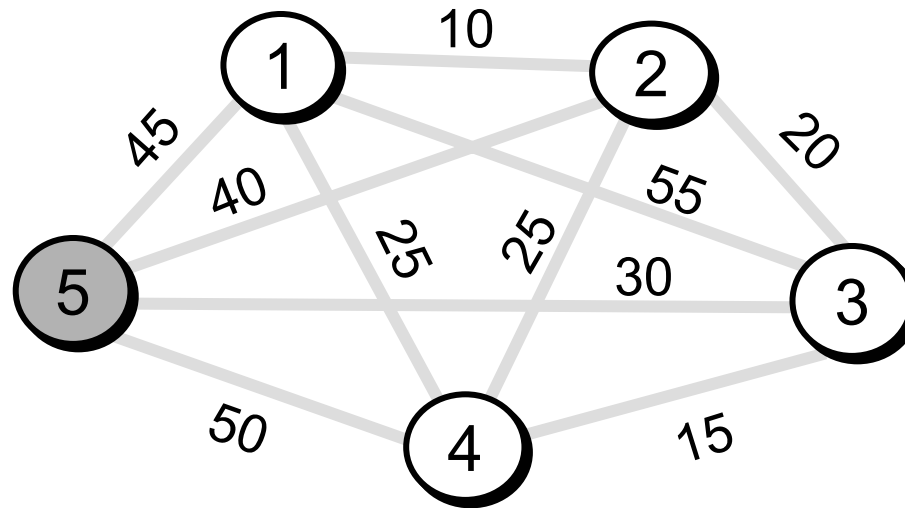


Solución: (5, 1, 2, 3, 4)

Coste total: $45+10+20+15+50=140$

El problema del viajante.

- **Ejemplo 2.** Empezando en el nodo 5.



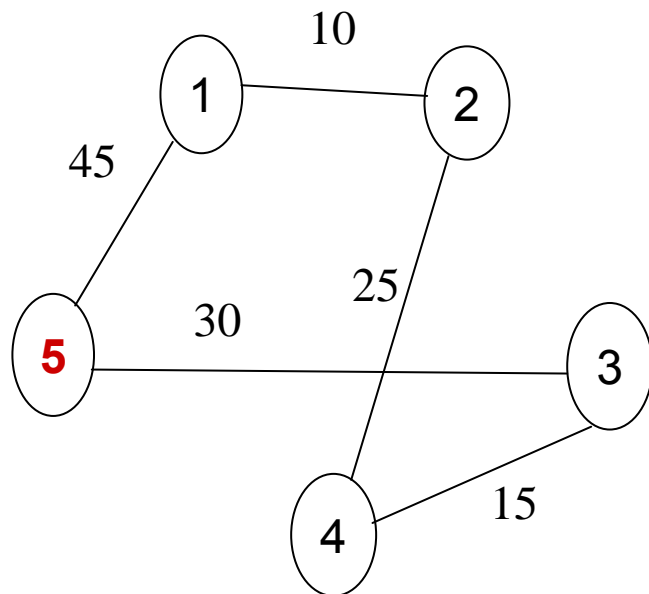
Solución: (5, 3, 4, 2, 1)

Coste total: $30+15+25+10+45 = 125$

- **Conclusión:** el algoritmo no es óptimo.

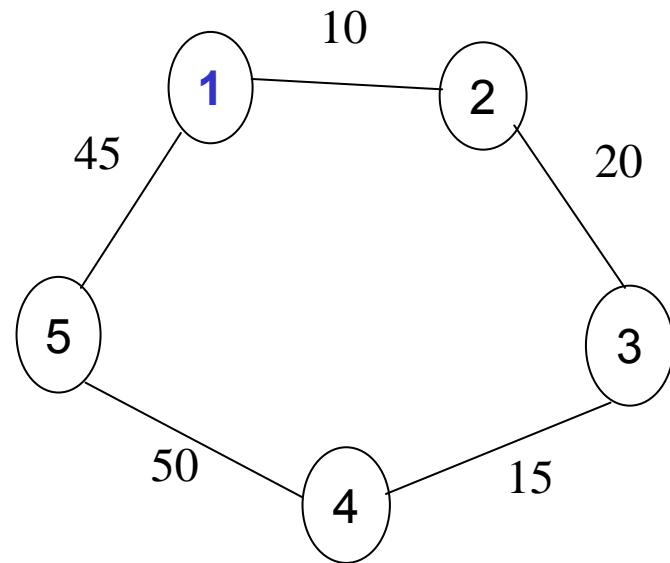
El Problema del Viajante

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



Solución: (5, 3, 4, 2, 1), **125**

Solución empezando en el nodo 1



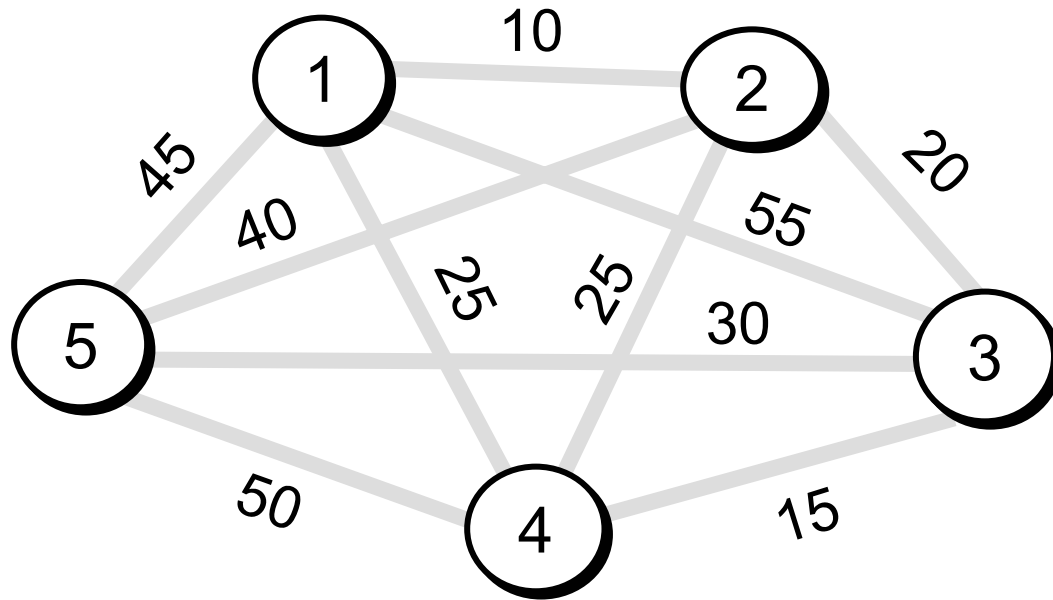
Solución: (5, 1, 2, 3, 4), **140**

El problema del viajante.

- **Heurística voraz 2) Candidatos = A**
 - Una solución será un conjunto de aristas (a_1, a_2, \dots, a_n) que formen un ciclo hamiltoniano, sin importar el orden.
 - **Representación de la solución:** $s = (a_1, a_2, \dots, a_n)$, donde cada a_i es una arista, de la forma $a_i = (v_i, w_i)$.
 - **Inicialización:** empezar con un grafo sin aristas.
 - **Selección:** seleccionar la arista candidata de menor coste.
 - **Factible:** una arista se puede añadir a la solución actual si no se forma un ciclo (excepto para la última arista añadida) y si los nodos unidos no tienen grado mayor que 2.

El problema del viajante.

- Ejemplo 3.

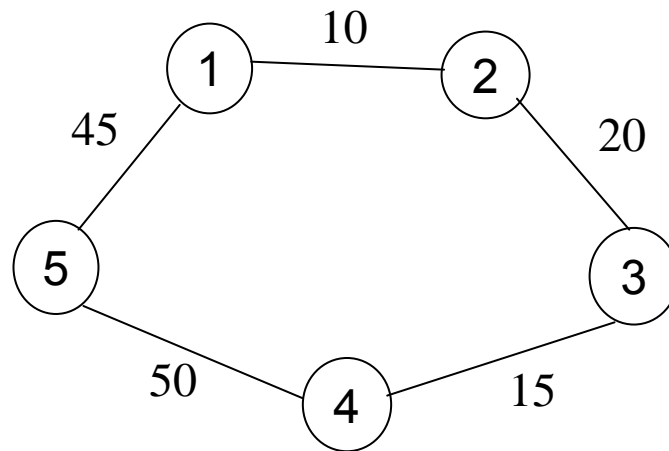


Solución: ((1, 2), (4, 3), (2, 3), (1, 5), (4, 5))

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

El Problema del Viajante

- Solucion con la segunda heurística



- Solución: $((1, 2), (4, 3), (2, 3), (1, 5), (4, 5))$
Coste = $10 + 15 + 20 + 45 + 50 = 140$
- En todos los casos la eficiencia es la del algoritmo de ordenación que se use

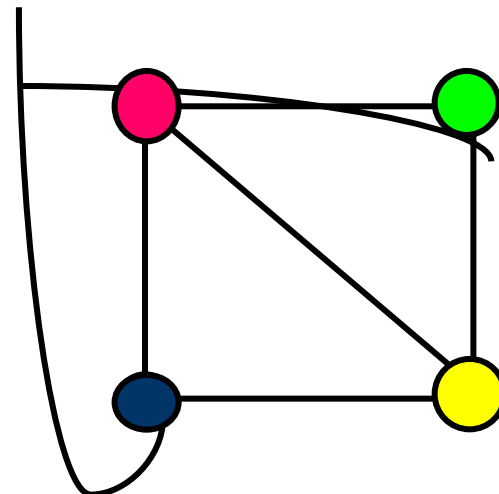
El problema del viajante.

Conclusiones:

- Ninguno de los dos algoritmos garantiza la solución óptima.
- Sin embargo, “normalmente” ambos dan soluciones buenas, próximas a la óptima. ±
- Posibles mejoras:
 - Buscar heurísticas mejores, más complejas.
 - Repetir la heurística 1 con varios orígenes.
 - A partir de la solución del algoritmo intentar hacer **modificaciones locales** para mejorar esa solución.

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 rápido
- El problema reúne 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

El problema del coloreo de un grafo

- Podemos usar una **heurística voraz** para obtener una solución:
 - Inicialmente ningún nodo tiene color asignado.
 - Tomamos un color *colorActual* := 1.
 - Para cada uno de los nodos sin colorear:
 - Comprobar si es posible asignarle el color actual.
 - Si se puede, se asigna. En otro caso, se deja sin colorear.
 - Si quedan nodos sin colorear, escoger otro color (*colorActual* := *colorActual* + 1) y volver al paso anterior.

El problema del coloreo de un grafo

- La estructura básica del esquema voraz se repite varias veces, una por cada color, hasta que todos los nodos estén coloreados.
- Función de **selección**: cualquier candidato restante.
- **Factible(x)**: se puede asignar un color a **x** si ninguno de sus adyacentes tiene ese mismo color.

```
para todo nodo y adyacente a x hacer
    si  $c_y == colorActual$  entonces
        devolver false
finpara
devolver true
```

- ¿Garantiza el algoritmo la solución óptima?

Implementacion del algoritmo

Función **COLOREO**

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

Begin

NuevoColor = \emptyset

Para cada vértice no coloreado v de G **Hacer**

Si v no es adyacente a ningún vértice en NuevoColor
Entonces

Marcar v como coloreado

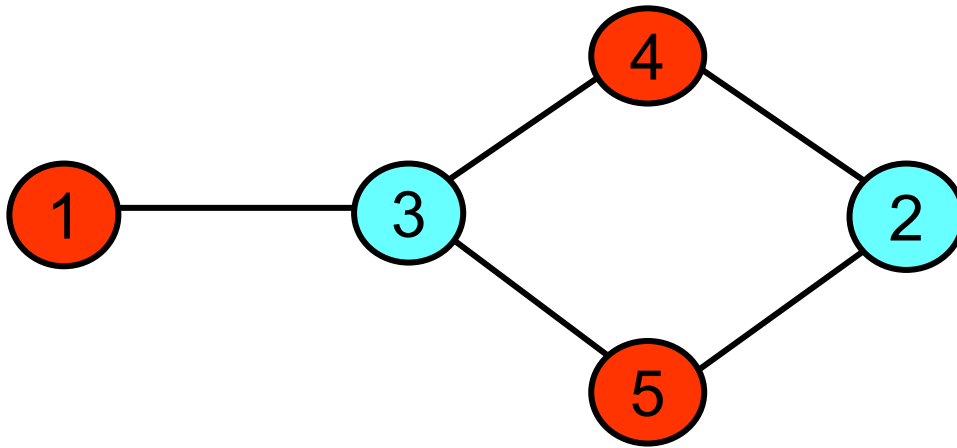
Añadir v a NuevoColor

End

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

Ejemplo

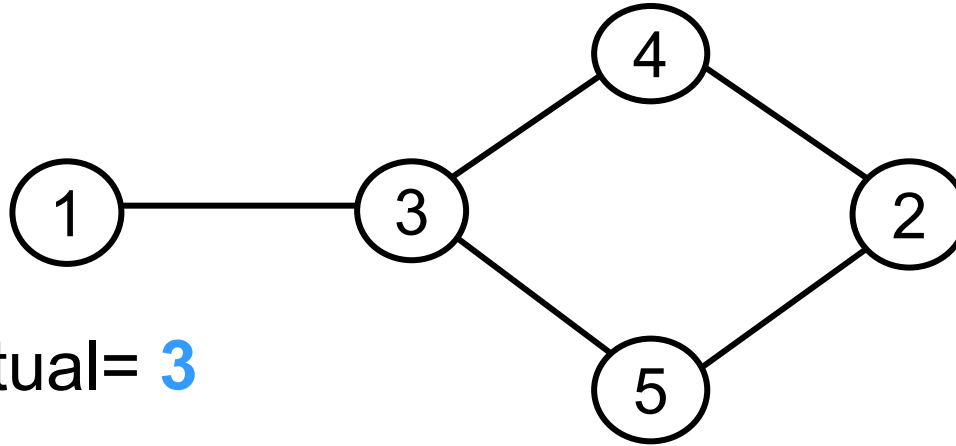
- **Representación de la solución:** una solución tiene la forma (c_1, c_2, \dots, c_n) , donde c_i es el color asignado al nodo i .
- La solución es válida si para toda arista $(v, w) \in A$, se cumple que $c_v \neq c_w$.



- $S = (1, 2, 2, 1, 1)$, Total: 2 colores

Ejemplo

Pero...

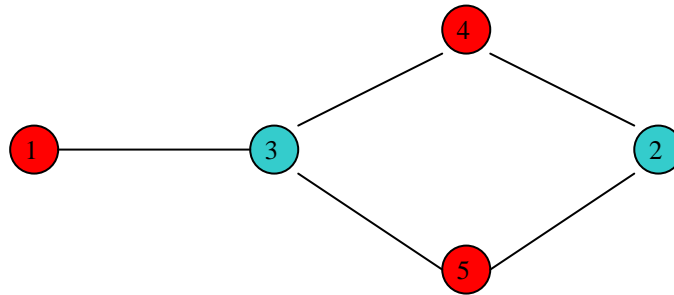


colorActual= 3

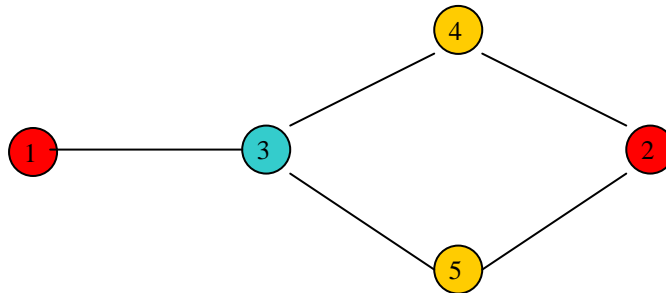
c_1	c_2	c_3	c_4	c_5

- **Resultado:** se necesitan 3 colores. Recordar que el óptimo es 2 colores.
- **Conclusión:** el algoritmo no es óptimo.

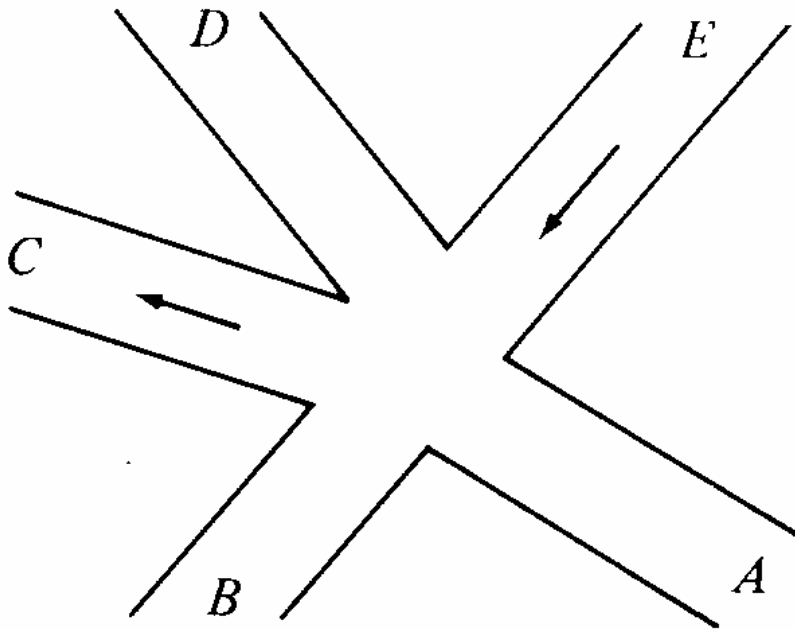
Ejemplo



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



- 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 vertices representan turnos, y cuyas aristas

unen esos turnos que no pueden realizarse simultáneamente sin que haya colisiones, y el problema del cruce con semáforos se convierte en un problema de coloreo de los vertices de un grafo

El problema de la Mochila

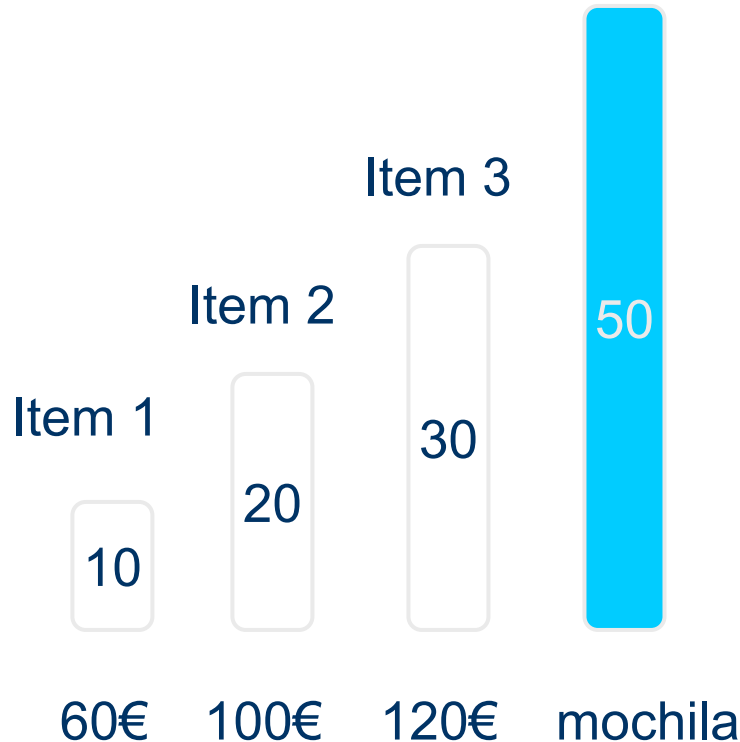
- Tenemos n objetos y una mochila. El objeto i tiene un peso w_i y la mochila tiene una capacidad M .
- Si metemos en la mochila la fracción x_i , $0 \leq x_i \leq 1$, del objeto i , generamos un beneficio de valor $p_i x_i$
- El objetivo es rellenar la mochila de tal manera que se maximice el beneficio que produce el peso total de los objetos que se transportan, con la limitación de la capacidad de valor M

$$\text{maximizar } \sum_{1 \leq i \leq n} p_i x_i$$

$$\text{sujeto a } \sum_{1 \leq i \leq n} w_i x_i \leq M$$

$$\text{con } 0 \leq x_i \leq 1, \quad 1 \leq i \leq n$$

Ejemplo: Mochila 0/1

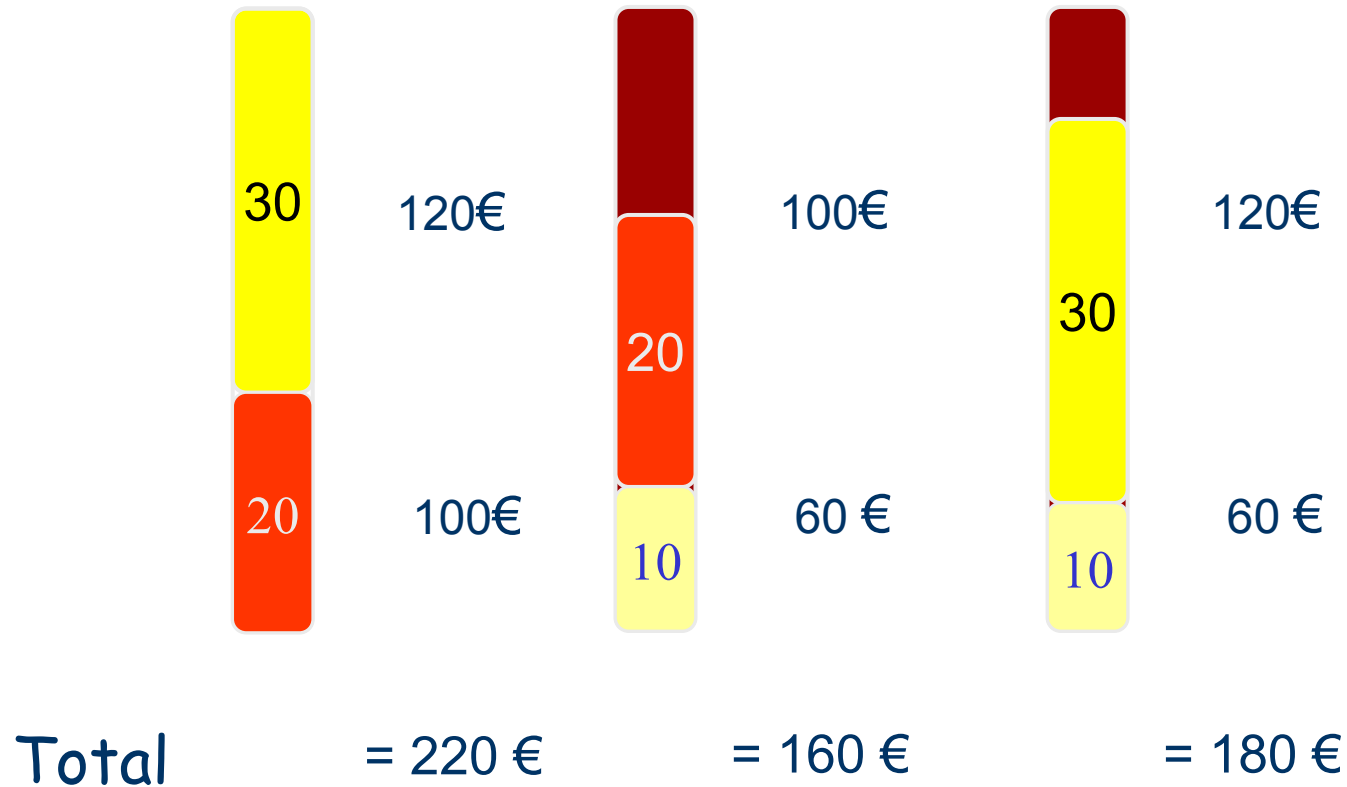


Es un claro problema de tipo greedy

Sus aplicaciones son innumerables

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

Ejemplo: Mochila 0/1



¿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 $P(I)/W(I) \geq P(I+1)/W(I+1)$. 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

Algoritmos voraces.

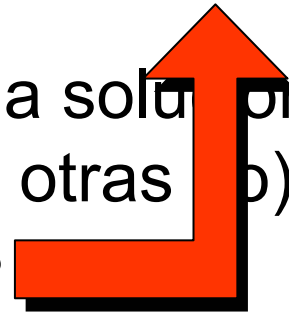
Conclusiones:

- Avance rápido se basa en una **idea intuitiva**:
 - Empezamos con una solución “vacía”, y la construimos paso a paso.
 - En cada paso se selecciona un candidato (el más prometedor) y se decide si se mete o no (función factible).
 - Una vez tomada una decisión no se vuelve a deshacer. Acabamos cuando tenemos una solución o nos quedamos sin candidatos.
- **Ojo**: en algunos problemas los algoritmos voraces sí garantizan la **solución óptima**.

Algoritmos voraces.

Conclusiones:

- **Primera cuestión:** ¿cuáles son los candidatos?, ¿cómo se representa una solución al problema?
- **Cuestión clave:** diseñar una función de selección adecuada.
 - Algunas pueden garantizar la solución óptima.
 - Otras pueden ser más heurísticas...
- **Función factible:** garantizar las **BACKTRACKING**
- En general los algoritmos voraces son la solución rápida a muchos problemas (a veces óptimas, otras no).
- ¿Y si podemos deshacer decisiones...?



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 basados en Programación Dinámica

Tema 6. Algoritmos para la Exploración de Grafos
 (“Backtracking”, “Branch and Bound”)

Tema 7. Otras metodologías algorítmicas