Algorítmica
Algoritmos Voraces

Conrado Martínez
U. Politècnica Catalunya

ALG Q1-2022–2023

Temario

Parte III

Algoritmos Voraces

1. Introducción a los Algoritmos Voraces
2. Compresión de Datos: Códigos de Huffman
3. Árboles de Expansión Mínimos: Algoritmos de Kruskal y de Prim
4. Particiones

Parte III

Algoritmos Voraces

Algoritmos Voraces

- Los algoritmos voraces suelen emplearse para resolver problemas de optimización combinatoria: Dada una entrada, buscamos una solución óptima al problema de acuerdo a una cierta función objetivo. Las soluciones consisten en una secuencia de elementos.
- Por ejemplo, dado un grafo $G = \langle V, E \rangle$ y dos vértices $u, v \in G$ buscamos un camino de u a v con el mínimo número de aristas. La solución es la secuencia de vértices o aristas que conforman el camino.

Algoritmos Voraces

- Los algoritmos voraces suelen emplearse para resolver problemas de optimización combinatoria: Dada una entrada, buscamos una solución óptima al problema de acuerdo a una cierta función objetivo. Las soluciones consisten en una secuencia de elementos.
- Por ejemplo, dado un grafo $G = \langle V, E \rangle$ y dos vértices $u, v \in G$ buscamos un camino de u a v con el mínimo número de aristas. La solución es la secuencia de vértices o aristas que conforman el camino.

Algoritmos Voraces

Un algoritmo voraz obtiene una solución óptima tomando una serie de decisiones irreversibles (no hay *backtracking*).

- Las decisiones tomadas son localmente óptimas (decisiones miopes) con el fin de consruir, incrementalmente, la solución globalmente óptima.
- Pero en muchas situaciones adoptar decisiones localmente óptimas **no** nos conduce a una solución globalmente óptima.
- A veces empleamos los algoritmos voraces para obtener soluciones aproximadas, subóptimas pero suficientemente cercanas a la optimalidad, y todo ello de manera mucho más eficiente

Algoritmos Voraces

Un algoritmo voraz obtiene una solución óptima tomando una serie de decisiones irreversibles (no hay *backtracking*).

- Las decisiones tomadas son localmente óptimas (decisiones miopes) con el fin de consruir, incrementalmente, la solución globalmente óptima.
- Pero en muchas situaciones adoptar decisiones localmente óptimas **no** nos conduce a una solución globalmente óptima.
- A veces empleamos los algoritmos voraces para obtener soluciones aproximadas, subóptimas pero suficientemente cercanas a la optimalidad, y todo ello de manera mucho más eficiente

Algoritmos Voraces

Un algoritmo voraz obtiene una solución óptima tomando una serie de decisiones irreversibles (no hay *backtracking*).

- Las decisiones tomadas son localmente óptimas (decisiones miopes) con el fin de consruir, incrementalmente, la solución globalmente óptima.
- Pero en muchas situaciones adoptar decisiones localmente óptimas **no** nos conduce a una solución globalmente óptima.
- A veces empleamos los algoritmos voraces para obtener soluciones aproximadas, subóptimas pero suficientemente cercanas a la optimalidad, y todo ello de manera mucho más eficiente

Algoritmos Voraces

- Los algoritmos voraces son por naturaleza iterativos; en cada iteración se toma una decisión miope e irreversible localmente óptima respecto a una "componente" de la solución, y a continuación se resuelve el subproblema resultante de tomar dicha decisión.
- Cada decisión puede depender de las decisiones anteriores, pero no de elecciones futuras. El esquema voraz también se emplea a menudo para diseñar algoritmos on-line, que deben tomar decisiones para una secuencia de peticiones, sin conocimiento de las decisiones futuras
- Un algoritmo voraz jamás revierte una decisión ya tomada (nunca hace *backtrack*).

Algoritmos Voraces

- Los algoritmos voraces son por naturaleza iterativos; en cada iteración se toma una decisión miope e irreversible localmente óptima respecto a una “componente” de la solución, y a continuación se resuelve el subproblema resultante de tomar dicha decisión.
- Cada decisión puede depender de las decisiones anteriores, pero no de elecciones futuras. El esquema voraz también se emplea a menudo para diseñar algoritmos on-line, que deben tomar decisiones para una secuencia de peticiones, sin conocimiento de las decisiones futuras
- Un algoritmo voraz jamás revierte una decisión ya tomada (nunca hace *backtrack*).

Algoritmos Voraces

- Los algoritmos voraces son por naturaleza iterativos; en cada iteración se toma una decisión miope e irreversible localmente óptima respecto a una “componente” de la solución, y a continuación se resuelve el subproblema resultante de tomar dicha decisión.
- Cada decisión puede depender de las decisiones anteriores, pero no de elecciones futuras. El esquema voraz también se emplea a menudo para diseñar algoritmos on-line, que deben tomar decisiones para una secuencia de peticiones, sin conocimiento de las decisiones futuras
- Un algoritmo voraz jamás revierte una decisión ya tomada (nunca hace *backtrack*).

Algoritmos Voraces

Para que una estrategia voraz funcione correctamente es necesario que el problema consdierado reúna dos características:

- Propiedad de la decisión voraz (*greedy choice property*): un óptimo global es alcanzable tomando decisiones localmente óptimas.
- Subestructura óptima: una vez tomadas una serie de decisiones locales, existe una solución globalmente óptima del problema original que contiene la solución parcial representada por las decisiones ya tomadas.

Frecuentemente, el criterio de optimización local usado para tomar las decisiones voraces induce un orden global que se utilizará para ordenar la toma de decisiones del algoritmo voraz.

El problema de la Mochila Fraccionaria

Dado un conjunto de n objetos, cada uno con peso $w_i > 0$ y valor $v_i > 0$, y una capacidad o peso máximo admisible $W > 0$ de la mochila, el objetivo es determinar un conjunto de objetos o fracciones de objetos que maximicen el valor acumulado y cuyo peso combinado no exceda W.

En definitiva, queremos determinar los valores $x_i \in [0, 1]$, $1 \leq i \leq n$, tales que

$$\sum_{i=1}^{n} x_i v_i$$

es máximo y $\sum_{i=1}^{n} x_i w_i \leq W$. Asumiremos, sin pérdida de generalidad, que $\sum_{i=1}^{n} w_i > W$, de otro modo la solución buscada es trivialmente $x_i = 1$ para todo i.

El problema de la Mochila Fraccionaria

Ejemplo

$$n = 5, W = 100$$

Item	1	2	3	4	5
w	10	20	30	40	50
v	20	30	66	40	60

El problema de la Mochila Fraccionaria

procedure GREEDYFKNAPSACK(n, v, w, W)
  $O := \{1, \ldots, n\}$; $val := 0$;
  while $W > 0$ **do**
    Sea $i \in O$ un elemento con la propiedad P
    if $w[i] \leq W$ **then**
      $x[i] := 1$; $W := W - w[i]$; $val := val + v[i]$;
    else
      $x[i] := W/w[i]$; $val := val + v[i] * x[i]$; $W := 0$
    end if
    Eliminar i de O
  end while
  return x
end procedure

El problema de la Mochila Fraccionaria

Criterio #1: el objeto más valioso

Ejemplo

$n = 5, W = 100$

Item	1	2	3	4	5
w	10	20	30	40	50
v	20	30	66	40	60
x	0	0	1	0.5	1

Valor total 146

El problema de la Mochila Fraccionaria

Criterio #2: el objeto más ligero

Ejemplo

$n = 5, W = 100$

Item	1	2	3	4	5
w	10	20	30	40	50
v	20	30	66	40	60
x	1	1	1	1	0

Valor total 156

El criterio #1 (el objeto más valioso) no nos da la solución óptima!

El problema de la Mochila Fraccionaria

Criterio #3: el objeto con mayor valor por unidad de peso (mayor v/w)

Ejemplo

$n = 5, W = 100$

Item	1	2	3	4	5
w	10	20	30	40	50
v	20	30	66	40	60
v/w	2.0	1.5	2.2	1.0	1.2
x	1	1	1	0	0.8

Valor total 164

El criterio #2 (el objeto más ligero) tampoco nos da una solución óptima!

El problema de la Mochila Fraccionaria

Teorema

Si en GREEDYFKNAPSACK *seleccionamos en cada iteración el objeto con mayor ratio valor/peso, obtendremos siempre una solución óptima del problema de la Mochila Fraccionaria.*

Demostración

Supongamos que ordenamos (y reindexamos) los objetos de tal modo que

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$$

El problema de la Mochila Fraccionaria

Demostración (continúa)

Sea $X = (x_1, \ldots, x_n)$, donde $x_i \in [0, 1]$, la solución voraz. Bajo la suposición de que $\sum w_i > W$ al menos una $x_i < 1$, ya que no podremos colocar todos los objetos enteramente. Sea j el menor índice tal que $x_j < 1$. Por las propiedades del algoritmo tendremos:

1. $x_i = 1$, para toda $i < j$
2. $x_i = 0$, para toda $i > j$
3. Adicionalmente, toda la capacidad de la mochila es usada: $\sum_{i=1}^{n} x_i w_i = W$

El problema de la Mochila Fraccionaria

Demostración (continúa)

Sea $Y = (y_1, \ldots, y_n)$, $y_i \in [0, 1]$, una solución factible, i.e.,

$$\sum_{i=1}^{n} y_i w_i \leq W$$

- Se cumple $\sum_{i=1}^{n} y_i w_i \leq W = \sum_{i=1}^{n} x_i w_i$
- Por lo tanto
  $0 \leq \sum_{i=1}^{n} x_i w_i - \sum_{i=1}^{n} y_i w_i = \sum_{i=1}^{n} (x_i - y_i) w_i$
- Y la diferencia de valor de las soluciones X e Y es

$$v(X) - v(Y) = \sum_{i=1}^{n} x_i v_i - \sum_{i=1}^{n} y_i v_i = \sum_{i=1}^{n} (x_i - y_i) v_i$$

$$= \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_i}{w_i}$$

El problema de la Mochila Fraccionaria

Demostración (continúa)

Sea $Y = (y_1, \ldots, y_n)$, $y_i \in [0, 1]$, una solución factible, i.e.,

$$\sum_{i=1}^{n} y_i w_i \leq W$$

- Se cumple $\sum_{i=1}^{n} y_i w_i \leq W = \sum_{i=1}^{n} x_i w_i$
- Por lo tanto
  $0 \leq \sum_{i=1}^{n} x_i w_i - \sum_{i=1}^{n} y_i w_i = \sum_{i=1}^{n} (x_i - y_i) w_i$
- Y la diferencia de valor de las soluciones X e Y es

$$v(X) - v(Y) = \sum_{i=1}^{n} x_i v_i - \sum_{i=1}^{n} y_i v_i = \sum_{i=1}^{n} (x_i - y_i) v_i$$

$$= \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_i}{w_i}$$

El problema de la Mochila Fraccionaria

Demostración (continúa)

Sea $Y = (y_1, \ldots, y_n)$, $y_i \in [0, 1]$, una solución factible, i.e.,

$$\sum_{i=1}^{n} y_i w_i \leq W$$

- Se cumple $\sum_{i=1}^{n} y_i w_i \leq W = \sum_{i=1}^{n} x_i w_i$
- Por lo tanto
  $0 \leq \sum_{i=1}^{n} x_i w_i - \sum_{i=1}^{n} y_i w_i = \sum_{i=1}^{n} (x_i - y_i) w_i$
- Y la diferencia de valor de las soluciones X e Y es

$$v(X) - v(Y) = \sum_{i=1}^{n} x_i v_i - \sum_{i=1}^{n} y_i v_i = \sum_{i=1}^{n} (x_i - y_i) v_i$$

$$= \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_i}{w_i}$$

El problema de la Mochila Fraccionaria

Demostración (continúa)

Vamos a acotar $v(X) - v(Y) = \sum_{i=1}^{n}(x_i - y_i)w_i\frac{v_i}{w_i}$.

- Si $i < j$ entonces $x_i = 1$, luego $x_i - y_i \geq 0$ y puesto que $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$, tendremos

$$(x_i - y_i)\frac{v_i}{w_i} \geq (x_i - y_i)\frac{v_j}{w_j}$$

- Si $i > j$, $x_i = 0$, y $x_i - y_i \leq 0$ pero puesto que $\frac{v_i}{w_i} \leq \frac{v_j}{w_j}$, ahora tenemos

$$(x_i - y_i)\frac{v_i}{w_i} \geq (x_i - y_i)\frac{v_j}{w_j}$$

- Tenemos la misma desigualdad tanto si $i < j$ como si $i > j$.

El problema de la Mochila Fraccionaria

Demostración (continúa)

Vamos a acotar $v(X) - v(Y) = \sum_{i=1}^{n}(x_i - y_i)w_i\frac{v_i}{w_i}$.

- Si $i < j$ entonces $x_i = 1$, luego $x_i - y_i \geq 0$ y puesto que $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$, tendremos

$$(x_i - y_i)\frac{v_i}{w_i} \geq (x_i - y_i)\frac{v_j}{w_j}$$

- Si $i > j$, $x_i = 0$, y $x_i - y_i \leq 0$ pero puesto que $\frac{v_i}{w_i} \leq \frac{v_j}{w_j}$, ahora tenemos

$$(x_i - y_i)\frac{v_i}{w_i} \geq (x_i - y_i)\frac{v_j}{w_j}$$

- Tenemos la misma desigualdad tanto si $i < j$ como si $i > j$.

El problema de la Mochila Fraccionaria

Demostración (continúa)

Vamos a acotar $v(X) - v(Y) = \sum_{i=1}^{n}(x_i - y_i)w_i\frac{v_i}{w_i}$.

- Si $i < j$ entonces $x_i = 1$, luego $x_i - y_i \geq 0$ y puesto que $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$, tendremos

$$(x_i - y_i)\frac{v_i}{w_i} \geq (x_i - y_i)\frac{v_j}{w_j}$$

- Si $i > j$, $x_i = 0$, y $x_i - y_i \leq 0$ pero puesto que $\frac{v_i}{w_i} \leq \frac{v_j}{w_j}$, ahora tenemos

$$(x_i - y_i)\frac{v_i}{w_i} \geq (x_i - y_i)\frac{v_j}{w_j}$$

- Tenemos la misma desigualdad tanto si $i < j$ como si $i > j$.

El problema de la Mochila Fraccionaria

Demostración (continúa)

- Usando las desigualdades obtenidas

$$v(X) - v(Y) = \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_i}{w_i}$$

$$\geq \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_j}{w_j} \geq \frac{v_j}{w_j} \sum_{i=1}^{n} (x_i - y_i) w_i \geq 0$$

- Puesto que $v(X) - v(Y) \geq 0$ para cualquier solución factible Y concluímos que X es una solución óptima.

El problema de la Mochila Fraccionaria

Demostración (continúa)

- Usando las desigualdades obtenidas

$$v(X) - v(Y) = \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_i}{w_i}$$

$$\geq \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_j}{w_j} \geq \frac{v_j}{w_j} \sum_{i=1}^{n} (x_i - y_i) w_i \geq 0$$

- Puesto que $v(X) - v(Y) \geq 0$ para cualquier solución factible Y concluímos que X es una solución óptima.

El problema de la Mochila Fraccionaria

procedure GREEDYFKNAPSACK(n, v, w, W)
  $O := \{1, \ldots, n\}$; $val := 0$;
  while $W > 0$ **do**
    Sea $i \in O$ un elemento con v_i/w_i máximo
    if $w[i] \leq W$ **then**
      $x[i] := 1$; $W := W - w[i]$; $val := val + v[i]$;
    else
      $x[i] := W/w[i]$; $val := val + v[i] * x[i]$; $W := 0$
    end if
    Eliminar i de O
  end while
  return x
end procedure

Coste? $\mathcal{O}(n^2)$

Podemos obtener un coste menor?

El problema de la Mochila Fraccionaria

```
procedure GreedyFKnapsack(n, v, w, W)
    O := {1, ..., n}; val := 0; i := 1;
    Ordenar O por orden descedente de ratio v/w
    while W > 0 ∧ i < n do
        if w[i] ≤ W then
            x[i] := 1; W := W − w[i]; val := val + v[i];
        else
            x[i] := W/w[i]; val := val + v[i] ∗ x[i]; W := 0
        end if
        i := i + 1
    end while
    return x
end procedure
```

Coste? $\mathcal{O}(n \log n)$

El problema de la Mochila Fraccionaria

Teorema

El problema de la Mochila Fraccionaria puede resolverse en tiempo $\mathcal{O}(n \log n)$.

Por contra, el problema de la Mochila Entera o 0-1 KNAPSACK, en el que cada objeto puede ser colocado entero o no colocado en absoluto en la mochila ($x_i \in \{0, 1\}$) es un problema NP-duro.

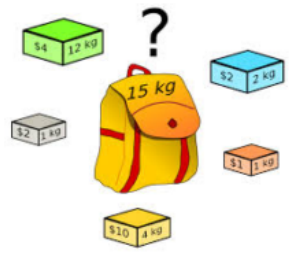
No obstante, la estrategia voraz es una buena heurística para obtener soluciones aproximadas (y también puede usarse para mejorar el rendimiento de algoritmos de backtracking y branch& bound, al permitir mejores podas al explorar espacio de soluciones).

El problema de la Mochila Fraccionaria

Ejemplo

$n = 3, W = 50$

Item	1	2	3
w	10	20	30
v	60	100	120
v/w	6	5	4



Con el criterio voraz tomaríamos $x = (1, 1, 0)$ y el valor total sería 160. No es una solución óptima. La solución $y = (0, 1, 1)$ alcanza un valor total de 220.

Planificación de Tareas y Actividades

Planteamiento general:

- Se nos dan n tareas o actividades, cada una de ellas con diferentes características (por ejemplo, duración, tiempo de inicio, coste de ejecución, ...), que han de ser procesadas en un sistema con un solo procesador/múltiples procesadores.
- El objetivo es determinar una planificación (*schedule*), es decir, cúando y dónde ejecutar cada tarea, y qué tareas no podrán ser ejecutadas, eventualmente, todo ello con el fin de optimizar un cierto criterio (minimizar el tiempo total de ejecución, el coste, maximizar el *throughput*, ...)

Problemas de planificación (con un único procesador)

1. INTERVAL SCHEDULING: Cada tarea tiene un tiempo de inicio y de final. En un instante de tiempo dado solo puede haber una tarea en ejecución. El objetivo es maximizar el número total de tareas ejecutadas.
2. WEIGHTED INTERVAL SCHEDULING: Como el anterior, pero cada tarea tiene un beneficio. El objetivo es maximizar el beneficio total obtenido con las tareas ejecutadas.
3. JOB SCHEDULING (minimización del retraso): Las tareas pueden comenzar a ejecutarse en cualquier momento, siempre que el procesador esté libre; cada tarea tiene una duración t_i y un tiempo límite d_i (deadline); se define el retraso ℓ_i como el tiempo que transcurre entre el deadline y el instante en que finaliza la ejecución de la tarea si dicho instante es posterior al deadline y 0 si es anterior ($\ell_i = \max(0, s_i + t_i - d_i)$). El objetivo es asignar tiempos de inicio s_i a las tareas de manera que en cada momento solo hay una tarea en ejecución y se minimiza el retraso máximo.

Interval scheduling

El problema de INTERVAL SCHEDULING (también conocido como de *selección de actividades* consiste en determinar un subconjunto de tareas que pueden ser ejecutadas sin conflicto y de máxima cardinalidad.

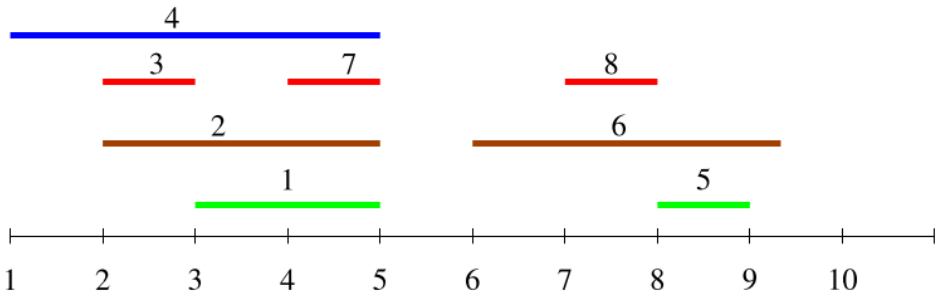
- Cada tarea i, $1 \leq i \leq n$ tiene un tiempo de inicio s_i y un tiempo de finalización f_i, con $s_i < f_i$.
- Solo existe un procesador para ejecutar las tareas, en un instante cualquiera de tiempo dado solo puede estar ejecutando **una** tarea.
- Cada tarea debe ejecutarse en su totalidad, desde su tiempo de inicio a su tiempo de finalización, o no ser ejecutada en absoluto.

El objetivo es hallar un subconjunto de tareas mutuamente compatibles de cardinalidad máxima. Dos tareas i y j distintas son compatibles si $f_i \leq s_j$ o $f_j \leq s_i$.

Interval scheduling

Ejemplo

Tarea :	1	2	3	4	5	6	7	8
Inicio (s_i):	3	2	2	1	8	6	4	7
Fin (f_i):	5	5	3	5	9	9	5	8



Interval scheduling

Criterio #1: tiempo de finalización más temprano (*earlier finish time*)

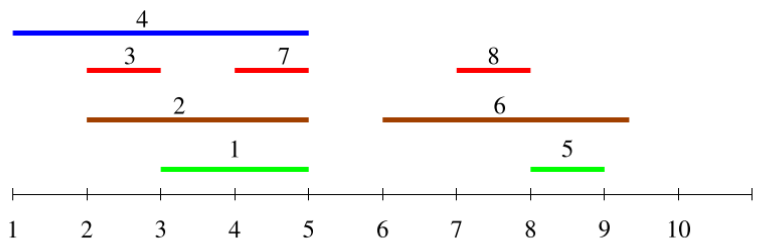
procedure INTERVALSCHEDULING(A)
  $S := \emptyset$; $T := \{1, \ldots, n\}$;
  while $T \neq \emptyset$ **do**
    Sea i la tarea con mínimo f_i en T
    $S := S \cup \{i\}$;
    Eliminar i de T, y todas las tareas $j \in T$ t.q. $s_j < f_i$
  end while
  return S
end procedure

Interval scheduling

Criterio #1: tiempo de finalización más temprano (*earlier finish time*)

Tarea:	3 4 2 1 7 8 5 6
s_i:	3 1 2 3 4 7 8 6
f_i:	3 5 5 5 5 8 9 9

Sol: 3 1 8 5



Interval scheduling: Corrección

Teorema

El algoritmo voraz con el criterio de tiempo de finalización más temprano obtiene un solución óptima del problema INTERVAL SCHEDULING.

Demostración

Probaremos que existe una solución óptima S_{OPT} que contiene la tarea i con tiempo de finalización f_i mínimo en T. Todas las tareas con $s_j < f_i$ son incompatibles la tarea i.

En consecuencia, $S_{\text{OPT}} \setminus \{i\}$ tiene que ser necesariamente una solución óptima para el conjunto de tareas

$$T' = T \setminus (\{i\} \cup \{j \in T : s_j < f_i\}),$$

pues de otro modo tendríamos una contradicción $\Longrightarrow$ subestructura óptima

Interval scheduling: Corrección

Demostración (continúa)

Demostraremos que al menos una solución óptima incluye una tarea con tiempo de finalización mínimo, y lo haremos por reducción al absurdo.

Sea i una tarea con mínimo f_i y supongamos que para cualquier solución óptima S_{OPT}, tenemos $i \notin S_{\text{OPT}}$. Consideremos una solución óptima S_{OPT} y sea k una tarea en S_{OPT} con tiempo de finalización mínimo. Por hipótesis, $f_k > f_i$.

Interval scheduling: Corrección

Demostración (continúa)

Toda otra tarea k' en S tiene que empezar después de que k termine, $f_k < s_{k'}$, pues en caso contrario no serían compatibles k y k' (tendríamos, sino, $s_{k'} \leq f_k < f_{k'}$!!). Esto significa que todas las tareas k' también son compatibles con la tarea i (en efecto: $f_i < f_k \leq s_{k'}$)

Por lo tanto $S' = S_{\text{OPT}} \cup \{i\} \setminus \{k\}$ es también una solución óptima ya que $|S'| = |S_{\text{OPT}}|$, pero contiene a la tarea i, en contradicción con la hipótesis inicial. $\square$

Interval scheduling: Coste

procedure INTERVALSCHEDULING(A)
  $S := \emptyset$; $T := \{1, \ldots, n\}$;
  while $T \neq \emptyset$ **do**
    Sea i la tarea con menor f_i en T
    $S := S \cup \{i\}$;
    Eliminar i y todas las tareas $j \in T$ t.q. $s_j < f_i$
  end while
  return S
end procedure

En una implementación ingenua hallar la tarea i y su eliminación de T tiene coste $\Theta(n)$ y el coste total es $\Theta(n^2)$.

Interval scheduling: Coste

procedure INTERVALSCHEDULING2(A)
    Ordenar A en orden ascendente de tiempo de finalización
    ▷ Sea $[a_1, \ldots, a_n]$ la lista resultante
    $S := \emptyset$; $j := 0$; $A[0].f := -\infty$
    for $i := 1$ **to** n **do**
        if $A[i].s \geq A[j].f$ **then**
            $S := S \cup \{i\}$; $j := i$
        end if
    end for
    return S
end procedure

El coste de INTERVALSCHEDULING2 es $\Theta(n \log n)$ correspondiente a la ordenación; el bucle **for** tiene coste $\Theta(n)$.

Interval scheduling: Coste

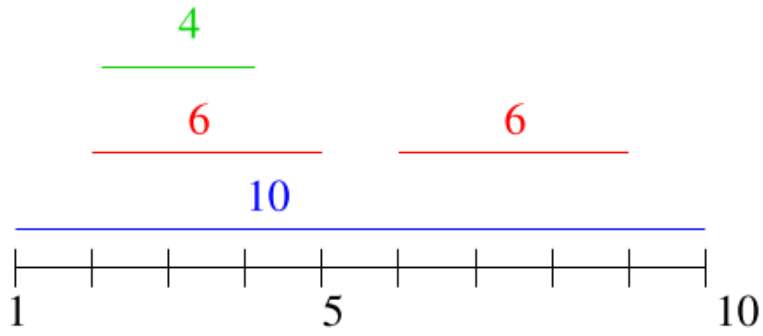
Las tareas se consideran por tiempo de finalización de menor a mayor (según el criterio voraz que conduce al óptimo); en todo momento j es el índice de la última tarea seleccionada y si la tarea $A[i]$, con $i > j$, cumple $A[i].s < A[j].f$ entonces i y j son incompatibles. En caso contrario,la tarea i es la tarea con tiempo de finalización mínimo compatible con las tareas ya seleccionadas.

Por lo tanto INTERVAL2SCHEDULING nos devolverá una solución óptima en tiempo $\Theta(n \log n)$.

Si los tiempos de finalización están dentro de un rango no muy grande podríamos emplear un método de ordenación digital para rebajar el coste del algoritmo a $\Theta(n)$.

Weighted Activity Selection

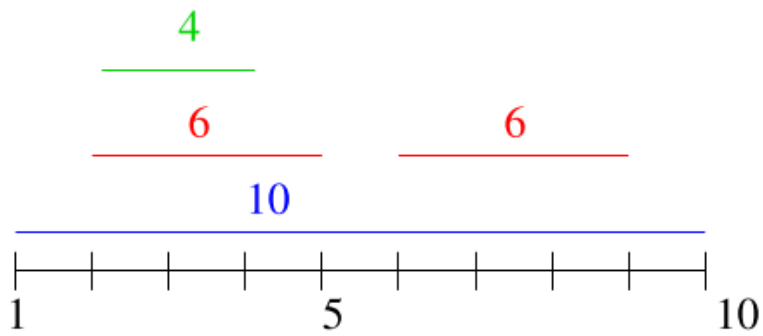
Generaliza el problema anterior: dadas n tareas con tiempos de inicio s_i, tiempos de finalización f_i y pesos w_i, el objetivo es hallar una selección de tareas mutuamente compatibles cuyo peso total es máximo.



INTERVALSCHEDULING2 seleccionaría la tarea en verde y la segunda tarea en rojo con peso total 10, lo cual **no** es óptimo.

Weighted Activity Selection

Si el criterio fuera escoger la tarea con mayor peso mutuamente compatible con las ya seleccionadas, la solución obtenida **tampoco es óptima**



El nuevo algoritmo voraz selecciona la tarea azul con peso 10; la solución óptima es seleccionar las dos tareas rojas con peso total 12.

En este problema fracasa todo criterio voraz. Existen no obstante soluciones “eficientes” con Programación Dinámica si los pesos no son muy grandes (tienen $\Theta(\log n)$ bits).

Lateness Minimization

Tenemos n tareas cuya ejecución puede comenzar en cualquier momento, siempre que el procesador esté libre; cada tarea tiene una duración t_i y un tiempo límite d_i (deadline).
El objetivo de la planificación es hallar tiempos de inicio s_i para todas las tareas de tal modo que:

1. Cualesquiera tareas i y j, $i \neq j$, no solapan en su ejecucíón: $(s_i, f_i) \cap (s_j, f_j) = \emptyset$ donde $f_k = s_k + t_k$, $k = 1, \ldots, n$
2. Se minimiza el retraso (*lateness*) máximo $L = \max_{1 \leq i \leq n} \ell_i$, donde el retraso de la tarea i es

$$\ell_i = \max(0, f_i - d_i),$$

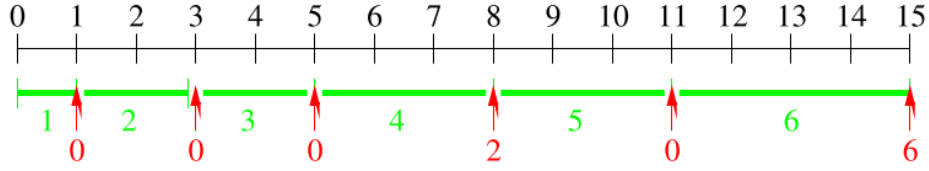
el tiempo en exceso más allá del *deadline* de la tarea.

Lateness Minimization

Ejemplo

t	1	2	2	3	3	4
d	9	8	15	6	14	9
s	0	1	3	5	8	11
ℓ	0	0	0	2	0	6

$L = 6$



Lateness Minimization

procedure GENERICLATENESS(A)
    Ordenar A de acuerdo al criterio X
    $S[1] := 0$; $t := A[1].t$; $L := \max(0, t - A[1].d)$;
    for $i := 2$ **to** n **do**
        $S[i] := t$;
        $t := t + A[i].t$;
        $L := \max(L, \max(0, t - A[i].d))$;
    end for
    return $\langle S, L \rangle$
end procedure

Lateness Minimization

Criterio #1: tareas con menor duración primero

i	t_i	d_i
1	1	6
2	5	5

$s_1 = 0, s_2 = 1$ tiene retraso $L = 1$, pero
$s_1 = 5, s_2 = 0$ tiene menor retraso $L = 0$!!

Lateness Minimization

Criterio #2: tareas con tiempo de inicio sin retraso más tardío (equivale a $d_i - t_i$ mínimo)

i	t_i	d_i	$d_i - t_i$
1	1	3	2
2	9	10	1

$s_2 = 0, s_1 = 9$ no minimiza el retraso

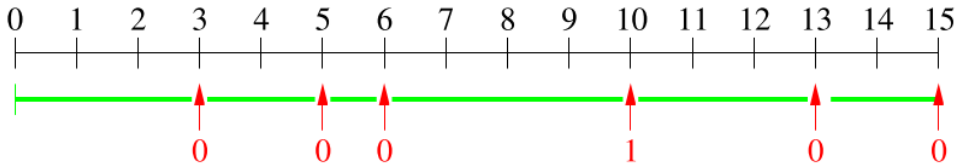
Lateness Minimization

Criterio #3: tareas con *deadline* más temprano (=las tareas más urgentes primero)

Ordenar por d_i creciente

procedure LATENESSURGENT(A)
  Ordenar A por d_i creciente
  $S[1] := 0$; $t := A[1].t$;
  $L := \text{máx}(0, t - A[1].d)$;
  for $i := 2$ **to** n **do**
    $S[i] := t$; $t := t + A[i].t$;
    $L := \text{máx}(L, \text{máx}(0, t - A[i].d))$
  end for
  return $\langle S, L \rangle$
end procedure

i	t	d	nuevo i
1	1	9	3
2	2	8	2
3	2	15	6
4	3	6	1
5	3	14	5
6	4	9	4



Urgentes Primero: Coste

```
procedure LatenessUrgent(A)
    Ordenar A por d_i creciente
    S[1] := 0; t := A[1].t; L := máx(0, t − A[0].d);
    for i := 2 to n do
        s_i := t; t := t + A[i].t;
        L := máx(L, máx(0, t − A[i].d))
    end for
    return ⟨S, L⟩
end procedure
```

Coste del bucle: $\Theta(n)$

Coste total: $\Theta(n \log n)$

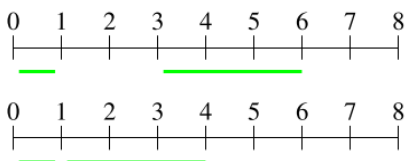
Urgentes Primero: Corrección

Lema

Existe una planificación óptima de las tareas de manera que en todo instante de tiempo hay alguna tarea en ejecución.

Demostración

Para cualquier planificación en la que haya momentos "ociosos" (ninguna tarea en ejecución) podemos encontrar una planificación alternativa eliminando los "huecos" de modo que no haya instantes de tiempo sin ninguna tarea en ejecución y el retraso total sea el mismo o menor. □



N.B.: LATENESSURGENT genera una planificación sin momentos ociosos.

Urgentes Primero: Corrección

Diremos que una planificación S tiene una inversión si $s_i < s_j$ pero $d_i > d_j$.

Si indexamos las tareas de manera que $d_0 \leq d_1 \leq \cdots \leq d_{n-1}$ entonces la planificación generada por LATENESSURGENT no tiene inversiones. Dando por sentado que todas las tareas tienen duración no nula ($t_i > 0$) tendremos que en la planificación de LATENESSURGENT cumple $s_0 < s_1 < \cdots < s_{n-1}$.

Lema

Dada una planificación S que contiene inversiones, el intercambio de dos tareas consecutivas invertidas reduce el número de inversiones en una unidad pero no incrementa el retraso total.

Urgentes Primero: Corrección

Demostración

Consideremos que en S la tarea i se ejecuta justo antes que la tarea j (son las tareas k-ésima y $(k+1)$-ésima en S, respectivamente) y que $d_i > d_j$, esto es, forman una inversión.
Nota: Si indexamos por d creciente tendremos que la primera tarea ejecutada es la 1, a continuación la 2, etc.
Si hay una inversión habrá alguna tarea $j \neq i$ que se ejecuta la i-ésima. Sea i el menor índice para el que eso ocurre. Entonces deducimos que $j > i$ y $s_j < s_i$ puesto que la tarea i se tendrá que ejecutar más tarde; pero $d_i < d_j$. Si S tiene inversiones siempre existirá al menos una inversión que involucra a dos tareas que se ejecutan consecutivamente.

Urgentes Primero: Corrección

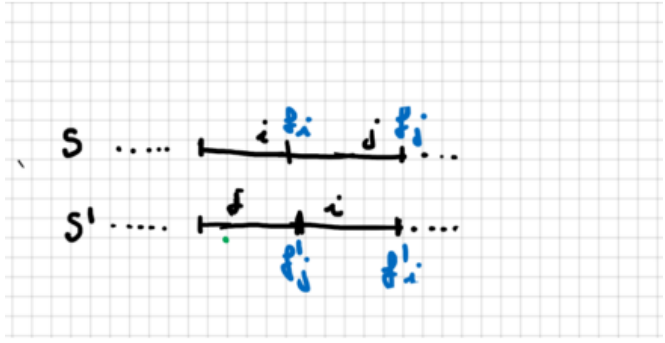
Demostración (continúa)

Si intercambiamos el orden de ejecución de las tareas i y j obtenemos una nueva planificación S' en la que $s'_m = s_m$ excepto si $m = i$ o $m = j$, y por tanto, los retrasos ℓ'_m en S' no cambian salvo que $m \notin \{i, j\}$, i.e., $\ell'_m = \ell_m$. Solo tendremos que ℓ'_i y ℓ'_j pueden cambiar y ser diferentes de ℓ_i y ℓ_j, respectivamente.

Urgentes Primero: Corrección

Demostración (continúa)

- Sean $f_i = s_i + t_i$, $f_j = s_j + t_j$ los tiempos de finalización en S y $f'_i = s'_i + t_i$, $f'_j = s'_j + t_j$ los tiempos de finalización en S'.
- $s_j = f_i$ y $f_i < f_j$; $s'_i = f'_j$ y $f'_j < f'_i$
- $s'_j = s_i$ y $f'_i = f_j$ ($\implies f'_j < f_j$)
- $f_j \leq d_j$ (recordemos: $d_j < d_i$)



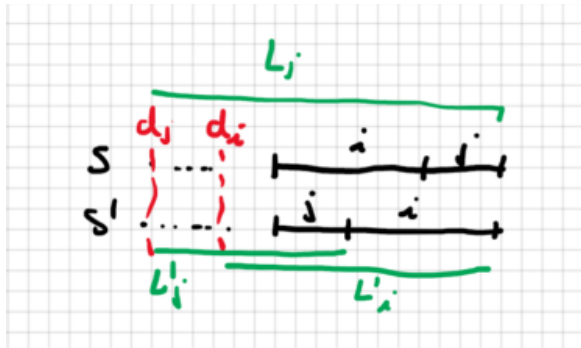
$f'_j < f_j = f'_i \leq d_j < d_i \wedge f_i < f_j < d_i \implies \ell'_i = \ell'_j = \ell_i = \ell_j = 0$

S y S' tienen el mismo retraso máximo

Urgentes Primero: Corrección

Demostración (continúa)

- Sean $f_i = s_i + t_i$, $f_j = s_j + t_j$ los tiempos de finalización en S y $f'_i = s'_i + t_i$, $f'_j = s'_j + t_j$ los tiempos de finalización en S'.
- $s_j = f_i$ y $f_i < f_j$; $s'_i = f'_j$ y $f'_j < f'_i$
- $s'_j = s_i$ y $f'_i = f_j$ ($\implies f'_j < f_j$)
- Si $d_i < f_i$ (recordemos: $d_j < d_i$)

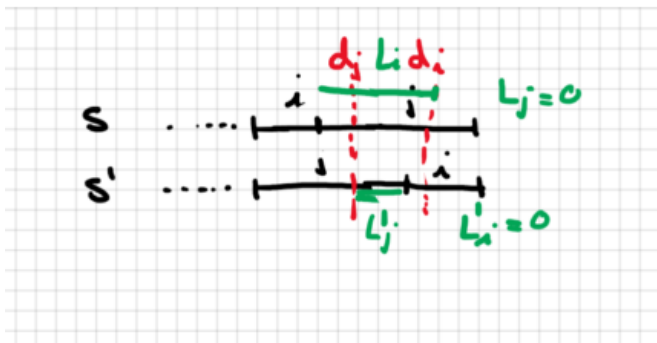


$\ell'_j = \max(0, f'_j - d_j) \leq \max(0, f_j - d_j) = \ell_j$
$\ell'_i = \max(0, f'_i - d_i) = \max(0, f_j - d_i) \leq$
$\max(0, f_j - d_j) = \ell_j$
$\max\{\ell'_i, \ell'_j\} \leq \ell_j \leq \max\{\ell_i, \ell_j\} \implies L' \leq L$
S' tiene el mismo o menor retraso máximo que S

Urgentes Primero: Corrección

Demostración (continúa)

- Sean $f_i = s_i + t_i$, $f_j = s_j + t_j$ los tiempos de finalización en S y $f'_i = s'_i + t_i$, $f'_j = s'_j + t_j$ los tiempos de finalización en S'.
- $s_j = f_i$ y $f_i < f_j$; $s'_i = f'_j$ y $f'_j < f'_i$
- $s'_j = s_i$ y $f'_i = f_j$ ($\implies f'_j < f_j$)
- Si $f_i \leq d_i \leq f_j$ (recordemos: $d_j < d_i$)



$$\ell'_j = f'_j - d_j \leq f_j - d_j = \ell_j$$
$$\ell'_i = f'_i - d_i = f_j - d_i \leq f_j - d_j = \ell_j$$
$$\text{máx}\{\ell'_i, \ell'_j\} \leq \ell_j = \text{máx}\{\ell_i, \ell_j\} \implies L' \leq L$$

S' tiene el mismo o menor retraso máximo que S

Urgentes Primero: Corrección

Demostración (continúa)

En los tres casos posibles, el retraso total de S' es menor o igual que el de S. □

Urgentes Primero: Conclusión

Teorema

El algoritmo LATENESSURGENT *obtiene una planificación óptima con retraso total mínimo en tiempo* $\mathcal{O}(n \log n)$

Demostración

Si consideramos una planificación S_{opt} que tiene inversiones y menor retraso total que LATENESSURGENT llegamos a una contradicción.

Podemos eliminar todo momento ocioso de la planificación (por el primer lema) e ir eliminando sucesivamente todas las inversiones manteniendo el retraso total óptimo (por el lema anterior); no podemos mejorarlo ya que asumimos que el retraso de S_{opt} es mínimo.

Urgentes Primero: Conclusión

Demostración (continúa)

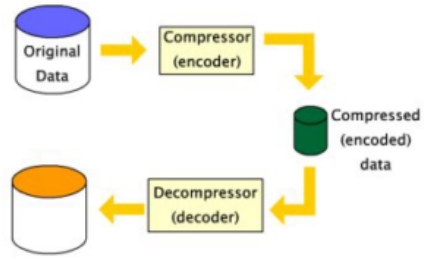
Tras eliminar todas las inversiones habremos llegado a una planificación sin momentos ociosos y que no contiene ninguna inversión, es decir, que asigna los tiempos de inicio por orden creciente de deadline, esto es, la planificación que nos da LATENESSURGENT y el retraso total es idéntico al de S_{opt}! □

Parte III

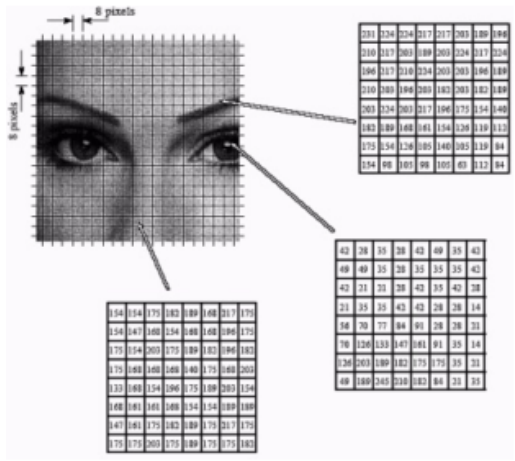
Algoritmos Voraces

Compresión de Datos

Nos dan un texto T que utiliza símbolos de un alfabeto finito Σ. Nuestro objetivo es representar T con el mínimo número de bits posible.

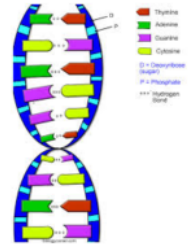


La compresión de datos (*data compression*) tiene como finalidad reducir el tiempo de transmisión de archivos grandes y/o reducir el espacio de almacenamiento. Si usamos una codificación de longitud variable necesitamos un sistema sencillo de codificación y decodificación $\Longrightarrow$ códigos prefijos (*prefix codes*), en los que ningún código es prefijo de otro código



Example.

$$\underbrace{AAACAGTTGCAT\cdots GGTCCCTAGG}_{130{,}000{,}000}$$



- Codificación de longitud fija: $A = 00$, $C = 01$, $G = 10$ and $T = 11$. Se necesitan 260 millones de bits ($\approx$ 260 MB)
- Codificación de longitud variable: supongamos que A aparece $70 \cdot 10^6$ veces, C aparece $3 \cdot 10^6$ veces, G aparece $20 \cdot 10^6$ y T aparece $37 \cdot 10^7$ veces, es mejor asignar un código corto a A y códigos más largos a C y G, p.e., $A = 0$, $T = 10$, $G = 110$, $C = 111$.

Códigos prefijos

Definición

Dado un alfabeto (conjunto de símbolos) Σ, un código prefijo ϕ asigna una cadena de bits a cada símbolo $x \in \Sigma$ y se cumple que, para cualesquiera símbolos distintos $x, y \in \Sigma$, $\phi(x)$ no es un prefijo de $\phi(y)$.

Ejemplo

- Si $\phi(A) = 1$ y $\phi(C) = 101 \implies \phi$ no es un código prefijo!
- Si $\phi(A) = 1$, $\phi(T) = 01$, $\phi(G) = 000$ y $\phi(C) = 001$ entonces ϕ es un código prefijo
- Los códigos prefijos son fáciles de decodificar, sin necesidad de *backtracking*, de forma unívoca de izquierda a derecha:

$$\underbrace{000}_{G}\underbrace{1}_{A}\underbrace{01}_{T}\underbrace{1}_{A}\underbrace{001}_{C}\underbrace{1}_{A}\underbrace{01}_{T}\underbrace{000}_{G}\underbrace{001}_{C}\underbrace{01}_{T}$$

Códigos prefijos

Definición

Dado un alfabeto (conjunto de símbolos) Σ, un código prefijo ϕ asigna una cadena de bits a cada símbolo $x \in \Sigma$ y se cumple que, para cualesquiera símbolos distintos $x, y \in \Sigma$, $\phi(x)$ no es un prefijo de $\phi(y)$.

Ejemplo

- Si $\phi(A) = 1$ y $\phi(C) = 101 \implies \phi$ no es un código prefijo!
- Si $\phi(A) = 1$, $\phi(T) = 01$, $\phi(G) = 000$ y $\phi(C) = 001$ entonces ϕ es un código prefijo
- Los códigos prefijos son fáciles de decodificar, sin necesidad de *backtracking*, de forma unívoca de izquierda a derecha:

Códigos prefijos

Definición

Dado un alfabeto (conjunto de símbolos) Σ, un código prefijo ϕ asigna una cadena de bits a cada símbolo $x \in \Sigma$ y se cumple que, para cualesquiera símbolos distintos $x, y \in \Sigma$, $\phi(x)$ no es un prefijo de $\phi(y)$.

Ejemplo

- Si $\phi(A) = 1$ y $\phi(C) = 101 \implies \phi$ no es un código prefijo!
- Si $\phi(A) = 1$, $\phi(T) = 01$, $\phi(G) = 000$ y $\phi(C) = 001$ entonces ϕ es un código prefijo
- Los códigos prefijos son fáciles de decodificar, sin necesidad de *backtracking*, de forma unívoca de izquierda a derecha:

$$\underbrace{000}_{G}\ \underbrace{1}_{A}\ \underbrace{01}_{T}\ \underbrace{1}_{A}\ \underbrace{001}_{C}\ \underbrace{1}_{A}\ \underbrace{01}_{T}\ \underbrace{000}_{G}\ \underbrace{001}_{C}\ \underbrace{01}_{T}$$

Árbol de prefijos

Un código prefijo puede representarse mediante un árbol binario etiquetado (de hecho, un *trie*).

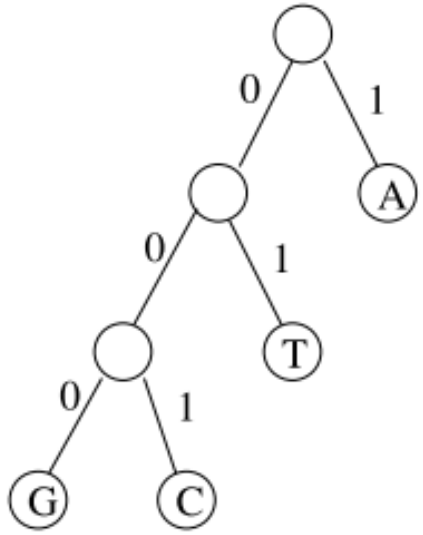
Definición

Un árbol de prefijos T (*prefix tree*) es un árbol binario que cumple las siguientes propiedades:

- Cada símbolo de Σ distinto se asocia a una hoja (nodo sin descendientes)
- La arista que une a un nodo con la raíz de su subárbol izquierdo se etiqueta 0, la arista al subárbol derecho se etiqueta 1
- Las etiquetas en el camino de la raíz a una hoja especifican el código asignado al símbolo asociado la hoja.

Árbol de prefijos

Σ	ϕ
A	1
T	01
G	000
C	001



Longitud de la codificación

- Dado un texto S de longitud $|S| = n$ escrito con símbolos de Σ y un código prefijo ϕ, denotaremos $B(S)$ a la longitud del texto codificado
- Sea $f(x, S)$ la frecuencia relativa de $x \in \Sigma$ en el texto S (Nota: $\sum_{x \in \Sigma} f(x, S) = 1$). Si queda claro por el contexto cúal es el texto S, escribiremos $f(x)$ en vez de $f(x, S)$.
- Entonces $f(x) \cdot n$ es la frecuencia absoluta de x en S y

$$B(S) = \sum_{x \in \Sigma} n \cdot f(x) \cdot |\phi(x)| = n \cdot \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$$

- $\alpha(S) = \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$ es la longitud media del código ϕ o número promedio de bits por símbolo, y también se le conoce como factor de compresión.

Longitud de la codificación

- Dado un texto S de longitud $|S| = n$ escrito con símbolos de Σ y un código prefijo ϕ, denotaremos $B(S)$ a la longitud del texto codificado
- Sea $f(x, S)$ la frecuencia relativa de $x \in \Sigma$ en el texto S (Nota: $\sum_{x \in \Sigma} f(x, S) = 1$). Si queda claro por el contexto cúal es el texto S, escribiremos $f(x)$ en vez de $f(x, S)$.
- Entonces $f(x) \cdot n$ es la frecuencia absoluta de x en S y

$$B(S) = \sum_{x \in \Sigma} n \cdot f(x) \cdot |\phi(x)| = n \cdot \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$$

- $\alpha(S) = \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$ es la longitud media del código ϕ o número promedio de bits por símbolo, y también se le conoce como factor de compresión.

Longitud de la codificación

- Dado un texto S de longitud $|S| = n$ escrito con símbolos de Σ y un código prefijo ϕ, denotaremos $B(S)$ a la longitud del texto codificado
- Sea $f(x, S)$ la frecuencia relativa de $x \in \Sigma$ en el texto S (Nota: $\sum_{x \in \Sigma} f(x, S) = 1$). Si queda claro por el contexto cúal es el texto S, escribiremos $f(x)$ en vez de $f(x, S)$.
- Entonces $f(x) \cdot n$ es la frecuencia absoluta de x en S y

$$B(S) = \sum_{x \in \Sigma} n \cdot f(x) \cdot |\phi(x)| = n \cdot \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$$

- $\alpha(S) = \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$ es la longitud media del código ϕ o número promedio de bits por símbolo, y también se le conoce como factor de compresión.

Longitud de la codificación

- Dado un texto S de longitud $|S| = n$ escrito con símbolos de Σ y un código prefijo ϕ, denotaremos $B(S)$ a la longitud del texto codificado
- Sea $f(x, S)$ la frecuencia relativa de $x \in \Sigma$ en el texto S (Nota: $\sum_{x \in \Sigma} f(x, S) = 1$). Si queda claro por el contexto cúal es el texto S, escribiremos $f(x)$ en vez de $f(x, S)$.
- Entonces $f(x) \cdot n$ es la frecuencia absoluta de x en S y

$$B(S) = \sum_{x \in \Sigma} n \cdot f(x) \cdot |\phi(x)| = n \cdot \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$$

- $\alpha(S) = \sum_{x \in \Sigma} f(x) \cdot |\phi(x)|$ es la longitud media del código ϕ o número promedio de bits por símbolo, y también se le conoce como factor de compresión.

Longitud de la codificación

- En términos del árbol de prefijos T de ϕ, la longitud $|\phi(x)|$ del código de x es la profundidad $d_T(x)$ de la hoja etiquetada con x en T
- Por lo tanto

$$\alpha(T) = \sum_{x \in \Sigma} f(x) \cdot d_T(x)$$

  dicho de otro modo, $\alpha(T)$ es la profundidad promedio de las hojas no vacías (la "probabilidad" de una hoja no vacía es la frecuencia del símbolo asociado a la hoja

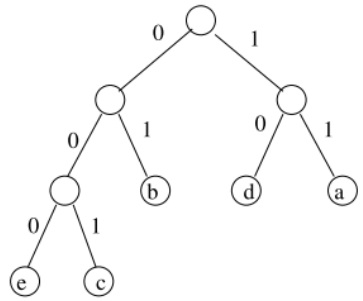
Códigos de longitud variable vs. de longitud fija

Ejemplo

- Sea $\Sigma = \{a, b, c, d, e\}$ y S un texto con alfabeto Σ con las siguientes frecuencias:

$$f(a) = 0{,}32,\ f(b) = 0{,}25,\ f(c) = 0{,}20,\ f(d) = 0{,}18,\ f(e) = 0{,}05$$

- Un código ϕ_0 de longitud fija necesita $\lceil \lg 5 \rceil = 3$ bits por código, de manera que $\alpha(\phi_0) = 3$
- Consideremos el código prefijo ϕ_1:



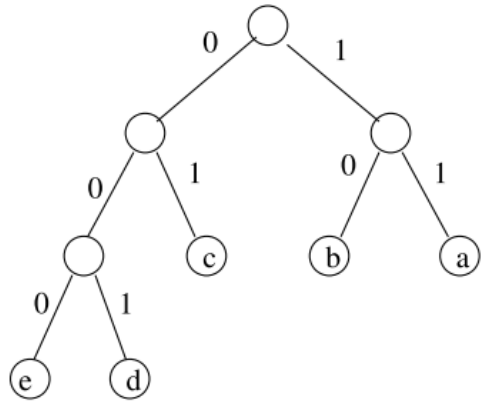
$$\alpha(\phi_1) = 0{,}32 \cdot 2 + 0{,}25 \cdot 2 + 0{,}20 \cdot 3 + 0{,}18 \cdot 2 + 0{,}05 \cdot 3 = 2{,}25$$

- En promedio, ϕ_1 necesita 2.25 bits/símbolo frente a los 3 bits/símbolo del código de longitud fija ϕ_0. Ie

Códigos de longitud variable vs. de longitud fija

¿Es $\alpha = 2{,}25$ el menor factor de compresión posible? ¿Cúal es la máxima compresión alcanzable?

Consideremos el código prefijo ϕ_2:



$$\alpha(\phi_2) = 0{,}32 \cdot 2 + 0{,}25 \cdot 2 + 0{,}20 \cdot 2 + 0{,}18 \cdot 3 + 0{,}05 \cdot 3 = 2{,}23$$

¡$\alpha(\phi_2)$ es mejor que $\alpha(\phi_1)$!
¿Es $\alpha(\phi_2)$ la mínima posible? ($\Longrightarrow$ mejor compresión en promedio)
Hay que considerar todos los códigos prefijos posibles para el alfabeto Σ.

Códigos prefijos óptimos

Dado un texto, un código prefijo óptimo es un código prefijo ϕ con $\alpha(\phi)$ mínima.

Intuitivamente, en el árbol de prefijos de un código prefijo óptimo, los símbolos muy frecuentes están a poca profundidad, a poca distancia de la raíz (códigos cortos), mientras que los símbolos de muy baja frecuencia habrán de estar a mucha mayor profundidad.

Árboles de prefijos óptimos

Un árbol de prefijos T se dice completo si todas sus hojas tienen símbolos asociados.

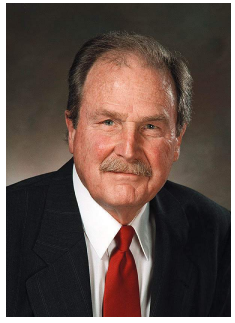
- Si $m = |\Sigma|$ un árbol completo contiene exactamente m hojas y $m - 1$ nodos internos ($2m - 1$ nodos en total). Cada una de las hojas tiene asociado uno de los m símbolos.
- Si un árbol de prefijos no es completo entonces tiene $> 2m - 1$ nodos y algunas de sus hojas **no** tienen asociado ningún símbolo de Σ.

Proposición

El árbol de prefijos que representa a un código prefijo óptimo es completo.

Códigos de Huffman

David Huffman (1925–99) propuso en 1952 un algoritmo voraz eficiente para construir códigos prefijos óptimos.



El algoritmo de Huffman genera un árbol de prefijos completo, con las hojas tan cerca de la raíz como resulte posible, con los símbolos de alta frecuencia a poca profundidad y los símbolos de baja frecuencia más alejados.

Códigos de Huffman

Se nos dan las frecuencias relativas $f(x)$ para todos los símbolos $x \in \Sigma$.

- El algoritmo mantiene una cola de prioridad Q cuyos elementos son nodos de un árbol prefijo y cuyas prioridades son la suma de las $f(x)$ en las hojas del subárbol enraizado en el nodo
- Se construye el árbol de prefijos (conocido como árbol de Huffman) de abajo a arriba como sigue:
  - Se insertan m hojas, cada una con un símbolo $x \in \Sigma$ con prioridad $f(x)$
  - Se extraen los dos elementos de prioridad mínima en Q y se crea un nodo cuyos hijos son los dos elementos extraídos y cuya prioridad es la suma de las prioridades. El elemento se inserta en Q.
  - Repetir el paso anterior hasta que solo queda un nodo (raíz del árbol de prefijos) en Q

El árbol de prefijos construído corresponde a un código prefijo óptimo (denominado código de Huffman)

Códigos de Huffman

```
procedure HUFFMAN(Σ, f)
    Q := ∅;
    for x ∈ Σ do
        Crear una hoja ℓ(x) con símbolo x y prioridad f(x)
        Q.INSERT(ℓ(x), f(x));
    end for
    while Q.SIZE() > 1 do
        x := Q.EXTRACT-MIN()
        y := Q.EXTRACT-MIN()
        Crear un nuevo nodo z
        z.left := x; z.right := y;
        f(z) := f(x) + f(y)
        Q.INSERT(z, f(z))
    end while
    return Q.EXTRACT-MIN()
end procedure
```

Si Q es un heap el algoritmo tiene coste $\mathcal{O}(n \lg n)$.

Códigos de Huffman

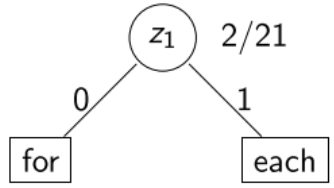
Ejemplo

Consideremos el texto: *for each rose, a rose is a rose, the rose* y el alfabeto $\Sigma = \{$for,each,rose,a,is,the,',',␣$\}$

Frecuencias: $f(\text{for}) = 1/21,\ f(\text{rose}) = 4/21,\ f(\text{is}) = 1/21,\ f(\text{a}) = 2/21,\ f(\text{each}) = 1/21,\ f(\text{','}) = 2/21,\ f(\text{the}) = 1/21,\ f(␣) = 9/21$

Cola de prioridad:

$Q_0 = [(\text{for} : 1/21), (\text{each} : 1/21), (\text{is} : 1/21), (\text{a} : 2/21), (\text{','} : 2/21), (\text{the} : 2/21), (\text{rose} : 4/21), (␣ : 9/21)]$

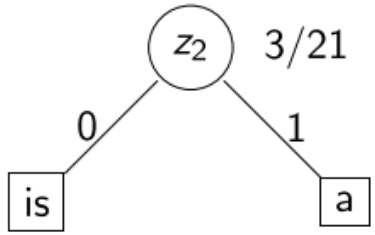


$Q_1 = [(\text{is} : 1/21), (\text{a} : 2/21), (\text{','} : 2/21), (\text{the} : 2/21), (z_1 : 2/21), (\text{rose} : 4/21), (␣ : 9/21)]$

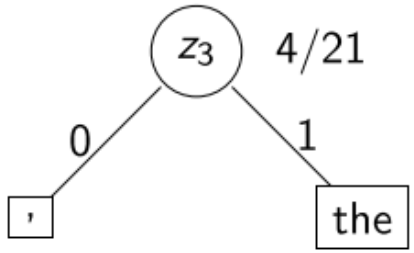
Códigos de Huffman

Ejemplo

$Q_1 = [(\text{is} : 1/21), (\text{a} : 2/21), (\text{','} : 2/21), (\text{the} : 2/21), (z_1 : 2/21), (\text{rose} : 4/21), (\text{b} : 9/21)]$



$Q_2 = [(\text{','} : 2/21), (\text{the} : 2/21), (z_1 : 3/21), (z_2 : 3/21), (\text{rose} : 4/21), (\text{b} : 9/21)]$

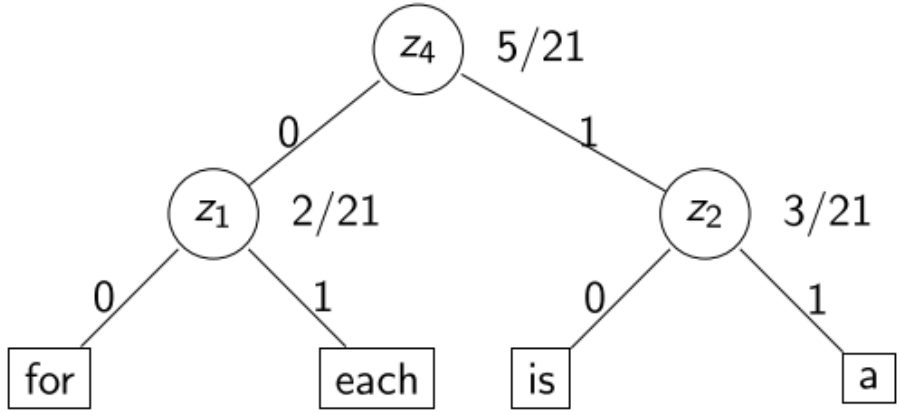


$Q_3 = [(z_1 : 2/21), (z_2 : 3/21), (\text{rose} : 4/21), (z_3 : 4/21), (\text{b} : 9/21)]$

Códigos de Huffman

Ejemplo

$Q_3 = [(z_1 : 2/21), (z_2 : 3/21), (\text{rose} : 4/21), (z_3 : 4/21), (\flat : 9/21)]$

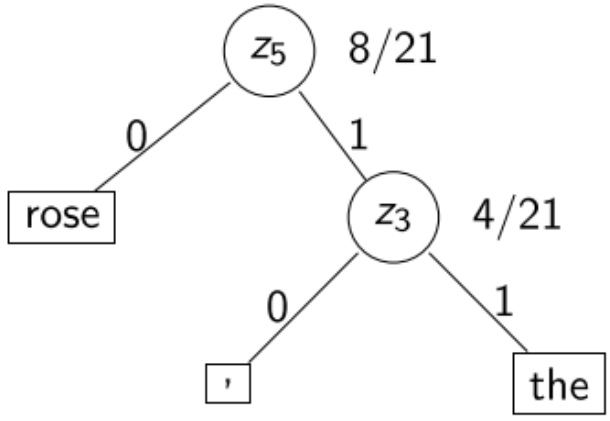


$Q_4 = [(\text{rose} : 4/21), (z_3 : 4/21), (z_4 : 5/21), (\flat : 9/21)]$

Códigos de Huffman

Ejemplo

$Q_4 = [(\text{rose} : 4/21), (z_3 : 4/21), (z_4 : 5/21), (\flat : 9/21)]$

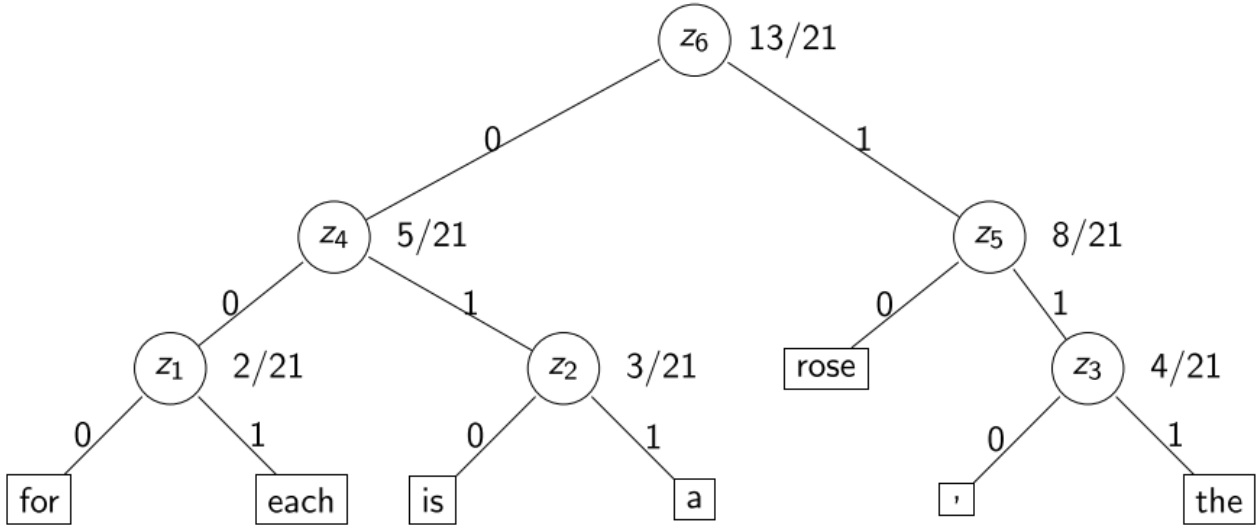


$Q_5 = [(z_4 : 5/21), (z_5 : 8/21), (\flat : 9/21)]$

Códigos de Huffman

Ejemplo

$Q_5 = [(z_4 : 5/21), (z_5 : 8/21), (\flat : 9/21)]$

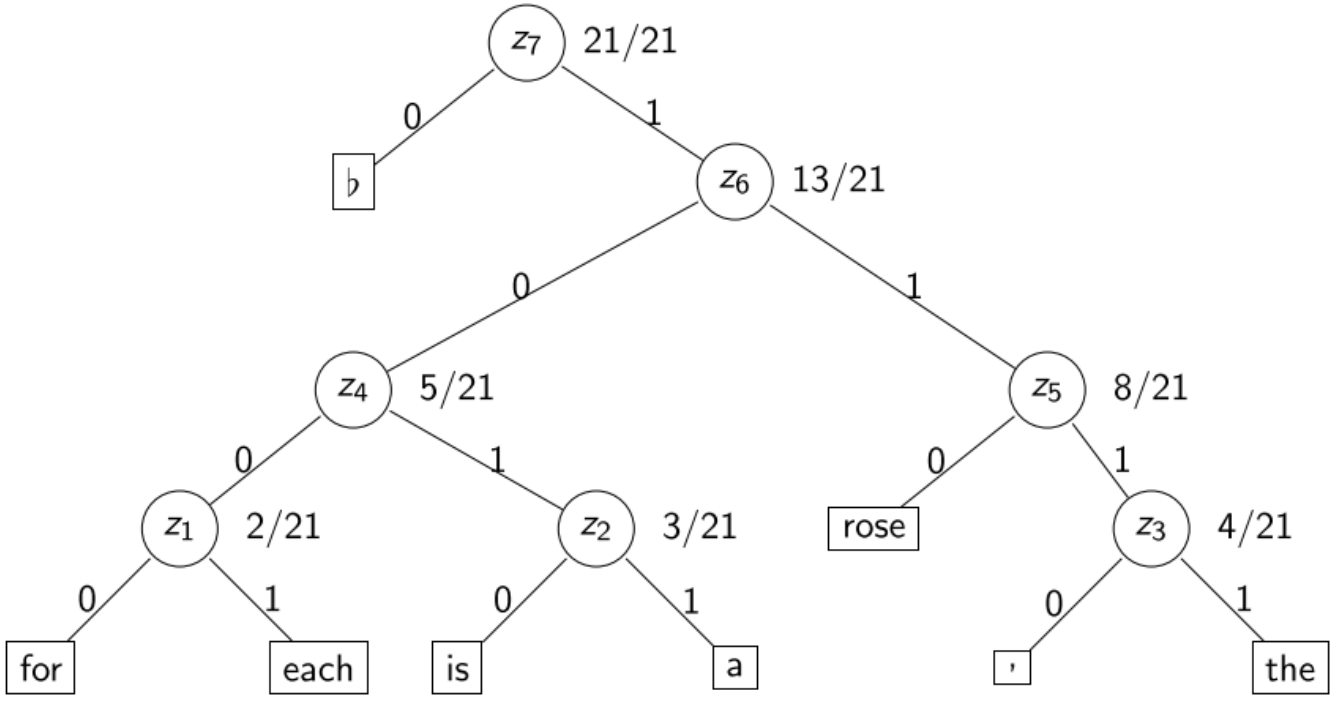


$Q_6 = [(\flat : 9/21), (z_6 : 13/21)]$

Códigos de Huffman

Ejemplo

$Q_6 = [(\flat : 9/21), (z_6 : 13/21)]$



$Q_7 = [(z_7 : 21/21)]$

Códigos de Huffman

Ejemplo

- El texto *for each rose, a rose is a rose, the rose* se codifica como

  1000010010110111001011011001010010110110111001111 0110

- Existen otros códigos prefijos óptimos (p.e., asignar $z.left := x$ y $z.right := y$ es arbitrario! y hay decisiones arbitrarias a tomar cuando dos nodos x e y tienen igual frecuencia $f(x) = f(y)$)
- La longitud del texto codificado es 53, el factor de compresión es $53/21 = 2{,}523\ldots$.
- Con un código de longitud fija se necesitan 4 bits por símbolo y 84 bits para codificar el texto (en vez de 53).

Códigos de Huffman: Corrección del algoritmo

Teorema (Propiedad de la decisión voraz)

Sea Σ un alfabeto finito y $x, y \in \Sigma$ los dos símbolos con las menores frecuencias. Existe un código prefijo óptimo ϕ tal que $|\phi(x)| = |\phi(y)|$ y $\phi(x)$ y $\phi(y)$ solo difieren en su último bit.

Demostración

Sea T el árbol de prefijo de ϕ, y supongamos que x e y no están a la misma profundidad ($|\phi(x)| \neq |\phi(y)|$). Puesto que T es completo habrá dos hojas con símbolos a y b que están a máxima profundidad y son descendientes de un mismo nodo interno (son hermanas). Supondremos además, sin pérdida de generalidad, que $f(a) \leq f(b)$ y $f(x) \leq f(y)$.

Códigos de Huffman: Corrección del algoritmo

Demostración (continúa)

Construímos un nuevo árbol T' intercambiando $a \leftrightarrow x$ y $b \leftrightarrow y$. Puesto que $f(x) \leq f(a)$ y $d_T(a) = d_{T'}(x) \leq d_T(x)$, y que $f(y) \leq f(b)$ y $d_T(b) = d_{T'}(y) \leq d_T(y)$ deducimos que $B(T') \leq B(T)$, por tanto T' es óptimo. Y en T' x e y tienen códigos de igual longitud y que solo difieren en el último bit. $\square$

Códigos de Huffman: Corrección del algoritmo

Teorema (Estructura subóptima)

Sea T' un árbol de prefijos para un código óptimo correspondiente al alfabeto $\Sigma' = \Sigma \setminus \{x, y\} \cup \{z\}$ donde x e y son los símbolos con mínimas frecuencias en Σ y $z \notin \Sigma$ es un nuevo símbolo con frecuencia $f(x) + f(y)$. Si reemplazamos la hoja z en T' por un subárbol con un nodo $\langle z, f(z) \rangle$ que tiene x e y como hijos izquierdo y derecho, respectivamente, obtenemos un nuevo árbol T que es óptimo para Σ.

Demostración

Sea T_0 un árbol de prefijos cualquiera para el alfabeto Σ. Nuestro objetivo es probar que el árbol T descrito en el enunciado cumple $B(T) \leq B(T_0)$.

Códigos de Huffman: Corrección del algoritmo

Demostración (continúa)

Por la propiedad de la decisión voraz solo necesitamos considerar árboles de prefijos T_0 en los que x e y estén en hojas hermanas y su padre z común tiene frecuencia $f(x) + f(y)$.
Sea T_0' el árbol que se obtiene reemplazando el subárbol enraizado en z por una hoja etiquetada z y con frecuencia $f(z) = f(x) + f(y)$. Entonces T_0' es un árbol de prefijos para el alfabeto $\Sigma' = \Sigma \setminus \{x, y\} \cup \{z\}$ donde $f(z) = f(x) + f(y)$.
Como T' es el árbol de prefijos óptimo para Σ' sabemos que $B(T') \leq B(T_0')$.

Códigos de Huffman: Corrección del algoritmo

Demostración (continúa)

Ahora comparamos $B(T_0)$ y $B(T_0')$ y $B(T)$ con $B(T')$:

$$\begin{aligned} B(T_0) &= B(T_0') + f(x) + f(y) \\ B(T) &= B(T') + f(x) + f(y) \\ &\leq B(T_0') + f(x) + f(y) = B(T_0). \end{aligned}$$

□

Códigos de Huffman

Los códigos de Huffman son óptimos bajo una serie de supuestos:

- Solo consideramos compresión sin pérdida (*lossless compression*), el proceso de codificación y decodificación son unívocos, al decodificar se recupera exactamente el texto original
- El alfabeto Σ debe ser conocido de antemano
- Las frecuencias $f(x)$ deben darse de antemano o recolectadas en un primer recorrido de los datos, previo a la construcción del código y la codificación ($\Longrightarrow$ puede ser infactible o demasiado lento en determinadas aplicaciones!)
- Un buen lugar para saber más y explorar extensiones de la comprensión de datos mediante códigos de Huffman es este artículo en Wikipedia
  https://en.wikipedia.org/wiki/Huffman_coding

Parte III

Algoritmos Voraces

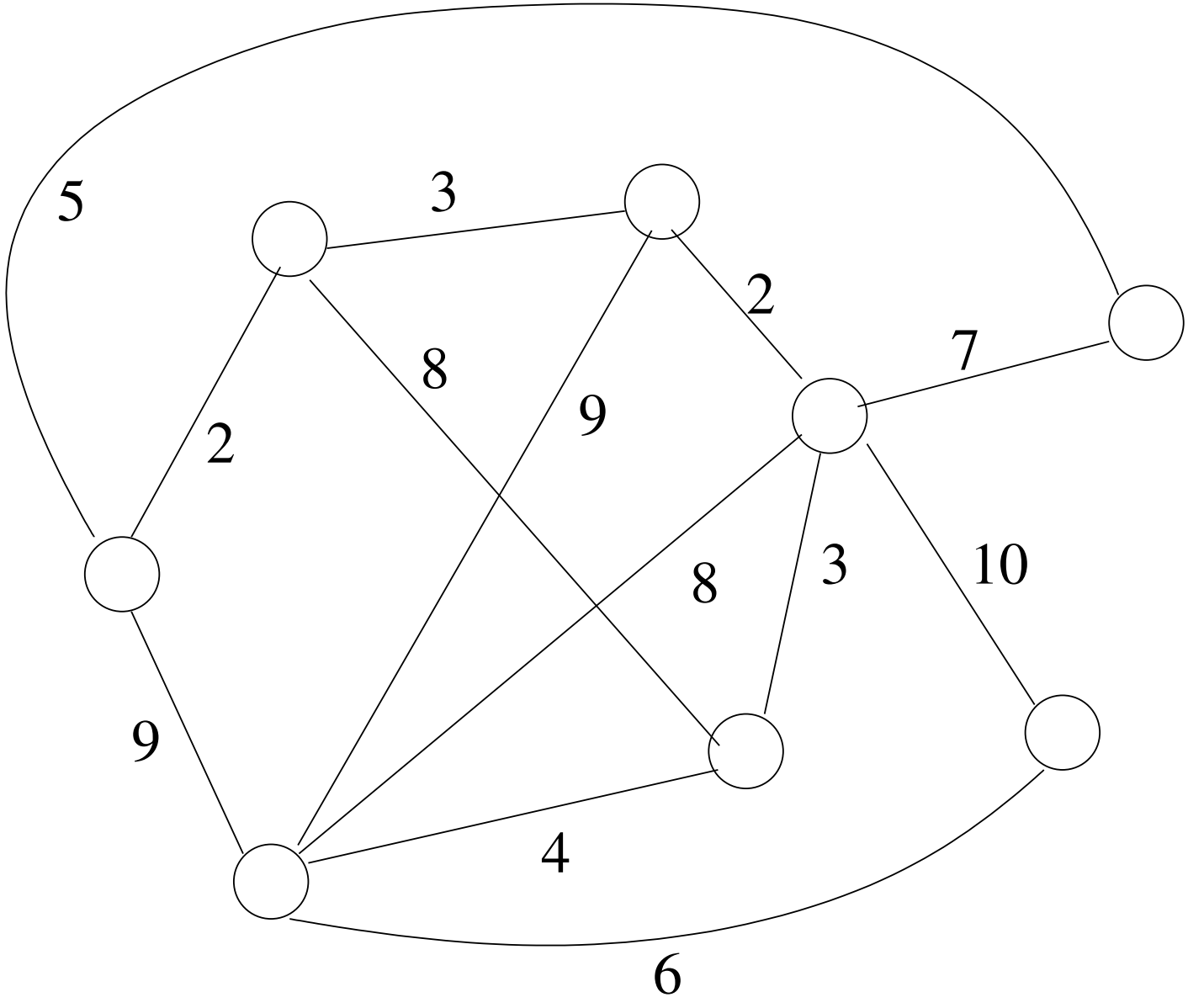
Árboles de Expansión Mínimos

Dado un grafo no dirigido y conexo $G = \langle V, E \rangle$ con pesos o costes en las aristas $\omega : E \to \mathbb{R}$, un árbol de expansión mínimo (*MST: minimum spanning tree*) $T = \langle V, A \rangle$ es un subgrafo de G tal que tiene el mismo conjunto de vértices ($V(T) = V(G)$), es un árbol (es decir, es conexo y acíclico) y su coste

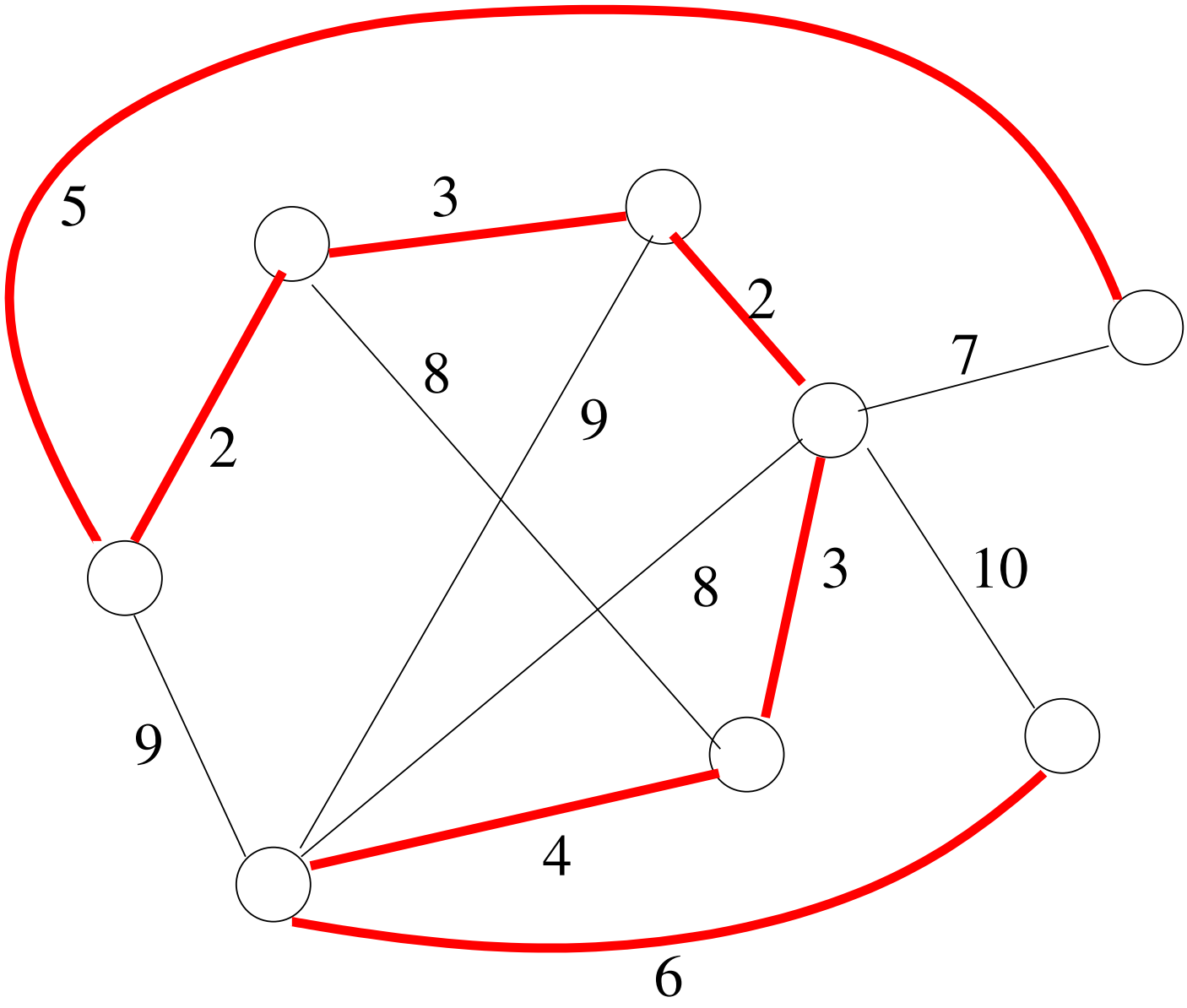
$$\omega(T) = \sum_{e \in A} \omega(e)$$

es mínimo entre todos los posibles árboles de expansión de G.

Árboles de Expansión Mínimos



Árboles de Expansión Mínimos



Árboles de Expansión Mínimos

Existen multitud de algoritmos para calcular un MST de un grafo. No obstante todos ellos siguen un esquema voraz:

$A := \emptyset$; *Candidatas* $:= E$
while $|A| \neq |V(G)| - 1$ **do**
    Seleccionar una arista $e \in$ *Candidatas* que
        no crea un ciclo en T
    $A := A \cup \{e\}$
    Candidatas $:=$ *Candidatas* $- \{e\}$
end while

Árboles de Expansión Mínimos

Definición

Diremos que un conjunto de aristas $A \subset E(G)$ es prometedor si y sólo si:

1. A no contiene ciclos
2. A es un subconjunto de las aristas de un MST del grafo G

Árboles de Expansión Mínimos

Un corte del grafo G es una partición de su conjunto de vértices en dos subconjuntos C y C' no vacíos y disjuntos:

$$V(G) = C \cup C'; \qquad C \cap C' = \emptyset$$

Una arista e respeta un corte $\langle C, C' \rangle$ si sus dos extremos están ambos en C o ambos en C', en caso contrario se dice que e cruza el corte.

Árboles de Expansión Mínimos

Teorema

Sea A un conjunto prometedor de aristas que respetan un cierto corte $\langle C, C' \rangle$ del grafo G. Sea $e \in E(G)$ la arista de mínimo peso que cruza el corte $\langle C, C' \rangle$. Entonces

$$A \cup \{e\}$$

es prometedor

Árboles de Expansión Mínimos

El teorema anterior nos da la "receta" para diseñar algoritmos de cálculo de un MST: una vez que hayamos definido cuál es el corte que corresponde a cada iteración, la selección de la arista consistirá en localizar la arista e de mínimo peso que cruza el corte. Por definición, como A respeta el corte y e lo cruza, e no puede crear un ciclo en A.
Más aún: la corrección de los algoritmos que se fundamentan en esta idea queda automáticamente establecida.

Árboles de Expansión Mínimos

Demostración

Sea A' el conjunto de aristas de un MST T' tal que $A \subset A'$. Puesto que A respeta el corte, al menos una arista de las que cruzan el corte tendría que pertenecer a A', en caso contrario A' no sería conexo. Suponga-mos que una de dichas aristas es e'. Si e' es la arista de mínimo peso que cruza el corte, como $A \cup \{e'\}$ es prometedor, hemos demostrado el teorema. ¿Pero que ocurre si e' no es la arista de mínimo peso que cruza el corte?

Árboles de Expansión Mínimos

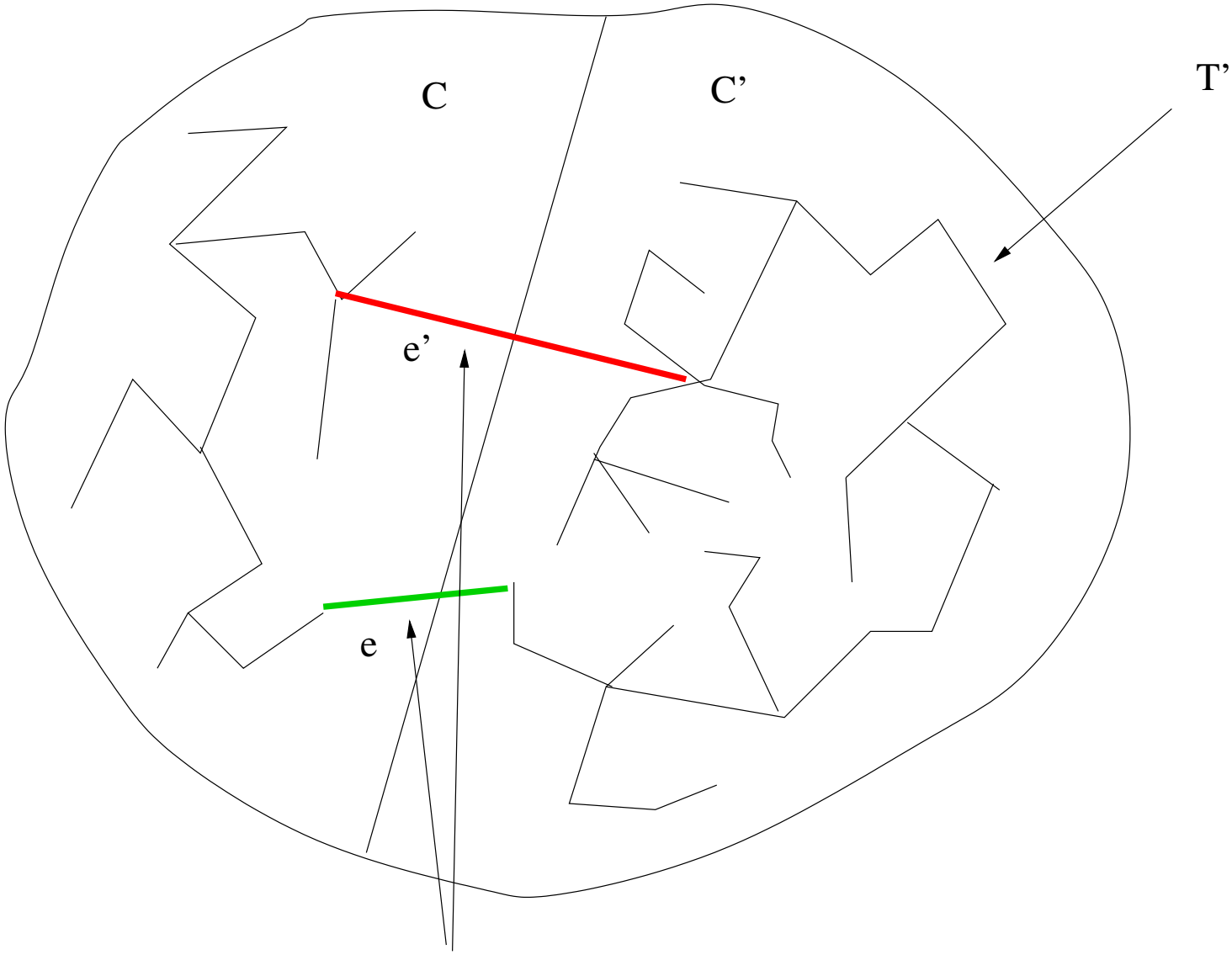
Demostración (continúa)

El coste del MST T' incluirá el coste de las aristas de A, el coste $\omega(e')$ y el coste de otras aristas. Si agregásemos a T' la arista e de mínimo peso que cruza el corte, crearíamos un ciclo, porque T' es un árbol. Sea e' la arista de T' que cruza el corte forma parte de dicho ciclo. De manera que

$$T = T' \cup \{e\} - \{e'\}$$

también es un árbol de expansión. Y su coste es menor o igual que el de T' puesto que sólo cambiamos el coste de e' por el de e, que es el mínimo. Como T' es MST, llegamos a una contradicción a menos que e y e' tengan el mismo coste, y por tanto T y T' tendrían el mismo coste. El teorema queda demostrado, ya que $A \cup \{e\}$ es un subconjunto de las aristas de T, que es un MST. $\square$

Árboles de Expansión Mínimos



we can replace e' in T' by e to obtain a new spanning tree with smaller cost!!

Árboles de Expansión Mínimos

Un formalismo alternativo para razonar sobre la corrección de algoritmos que calculan un MST se basa en las siguientes propiedades.

Proposición

Sea T un MST de un grafo G. Asumamos que todos los pesos en G son distintos. Entonces se cumple:

1. *Propiedad del corte:*

$$e \in T \iff e \text{ es una arista de peso mínimo que cruza algún corte de } G$$

2. *Propiedad del ciclo:*

$$e \notin T \iff e \text{ es una arista de peso máximo en algún ciclo de } G$$

Árboles de Expansión Mínimos

Las dos propiedades anteriores dan origen a las denominadas regla azul y regla roja para hallar MSTs.

Regla azul: Si e es una arista de peso mínimo en un corte de G, dicha arista forma parte de algún MST de G (equivale al teorema que indica que $A \cup \{e\}$ es prometedor)

Regla roja: Si e es una arista de peso máximo en algún ciclo entonces e puede ser descartada

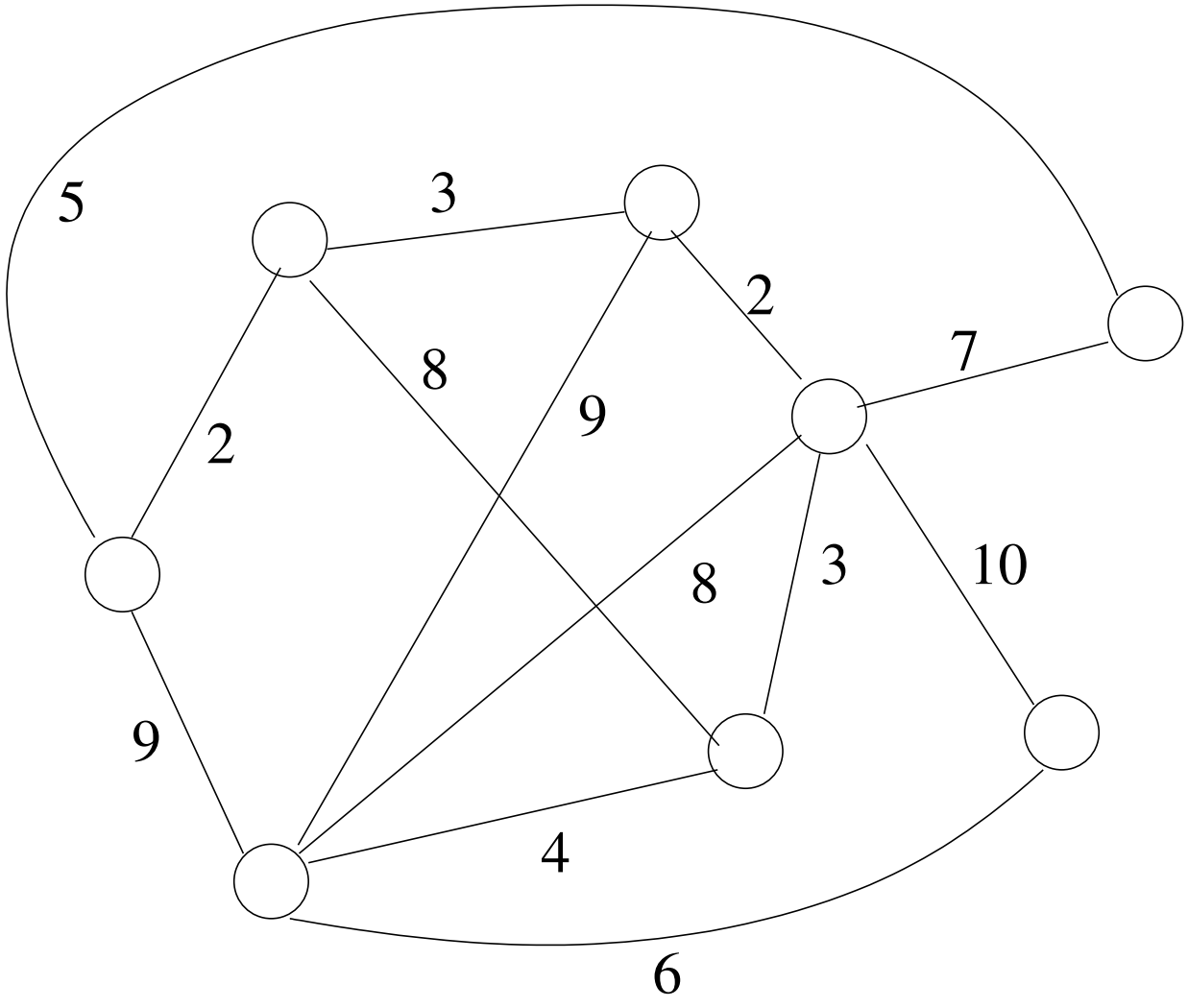
Algoritmo de Jarník-Prim

En el algoritmo de Jarník-Prim, se mantiene un conjunto de vértices *Vistos* $\subset V(G)$, y en cada etapa del algoritmo se selecciona la arista de mínimo peso que une a un vértice u de *Vistos* con un vértice v no visto (i.e., en $V(G) - \textit{Vistos}$).
Dicha arista no puede crear un ciclo y se añade al conjunto A.
Asimismo el vértice v pasa a ser de *Vistos*.
La corrección del algoritmo es inmediata; el conjunto A respeta el corte $\langle \textit{Vistos}, V(G) - \textit{Vistos} \rangle$ y seleccionamos la arista de mínimo peso que cruza el corte para agregarla a A.

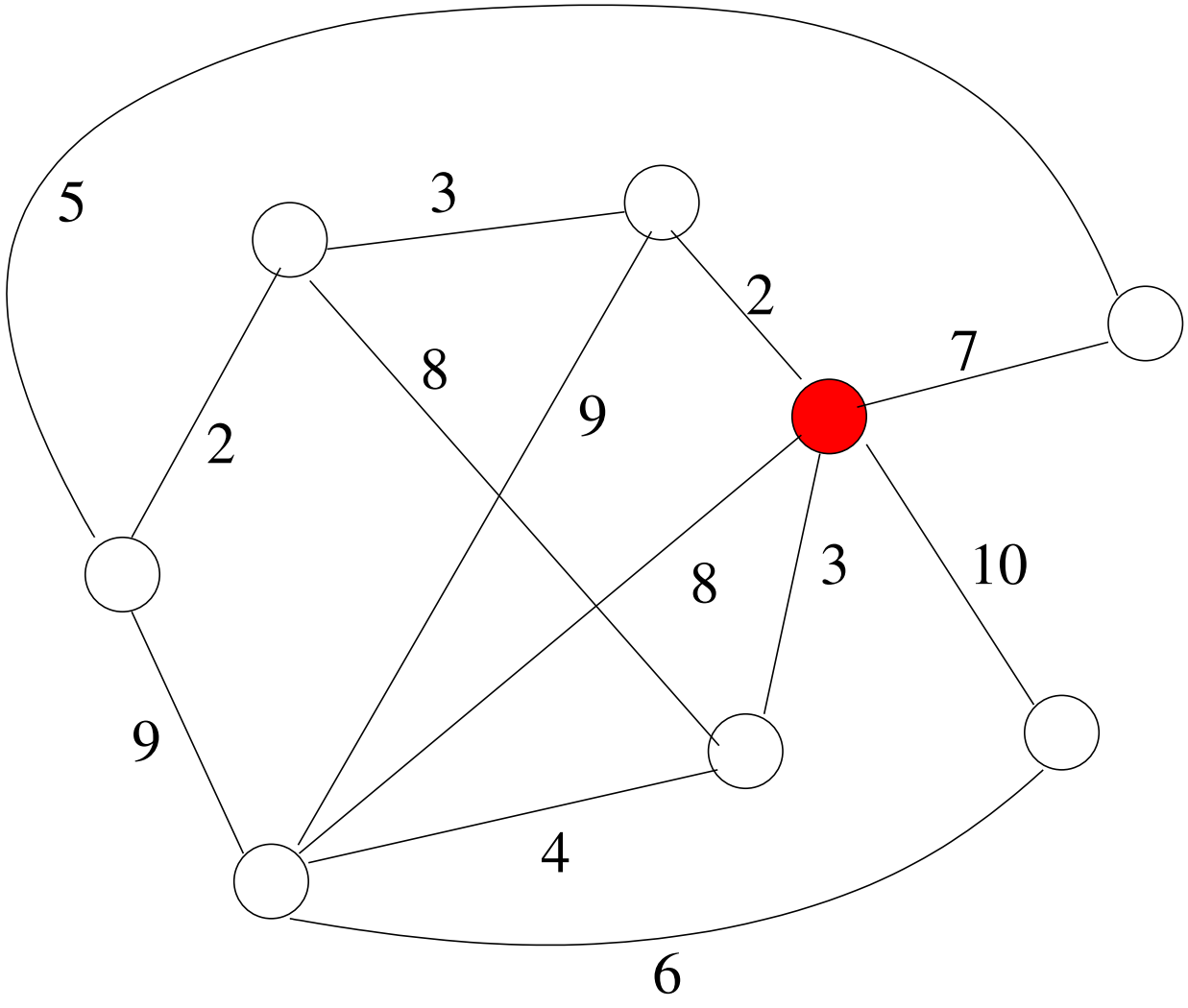
Algoritmo de Jarník-Prim

```
procedure JarnikPrim(G)
    A := ∅; Vistos := {s}
    Candidatas := ∅
    for v ∈ G.Adyacentes(s) do
        Candidatas := Candidatas ∪ {(s, v)}
    end for
    while |A| ≠ |V(G)| − 1 do
        Seleccionar la arista e = (u, v)
           de Candidatas de mínimo peso
        A := A ∪ {e}
        Sea u el vértice en Vistos:
        Vistos := Vistos ∪ {v}
        for w ∈ G.Adyacentes(v) do
            if w ∉ Vistos then
                Candidatas := Candidatas ∪ {(v, w)}
            end if
        end for
    end while
    return A
end procedure
```

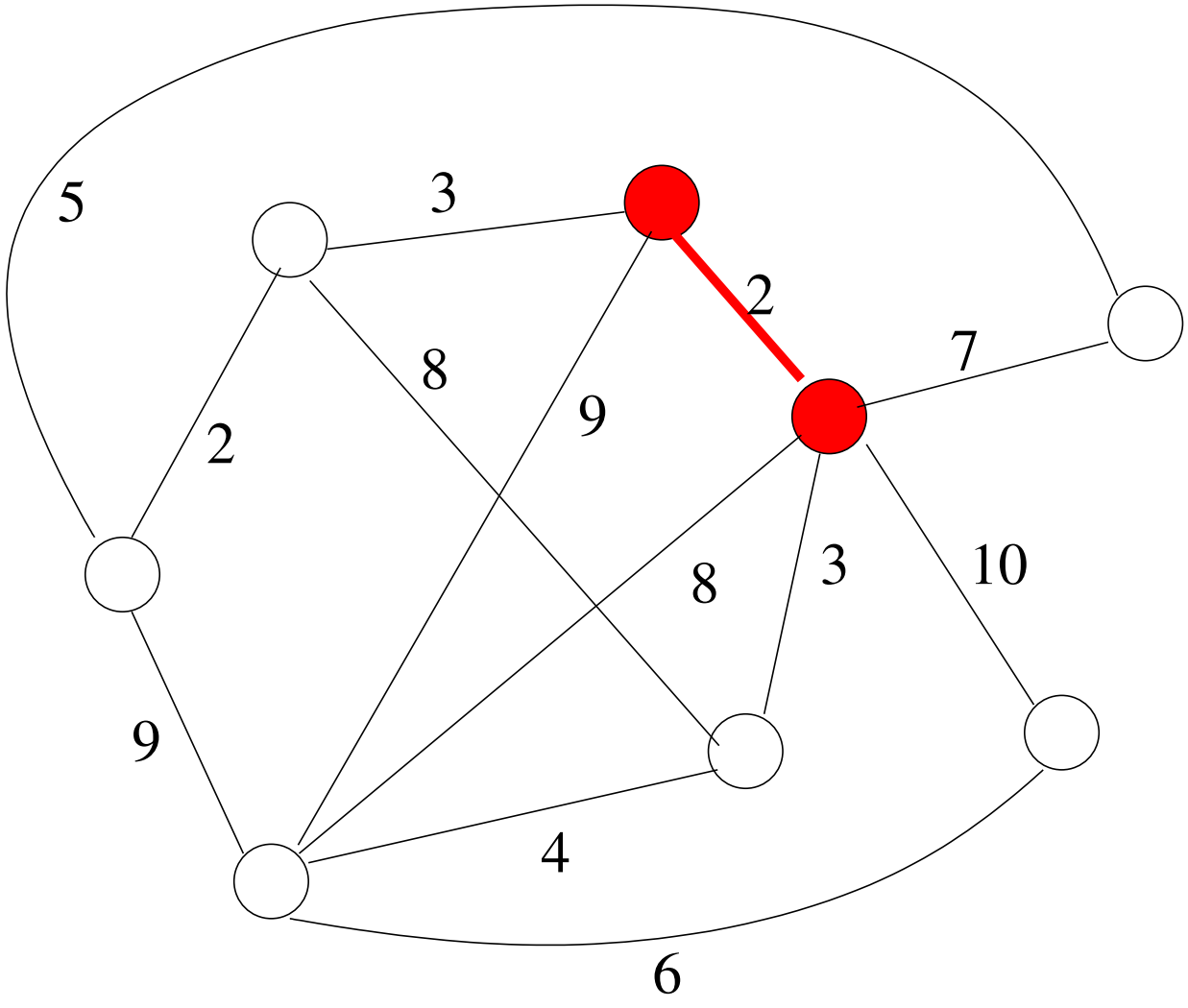
Algoritmo de Jarník-Prim



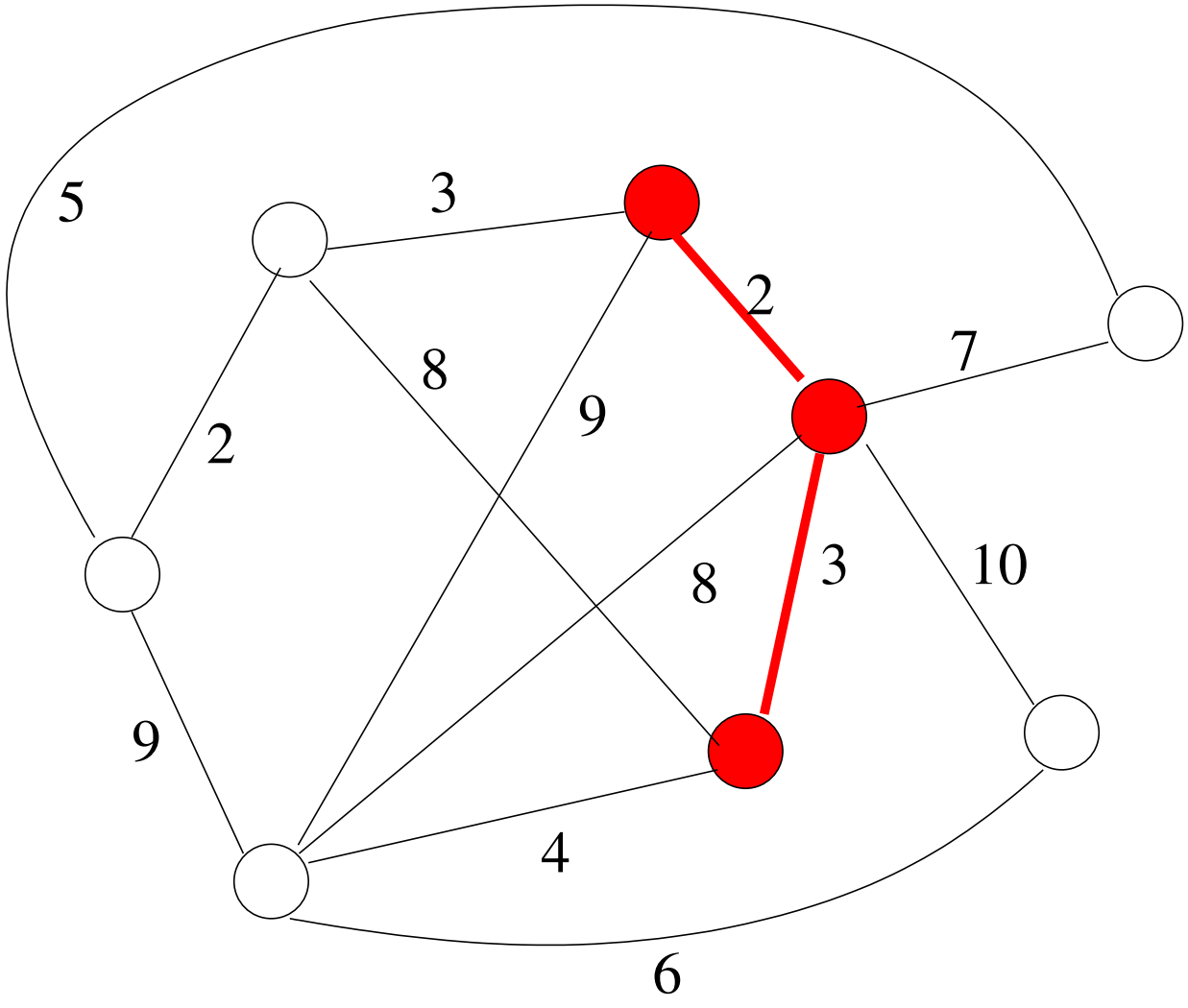
Algoritmo de Jarník-Prim



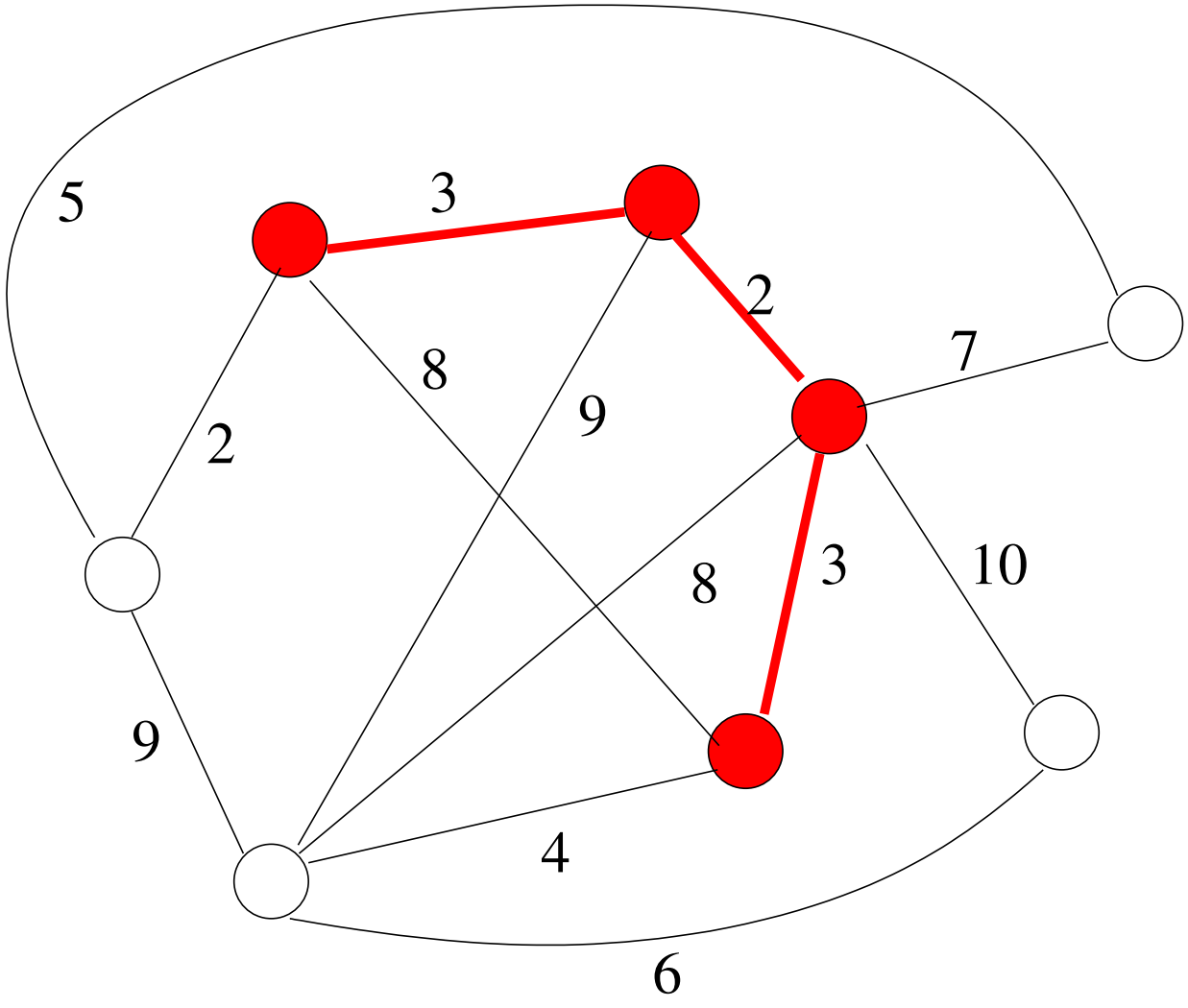
Algoritmo de Jarník-Prim



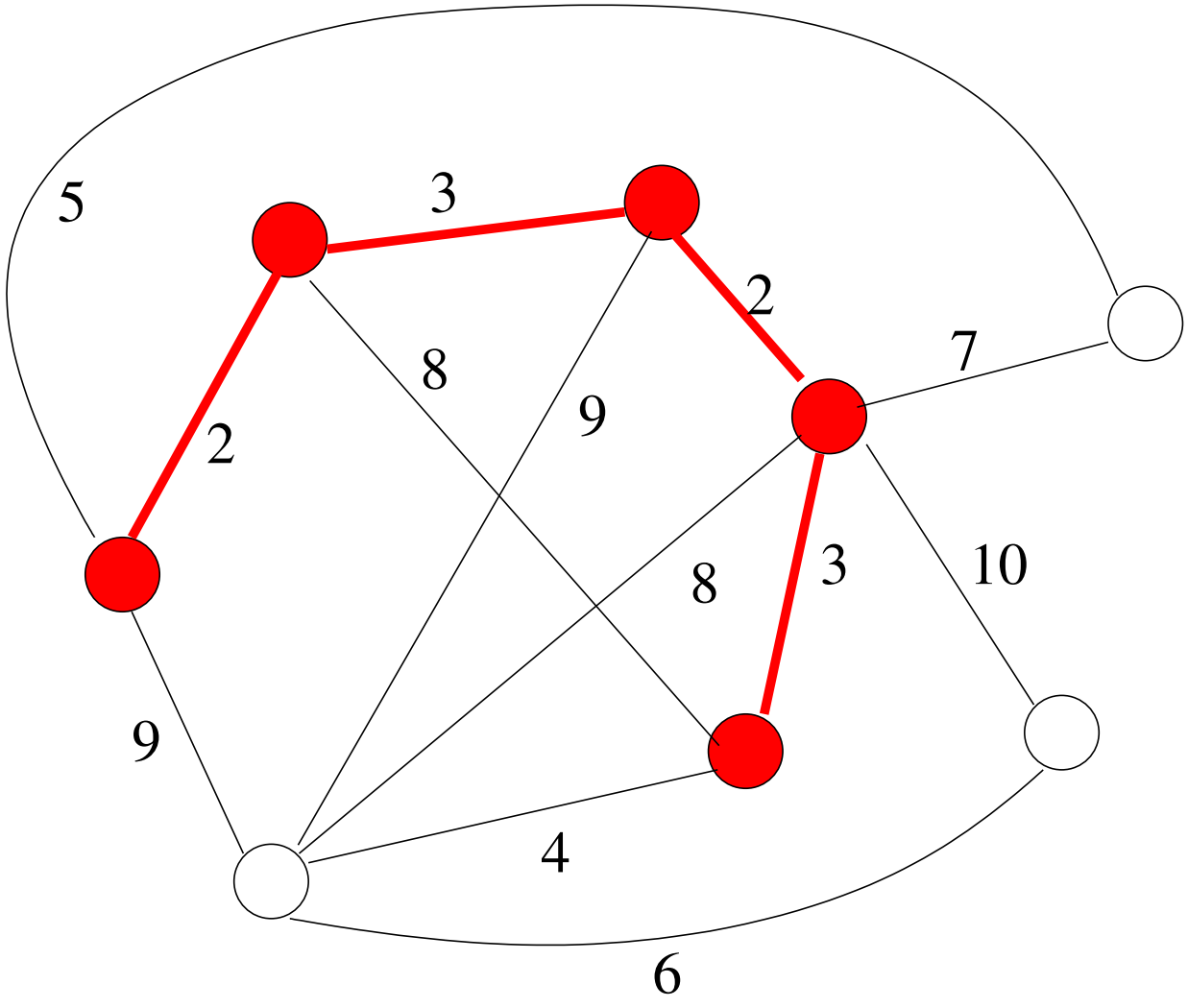
Algoritmo de Jarník-Prim



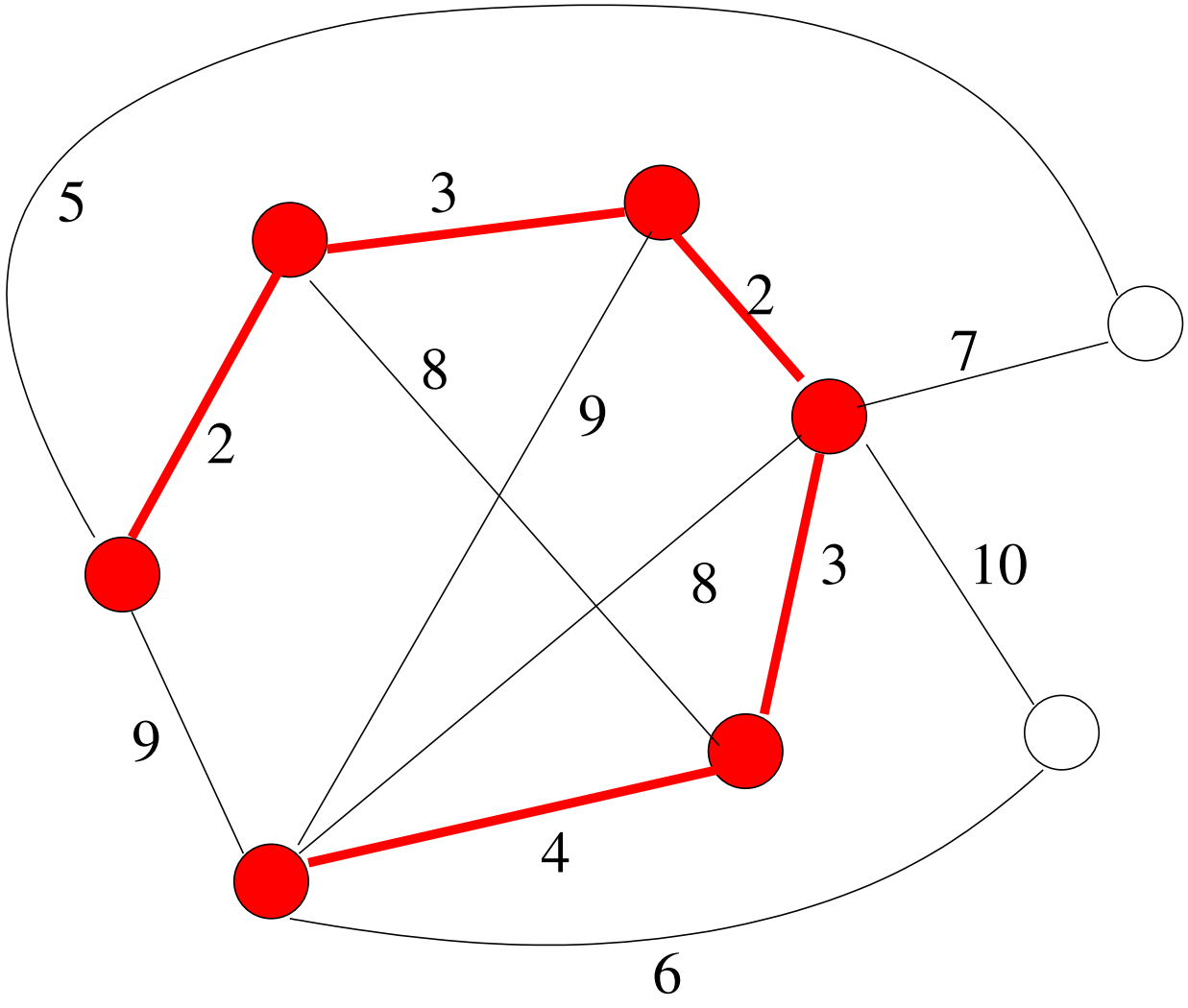
Algoritmo de Jarník-Prim



Algoritmo de Jarník-Prim



Algoritmo de Jarník-Prim



Algoritmo de Jarník-Prim

El coste del algoritmo de Jarník-Prim dependerá de cómo implementemos la selección de la arista candidata. El bucle principal hace $n - 1$ iteraciones, pues en cada iteración añadimos una arista al árbol A, o equivalentemente, en cada iteración *vemos* un vértice nuevo y terminaremos cuando todos estén *vistos*.
En el interior del bucle principal se hacen dos tareas "costosas": seleccionar la arista de mínimo peso en *Candidatas* y añadir ciertas aristas adyacentes al vértice recién visto al conjunto de *Candidatas*.

Algoritmo de Jarník-Prim

El coste de la segunda tarea es proporcional al grado del vértice v y en total aportará al coste

$$\sum_{v \in V(G)} \Theta(\text{grado}(v)) = \Theta(\sum_{v \in V(G)} \text{grado}(v)) = \Theta(m)$$

Pero si usamos una implementación ingenua para el conjunto de *Candidatas*, el coste de seleccionar una arista es $\mathcal{O}(m)$ y el coste total del algoritmo de Jarník-Prim es $\mathcal{O}(n \cdot m)$.

Algoritmo de Jarník-Prim

Para conseguir mejorar sensiblemente el coste del algoritmo el conjunto de *Candidatas* debe implementarse como un min-heap, siendo la prioridad de cada arista su coste.
Entonces la selección (y eliminación) de la arista de coste mínimo en cada iteración pasa a ser $\mathcal{O}(\log m)$.
Por otro lado, el coste de ir añadiendo aristas a *Candidatas* lo tenemos que recalcular:

$$\sum_{v \in V(G)} \Theta\left(\text{grado}(v)(1+\log m)\right) = \Theta\left(\sum_{v \in V(G)} \text{grado}(v)(1+\log m)\right)$$
$$= \Theta(m \log m) = \Theta(m \log n)$$

En total: $\Theta((n+m)\log n)$

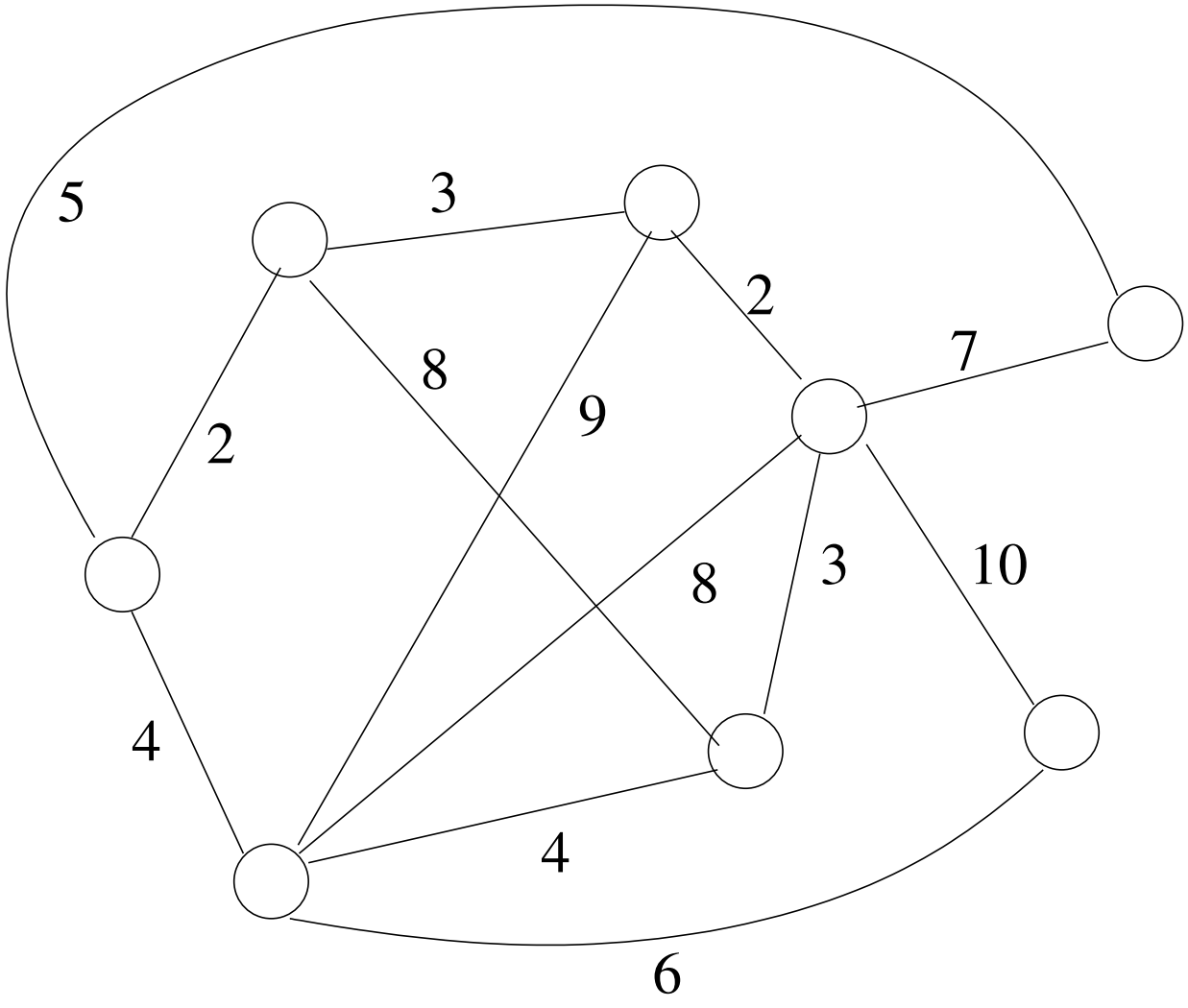
Algoritmo de Kruskal

En el algoritmo de Kruskal se mantiene en todo momento un conjunto A de aristas que es un bosque (un conjunto de árboles). En cada paso se toma una arista e de mínimo peso entre todas las posibles; si dicha arista cierra un ciclo se descarta, si no cierra un ciclo, e se agrega a A.
Cuando la arista $e = (u, v)$ escogida no cierra un ciclo, se define como corte el que forman los vértices en la misma componente conexa de u (mediante aristas de A) por un lado, y los restantes vértices del grafo por otro. Claramente A respeta el corte y e es la arista de mínimo peso que respeta ese corte.

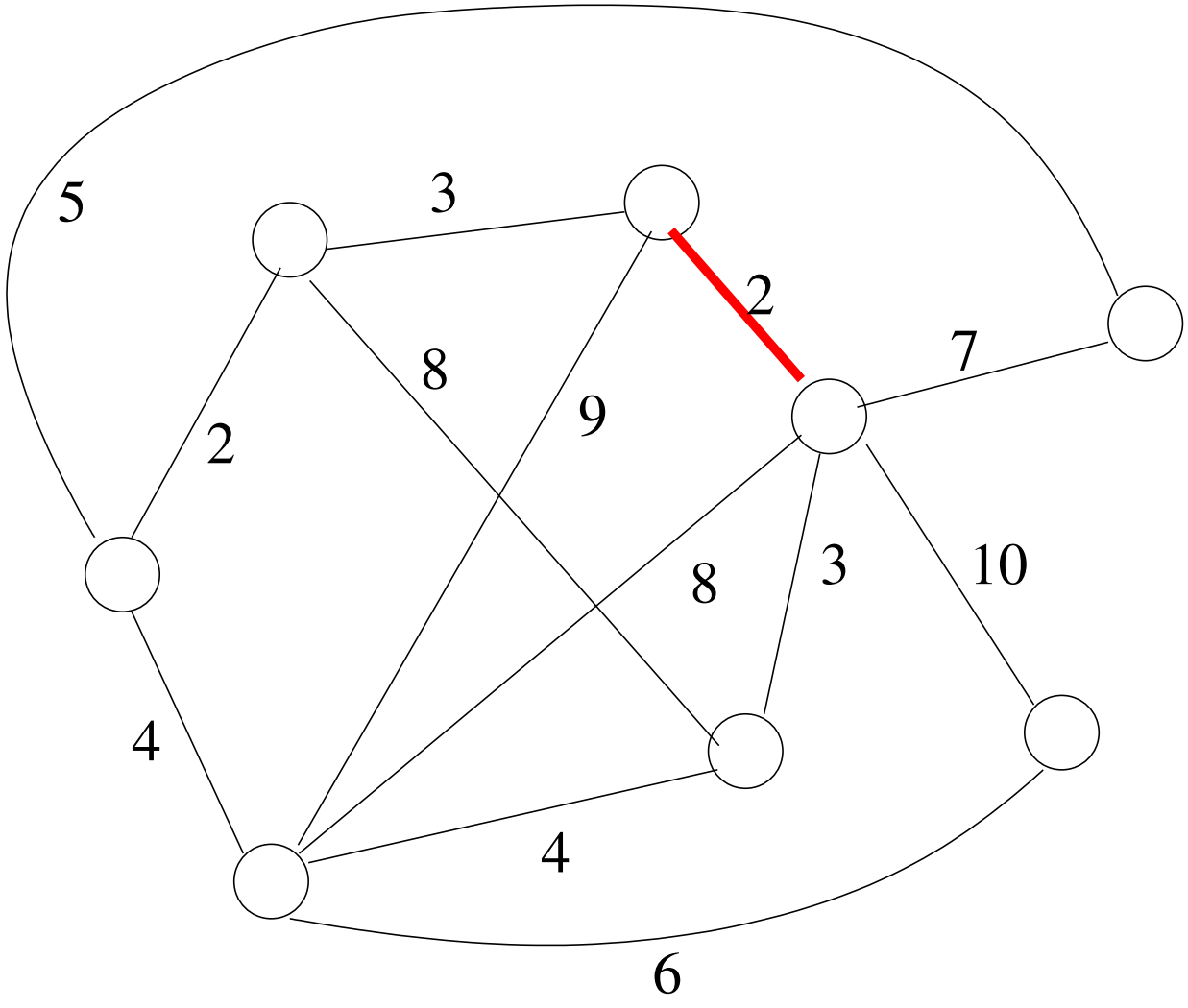
Algoritmo de Kruskal

procedure KRUSKAL($G = \langle V, E \rangle$)
    Ordenar E por peso creciente
    $A := \emptyset$
    while $|A| \neq |V(G)| - 1$ **do**
        $e = (u, v) :=$ SIGUIENTE(E)
        if e no cierra un ciclo en A **then**
            $A := A \cup \{e\}$
        end if
    end while
    return A
end procedure

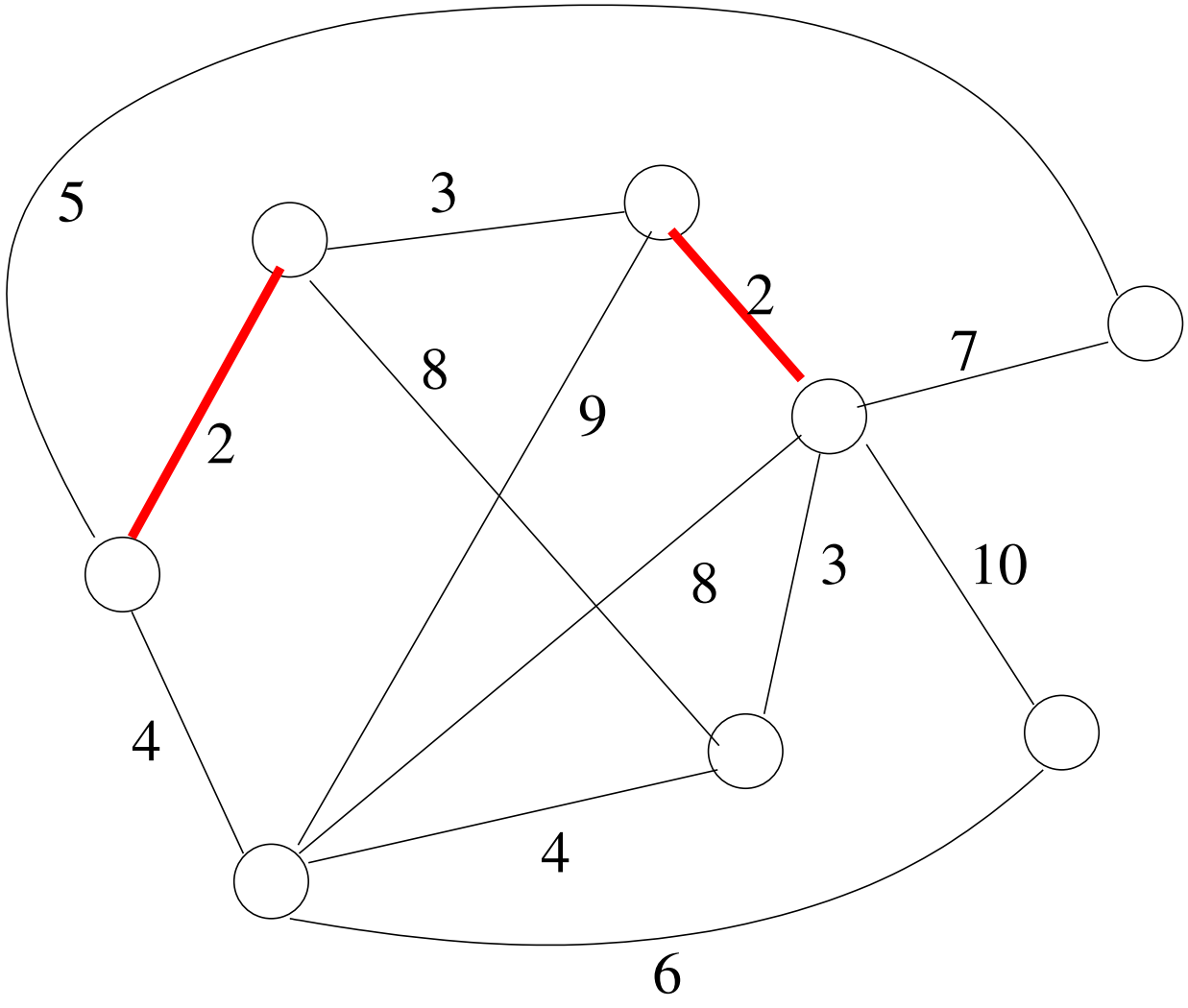
Algoritmo de Kruskal



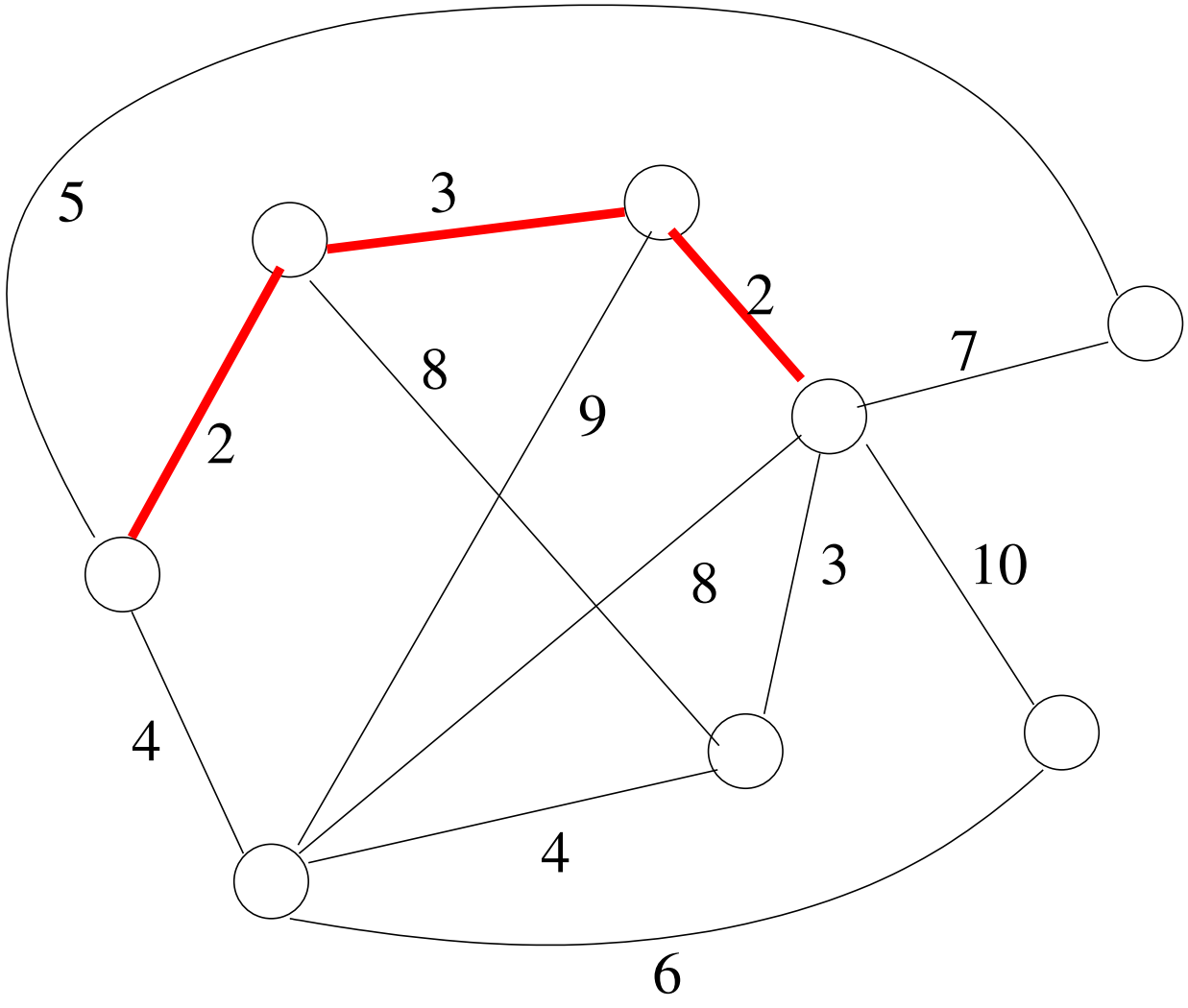
Algoritmo de Kruskal



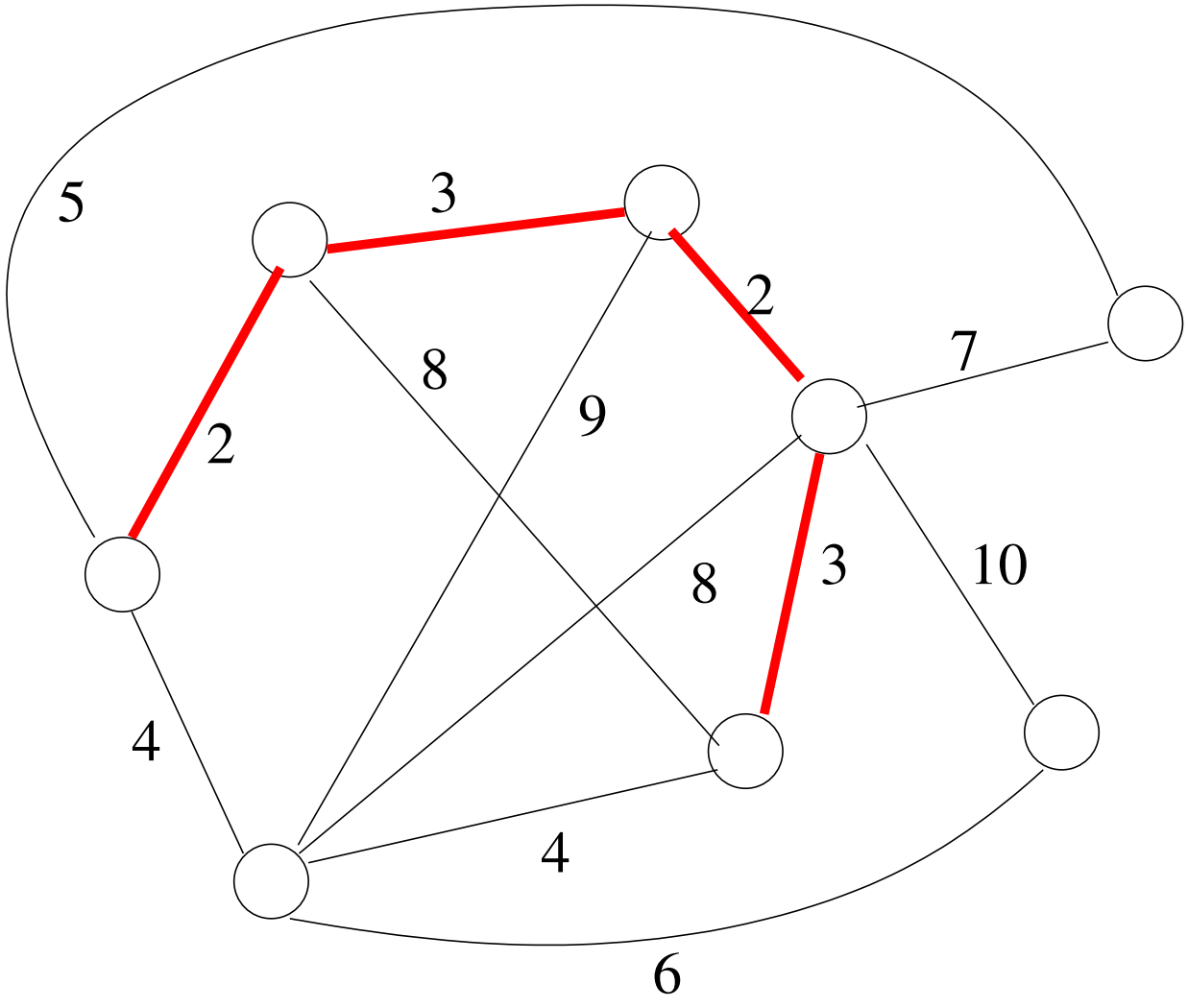
Algoritmo de Kruskal



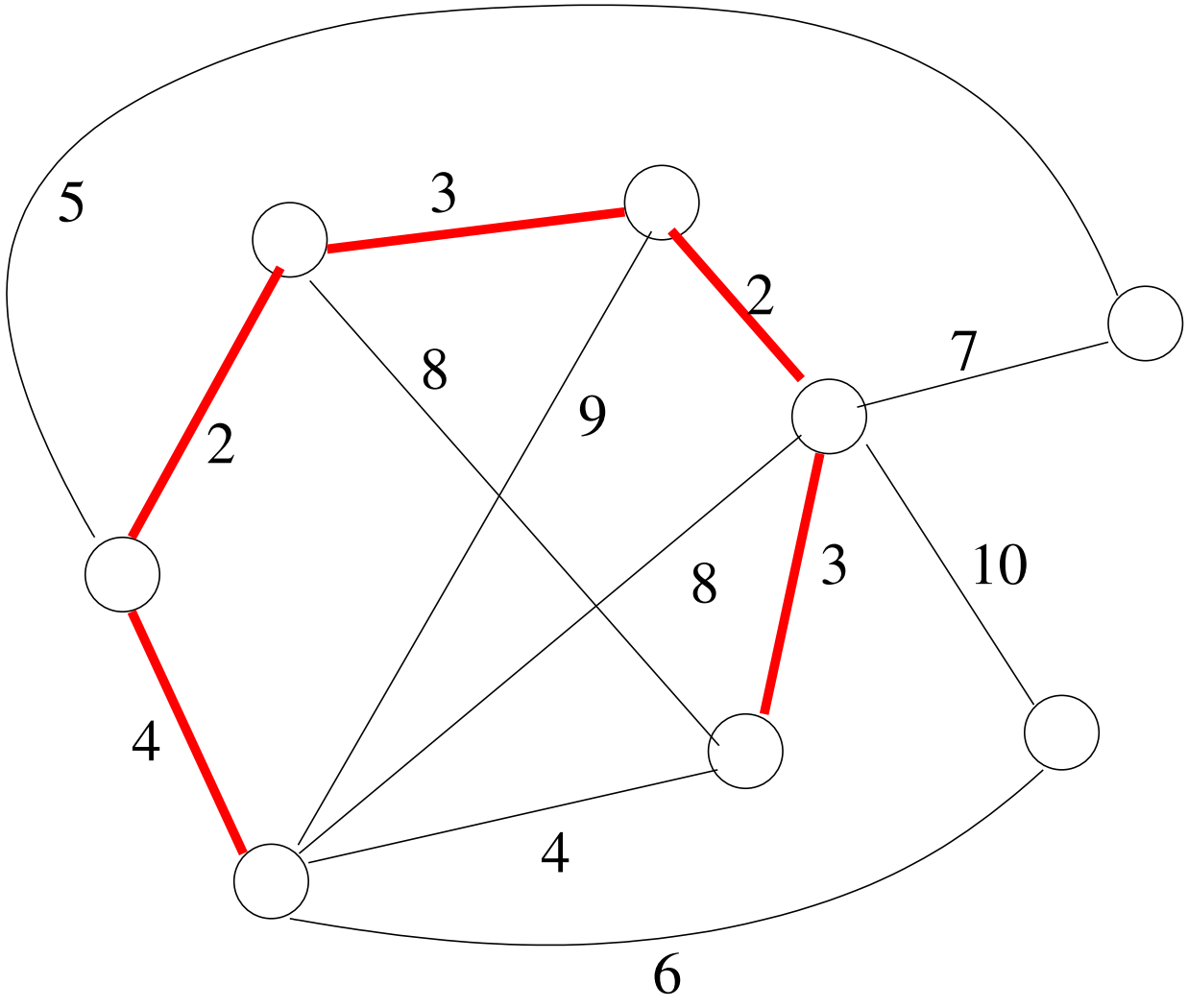
Algoritmo de Kruskal



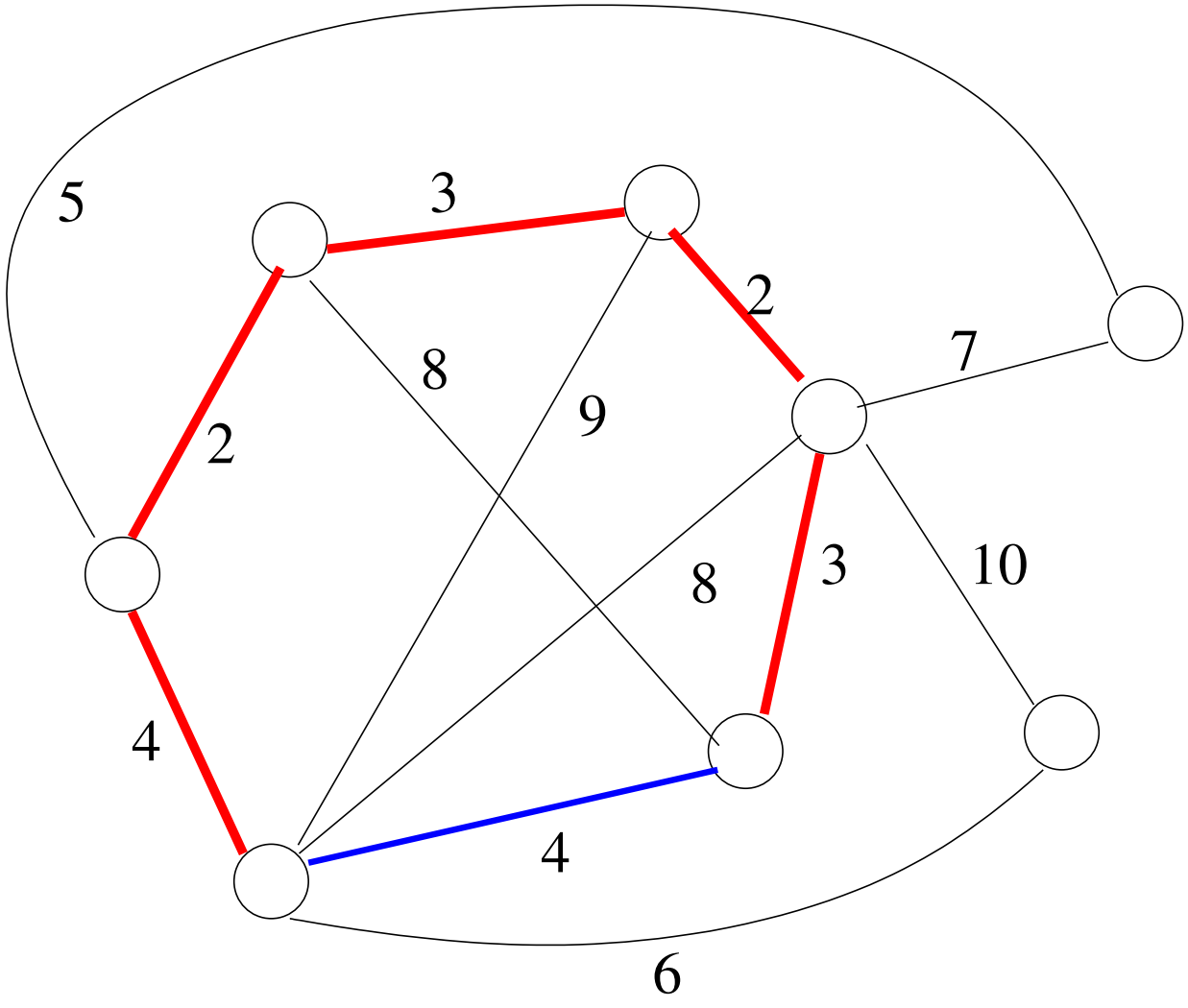
Algoritmo de Kruskal



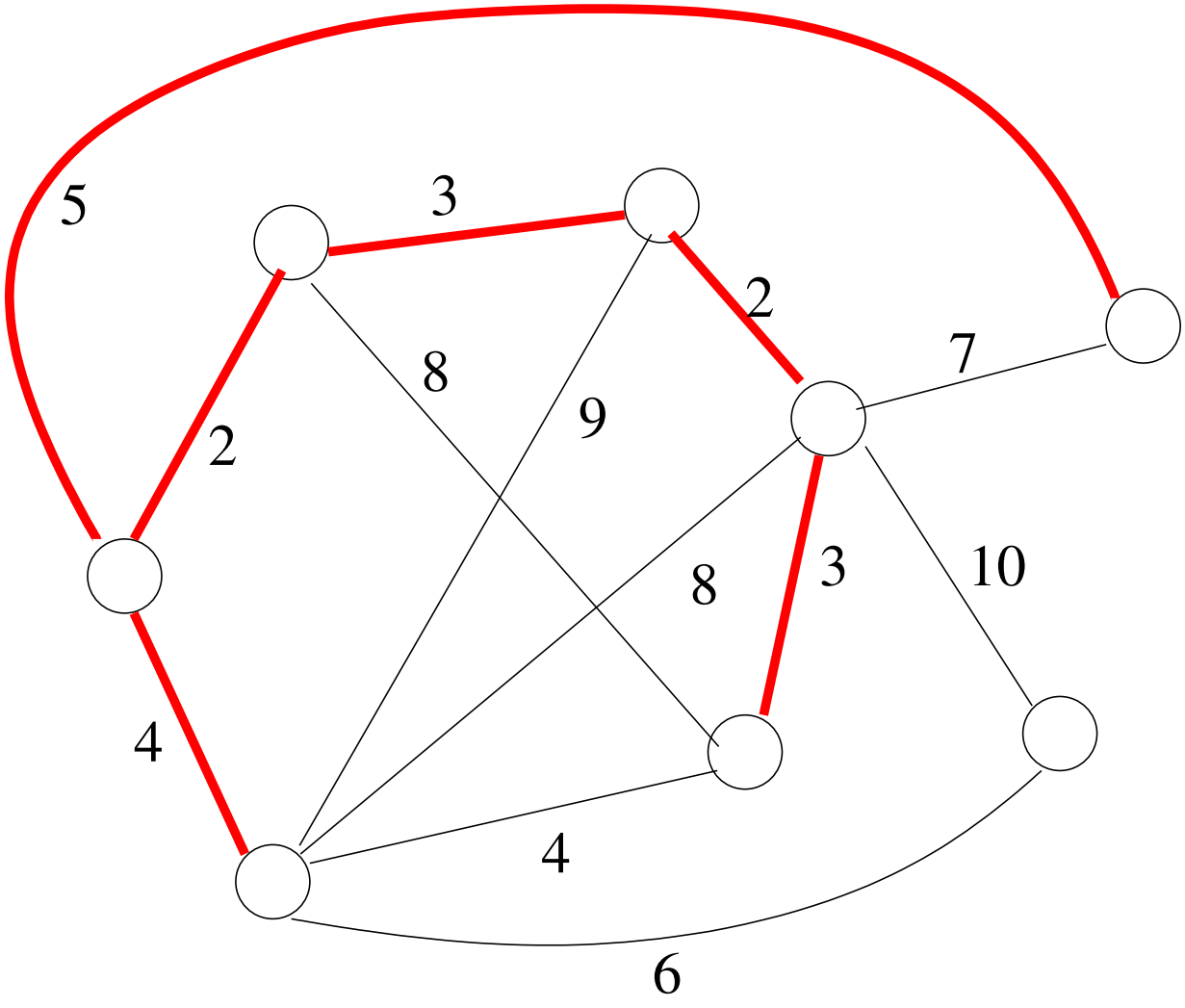
Algoritmo de Kruskal



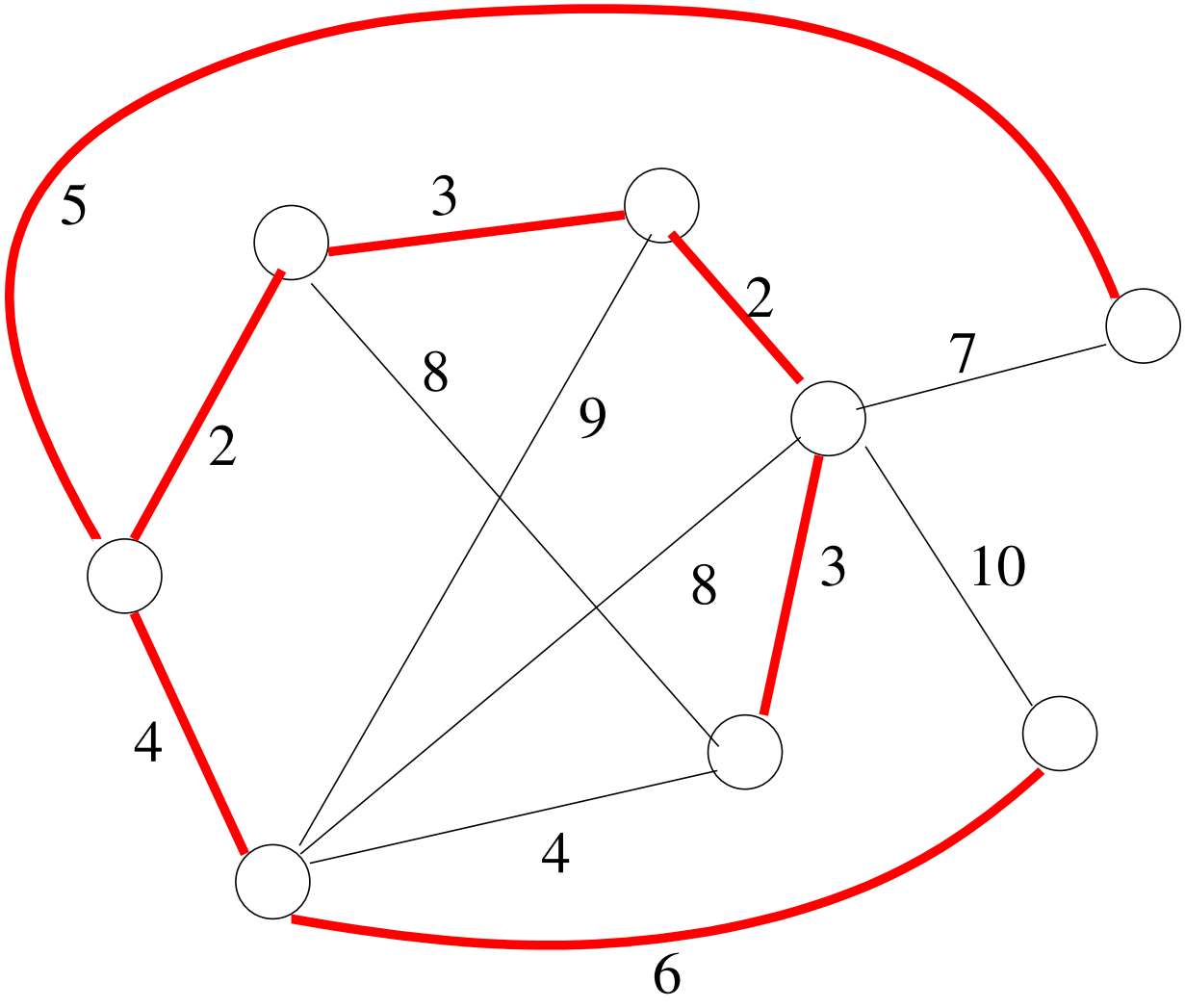
Algoritmo de Kruskal



Algoritmo de Kruskal



Algoritmo de Kruskal



Algoritmo de Kruskal

Para determinar si una cierta arista e cierra un ciclo en A de manera eficiente usaremos una estructura de datos “astuta” denominada PARTICION (también se les conoce como MFSETS y UNION-FIND).
Inicialmente creamos una PARTICION en la que cada vértice del grafo constituye un bloque por sí solo. Cada vez que agreguemos una arista $e = (u, v)$ al conjunto A, los bloques a los que pertenecen u y v se fusionan en un solo bloque (*merge*, *union*).
Cada bloque de vértices es pues una componente conexa de A; si existe un camino entre los vértices w y w' usando las aristas de A, w y w' pertenecerán a un mismo bloque de la PARTICIÓN.

Algoritmo de Kruskal

El test que determina si un nueva arista cierra un ciclo entonces consiste simplemente en ver si los vértices u y v pertenecen a un mismo bloque de la PARTICIÓN o no. Si ya están conectados, la nueva arista cerraría un ciclo; en caso contrario, la nueva arista no cierra un ciclo y los vértices pasan a estar conectados.
Habitualmente la clase PARTICIÓN proporciona una operación FIND que dado un elemento u nos devuelve el representante del bloque en el que está u. Dos elementos están en el mismo bloque si y sólo si los representantes de sus bloques son idénticos.

Algoritmo de Kruskal

procedure KRUSKAL($G = \langle V, E \rangle$)
    Ordenar E por peso creciente
    ▷ Se crea una partición inicial, cada vértice en un bloque distinto
    P.MAKE(V)
    $A := \emptyset$
    while $|A| \neq |V(G)| - 1$ **do**
        $e = (u, v) :=$ SIGUIENTE(E)
        if P.FIND(u) $\neq$ P.FIND(v) **then**
            $A := A \cup \{e\}$
            P.UNION(u, v)
        end if
    end while
    return A
end procedure

Algoritmo de Kruskal

Para calcular el coste del algoritmo de Kruskal tenemos por un lado el coste de la ordenación de las aristas $\Theta(m \log m) = \Theta(m \log n)$, siendo $m = |E(G)|$. La creación de la partición inicial tiene coste $\Theta(n)$, siendo $n = |V(G)|$. Por otro lado, el bucle siguiente hace al menos $n - 1$ iteraciones (tantas como aristas tiene el MST hallado) pero puede llegar a hacer m iteraciones en caso peor. El coste del cuerpo del bucle vendrá dominado por el coste de las llamadas a las operaciones FIND y UNION. Si el bucle principal realiza N iteraciones, $n - 1 \leq N \leq m$, se harán $2N$ llamadas a FIND. Por otro lado, se hacen siempre exactamente $n - 1$ llamadas a UNION.

Algoritmo de Kruskal

Es muy fácil obtener implementaciones de la PARTICIÓN que garanticen que el coste de FIND y UNION es $\mathcal{O}(\log n)$, lo que nos llevaría a un coste

$$\Theta(m \log n) + \mathcal{O}(m \log n) = \Theta(m \log n)$$

para el algoritmo.

Algoritmo de Kruskal

Se puede mejorar el rendimiento notablemente (aunque no en caso peor) de la siguiente forma:

1. En vez de ordenar las aristas por peso, se crea un min-heap con coste $\Theta(m)$
2. Se usa una versión de UNION-FIND que garantiza que $\mathcal{O}(m)$ FINDS + $\mathcal{O}(n)$ UNIONS tienen coste $\mathcal{O}((m+n)\alpha(m,n))$. La función $\alpha(m,n)$, relacionada con ka función inversa de Ackermann crece de manera extremadamente lenta, a los efectos prácticos, por muy grandes que sean m y n, $\alpha(m,n) \leq 4$.
3. El bucle principal seguirá teniendo coste $\mathcal{O}(m \log n)$ pero en la mayoría de casos será más cercano a $\mathcal{O}(n \log n)$ porque no se suelen hacer muchas más del mínimo de $n-1$ iteraciones.

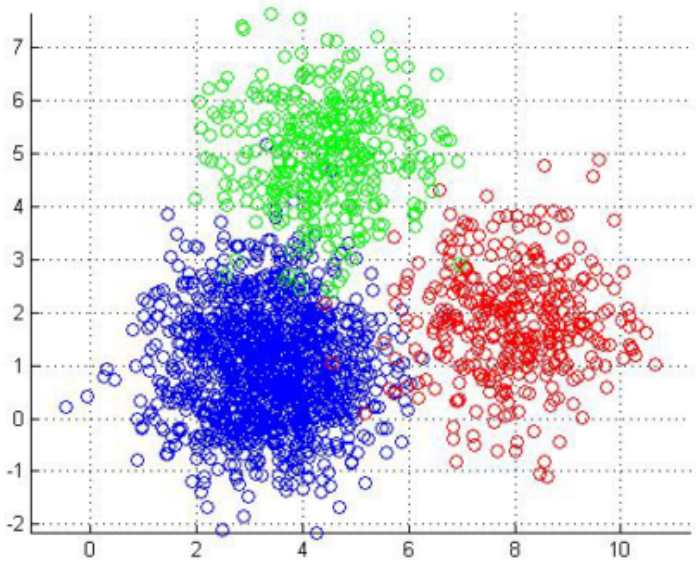
El coste del algoritmo vendrá dominado por el coste de extraer aristas durante las iteraciones del bucle principal y en caso peor es $\Theta(m \log n)$; en promedio será cercano a $\Theta(n \log n)$, lo cual es muy ventajoso, sobre todo si $m \gg n$.

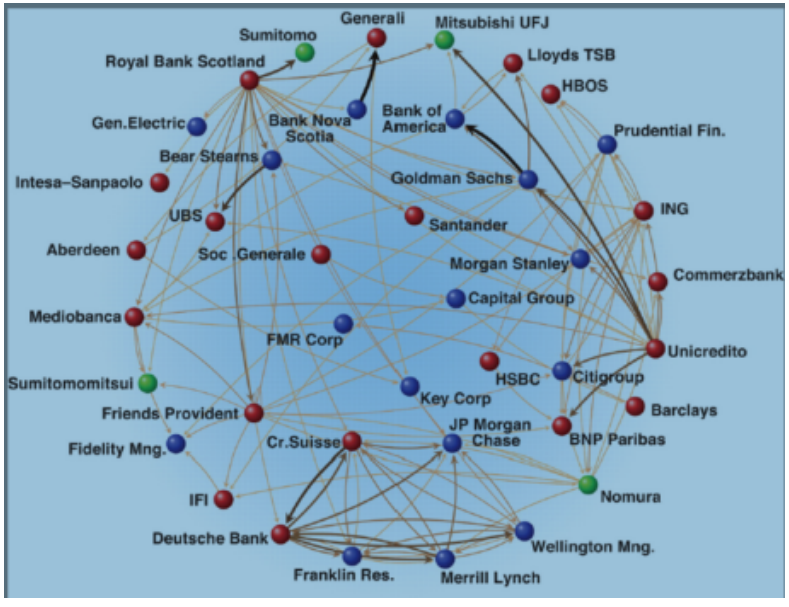
Clustering

- Clustering: el proceso de obtener una partición de un conjunto de n objetos de manera que objetos semejantes estén agrupados en una misma clase (clúster)
- Se nos da un conjunto X con n objetos y una función de distancia $d(\cdot, \cdot)$ (a veces se nos da una función de similitud, con propiedades opuestas a una distancia)
- A menudo la función de distancia es una métrica:
  - $d(x, x) = 0$, $d(x, y) > 0$ si $x \neq y$
  - $d(x, y) = d(y, x)$
  - Desigualdad triangular: $d(x, z) \leq d(x, y) + d(y, z)$

Clustering

Dado un conjunto de objetos $X = \{x_1, \ldots, x_n\}$, una función de distancia $d : X \times X \to \mathbb{R}^+$ y un valor $k > 0$, un k-clustering de X es una partición de X en k subcojuntos disjuntos.





Clustering

El objetivo es que si x e y están en el mismo clúster entonces $d(x, y) \leq d(x, z)$ para cualquier z que **no** esté en el mismo clúster.

Se define la separación s (*spacing*) de un k-clustering $C = \{C_1, \ldots, C_k\}$ como la mínima distancia entre dos objetos x_i e x_j que pertenezcan a clústers diferentes de C.

Problema del *linkage simple* (*single-link problem*)

Definición

Dado un conjunto X, una función de distancia sobre X y un valor $k > 0$, hallar un k-clustering que maximiza la separación. Esto es, si $\mathcal{C}(k, X)$ es el conjunto de todos los posibles k-clusterings de X, hallar un C^* tal que

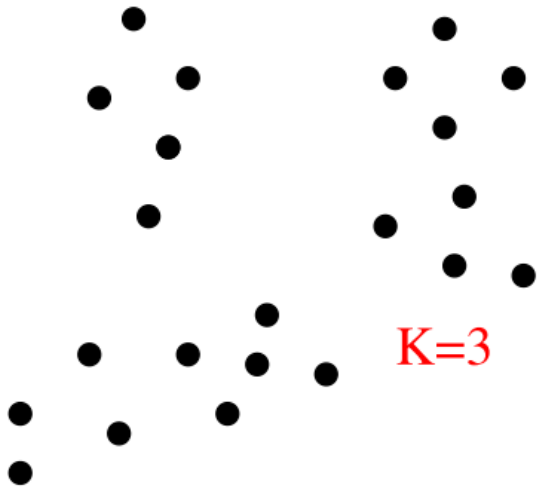
$$\text{spacing}(C^*) = \max_{C \in \mathcal{C}(k,X)} \text{spacing}(C)$$

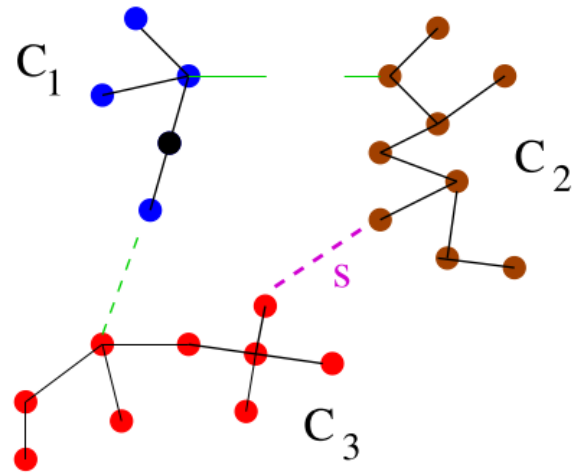
N.B. El tamaño de $\mathcal{C}(k, X)$ es exponencial respecto a $n = |X|$.

Problema del *linkage simple* (*single-link problem*)

Algoritmo TrKruskal:

- Se representa X como los vértices de un grafo no dirigido donde la arista (x, y) tiene peso $d(x, y)$
- Se aplica el algoritmo de Kruskal hasta que el bosque de expansión tiene exactamente k árboles (la estructura Union-Find consta de k bloques). Cada árbol constituye un clúster.





Problema del *linkage simple* (*single-link problem*)

Teorema

TrKruskal soluciona el problema del linkage simple en tiempo $\mathcal{O}(n^2 \lg n)$

Demostración

Coste:
La ordenación de las $n(n-1)/2$ aristas del grafo completo lleva tiempo $\mathcal{O}(n^2 \log n)$. El bucle principal hace a lo sumo $m = n(n-1)/2$ iteraciones y usando un estructura de datos Union-Find el coste de cada iteración es $\mathcal{O}(\log n)$, de manera que en caso peor el coste del bucle también es $\mathcal{O}(n^2 \log n)$.

Problema del *linkage simple* (*single-link problem*)

Demostración (continúa)

Corrección:
Sea $C = (C_1, \ldots, C_k)$ el k-clustering obtenido por TrKruskal, y sea $s = \text{spacing}(C)$.

Sea $C' = \{C'_1, \ldots, C'_k\}$ otro k-clustering ($C' \neq C$) con separación s'. Mostraremos que $s' \leq s$, esto es, que la separación de C es máxima y por tanto C una solución correcta del problema.

Problema del *linkage simple* (*single-link problem*)

Demostración (continúa)

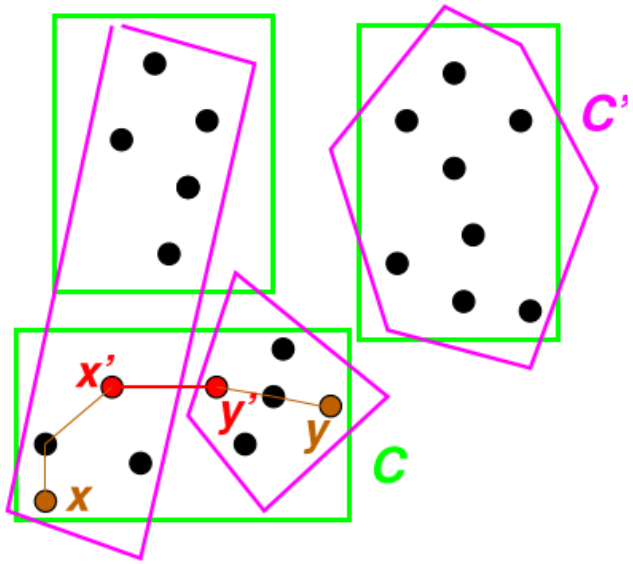
Sea $C_r \in C$ tal que $C_r \not\subseteq C'_t$ para todo clúster $C'_t \in C'$ (C_r tiene que existir porque $C \neq C'$).

Es decir, existen $x, y \in C_r$ tales que $x \in C'_p$ e $y \in C'_q$ para algunos p, q, $p \neq q$.

Problema del *linkage simple* (*single-link problem*)

Demostración (continúa)

Sea T el MST de G (TrKruskal nos devuelve un "bosque" $C \subset T$). Existe un camino $x \rightsquigarrow y$ en $C_r \implies$ toda arista en el camino tiene peso $\leq s \implies$ existe una arista (x', y') del camino donde x' e y' están en clústers distintos necesariamente y $d(x', y') \leq s$. Pero la separación s' en C' será $d(x', y')$ o menor, luego la separación s' de C' es $\leq s$.



□

Parte III

Algoritmos Voraces

Particiones

Una *partición* Π de un conjunto no vacío $\mathcal{A}$ es una colección de subconjuntos no vacíos $\Pi = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ tal que

1. Si $i \neq j$ entonces $\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$.
2. $\mathcal{A} = \bigcup_{1 \leq i \leq k} \mathcal{A}_i$.

Se suele denominar *bloque* de la partición Π a cada uno de los $\mathcal{A}_i$'s. Las particiones y las relaciones de equivalencia están estrechamente ligadas. Recordemos que $\equiv$ es una relación de equivalencia en $\mathcal{A}$ si y sólo si

1. $\equiv$ es reflexiva: para todo $a \in \mathcal{A}$, $a \equiv a$.
2. $\equiv$ es transitiva: si $a \equiv b$ y $b \equiv c$, entonces $a \equiv c$, para cualesquiera a, b y c en $\mathcal{A}$.
3. $\equiv$ es simétrica: $a \equiv b$ si y sólo si $b \equiv a$, para cualesquiera a y b en $\mathcal{A}$.

Dada una partición Π de $\mathcal{A}$, ésta induce una relación de equivalencia $\equiv_\Pi$ definida por

$$x \equiv_\Pi y \iff x \text{ e } y \text{ pertenecen a un mismo bloque } \mathcal{A}_i \in \Pi$$

Y a la inversa, dada una relación de equivalencia $\equiv$ en $\mathcal{A}$, ésta induce una partición $\Pi = \{\mathcal{A}_x\}_{x \in \mathcal{A}}$, donde

$$A_x = \{y \in \mathcal{A} \,|\, y \equiv x\}.$$

Al subconjunto de elementos equivalentes a x se le denomina *clase de equivalencia* de x. Cada uno de los bloques de la partición inducida por una relación $\equiv$ es por lo tanto una clase de equivalencia. Nótese que si $x \equiv y$ entonces $\mathcal{A}_x = \mathcal{A}_y$. Un elemento cualquiera de la clase $\mathcal{A}_x$ se denomina *representante* de la clase.

En muchos algoritmos, especialmente en algoritmos sobre grafos, es importante poder representar particiones de un conjunto finito de manera eficiente.

Vamos a suponer que el conjunto *soporte* sobre el que se define la partición es $\{1, \ldots, n\}$; sin excesiva dificultad, empleando alguna de las técnicas vistas en temas precedentes podemos representar de manera eficiente una biyección entre un conjunto finito $\mathcal{A}$ de cardinalidad n y el conjunto $\{1, \ldots, n\}$ y con ello una partición sobre el conjunto soporte $\mathcal{A}$ en caso necesario.

Adicionalmente, supondremos que el conjunto soporte es estático, es decir, ni se añaden ni se eliminan elementos. No obstante, con poca dificultad extra podemos obtener representaciones eficientes para particiones de conjuntos dinámicos.

Generalmente el tipo de operaciones que una clase `Particion` debe soportar son las siguientes: 1) dados dos elementos i y j, determinar si pertenecen al mismo bloque o no; 2) dados dos elementos i y j fusionar los bloques a los que pertenecen (si procede) en un sólo bloque, devolviendo la partición resultante.
Frecuentemente el primer tipo de operación se realiza mediante una operación FIND que dado un elemento i, devuelve un representante de la clase a la que pertenece i. Si dos elementos i y j tienen el mismo representante, entonces han de estar en el mismo bloque.

El segundo tipo de operación se llama MERGE o UNION. De ahí que las particiones se les llame a menudo estructuras union-find o mfsets (abreviación de *merge-find sets*).
La operación MAKE crea una partición de $\{1, \ldots, n\}$ consistente en n bloques cada uno de los cuales contiene un elemento.

En muchas aplicaciones antes de hacer cualquier UNION se habrá determinado previamente si los elementos i y j cuyos bloques habrían de unirse, están o no en el mismo bloque; para ello se habrá preguntado si $P.\text{FIND}(i) = P.\text{FIND}(j)$ o no. Por esta razón es habitual que en una clase `Particion` la operación UNION sólo actúe sobre elementos que son representantes de sus respectivas clases.

```
procedure F(...)
    ...
    ri := P.FIND(i)
    rj := P.FIND(j)
    if ri ≠ rj then
        P.UNION(ri, rj)
        ...
    end if
end procedure
```

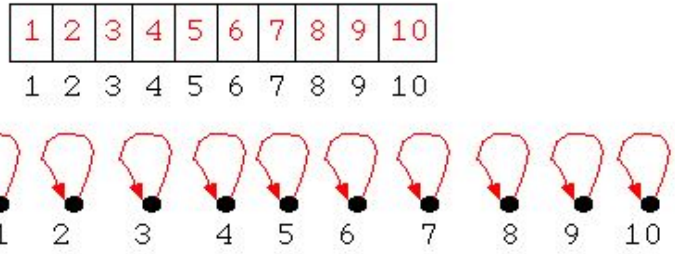
Puesto que la operación MAKE recibe como parámetro el número de elementos n del conjunto soporte, podremos utilizar implementaciones en vector, reclamando un vector con el número apropiado de componentes a la memoria dinámica si nuestro lenguaje de programación lo soporta. También será posible utilizar este tipo de implementación si el valor de n está acotado y dicha cota no es irrazonablemente grande. En cualquier caso podremos modificar sin demasiado esfuerzo las implementaciones que se verán a continuación para que sean completamente dinámicas y funcionen correctamente en aquellos casos en que no se puedan crear vectores cuyo tamaño se fija en tiempo de ejecución o si el conjunto soporte ha de soportar inserciones y borrados.

Quick-find consiste representar la partición mediante un vector P y almacenar en la componente i de P el representante de la clase a la que pertenece i. De este modo la operación FIND tiene coste constante, ya que basta examinar $P[i]$. Sin embargo, la operación UNION tendrá coste $\Theta(n)$ ya que cada uno de los elementos de la clase en la que está j (los k's tales que $P[k] = P[j]$) han de cambiar de representante ($P[k] := P[i]$). O bien los elementos del mismo bloque que i han de pasar al bloque de j.

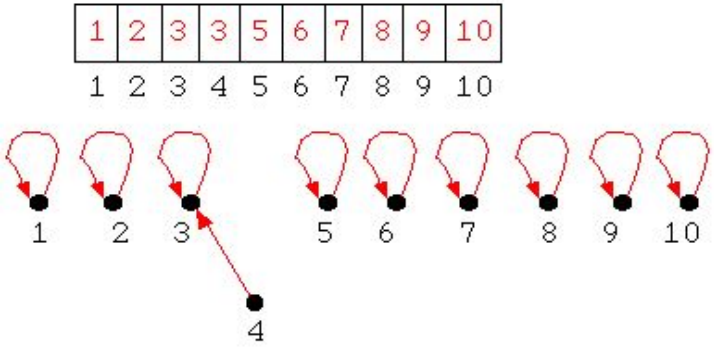
Con algunas modificaciones puede evitarse el recorrido completo del vector P y restringirlo a los elementos del bloque de j (o del bloque de i); pero aún así el coste de una UNION sigue siendo lineal en n en el caso peor, ya que cualquiera de los dos bloques pude contener una fracción considerable del total de los elementos.

Aunque resulta un tanto forzado, conviene contemplar la representación *quick-find* como un bosque de árboles; cada árbol representa a un bloque de la partición en un momento dado. Los árboles están representados mediante apuntadores al padre, siendo la raíz de cada árbol el representante del bloque correspondiente. Puesto que la raíz de un árbol no tiene padre, las raíces se apuntan a sí mismas. Del invariante de la representación de *quick-find* se sigue que todos los árboles tienen altura 1 (si sólo contienen un elemento) o altura 2 (todos los elementos de un bloque excepto el representante están en el segundo nivel, apuntando a la raíz).

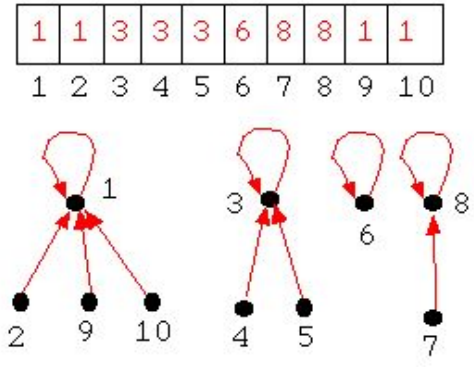
make(10)



union(3,4)



union(1,2); union(4,5); union(1,9);
union(2, 10); union(8,7)



Otra estrategia, *quick-union*, explota la correspondenicia entre bloques y árboles del siguiente modo: para unir los bloques de i y j se localizan al representante de i, digamos u, y se coloca a u como hijo de j. Alternativamente, podemos localizar ambos representantes y hacer que uno de ellos sea hijo del otro.

```
procedure MAKE(n)
    for i := 1 to n do
        P[i] := i
    end for
end procedure
procedure UNION(i, j)
    u := FIND(i)
    v := FIND(j)
    P[u] := v
end procedure
▷ 1 ≤ i ≤ n
procedure FIND(i)
    while P[i] ≠ i do
        i := P[i]
    end while
    return i
end procedure
```

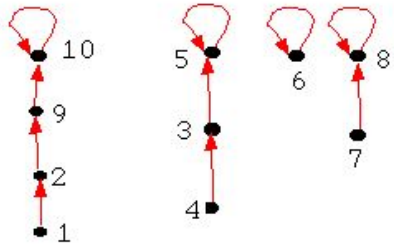
Aunque en general los árboles resultantes de una secuencia de uniones serán relativamente equilibrados y el coste de las operaciones será bajo, en caso peor podemos crear árboles poco equilibrados de modo que tanto una UNION como un FIND tengan coste proporcional al número de elementos involucrados. Por ejemplo, si realizamos una secuencia de UNIONes de tal modo que la clase en la que está j sólo contenga a j, obtendremos árboles equivalentes a listas.

make(10); union(4, 3)

1	2	3	3	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

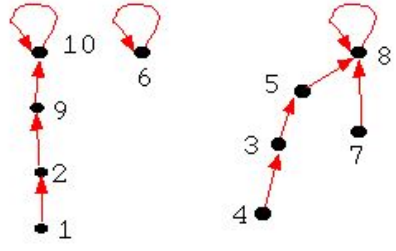
union(1,2); union(4,5); union(1,9);
union(2, 10); union(7,8)

2	9	5	3	5	6	8	8	10	10
1	2	3	4	5	6	7	8	9	10



union(4,7);

2	9	5	3	8	6	8	8	10	10
1	2	3	4	5	6	7	8	9	10



Los comentarios previos sugieren posibles soluciones al problema. En la *unión por peso* el árbol con menos elementos es que el que se añade como hijo del que tiene más elementos. En la *unión por rango* el árbol de menor altura es el que se pone como hijo del de mayor altura. Tanto una como otra estrategia son fáciles de implementar, pero requieren, en principio, que se almacene información auxiliar sobre el tamaño o la altura de los árboles.

Se puede evitar el uso de espacio auxiliar observando que sólo se necesita esta información de tamaño o altura para las raíces y que el espacio correspondiente a sus apuntadores es esencialmente inútil, ya que sólo se precisaría un bit que indique que i es una raíz o no. Por ejemplo, podemos adoptar el convenio de que si $P[i] < 0$ entonces i es una raíz y $-P[i]$ es el tamaño del árbol.

```
procedure UNION(i, j)
    u := FIND(i)
    v := FIND(j)
    if −P[u] > −P[v] then
        ▷ u es el árbol “grande”
        u :=: v
    end if
    ▷ u es el árbol “pequeño”
    P[v] := P[v] + P[u]
    P[u] := v
end procedure
▷ 1 ≤ i ≤ n
procedure FIND(i)
    while P[i] > 0 do
        i := P[i]
    end while
    return i
end procedure
```

El rendimiento $\mathcal{O}(\log n)$ de estas operaciones es consecuencia directa del siguiente lema.

Lema

Dado un bloque de tamaño k en una `Particion` *con unión por peso, la altura del árbol correspondiente es* $\leq \log_2 k$.

Demostración

Si $k = 0$, el lema es obviamente cierto. Supongamos que es cierto para todos los tamaños hasta k y demostraremos entonces que es cierto para $k + 1$. Sea t el árbol correspondiente a un bloque de tamaño $k + 1$. Dicho bloque es el resultado de la unión de dos bloques de tamaños r y s, $r \leq s \leq k$. El árbol t tiene altura $h(t) \leq \max\{\log_2 r + 1, \log_2 s\}$, aplicando la hipótesis de inducción, y por la definición de unión por peso. Supongamos que $\log_2 r + 1 \leq \log_2 s$. Entonces

$$\begin{aligned} h(t) &\leq \max\{\log_2 r + 1, \log_2 s\} \\ &= \log_2 s < \log_2(k+1). \end{aligned}$$

Demostración (continúa)

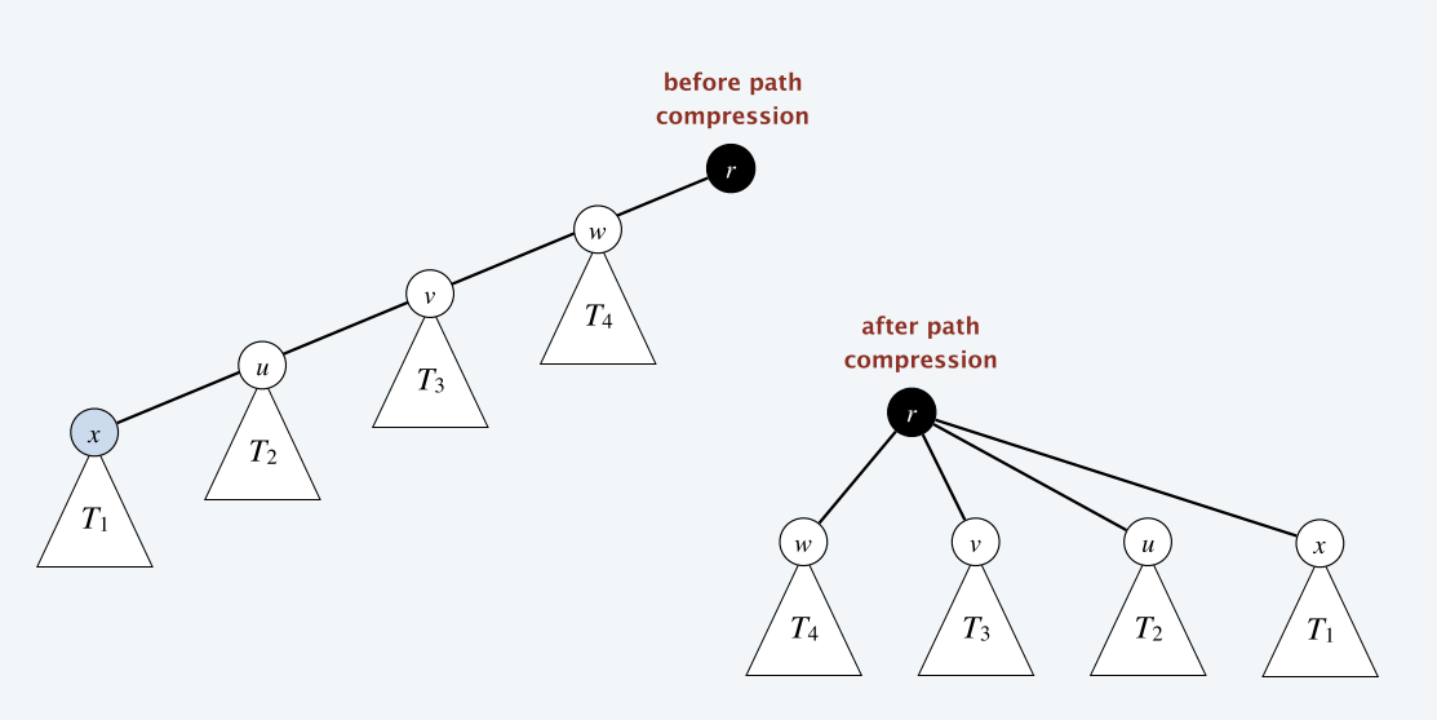
Por otro lado, si $\log_2 r + 1 > \log_2 s$, y teniendo en cuenta que $k + 1 = r + s \geq 2r$,

$$\begin{aligned} h(t) &\leq \text{máx}\{\log_2 r + 1, \log_2 s\} \\ &= \log_2 r + 1 = \log_2(2r) \leq \log_2(k+1). \end{aligned}$$

□

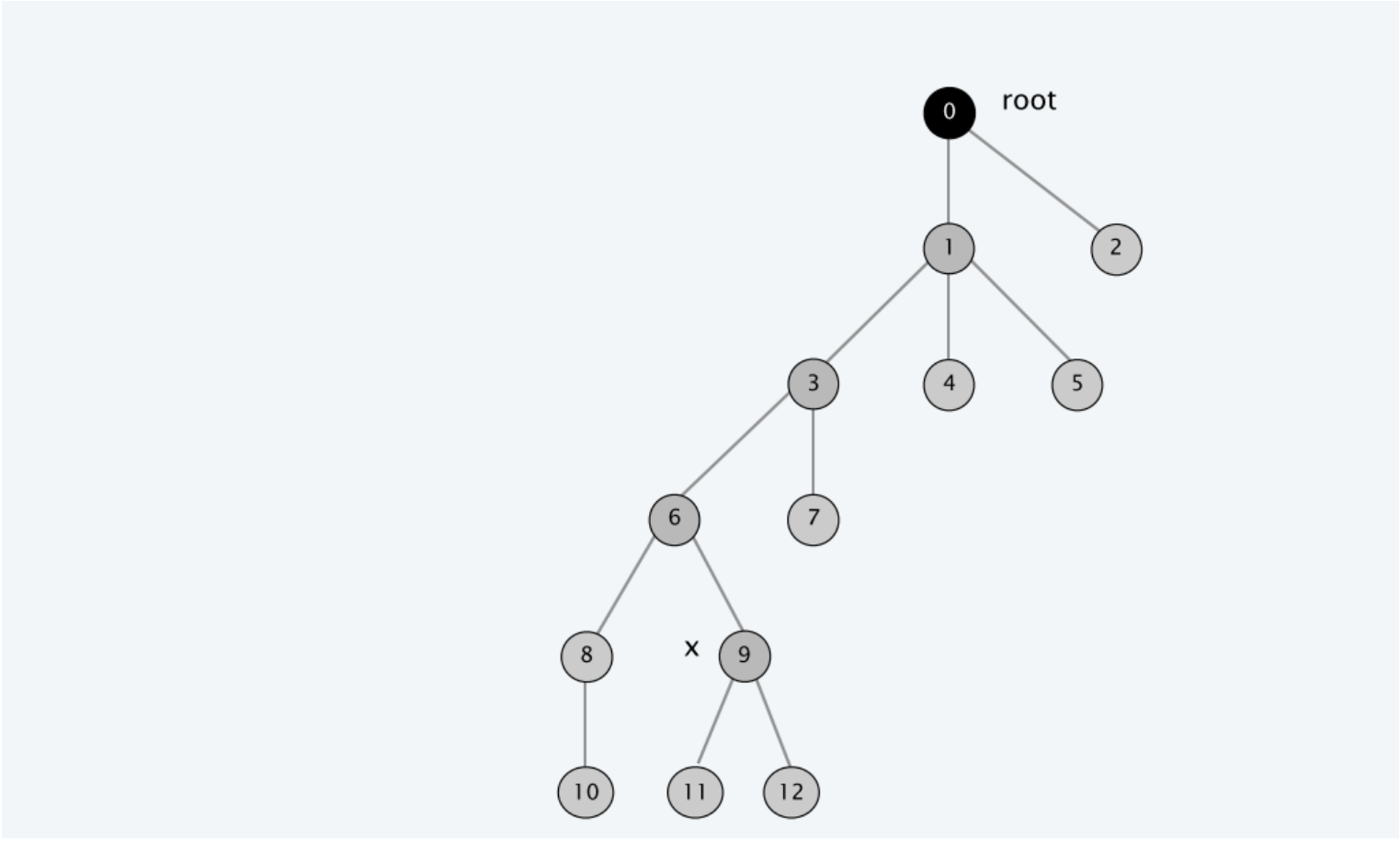
Todavía puede conseguirse un mejor rendimiento empleando una heurística de *compresión de caminos*. La idea es reducir la distancia a la raíz de los elementos en el camino de i hasta la raíz durante una operación FIND(i). Mientras se asciende desde i hasta la raíz se disminuye la altura del árbol. Subsiguientes FINDs que afecten a i o alguno de los elementos que eran antecesores suyos serán más rápidos. La compresión de caminos acorta caminos de manera que poco a poco los árboles adoptan la forma que tendrían con *quick-find*, pero no teniendo que encargarse de ello la operación UNION ni compactándose todo un árbol de una sola vez.

Compresión total : se modifica el apuntador de cada elemento en el camino desde i hasta la raíz, para que apunten a la raíz.

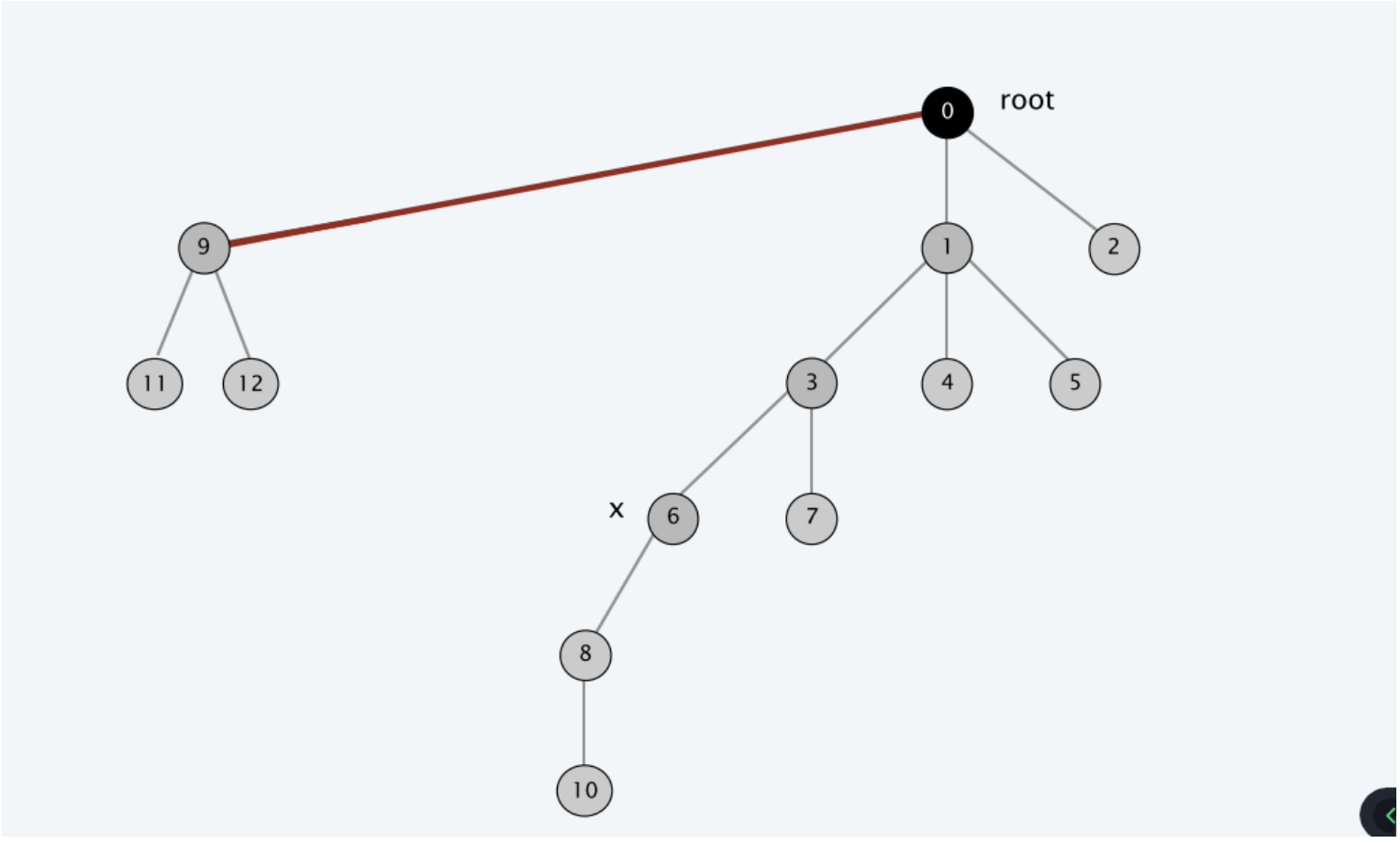


Fuente: Kevin Wayne
(https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf)

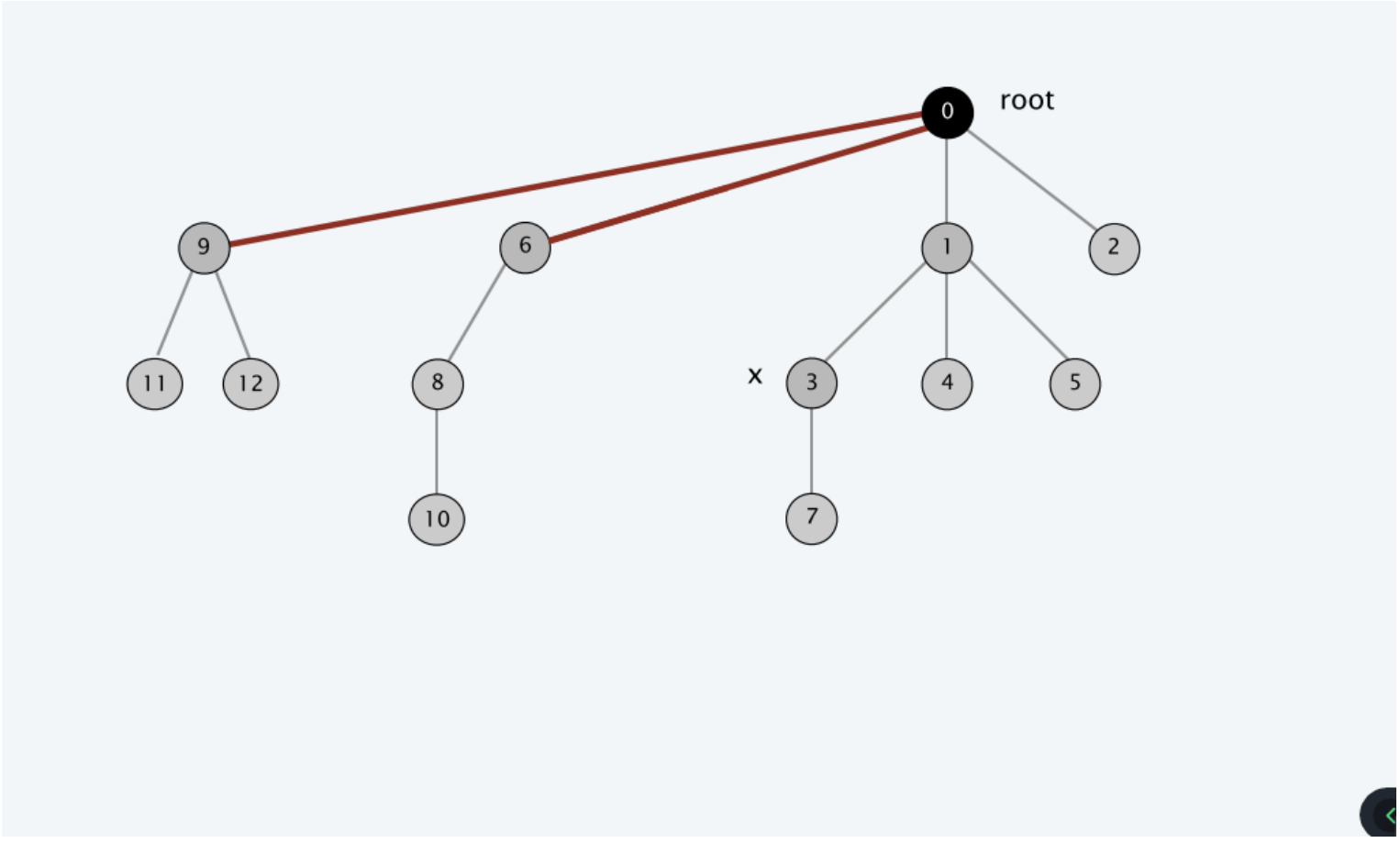
Compresión total : se modifica el apuntador de cada elemento en el camino desde i hasta la raíz, para que apunten a la raíz.



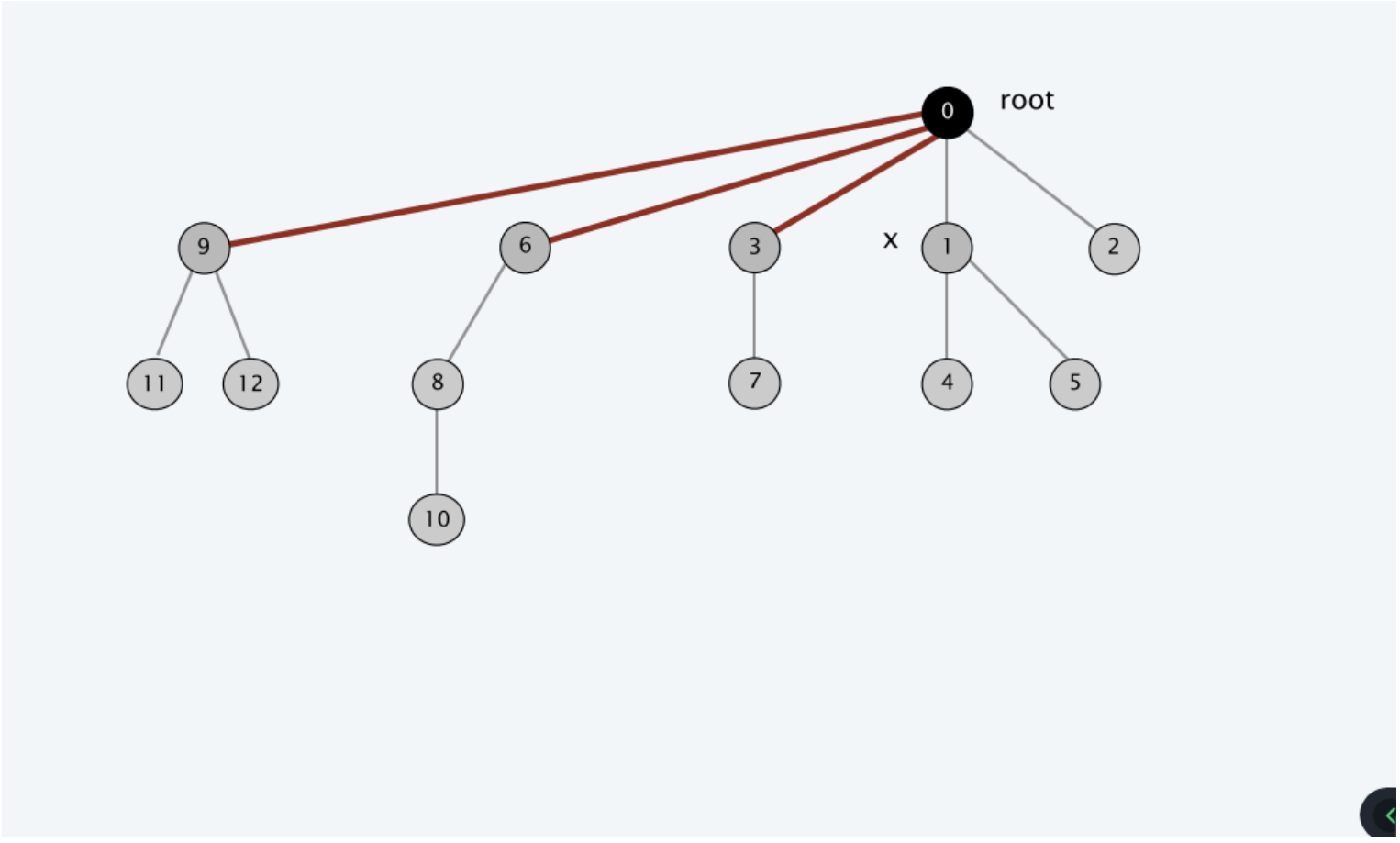
Compresión total : se modifica el apuntador de cada elemento en el camino desde i hasta la raíz, para que apunten a la raíz.



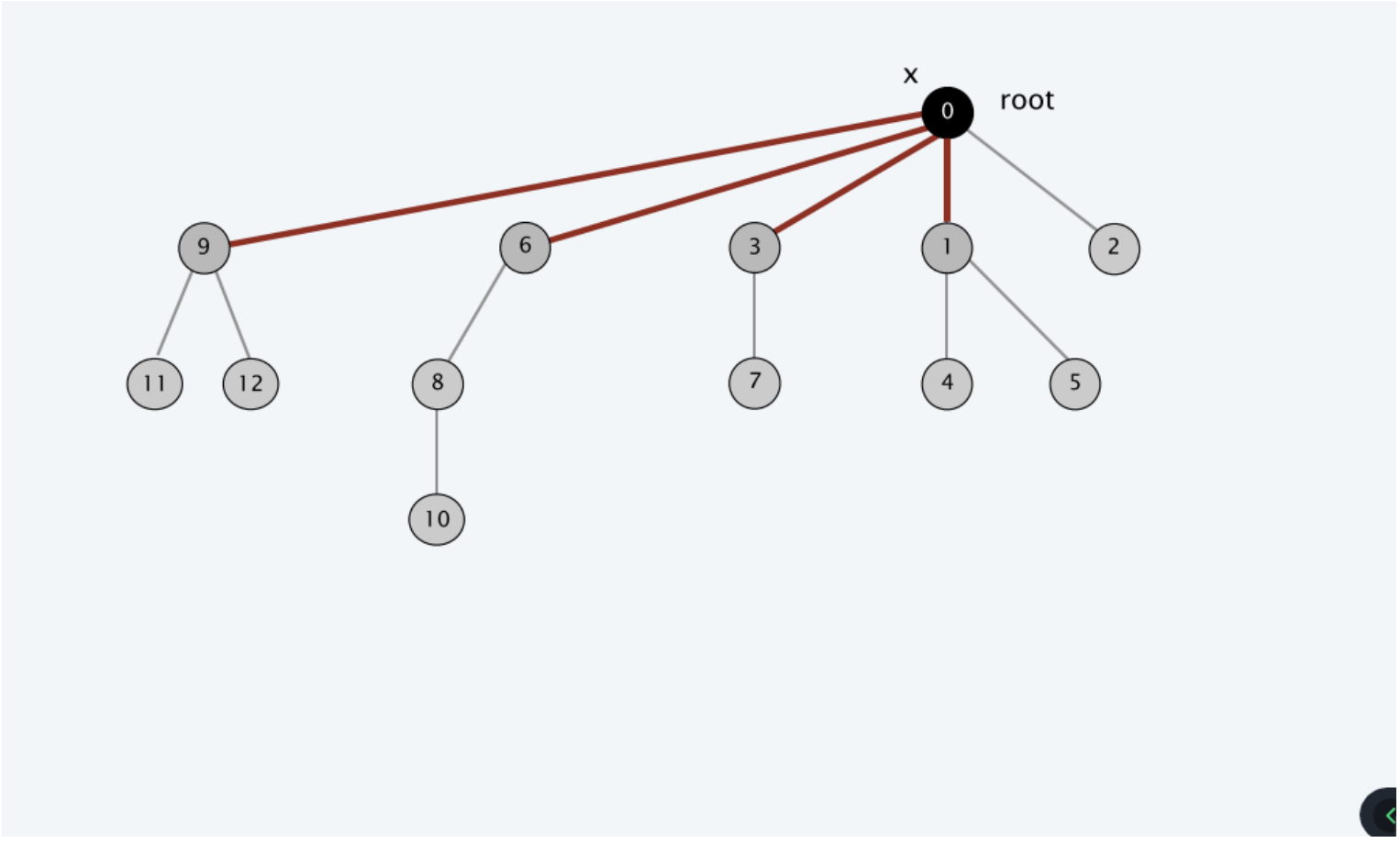
Compresión total : se modifica el apuntador de cada elemento en el camino desde i hasta la raíz, para que apunten a la raíz.



Compresión total : se modifica el apuntador de cada elemento en el camino desde i hasta la raíz, para que apunten a la raíz.

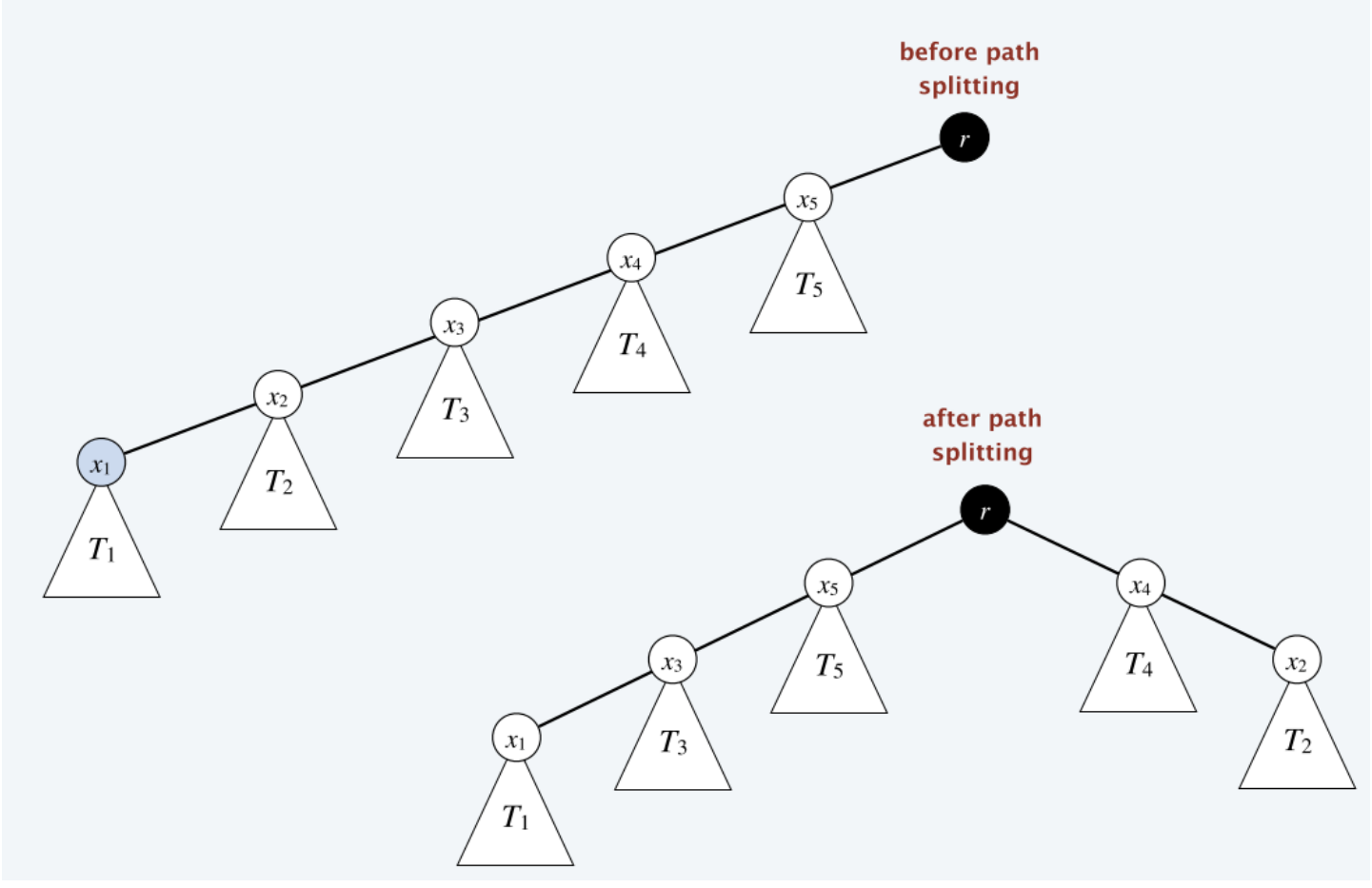


Compresión total : se modifica el apuntador de cada elemento en el camino desde i hasta la raíz, para que apunten a la raíz.



```
procedure FIND(i)
    if P[i] < 0 then
        return i
    else
        P[i] := FIND(P[i])
        return P[i]
    end if
end procedure
```

Path splitting: se modifica el apuntador de cada elemento en el camino desde i hasta FIND(i), excepto a FIND(i) y el hijo de éste, para que apunte a su “abuelo”.

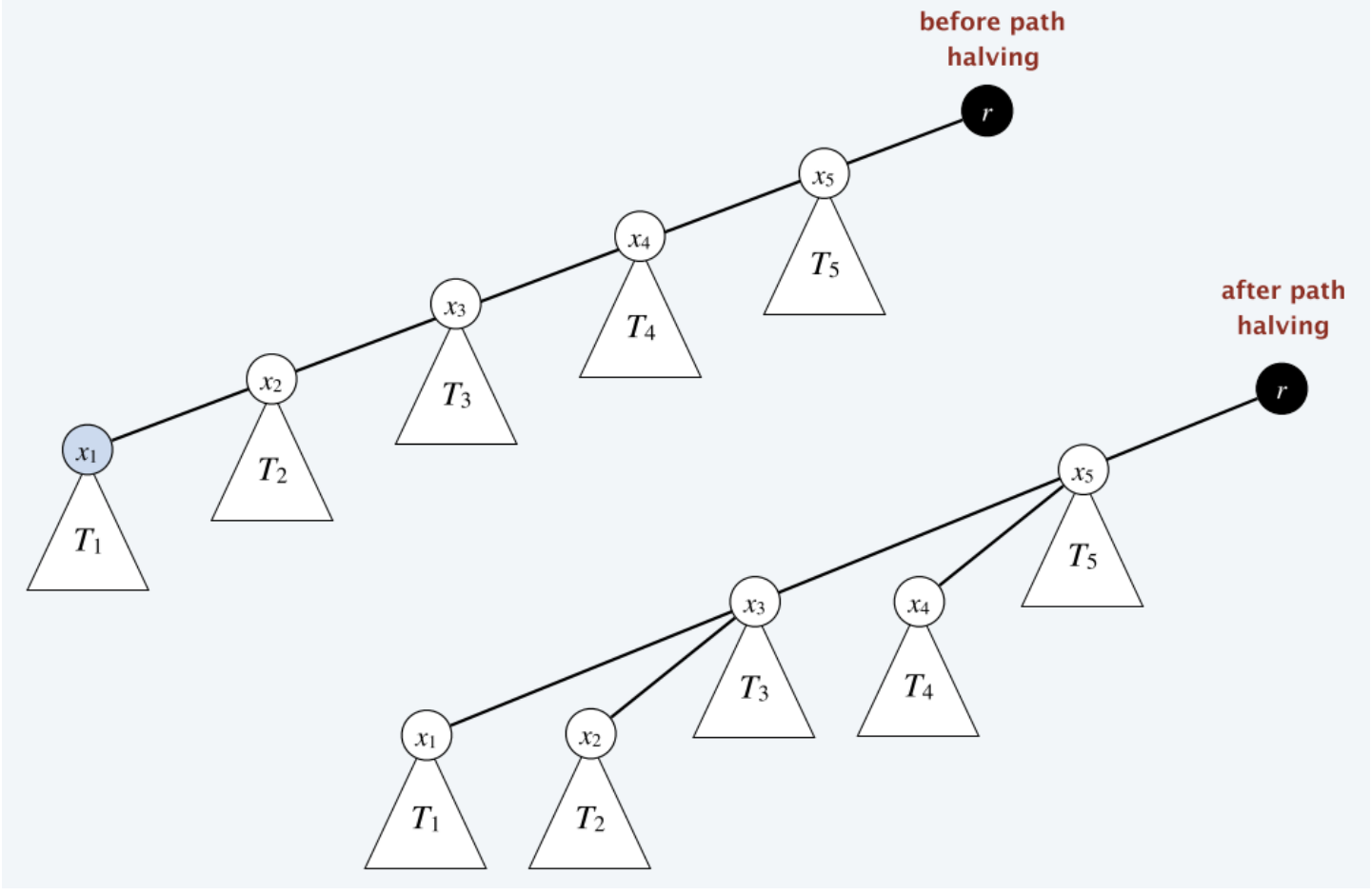


Fuente: Kevin Wayne
(https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf)

Path splitting: se modifica el apuntador de cada elemento en el camino desde i hasta FIND(i), excepto a FIND(i) y el hijo de éste, para que apunte a su “abuelo”.

procedure FIND(i)
    $j := i; pj := P[j]$
    while $P[pj] > 0$ **do**
        $P[j] := P[pj]$
        $j := pj$
        $pj := P[j]$
    end while
    return pj
end procedure

Path halving: se modifican los apuntadores de los elementos alternados en el camino desde i hasta FIND(i) para que apunten a su “abuelo”.



Fuente: Kevin Wayne
(https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf)

Se ha demostrado que una secuencia de m UNIONes y n FINDs usando una de estas dos técnicas tiene coste $\mathcal{O}((m+n)\cdot\alpha(m,n))$, donde $\alpha(m,n)$ es una *función inversa de Ackermann*. Crece extremadamente despacio. Puesto que $\alpha(m,n)\leq 4$ para cualesquiera valores de m y n concebibles en la práctica, el coste puede considerarse $\mathcal{O}(m+n)$. Aunque el coste de una UNION o un FIND no es constante, el coste amortizado (prácticamente) sí lo es ya que el coste de las $m+n$ operaciones es $\mathcal{O}((m+n)\,\alpha(m,n))$ en caso peor.