# Unidad 4. Exploración en grafos


Imparten

Francisco Palomo Lozano

Leopoldo Gutiérrez Galeano

Alfredo Sánchez-Roselly Navarro

Impartieron en cursos pasados

I. Medina, A. García, A. Salguero


# Introducción

Muchos problemas interesantes pueden modelarse con árboles o grafos. Su resolución implica, a veces, una exploración exhaustiva.
Si el grafo es muy grande (en especial, si es infinito), puede ser imposible construirlo. Al ser inviable explorarlo en su totalidad, lo más que podemos esperar es construir ciertas partes sobre la marcha.
Para ello se usa un grafo implícito para el que se dispone de una descripción de sus vértices y aristas, es decir, de las instrucciones sobre cómo generarlo. Esto permite construir partes de él conforme se explora.
Al construir únicamente las partes relevantes para la resolución del problema, es posible ahorrar tiempo y espacio.

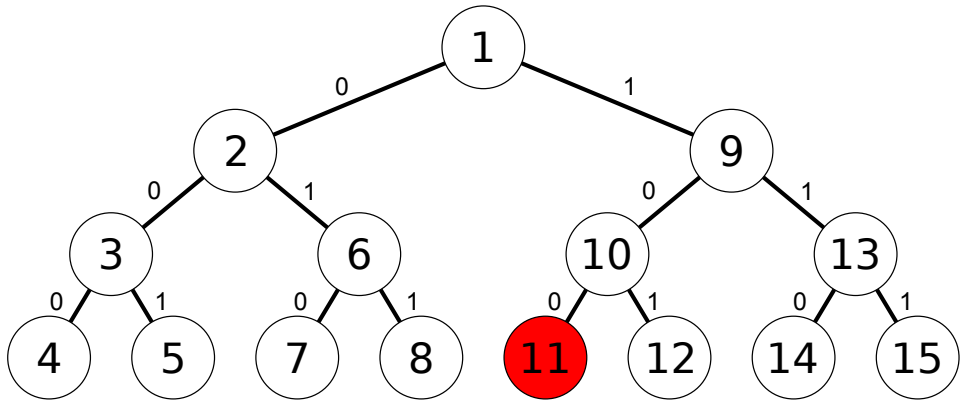
# Introducción

En resumen, el concepto de grafo se emplea de dos formas diferentes:

1. Como estructura de datos explícita.
2. Como estructura de datos implícita.

Un grafo explícito se almacena en memoria, mientras que un grafo implícito no tiene por qué estar completamente en memoria.
Suele ser necesario realizar un marcado de los vértices visitados para evitar visitar un mismo vértice dos veces. Esto permite además evitar los ciclos.

# Búsqueda por fuerza bruta




Solución

| |
|---|
| 1 |
| 0 |
| 0 |

## Idea general

- Se explora el espacio de búsqueda exhaustivamente.
- Para espacios finitos, encuentra una solución si existe.
- Para espacios infinitos, no siempre encuentra una solución.
- En algunos problemas es necesario buscar todas las soluciones.
- En un problema de optimización, se va guardando la mejor solución.
- En todo caso, el árbol de búsqueda puede ser enorme.

# Ordenación topológica

Dado un grafo $G = (V, A)$ orientado y acíclico, un orden topológico es un orden lineal $<$ definido sobre $V$ en el que $i < j$ si $(i, j) \in A$. Para toda arista, el vértice origen precede al destino en el orden topológico.
El algoritmo de ordenación topológica es un ejemplo de exploración exhaustiva de un grafo.
Este algoritmo recibe $G$ y devuelve una secuencia con los elementos de $V$ en orden topológico.
A partir de una secuencia vacía, y mediante una búsqueda en profundidad, se va insertando cada vértice por el principio conforme se terminan de recorrer recursivamente sus adyacentes.
Esto permite asegurar que cada vértice $i$ preceda en la secuencia a cualquier $j$ para el que $(i, j) \in A$.

# Ordenación topológica

$G$ puede representarse mediante listas de adyacencia con un vector $l$ de $n$ listas. La secuencia puede representarse mediante un vector $v$ de $n$ elementos que se rellena en orden inverso.

*ordenación-topológica*$(l, n) \to v$
desde $i \leftarrow 1$ hasta $n$
    $m[i] \leftarrow \bot$
$k \leftarrow n$
desde $i \leftarrow 1$ hasta $n$
    si $\neg m[i]$
        *profundidad*$(l, i, m, v, k)$

*profundidad*$(l, i, m, v, k) \to (m, v, k)$
$m[i] \leftarrow \top$
para todo $j \in l[i]$
    si $\neg m[j]$
        *profundidad*$(l, j, m, v, k)$
$v[k] \leftarrow i$
$k \leftarrow k - 1$

# Resolución de un laberinto

Un laberinto puede representarse mediante una matriz bidimensional, $I$.
Originalmente, cada celda está libre (' ') o bloqueada ('X').
El objetivo es comprobar si desde una posición inicial $(i, j)$ se puede llegar a una salida. Dicha posición inicial debe ser válida y corresponder a una celda libre.
A este efecto, se considera que una salida es cualquier celda libre del borde.
A la matriz se puede asociar un grafo implícito que podemos recorrer mediante una búsqueda en profundidad desde el vértice correspondiente a la posición inicial.
No es necesario construir dicho grafo, pues podemos trabajar directamente sobre la matriz mediante un marcado de sus celdas.
Durante la búsqueda de una salida se marcan las celdas visitadas ('V'). Si a partir de una celda visitada es posible encontrar una salida, se marca dicha celda como perteneciente a la solución ('S').

# Resolución de un laberinto

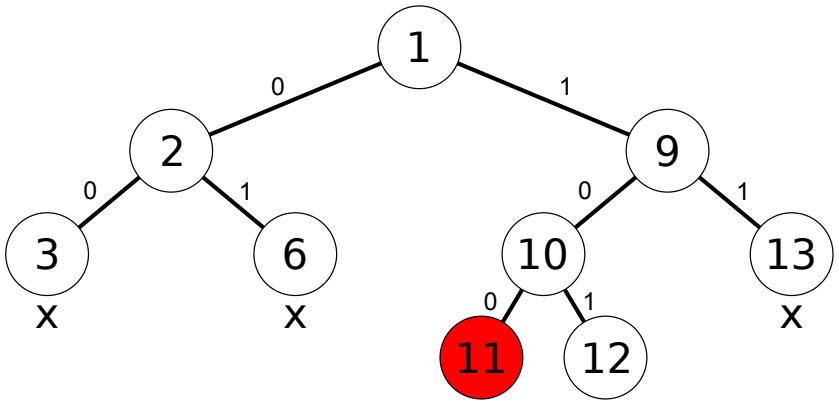
*solución-laberinto*$(l, m, n, i, j) \rightarrow (l, s)$
si $l[i,j] =$ ' '
  si *borde*$(m, n, i, j)$
    $s \leftarrow \top$
  si no
    $l[i,j] \leftarrow$ 'V'
    $s \leftarrow$ *solución-laberinto*$(l, m, n, i-1, j) \vee$
      *solución-laberinto*$(l, m, n, i+1, j) \vee$
      *solución-laberinto*$(l, m, n, i, j-1) \vee$
      *solución-laberinto*$(l, m, n, i, j+1)$
  si $s$
    $l[i,j] \leftarrow$ 'S'
si no
  $s \leftarrow \bot$

$$borde(m, n, i, j) \equiv i \in \{1, m\} \vee j \in \{1, n\}$$

# Búsqueda con retroceso

La búsqueda con retroceso consiste, básicamente, en una búsqueda en profundidad sobre un grafo implícito comenzando por un vértice prefijado.
Esta búsqueda se realiza recursivamente, generando partes del grafo conforme se progresa y evitando visitar un mismo vértice dos veces. Por lo tanto, el recorrido genera un árbol de búsqueda.
El camino desde la raíz del árbol de búsqueda al vértice actual representa una solución parcial, incompleta, al problema que se desea resolver.
Si al visitar un determinado vértice se detecta que la solución parcial no puede seguir completándose, se retrocede hasta un nivel que tenga vértices sin explorar y se prosigue la búsqueda.
Si esto no es posible, el problema no tiene solución.

# Búsqueda con retroceso




Solución

| |
|---|
| 1 |
| 0 |
| 0 |

## Idea general y poda del árbol de búsqueda

- Se toma una decisión en cada nivel y se refleja en la solución parcial.
- Se comprueba si la solución parcial es factible.
- Si es así, se ha de completar hasta formar una solución final.
- Si no es así, el vértice correspondiente a la decisión no se expande.
- En tal caso, se poda el subárbol formado por todos sus descendientes.
- La poda reduce el tamaño del árbol de búsqueda.

# Esquema general de la búsqueda con retroceso

```
vuelta-atrás(s, k) → s
prepara-nivel(k)
mientras ¬último-hijo-nivel(k)
        s[k] ← siguiente-hijo-nivel(k)
        si solución(s, k)
           procesa(s, k)
        si no
           si completable(s, k)
              vuelta-atrás(s, k + 1)
```

Hay que tomar una decisión en cada nivel $k$ del árbol de búsqueda. El algoritmo comienza su ejecución con $k = 1$.

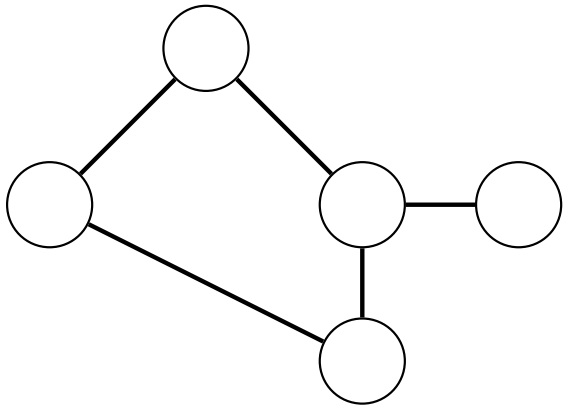
## Elementos

- $s$: vector con las decisiones tomadas en los niveles $1, \ldots, k$.
- *solución*$(s, k)$: ¿es $s$ una solución completa?
- *completable*$(s, k)$: ¿puede ser $s$ parte de una solución completa?
- *procesa*$(s, k)$: procesa la solución encontrada

# Coloreado de grafos

## Problema

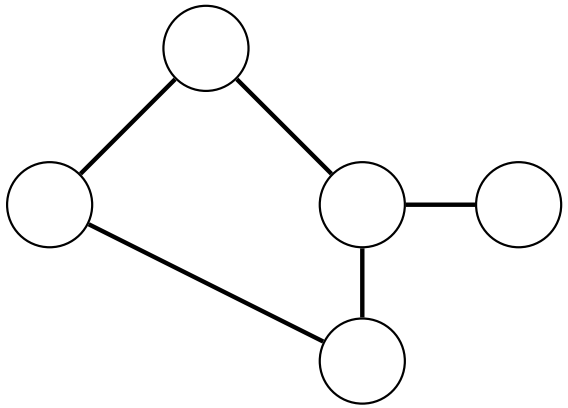
- ¿Se puede colorear un grafo con $m$ colores de forma que dos vértices adyacentes no posean el mismo color?
- ¿Para qué valores de $m$ es esto posible?



# Coloreado de grafos

## Problema

- ¿Se puede colorear un grafo con $m$ colores de forma que dos vértices adyacentes no posean el mismo color?
- ¿Para qué valores de $m$ es esto posible?

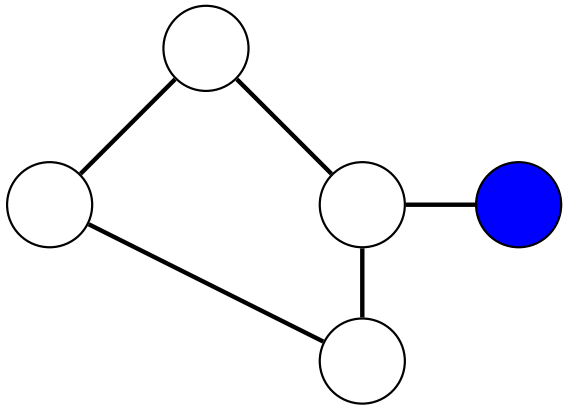


El orden en que se recorren los vértices afecta al número de colores que se obtiene.

# Coloreado de grafos

## Problema

- ¿Se puede colorear un grafo con $m$ colores de forma que dos vértices adyacentes no posean el mismo color?
- ¿Para qué valores de $m$ es esto posible?

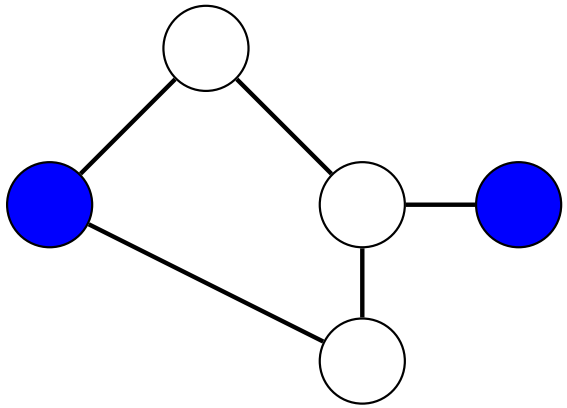


El orden en que se recorren los vértices afecta al número de colores que se obtiene.

# Coloreado de grafos

## Problema

- ¿Se puede colorear un grafo con $m$ colores de forma que dos vértices adyacentes no posean el mismo color?
- ¿Para qué valores de $m$ es esto posible?

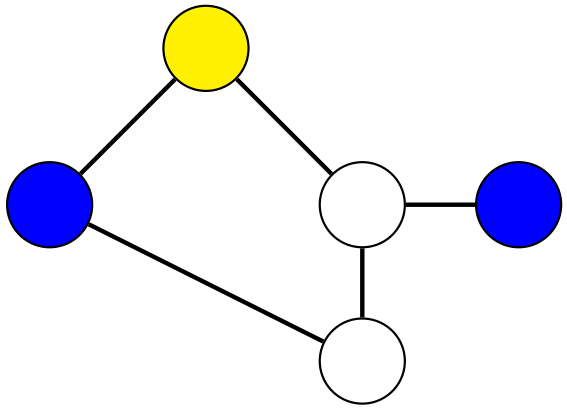


El orden en que se recorren los vértices afecta al número de colores que se obtiene.

# Coloreado de grafos

## Problema

- ¿Se puede colorear un grafo con $m$ colores de forma que dos vértices adyacentes no posean el mismo color?
- ¿Para qué valores de $m$ es esto posible?

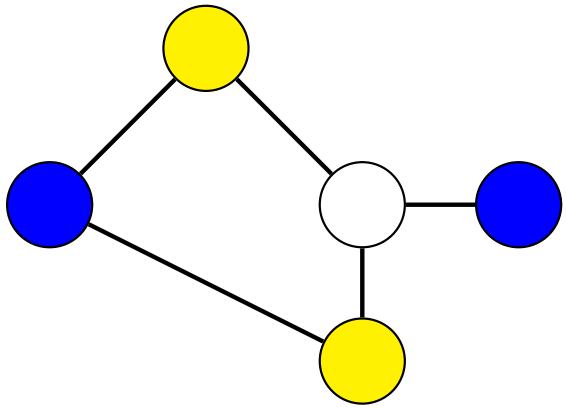


El orden en que se recorren los vértices afecta al número de colores que se obtiene.

# Coloreado de grafos

## Problema

- ¿Se puede colorear un grafo con $m$ colores de forma que dos vértices adyacentes no posean el mismo color?
- ¿Para qué valores de $m$ es esto posible?

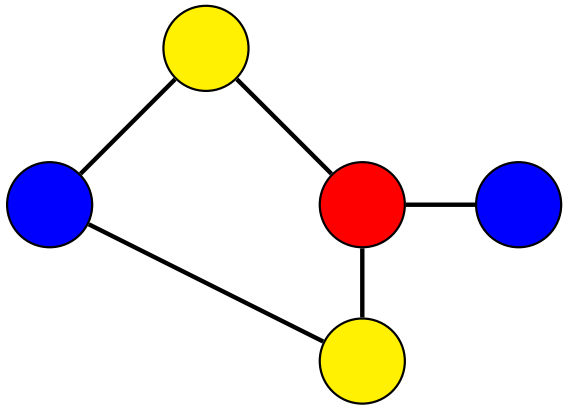


El orden en que se recorren los vértices afecta al número de colores que se obtiene.

# Coloreado de grafos

## Problema

- ¿Se puede colorear un grafo con $m$ colores de forma que dos vértices adyacentes no posean el mismo color?
- ¿Para qué valores de $m$ es esto posible?



El orden en que se recorren los vértices afecta al número de colores que se obtiene.

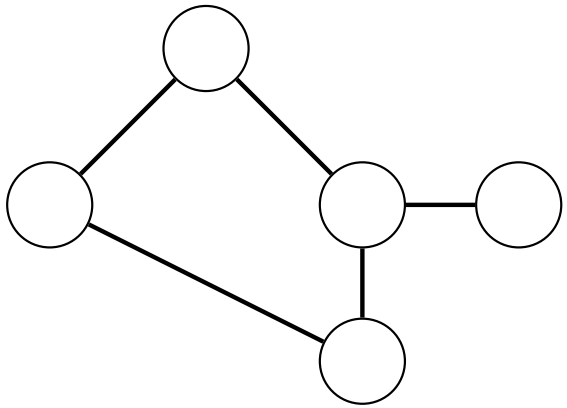
# Coloreado de grafos: búsqueda con retroceso

```
coloreado-grafo(l, n, m, s, k) → s
desde c ← 1 hasta m
        s[k] ← c
        si coloreable(l, s, k)
           si k = n
              imprimir(s, k)
           si no
              coloreado-grafo(l, n, m, s, k + 1)
```

```
coloreable(l, s, k) → r
r ← ⊤
j ← 1
mientras j < k ∧ r
                si k ∈ l[j] ∨ j ∈ l[k]
                   r ← s[j] ≠ s[k]
                j ← j + 1
```

# Coloreado de grafos: algoritmo devorador

Los vértices se seleccionan en orden decreciente de su grado.

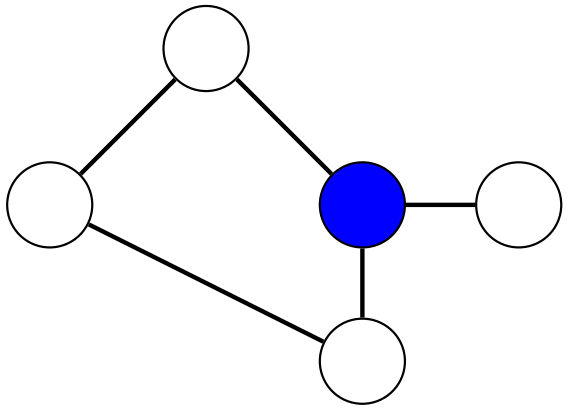


Aunque no existe una estrategia óptima que produzca un algoritmo devorador eficiente, en este ejemplo, se obtiene una solución óptima.

*coloreado-devorador*$(l, n) \to (s, m)$
desde $i \leftarrow 1$ hasta $n$
    $v[i] \leftarrow i$
    $s[i] \leftarrow 0$
*ordena-vértices*$(v, l, n)$
$m \leftarrow 1$
$C \leftarrow \{m\}$
desde $i \leftarrow 1$ hasta $n$
    $D \leftarrow C$
    para todo $j \in l[v[i]]$
        $D \leftarrow D - \{s[j]\}$
    si $D = \varnothing$
        $m \leftarrow m + 1$
        $C \leftarrow C \cup \{m\}$
        $D \leftarrow \{m\}$
    $s[v[i]] \leftarrow$ mín $D$

# Coloreado de grafos: algoritmo devorador

Los vértices se seleccionan en orden decreciente de su grado.

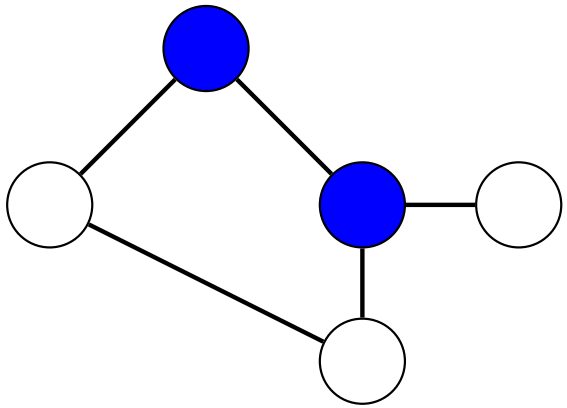


Aunque no existe una estrategia óptima que produzca un algoritmo devorador eficiente, en este ejemplo, se obtiene una solución óptima.

*coloreado-devorador*$(l, n) \to (s, m)$
desde $i \leftarrow 1$ hasta $n$
    $v[i] \leftarrow i$
    $s[i] \leftarrow 0$
*ordena-vértices*$(v, l, n)$
$m \leftarrow 1$
$C \leftarrow \{m\}$
desde $i \leftarrow 1$ hasta $n$
    $D \leftarrow C$
    para todo $j \in l[v[i]]$
        $D \leftarrow D - \{s[j]\}$
    si $D = \varnothing$
        $m \leftarrow m + 1$
        $C \leftarrow C \cup \{m\}$
        $D \leftarrow \{m\}$
    $s[v[i]] \leftarrow$ mín $D$

# Coloreado de grafos: algoritmo devorador

Los vértices se seleccionan en orden decreciente de su grado.

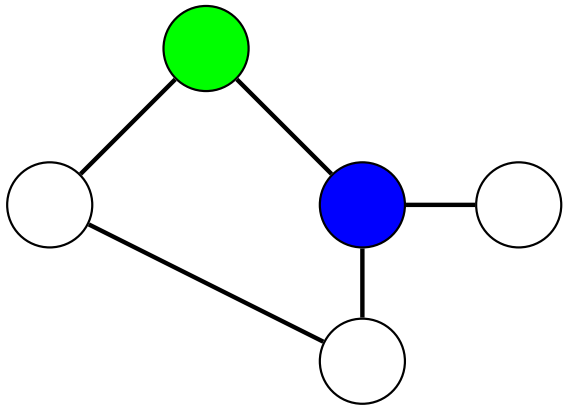


Aunque no existe una estrategia óptima que produzca un algoritmo devorador eficiente, en este ejemplo, se obtiene una solución óptima.

*coloreado-devorador*$(l, n) \to (s, m)$
desde $i \leftarrow 1$ hasta $n$
    $v[i] \leftarrow i$
    $s[i] \leftarrow 0$
*ordena-vértices*$(v, l, n)$
$m \leftarrow 1$
$C \leftarrow \{m\}$
desde $i \leftarrow 1$ hasta $n$
    $D \leftarrow C$
    para todo $j \in l[v[i]]$
        $D \leftarrow D - \{s[j]\}$
    si $D = \varnothing$
        $m \leftarrow m + 1$
        $C \leftarrow C \cup \{m\}$
        $D \leftarrow \{m\}$
    $s[v[i]] \leftarrow$ mín $D$

# Coloreado de grafos: algoritmo devorador

Los vértices se seleccionan en orden decreciente de su grado.

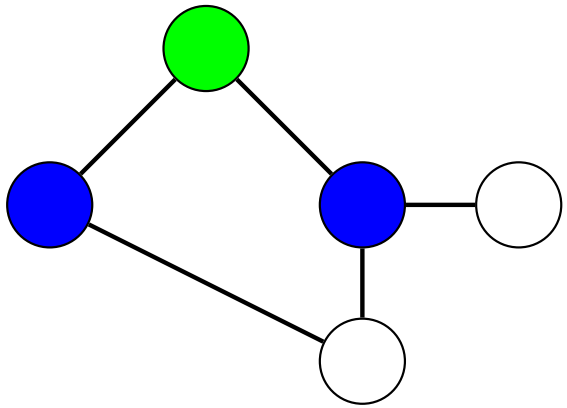


Aunque no existe una estrategia óptima que produzca un algoritmo devorador eficiente, en este ejemplo, se obtiene una solución óptima.

*coloreado-devorador*$(l, n) \to (s, m)$
desde $i \leftarrow 1$ hasta $n$
    $v[i] \leftarrow i$
    $s[i] \leftarrow 0$
*ordena-vértices*$(v, l, n)$
$m \leftarrow 1$
$C \leftarrow \{m\}$
desde $i \leftarrow 1$ hasta $n$
    $D \leftarrow C$
    para todo $j \in l[v[i]]$
        $D \leftarrow D - \{s[j]\}$
    si $D = \varnothing$
        $m \leftarrow m + 1$
        $C \leftarrow C \cup \{m\}$
        $D \leftarrow \{m\}$
    $s[v[i]] \leftarrow$ mín $D$

# Coloreado de grafos: algoritmo devorador

Los vértices se seleccionan en orden decreciente de su grado.

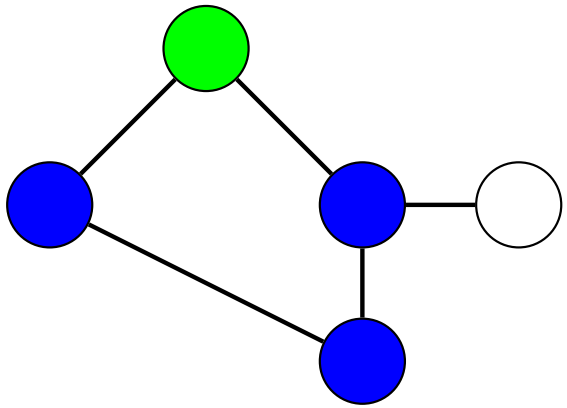


Aunque no existe una estrategia óptima que produzca un algoritmo devorador eficiente, en este ejemplo, se obtiene una solución óptima.

*coloreado-devorador*$(l, n) \to (s, m)$
desde $i \leftarrow 1$ hasta $n$
    $v[i] \leftarrow i$
    $s[i] \leftarrow 0$
*ordena-vértices*$(v, l, n)$
$m \leftarrow 1$
$C \leftarrow \{m\}$
desde $i \leftarrow 1$ hasta $n$
    $D \leftarrow C$
    para todo $j \in l[v[i]]$
        $D \leftarrow D - \{s[j]\}$
    si $D = \varnothing$
        $m \leftarrow m + 1$
        $C \leftarrow C \cup \{m\}$
        $D \leftarrow \{m\}$
    $s[v[i]] \leftarrow$ mín $D$

# Coloreado de grafos: algoritmo devorador

Los vértices se seleccionan en orden decreciente de su grado.

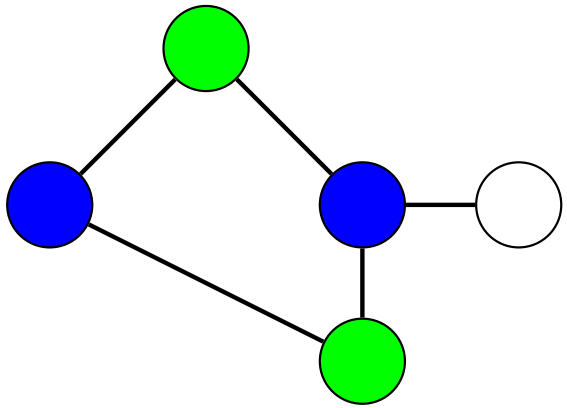


Aunque no existe una estrategia óptima que produzca un algoritmo devorador eficiente, en este ejemplo, se obtiene una solución óptima.

*coloreado-devorador*$(l, n) \to (s, m)$
desde $i \leftarrow 1$ hasta $n$
    $v[i] \leftarrow i$
    $s[i] \leftarrow 0$
*ordena-vértices*$(v, l, n)$
$m \leftarrow 1$
$C \leftarrow \{m\}$
desde $i \leftarrow 1$ hasta $n$
    $D \leftarrow C$
    para todo $j \in l[v[i]]$
        $D \leftarrow D - \{s[j]\}$
    si $D = \varnothing$
        $m \leftarrow m + 1$
        $C \leftarrow C \cup \{m\}$
        $D \leftarrow \{m\}$
    $s[v[i]] \leftarrow$ mín $D$

# Coloreado de grafos: algoritmo devorador

Los vértices se seleccionan en orden decreciente de su grado.

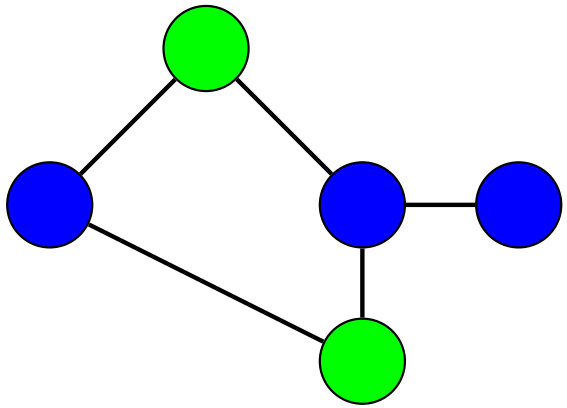


Aunque no existe una estrategia óptima que produzca un algoritmo devorador eficiente, en este ejemplo, se obtiene una solución óptima.

*coloreado-devorador*$(l, n) \to (s, m)$
desde $i \leftarrow 1$ hasta $n$
    $v[i] \leftarrow i$
    $s[i] \leftarrow 0$
*ordena-vértices*$(v, l, n)$
$m \leftarrow 1$
$C \leftarrow \{m\}$
desde $i \leftarrow 1$ hasta $n$
    $D \leftarrow C$
    para todo $j \in l[v[i]]$
        $D \leftarrow D - \{s[j]\}$
    si $D = \varnothing$
        $m \leftarrow m + 1$
        $C \leftarrow C \cup \{m\}$
        $D \leftarrow \{m\}$
    $s[v[i]] \leftarrow$ mín $D$

# Coloreado de grafos: algoritmo devorador

Los vértices se seleccionan en orden decreciente de su grado.

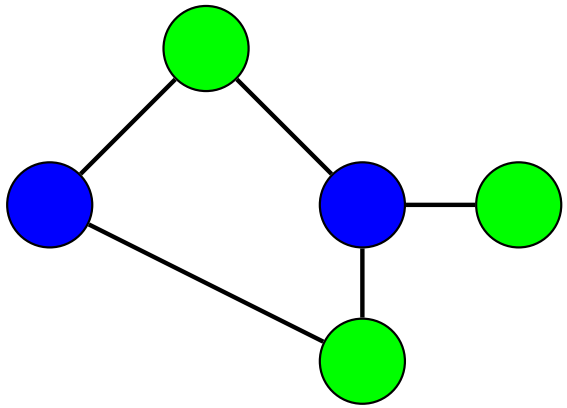


Aunque no existe una estrategia óptima que produzca un algoritmo devorador eficiente, en este ejemplo, se obtiene una solución óptima.

*coloreado-devorador*$(l, n) \to (s, m)$
desde $i \leftarrow 1$ hasta $n$
    $v[i] \leftarrow i$
    $s[i] \leftarrow 0$
*ordena-vértices*$(v, l, n)$
$m \leftarrow 1$
$C \leftarrow \{m\}$
desde $i \leftarrow 1$ hasta $n$
    $D \leftarrow C$
    para todo $j \in l[v[i]]$
        $D \leftarrow D - \{s[j]\}$
    si $D = \varnothing$
        $m \leftarrow m + 1$
        $C \leftarrow C \cup \{m\}$
        $D \leftarrow \{m\}$
    $s[v[i]] \leftarrow$ mín $D$

# Coloreado de grafos: algoritmo devorador

Los vértices se seleccionan en orden decreciente de su grado.



Aunque no existe una estrategia óptima que produzca un algoritmo devorador eficiente, en este ejemplo, se obtiene una solución óptima.

*coloreado-devorador*$(l, n) \to (s, m)$
desde $i \leftarrow 1$ hasta $n$
    $v[i] \leftarrow i$
    $s[i] \leftarrow 0$
*ordena-vértices*$(v, l, n)$
$m \leftarrow 1$
$C \leftarrow \{m\}$
desde $i \leftarrow 1$ hasta $n$
    $D \leftarrow C$
    para todo $j \in l[v[i]]$
        $D \leftarrow D - \{s[j]\}$
    si $D = \varnothing$
        $m \leftarrow m + 1$
        $C \leftarrow C \cup \{m\}$
        $D \leftarrow \{m\}$
    $s[v[i]] \leftarrow$ mín $D$

# Coloreado de grafos: algunas curiosidades

## Grafos planos

- Un grafo plano es el que se puede dibujar sin cruces de aristas.
- La mayoría de los mapas políticos son grafos planos.
- Caracterizaciones formales
  - Teorema de Kuratoswki.
  - Teorema de Wagner.
  - Criterio de Fraysseix–Rosenstiehl.
- Existen algoritmos para comprobar si un grafo es plano en $O(n)$.

## Resultados conocidos

- Todo grafo plano es 4-coloreable en $O(n^2)$.
- Todo grafo plano es 5-coloreable en $O(n)$.
- Si el grafo no es plano, el problema es computacionalmente difícil.

# Mochila con pesos reales

## Principales versiones

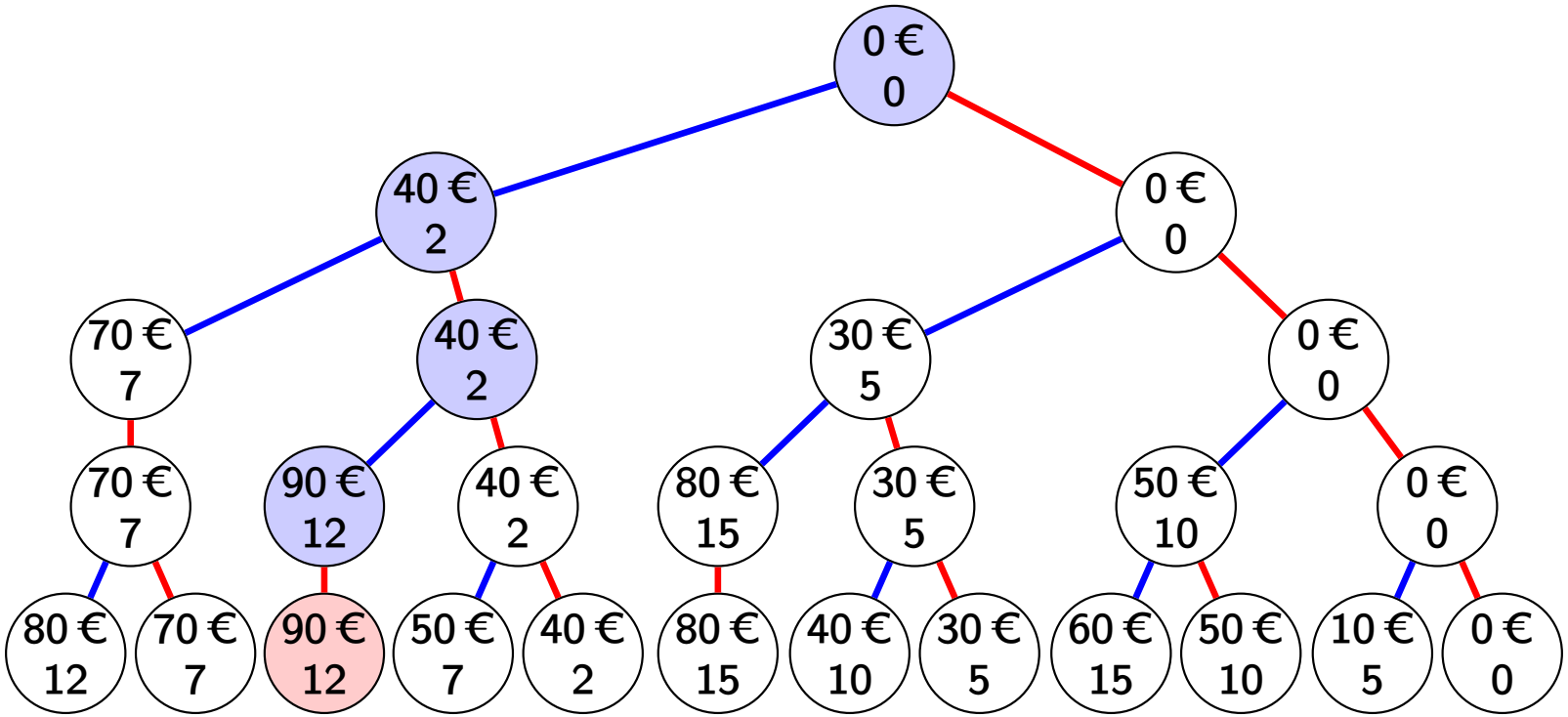
- Mochila continua: algoritmo devorador
- Mochila discreta con pesos enteros: programación dinámica

## Cómo resolver la versión discreta

- Algoritmos devoradores: ordenar por $v/p$ no es óptimo.
- Programación dinámica: existen algoritmos, pero son más complejos
- Búsqueda con retroceso: esquema general, guardando la mejor solución encontrada hasta ahora.

# Versión 1: poda por capacidad de la mochila (I)

- Objetos: {(40 €, 2), (30 €, 5), (50 €, 10), (10 €, 5)}
- Capacidad de la mochila $c = 16$
- No nos introducimos en vértices con peso total superior a $c$



# Versión 1: poda por capacidad de la mochila (II)

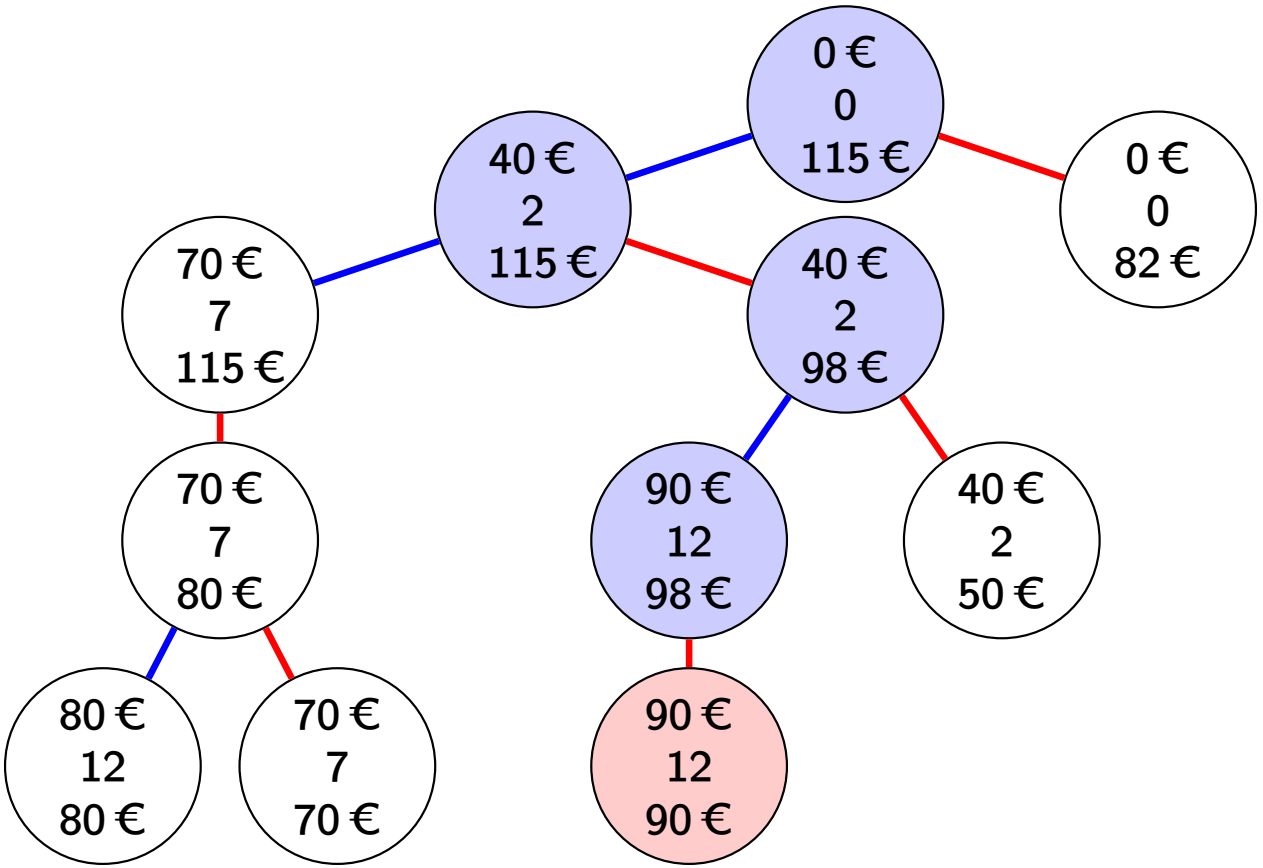
$mdr(v, p, n, c) \rightarrow S$
desde $i \leftarrow 1$ hasta $n$
    $S[i] \leftarrow \perp$
$M \leftarrow S$
$S \leftarrow mdr\text{-}aux(v, p, n, c, S, 1, M)$

# Versión 1: poda por capacidad de la mochila (III)

$mdr\text{-}aux(v, p, n, c, S, k, M) \rightarrow M$
si $p[k] + peso(S) \leq c$
  $S[k] \leftarrow \top$
  si $k = n$
    si $valor(S) > valor(M)$
      $M \leftarrow S$
  si no
    $M \leftarrow mdr\text{-}aux(v, p, n, c, S, k + 1, M)$
$S[k] \leftarrow \bot$
si $k = n$
  si $valor(S) > valor(M)$
    $M \leftarrow S$
si no
  $M \leftarrow mdr\text{-}aux(v, p, n, c, S, k + 1, M)$

# Versión 2: acotación superior (I)

Se puede usar el algoritmo devorador para dar una cota superior del valor obtenible y no seguir por vértices menos prometedores.



# Versión 2: acotación superior (II)

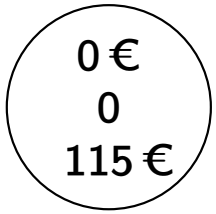
$mdr\text{-}aux(v, p, n, c, S, k, M) \rightarrow M$
si $p[k] + peso(S) \leq c$
  $S[k] \leftarrow \top$
  si $k = n$
    $M \leftarrow S$
  si no
    $M \leftarrow mdr\text{-}aux(v, p, n, c, S, k + 1, M)$
$S[k] \leftarrow \bot$
si $cota(v, p, n, c, S, k + 1) > valor(M)$
  si $k = n$
    $M \leftarrow S$
  si no
    $M \leftarrow mdr\text{-}aux(v, p, n, c, S, k + 1, M)$

# Versión 2: acotación superior (III)

$cota(v, p, n, c, S, k) \rightarrow m$
$m \leftarrow valor(S)$
$q \leftarrow peso(S)$
mientras $k \leq n \wedge q + p[k] \leq c$
$m \leftarrow m + v[k]$
$q \leftarrow q + p[k]$
$k \leftarrow k + 1$
si $k \leq n \wedge q < c$
$m \leftarrow m + v[k] \cdot (c - q)/p[k]$

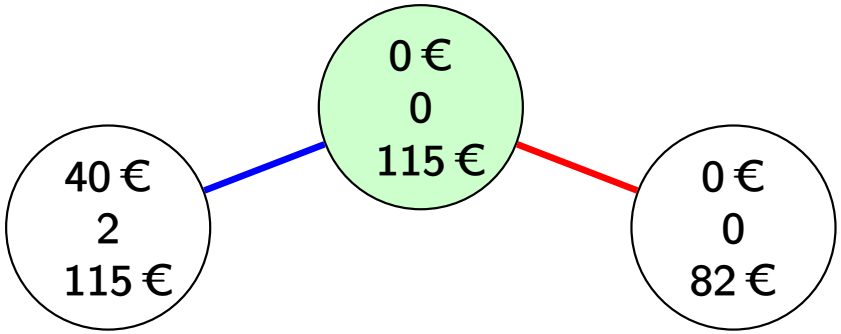
# Versión 3: expansión ordenada por cota superior

- A cada paso elegimos el siguiente vértice con la mayor cota superior que sea mayor al valor máximo hasta ahora
- En vez de una búsqueda en profundidad, los vértices sin expandir se añaden a un montículo



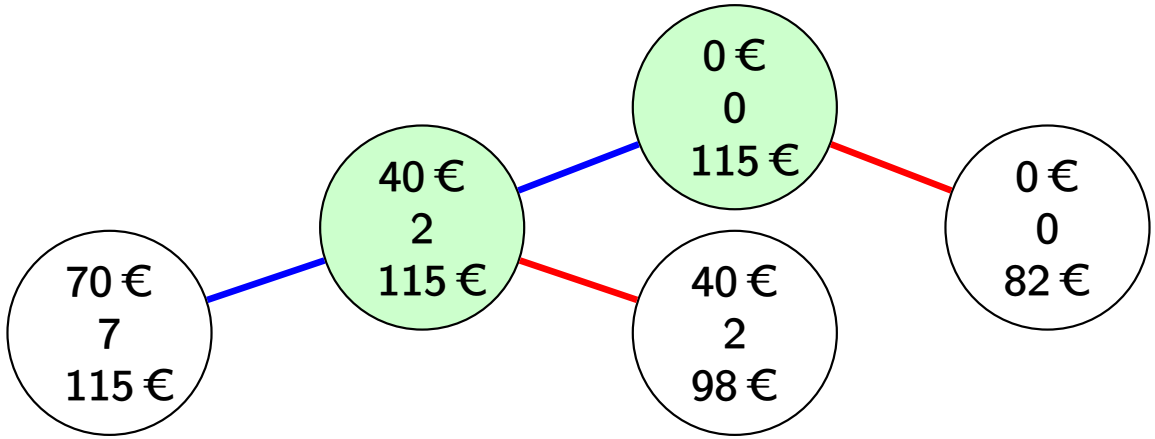
# Versión 3: expansión ordenada por cota superior

- A cada paso elegimos el siguiente vértice con la mayor cota superior que sea mayor al valor máximo hasta ahora
- En vez de una búsqueda en profundidad, los vértices sin expandir se añaden a un montículo



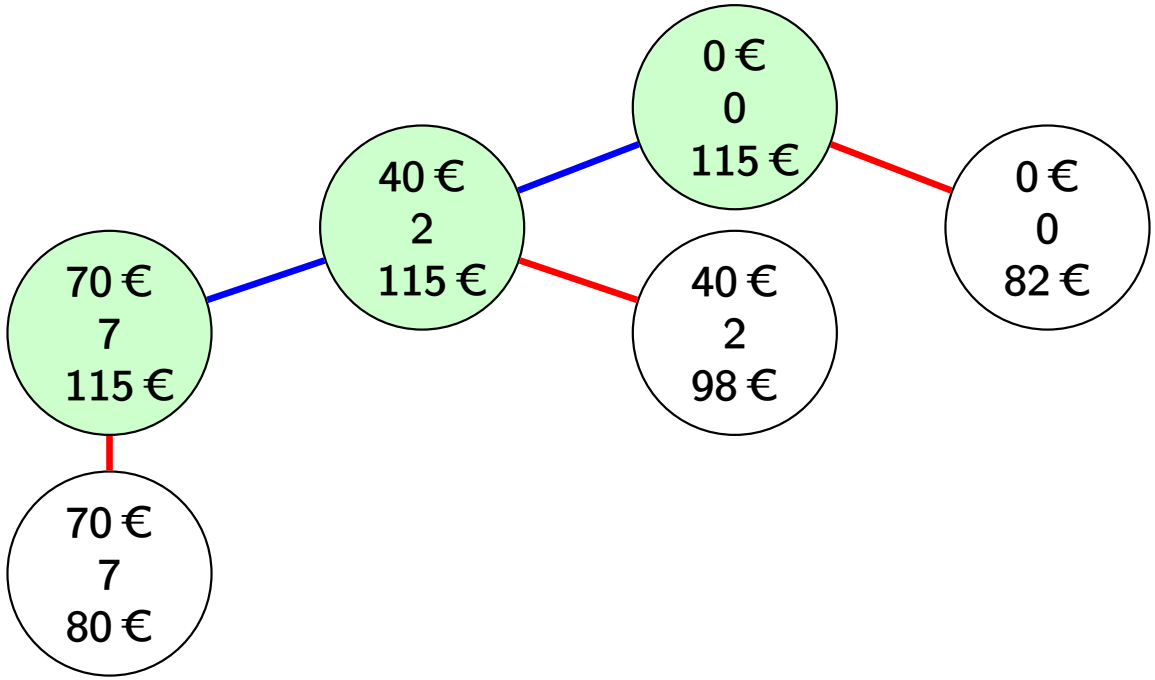
# Versión 3: expansión ordenada por cota superior

- A cada paso elegimos el siguiente vértice con la mayor cota superior que sea mayor al valor máximo hasta ahora
- En vez de una búsqueda en profundidad, los vértices sin expandir se añaden a un montículo



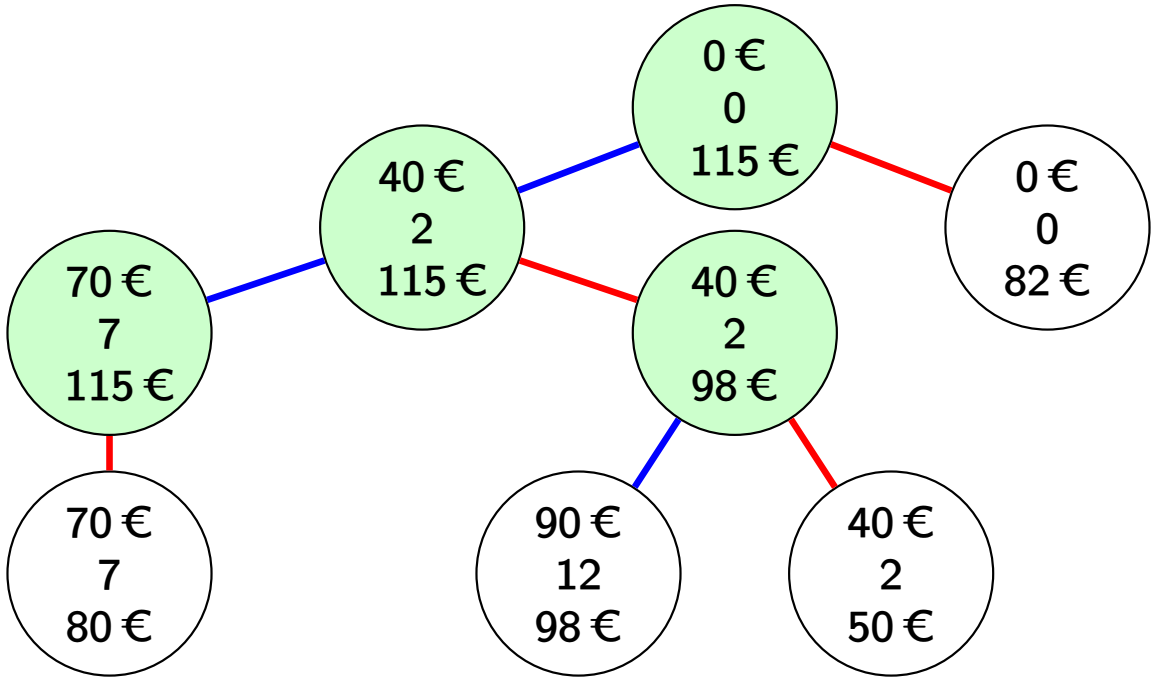
# Versión 3: expansión ordenada por cota superior

- A cada paso elegimos el siguiente vértice con la mayor cota superior que sea mayor al valor máximo hasta ahora
- En vez de una búsqueda en profundidad, los vértices sin expandir se añaden a un montículo



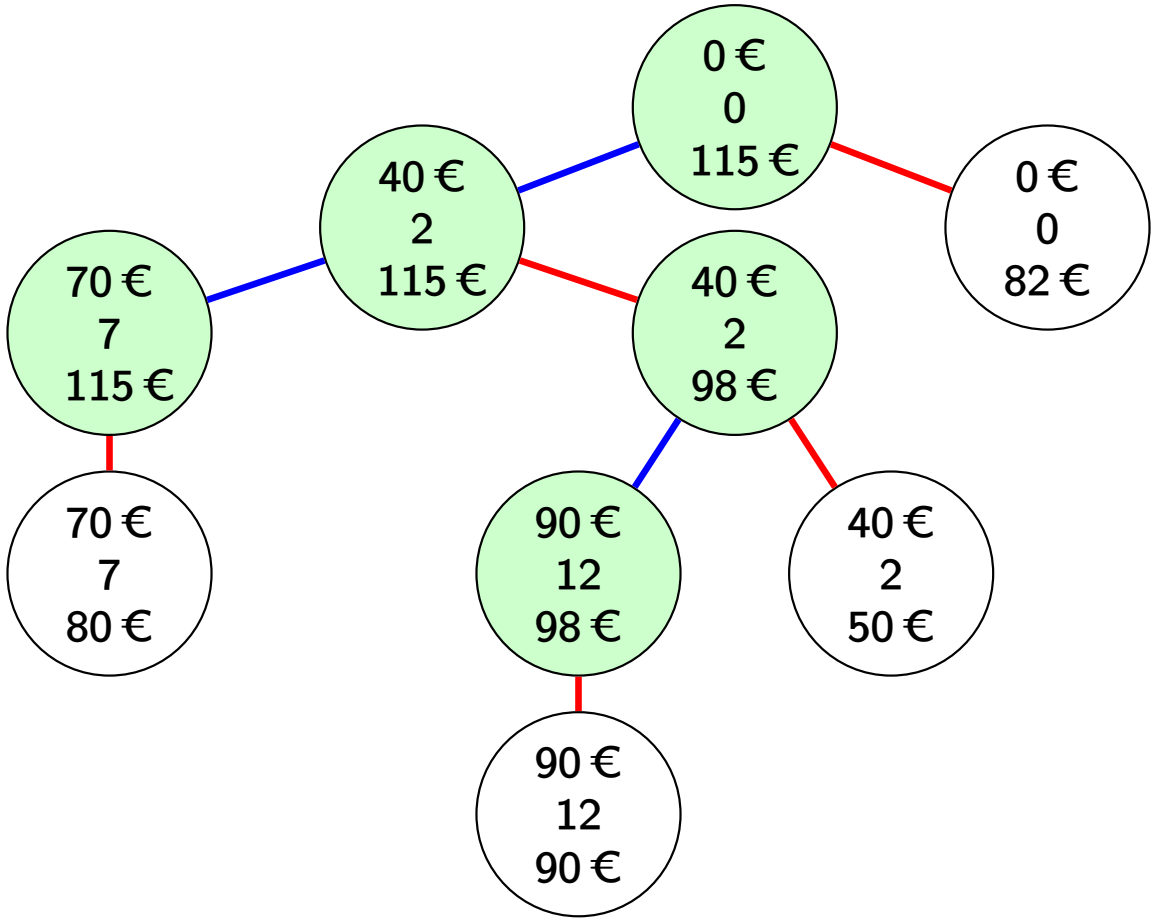
# Versión 3: expansión ordenada por cota superior

- A cada paso elegimos el siguiente vértice con la mayor cota superior que sea mayor al valor máximo hasta ahora
- En vez de una búsqueda en profundidad, los vértices sin expandir se añaden a un montículo



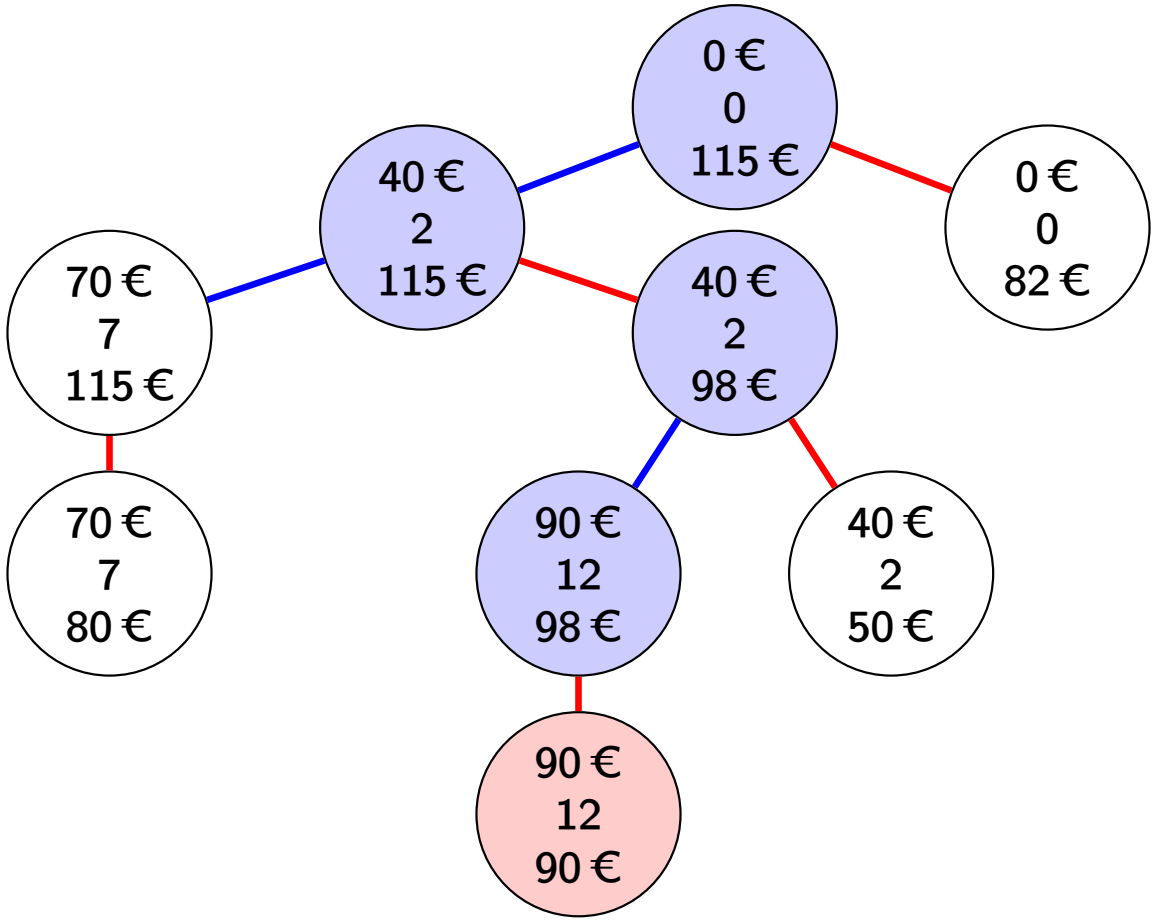
# Versión 3: expansión ordenada por cota superior

- A cada paso elegimos el siguiente vértice con la mayor cota superior que sea mayor al valor máximo hasta ahora
- En vez de una búsqueda en profundidad, los vértices sin expandir se añaden a un montículo



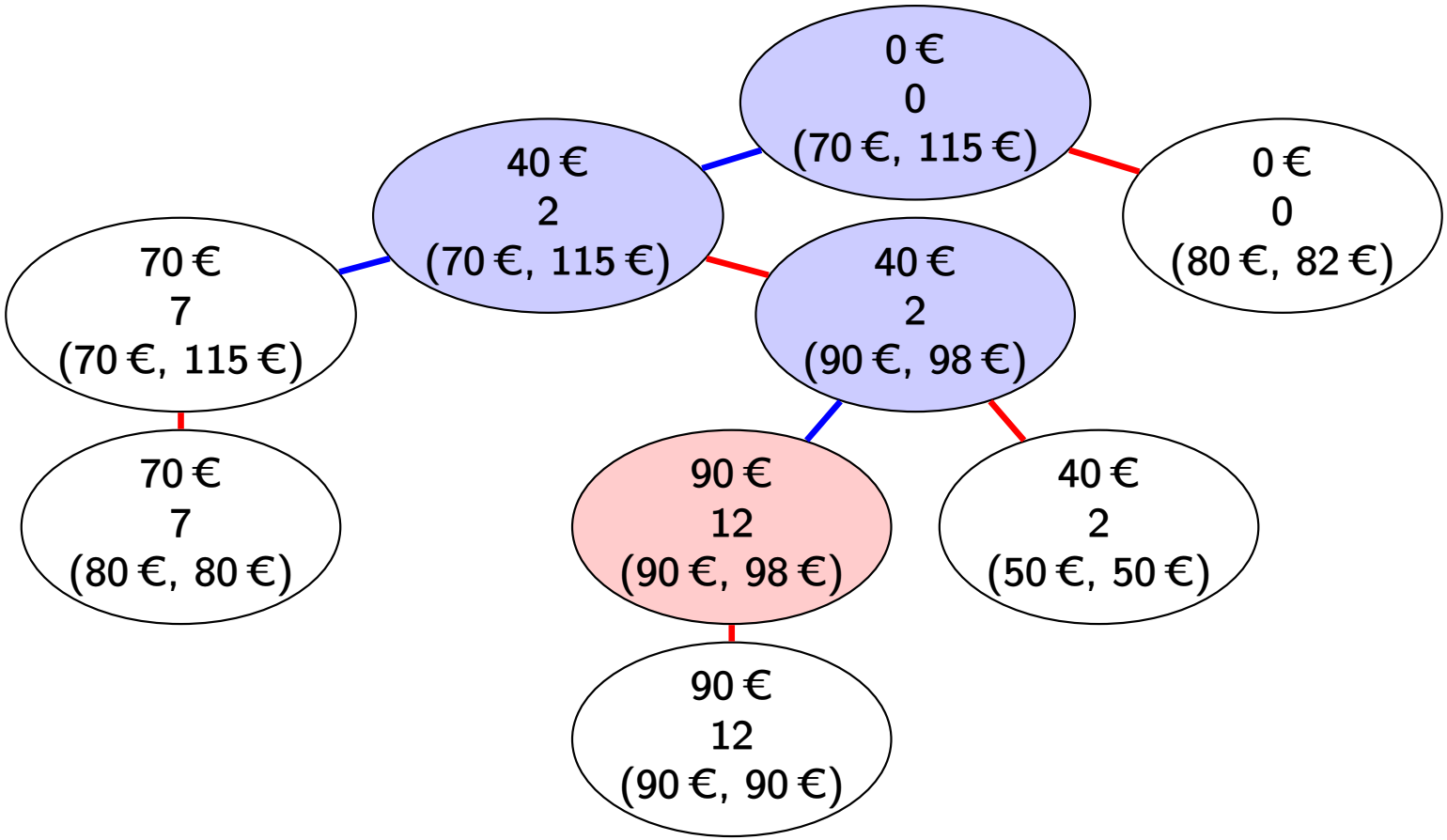
# Versión 3: expansión ordenada por cota superior

- A cada paso elegimos el siguiente vértice con la mayor cota superior que sea mayor al valor máximo hasta ahora
- En vez de una búsqueda en profundidad, los vértices sin expandir se añaden a un montículo



# Versión 4: uso de cota inferior para acelerar poda

- La poda no comienza hasta encontrar una buena solución
- Se puede acelerar estimando cotas inferiores del valor con el algoritmo devorador en el caso discreto



# Introducción

Se desea encontrar el camino más corto entre dos vértices de un grafo $G = (V, A)$.
El algoritmo de Dijkstra calcula el camino más corto desde un vértice a todos los demás en $O(n^2)$.

## Ejemplo

Búsqueda de caminos en un videojuego RTS:

- Mapa compuesto por $100 \times 100$ celdas.
- $n = |V| = 10\,000$ vértices.
- Los caminos son dinámicos
  - Terreno modificable
  - Bloqueo de unidades
- El tiempo del algoritmo puede ser considerable.

# Algoritmo A*

Es un algoritmo de búsqueda por el mejor con una función de evaluación $f$, definida para todo nodo $n$, en la que $f(n) = g(n) + h(n)$ y:

- $g(n)$ es el coste real del camino desde el nodo inicial a $n$.
- $h(n)$ es el coste estimado del camino óptimo desde $n$ a un nodo final.

donde $g(n), h(n) \geq 0$ y, para todo nodo final, $h(n) = 0$.
El objetivo es encontrar una solución, un camino desde el nodo inicial a un nodo final, a ser posible, óptimo.
La función de evaluación dirige la búsqueda: el siguiente nodo a explorar es aquel con menor $f$. En caso de empate, puede elegirse el de menor $h$, siempre dando prioridad a los nodos finales.
La función $h$ es una heurística, pues no suele conocerse el coste óptimo real $h^*$ de alcanzar un objetivo, y su valor en cada nodo permanece inalterado.
El algoritmo A* es completo. Siempre encuentra una solución si existe.
Cuanto menor es $h^* - h$, más se explora en profundidad y antes se obtiene una solución, aunque quizás no la mejor. Sin embargo, si $h$ cumple ciertas condiciones, también es óptima.

# Heurística y optimalidad en A*

## Definición (Heurística admisible)

Una función heurística $h$ es admisible, u optimista, si siempre se cumple que $h(n) \leq h^*(n)$ para todo nodo $n$.

El algoritmo A* garantiza soluciones óptimas cuando la heurística es admisible. Cuando la optimalidad no es clave, una heurística pesimista que sobrestime ocasionalmente $h^*$ puede ser bastante más eficiente.

## Definición (Heurística consistente)

Una función heurística $h$ es consistente, o monótona, si para todo nodo $n$ y sucesor inmediato $s$ de $n$, $h(n) \leq c(n,s) + h(s)$, donde $c(n,s)$ es el coste de alcanzar $s$ desde $n$.

Una heurística consistente es siempre admisible. Las heurísticas admisibles no siempre son consistentes. Por ejemplo, $h(n) = 0$ es admisible para costes $c(n,s)$ no negativos, pues es consistente: en tal caso, A* se comporta como el algoritmo de Dijkstra.

# Esquema general del algoritmo A*

$abierta \leftarrow \{inicial\}$
$cerrada \leftarrow \varnothing$
$fin \leftarrow \bot$
mientras $\neg fin \wedge abierta \neq \varnothing$
    $actual \leftarrow extrae\text{-}mejor(abierta)$
    $cerrada \leftarrow cerrada \cup \{actual\}$
    si $es\text{-}objetivo(actual)$
        $fin \leftarrow \top$
    si no
        $hijos \leftarrow sucesores(actual)$
        $actualiza(hijos, abierta, cerrada)$
        $abierta \leftarrow abierta \cup hijos$

**Nodos**

- *inicial*: nodo inicial
- *actual*: nodo actual
- *hijos*: sucesores del nodo actual
- *abierta*: nodos por expandir
- *cerrada*: nodos expandidos

*extrae-mejor* extrae de *abierta* el nodo con mejor evaluación.
*actualiza* actualiza el coste de los sucesores de *actual* y modifica *hijos*, *abierta* y *cerrada* en consecuencia. Si se emplea una heurística consistente, no es necesario modificar *cerrada*.

# Algoritmo A*

$A^*(inicial, final) \rightarrow camino$
$(abierta, cerrada) \leftarrow (\langle inicial, g(inicial), h(inicial)\rangle, \varnothing)$
$camino \leftarrow \langle\rangle$
mientras $camino = \langle\rangle \wedge abierta \neq \varnothing$
    $k \leftarrow extrae\text{-}mejor(abierta)$
    $cerrada \leftarrow cerrada \cup \{k\}$
    si $k = final$
        $camino \leftarrow recupera\text{-}camino(inicial, final, predecesor)$
    si no
        para todo $j \in sucesores(k)$
            $c \leftarrow g(k) + coste(k, j)$
            si $(j \in abierta \vee j \in cerrada) \wedge c < g(j)$
                $elimina(j, abierta)$
                $elimina(j, cerrada)$
            si $j \notin abierta \wedge j \notin cerrada$
                $abierta \leftarrow abierta \cup \{\langle j, c, h(j)\rangle\}$
                $predecesor[j] \leftarrow k$

# Algoritmo A*

- En *abierta* y *cerrada* almacenamos tuplas.
  - Cada tupla representa un nodo, su valor de $g$ y su valor de $h$.
  - La representación de los nodos depende del problema a resolver.
  - El valor de $f$ aparece desglosado en $g$ y $h$ por conveniencia.
- Los sucesores pueden ser de tres tipos:
  - Nuevos: no han aparecido antes.
  - Abiertos: han aparecido antes, pero están pendientes de explorar.
  - Cerrados: han aparecido antes y han sido explorados.
- Los nuevos se insertan en *abierta* (conviene una cola de prioridad).
- Los abiertos se actualizan si es necesario.
- Los cerrados se actualizan y se reabren si es necesario.
- Las actualizaciones se producen si la nueva $g$ es menor que la previa.
- El camino se recupera a partir de los predecesores.

$recupera\text{-}camino(inicial, final, predecesor) \rightarrow c$
$(c, k) \leftarrow (\langle\rangle, final)$
mientras $k \neq inicial$
$\quad (c, k) \leftarrow (\langle k \rangle + c, predecesor[k])$
$c \leftarrow \langle inicial \rangle + c$

# Referencias



Narciso Martí Oliet, Yolanda Ortega Mallén y José Alberto Verdejo López.
Estructuras de datos y métodos algorítmicos: ejercicios resueltos.
Prentice Hall, 2004.


Richard Neapolitan y Kumarss Naimipour.
Foundations of Algorithms.
4.ª edición, Jones and Bartlett Publishers, 2011.


Stuart J. Russell y Peter Norvig.
Artificial Intelligence. A Modern Approach.
3.ª edición, Prentice Hall, 2010.