

## Tema 4: Exploración en grafos

A. Salguero, F. Palomo, A. García, I. Medina

Universidad de Cádiz

Diseño de Algoritmos  
Curso 2016/17

# Introducción

Muchos problemas interesantes pueden resolverse mediante el empleo de grafos.

A veces se requiere una exploración exhaustiva de todos los vértices o aristas de un grafo para resolver un problema. Sin embargo, existen problemas donde esto no es necesario.

Si el grafo posee demasiados vértices o aristas, y en especial si es infinito, puede ser imposible construirlo explícitamente. 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. Esto permite construir partes de él a medida que se recorre.

# Introducción

Al construir únicamente las partes relevantes para la resolución del problema, es posible ahorrar tiempo y espacio.

En resumen, el concepto de *grafo* se emplea de dos maneras muy diferentes:

- 1 Como estructura de datos *explícita* almacenada en memoria.
- 2 Como grafo *implícito* que nunca llega a estar completo en memoria.

En general, los ciclos representan un problema para cualquier algoritmo de exploración de grafos. Sin embargo, en ocasiones, basta emplear grafos orientados acíclicos o incluso árboles.

Cuando esto no sea posible, pueden necesitarse técnicas específicas como el marcado de vértices o aristas.

# Ordenación topológica

Dado un grafo  $G = \langle V, A \rangle$  orientado y acíclico, un orden topológico es un orden lineal  $<$  definido sobre  $V$  tal que  $i < j \implies \langle j, i \rangle \notin A$ . 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 explorar recursivamente sus adyacentes.


Esto permite asegurar que cada vértice  $i$  preceda en la secuencia a cualquier  $j$  para el que  $\langle i, j \rangle \in A$ .

# Ordenación topológica

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

*ordenación-topológica* :  $I \times n \rightarrow v$

desde  $i \leftarrow 1$  hasta  $n$

$m[i] \leftarrow \perp$  

$k \leftarrow n$

desde  $i \leftarrow 1$  hasta  $n$

si  $\neg m[i]$

*profundidad*( $I, i, m, v, k$ )

*profundidad* :  $I \times i \times m \times v \times k \rightarrow m \times v \times k$

$m[i] \leftarrow \top$

para todo  $j \in I[i]$  

si  $\neg m[j]$

*profundidad*( $I, j, m, v, k$ )

$v[k] \leftarrow i$

$k \leftarrow k - 1$

# Búsqueda con retroceso

La búsqueda con retroceso consiste, básicamente, en una búsqueda primero en profundidad sobre un grafo implícito.

Esta búsqueda se realiza recursivamente, generando partes del grafo conforme se progresa. Normalmente, se exige que el grafo sea orientado y acíclico. De hecho, en la mayoría de las ocasiones, el grafo es un árbol.

Las soluciones del problema se obtienen extendiendo soluciones parciales de problemas más pequeños que se generan durante el recorrido.

Si se detecta que una solución parcial no se puede extender para formar una completa, se retrocede hasta un vértice que tenga vecinos sin explorar y se prosigue la búsqueda.

## Ejemplo: resolución de un laberinto

Un laberinto puede representarse mediante una matriz bidimensional. 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.

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').

# Ejemplo: resolución de un laberinto

*solución* :  $l \times m \times n \times i \times j \rightarrow l \times s$

si  $l[i, j] = ' '$

  si *borde*( $m, n, i, j$ )

$s \leftarrow \top$

  si no

$l[i, j] \leftarrow 'V'$

$s \leftarrow \text{solución}(l, m, n, i - 1, j) \vee$

$\text{solución}(l, m, n, i + 1, j) \vee$

$\text{solución}(l, m, n, i, j - 1) \vee$

$\text{solución}(l, m, n, i, j + 1)$

  si  $s$

$l[i, j] \leftarrow 'S'$

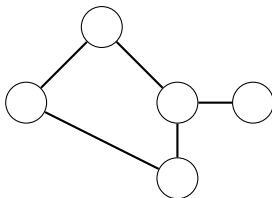
si no

$s \leftarrow \perp$

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



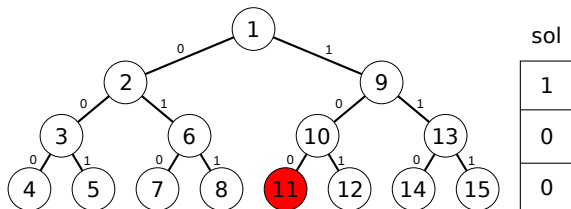
# Coloreado de grafos



## Problema

- ¿Se pueden colorear los nodos con  $m$  colores de forma que dos vértices adyacentes no tengan el mismo color?
- ¿Qué valores de  $m$  valdrían para el grafo de la izquierda?

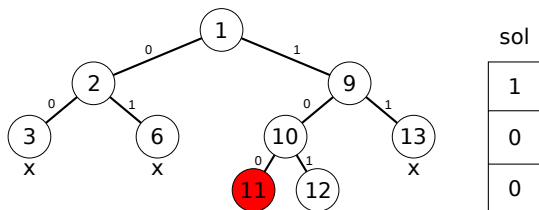
# Búsqueda de fuerza bruta por recorrido en profundidad



## Idea general

- Se recorre el espacio completo de decisiones, buscando una solución.
- Si es un problema de optimización, se va guardando la mejor solución obtenida hasta ahora.
- El árbol puede ser *muy* grande.

# Búsqueda con retroceso



## Mejora: poda del árbol

- A cada paso, se comprueba si la solución parcial podría ser parte de la solución final.
- Si no es así, el nodo no se *expande*: nos hemos ahorrado visitar sus descendientes.

# Esquema general de la búsqueda en retroceso

```

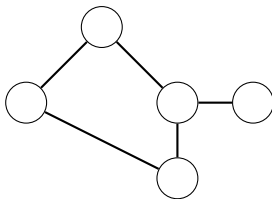
vuelta-atrás:  $S \times k \rightarrow S$ 
preparar-nivel( $k$ )
mientras  $\neg \text{último-hijo-nivel}(k)$ 
     $S[k] \leftarrow \text{siguiente-hijo-nivel}(k)$ 
    si solución( $S, k$ )
        procesar( $S, k$ )
    si no
        si completable( $S, k$ )
            vuelta-atrás( $S, k + 1$ )

```

## Elementos

- Hay que tomar  $n$  decisiones
- $S$ : vector de  $n$  elementos con las decisiones tomadas en los niveles 1.. $k$  del árbol
- *solución*( $sol, k$ ): ¿es  $sol$  una solución completa?
- *completable*( $sol, k$ ): ¿puede ser  $sol$  parte de una solución completa?
- *procesar*( $sol, k$ ) usa la solución

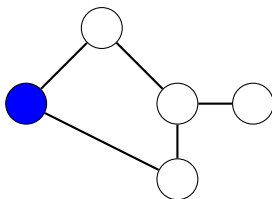
# Coloreado de grafos: búsqueda con retroceso (I)



## Problema

- ¿Se pueden colorear los nodos con  $m$  colores de forma que dos vértices adyacentes no tengan el mismo color?
- ¿Qué valores de  $m$  valdrían para el grafo de la izquierda?

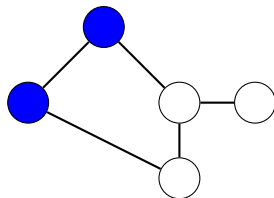
# Coloreado de grafos: búsqueda con retroceso (I)



## Problema

- ¿Se pueden colorear los nodos con  $m$  colores de forma que dos vértices adyacentes no tengan el mismo color?
- ¿Qué valores de  $m$  valdrían para el grafo de la izquierda?

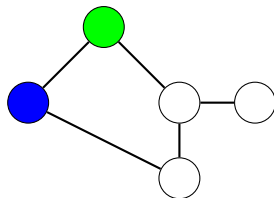
# Coloreado de grafos: búsqueda con retroceso (I)



## Problema

- ¿Se pueden colorear los nodos con  $m$  colores de forma que dos vértices adyacentes no tengan el mismo color?
- ¿Qué valores de  $m$  valdrían para el grafo de la izquierda?

# Coloreado de grafos: búsqueda con retroceso (I)

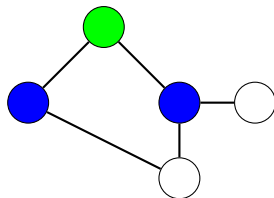


## Problema

- ¿Se pueden colorear los nodos con  $m$  colores de forma que dos vértices adyacentes no tengan el mismo color?
- ¿Qué valores de  $m$  valdrían para el grafo de la izquierda?



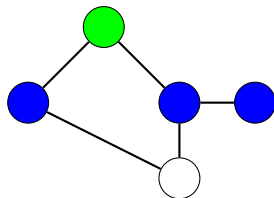
# Coloreado de grafos: búsqueda con retroceso (I)



## Problema

- ¿Se pueden colorear los nodos con  $m$  colores de forma que dos vértices adyacentes no tengan el mismo color?
- ¿Qué valores de  $m$  valdrían para el grafo de la izquierda?

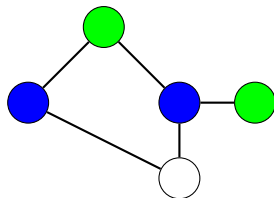
# Coloreado de grafos: búsqueda con retroceso (I)



## Problema

- ¿Se pueden colorear los nodos con  $m$  colores de forma que dos vértices adyacentes no tengan el mismo color?
- ¿Qué valores de  $m$  valdrían para el grafo de la izquierda?

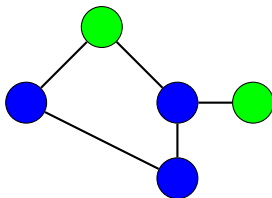
# Coloreado de grafos: búsqueda con retroceso (I)



## Problema

- ¿Se pueden colorear los nodos con  $m$  colores de forma que dos vértices adyacentes no tengan el mismo color?
- ¿Qué valores de  $m$  valdrían para el grafo de la izquierda?

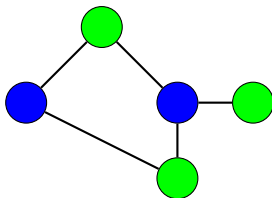
# Coloreado de grafos: búsqueda con retroceso (I)



## Problema

- ¿Se pueden colorear los nodos con  $m$  colores de forma que dos vértices adyacentes no tengan el mismo color?
- ¿Qué valores de  $m$  valdrían para el grafo de la izquierda?

# Coloreado de grafos: búsqueda con retroceso (I)



## Problema

- ¿Se pueden colorear los nodos con  $m$  colores de forma que dos vértices adyacentes no tengan el mismo color?
- ¿Qué valores de  $m$  valdrían para el grafo de la izquierda?

## Coloreado de grafos: búsqueda con retroceso (II)



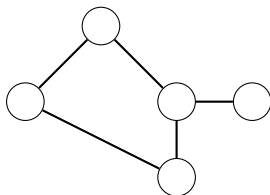
$\text{coloreado-grafo} : L \times n \times m \times S \times k \rightarrow S$   
 desde  $c \leftarrow 1$  hasta  $m$   
      $S[k] \leftarrow c$   
     si  $\text{coloreable}(L, S, k)$   
         si  $k = n$   
             imprimir( $S, k$ )  
         si no  
              $\text{coloreado-grafo}(L, n, m, S, k + 1)$



$\text{coloreable} : L \times S \times k \rightarrow r$   
 $r \leftarrow \text{True}$   
 $j \leftarrow 1$   
 mientras  $j < k \wedge r$   
     si  $k \in L[j] \vee j \in L[k]$   
          $r \leftarrow S[j] \neq S[k]$   
          $j \leftarrow j + 1$

# Coloreado de grafos: algoritmo devorador

En este ejemplo, ordenamos por el número de aristas incidentes.



*coloreado-devorador* :  $L \times n \rightarrow S \times m$

desde  $i \leftarrow 1$  hasta  $n$

$v[i] \leftarrow i$

$S[i] \leftarrow 0$

*ordenar-vertices*( $v, L, n$ )

$m \leftarrow 1$

$C \leftarrow \{m\}$

desde  $i \leftarrow 1$  hasta  $n$

$D \leftarrow C$

para todo  $j \in L[v[i]]$

$D \leftarrow D - \{S[j]\}$

si  $D = \emptyset$

$m \leftarrow m + 1$

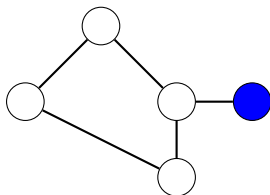
$D \leftarrow \{m\}$

$C \leftarrow C \cup \{m\}$

$S[v[i]] \leftarrow \min(D)$

# Coloreado de grafos: algoritmo devorador

En este ejemplo, ordenamos por el número de aristas incidentes.



*coloreado-devorador* :  $L \times n \rightarrow S \times m$

desde  $i \leftarrow 1$  hasta  $n$

$v[i] \leftarrow i$

$S[i] \leftarrow 0$

*ordenar-vertices*( $v, L, n$ )

$m \leftarrow 1$

$C \leftarrow \{m\}$

desde  $i \leftarrow 1$  hasta  $n$

$D \leftarrow C$

para todo  $j \in L[v[i]]$

$D \leftarrow D - \{S[j]\}$

si  $D = \emptyset$

$m \leftarrow m + 1$

$D \leftarrow \{m\}$

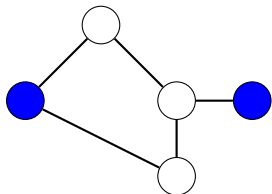
$C \leftarrow C \cup \{m\}$

$S[v[i]] \leftarrow \min(D)$



# Coloreado de grafos: algoritmo devorador

En este ejemplo, ordenamos por el número de aristas incidentes.



*coloreado-devorador* :  $L \times n \rightarrow S \times m$

desde  $i \leftarrow 1$  hasta  $n$

$v[i] \leftarrow i$

$S[i] \leftarrow 0$

*ordenar-vertices*( $v, L, n$ )

$m \leftarrow 1$

$C \leftarrow \{m\}$

desde  $i \leftarrow 1$  hasta  $n$

$D \leftarrow C$

para todo  $j \in L[v[i]]$

$D \leftarrow D - \{S[j]\}$

si  $D = \emptyset$

$m \leftarrow m + 1$

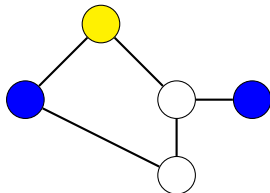
$D \leftarrow \{m\}$

$C \leftarrow C \cup \{m\}$

$S[v[i]] \leftarrow \min(D)$

# Coloreado de grafos: algoritmo devorador

En este ejemplo, ordenamos por el número de aristas incidentes.



*coloreado-devorador* :  $L \times n \rightarrow S \times m$

desde  $i \leftarrow 1$  hasta  $n$

$v[i] \leftarrow i$

$S[i] \leftarrow 0$

*ordenar-vertices*( $v, L, n$ )

$m \leftarrow 1$

$C \leftarrow \{m\}$

desde  $i \leftarrow 1$  hasta  $n$

$D \leftarrow C$

para todo  $j \in L[v[i]]$

$D \leftarrow D - \{S[j]\}$

si  $D = \emptyset$

$m \leftarrow m + 1$

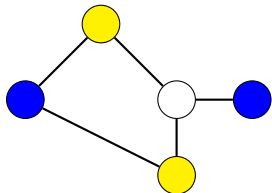
$D \leftarrow \{m\}$

$C \leftarrow C \cup \{m\}$

$S[v[i]] \leftarrow \min(D)$

# Coloreado de grafos: algoritmo devorador

En este ejemplo, ordenamos por el número de aristas incidentes.



*coloreado-devorador* :  $L \times n \rightarrow S \times m$

desde  $i \leftarrow 1$  hasta  $n$

$v[i] \leftarrow i$

$S[i] \leftarrow 0$

*ordenar-vertices*( $v, L, n$ )

$m \leftarrow 1$

$C \leftarrow \{m\}$

desde  $i \leftarrow 1$  hasta  $n$

$D \leftarrow C$

para todo  $j \in L[v[i]]$

$D \leftarrow D - \{S[j]\}$

si  $D = \emptyset$

$m \leftarrow m + 1$

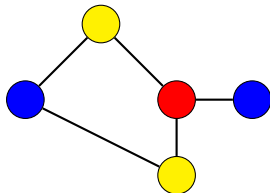
$D \leftarrow \{m\}$

$C \leftarrow C \cup \{m\}$

$S[v[i]] \leftarrow \text{mín}(D)$

# Coloreado de grafos: algoritmo devorador

En este ejemplo, ordenamos por el número de aristas incidentes.



*coloreado-devorador* :  $L \times n \rightarrow S \times m$

desde  $i \leftarrow 1$  hasta  $n$

$v[i] \leftarrow i$

$S[i] \leftarrow 0$

*ordenar-vertices*( $v, L, n$ )

$m \leftarrow 1$

$C \leftarrow \{m\}$

desde  $i \leftarrow 1$  hasta  $n$

$D \leftarrow C$

para todo  $j \in L[v[i]]$

$D \leftarrow D - \{S[j]\}$

si  $D = \emptyset$

$m \leftarrow m + 1$

$D \leftarrow \{m\}$

$C \leftarrow C \cup \{m\}$

$S[v[i]] \leftarrow \min(D)$

# Coloreado de grafos: algunas curiosidades

## Grafos planos

- Informalmente: grafos dibujables sin cruces de aristas
- Ejemplo común: mapas (adyacencia de países)
- Existen definiciones formales (Kuratowski, Wagner) y algoritmos  $O(n)$  para comprobar si un grafo es planar

## Resultados conocidos

- Se puede colorear cualquier grafo *plano* (dibujable sin aristas cruzadas) con 4 colores
- En grafos planos existen algoritmos  $O(n^2)$  para 4 colores, y  $O(n)$  para 5 colores
- Los grafos no planos son mucho más complicados (es un problema “difícil”)

# Problema de la mochila discreta con pesos reales

## Anteriores temas

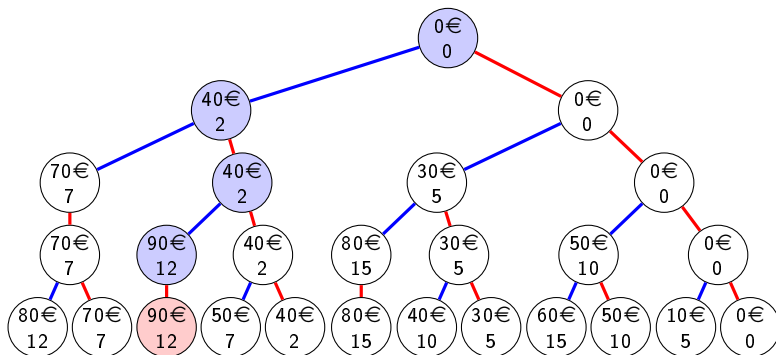
- Mochila continua: algoritmo devorador
- Mochila discreta con pesos enteros: programación dinámica

## Posibles soluciones

- Algoritmo devorador: ordenar por relación  $v/p$  no es óptimo en la versión discreta
- Programación dinámica: existen algoritmos para este problema, pero son bastante más complejos
- En este tema lo plantearemos como un problema de búsqueda con retroceso: mismo esquema, pero guardando la mejor solución encontrada hasta ahora

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

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



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

$mdr : v \times p \times n \times c \rightarrow S$

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)

$mdr\text{-}aux : v \times p \times n \times c \times S \times k \times M \rightarrow M$

si  $p[k] + peso(S) \leq c$

$S[k] \leftarrow \top$

si  $k = n$

si  $valor(S) > valor(M)$

$M \leftarrow S$

si no

$M \leftarrow mdr\text{-}aux(v, p, n, c, S, k + 1, M)$

$S[k] \leftarrow \perp$

si  $k = n$

si  $valor(S) > valor(M)$

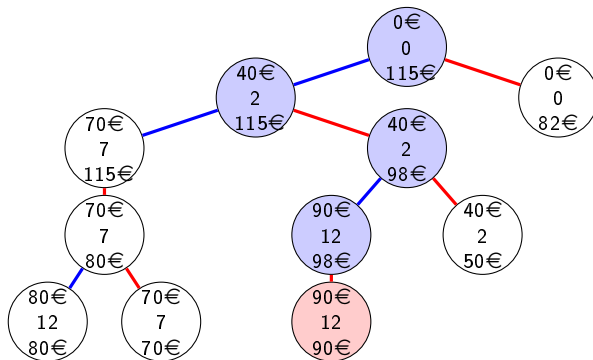
$M \leftarrow S$

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 nodos menos prometedores.



# Versión 2: acotación superior (II)

$mdr\text{-}aux : v \times p \times n \times c \times S \times k \times M \rightarrow M$

si  $p[k] + peso(S) \leq c$

$S[k] \leftarrow \top$

si  $k = n$

$M \leftarrow S$

si no

$M \leftarrow mdr\text{-}aux(v, p, n, c, S, k + 1, M)$

$S[k] \leftarrow \perp$

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)

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

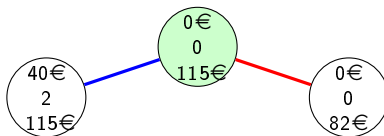
## Versión 3: expansión ordenada por cota superior

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



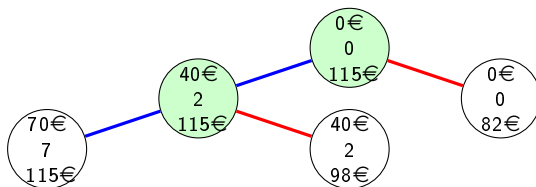
## Versión 3: expansión ordenada por cota superior

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



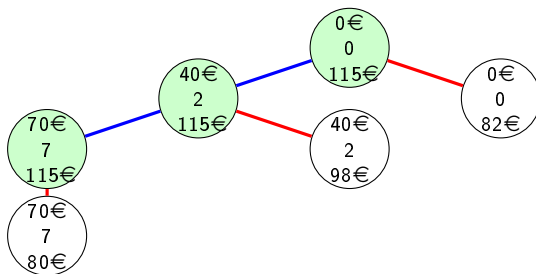
## Versión 3: expansión ordenada por cota superior

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



# Versión 3: expansión ordenada por cota superior

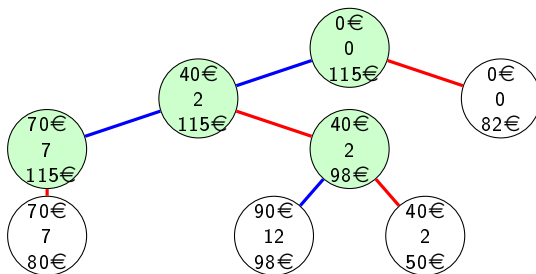
- A cada paso elegimos el siguiente nodo con la mayor cota superior que sea mayor al valor máximo hasta ahora
- En vez de un recorrido primero en profundidad, los nodos sin expandir se añaden a un montículo





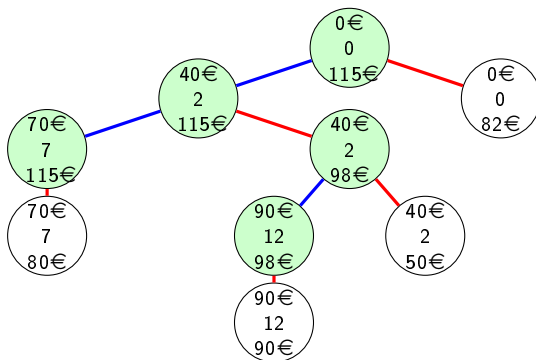
# Versión 3: expansión ordenada por cota superior

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



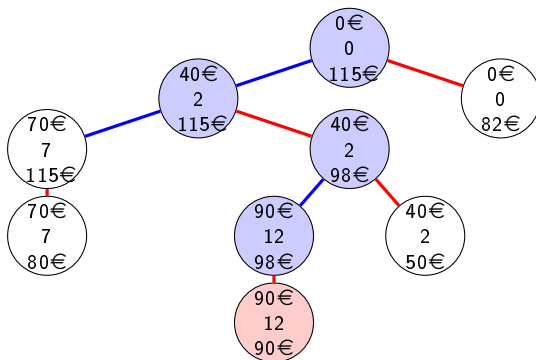
# Versión 3: expansión ordenada por cota superior

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



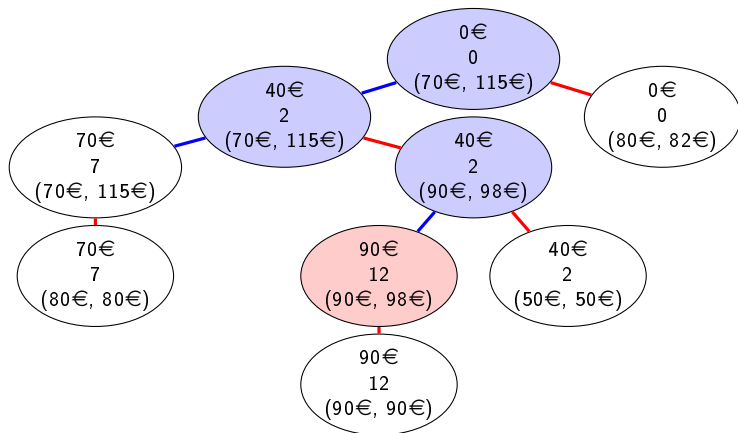
# Versión 3: expansión ordenada por cota superior

- A cada paso elegimos el siguiente nodo con la mayor cota superior que sea mayor al valor máximo hasta ahora
- En vez de un recorrido primero en profundidad, los nodos 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

Encontrar el camino más corto entre dos vértices de un grafo  $G = \langle V, A \rangle$ .

El algoritmo de Dijkstra, de complejidad  $t(n) \in \Theta(n^2)$ , calcula la menor distancia desde un vértice origen a todos los demás.

Ej: Tablero de  $50 \times 50$  celdas.

- $n = |V| = 250$  vértices
- $t(250) \approx 250^2 = 62500$

# Fuerza bruta

Usando un esquema de exploración primero en profundidad.  
Tantos nodos hijos como adyacentes  $d$  tenga cada vértice en el grafo  $G$ .

$$t(n, d) \in \Theta(d^n)$$

Ej: Tablero de 50 x 50 celdas.

- $n = |V| = 250$  vértices
- 8 celdas adyacentes
- $t(250, 8) \approx 8^{250} \approx 6 \times 10^{255}$

Se necesita una estrategia de marcado para evitar ciclos (además, reduce el número de nodos del árbol considerablemente).  
Peor que Dijkstra.

# Algoritmo A\*

Expandir el árbol de forma ordenada.

Es necesario una heurística: distancia directa entre los vértices (euclídea, de Manhattan...), sin tener en cuenta los obstáculos.

Distancia estimada del camino entre el origen y el destino que pasa por el nodo k: distancia del mejor

cota inferior  $\rightarrow$  poda

# Estructuras de datos



- $L[i]$ : lista de adyacencia del nodo  $i$
- $P[i]$ : nodo desde el que se alcanza el nodo  $i$
- *closed*: lista de nodos procesados
- *opened*: lista de nodos pendientes
- $G[i]$ : distancia del camino más corto entre el origen y el nodo  $i$
- $H[i]$ : distancia estimada del camino más corto entre el el nodo  $i$  y el destino
- $F[i]: G[i] + H[i]$



# Uso de montículos

- $push(\text{monticulo}, i, \phi)$ : introduce el elemento  $i$  en el montículo por mínimos, y lo reordena de acuerdo a la función de coste  $\phi$ .
- $pop(\text{monticulo}, \phi)$ : extrae el tope del montículo y lo reordena de acuerdo a la función de coste  $\phi$ .
- $update(\text{monticulo}, \phi)$ : reordena el montículo de acuerdo a la función de coste  $\phi$ .

# Algoritmo A\*

```
 $A^* : L \times begin \times target \rightarrow P$   
 $\langle closed, cur \rangle \leftarrow \langle \emptyset, begin \rangle$   
 $\langle G[cur], H[cur], P[cur] \rangle \leftarrow \langle 0, estimatedDistance(cur, target), \emptyset \rangle$   
 $F[cur] \leftarrow G[cur] + H[cur]$   
 $push(opened, cur, F)$   
mientras  $\neg found \wedge |opened| > 0$   
     $cur \leftarrow pop(opened, F)$   
     $closed \leftarrow closed \cup \{cur\}$   
    si  $cur = target$   
         $found \leftarrow \top$   
    si-no  
        para todo  $j \in L[cur]$   
            si  $j \notin closed$   
                si  $j \notin open$   
                     $\langle P[j], G[j], H[j] \rangle \leftarrow \langle cur, G[cur] + distance(cur, j), estimatedDistance(j, target) \rangle$   
                     $F[j] \leftarrow G[j] + H[j]$   
                     $push(opened, j, F)$   
                si-no  
                     $d \leftarrow distance(cur, j)$   
                    si  $G[j] > G[cur] + d$   
                         $\langle P[j], G[j], F[j] \rangle \leftarrow \langle cur, G[cur] + d, G[j] + H[j] \rangle$   
                         $update(opened)$ 
```

# Recuperación del camino

$recupera : begin \times target \times P \rightarrow C$

$cur \leftarrow target$

$C \leftarrow target$

mientras  $P[cur] \neq begin$

$cur \leftarrow P[cur]$

$C \leftarrow cur + C$

- Donde el operador  $+$  ha sido sobrecargado para actuar como operador de concatenación de listas

# 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.

4ª edición, Jones and Bartlett Publishers, 2011.