Backtracking

IIC2133

1-2016

Un cierto tipo de problemas de búsqueda

En muchos problemas buscamos

- un conjunto de (todas las) soluciones factibles
- una solución óptima que satisface ciertas restricciones

Estos problemas pueden resolverse usando *back-tracking*:

- la solución buscada puede expresarse como una tupla o vector (x_1, …, x_n), en que x_i es elegido de algún conjunto finito S_i
- el problema es encontrar un vector que maximice (o minimice o satisfaga) una *función criterio* P(x_1, …, x_n)
- … o en encontrar todos los vectores que satisfacen P

Ejemplo

El problema de las 8 reinas

¿Cómo disponemos 8 reinas en un tablero de ajedrez de 8×8 de modo que ningunas dos de ellas estén en la misma fila, columna o diagonal?

Numeremos las filas, columnas y reinas 1 a 8

Como cada reina debe estar en una fila diferente, suponemos que la reina i queda en la fila i

Las soluciones pueden ser representadas como tuplas $(x_1, \ldots, x_8)$:

- x_i es la columna en la cual va a quedar la reina i

Acerca de la complejidad de estos problemas

Si m_i es el tamaño de S_i, entonces hay

$$m = m_1 \times m_2 \times \ldots \times m_n$$

tuplas que son candidatos posibles para satisfacer *P*

Una estrategia sería formar las tuplas y evaluar cada una con *P*,

… registrando aquellas que producen el óptimo

… o aquellas que simplemente satisfacen *P*

La ventaja de *backtracking*

Un algoritmo de *backtracking* puede producir la misma respuesta

… **haciendo muchos menos que *m* ensayos**:

La idea básica:

- construir el vector solución de a una componente a la vez
- usar funciones criterios modificadas $P_i(x_1, \ldots, x_i)$ —*funciones de acotamiento*— para probar si el vector que está siendo formado tiene alguna posibilidad de éxito

Ventaja:

- si nos damos cuenta de que el vector parcial $(x_1, \ldots, x_i)$ no puede llevarnos a una solución válida

  … entonces podemos ignorar completamente $m_{i+1} \times \ldots \times m_n$ vectores de prueba posibles

Las soluciones deben satisfacer un conjunto de restricciones

Restricciones explícitas:

- restringen cada x_i a tomar valores sólo de un conjunto S_i dado
- dependen de la instancia particular, *I*, del problema que se está resolviendo
- todas las tuplas que satisfacen estas restricciones definen un posible *espacio de soluciones* para *I*

Restricciones implícitas:

- determinan cuáles de las tuplas en el espacio de soluciones de *I* satisfacen *P*

  … es decir, describen la forma en que los x_i deben relacionarse entre ellos

Restricciones en el problema de las reinas

Restricciones explícitas:

- $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \leq i \leq 8$
- el espacio de soluciones consiste en 8^8 tuplas

Restricciones implícitas:

- ningunos dos *xi*'s pueden ser iguales
- todas las soluciones son permutaciones de la tupla {1, 2, 3, 4, 5, 6, 7, 8}
- reduce el espacio de soluciones de 8^8 tuplas (16,777,216) a sólo 8! tuplas (40,320)
- ningunas dos reinas pueden estar en la misma diagonal

Una solución es {4, 6, 8, 2, 7, 1, 3, 5}

Otro ejemplo
Suma de subconjuntos

Dados

… un conjunto de números positivos w_i , $1 \leq i \leq n$

… y un número positivo m

… encontrar todos los subconjuntos de w_i cuya suma sea m

P.ej., si $n = 4$

$(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$

$m = 31$

… entonces los subconjuntos buscados son

$\{11, 13, 7\}$ y $\{24, 7\}$

El espacio de soluciones tiene 2^n tuplas

En lugar de representar el vector solución por los w_i que suman m,

… lo representaremos por los índices de estos w_i :

- las soluciones del ej. son los vectores {1, 2, 4} y {3, 4}

En general

- todas las soluciones son tuplas $(x_1, \ldots, x_k)$, $1 \le k \le n$
- las tuplas pueden ser de diferentes tamaños

Restricciones explícitas:

- $x_i \in \{ j \mid j$ es un entero y $1 \le j \le n \}$

Restricciones implícitas:

- ningunos dos x_i son iguales
- la suma de los correspondientes w_{xi} es m

Para evitar generar múltiples veces el mismo subconjunto, imponemos la restricción implícita

- $x_i < x_{i+1}$, $1 \le i < k$

Solución al problema de las n reinas
(generalización del problema de las 8 reinas)

Sea $(x_1, \ldots, x_n)$ una solución:

- x_i es la columna de la fila i en la cual ponemos a la reina i
- los x_i's son todos distintos —no puede haber dos reinas en la misma columna

¿Cómo probamos si dos reinas en las casillas (i, j) y (p, q) están en una misma diagonal?

- las casillas en una misma diagonal ➘ (⬋) tienen la misma diferencia (suma) entre (de) su número de fila y (más) su número de columna

Luego, las dos reinas están en una misma diagonal si y sólo si

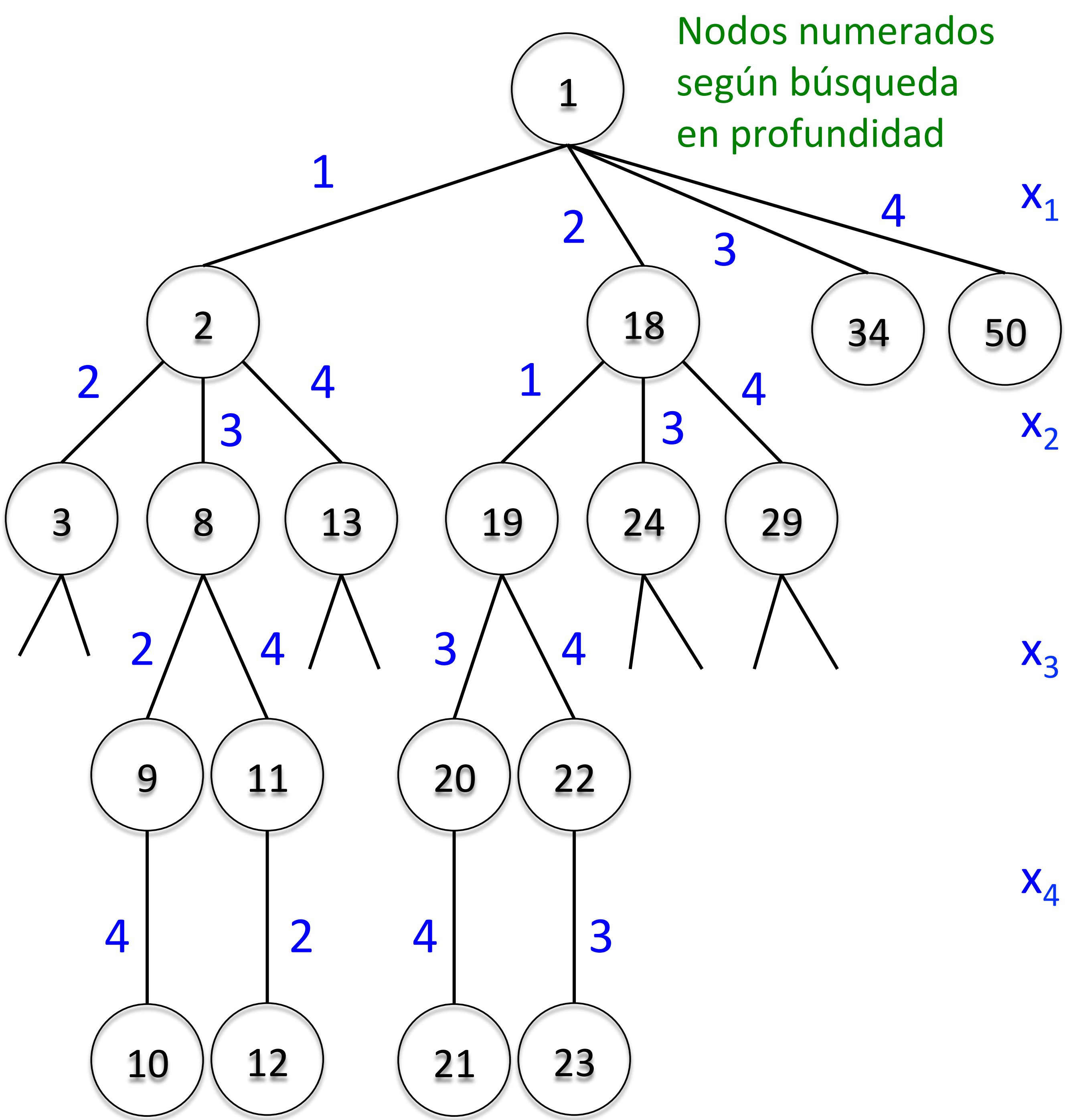
$$i - j = p - q \quad \vee \quad i + j = p + q$$

… que implican $j - q = i - p$ y $j - q = p - i$

Las dos reinas están en la misma diagonal si y sólo si

$$|j - q| = |i - p|$$

Espacio de soluciones para 4 reinas organizado como árbol (parcial)



`place(k,i)` = la reina *k* puede ponerse en la columna *i*; prueba dos condiciones:

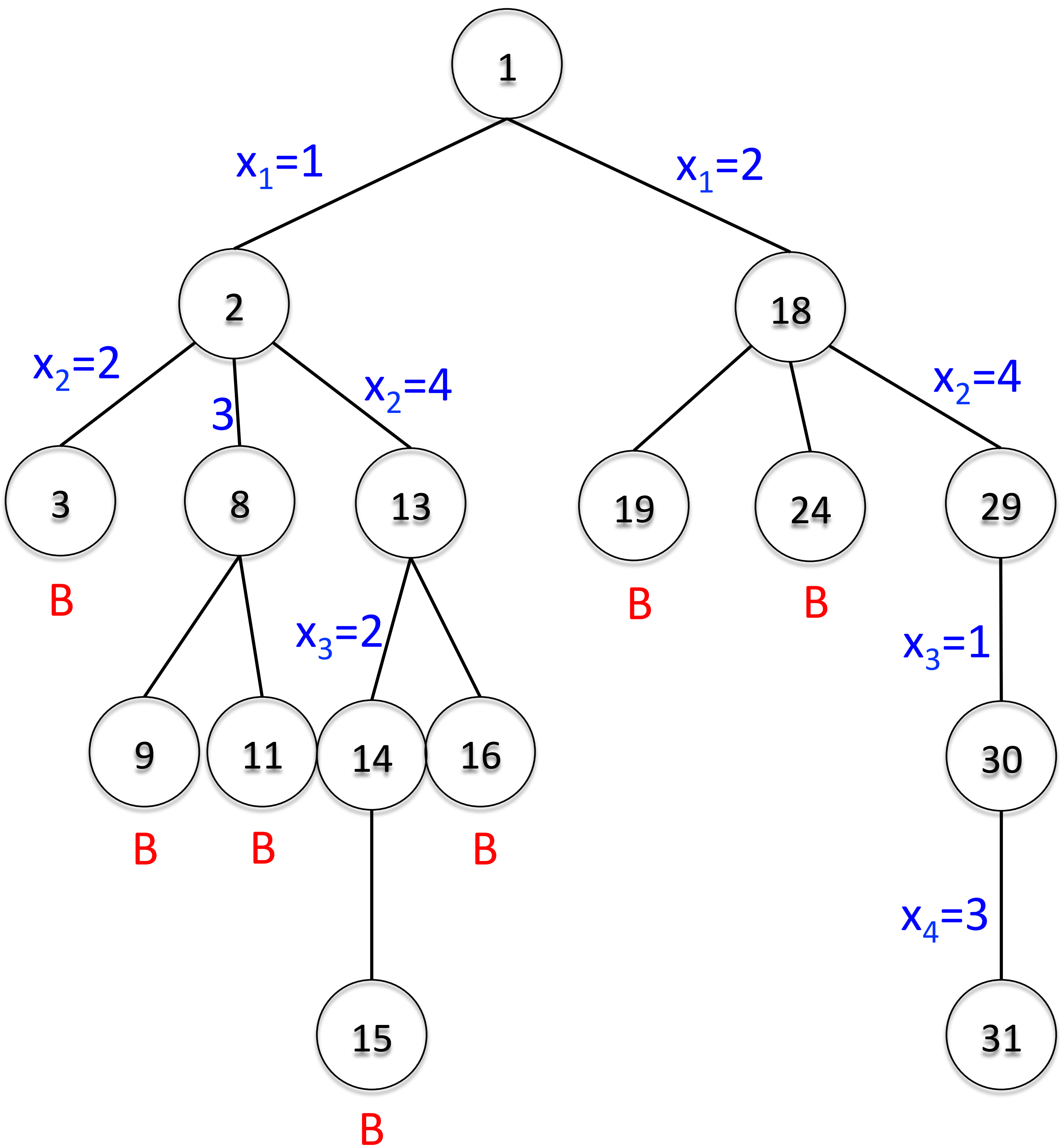
- si *i* es distinto de todos los valores previos $x_1, \ldots, x_{k-1}$
- si no hay otra reina en la misma diagonal

```
place(k,i):
    for j = 1 … k-1:
        if (x[j] == i) ∨ |x[j]-i| == |j-k|:
            return false
    return true
```

La llamada inicial es `nQueens(1,n)` y $x[1 : n]$ es global

```
nQueens(k,n):
    for i = 1 … n:
        if place(k,i):
            x[k] = i
            if k == n: output x[1 : n]
            else: nQueens(k+1,n)
```

Árbol generado por backtracking
(ver figura en la pizarra)



Solución al problema de la suma de subconjuntos

Dados n números positivos distintos w_i , encontrar todas las combinaciones de ellos cuya suma sea m

Solución usando tuplas $(x_1, \ldots, x_n)$ de tamaño fijo = n:

- x_i es 1 o 0 dependiendo de si w_i es incluido o no
- (las soluciones al problema son {0,0,1,1} y {1,1,0,1})
- la función de cota $B_k(x_1, \ldots, x_k)$ es verdadera si y sólo si

$$\sum_{i=1,\ldots,k} w_i x_i + \sum_{i=k+1,\ldots,n} w_i \geq m$$

Las funciones de cota pueden fortalecerse si los w_i's están ordenados de menor a mayor:

$x_1, \ldots, x_k$ no lleva a una solución si $\sum_{i=1,\ldots,k} w_i x_i + w_{k+1} > m$

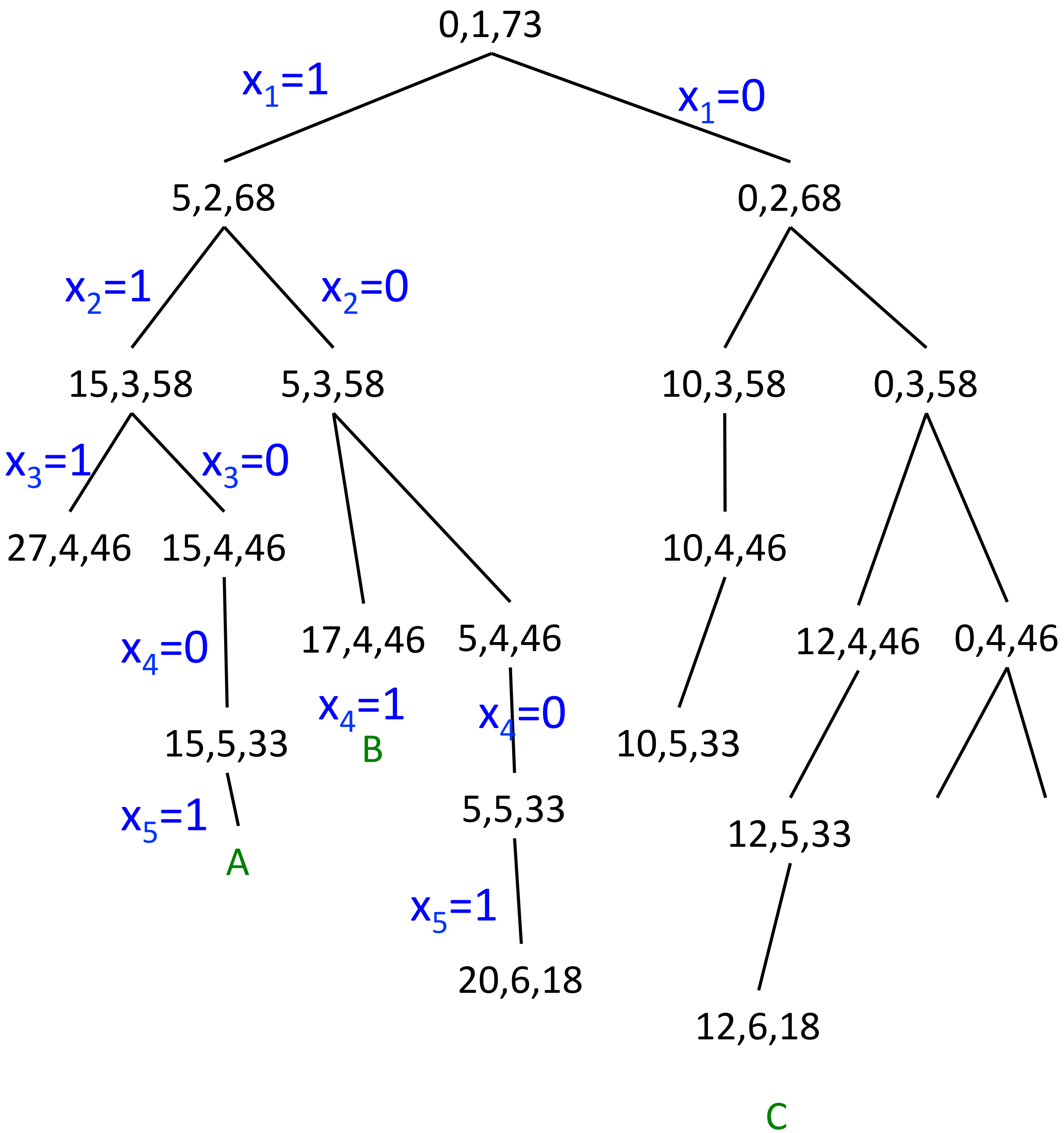
Funciones de cota:

- $B_k(x_1, \ldots, x_k) = true \Leftrightarrow \sum_{i=1,\ldots,k} w_i x_i + \sum_{i=k+1,\ldots,n} w_i \geq m$
  $\wedge \sum_{i=1,\ldots,k} w_i x_i + w_{k+1} \leq m$

La llamada inicial es `sumofSubsets(0,1,`R`)`, en que $R = \sum_{i=1,\dots,n} w_i$:

```
sumofSubsets(s,k,r):
   x[k] = 1
   if s+w[k] == m:
      output x[1:k]
   else:
      if s+w[k]+w[k+1] ≤ m:
         sumofSubsets(s+w[k], k+1, r−w[k])
   if (s+r−w[k] ≥ m) ∧ (s+w[k+1] ≤ m):
      x[k] = 0
      sumofSubsets(s, k+1, r−w[k])
```

Árbol generado por backtracking
($n = 6$, $m = 30$, $w = \{5,10,12,13,15,18\}$)



El proceso de backtracking

Queremos encontrar todas las respuestas

Sean:

- $(x_1, \ldots, x_i)$ una secuencia de valores para los primeros i componentes del vector solución
- $T(x_1, \ldots, x_i)$ el conjunto de los valores posibles para x_{i+1}
- B_{i+1} una función de cota, expresada como predicado —si $B_{i+1}(x_1, \ldots, x_{i+1})$ es falsa, la secuencia$(x_1, \ldots, x_{i+1})$ no puede ser extendida para alcanzar una respuesta

Los candidatos para la posición i+1 del vector solución $(x_1, \ldots, x_n)$ son los valores generados por T y que satisfacen B_{i+1}

Dos formulaciones del algoritmo general de *backtracking*

```
backtrack(k): —formulación recursiva
   for each x[k] ∈ T(x[1],…,x[k−1]):
      if B_k(x[1],…,x[k]):
         if x[1],…,x[k] es una ruta a una respuesta:
            output x[1:k]
         if k < n: backtrack(k+1)
```

```
ibacktrack(n): —formulación iterativa
   k = 1
   while k ≠ 0:
      if queda un x[k] ∈ T(x[1],…,x[k−1] que no hemos probado and B_k(x[1],…,x[k]):
         if x[1],…,x[k] es una ruta a una respuesta:
            output x[1:k]
         k = k+1
      else:
         k = k−1
```

Coloración de grafos

Problema de *decisión*: determinar si G puede ser coloreado de manera que dos vértices adyacentes tengan siempre colores distintos y sólo se use m colores

Problema de *optimización*: encontrar el menor entero m para el cual G puede ser coloreado —m es el *número cromático* de G

Nuestro problema: Determinar todas las formas diferentes en las que G puede ser coloreado usando a lo más m colores

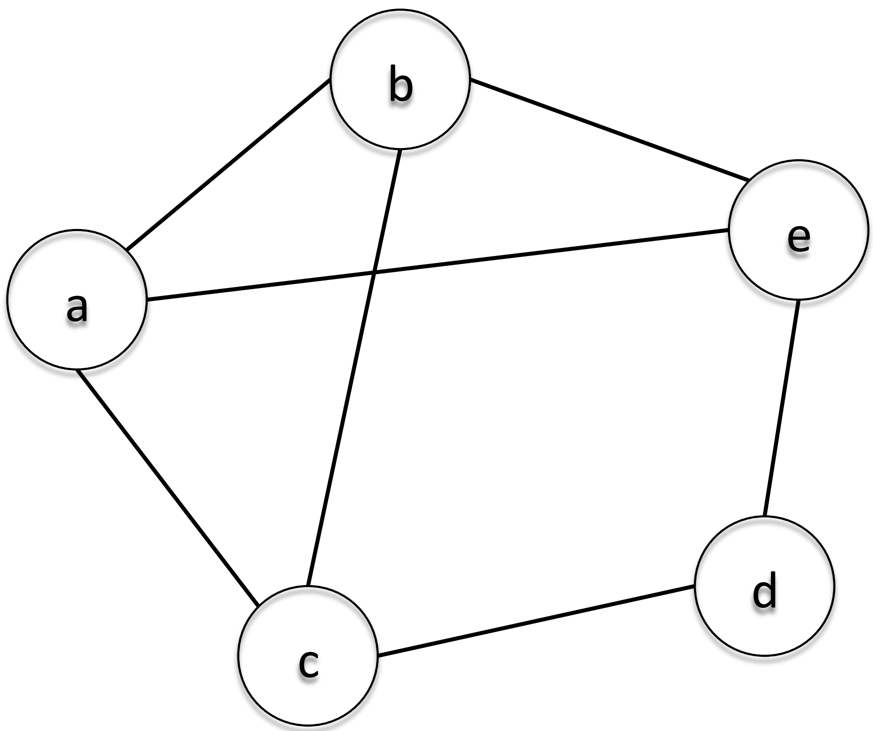
Representamos:

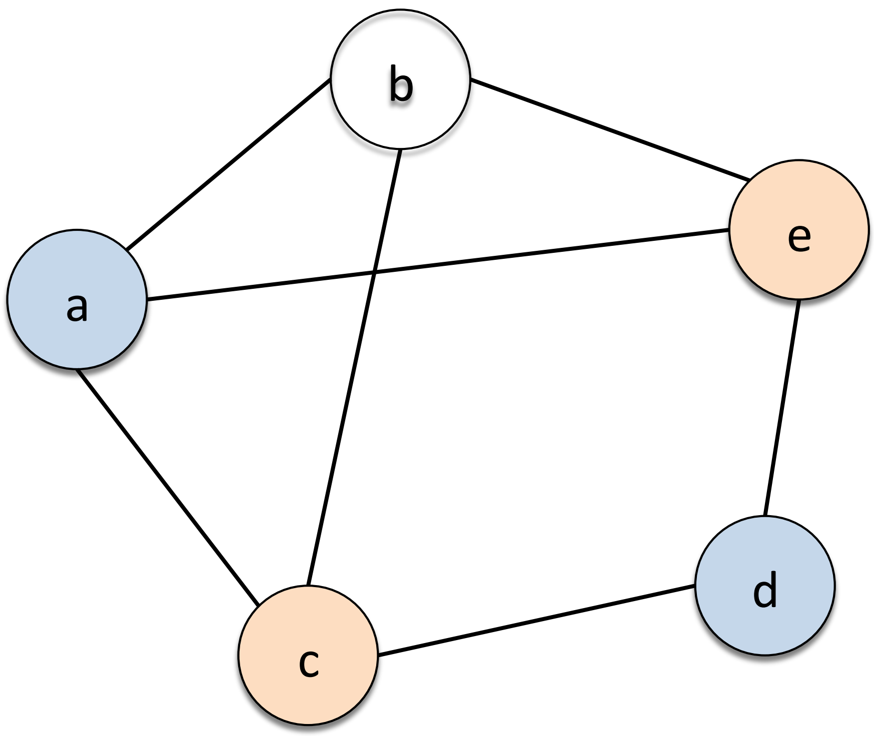
- G por su matriz de adyacencias $G[1 : n][1 : n]$:

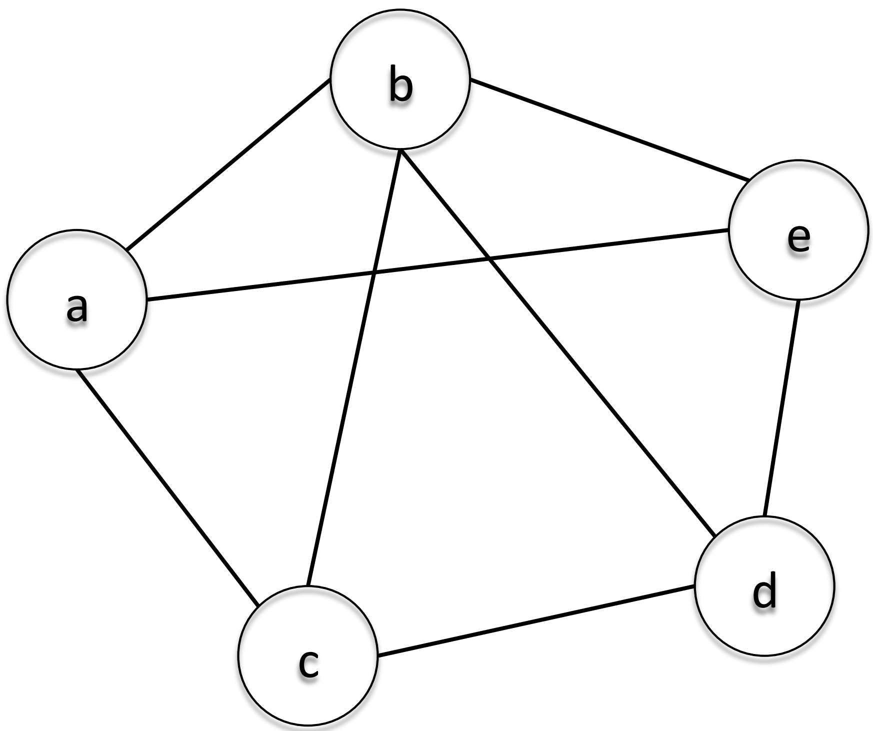
  $G[i][j] = 1$ si (i, j) es una arista de G

  $G[i][j] = 0$ en caso contrario

- los colores por los enteros 1, 2, …, m; un color igual a 0 siginifica que no existe un color distinto para asignar
- las soluciones por las tuplas $(x_1, \ldots, x_n)$, en que x_i es el color del nodo i


b
e
a
d
c


b
a
e
d
c


b
e
a
d
c

Inicialmente

- ponemos el arreglo $x[1 : n]$ en 0
- hacemos la llamada `mColoring(1)`

```
mColoring(k):
   while true:
      nextValue(k)
      if x[k] == 0: break
      if k == n: output x[1:n]
      else mColoring(k+1)

nextValue(k):
   while true:
      x[k] = (x[k]+1) % (m+1)
      if x[k] == 0: return
      for j = 1 … n:
         if G[k][j] ∧ x[k] == x[j]: break
      if j == n+1: return
```