4 Búsqueda con retroceso

4 Búsqueda con retroceso: Introducción

- Problemas que consideraremos:
  - **Búsqueda** de la mejor o **del conjunto de todas las soluciones que satisfacen ciertas condiciones**.
  - Cada solución es el resultado de una **secuencia de decisiones**.
  - Existe una función objetivo que debe ser satisfecha por cada selección u optimizada (si sólo queremos la mejor).
  - En algunos problemas de este tipo se conoce un criterio óptimo de selección en cada decisión: técnica **voraz**.
  - En otros problemas se cumple el principio de optimalidad de Bellman y se puede aplicar la técnica de la **programación dinámica**.
  - Existen otros problemas en los que no hay más remedio que **buscar**.

4 Búsqueda con retroceso: Introducción

- Planteamiento del problema:
  - Se trata de hallar todas las soluciones que satisfagan un predicado P.
  - La solución debe poder expresarse como una tupla $(x_1,\ldots,x_n)$ donde cada x_i pertenece a un C_i.
  - Si $|C_i| = t_i$, entonces hay

$$t = \prod_{i=1}^{n} t_i$$

    n-tuplas candidatas para satisfacer P.

  - Método de **fuerza bruta**: examinar las t n-tuplas y seleccionar las que satisfacen P.

  - **Búsqueda con retroceso** (*backtracking*, en inglés): formar cada tupla de manera progresiva, elemento a elemento, comprobando para cada elemento x_i añadido a la tupla que $(x_1,\ldots,x_i)$ puede conducir a una tupla completa satisfactoria.

4 Búsqueda con retroceso: Introducción

– Deben existir unas funciones objetivo parciales o **predicados acotadores** $P_i(x_1,\ldots,x_i)$.

↓

Dicen si $(x_1,\ldots,x_i)$ puede conducir a una solución.

– Diferencia entre fuerza bruta y búsqueda con retroceso:

si se comprueba que $(x_1,\ldots,x_i)$ no puede conducir a ninguna solución, se evita formar las $t_{i+1}\times\cdots\times t_n$ tuplas que comienzan por $(x_1,\ldots,x_i)$

– Para saber si una *n*-tupla es solución, suele haber dos tipos de restricciones:

- explícitas: describen el conjunto C_i de valores que puede tomar x_i (todas las tuplas que satisfacen estas restricciones definen un **espacio de soluciones posibles**);
- implícitas: describen las relaciones que deben cumplirse entre los x_i (qué soluciones posibles satisfacen el predicado objetivo *P*).

4 Búsqueda con retroceso: Introducción

- **Ejemplo**: el problema de las ocho reinas

  - El problema consiste en colocar ocho reinas en un tablero de ajedrez sin que se den jaque (dos reinas se dan jaque si comparten fila, columna o diagonal).

    Fuerza bruta: $\binom{64}{8} = 4.426.165.368$

  - Puesto que no puede haber más de una reina por fila, podemos replantear el problema como: "colocar una reina en cada fila del tablero de forma que no se den jaque".
    En este caso, para ver si dos reinas se dan jaque basta con ver si comparten columna o diagonal.

  - Por lo tanto, toda solución del problema puede representarse con una 8-tupla $(x_1,\ldots,x_8)$ en la que x_i es la columna en la que se coloca la reina que está en la fila i del tablero.

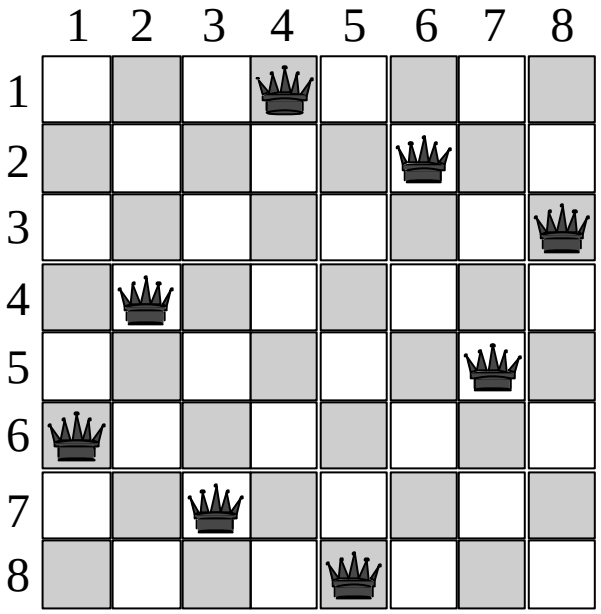
4 Búsqueda con retroceso: Introducción

- Restricciones explícitas:
  $C_i = \{1,2,3,4,5,6,7,8\}$, $1 = i = 8$

  $\Rightarrow$ El espacio de soluciones consta de 8^8 8-tuplas (16.777.216 8-tuplas)
- Restricciones implícitas:

  no puede haber dos reinas en la misma columna ni en la misma diagonal
- En particular, se deduce que todas las soluciones son permutaciones de la 8-tupla (1,2,3,4,5,6,7,8).

  $\Rightarrow$ El espacio de soluciones se reduce de 8^8 8-tuplas (16.777.216) a 8! 8-tuplas (40.320)

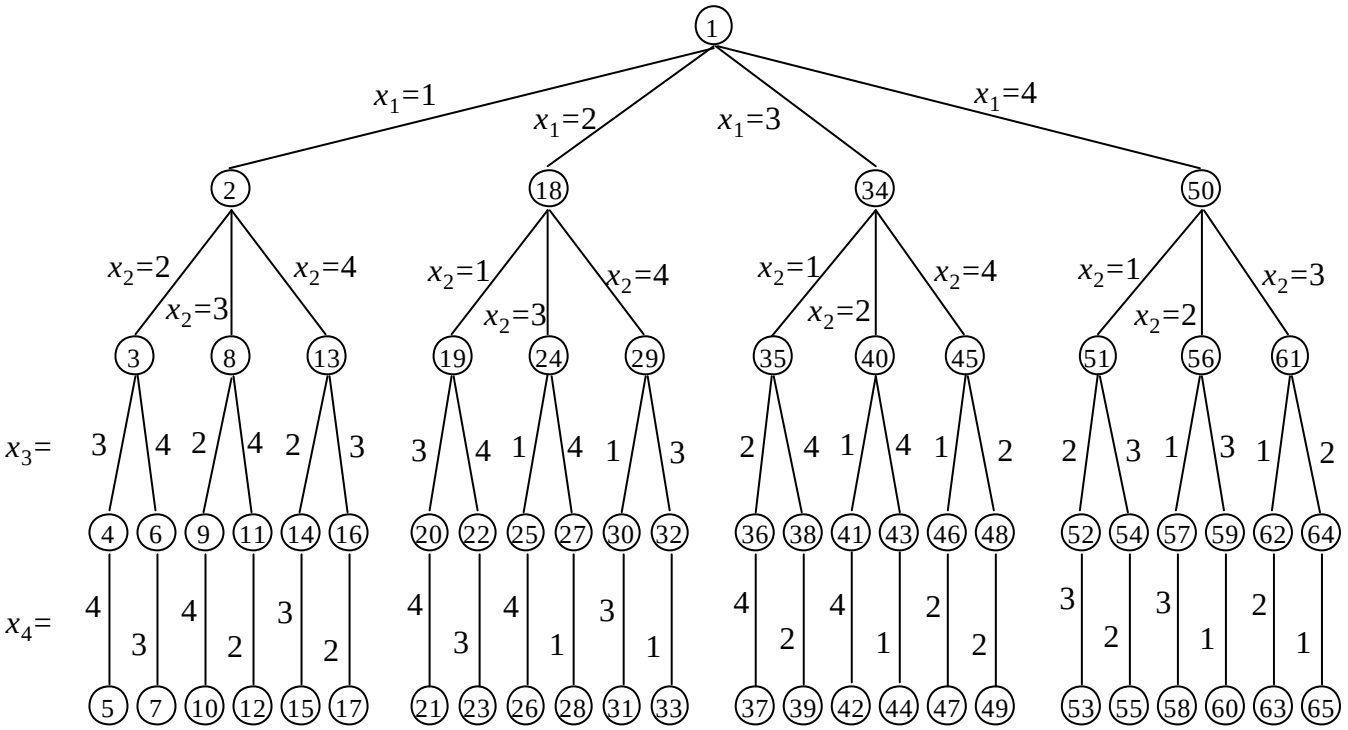


Una solución: (4,6,8,2,7,1,3,5)

4 Búsqueda con retroceso: Introducción

- Volviendo al planteamiento general:
  - Para facilitar la búsqueda, se adopta una **organización en árbol** del espacio de soluciones.

- En el ejemplo, para el problema de las cuatro reinas (en un tablero 4×4):



El espacio de soluciones está definido por todos los caminos desde la raíz a cada hoja (hay 4! hojas).

4 Búsqueda con retroceso: Introducción

- Esquema algorítmico:
  - Sea $(x_1,\ldots,x_i)$ un camino de la raíz hasta un nodo del árbol del espacio de estados.
  - Sea $G(x_1,\ldots,x_i)$ el conjunto de los valores posibles de x_{i+1} tales que $(x_1,\ldots,x_{i+1})$ es un camino hasta un nodo del árbol.
  - Suponemos que existe algún predicado acotador A tal que $A(x_1,\ldots,x_{i+1})$ es falso si el camino $(x_1,\ldots,x_{i+1})$ no puede extenderse para alcanzar un nodo de respuesta (i.e., una solución).
  - Por tanto, los candidatos para x_{i+1} son los valores de G tales que satisfacen A.
  - Supongamos, finalmente, que existe un predicado R que determina si un camino $(x_1,\ldots,x_{i+1})$ termina en un nodo respuesta (i.e., es ya una solución).

4 Búsqueda con retroceso: Introducción

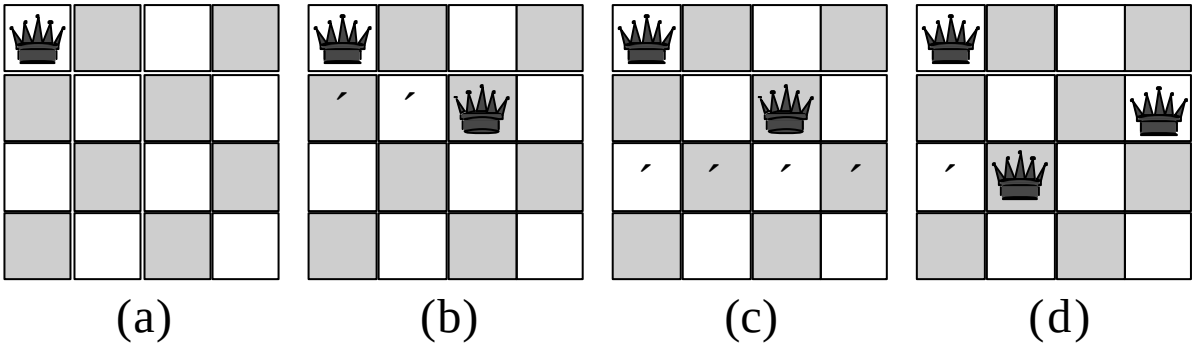
```
algoritmo retroceso(ent k:entero;
     entsal solución:vector[1..n]de elmto)
{Pre: solución[1..k-1] es 'prometedora'}
variable nodo:elmto
principio
  para todo nodo en G(solución,1,k-1) hacer
    solución[k]:=nodo;
    si A(solución,1,k)
      entonces
        si R(solución,1,k)
          entonces guardar(solución,1,k)
        fsi;
        retroceso(k+1,solución)
    fsi
  fpara
fin
```

La llamada inicial es:

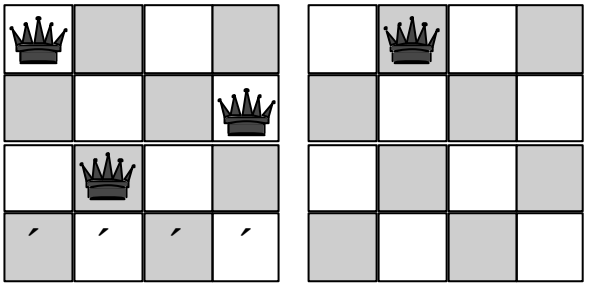
```
...
retroceso(1,solución);
...
```

4 Búsqueda con retroceso: Introducción

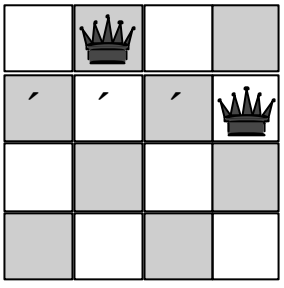
- En el ejemplo de las cuatro reinas:



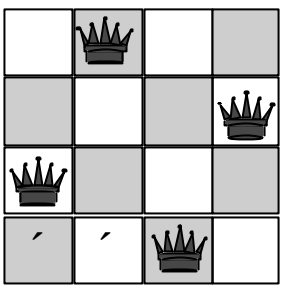
(a) (b) (c) (d)



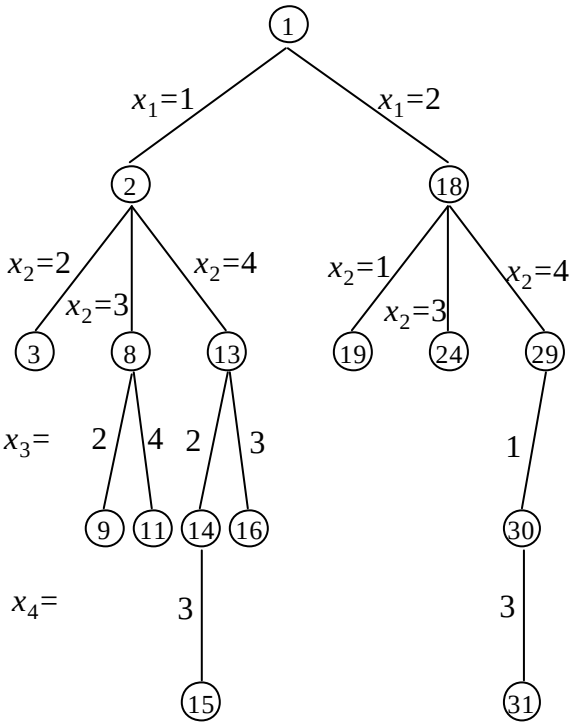
(e) (f)



(g)



(h)



(16 nodos frente a los 65 del árbol completo)

4 Búsqueda con retroceso: Introducción

- De nuevo, en general:

  - Nótese que el árbol no se construye explícitamente sino implícitamente mediante las llamadas recursivas del algoritmo de búsqueda.

  - El algoritmo no hace llamadas recursivas cuando:
    - $k = n+1$, o cuando
    - ningún nodo generado por G satisface A.

  - *Backtracking* =

    = búsqueda de primero en profundidad y con predicados acotadores

4 Búsqueda con retroceso: Introducción

- El algoritmo anterior halla todas las soluciones y además éstas pueden ser de longitud variable.

❖ Variantes:

- Limitar el número de soluciones a una sola añadiendo un parámetro booleano de salida que indique si se ha encontrado una solución.

- Forzar a que sólo los nodos hoja puedan significar solución (realizando la recursión sólo si no se ha encontrado un nodo solución):

```
si R(solución,1,k)
  entonces guardar(solución,1,k)
  sino retroceso(k+1,solución)
fsi
```

- Resolver problemas de optimización: además de la solución actual en construcción hay que guardar la mejor solución encontrada hasta el momento.
  Se mejora la eficiencia de la búsqueda si los predicados acotadores permiten eliminar los nodos de los que se sabe que no pueden llevar a una solución mejor que la ahora disponible (**poda**; métodos de **ramificación y acotación**).

4 Búsqueda con retroceso: Introducción

- Sobre la eficiencia:
  - Depende de:
    - el tiempo necesario para generar un elemento `solución[k]`,
    - el número de elementos solución que satisfacen las restricciones explícitas G,
    - el tiempo de ejecución de los predicados acotadores A, y
    - el número de elementos `solución[k]` que satisfacen los predicados A.
  - Mejoras:
    - Si se consigue que los predicados acotadores reduzcan mucho el número de nodos generados
      (aunque un buen predicado acotador precisa mucho tiempo de evaluación; compromiso…)
      - Si lo reducen a un solo nodo generado (solución **voraz**): $O(n)$ nodos a generar en total
      - En el peor caso, $O(p(n)\times 2^n)$ ó $O(p(n)\times n!)$, con $p(n)$ un polinomio
    - Si es posible: **reordenar** las selecciones de forma que $|C_1| < |C_2| < \cdots < |C_n|$, y así cabe esperar que se explorarán menos caminos.

4 Búsqueda con retroceso: Introducción

– Estimación *a priori* del número de nodos generados:

- Idea: **generar un camino aleatorio** en el árbol del espacio de estados.
- Sea x_i el nodo del camino aleatorio en el nivel i y sea m_i el número de hijos de x_i que satisfacen el predicado acotador A.
- El siguiente nodo del camino aleatorio se obtiene aleatoriamente de entre esos m_i.
- La generación termina en un nodo de respuesta (solución) o en un nodo por el que no se puede seguir (ninguno de sus hijos satisfacen el predicado acotador).
- Si los predicados acotadores son **estáticos** (no cambian en toda la búsqueda; esto no es lo habitual; lo habitual es que se hagan cada vez más restrictivos) y más aún si los nodos de un mismo nivel tienen todos igual grado, entonces:

  El número estimado de nodos que se generará con el algoritmo de búsqueda con retroceso es:

$$m = 1 + m_1 + m_1 m_2 + m_1 m_2 m_3 + \cdots$$

4 Búsqueda con retroceso: Introducción

```
función estimación devuelve entero
variables k,m,r,card:entero;
          nodo:elmto;
          sol:vector[1..n]de elmto
principio
  k:=1; m:=0; r:=1;
  repetir
    card:=0;
    para todo nodo en G(sol,1,k-1) hacer
      sol[k]:=nodo;
      si A(sol,1,k)
        entonces card:=card+1
      fsi
    fpara;
    si card≠0
      entonces
        r:=r*card;
        m:=m+r;
        sol[k]:=eleccAleat(G(sol,1,k-1));
        k:=k+1
    fsi
  hastaQue R(sol,1,k) or (card=0);
  devuelve m
fin
```

4 El problema de las ocho reinas

- Consideraremos el problema más general de colocar n reinas en un tablero de dimensiones $n \times n$, sin que se den jaque.
  - Cada solución se representa por una n-tupla $(x_1,\ldots,x_n)$, en la que x_i es la columna de la i-ésima fila en la que se coloca la i-ésima reina.
  - El espacio de soluciones se reduce a $n!$ elementos teniendo en cuenta que todas ellas han de ser permutaciones de $(1,2,\ldots,n)$, es decir, todas las x_i han de ser distintas.
  - Además, no deben compartir diagonal.

4 El problema de las ocho reinas

- Representación de la información

  Debe permitir interpretar fácilmente la solución:

  ```
  x:vector[1..n]de entero;
  {x[i]=columna de la reina en i-ésima fila}
  ```

- Evaluación del predicado acotador:
  - Utilizaremos una función `buenSitio` que devuelva el valor verdad si la *k*-ésima reina se puede colocar en el valor `x[k]`, es decir, si está en distinta columna y diagonal que las *k*-1 reinas anteriores.
  - Dos reinas están en la misma diagonal ↗ si tienen el mismo valor de "fila+columna", mientras que están en la misma diagonal ↘ si tienen el mismo valor de "fila-columna".

$$
\begin{aligned}
& (f_1 - c_1 = f_2 - c_2) \quad \vee \quad (f_1 + c_1 = f_2 + c_2) \\
\Leftrightarrow \quad & (c_1 - c_2 = f_1 - f_2) \quad \vee \quad (c_1 - c_2 = f_2 - f_1) \\
\Leftrightarrow \quad & |c_1 - c_2| = |f_1 - f_2|
\end{aligned}
$$

4 El problema de las ocho reinas

```
funcion buenSitio(k:entero;
                  x:vector[1..n]de entero)
        devuelve bool
{devuelve verdad si y sólo si se puede colocar
 una reina en la fila k y columna x[k], habiendo
 sido colocadas ya las k-1 reinas anteriores}
variables i:entero; amenaza:bool
principio
  i:=1; amenaza:=falso;
  mq (i<k) and not amenaza hacer
    si x[i]=x[k] or abs(x[i]-x[k])=abs(i-k)
      entonces amenaza:=verdad
      sino i:=i+1
    fsi
  fmq;
  devuelve not amenaza
fin
```

El coste de la función es $O(k\text{-}1)$.

4 El problema de las ocho reinas

Versión recursiva:

```
algoritmo colocarReinas(ent k:entero;
        entsal sol:vector[1..n]de entero)
{sol[1..k-1] están bien colocadas}
variables i:entero
principio
  para i:=1 hasta n hacer
    sol[k]:=i;
    si buenSitio(k,sol)
      entonces
        si k=n
          entonces escribir(sol)
          sino colocarReinas(k+1,sol)
        fsi
    fsi
  fpara
fin
```

```
...
colocarReinas(1,sol);
...
```

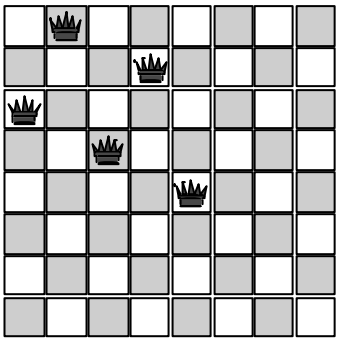
4 El problema de las ocho reinas

Versión iterativa:

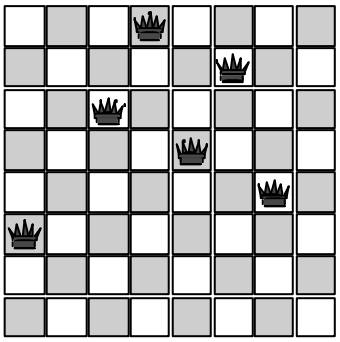
```
algoritmo nReinas(ent n:entero)
variables k:entero; x:vector[1..n]de entero
principio
  x[1]:=0; k:=1;
  mq k>0 hacer {para frenar el último retroceso}
    x[k]:=x[k]+1;
    mq x[k]≤n and not buenSitio(k,x) hacer
      x[k]:=x[k]+1
    fmq;
    si x[k]≤n  {se ha encontrado un buen sitio}
      entonces
        si k=n  {¿es una solución completa?}
          entonces
            escribir(x) {si: escribirla}
          sino
            k:=k+1;
            x[k]:=0 {ir a la siguiente fila}
        fsi
      sino
        k:=k-1  {retroceso}
    fsi
  fmq
fin
```

El problema de las ocho reinas

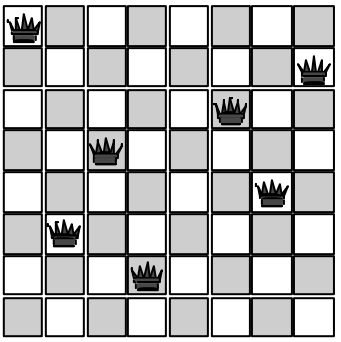
- Estimación del coste:
  - Se realizaron cinco evaluaciones de la función `estimación`, con los siguientes resultados:



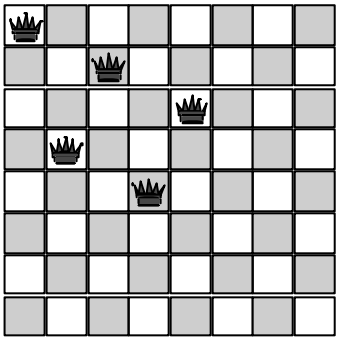
(8,5,4,3,2)→1649



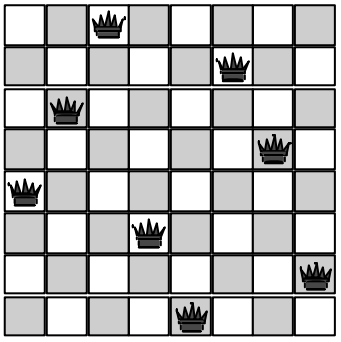
(8,5,3,1,2,1)→749



(8,6,4,2,1,1,1)→1401



(8,6,4,3,2)→1977



(8,5,3,2,2,1,1,1)→2329

Con cada elección se guarda el número de columnas en que es posible colocar la reina y a partir de él se calcula el valor que devuelve la función. Recordar: $m = 1 + m_1 + m_1 m_2 + m_1 m_2 m_3 + \cdots$

4 El problema de las ocho reinas

– La media para las cinco evaluaciones es 1625.

– El número total de nodos del espacio de estados es:

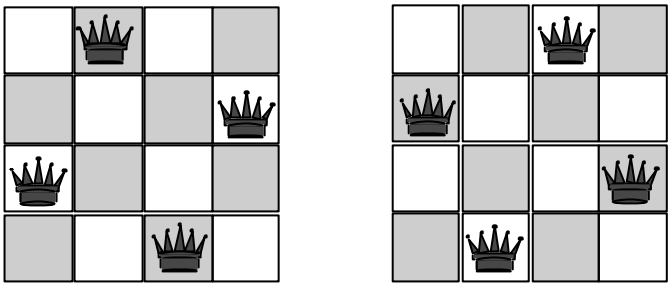
$$1+\sum_{j=0}^{7}\left(\prod_{i=0}^{j}(8-i)\right)=69281$$

– Por tanto, únicamente se recorrería un 2'34% del número total de nodos (si la estimación es acertada).

(En este caso, la estimación es algo optimista pues se puede comprobar que el número de nodos explorados es 2057 y, por tanto, se recorre un 2'97%.)

– Número de soluciones para n=8: 92.

– Mejora adicional: observar que algunas soluciones son simplemente rotaciones o reflexiones de otras.



Para encontrar soluciones "no equivalentes", el algoritmo sólo debe probar con $x[1]=2,3,\ldots,\lceil n/2 \rceil$

El problema de las ocho reinas

- Para *n* mayor:

  la mejora se hace más considerable.

- Por ejemplo, *n*=12:

  - número de permutaciones: 12! = 479.001.600
  - número total de nodos del espacio de estados: 1.302.061.345
  - número de nodos explorados: 856.189

    → se recorre sólo un 0'065%

  (se obtiene la primera solución tras visitar 262 nodos)

4 El problema de las ocho reinas

- Novedad:
  Recientemente se ha encontrado una solución compacta para el problema general de las n reinas en un tablero $n \times n$ (excepto para $n=2$, $n=3$, $n=8$ y $n=9$):

  - Si n es impar ($n=2p-1$) y $n \neq 3$ y $n \neq 9$:
    - Si n no es múltiplo de 3:
      $(2k-1,k)$, $1 \leq k \leq p$
      $(2m,m+p)$, $1 \leq m \leq p-1$
    - Si n es múltiplo de 3 y $n \neq 3$ y $n \neq 9$:
      $(2k,k)$, $1 \leq k \leq p-1$, $k \neq 4$
      $(2m+1,m+p-1)$, $1 \leq m \leq p-2$
      $(n,4)$
      $(8,n-1)$
      $(1,n)$

4 El problema de las ocho reinas

– Si n es par ($n=2p$) y $n\neq 2$ y $n\neq 8$:

- Si not($n\equiv 2$ mód 3):

  $(2k,k)$, $1\leq k\leq p$

  $(2m-1,m+p)$, $1\leq m\leq p$

- Si $n\equiv 2$ mód 3 y $n\neq 2$ y $n\neq 8$:

  $(2k-1,k)$, $1\leq k\leq p$, $k\neq 4$

  $(2m,m+p)$, $1\leq m\leq p-1$

  $(n,4)$

  $(7,n)$

4 El problema de la suma de subconjuntos

- Problema:
  - Dados un conjunto $W=\{w_1,\ldots,w_n\}$ de n números positivos y otro número positivo M, se trata de encontrar todos los subconjuntos de W cuya suma es M.
  - Ejemplo: si $W=\{11,13,24,7\}$ y $M=31$, entonces la solución es $\{11,13,7\}$ y $\{24,7\}$.
- Primera representación de la solución:
  - La solución puede representarse simplemente con los índices de los elementos de W.
  - En el ejemplo: (1,2,4) y (3,4).
  - En general, todas las soluciones son k-tuplas $(x_1,x_2,\ldots,x_k)$, $1\leq k\leq n$.

4 El problema de la suma de subconjuntos

- Restricciones sobre las soluciones (para la primera representación):
  - Explícitas:

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

  - Implícitas:

$$i \neq j \Rightarrow x_i \neq x_j$$

$$\sum_{i=1}^{k} w_{x_i} = M$$

$$x_i < x_{i+1},\ 1 \le i < n$$ (para evitar generar varias instancias de la misma tupla)

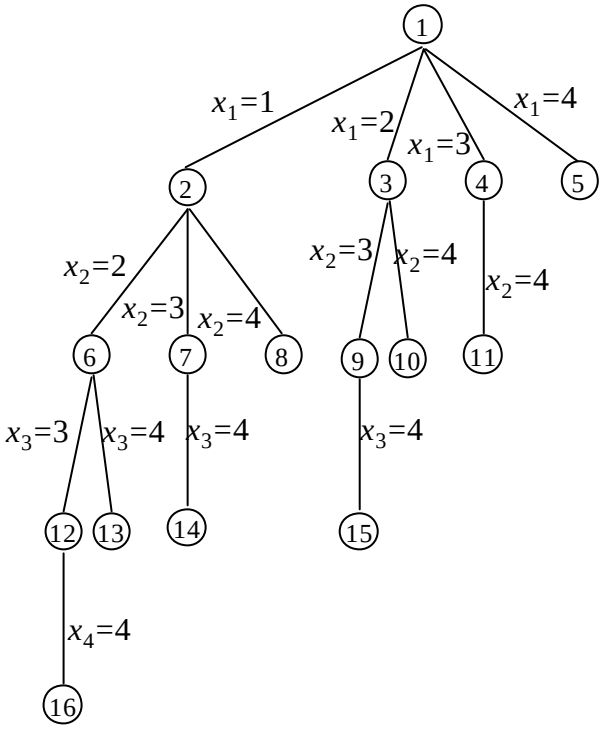
4 El problema de la suma de subconjuntos

- Segunda representación de la solución:
  - Cada solución puede representarse con una n-tupla $(x_1,x_2,\ldots,x_n)$, tal que $x_i \in \{0,1\}$, $1 \leq i \leq n$,

    de forma que:
    - $x_i = 0$ si w_i no se elige y
    - $x_i = 1$ si w_i se elige.
  - En el ejemplo anterior: (1,1,0,1) y (0,0,1,1).
- Conclusión:
  - Pueden existir varias formas de formular un problema, con distintas representaciones de las soluciones (aunque siempre verificando éstas un conjunto de restricciones explícitas e implícitas).
  - En el problema que consideramos, ambas representaciones nos llevan a un espacio de estados que consta de 2^n tuplas.

4 El problema de la suma de subconjuntos

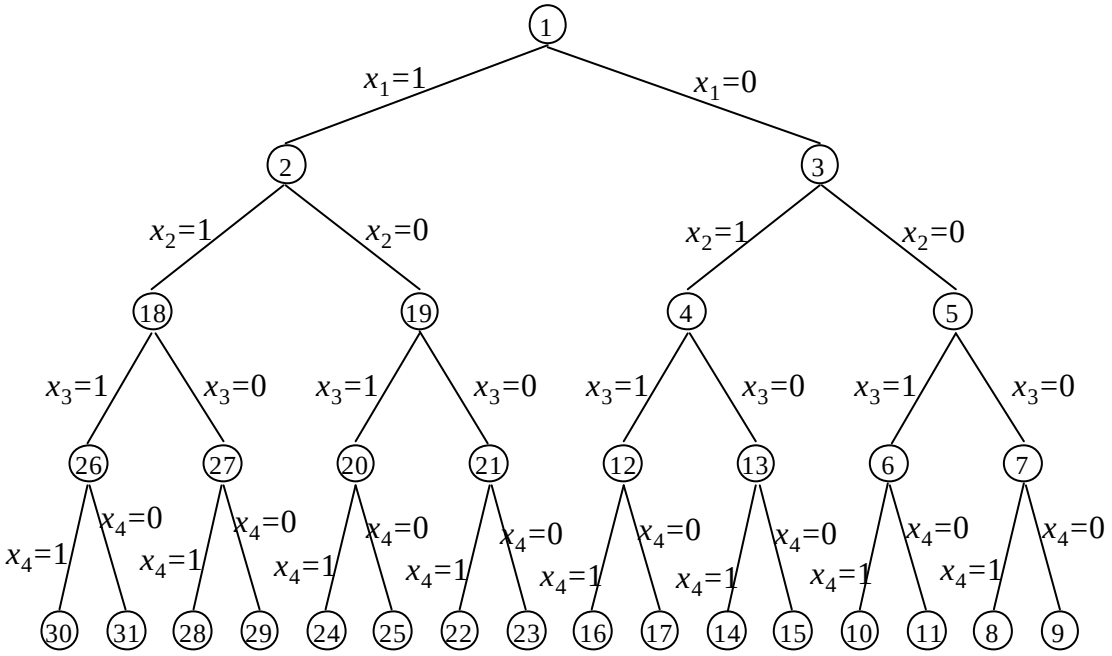
- Arbol del espacio de soluciones (n=4) para la primera representación (tuplas de tamaño variable):
  - Un arco del nivel i al nivel i+1 representa un valor para x_i.
  - El espacio de soluciones está definido por todos los caminos desde la raíz hasta cualquier nodo del árbol.



  - Los nodos se han numerado según un recorrido en anchura (utilizando para ello una cola).

4 El problema de la suma de subconjuntos

- Arbol del espacio de soluciones (n=4) para la segunda representación (tuplas de tamaño fijo):
  - Los arcos del nivel i al nivel i+1 están etiquetados con el valor de x_i (1 ó 0).
  - Todos los caminos desde la raíz a una hoja definen el espacio de soluciones (2^4 hojas que representan las 16 posibles 4-tuplas).



  - Los nodos están numerados de acuerdo con un recorrido de "D-búsqueda" (consiste en explorar primero el último de los nodos añadido a la lista de nodos por explorar, es decir, utilizando una pila).

4 *El problema de la suma de subconjuntos*

- Estudiemos una solución de búsqueda con retroceso basada en la segunda representación (tuplas de tamaño fijo).

  - El elemento x[i] del vector solución toma el valor 1 ó 0 dependiendo de si el número w[i] se incluye o no en la solución.

  - Generación de los hijos de un nodo:
    - para un nodo del nivel *i* el hijo izquierdo corresponde a x[i]=1 y el derecho a x[i]=0.

4 El problema de la suma de subconjuntos

– **Función acotadora**:

La tupla (x[1],…x[k]) sólo puede conducir a una solución si:

$$\sum_{i=1}^{k} w[i]\,x[i] + \sum_{i=k+1}^{n} w[i] \geq M$$

Si además se sabe que los **w[i]** están **ordenados** de forma **no-decreciente**, la tupla (x[1],…x[k]) no puede conducir a una solución si:

$$\sum_{i=1}^{k} w[i]\,x[i] + w[k+1] > M$$

– Es decir, una función acotadora es:

$$B_k(x[1],\ldots,x[k]) = \text{verdad si y sólo si}$$
$$\left(\sum_{i=1}^{k} w[i]\,x[i] + \sum_{i=k+1}^{n} w[i] \geq M\right) \wedge$$
$$\wedge \left(\sum_{i=1}^{k} w[i]\,x[i] + w[k+1] \leq M\right)$$

4 El problema de la suma de subconjuntos

```
algoritmo sumasub(ent s,k,r:entero)
{Encuentra todos los subconjuntos del vector
 global w cuya suma es M.
 Los valores de x[j], que es otro vector global,
 1≤j<k ya han sido calculados.
 s=Σ_{j=1}^{k-1} w[j]*x[j]; r=Σ_{j=k}^{n} w[j].

 Los w[j] están en orden no decreciente.
 Se asume que w[1]≤M y Σ_{i=1}^{n} w[i]≥M.}
principio
  {Generación del hijo izquierdo.
   Nótese que s+w[k]≤M porque B_{k-1}(x[1],...,x[k-1])=verdad }
  x[k]:=1;
  si s+w[k]=M {se ha encontrado un subconjunto}
    entonces escribir(x[1..k])
    sino
      si s+w[k]+w[k+1]≤M
        entonces {B_k(x[1],...,x[k])=verdad}
          sumasub(s+w[k],k+1,r-w[k])
      fsi
  fsi
  ...
```

4 El problema de la suma de subconjuntos

```
  ...
  {Generación del hijo derecho y evaluación de Bk}
  si (s+r-w[k]≥M) and (s+w[k+1]≤M)   {Bk =verdad}
    entonces
      x[k]:=0;
      sumasub(s,k+1,r-w[k])
  fsi
fin
```

Nótese que el algoritmo evita calcular

$$\sum_{i=1}^{k} w[i]x[i] \quad \text{y} \quad \sum_{i=k+1}^{n} w[i]$$

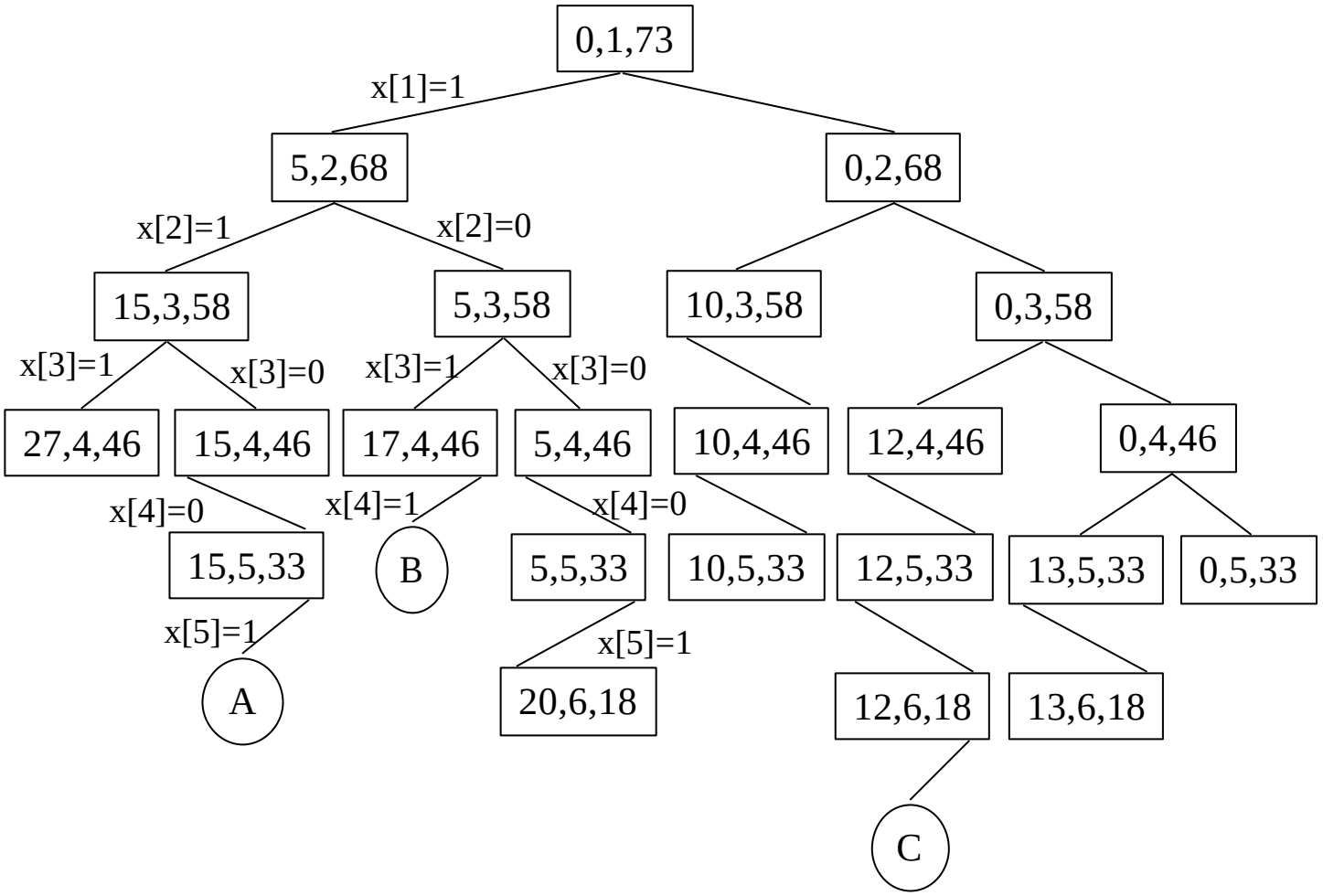
cada vez guardando esos valores en s y r.

La llamada inicial es:

```
...
sumasub(0,1, Σ_{i=1}^{n} w[i])
...
```

4 *El problema de la suma de subconjuntos*

– Ejemplo: n=6, M=30, W=(5,10,12,13,15,18)
Los rectángulos son s,k,r en cada llamada.
A=(1,1,0,0,1); B=(1,0,1,1); C=(0,0,1,0,0,1).
Se construyen 23 nodos (del total de 2^6-1=63)



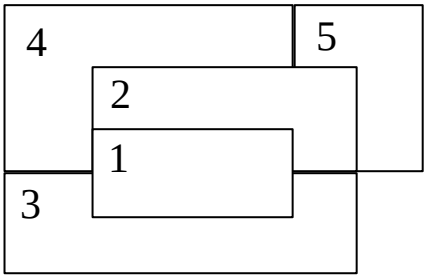
4.4 Coloreado de grafos

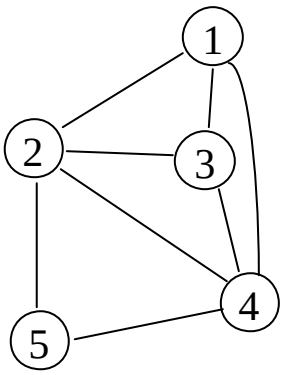
- Problema de decisión:
  - Dados un grafo *G* y un número entero positivo *m*, ¿es *G* ***m*-coloreable**?
  - Es decir, ¿se puede pintar con colores los nodos de *G* de modo que no haya dos vértices adyacentes con el mismo color y se usen sólo *m* colores?

- Problema de optimización:
  - Dado un grafo *G*, ¿cuál es su **número cromático**?
  - Es decir, ¿cuál es el menor número *m* de colores con el que se puede colorear *G*?

4 Coloreado de grafos

- Un subproblema muy famoso:
  - Dado un mapa, ¿pueden pintarse sus regiones (autonomías, países, o lo que sea) de tal forma que no haya dos regiones adyacentes de igual color y no se empleen más de 4 colores?





  - Cada región se modela con un nodo y si dos regiones son adyacentes sus correspondientes nodos se conectan con un arco.
  - Así se obtiene siempre un grafo “plano” (puede dibujarse en un plano sin cruzar sus arcos).
  - El mapa de la figura requiere 4 colores.
  - Desde hace muchos años se sabía que 5 colores eran suficientes para pintar cualquier mapa, pero no se había encontrado ningún mapa que requiriera más de 4.
  - Recientemente, después de varios cientos de años, se ha demostrado que 4 colores siempre son suficientes.

4 Coloreado de grafos

- El problema que consideraremos aquí:
  - Dado un grafo cualquiera, determinar **todas** las formas posibles en las que puede pintarse utilizando no más de *m* colores.
  - Representación elegida: matriz de adyacencias.

```
tipo grafo = vector[1..n,1..n] de bool
```

  - Justificación de la elección: sólo necesitaremos saber si un arco existe o no.
  - Representación de los colores: enteros de 1 a *m*.

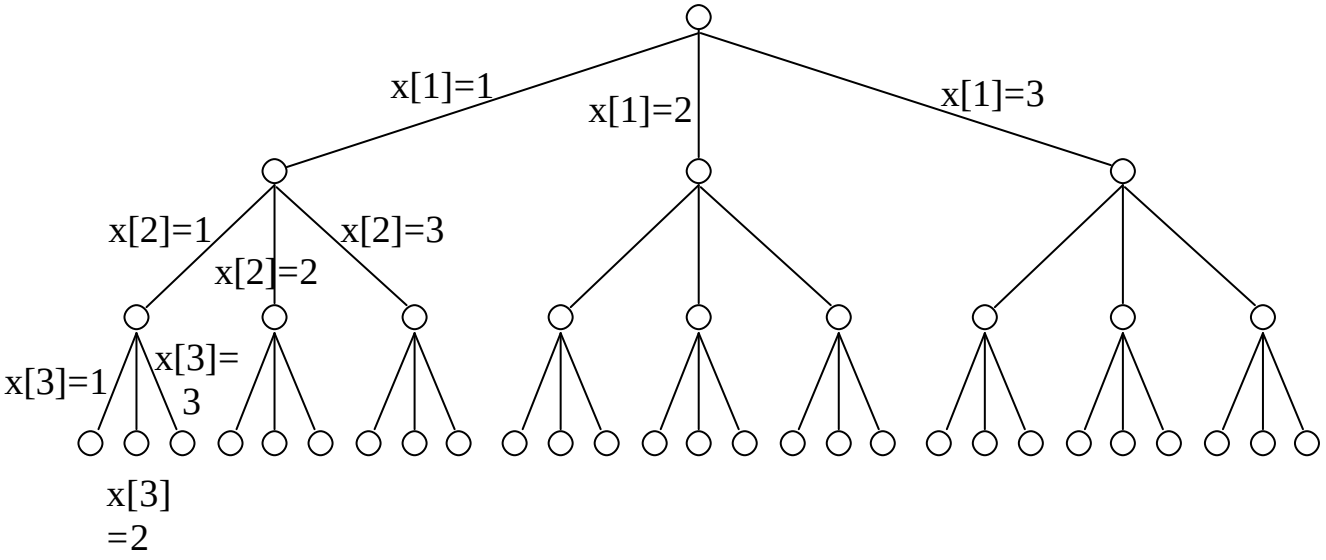
```
tipo color = 0..m
```

  - Representación de la solución: vector de colores.

```
tipo sol = vector[1..n] de color
```

4.4 Coloreado de grafos

- Espacio de estados para n=3 y m=3.



Grado m y altura n.

Cada nodo de nivel i tiene m hijos que corresponden a las m posibles asignaciones a $x[i]$, $1 \leq i \leq n$.

4.4 Coloreado de grafos

- Solución de búsqueda con retroceso:

```
algoritmo m_coloreado(ent k:entero;
                      entsal x:sol)
{Se usa una variable global g de tipo grafo.
 En x se tiene la parte de la solución ya calculada
 (es decir, hasta x[k-1]) y k es el índice del
 siguiente vértice al que se va a asignar color.}
principio
  repetir
  {generar todos los colores 'legales' para x[k]}
    siguienteValor(x,k); {x[k]:=un color legal}
    si x[k]≠0  {se ha encontrado un color legal}
      entonces
        si k=n
          entonces escribir(x)
          sino m_coloreado(k+1,x)
        fsi
    fsi
  hastaQue x[k]=0
fin
```

4 Coloreado de grafos

```
algoritmo siguienteValor(entsal x:sol;
                         ent k:entero)
{x[1]…x[k-1] tienen colores asociados de forma
 que todos los vértice adyacentes tienen distinto
 color.
 x[k] tiene el anterior color para el que se ha
 probado (0 si no se ha probado con ninguno).
 Se calcula el siguiente color para x[k]
 diferente del de todos los vértices adyacentes
 a k (0 si no hay ninguno).}
variables encontrado:booleano; j:entero
principio
  repetir
    x[k]:=(x[k]+1) mod (m+1); {siguiente color}
    si x[k]≠0
      entonces
        encontrado:=verdad;
        j:=1;
        mq (j≤n) and encontrado hacer
          si g[k,j] and (x[k]=x[j])
            entonces encontrado:=falso
            sino j:=j+1
          fsi
        fmq
    fsi
  hastaQue (x[k]=0) or encontrado
fin
```

4 Coloreado de grafos

- La llamada inicial es:

```
...
{g contiene la matriz de adyacencia del grafo}
para i:=1 hasta n hacer
  x[i]:=0
fpara;
m_coloreado(1,x);
...
```

4.4 Coloreado de grafos

- Cota superior del coste temporal:
  - Número de nodos internos del árbol del espacio de estados:

$$\sum_{i=0}^{n-1} m^i$$

  - Coste de '`siguienteValor`' para cada nodo interno:

$$O(mn)$$

  - Cota del tiempo total:

$$\sum_{i=1}^{n} m^i n = n\left(m^{n+1}-1\right)/\left(m-1\right) = O\left(nm^n\right)$$

- Recordar que ya vimos una heurística voraz muy eficiente…

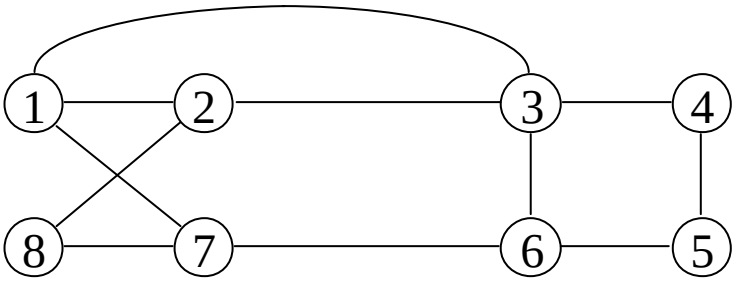
4.4 Ciclos hamiltonianos

- Problema: encontrar todos los ciclos hamiltonianos de un grafo.
  - Sea $G=(V,A)$ un grafo conexo con n vértices.
  - Un ciclo hamiltoniano es un camino que visita una vez cada vértice y vuelve al vértice inicial.

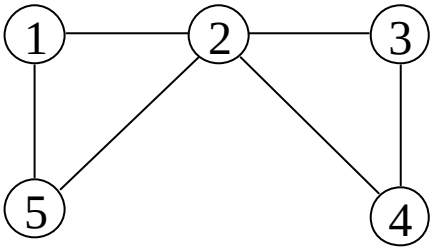
    Es decir, $v_1 v_2 \ldots v_{n+1}$ tal que:
    - $v_i \in V$, $i=1,\ldots,n+1$,
    - $(v_i, v_{i+1}) \in A$, $i=1,\ldots,n$,
    - $v_1 = v_{n+1}$,
    - $v_i \neq v_j$, $\forall i,j=1,\ldots,n$ tales que $i \neq j$.

4.4 Ciclos hamiltonianos

- Ejemplos:



Hamiltoniano: 1-2-8-7-6-5-4-3-1



No contiene ningún hamiltoniano.

4.4 *Ciclos hamiltonianos*

- No se conoce un algoritmo eficiente para resolver el problema.

- Nótese la relación entre el problema del cálculo de un hamiltoniano y el problema del viajante de comercio (para el caso de un grafo con todas las distancias entre vértices iguales)

  – Recordar que ya vimos una heurística voraz muy eficiente pero subóptima para el problema del viajante de comercio.

  – Recordar que vimos también la solución de programación dinámica y comparamos su eficiencia con la solución de “fuerza bruta”.

4.4 Ciclos hamiltonianos

- Solución de búsqueda con retroceso:
  - En el vector solución $(x_1,\ldots,x_n)$, x_i representa el vértice visitado en i-ésimo lugar en el ciclo.
  - Cálculo de los posibles valores para x_k si $x_1,\ldots,x_{k-1}$ ya tienen valores asignados:
    - $k=1$: x_1 puede ser cualquiera de los n vértices, pero para evitar escribir el mismo ciclo n veces obligamos que $x_1=1$;
    - $1<k<n$: x_k puede ser cualquier vértice distinto de $x_1,\ldots,x_{k-1}$ y conectado por un arco con x_{k-1}.
    - $k=n$: x_n sólo puede ser el vértice que queda sin visitar y debe estar conectado por sendos arcos con x_1 y x_{n-1}.

4 Ciclos hamiltonianos

– Representación elegida: matriz de adyacencias.

```
tipo grafo = vector[1..n,1..n] de bool
```

– Justificación de la elección: sólo necesitaremos saber si un arco existe o no.

– Representación de la solución: vector de vértices.

```
tipo sol = vector[1..n] de 1..n
```

```
algoritmo siguienteValor(entsal x:sol;
                         ent k:entero)
{x[1],...,x[k-1] es un camino de k-1 vértices
 distintos.
 Si x[k]=0, no se ha asignado ningún vértice
 todavía a x[k].
 Al terminar, x[k] toma el valor del siguiente
 (en orden ascendente) vértice tal que:
 (1) no aparece ya en x[1],...,x[k-1], y
 (2) está conectado por un arco a x[k-1].
 Además, si k=n, se debe exigir a x[k] que:
 (3) está conectado por un arco a x[1].
 Si no hay tal vértice, x[k]=0.}
...
```

4 Ciclos hamiltonianos

```
...
variables encontrado:booleano; j:entero
principio
  repetir
    x[k]:=(x[k]+1) mod (n+1);
    si x[k]≠0
      entonces
        encontrado:=falso;
        si g[x[k-1],x[k]]
          entonces
            j:=1;
            encontrado:=verdad;
            mq (j≤k-1) and encontrado hacer
              si x[j]=x[k]
                entonces encontrado:=falso
                sino j:=j+1
              fsi
            fmq;
            si encontrado entonces
              si (k=n) and not g[x[n],1]
                entonces encontrado:=falso
              fsi
            fsi
        fsi
    fsi
  hastaQue (x[k]=0) or encontrado
fin
```

4 Ciclos hamiltonianos

```
algoritmo hamiltoniano(ent k:entero;
                       entsal x:sol)
{Se usa una variable global g de tipo grafo.
 Cálculo de los ciclos hamiltonianos de un grafo
 mediante búsqueda con retroceso.
 En x se tiene la parte de la solución ya calculada
 (es decir, hasta x[k-1]) y k es el índice del
 siguiente vértice del ciclo que se va a asignar.}
principio
  repetir
  {generar todos los valores 'legales' para x[k]}
    siguienteValor(x,k);
    si x[k]≠0
      entonces
        si k=n
          entonces
            escribir(x,'1')
          sino hamiltoniano(k+1,x)
        fsi
    fsi
  hastaQue x[k]=0
fin
```

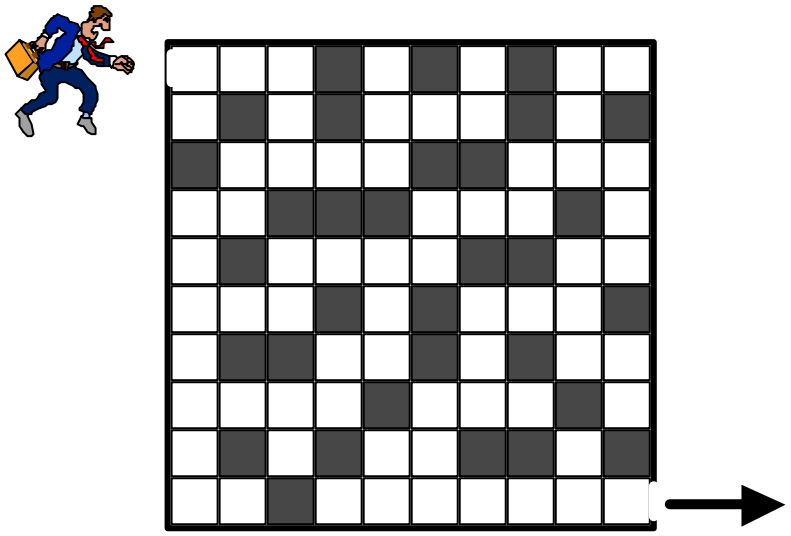
4 Ciclos hamiltonianos

- La llamada inicial es:

```
...
{g contiene la matriz de adyacencia del grafo}
x[1]:=1;
para i:=2 hasta n hacer
  x[i]:=0
fpara;
hamiltoniano(2,x);
...
```

4A *Atravesar un laberinto*

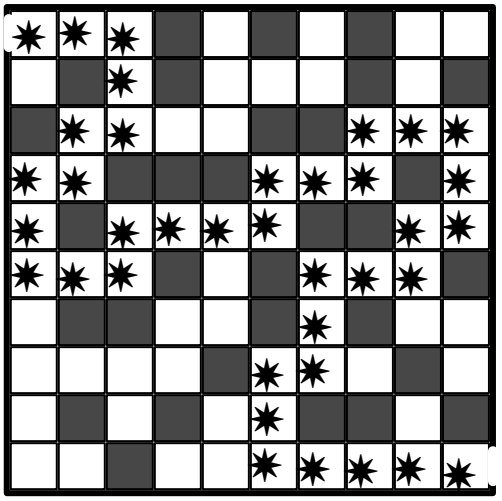
- Problema:
  - Nos encontramos en una entrada de un laberinto y debemos intentar atravesarlo.



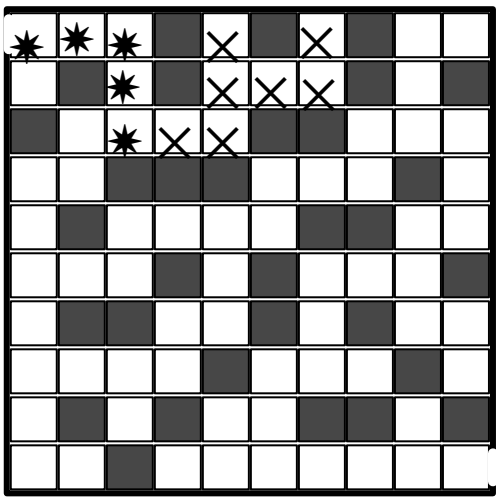
  - Representación: matriz de dimensión $n \times n$ de casillas marcadas como *libre* u *ocupada* por una pared.
  - Es posible pasar de una casilla a otra moviéndose sólamente en vertical u horizontal.
  - Se debe ir de la casilla (1,1) a la casilla (n,n).

4A *Atravesar un laberinto*

– Diseñaremos un algoritmo de búsqueda con retroceso de forma que se marcará en la misma matriz del laberinto un camino solución (si existe).



– Si por un camino recorrido se llega a una casilla desde la que es imposible encontrar una solución, hay que volver atrás y buscar otro camino.



– Además hay que marcar las casillas por donde ya se ha pasado para evitar meterse varias veces en el mismo callejón sin salida, dar vueltas alrededor de columnas…

4.4 Atravesar un laberinto

- Estructura de datos:

```
tipos
  casilla = (libre,pared,camino,imposible)
  laberinto = vector[1..n,1..n] de casilla
```

- Solución de búsqueda con retroceso:

```
algoritmo buscarCamino(entsal lab:laberinto;
                       sal éxito:booleano)
principio
  ensayar(1,1,lab,éxito)
fin
```

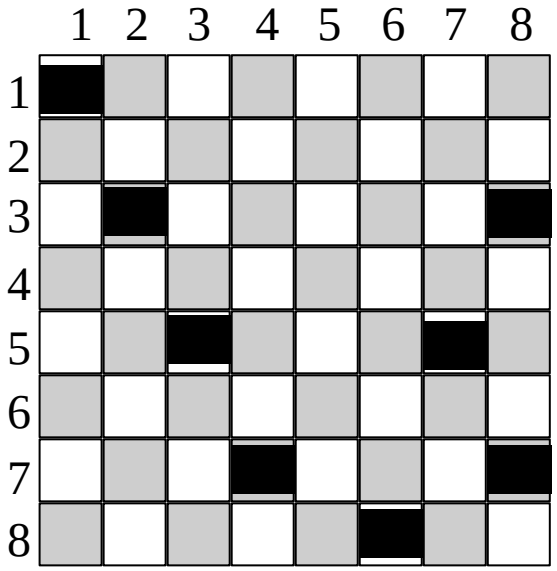
```
algoritmo ensayar(ent x,y:entero;
                  entsal lab:laberinto;
                  sal encontrado:booleano)
principio
  si (x<1)∨(x>n)∨(y<1)∨(y>n)
    entonces {posición fuera del laberinto}
      encontrado:=falso
    sino
      si lab[x,y]≠libre
        entonces encontrado:=falso
        sino
...
```

4 Atravesar un laberinto

```
...
          lab[x,y]:=camino;
          si (x=n)∧(y=n)
            entonces {se ha encontrado una soluc.}
              encontrado:=verdad
            sino
              ensayar(x+1,y,lab,encontrado);
              si not encontrado
                entonces
                  ensayar(x,y+1,lab,encontrado)
              fsi;
              si not encontrado
                entonces
                  ensayar(x-1,y,lab,encontrado)
              fsi;
              si not encontrado
                entonces
                  ensayar(x,y-1,lab,encontrado)
              fsi;
              si not encontrado
                entonces
                  lab[x,y]:=imposible
              fsi
          fsi
      fsi
  fsi
fin
```

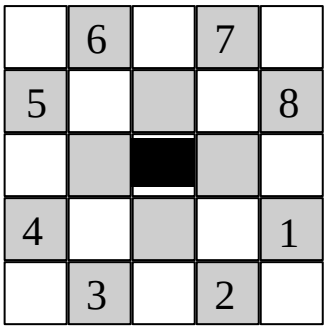
4 El recorrido del caballo de ajedrez

- Problema:
  - Se trata de encontrar una sucesión de movimientos “legales” de un caballo de ajedrez de forma que éste pueda visitar todos y cada uno de los escaques (cuadros) de un tablero sin repetir ninguno.



4 El recorrido del caballo de ajedrez

– Movimientos “legales”:




```
variables dx,dy:vector[1..8] de entero
...
dx:=[2,1,-1,-2,-2,-1,1,2]
dy:=[1,2,2,1,-1,-2,-2,-1]
```

– Estructura de datos: matriz de naturales

- inicialmente: todos cero
- al terminar: cada componente guarda el número de orden en que se visitó el escaque correspondiente

```
variable
  tab:vector[1..n,1..n] de entero
```

4 El recorrido del caballo de ajedrez

```
algoritmo ensaya(ent i,x,y:entero;
                 sal éxito:booleano)
{Ensaya el movimiento al i-ésimo escaque desde
 el x,y. Si el movimiento es posible y tras él se
 puede seguir moviendo hasta encontrar la solución
 entonces éxito devuelve verdad, si no falso.}
variables k,u,v:entero
principio
  k:=0;
  repetir {se ensaya con los 8 mov. posibles}
    k:=k+1; éxito:=falso;
    u:=x+dx[k]; v:=y+dy[k];
    si (1≤u)∧(u≤n)∧(1≤v)∧(v≤n) entonces
      si tab[u,v]=0 entonces
        tab[u,v]:=i;
        si i<n*n
          entonces
            ensaya(i+1,u,v,éxito);
            si not éxito
              entonces tab[u,v]:=0
            fsi
          sino éxito:=verdad
        fsi
      fsi
    fsi
  hastaQue éxito or (k=8)
fin
```

4 El recorrido del caballo de ajedrez

```
algoritmo caballo
constante n=8
variables dx,dy:vector[1..8] de entero;
          tab:vector[1..n,1..n] de entero;
          i,j:entero; éxito:booleano
principio
  dx[1]:=2; dx[2]:=1; dx[3]:=-1; dx[4]:=-2;
  dx[5]:=-2; dx[6]:=-1; dx[7]:=1; dx[8]:=2;
  dy[1]:=1; dy[2]:=2; dy[3]:=2; dy[4]:=1;
  dy[5]:=-1; dy[6]:=-2; dy[7]:=-2; dy[8]:=-1;
  para i:=1 hasta n hacer
    para j:=1 hasta n hacer
      tab[i,j]:=0
    fpara;
  fpara;
  escribir('Introduce i inicial:'); leer(i);
  escribir('Introduce j inicial:'); leer(j);
  tab[i,j]:=1;
  ensaya(2,i,j,éxito);
...
```

4 El recorrido del caballo de ajedrez

```
...
  si éxito
    entonces
      para i:=1 hasta n hacer
        para j:=1 hasta n hacer
          escribir(tab[i,j]);
        fpara;
        escribirLínea
      fpara
    sino
      escribir('No hay solución')
  fsi
fin
```

4 El recorrido del caballo de ajedrez

- Resultado de la ejecución:

```
Introduce i inicial: 1
Introduce j inicial: 1
    1   60   39   34   31   18    9   64
   38   35   32   61   10   63   30   17
   59    2   37   40   33   28   19    8
   36   49   42   27   62   11   16   29
   43   58    3   50   41   24    7   20
   48   51   46   55   26   21   12   15
   57   44   53    4   23   14   25    6
   52   47   56   45   54    5   22   13
```

- Inconveniente de esta solución:
  su ineficiencia

(7'24" en un 68040LC 50/25 MHz)
(se visitan 8.250.733 nodos)

4 El recorrido del caballo de ajedrez

- Una heurística voraz:
  - En cada escaque, seleccionar el siguiente a visitar con la regla:

    Se elige aquel escaque no visitado desde el cual se puede acceder a un menor número de escaques no visitados.

  - La heurística se basa en la idea: si ahora tenemos oportunidad de desplazarnos a un escaque "muy aislado" debemos hacerlo, pues más adelante será más difícil llegar a él de nuevo.

  - El algoritmo resultante **no** es de búsqueda con retroceso.

4 El recorrido del caballo de ajedrez

```
función accesibles(ent x,y:entero)
        devuelve entero
{Devuelve el número de escaques no visitados
 accesibles desde x,y.}
variables k,u,v,num:entero
principio
  num:=0;
  para k:=1 hasta 8 hacer
    u:=x+dx[k];
    v:=y+dy[k];
    si (1≤u)∧(u≤n)∧(1≤v)∧(v≤n)
      entonces
        si tab[u,v]=0
          entonces num:=num+1
        fsi
    fsi
  fpara
  devuelve num
fin
```

4 El recorrido del caballo de ajedrez

```
algoritmo caballo
constante n=8
variables dx,dy:vector[1..8] de entero;
          tab:vector[1..n,1..n] de entero;
          x,y,i,j,k,kk,u,v,num,menor:entero;
          parar:booleano
principio
  dx[1]:=2; dx[2]:=1; dx[3]:=-1; dx[4]:=-2;
  dx[5]:=-2; dx[6]:=-1; dx[7]:=1; dx[8]:=2;
  dy[1]:=1; dy[2]:=2; dy[3]:=2; dy[4]:=1;
  dy[5]:=-1; dy[6]:=-2; dy[7]:=-2; dy[8]:=-1;
  para i:=1 hasta n hacer
    para j:=1 hasta n hacer
      tab[i,j]:=0
    fpara;
  fpara;
  escribir('Introduce x inicial:'); leer(x);
  escribir('Introduce y inicial:'); leer(y);
  tab[x,y]:=1;
  i:=1;
  parar:=falso;
...
```

4 El recorrido del caballo de ajedrez

```
...
  mq (i<n*n) and not parar hacer
    i:=i+1; menor:=9;
    para k:=1 hasta 8 hacer
      u:=x+dx[k]; v:=y+dy[k];
      si (1≤u)∧(u≤n)∧(1≤v)∧(v≤n) entonces
        si tab[u,v]=0 entonces
          num:=accesibles(u,v);
          si num<menor entonces
            menor:=num;
            kk:=k
          fsi
        fsi
      fsi
    fpara;
    si menor=9
      entonces
        parar:=verdad
      sino
        x:=x+dx[kk];
        y:=y+dy[kk];
        tab[x,y]:=i
    fsi
  fmq;
...
```

4 El recorrido del caballo de ajedrez

```
...
  si not parar
    entonces
      para i:=1 hasta n hacer
        para j:=1 hasta n hacer
          escribir(tab[i,j])
        fpara;
        escribirLínea
      fpara
    sino
      escribir('No encuentro solución')
  fsi
fin
```

4 *El recorrido del caballo de ajedrez*

- Resultado de la ejecución:

```
Introduce x inicial: 1
Introduce y inicial: 1
    1   34    3   18   49   32   13   16
    4   19   56   33   14   17   50   31
   57    2   35   48   55   52   15   12
   20    5   60   53   36   47   30   51
   41   58   37   46   61   54   11   26
    6   21   42   59   38   27   64   29
   43   40   23    8   45   62   25   10
   22    7   44   39   24    9   28   63
```

- Mucho más eficiente: $\Theta(n^2)$

(menos de un segundo en el mismo computador)

(se visitan 64 nodos)

4 El recorrido del caballo de ajedrez

- Pero la heurística voraz no funciona en todos los casos:
  - Si n=5:

```
Introduce x inicial: 1
Introduce y inicial: 3
No encuentro solución
```

(se visitan 17 nodos)

Sin embargo, si se ejecuta el algoritmo de búsqueda con retroceso para n=5:

```
Introduce i inicial: 1
Introduce j inicial: 3
   25   14    1    8   19
    4    9   18   13    2
   15   24    3   20    7
   10    5   22   17   12
   23   16   11    6   21
```

(se visitan 365.421 nodos)

(18” de ejecución en el mismo computador)

4 El recorrido del caballo de ajedrez

- Solución:
  - Mejorar el algoritmo de búsquedad con retroceso cambiando el orden de ensayo de las casillas accesibles desde una dada.

  - En lugar de ensayar de forma consecutiva en el sentido de las agujas del reloj,
    - se ensayará en primer lugar la casilla desde la que se pueda acceder al menor número de casillas no visitadas;
    - si desde esa casilla no se llega a una solución, se intentará con la casilla con siguiente menor número de casillas accesibles no visitadas; etc.

  - Para poder implementar el retroceso:
    - cada vez que se llegue a una casilla se almacenan los movimientos posibles en una **cola de movimientos con prioridades**, donde la prioridad de un movimiento es el número de casillas accesibles no visitadas tras realizar ese movimiento

4 El recorrido del caballo de ajedrez

- Colas de movimientos con prioridades:

```
módulo colasMovPri
  exporta tipo mov = registro
                       valor:1..8;
                       peso:0..8
                     freg
    {'valor' es el movimiento según sentido de
     las agujas del reloj;
     'peso' es el nº de casillas no visitadas
     accesibles si se realiza el movimiento}
    función menor(m1,m2:mov) dev booleano
    principio
      devuelve m1.peso<m2.peso
    fin
    tipo cmp {tipo opaco: cola de mov. con
              prioridades; un mov. m1 tiene
              prioridad sobre otro m2 si
              menor(m1,m2)=verdad}
    algoritmo creaVacía(sal c:cmp)
    algoritmo añadir(e/s c:cmp; ent m:mov)
    función min(c:cmp) devuelve mov
    algoritmo eliminarMin(e/s c:cmp)
    función esVacía(c:cmp) dev booleano
  implementación ...
fin
```

4 El recorrido del caballo de ajedrez

```
algoritmo ensaya(ent i,x,y:entero;
                 sal éxito:booleano)
{...}
variables k,u,v:entero;
          m:mov; cola:cmp
principio
  {se almacenan los movimientos posibles}
  creaVacía(cola);
  para k:=1 hasta 8 hacer
    u:=x+dx[k]; v:=y+dy[k];
    si (1≤u)∧(u≤n)∧(1≤v)∧(v≤n) entonces
      si tab[u,v]=0 entonces
        m.valor:=k; m.peso:=accesibles(u,v);
        añadir(cola,m)
      fsi
    fsi
  fpara;
...
```

4 El recorrido del caballo de ajedrez

```
...
  {se ensaya por orden de menor a mayor nº de
   casillas accesibles no visitadas}
  éxito:=falso;
  mq not esVacía(cola) and not éxito hacer
    m:=min(cola); eliminarMin(cola);
    k:=m.valor;
    u:=x+dx[k]; v:=y+dy[k];
    tab[u,v]:=i;
    si i<n*n
      entonces
        ensaya(i+1,u,v,éxito);
        si not éxito
          entonces tab[u,v]:=0
        fsi
      sino éxito:=verdad
    fsi
  fmq
fin
```

4 El recorrido del caballo de ajedrez

- El algoritmo resultante es mucho más eficiente que el original.
  - Y ahora sí encuentra la solución para el caso n=5:

```
Introduce i inicial: 1
Introduce j inicial: 3
   23    6    1   16   21
   12   17   22    7    2
    5   24   11   20   15
   10   13   18    3    8
   25    4    9   14   19
```

(se visitan 1.734 nodos, en menos de un segundo)

4 El problema de la mochila 0-1

- Recordar…
  - Se tienen n objetos y una mochila.
  - El objeto i tiene peso p_i y la inclusión del objeto i en la mochila produce un beneficio b_i.
  - El objetivo es llenar la mochila, de capacidad C, de manera que se maximice el beneficio.

$$\text{maximizar} \quad \sum_{1 \le i \le n} b_i x_i$$

$$\text{sujeto a} \quad \sum_{1 \le i \le n} p_i x_i \le C$$

$$\text{con} \quad x_i \in \{0,1\},\ b_i > 0,\ p_i > 0,\ 1 \le i \le n$$

- Soluciones a problemas parecidos:
  - Mochila con objetos fraccionables, es decir, $0 \le x_i \le 1$, $1 \le i \le n$: solución voraz.
  - Mochila 0-1 pero siendo los pesos, beneficios y capacidad de la mochila números naturales: solución de programación dinámica.

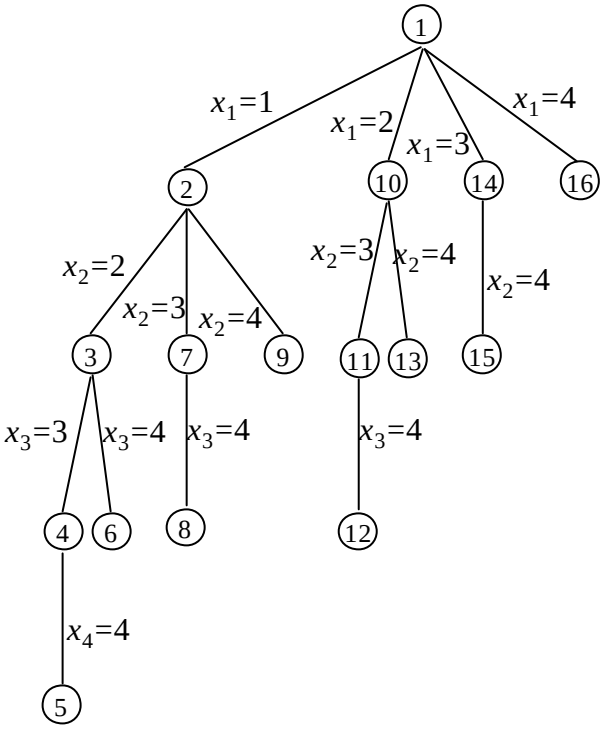
→ Niguna de las dos soluciones funciona en el caso general de la mochila 0-1.

4 El problema de la mochila 0-1

- Espacio de soluciones:
  - 2^n modos de asignar los valores 0 ó 1 a las x_i.
  - Dos formas de representar la solución: tuplas de tamaño fijo o variable.

  - Tuplas de tamaño variable:

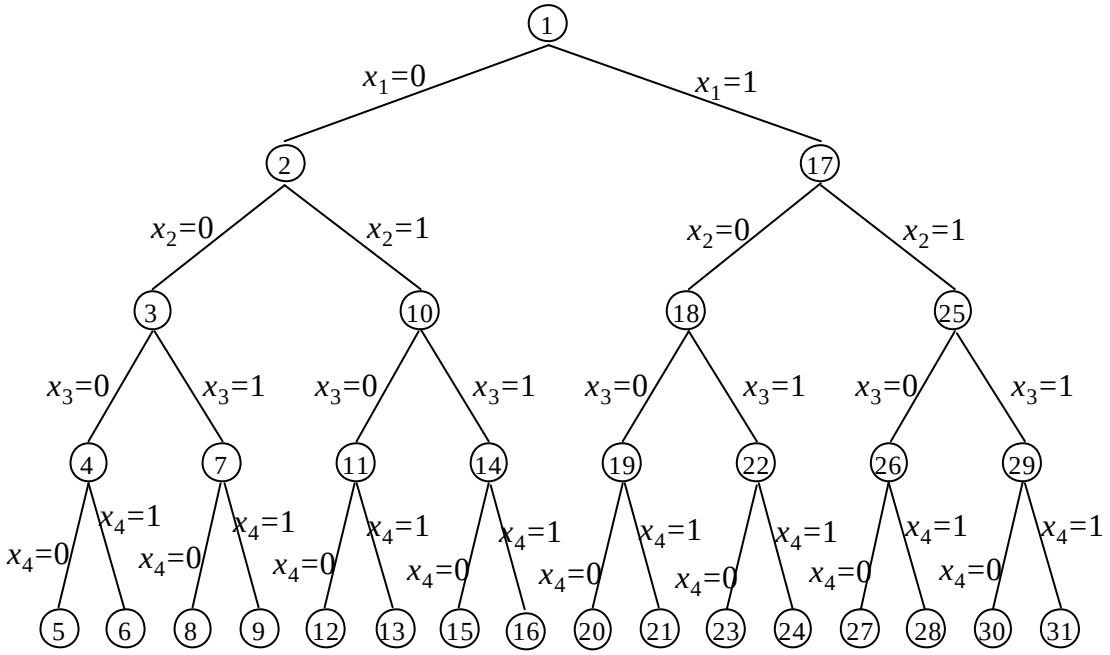
    x_i=objeto introducido en *i*-ésimo lugar



4 El problema de la mochila 0-1

– Tuplas de tamaño fijo:

x_i=0 si el objeto *i*-ésimo no se introduce
x_i=1 si el objeto se introduce



Elegimos ésta última representación.

4 El problema de la mochila 0-1

- Elección de funciones acotadoras: intentar **podar** ramas que no puedan producir soluciones mejores que la disponible actualmente
  - se llama “poda basada en el coste de la mejor solución en curso”
  - calcular una cota superior del valor de la mejor solución posible al expandir un nodo y sus descendientes
  - si esa cota es menor o igual que el valor de la mejor solución disponible hasta el momento, no expandir ese nodo
  - ¿cómo calcular esa cota superior?
    - suponer que en el nodo actual ya se han determinado x_i, $1 \le i \le k$;
    - relajar el requisito de integridad: $x_i \in \{0,1\}$, $k+1 \le i \le n$ se sustituye por $0 \le x_i \le 1$, $k+1 \le i \le n$
    - aplicar el algoritmo voraz

4 El problema de la mochila 0-1

```
constante n=... {número de objetos}
tipo vectReal=vector[1..n] de real
```

```
{Pre: ∀i∈1..n:peso[i]>0 ∧
 ∀i∈1..n-1:benef[i]/peso[i]≥benef[i+1]/peso[i+1]}
función cota(benef,peso:vectReal;
             cap,ben:real; obj:entero)
        devuelve real
{cap=capacidad aún libre de la mochila;
 ben=beneficio actual;
 obj=índice del primer objeto a considerar}
principio
  si obj>n or cap=0.0
    entonces devuelve ben
    sino
      si peso[obj]>cap
        entonces
          dev ben+cap/peso[obj]*benef[obj]
        sino
          dev cota(benef,peso,cap-peso[obj],
                   ben+benef[obj],obj+1)
      fsi
  fsi
fin
```

4 El problema de la mochila 0-1

```
tipo solución=vector[1..n] de 0..1
```

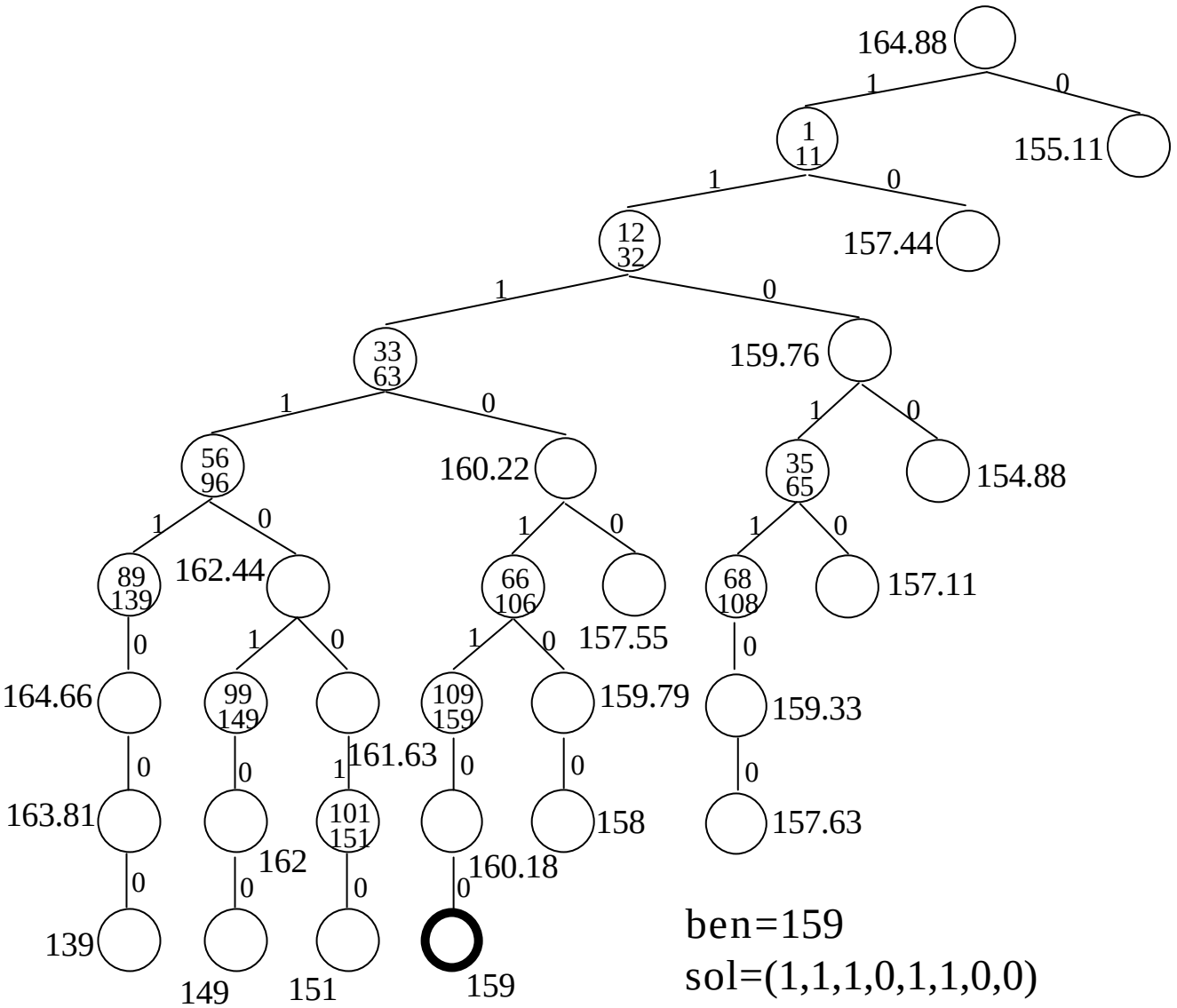
```
{variables globales:
 benef,peso:vectReal; cap:real}
algoritmo búsqueda(ent solAct:solución;
                   ent benAct,pesAct:real;
                   ent obj:entero;
                   e/s sol:solución;
                   e/s ben:real)
variable decisión:0..1
principio
  para decisión:=0 hasta 1 hacer
    solAct[obj]:=decisión;
    benAct:=benAct+decisión*benef[obj];
    pesAct:=pesAct+decisión*peso[obj];
    si pesAct≤cap and ben<cota(benef,peso,
       cap-pesAct,benAct,obj+1) entonces
      si obj=n
        entonces si benAct>ben entonces
                   sol:=solAct; ben:=benAct
                 fsi
        sino búsqueda(solAct,benAct,pesAct,
                      obj+1,sol,ben)
      fsi
    fsi
  fpara
fin
```

4 El problema de la mochila 0-1

```
algoritmo mochila01(ent benef,peso:vectReal;
                    ent cap:real;
                    sal sol:solución;
                    sal ben:real)
variables obj:entero; solAct:solución
principio
  para obj:=1 hasta n hacer
    solAct[obj]:=0; sol[obj]:=0
  fpara;
  ben:=0.0;
  búsqueda(solAct,0.0,0.0,1,sol,ben)
fin
```

El problema de la mochila 0-1

- Ejemplo:
  - benef=(11,21,31,33,43,53,55,65)
  - peso=(1,11,21,23,33,43,45,55)
  - cap=110
  - n=8



54 El problema de la mochila 0-1

– De los 2^9-1 nodos del espacio de estados, sólo se generaron 33.

– Se podía haber reducido a 26 simplemente sustituyendo la condición

```
ben < cota(...)
```

en el algoritmo “búsqueda” por:

```
ben < ⌊cota(...)⌋
```

4 *El problema de la mochila 0-1*

- Hasta ahora hemos visto algoritmos de búsqueda con retroceso basados en árbol de espacio de estados **estático**.

- Se pueden diseñar algoritmos basados en árboles de espacio de estados **dinámicos**.

El problema de la mochila 0-1

- Solución del problema de la mochila 0-1 basada en un árbol dinámico:
  - Resolver el problema sin la restricción de integridad (con el algoritmo voraz):

$$\text{maximizar} \quad \sum_{1 \le i \le n} b_i x_i$$

$$\text{sujeto a} \quad \sum_{1 \le i \le n} p_i x_i \le C \qquad (*)$$

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

    - Si la solución obtenida es entera (0-1), también es óptima para el problema 0-1.
    - Si no es entera, existe exactamente un x_i tal que $0<x_i<1$.

      Se parte el espacio de soluciones en dos subespacios: en uno (subárbol izquierdo) $x_i=0$ y en otro (subárbol derecho) $x_i=1$.
  - En general, en cada nodo del árbol, se usa el algoritmo voraz para resolver el problema (*) con las restricciones añadidas correspondientes a las asignaciones ya realizadas a lo largo del camino desde la raíz al nodo.
    - Si la solución es entera, ya se tiene el óptimo para ese nodo.
    - Si no lo es, existe exactamente un x_i tal que $0<x_i<1$, etc.

4 Reconstrucción de puntos a partir de las distancias

- Cálculo de las distancias a partir de un conjunto de puntos:
  - Se tienen n puntos distintos localizados en el eje x, con coordenadas $x_1, x_2, \ldots, x_n$, de forma que $x_1=0$ y que $x_1<x_2<\ldots<x_n$.
  - Esos n puntos determinan $m=n(n-1)/2$ distancias $d_1, d_2, \ldots, d_m$, entre cada par de puntos $(x_i-x_j, i>j)$.
  - Dado el conjunto de puntos, es fácil construir el conjunto de distancias en tiempo $O(n^2)$.
  - Más aún, en tiempo $O(n^2 \log n)$, se puede construir el conjunto de distancias ordenado.

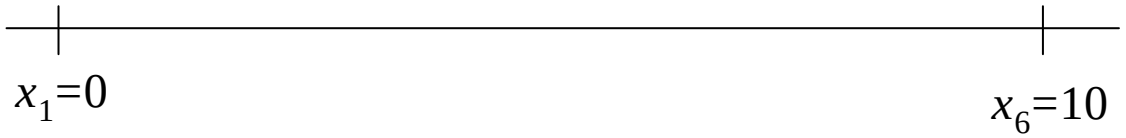
- Problema inverso: cálculo de las coordenadas de los puntos a partir de las distancias (si existen).
  - Tiene aplicaciones en física y biología molecular.
  - No se conoce todavía un algoritmo que lo resuelva en tiempo polinómico.
  - El algoritmo que vamos a presentar "parece" comportarse en tiempo $O(n^2 \log n)$, pero ni se ha demostrado esta conjetura ni se ha encontrado un contraejemplo.

4 Reconstrucción de puntos a partir de las distancias

- Veamos un ejemplo:

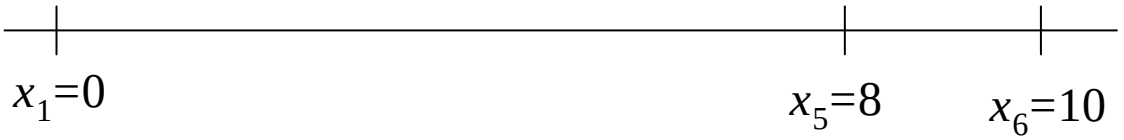
  D={1,2,2,2,3,3,3,4,5,5,5,6,7,8,10}

  - Como $|D| = m = n(n-1)/2$, n=6.
  - Se fija x_1=0 (por ejemplo).
  - Claramente, x_6=10, porque 10 es el máximo de D.



D={1,2,2,2,3,3,3,4,5,5,5,6,7,8}

  - La distancia mayor restante es 8, luego x_2=2 o x_5=8. Por simetría, puede observarse que ambas elecciones dan una misma solución (en realidad, una da la imagen especular de la otra) o no dan solución. Tomamos x_5=8.
  - Se borran las distancias x_6-x_5=2 y x_5-x_1=8 de D.

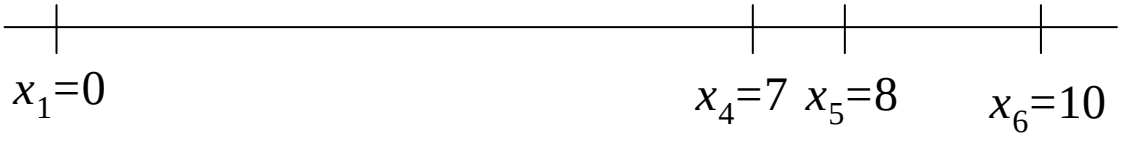


D={1,2,2,3,3,3,4,5,5,5,6,7}

4 Reconstrucción de puntos a partir de las distancias

– El siguiente valor más grande de D es 7 luego, x_4=7 o x_2=3.

◆ Si x_4=7, las distancias x_6-7=3 y x_5-7=1 también deben estar en D. Lo están.



D={2,2,3,3,4,5,5,5,6}

La distancia mayor es ahora 6, así que x_3=6 o x_2=4.

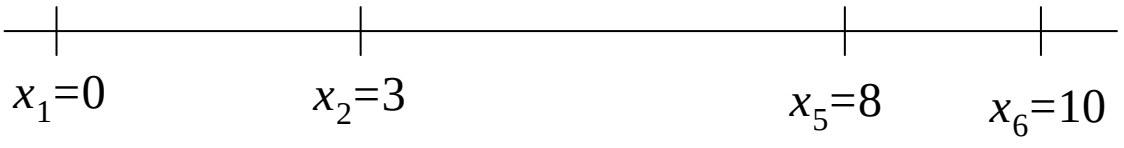
Pero si x_3=6, entonces x_4-x_3=1, lo cual es imposible porque 1 ya no está en D.

Y si x_2=4, entonces x_2-x_1=4 y x_5-x_2=4, lo cual también es imposible porque 4 sólo aparece una vez en D.

Retroceso...

4 Reconstrucción de puntos a partir de las distancias

- Si x_2=3, las distancias 3-x_1=3 y x_5-3=5 también deben estar en D. Lo están.

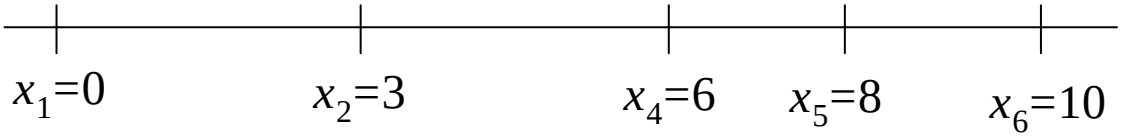


D={1,2,2,3,3,4,5,5,6}

Ahora hay que escoger entre x_4=6 o x_3=4.

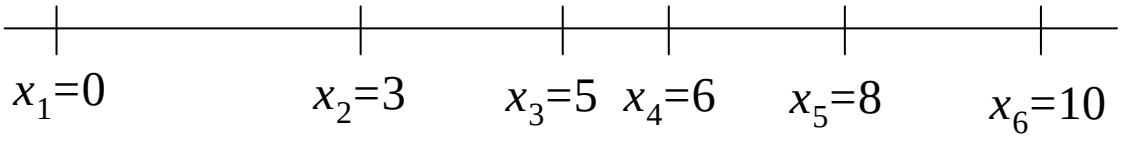
x_3=4 es imposible porque D sólo tiene una ocurrencia de 4 y harían falta dos.

x_4=6 sí es posible.



D={1,2,3,5,5}

La única elección que queda es x_3=5:

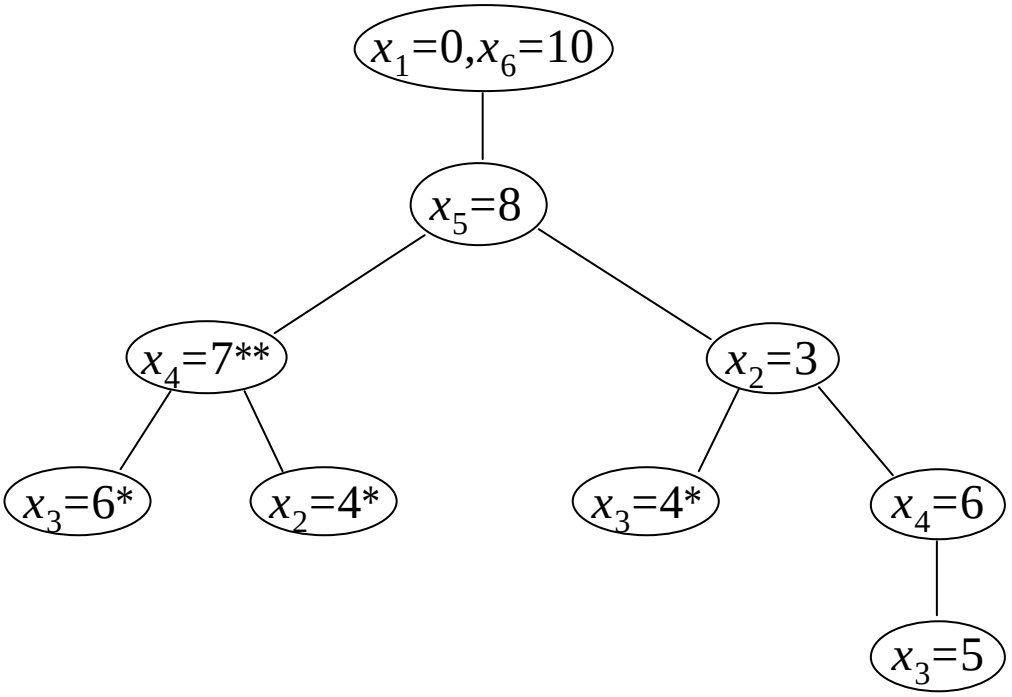


D={}

Y se ha llegado a una solución.

4A Reconstrucción de puntos a partir de las distancias

– El árbol asociado al proceso anterior es:



(*) indica que los puntos elegidos son inconsistentes con las distancias dadas

(**) indica que ese nodo tiene dos nodos imposibles como hijos (luego es un camino incorrecto)

4 Reconstrucción de puntos a partir de las distancias

```
tipos saco = multiconjunto de entero;
      sol = vector[1..n] de entero
```

```
algoritmo reconstruir(ent D:saco;
                      sal x:sol;
                      sal éxito:booleano)
{Pre: |D|=n(n-1)/2}
principio
  éxito:=falso;
  x[1]:=0;
  x[n]:=máximo(D);
  eliminarMáx(D);
  x[n-1]:=máximo(D);
  eliminarMáx(D);
  si x[n]-x[n-1]∈D
    entonces
      eliminar(x[n]-x[n-1],D);
      colocar(x,D,2,n-2,éxito)
    sino
      éxito:=falso
  fsi
fin
```

4 Reconstrucción de puntos a partir de las distancias

```
algoritmo colocar(e/s x:sol; e/s D:saco;
                  ent iz,de:entero;
                  sal éxito:booleano)
{Algoritmo de búsqueda con retroceso para colocar
 los puntos x[iz]..x[de].
 x[1]..x[iz-1] y x[de+1]..x[n] ya se han colocado
 provisionalmente.}
variable maxD:entero
principio
  si D=∅ entonces éxito:=verdad
  sino
    maxD:=máximo(D);
    {comprueba si es factible x[de]=maxD}
    si |x[j]-maxD|∈D,∀ 1≤j<iz,∀ de<j≤n entonces
      x[de]:=maxD; {intenta x[de]=maxD}
      para 1≤j<iz,de<j≤n hacer
        eliminar(|x[j]-maxD|,D)
      fpara;
      colocar(x,D,iz,de-1,éxito);
      si not éxito entonces {retroceso}
        para 1≤j<iz,de<j≤n hacer
          {deshace la eliminación}
          insertar(|x[j]-maxD|,D)
        fpara
      fsi
    fsi;
...
```

4 Reconstrucción de puntos a partir de las distancias

```
...
    {Si falló el primer intento, se intenta ver si
     x[iz]=x[n]-maxD}
    si not éxito and |x[n]-maxD-x[j]|∈D,
                     ∀ 1≤j<iz,∀ de<j≤n entonces
      x[iz]:=x[n]-maxD;
      para 1≤j<iz,de<j≤n hacer
        eliminar(|x[n]-maxD-x[j]|,D)
      fpara;
      colocar(x,D,iz+1,de,éxito);
      si not éxito entonces {retroceso}
        para 1≤j<iz,de<j≤n hacer
          {deshace la eliminación}
          insertar(|x[n]-maxD-x[j]|,D)
        fpara
      fsi
    fsi
  fsi
fin
```

4 Reconstrucción de puntos a partir de las distancias

- Análisis de la eficiencia:
  - Caso fácil: No se realiza ningún retroceso.
    - Hay como máximo $O(n^2)$ operaciones de "eliminar" un elemento de D.

      En efecto, porque no se realiza ninguna inserción en D e inicialmente tiene $O(n^2)$ elementos.
    - Hay como máximo $O(n^2)$ operaciones de búsqueda ($\in$) de un elemento.

      En efecto, en cada ejecución de colocar hay como máximo $2n$ búsquedas y colocar se ejecuta $O(n)$ veces (si no hay retrocesos).
    - Si D se guarda en un árbol AVL, el coste de las operaciones de eliminar y buscar es $O(\log n)$.

    → el coste es $O(n^2 \log n)$

4 Reconstrucción de puntos a partir de las distancias

– En general, ocurren retrocesos, y:

- No se conoce ninguna cota polinómica del número de retrocesos (luego, en principio, no se conoce ninguna solución polinómica).

Sin embargo:

- ¡ No se conoce ningún ejemplo en el que el número de retrocesos sea mayor que $O(1)$!

Luego, es posible que el algoritmo sea $O(n^2 \log n)$.

Por ejemplo, se ha demostrado que si los n puntos se generan de acuerdo con una distribución uniforme en el intervalo [0,max], con max=$\Theta(n^2)$, entonces con probabilidad 1 se efectúa como máximo un retroceso durante todo el algoritmo.

4.4 Árboles de juego: tic-tac-toe

- Estudio de ciertos juegos de estrategia que pueden representarse mediante árboles:
  - cada nodo se corresponde con una **configuración** posible del juego (por ejemplo, estado de un tablero), y
  - cada arco es una transición legal, o **jugada**, desde una configuración posible a una de sus sucesoras.

- Por simplificar, consideremos que:
  - hay dos jugadores que juegan alternadamente,
  - los dos jugadores están sometidos a las mismas reglas (juego *simétrico*),
  - el azar no interviene (juego *determinista*),
  - una partida no puede durar indefinidamente, y
  - ninguna configuración tiene un número infinito de posibles sucesoras.

4.4 Árboles de juego: tic-tac-toe

– La configuración inicial del juego es la raíz del árbol correspondiente.

– Las hojas del árbol corresponden a las configuraciones terminales del juego, en las que no existe jugada siguiente

  - bien porque uno de los jugadores ha ganado
  - bien porque no ha ganado ninguno (situación de empate)

– Los niveles impares del árbol están asociados con las configuraciones en las que debe jugar uno de los dos jugadores, mientras que los niveles pares se asocian a las configuraciones en las que debe jugar el otro.

4.4 Árboles de juego: tic-tac-toe

– A cada nodo del árbol se le asocia una etiqueta llamada **función de utilidad**.

– Por ejemplo, una función de utilidad habitual toma tres valores posibles:

- "configuración ganadora",
- "configuración perdedora" y
- "configuración empatadora o nula".

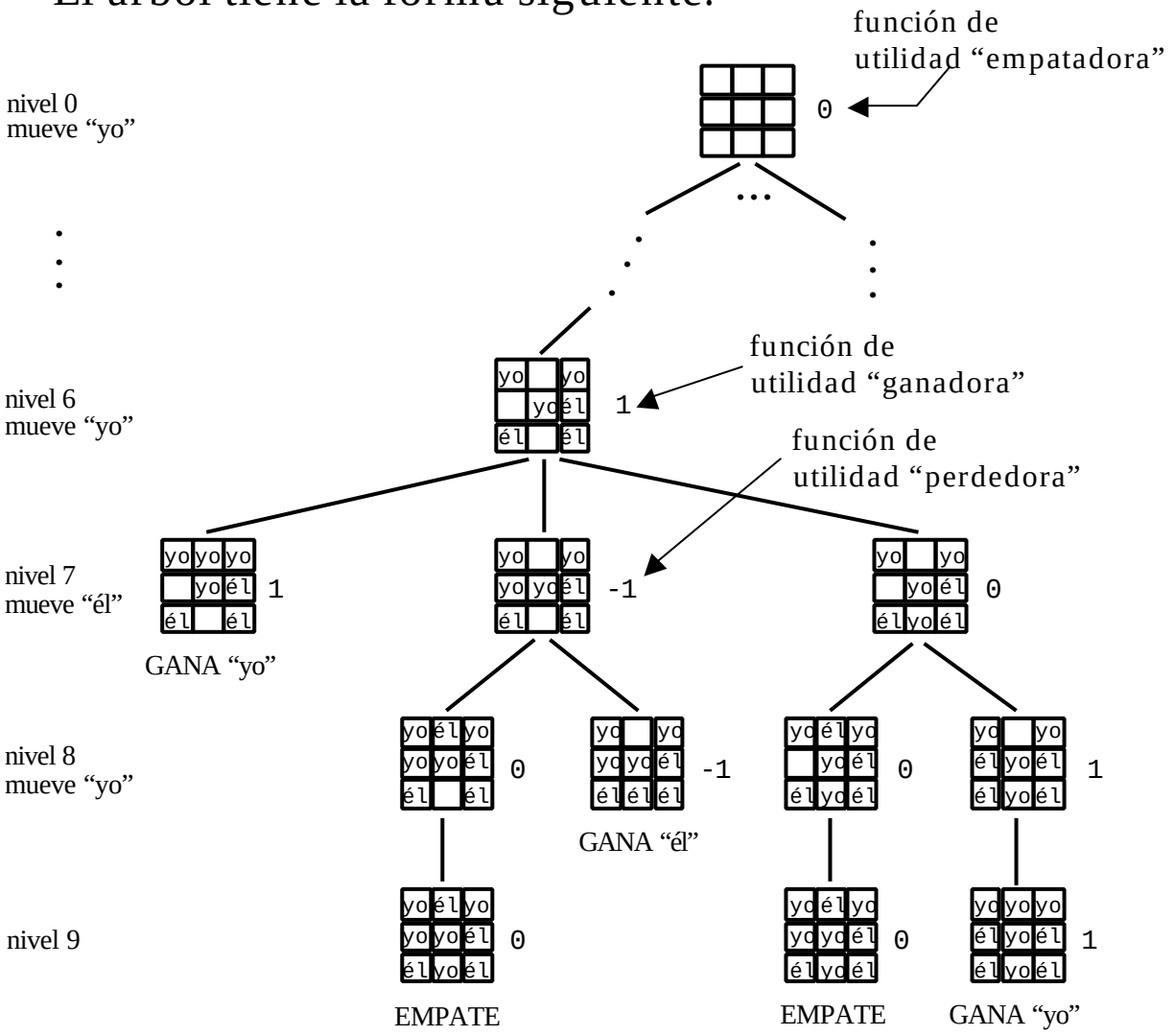
– Interpretación:

Situación (o posibilidades) que tiene el jugador (tomamos partido por uno de ellos) en esa configuración, suponiendo que ninguno de los dos jugadores se equivocará y ambos realizarán en lo sucesivo la mejor jugada posible.

– La función de utilidad puede asignarse de forma sistemática.

4 Árboles de juego: tic-tac-toe

- Ejemplo: el juego del *tic-tac-toe*.
  - Una especie de tres en raya con más de tres fichas por jugador.
  - Llamaremos a los dos jugadores “él” y “yo”.
  - Supondremos, por ejemplo, que empieza “yo”.
  - El árbol tiene la forma siguiente:



4.4 Árboles de juego: tic-tac-toe

- Cálculo de la utilidad:
  - Se da valor, en primer lugar, a las hojas:
    - la utilidad en una hoja vale 1, 0 ó -1 si la configuración del juego corresponde a una victoria, empate o derrota, respectivamente, del jugador por el que hemos tomado partido ("yo").
  - Los valores de la función de utilidad se propagan hacia arriba del árbol de acuerdo a la siguiente regla (**estrategia *minimax***):
    - si un nodo corresponde a una configuración del juego en la que juega "yo" (nivel 0 ó par), se supone que ese jugador hará la mejor jugada de entre las posibles y, por tanto, el valor de la función de utilidad en la configuración actual coincide con el valor de esa función en la configuración de la mejor jugada posible (para "yo") que se puede realizar desde la actual (**nodo *max***);
    - si un nodo corresponde a una configuración del juego en la que juega "él" (nivel impar del árbol), se supone que ese jugador hará la mejor jugada de entre las posibles y, por tanto, el valor de la función de utilidad en la configuración actual coincide con el valor de esa función en la configuración de la peor jugada posible (para "yo") (**nodo *min***).

4 Árboles de juego: *tic-tac-toe*

– Si la raíz tuviera el valor 1, entonces “yo” tendría una estrategia que le permitiría ganar siempre (no es el caso del tic-tac-toe).

– En el *tic-tac-toe* la función de utilidad de la raíz vale 0 (ningún jugador tiene una estrategia ganadora):

si no se cometen errores, se puede garantizar al menos un empate.

– Si la raíz tuviera un valor -1, el otro jugador (“él”) tendría una estrategia ganadora.

4.4 Árboles de juego: tic-tac-toe

– La función de utilidad puede tener un rango más amplio (por ejemplo, los números enteros).

– Ejemplo: **Ajedrez**

- se ha estimado que el árbol del juego tiene más de 10^{100} nodos
- si un computador pudiera generar 10^{11} nodos por segundo, necesitaría más de 10^{80} años para construirlo
- es imposible dar valor a la función de utilidad de cada nodo de abajo hacia arriba (como en el caso del *tic-tac-toe*)
- para cada configuración actual del juego, se toma dicha configuración como raíz del árbol
- se construyen varios niveles del árbol (el número depende de la velocidad del computador o puede ser seleccionado por el usuario)
- la mayor parte de las hojas del árbol construido son ambiguas (no indican triunfos, derrotas ni empates)
- la función de utilidad de las posiciones del tablero intenta estimar la **probabilidad** de que el computador gane desde esa posición

4.4 Árboles de juego: tic-tac-toe

- Implementación (±) detallada:
  - representación de una configuración del juego (definición del tipo de dato *configuración*)
  - enumeración de las configuraciones accesibles desde una dada mediante la ejecución de una jugada, y
  - cálculo de la función de utilidad para una configuración dada, sabiendo quién juega en ese momento

4 Árboles de juego: tic-tac-toe

```
algoritmo juegaElComputador(e/s c:config)
{c es una configuración no final; el algoritmo
 realiza la mejor jugada posible a partir de c,
 la comunica al usuario y actualiza c}
variables maxUtilidad,laUtilidad:entero;
          i:1..maxEntero;
          mejorJugada,unaJugada:config
principio
  i:=1;
  {cálculo de la 1ª config. accesible desde c}
  jugada(c,i,unaJugada);
  mejorJugada:=unaJugada;
  maxUtilidad:=utilidad(mejorJugada,falso);
  {"falso" indica que cuando la configuración
   sea "mejorJugada", no juego yo}
  mq not esLaUltimaJugada(c,i) hacer
    i:=i+1;
    jugada(c,i,unaJugada);
    laUtilidad:=utilidad(unaJugada,falso);
    si laUtilidad>maxUtilidad entonces
      mejorJugada:=unaJugada;
      maxUtilidad:=laUtilidad
    fsi
  fmq;
  comunicaAlUsuario(mejorJugada);
  c:=mejorJugada
fin
```

4 Árboles de juego: tic-tac-toe

```
algoritmo jugada(ent c:config;
                 ent i:1..maxEntero;
                 sal unaJugada:config)
{Pre: c admite al menos i jugadas diferentes.}
{Post: unaJugada es la i-ésima jugada posible
       desde c.}
...
algoritmo esLaÚltimaJug(ent c:config;
                        ent i:0..maxEntero)
          devuelve booleano
{Devuelve verdad si y sólo si c admite
 exactamente i jugadas diferentes.}
...
algoritmo comunicaAlUsuario(ent c:config)
{Muestra en pantalla la configuración c.}
...
algoritmo utilidadFinal(ent c:config)
          devuelve entero
{Pre: c es una configuración final del juego.}
{Post: devuelve la utilidad de c.}
...
```

4 Árboles de juego: tic-tac-toe

```
algoritmo utilidad(ent c:config;
                   ent juegoYo:booleano)
          devuelve entero
{Calcula la utilidad de c teniendo en cuenta
 quién juega.}
variables laUtilidad:entero;
          i:1..maxEntero;
          unaJugada:config
principio
  si esLaUltimaJug(c,0)
    entonces
      devuelve utilidadFinal(c)
    sino
      i:=1;
      jugada(c,i,unaJugada);
      laUtilidad:=
        utilidad(unaJugada,not juegoYo);
...
```

4 Árboles de juego: tic-tac-toe

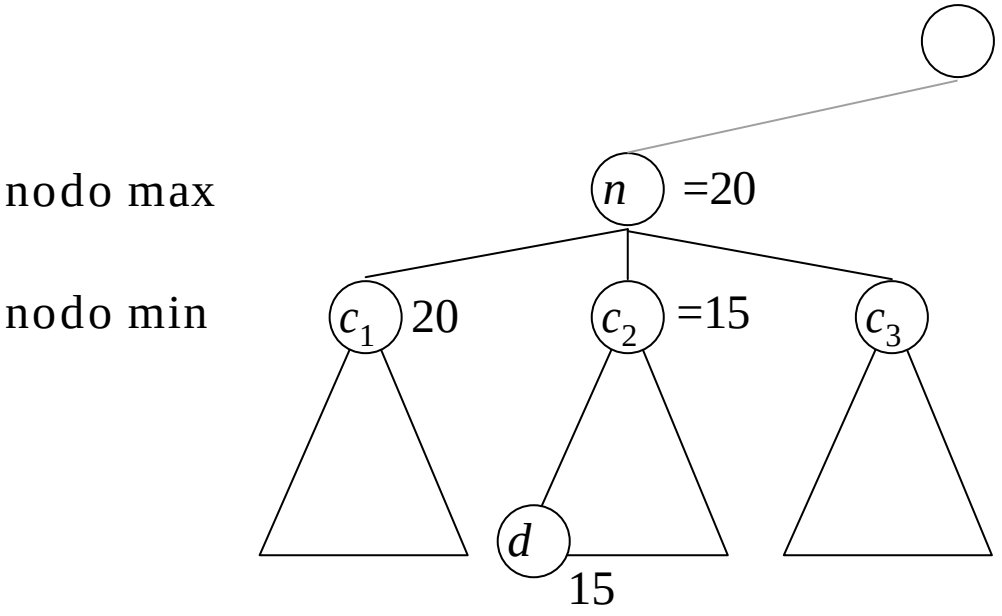
```
...
      mq not esLaUltimaJug(c,i) hacer
        i:=i+1;
        jugada(c,i,unaJugada);
        si juegoYo
          entonces
            laUtilidad:=
              max(laUtilidad,
                  utilidad(unaJugada,falso))
          sino
            laUtilidad:=
              min(laUtilidad,
                  utilidad(unaJugada,verdad))
        fsi
      fmq;
      devuelve laUtilidad
  fsi
fin
```

54 Árboles de juego: tic-tac-toe

- Sobre el tamaño del árbol:
  - Si empieza el computador, se examinan 97162 nodos.
  - Si el computador juega en segundo lugar:
    - se examinan 5185, si el jugador humano colocó primero en el centro,
    - se examinan 9761, si el humano colocó en una esquina,
    - se examinan 13233, si el humano colocó en una casilla que no es ni el centro ni una esquina.
- Se precisan métodos para evaluar menos nodos sin perder información…
- El más conocido es el método de "poda α–β"

4.4 Árboles de juego: tic-tac-toe

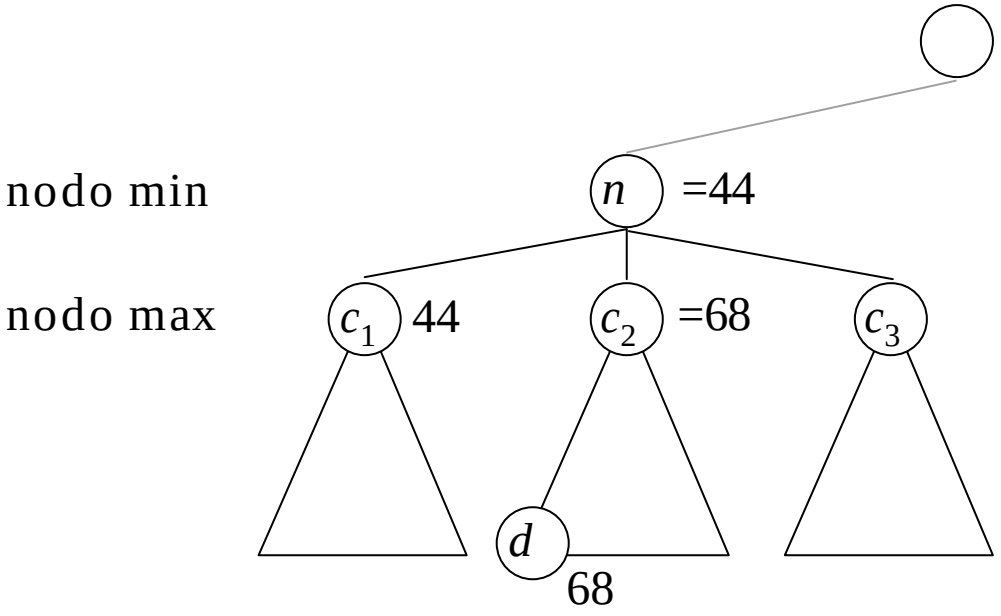
- Poda α:



  - Una vez que se encuentra el descendiente *d* de c_2, no es necesario seguir calculando descendientes de c_2 (porque c_2 es un nodo "min", tendrá un valor menor o igual a 15, y su padre es "max" y ya tiene un valor mayor o igual a 20).

Árboles de juego: tic-tac-toe

- Poda β:



  - Una vez que se encuentra el descendiente *d* de c_2, no es necesario seguir calculando descendientes de c_2 (porque c_2 es un nodo "max", tendrá un valor mayor o igual a 68, y su padre es "min" y ya tiene un valor menor o igual a 44).

4.4 Árboles de juego: *tic-tac-toe*

- Resultados prácticos:
  - La poda α–β restringe la búsqueda a $O(\sqrt{n})$ nodos (si el número original es n).
  - Este ahorro es muy grande y supone que las búsquedas que usan la poda α–β pueden ir al doble de la profundidad en comparación con un árbol no podado.
  - En el ejemplo del *tic-tac-toe*, el número inicial de 97162 nodos se reduce a 4493.