Algorítmica

Tema 5: Algoritmos para la Exploración de Grafos (BK y BB)

Bibibliografía:

G. BRASSARD, P. BRATLEY. Fundamentos de Algoritmia. Prentice Hall (1997).
E. HOROWITZ, S. SAHNI, S. RAJASEKARAN. Computer Algorithms. Computer Science Press (1998).

Objetivos

- Comprender la filosofía de diseño de algoritmos *Backtracking* (“vuelta atrás”) y *Branch and Bound* (“ramifica y poda”)
- Conocer las características de un problema resoluble mediante dichas técnicas
- Resolución de diversos problemas

Índice

Índice

I. LA TÉCNICA BACKTRACKING

1. **Introducción: El método General**
   - **Resolución de problemas cuando la solución se puede expresar como una n-tupla**
   - **Ejemplo: El problema de las 8 reinas**
   - **Ejemplo: La suma de subconjuntos**
2. **Espacio de soluciones: Organización del Árbol**
   - **Ejemplo: Espacio solución para el problema de las N-reinas (N=4)**
   - **Ejemplo: Espacio solución para la suma de subconjuntos**
   - **Terminología utilizada para la organización en árbol**
   - **Ejemplo: *Backtracking* en el problema de las 4 reinas**
3. **Procedimiento *Backtracking***
   - **Procedimiento Iterativo**
   - **Procedimiento Recursivo**

Método general

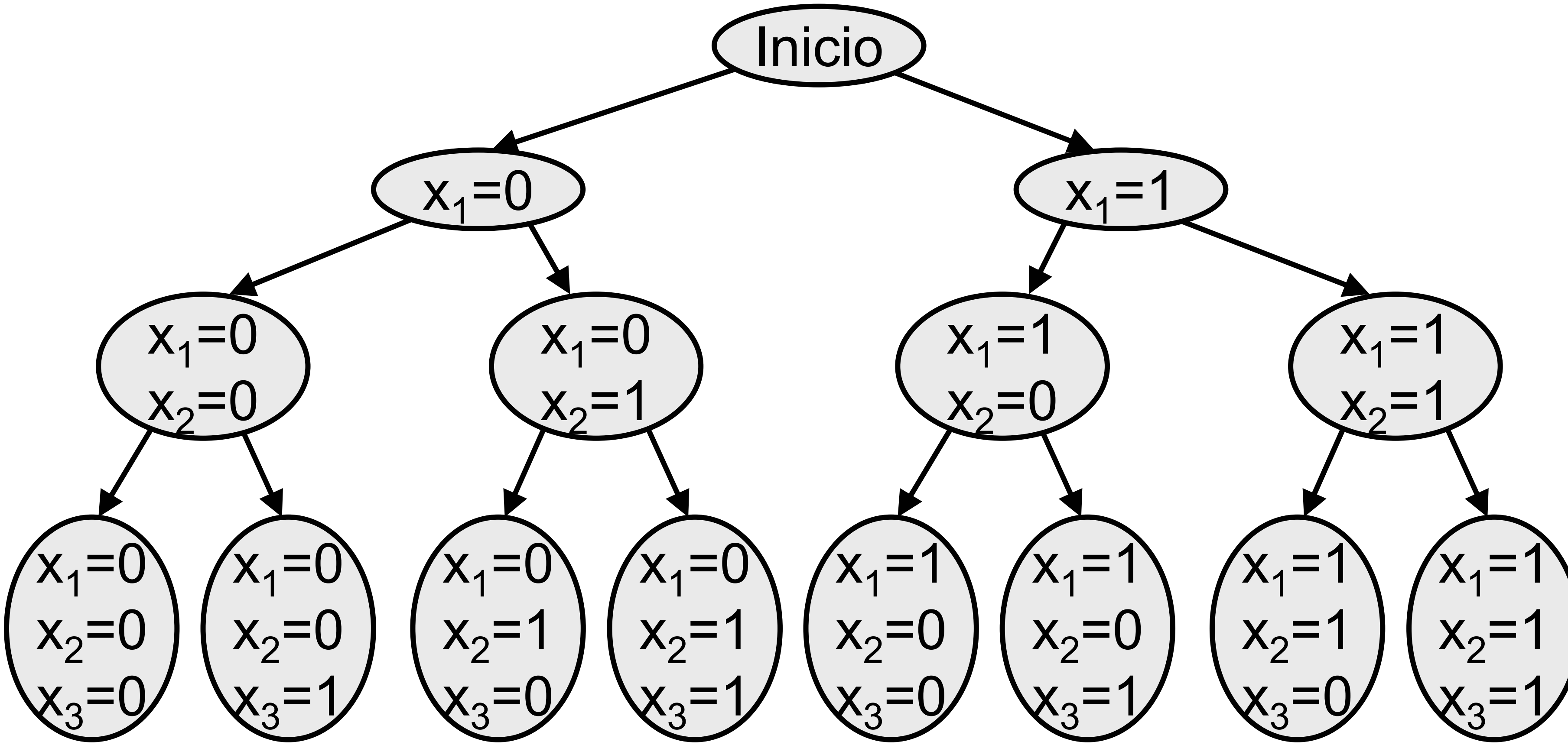
- El **backtracking** (o método de **retroceso** o **vuelta atrás**) es una técnica general de resolución de problemas, aplicable en problemas de **optimización**, **juegos** y otros tipos.
- El **backtracking** realiza una búsqueda exhaustiva y sistemática en el espacio de soluciones. Por ello, suele resultar muy ineficiente.
- Se puede entender como "opuesto" a avance rápido:
  - **Avance rápido:** añadir elementos a la solución y no deshacer ninguna decisión tomada.
  - **Backtracking:** añadir y quitar todos los elementos. Probar todas las combinaciones.

Método general

- Una **solución** se puede expresar como una tupla: $(x_1, x_2, ..., x_n)$, satisfaciendo unas restricciones y tal vez optimizando cierta función objetivo.
- En cada momento, el algoritmo se encontrará en cierto nivel **k**, con una solución parcial $(x_1, ..., x_k)$.
  - Si se puede añadir un nuevo elemento a la solución $\mathbf{x_{k+1}}$, se genera y se avanza al nivel **k+1**.
  - Si no, se prueban otros valores para $\mathbf{x_k}$.
  - Si no existe ningún valor posible por probar, entonces se retrocede al nivel anterior **k-1**.
  - Se sigue hasta que la solución parcial sea una solución completa del problema, o hasta que no queden más posibilidades por probar.

Método general

- El resultado es equivalente a hacer un **recorrido en profundidad** en el árbol de soluciones.

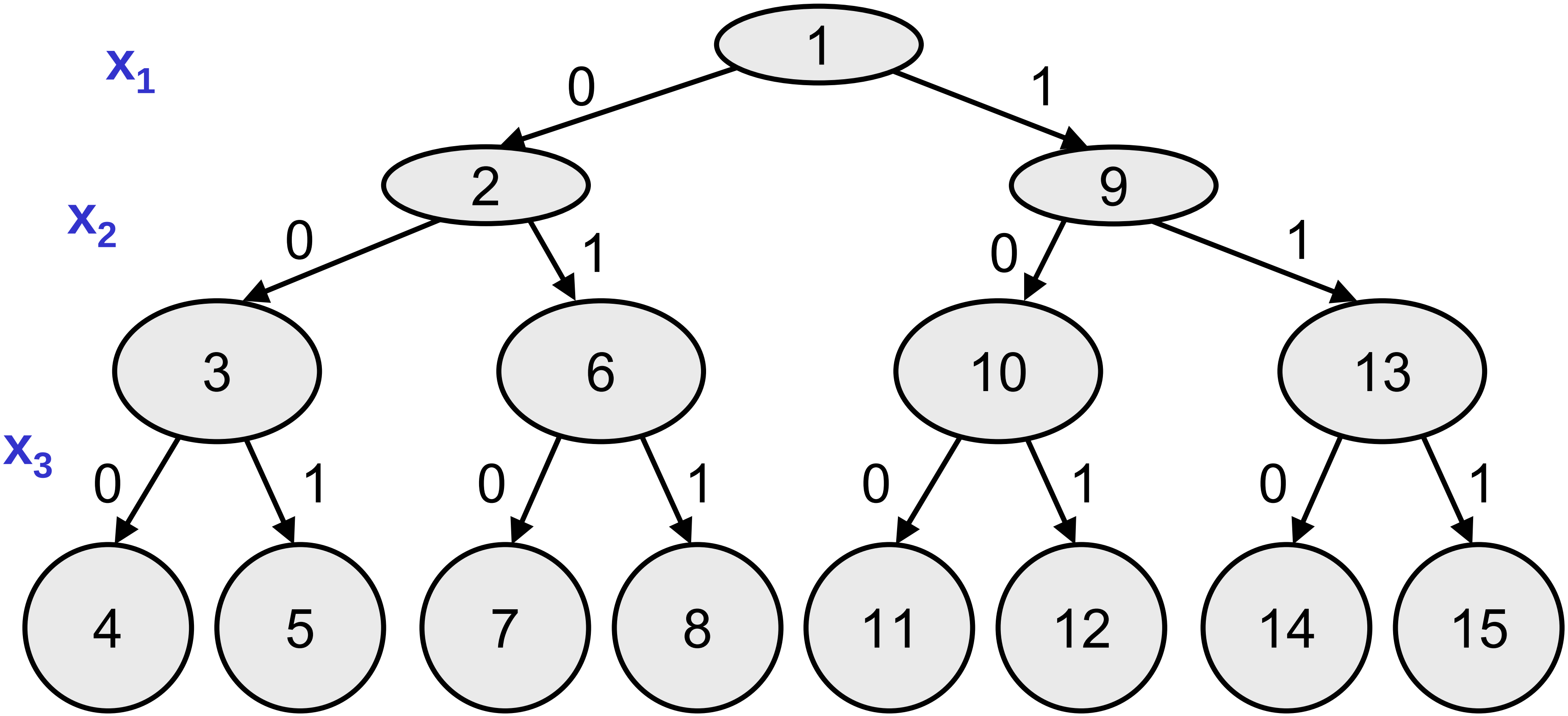


Método general

- El resultado es equivalente a hacer un **recorrido en profundidad** en el árbol de soluciones.

Método general

- Representación simplificada del árbol.



Método general

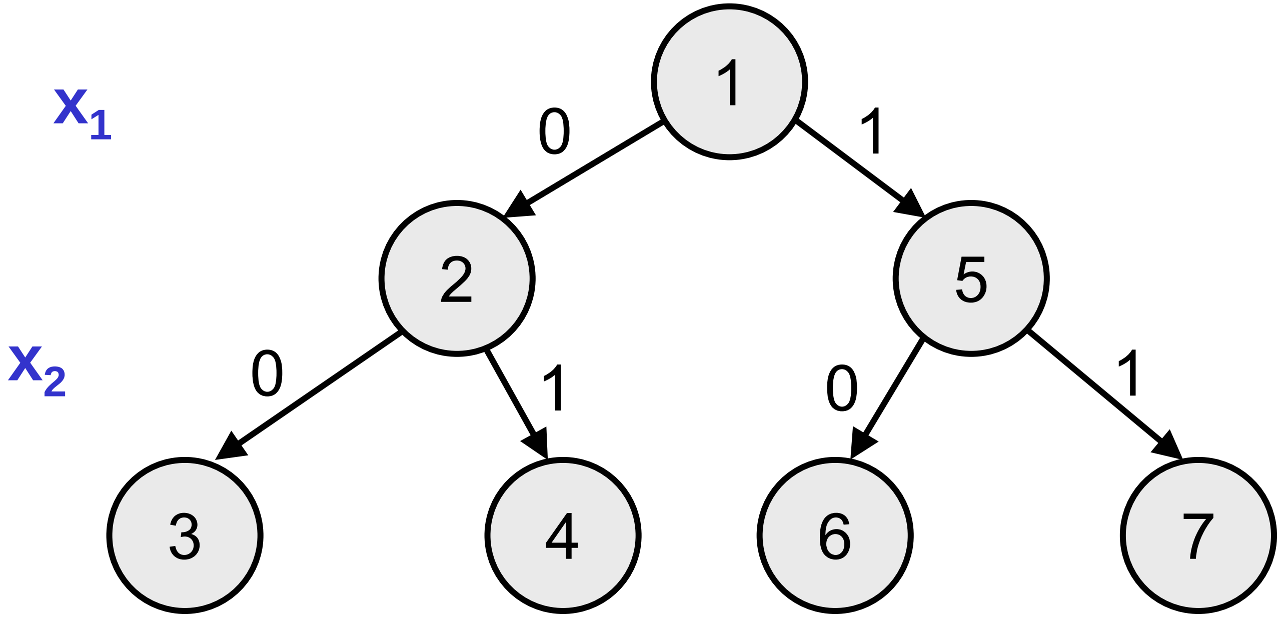
- **Árboles de backtracking:**
  - El árbol es simplemente una forma de representar la ejecución del algoritmo.
  - Es **implícito**, no almacenado (no necesariamente).
  - El recorrido es en **profundidad**, normalmente de izquierda a derecha.
  - La primera decisión para aplicar backtracking: ¿cómo es la forma del árbol?
  - **Preguntas relacionadas:** ¿Qué significa cada valor de la tupla solución (x_1, ..., x_n)? ¿Cómo es la representación de la solución al problema?

Método general

- Tipos comunes de árboles de backtracking:
  - Árboles binarios.
  - Árboles n-arios.
  - Árboles permutacionales.
  - Árboles combinatorios.

Método general

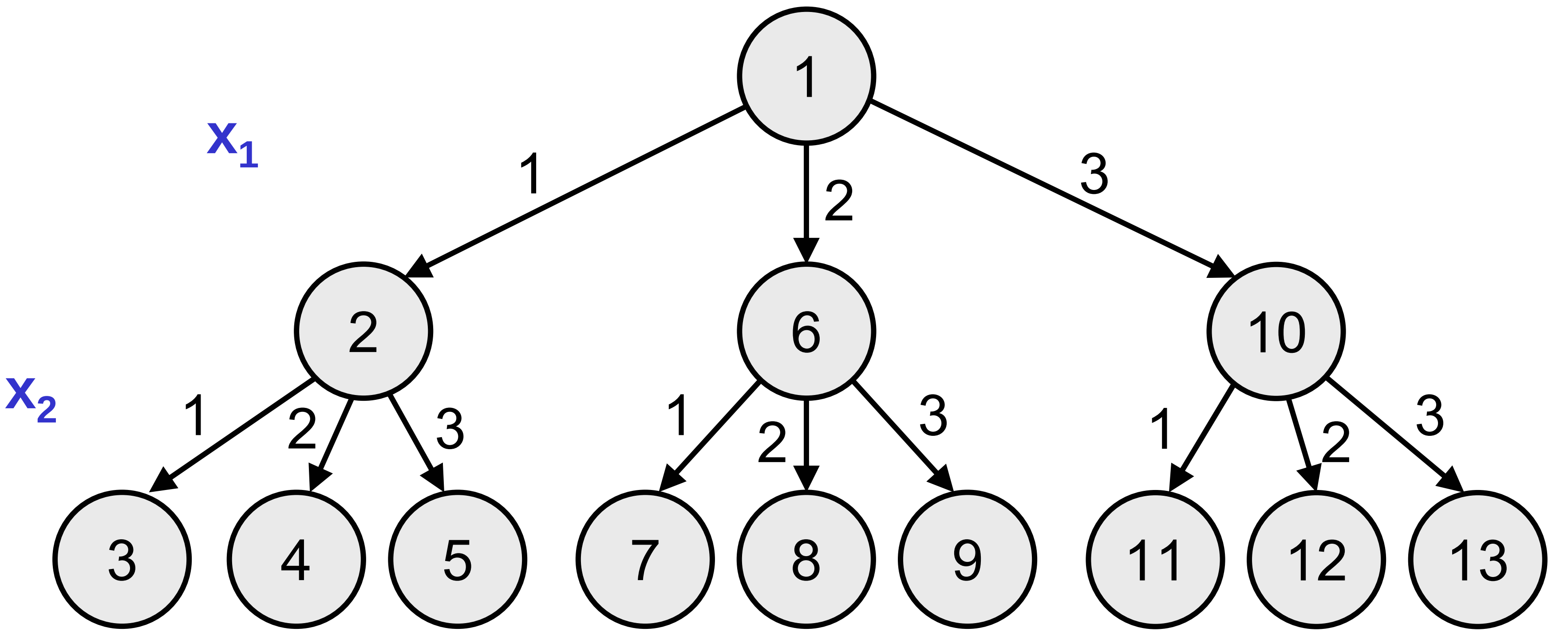
- **Árboles binarios: s=** $(x_1, x_2, \ldots, x_n)$, con $x_i \in \{0, 1\}$



- **Tipo de problemas:** elegir ciertos elementos de entre un conjunto, sin importar el orden de los elementos.
  - Problema de la mochila 0/1.
  - Encontrar un subconjunto de {12, 23, 1, 8, 33, 7, 22} que sume exactamente 50.

Método general

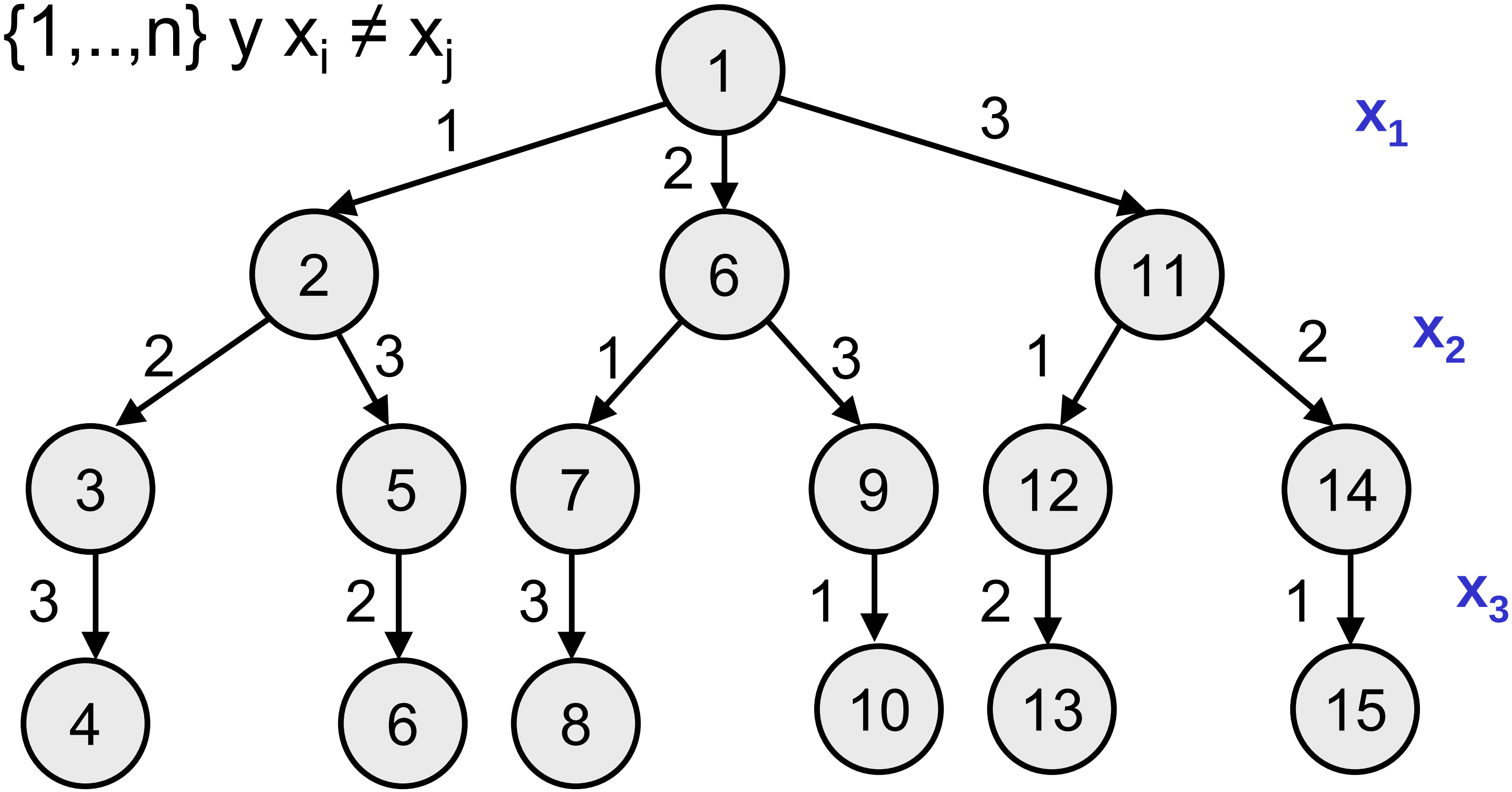
- **Árboles k-arios: s=** $(x_1, x_2, ..., x_n)$, con $x_i \in \{1,..,k\}$



- **Tipo de problemas:** varias opciones para cada x_i.
  - Problema del cambio de monedas.
  - Problema de las **n** reinas.

Método general

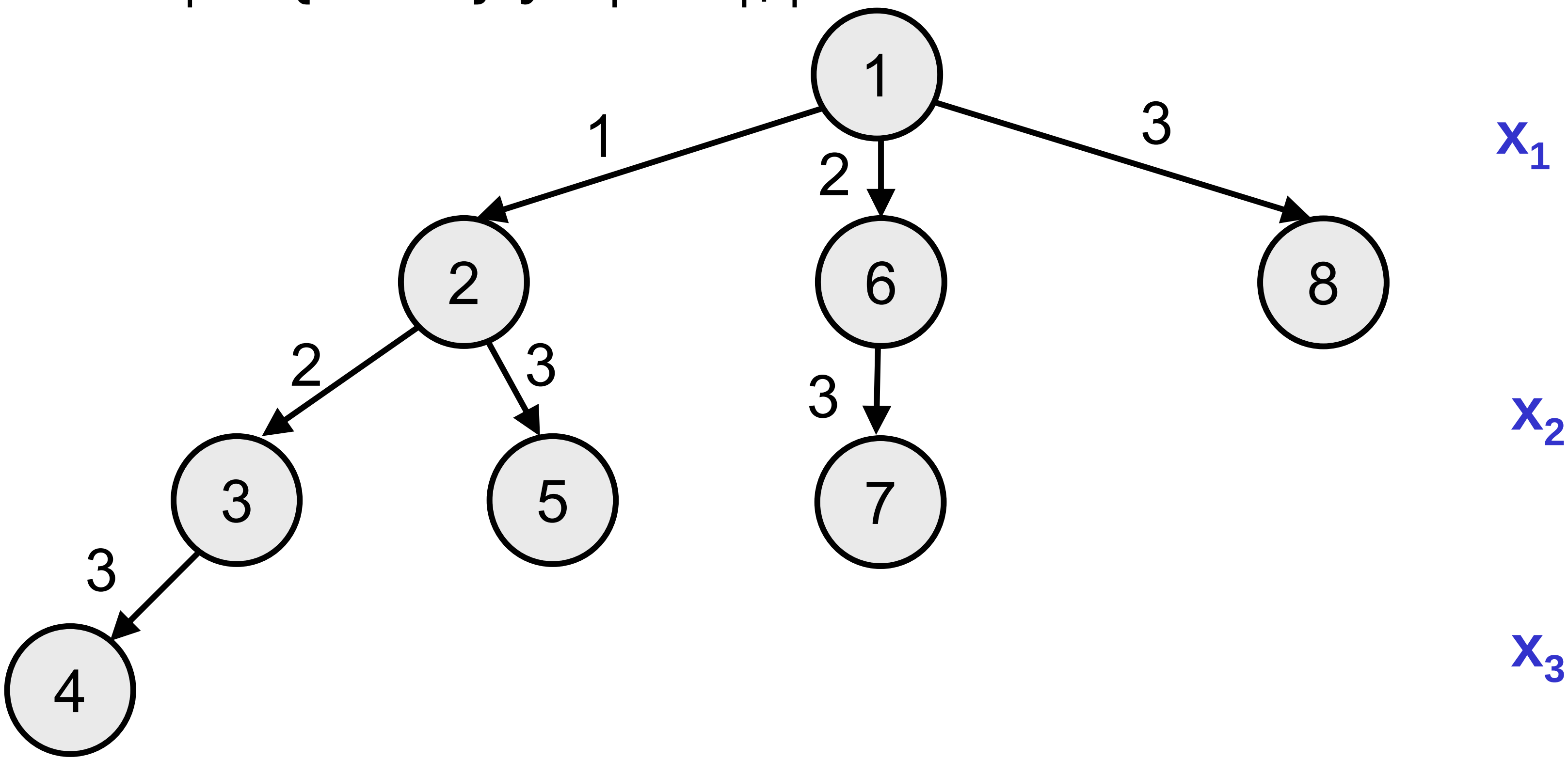
- **Árboles permutacionales: s=** $(x_1, x_2, ..., x_n)$, con $x_i \in \{1,..,n\}$ y $x_i \neq x_j$



- **Tipo de problemas:** los x_i no se pueden repetir.
  – Generar todas las permutaciones de (1, ..., n).
  – Asignar **n** trabajos a **n** personas, asignación uno-a-uno.

Método general

- **Árboles combinatorios: s=** $(x_1, x_2, \ldots, x_m)$, con $m \le n$, $x_i \in \{1,..,n\}$ y $x_i < x_{i+1}$



- **Tipo de problemas:** los mismos que con árb. binarios.
  - Binario: (0, 1, 0, 1, 0, 0, 1) → Combinatorio: (2, 4, 7)

Método general

Cuestiones a resolver antes de programar:

- ¿Qué tipo de árbol es adecuado para el problema?
  - → ¿Cómo es la representación de la solución?
  - → ¿Cómo es la tupla solución? ¿Qué indica cada $\mathbf{x_i}$ y qué valores puede tomar?
- ¿Cómo generar un recorrido según ese árbol?
  - → Generar un nuevo nivel.
  - → Generar los hermanos de un nivel.
  - → Retroceder en el árbol.
- ¿Qué ramas se pueden descartar por no conducir a soluciones del problema?
  - → Poda por restricciones del problema.
  - → Poda según el criterio de la función objetivo.

Método general

- **Esquema general (no recursivo)**. Problema de satisfacción de restricciones: buscamos cualquier solución que cumpla cierta propiedad, y se supone que existe alguna.

Backtracking (var s: TuplaSolución)

nivel:= 1
s:= $s_{INICIAL}$
fin:= false
repetir
    Generar (nivel, s)
    si **Solución (nivel, s)** entonces
        fin:= true
    sino si **Criterio (nivel, s)** entonces
        nivel:= nivel + 1
    sino mientras NOT **MasHermanos (nivel, s)** hacer
        Retroceder (nivel, s)
hasta fin

Método general

- **Variables**:
  - **s**: Almacena la solución parcial hasta cierto punto.
  - **$s_{INICIAL}$**: Valor de inicialización.
  - **nivel**: Indica el nivel actual en el que se encuentra el algoritmo.
  - **fin**: Valdrá **true** cuando hayamos encontrado alguna solución.
- **Funciones**:
  - **Generar (nivel, s)**: Genera el siguiente hermano, o el primero, para el **nivel** actual.
  - **Solución (nivel, s)**: Comprueba si la tupla (s[1], ..., s[nivel]) es una solución válida para el problema.

Método general

- **Funciones**:
  - **Criterio (nivel, s)**: Comprueba si a partir de (s[1], ..., s[nivel]) se puede alcanzar una solución válida. En otro caso se rechazarán todos los descendientes (**poda**).
  - **MasHermanos (nivel, s)**: Devuelve **true** si hay más hermanos del nodo actual que todavía no han sido generados.
  - **Retroceder (nivel, s)**: Retrocede un nivel en el árbol de soluciones. Disminuye en 1 el valor de **nivel**, y posiblemente tendrá que actualizar la solución actual, quitando los elementos retrocedidos.
- Además, suele ser común utilizar variables temporales con el valor actual (beneficio, peso, etc.) de la tupla solución.

Método general

- **Ejemplo de problema:** Encontrar un subconjunto del conjunto T= {t_1, t_2, ..., t_n} que sume exactamente P.

- **Variables**:
  – Representación de la solución con un árbol binario.
  – **s**: array [1..n] de {-1, 0, 1}
    • s[i] = 0 → el número i-ésimo no se utiliza
    • s[i] = 1 → el número i-ésimo sí se utiliza
    • s[i] = -1 → valor de inicialización (número i-ésimo no estudiado)
  – $\mathbf{s_{INICIAL}}$: (-1, -1, ..., -1)
  – **fin**: Valdrá **true** cuando se haya encontrado solución.
  – **tact:** Suma acumulada hasta ahora (inicialmente 0).

Método general

Funciones:

- **Generar (nivel, s)**

  s[nivel]:= s[nivel] + 1

  si s[nivel]==1 **entonces** tact:= tact + t_{nivel}

- **Solución (nivel, s)**

  devolver (nivel==n) Y (tact==P)

- **Criterio (nivel, s)**

  devolver (nivel<n) Y (tact≤P)

- **MasHermanos (nivel, s)**

  devolver s[nivel] < 1

- **Retroceder (nivel, s)**

  tact:= tact – t_{nivel}*s[nivel]

  s[nivel]:= -1

  nivel:= nivel – 1

Método general

- **Algoritmo:** ¡el mismo que el esquema general!

```
Backtracking (var s: TuplaSolución)
    nivel:= 1
    s:= s_INICIAL
    fin:= false
    repetir
        Generar (nivel, s)
        si Solución (nivel, s) entonces
            fin:= true
        sino si Criterio (nivel, s) entonces
            nivel:= nivel + 1
        sino
            mientras NOT MasHermanos (nivel, s) hacer
                Retroceder (nivel, s)
        finsi
    hasta fin
```

Método general

Variaciones del esquema general:

1) ¿Y si no es seguro que exista una solución?

2) ¿Y si queremos almacenar todas las soluciones (no sólo una)?

3) ¿Y si el problema es de optimización (maximizar o minimizar)?

Método general

- **Caso 1)** Puede que no exista ninguna solución.

Backtracking (var s: TuplaSolución)

```
nivel:= 1
s:= s_INICIAL
fin:= false
repetir
    Generar (nivel, s)
    si Solución (nivel, s) entonces
        fin:= true
    sino si Criterio (nivel, s) entonces
        nivel:= nivel + 1
    sino
        mientras NOT MasHermanos (nivel, s) AND (nivel>0)
            hacer Retroceder (nivel, s)
    finsi
hasta fin OR (nivel==0)
```

Para poder generar todo el árbol de backtracking

Método general

- **Caso 2)** Queremos almacenar todas las soluciones.

Backtracking (var s: TuplaSolución)

nivel:= 1
s:= $s_{INICIAL}$
fin:= false
repetir
    Generar (nivel, s)
    si Solución (nivel, s) **entonces**
        Almacenar (nivel, s)
    si Criterio (nivel, s) **entonces**
        nivel:= nivel + 1
    sino
        mientras NOT MasHermanos (nivel, s) **AND (nivel>0)**
            hacer Retroceder (nivel, s)
    finsi
hasta **nivel==0**

- En algunos problemas los nodos intermedios pueden ser soluciones
- O bien, retroceder después de encontrar una solución

Método general

- **Caso 3)** Problema de optimización (maximización).

Backtracking (var s: TuplaSolución)

> **voa**: valor óptimo actual
> **soa:** solución óptima actual

```
Backtracking (var s: TuplaSolución)
    nivel:= 1
    s:= s_INICIAL
    voa:= -∞; soa:= Ø
    repetir
        Generar (nivel, s)
        si Solución (nivel, s)  AND Valor(s) > voa entonces
            voa:= Valor(s); soa:= s
        si Criterio (nivel, s) entonces
            nivel:= nivel + 1
        sino
            mientras NOT MasHermanos (nivel, s) AND (nivel>0)
                    hacer Retroceder (nivel, s)
        finsi
    hasta  nivel==0
```

Método general

- **Ejemplo de problema:** Encontrar un subconjunto del conjunto T= {t_1, t_2, ..., t_n} que sume exactamente P, usando el menor número posible de elementos.

- **Funciones**:
  - **Valor(s)**
    devolver s[1] + s[2] + ... + s[n]
  - ¡Todo lo demás no cambia!

- **Otra posibilidad:** incluir una nueva variable:
  vact: entero. Número de elementos en la tupla actual.
  - **Inicialización** (añadir)**:** vact:= 0
  - **Generar** (añadir)**:** vact:= vact + s[nivel]
  - **Retroceder** (añadir)**:** vact:= vact – s[nivel]

Backtracking: Resumen

- **Si** tenemos que tomar una serie de decisiones entre una gran variedad de opciones donde,
  - *No tenemos suficiente información* como para saber cuál elegir
  - Cada decisión nos lleva a un nuevo conjunto de decisiones
  - Alguna sucesión de decisiones (pero posiblemente más de una) pueden ser solución de nuestro problema

- **Entonces** necesitamos un método de búsqueda de esas sucesiones que conduzca a encontrar una que nos convenga

Backtracking: Resumen

- Características del problema:
  - La solución debe poder expresarse como una n-tupla,

    $$(x_1,x_2,x_3,\ldots,x_n),$$

    donde cada x_i es seleccionado de un conjunto finito S_i

  - El problema se (re)formula como la búsqueda de aquella tupla que maximiza (minimiza) un determinado criterio $P(x_1, \ldots, x_n)$

- *Backtracking* es un método de búsqueda sistemática de la solución óptima al problema

Backtracking Vs Fuerza Bruta

FUERZA BRUTA

- Problema:
  - Generar todas las posibles combinaciones de *n* bits
- Aplicaciones:
  - Selección de elementos en un conjunto
    - Selección de actividades
    - Mochila
    - Etc.

00000
00001
00010
00011
00100
00101
00110
00111
....

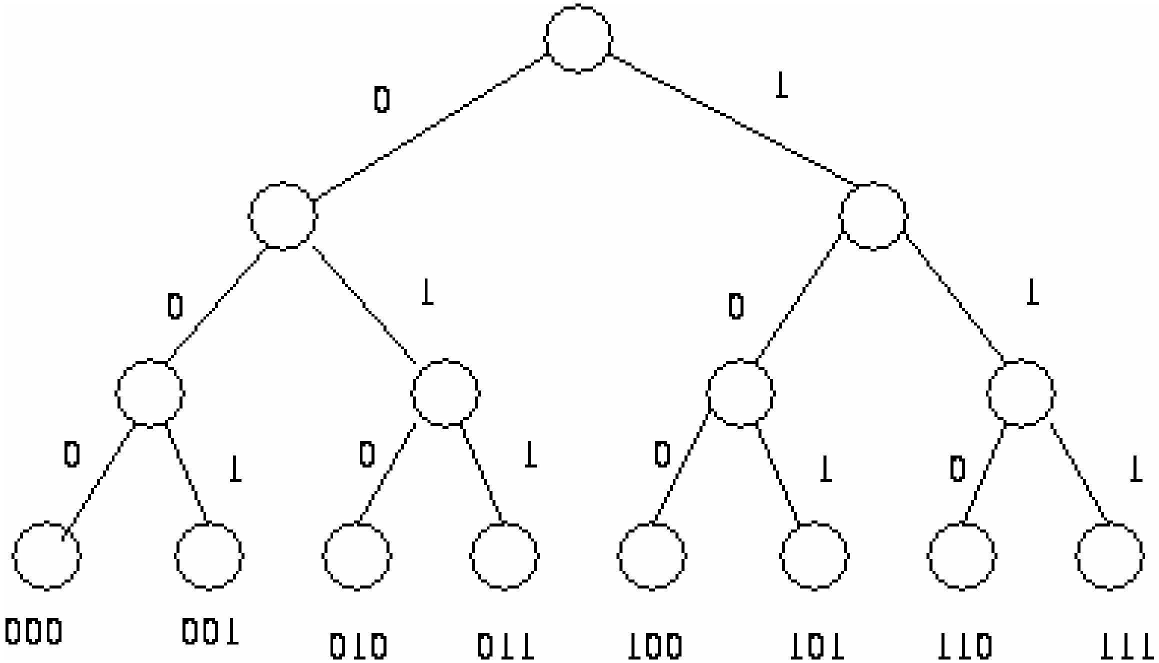
Backtracking Vs Fuerza Bruta

FUERZA BRUTA: ANÁLISIS DE EFICIENCIA

- Ecuación de Recurrencia,

  $T(n) = 2\,T(n-1) + 1$, es del orden $O(2^n)$

Arbol de Recurrencia



Backtracking Vs Fuerza Bruta

¿Qué hemos visto?

- Se impone una estructura de árbol sobre el conjunto de posibles soluciones (espacio de soluciones)
- La forma en la que se generan las soluciones es equivalente a realizar un recorrido en pre-orden del árbol, el espacio de soluciones
- Se procesan las hojas (que se corresponden con soluciones completas)
- Pregunta:
  - ¿Se puede mejorar el proceso? ¿Cuando? ¿Cómo?

BK y *BB* !!!!

Backtracking

- ¿Se puede mejorar el proceso?
  - Sí, eliminando la necesidad de alcanzar una hoja para procesar
- ¿Cuándo?
  - Cuando para un nodo interno del árbol podemos asegurar que no alcanzamos una solución (no nos lleva a nodos hoja útiles), entonces podemos podar la rama
- ¿Cómo?
  - Realizamos una vuelta atrás (*backtracking*)

VENTAJA: Alcanzamos la misma solución con menos pasos

Diferencias con otras Técnicas

- En los algoritmos *greedy* se construye la solución buscada, aprovechando la posibilidad de calcularla a trozos. Pero, con *backtracking* la elección de un sucesor en una etapa no implica su elección definitiva

- El tipo de problemas con el que estamos tratando no se puede dividir en subproblemas independientes. No es aplicable divide y vencerás

Backtracking: Notación

- **Solución Parcial:** Vector solución para el que aún no se han asignado todos sus componentes

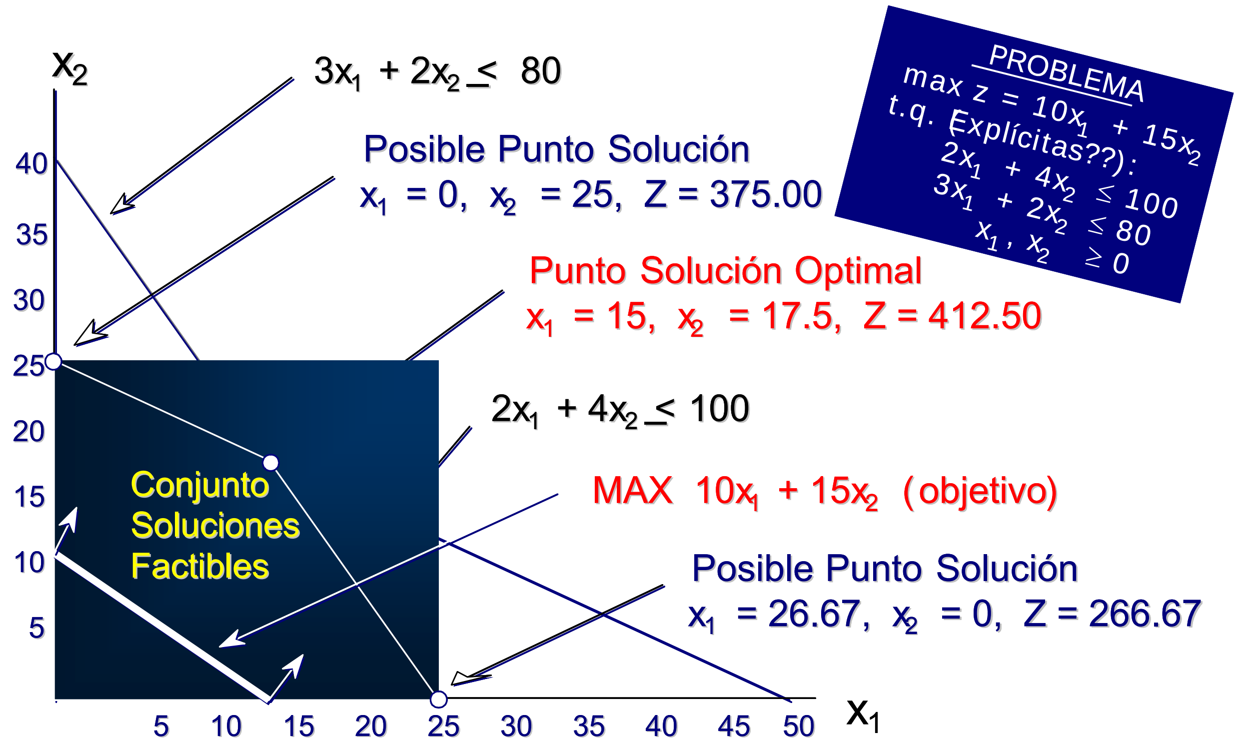
- **Función de Poda**: Aquella función que nos permite identificar cuando una solución parcial no conduce a una solución del problema

- **Restricciones Explícitas:** Reglas que restringen el conjunto de valores que puede tomar cada una de las componentes x_i del vector solución (determinan el espacio de soluciones). Ejemplos comunes,
  - $x_i \geq 0 \quad \Rightarrow \quad S_i = \{n^{os} \text{ reales no negativos}\}$
  - $x_i = 0,1 \quad \Rightarrow \quad S_i = \{0,1\}$
  - $l_i \leq x_i \leq u_i \quad \Rightarrow \quad S_i = \{a: l_i \leq a \leq u_i\}$

Backtracking: Notación

- **Restricciones Implícitas**: Son aquellas que determinan cuando una solución parcial nos puede llevar a una solución —verifica la función criterio $P(x_1, ..., x_n)$—
  - Describen la forma en que se relacionan las x_i

Ejemplo: Interpretación de las restricciones



El problema de las ocho reinas

- Un clásico problema combinatorio es el de colocar ocho reinas en una tablero de ajedrez de modo que no haya dos que se ataquen, es decir, que estén en la misma fila, columna o diagonal
- Las filas y columnas se numeran del 1 al 8
- Las reinas se numeran del 1 al 8

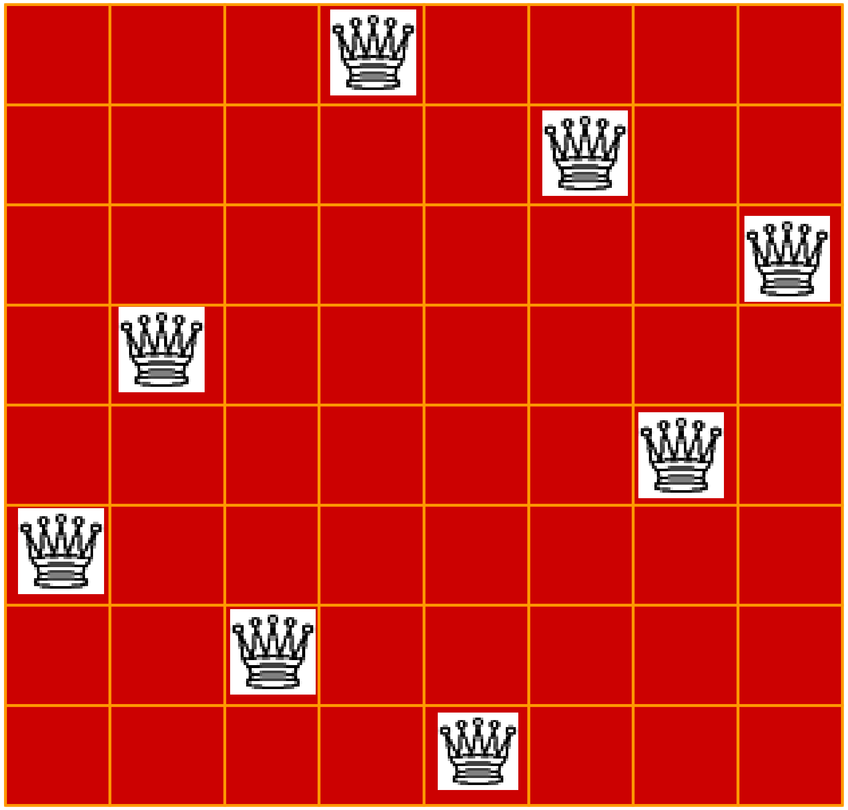
x			x			x	
	x		x		x		
		x	x	x			
x	x	x	Q	x	x	x	x
		x	x	x			
	x		x		x		
x			x			x	
			x				x

Como cada reina debe estar en una fila diferente, sin perdida de generalidad podemos suponer que la reina *i* se coloca en la fila *i*.

Todas las soluciones para este problema, pueden representarse como 8 tuplas $(x_1,...,x_n)$ en las que x_i es la columna en la que se coloca la reina *i*.

El problema de las ocho reinas

- Las restricciones explícitas son $S_i = \{1,2,3,4,5,6,7,8\},\ 1 \leq i \leq n$
- Espacio solución con $8^8 = 2^{24} = 16M$ tuplas
- Restricciones implícitas: ningún par de x_i puedan ser iguales (todas las reinas deben estar en columnas diferentes). Ningún par de reinas pueden estar en la misma diagonal
- La primera de estas dos restricciones implica que todas las soluciones son permutaciones de $(1,2,3,4,5,6,7,8)$
- Esto lleva a reducir el tamaño del espacio solución de 8^8 tuplas a $8! = 40,320$



Una posible solución del problema es la (*4,6,8,2,7,1,3,5*)

Problema de la suma de subconjuntos

- Dados $n+1$ números positivos:

  w_i, $1 \leq i \leq n$, y uno mas M,

- se trata de encontrar todos los subconjuntos de números w_i cuya suma valga M

- Por ejemplo, si $n = 4$, $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$ y $M = 31$, entonces los subconjuntos buscados son $(11, 13, 7)$ y $(24, 7)$

Problema de la suma de subconjuntos

- Para representar la solución podríamos notar el vector solución con los índices de los correspondientes w_i
- Las dos soluciones se describen por los vectores *(1,2,4)* y *(3,4)*
- Todas las soluciones son *k*-tuplas $(x_1, x_2,...,x_k)$, $1 \leq k \leq n$, y soluciones diferentes pueden tener tamaños de tupla diferentes
- Restricciones explícitas: $x_i \in \{j: j \text{ es entero y } 1 \leq j \leq n\}$
- Restricciones implícitas: que no haya dos iguales y que la suma de los correspondientes w_i sea M
- Además, como por ejemplo *(1,2,4)* y *(1,4,2)* representan el mismo subconjunto, otra restricción implícita que hay que imponer es que $x_i < x_{i+1}$, para $1 \leq i < n$

Problema de la suma de subconjuntos

- Puede haber diferentes formas de formular un problema de modo que todas las soluciones sean tuplas que satisfacen algunas restricciones

- Otra formulación del problema:
  - Cada subconjunto solución se representa por una n-tupla $(x_1,...,x_n)$ tal que $x_i \in \{0,1\}$, $1 \leq i < n$, con $x_i = 0$ si w_i no se elige y $x_i = 1$ si w_i se elige
  - Las soluciones del anterior caso son $(1,1,0,1)$ y $(0,0,1,1)$
  - Esta formulación expresa todas las soluciones usando un tamaño de tupla fijo

- Se puede comprobar que para estas dos formulaciones, el espacio solución consiste en ambos casos de 2^4 tuplas distintas

Índice

I. LA TÉCNICA BACKTRACKING

1. Introducción: El método General
   - Resolución de problemas cuando la solución se puede expresar como una n-tupla
   - Ejemplo: El problema de las 8 reinas
   - Ejemplo: La suma de subconjuntos
2. Espacio de soluciones: Organización del Árbol
   - Ejemplo: Espacio solución para el problema de las N-reinas (N=4)
   - Ejemplo: Espacio solución para la suma de subconjuntos
   - Terminología utilizada para la organización en árbol
   - Ejemplo: Backtracking en el problema de las 4 reinas
3. Procedimiento *Backtracking*
   - Procedimiento Iterativo
   - Procedimiento Recursivo

Espacios de soluciones

- Los algoritmos backtracking determinan las soluciones del problema buscando en el espacio de soluciones del caso considerado sistemáticamente
- Esta búsqueda se puede representar usando una organización en árbol para el espacio solución.
- Para un espacio solución dado, pueden caber muchas organizaciones en árbol.
- Los siguientes ejemplos examinan algunas de las formas posibles para estas organizaciones.

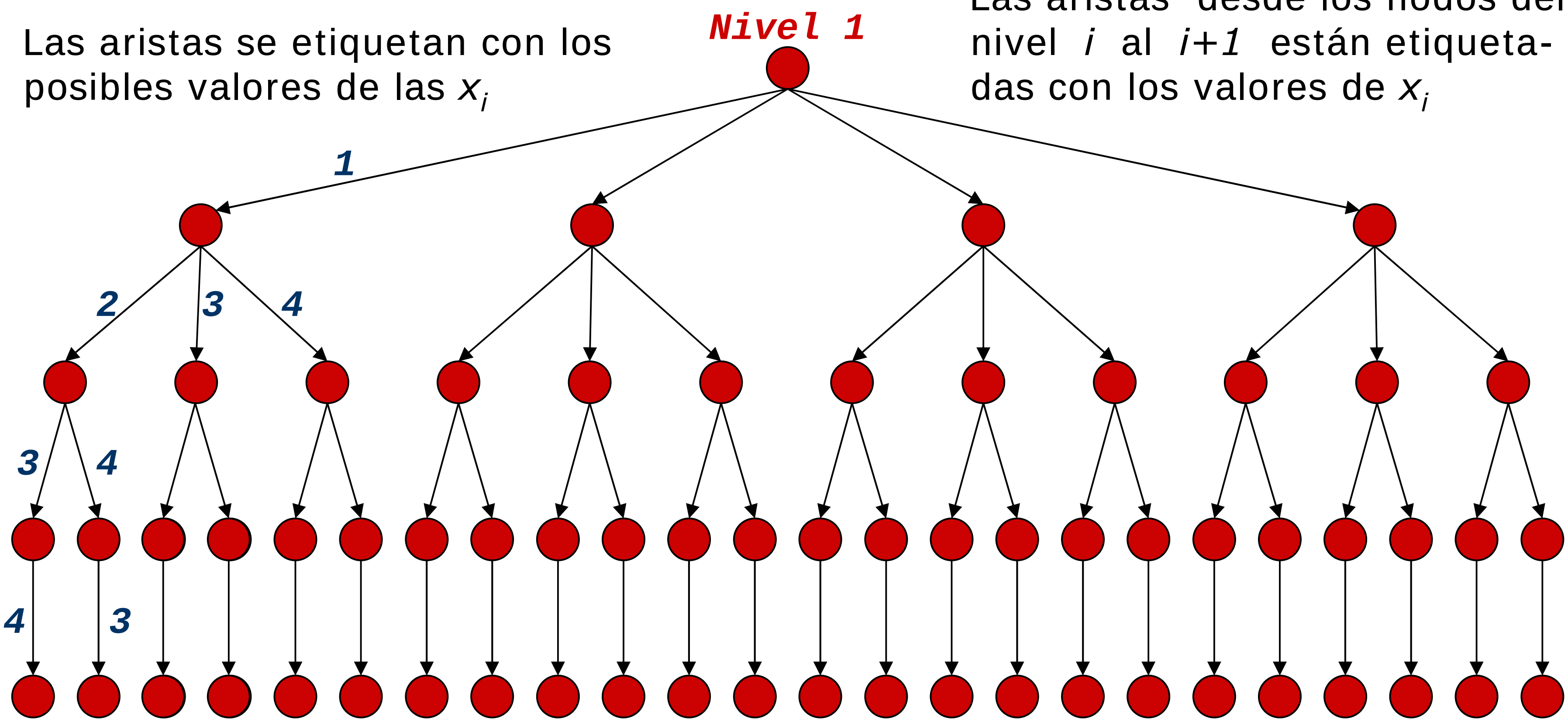
Ejemplo de espacio de soluciones

- La generalización del problema de las *8* reinas es el de las *N* reinas: Colocar *N* reinas en un tablero *NxN* de modo que no haya dos que se ataquen
- Ahora el espacio de soluciones consiste en las *N!* permutaciones de la *N*-tupla *(1,2,...,N)*
- La generalización nos sirve a efectos didácticos para poder hablar del problema de las *4* reinas
- La siguiente figura muestra una posible organización de las soluciones del problema de las *4* reinas en forma de árbol
- A un árbol como ese se le llama Árbol de (búsqueda de soluciones) Permutación

4-Reinas: Arbol de permutación

Las aristas se etiquetan con los posibles valores de las x_i

Las aristas desde los nodos del nivel i al $i+1$ están etiquetadas con los valores de x_i



El subárbol de la izquierda contiene todas las soluciones con $x_1=1$ y $x_2=2$, $x_2=3$, $x_2=4$, ... El espacio de soluciones está definido por todos los caminos desde el nodo raíz a un nodo hoja. Hay $4! = 24$ nodos hoja en la figura

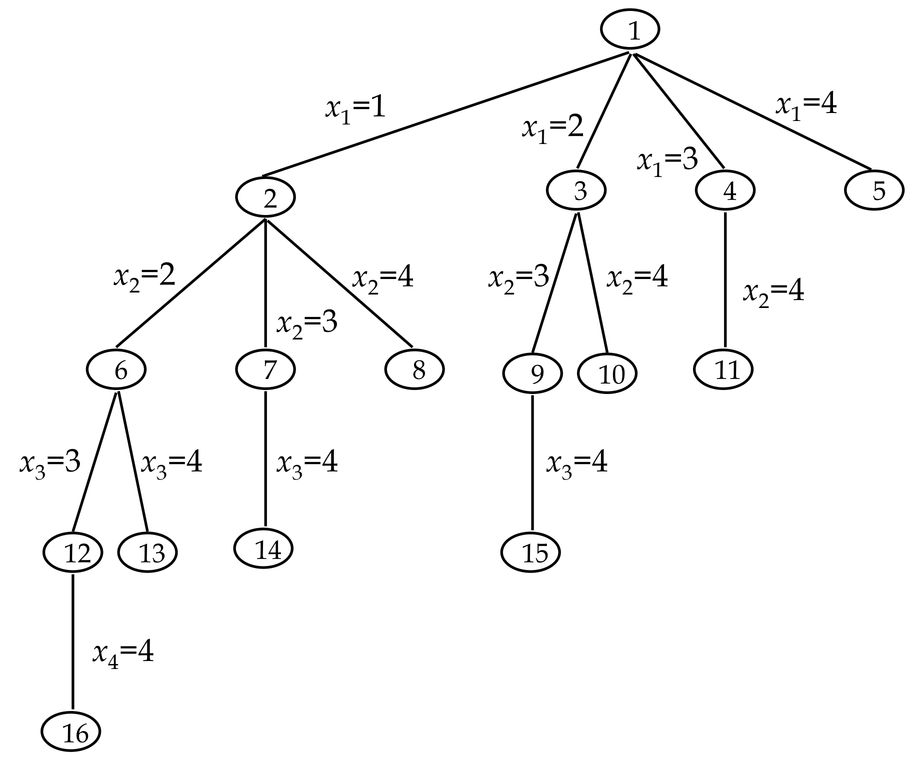
Ejemplos de Árboles en la Suma de Subconjuntos

- Vimos dos posibles formulaciones del espacio solución del problema de la suma de subconjuntos.
  - La primera corresponde a la formulación por el tamaño de la tupla
  - La segunda considera un tamaño de tupla fijo

- Con ambas formulaciones, tanto en este problema como en cualquier otro, el número de soluciones tiene que ser el mismo

Suma de subconjuntos: árbol 1

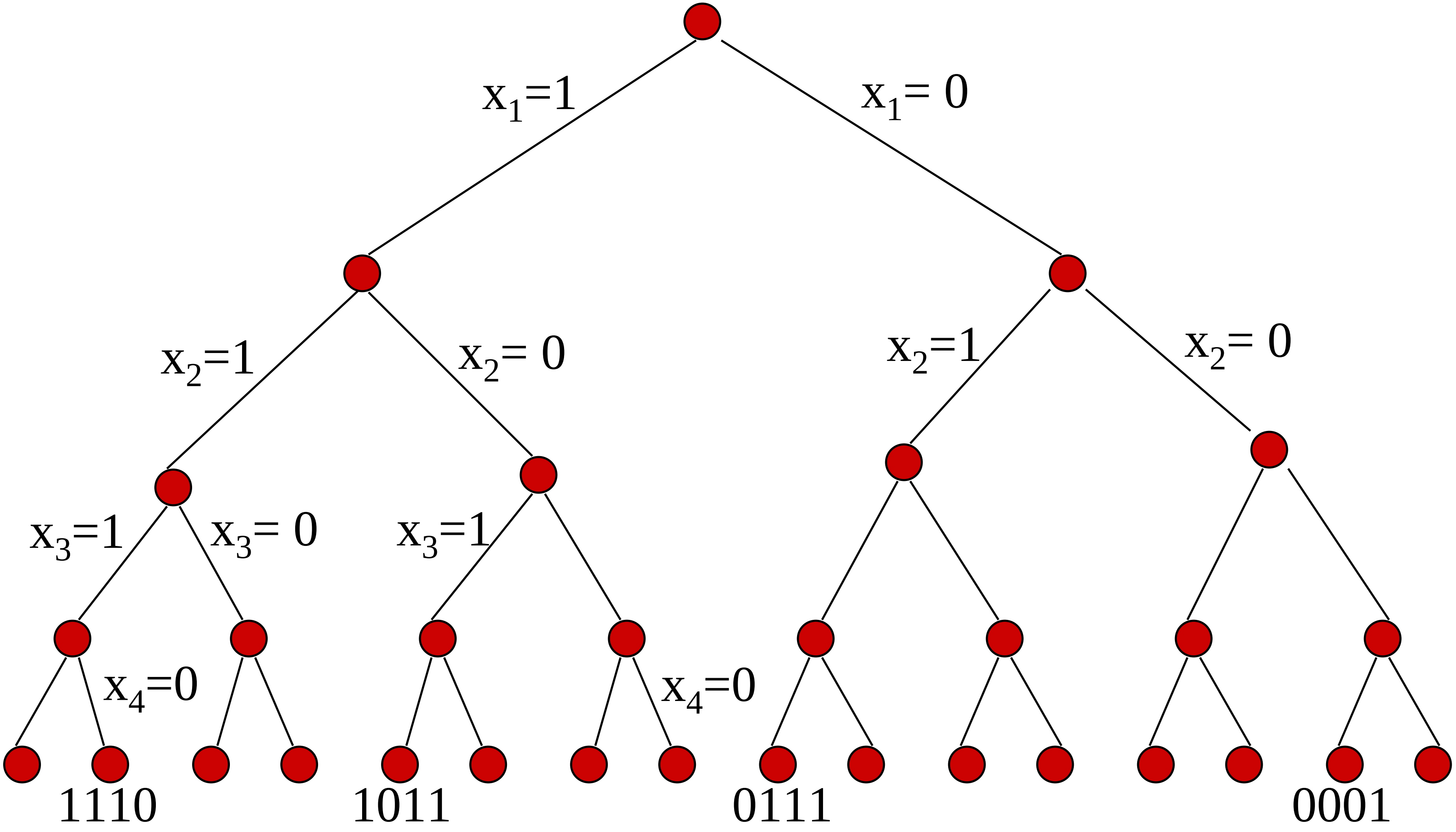
- Las aristas se etiquetan de modo que una desde el nivel de nodos *i* hasta el *i+1* representa un valor para x_i
- En cada nodo, el espacio solución se particiona en espacios subsolución
- Las posibles soluciones son 16 tuplas, que corresponden al camino desde la raíz a cada nodo, *()*, *(1)*, *(12)*, *(123)*, *(1234)*, *(124)*, *(134)*, *(14)*, *(2)*, *(23)*, etc.
- Así, el subarbol de la izquierda define todos los subconjuntos conteniendo w_1, el siguiente todos los que contienen w_2 pero no w_1, etc.



Suma de subconjuntos: árbol 2

- Una arista del nivel i al $i+1$ se etiqueta con el valor de x_i *(0 o 1)*
- Todos los caminos desde la raíz a las hojas definen el espacio solución
- El subarbol de la izquierda define todos los subconjuntos conteniendo w_1, mientras que el de la derecha define todos los subconjuntos que no contienen w_1, etc.
- Consideramos el caso de $n = 4$
- Hay 2^4 nodos hoja, que representan *16* posibles tuplas

Suma de subconjuntos: árbol 2

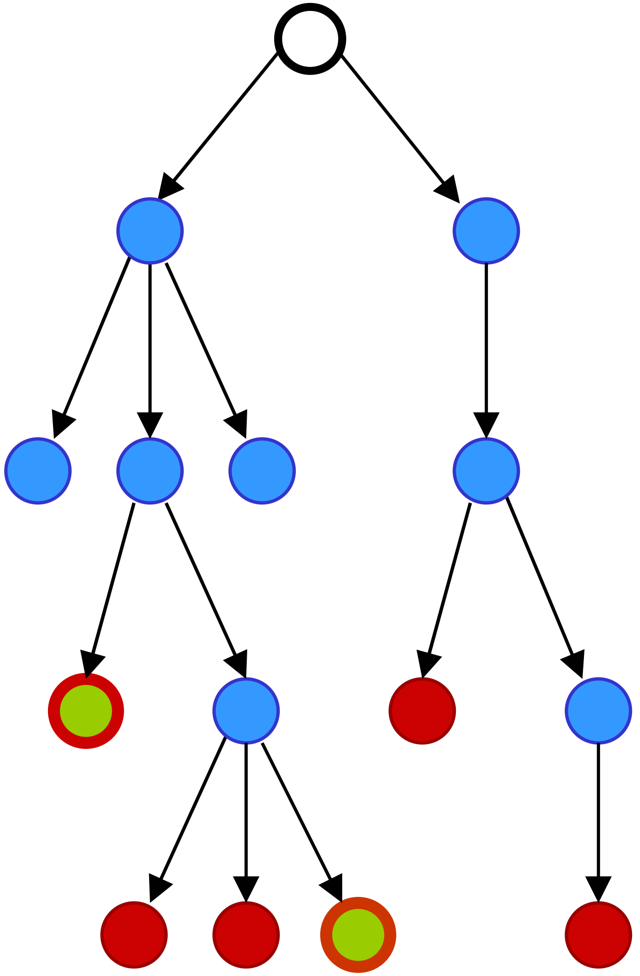


Terminología

- **Estado del problema:** Cada uno de los nodos del árbol
- **Estado solución**: Son aquellos estados (nodos) *S* del problema (árbol) para los que el camino desde la raíz a *S* representa una *n*-tupla en el espacio solución
  - En el primero de los árboles anteriores, todos los nodos son estados solución, mientras que en el segundo sólo los nodos hoja son estados solución
- **Estados respuesta:** Son aquellos estados solución *S* que representan una *n*-tupla del conjunto de soluciones del problema (aquellas que satisfacen las restricciones implícitas)

Generación de estados de un problema

- Cuando se ha concebido un árbol de estados para algún problema, podemos resolver este problema generando sistemáticamente sus estados, determinando cuales de estos son estados solución, y finalmente determinando que estados solución son estados respuesta



Terminología. Generación de Estados

- **Nodo vivo:** Estado del problema que ya ha sido generado, pero del que aún no se han generado todos sus hijos

- **Nodo muerto:** Estado del problema que ya ha sido generado, y o bien se ha podado o bien se han generado todos los descendientes

- ***E*-nodo (nodo de expansión):** Nodo vivo del que actualmente se están generando los descendientes

Generación de estados de un problema

- Hay dos formas diferentes de generar los estados del problema
  - Las dos comienzan con el nodo raíz y generan otros nodos
  - En ambos métodos de generar estados del problema tendremos una lista de nodos vivos

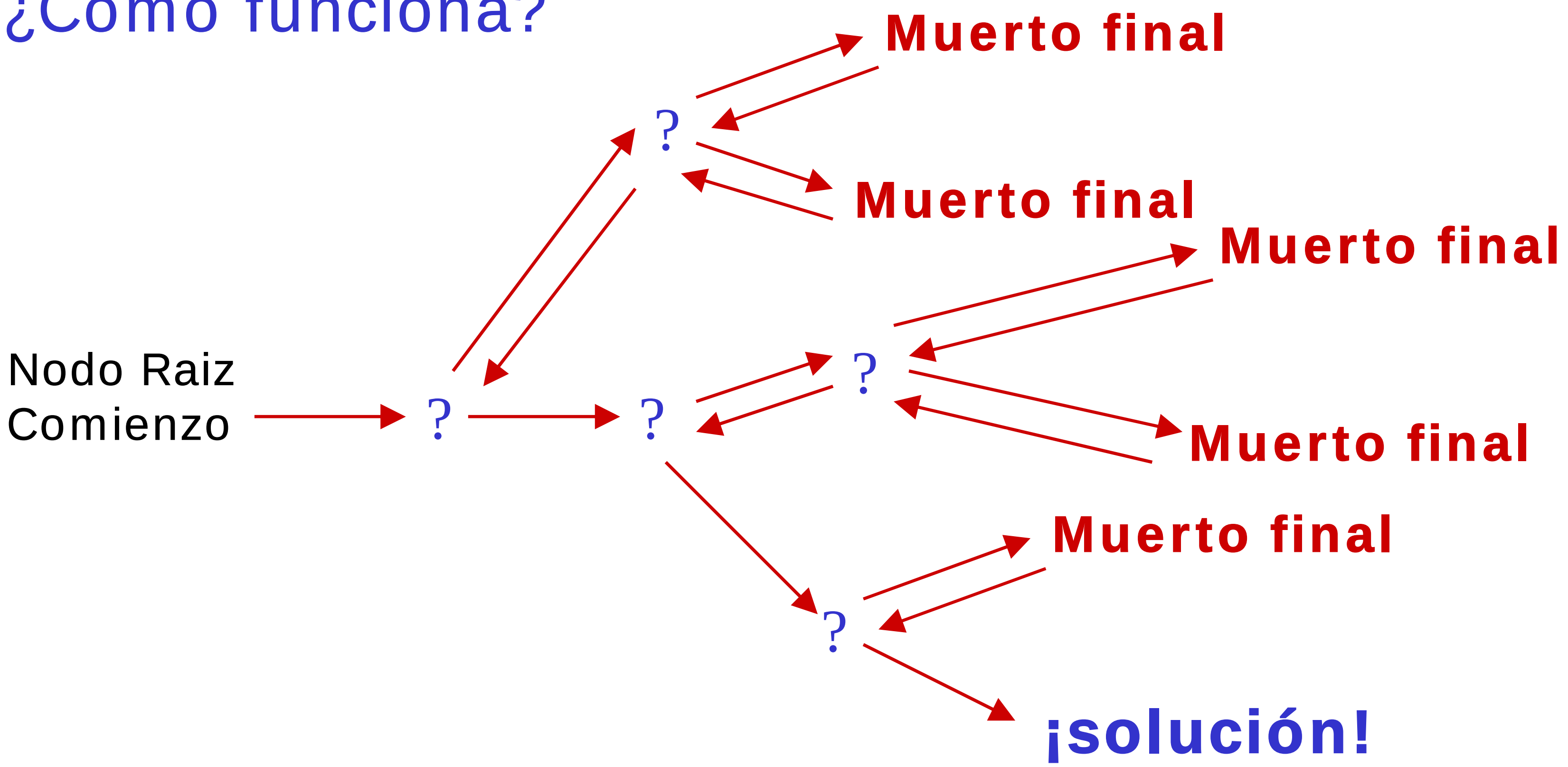
- En el primer método, tan pronto como un nuevo hijo *C* del *E*-nodo en curso *R* ha sido generado, este hijo se convierte en un nuevo *E*-nodo
  - *R* se convertirá de nuevo en *E*-nodo cuando el subárbol *C* haya sido explorado completamente
  - Esto corresponde a una generación primero en profundidad de los estados del problema

- Adicionalmente se usan funciones de acotación para matar nodos vivos sin tener que generar todos sus nodos hijos

Generación de estados de un problema

- A esta forma de generación (exploración) se le llama ***Backtracking***

¿Cómo funciona?

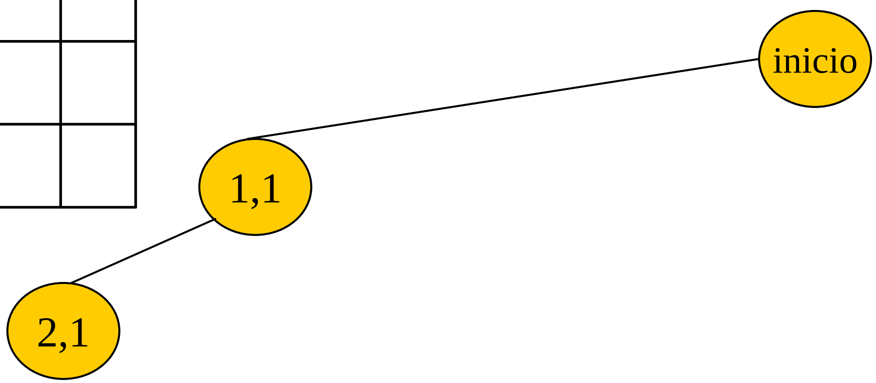


Generación de estados de un problema

- En el segundo método, el *E*-nodo permanece como *E*-nodo hasta que se hace nodo muerto
- También se usan funciones de acotación para detener la exploración en un subárbol
- El método se adapta muy bien a la resolución de problemas de optimización combinatoria (espacio de soluciones discreto): Problema de la Mochila, Soluciones enteras, etc
- En este método, la construcción de las funciones de acotación es (casi) más importante que los mecanismos de exploración en si mismos
- A esta segunda forma de generación (exploración) se le llama ***Branch and Bound***

Ejemplo *BK*... para ***n*** = *4*

R			
X			

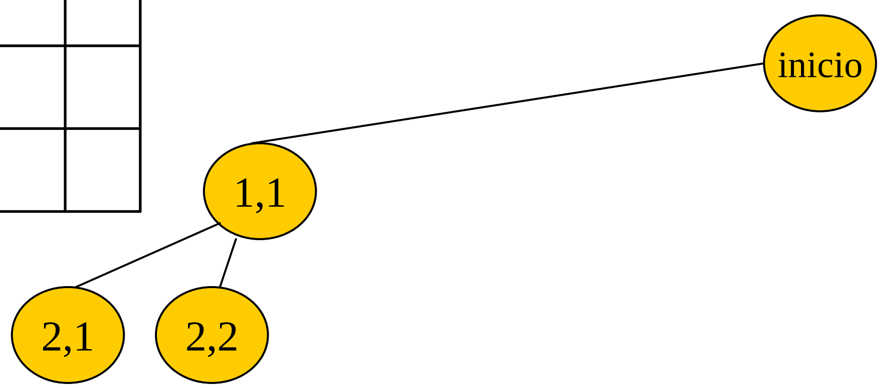


La estrategia ASEGURA
no ocupar el mismo renglón

NO se cumple el criterio
(misma columna)

Ejemplo... para *n* = *4*

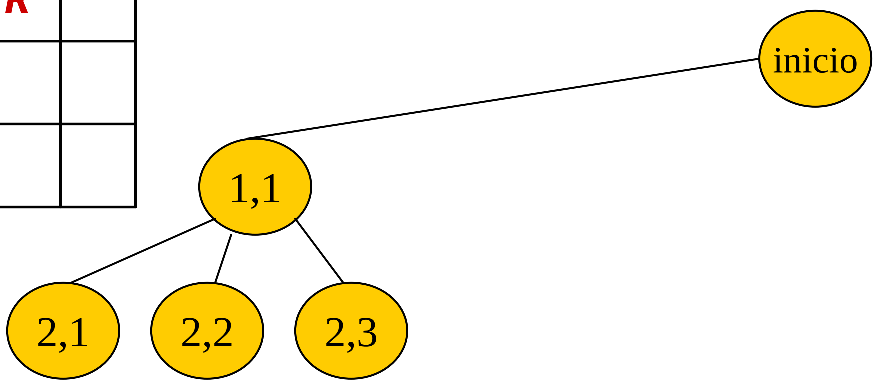
R			
x	x		



NO se cumple el criterio (misma diagonal)

Ejemplo... para *n* = *4*

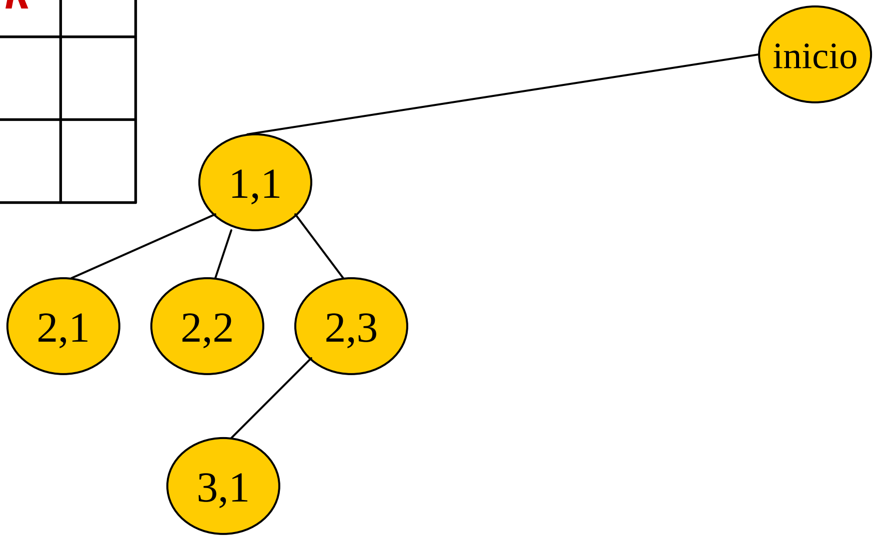
R			
x	*x*	*R*	



OK... adelante en la búsqueda !

Ejemplo... para ***n*** = *4*

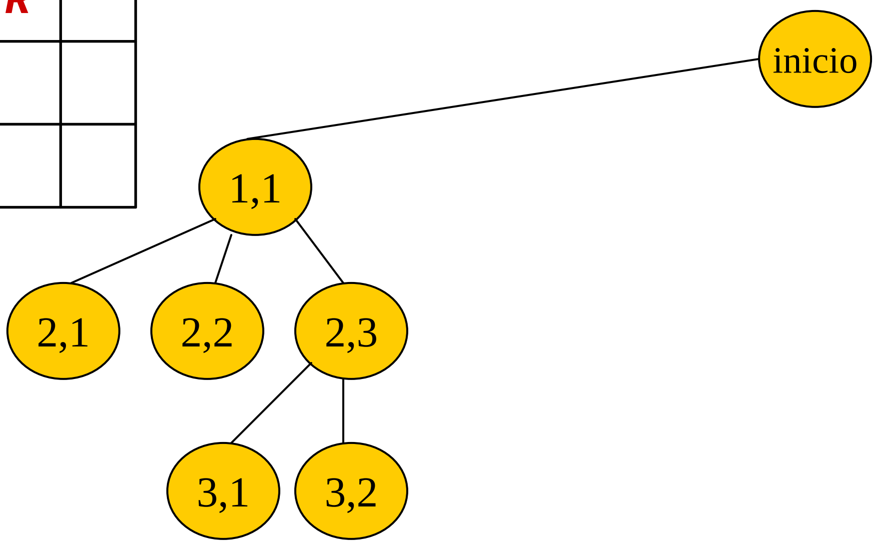
R			
x	***x***	***R***	
x			



NO se cumple el criterio (misma columna)

Ejemplo... para ***n*** = *4*

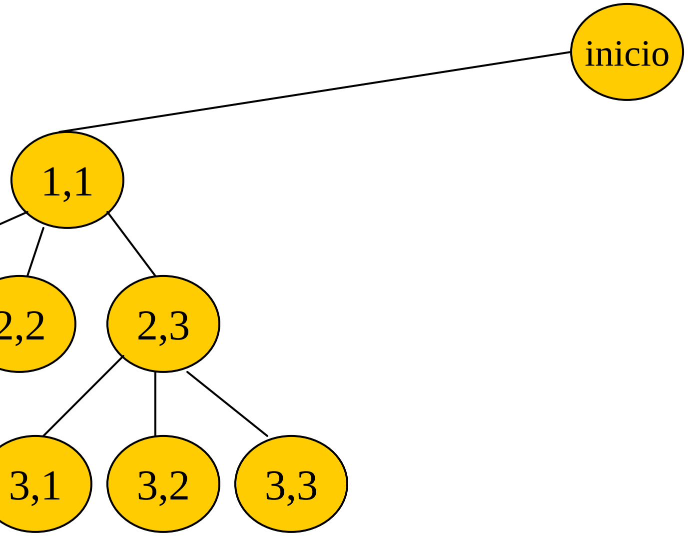
R			
x	x	R	
x	x		



NO se cumple el criterio (misma diagonal que 2,3)

Ejemplo… para ***n*** = *4*

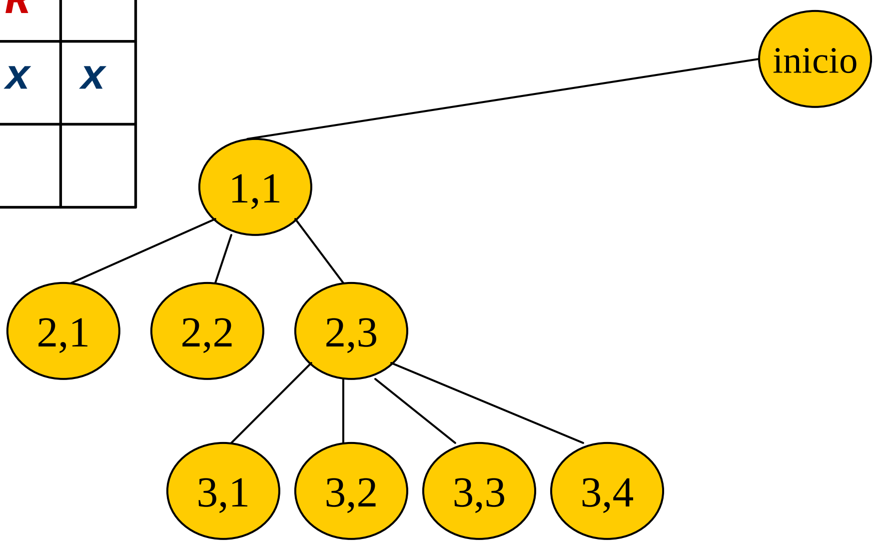
R			
x	***x***	***R***	
x	***x***	***x***	



NO se cumple el criterio
(misma diagonal que 1,1)

Ejemplo... para ***n*** = *4*

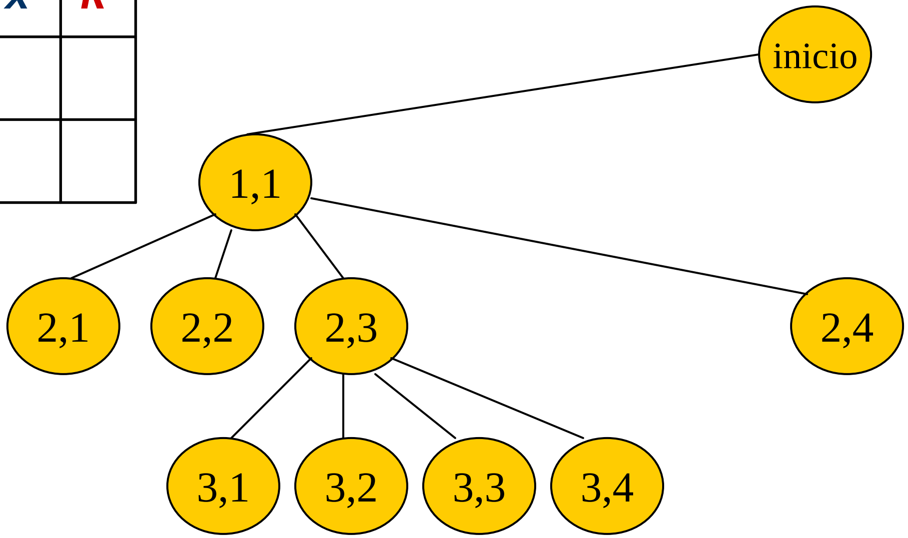
R			
x	***x***	***R***	
x	***x***	***x***	***x***



NO se cumple el criterio
(misma diagonal que 2,3)

Ejemplo… para *n* = *4*

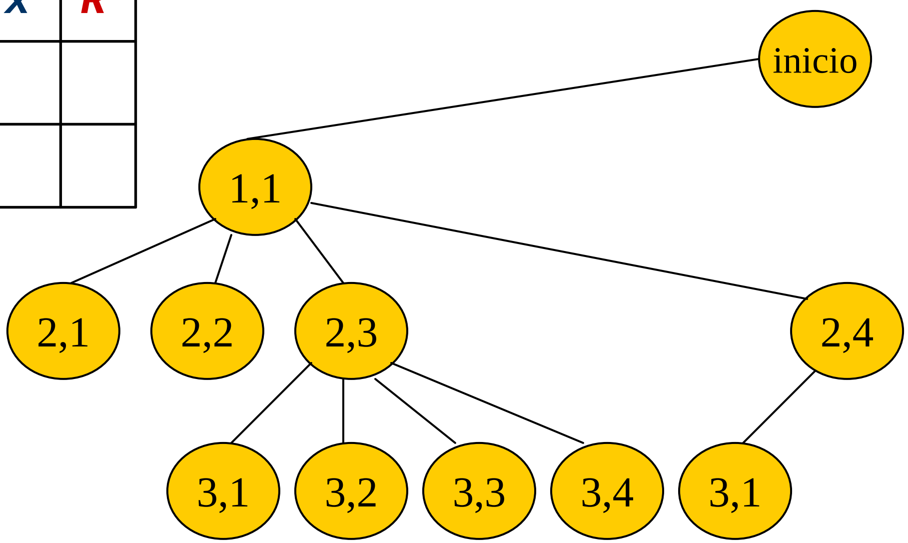
R			
x	***x***	***x***	***R***



OK… adelante con la búsqueda!

Ejemplo… para ***n*** = *4*

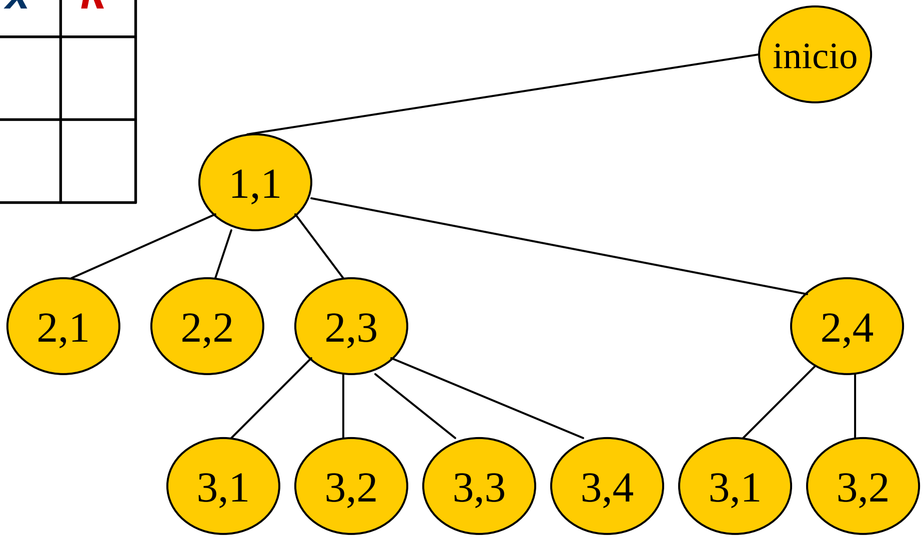
R			
x	*x*	*x*	*R*
x			



NO se cumple criterio (misma columna)

Ejemplo… para ***n*** = *4*

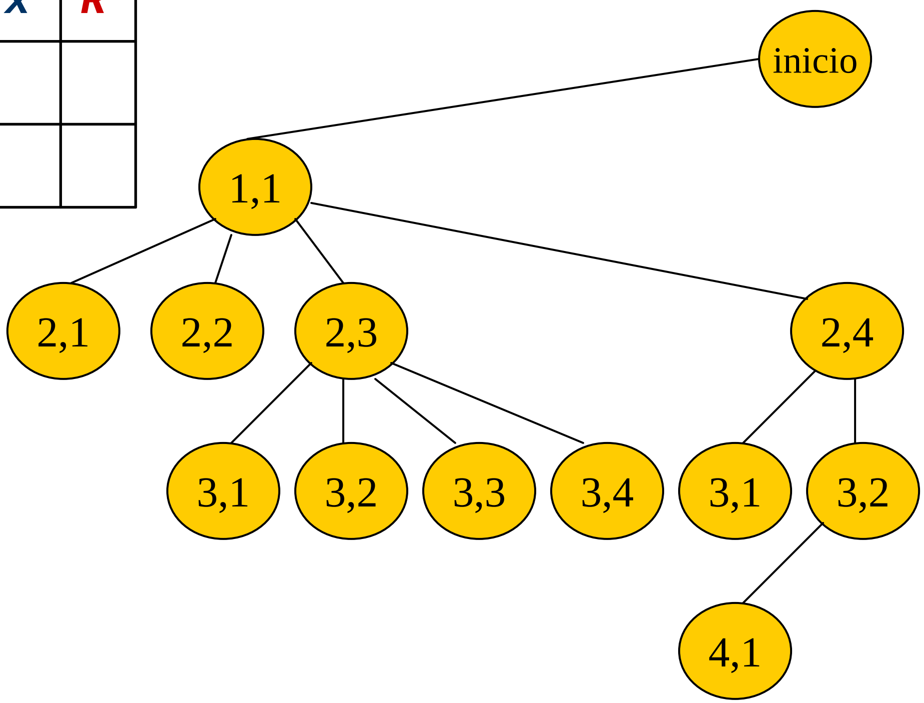
R			
x	*x*	*x*	*R*
x	*R*		



OK… adelante con la búsqueda!

Ejemplo... para ***n*** = *4*

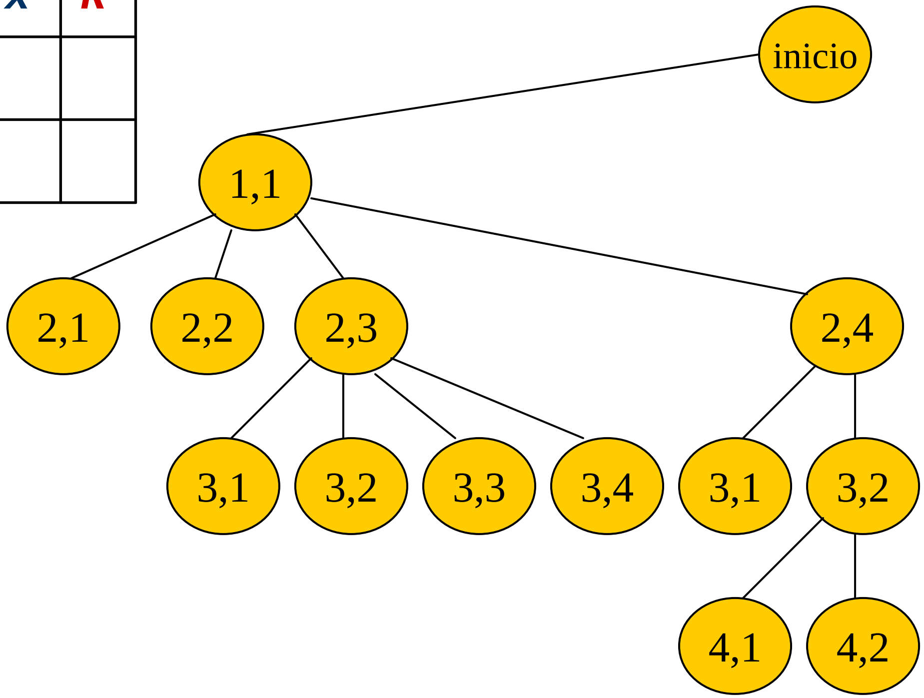
R			
x	***x***	***x***	***R***
x	***R***		
x			



NO se cumple criterio (misma columna)

Ejemplo... para *n* = *4*

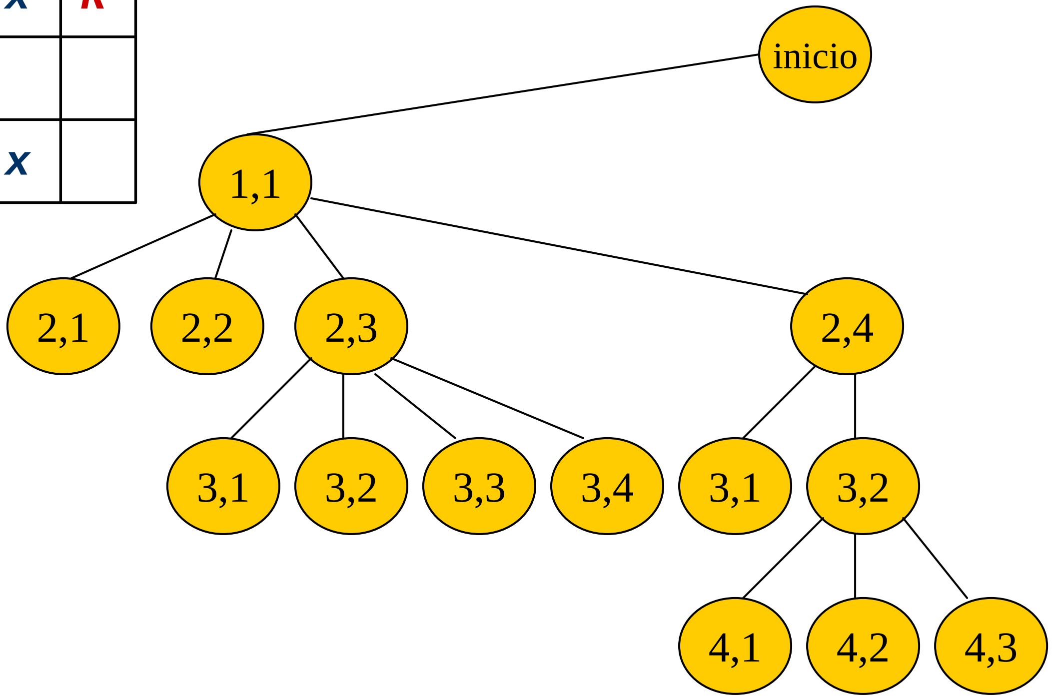
R			
x	***x***	***x***	***R***
x	***R***		
x	***x***		



NO se cumple criterio (misma columna)

Ejemplo… para *n* = *4*

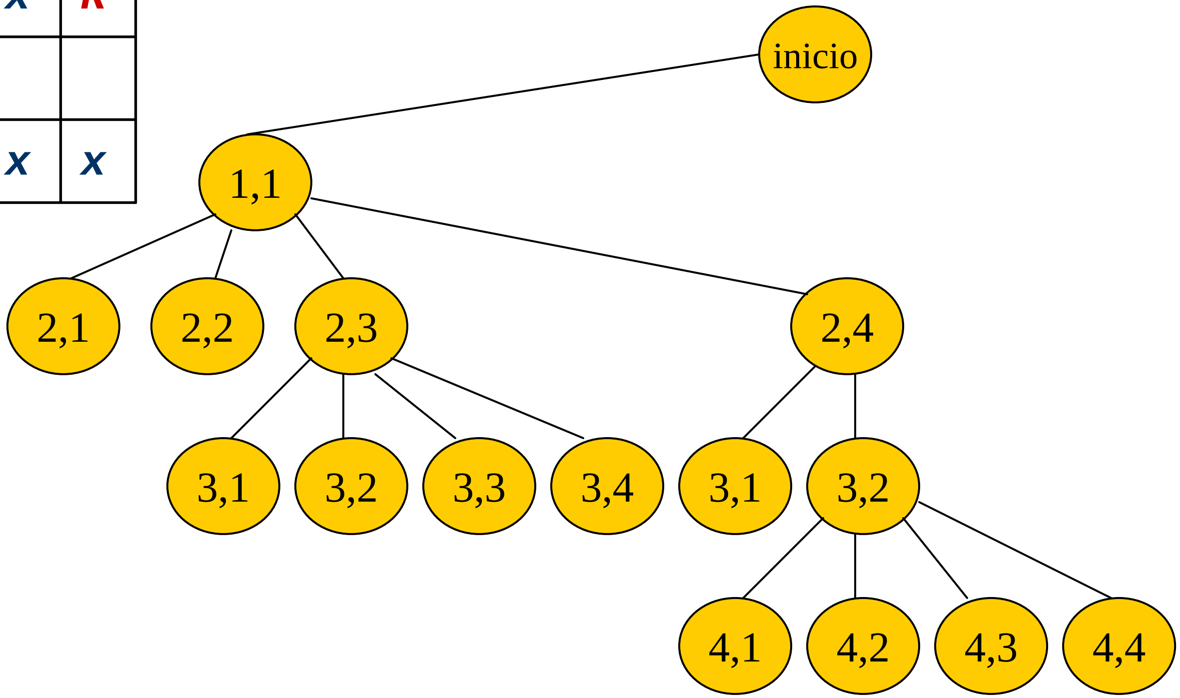
R			
x	x	x	R
x	R		
x	x	x	



NO se cumple criterio
(misma diagonal 3,2)

Ejemplo... para ***n*** = *4*

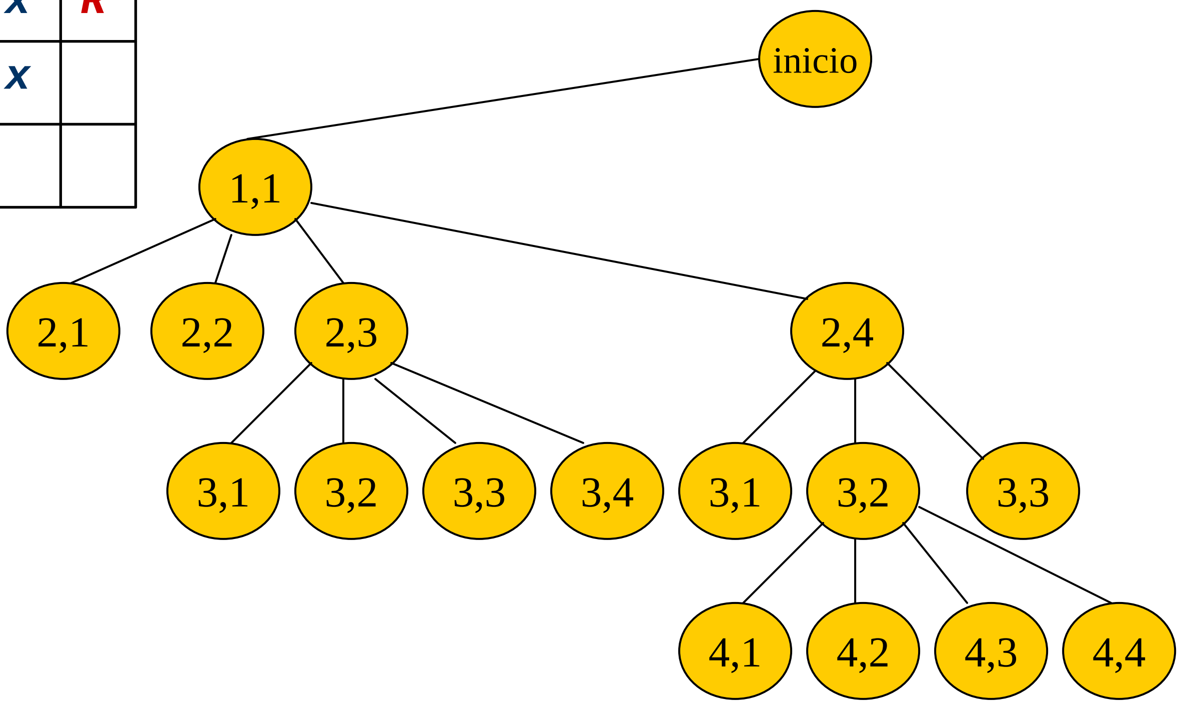
R			
x	*x*	*x*	*R*
x	*R*		
x	*x*	*x*	*x*



NO se cumple criterio (misma columna)

Ejemplo... para ***n*** = *4*

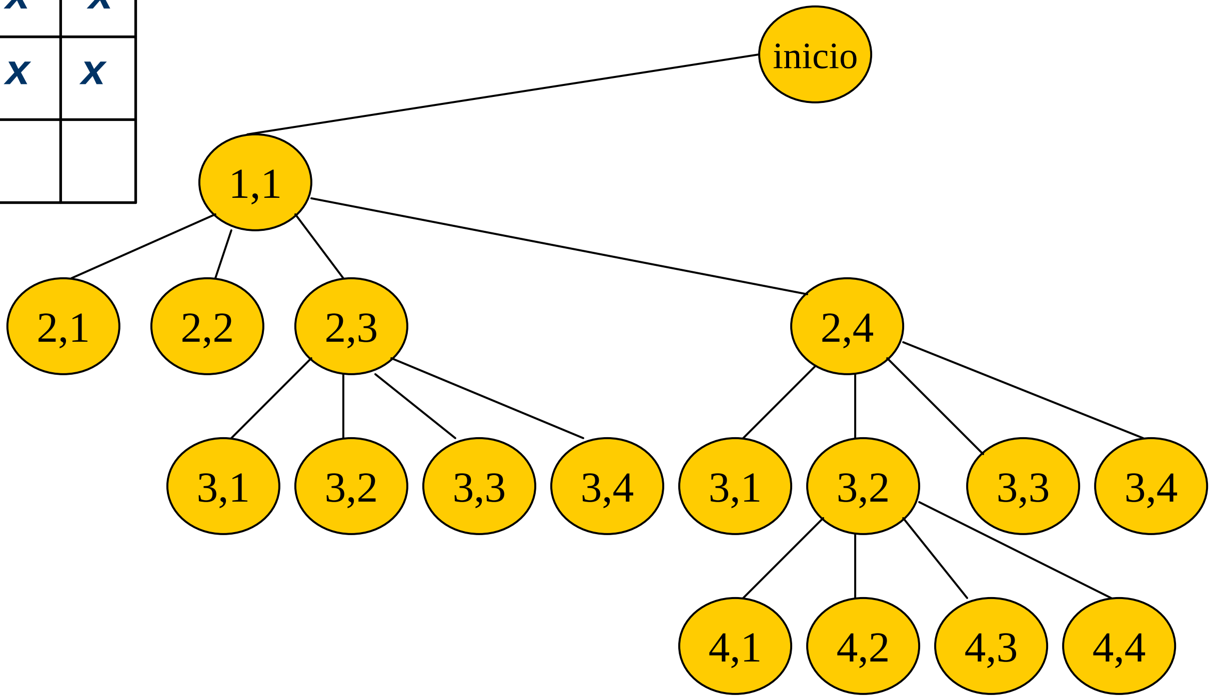
R			
x	***x***	***x***	***R***
x	***x***	***x***	



NO se cumple criterio (misma diagonal)

Ejemplo... para n = 4

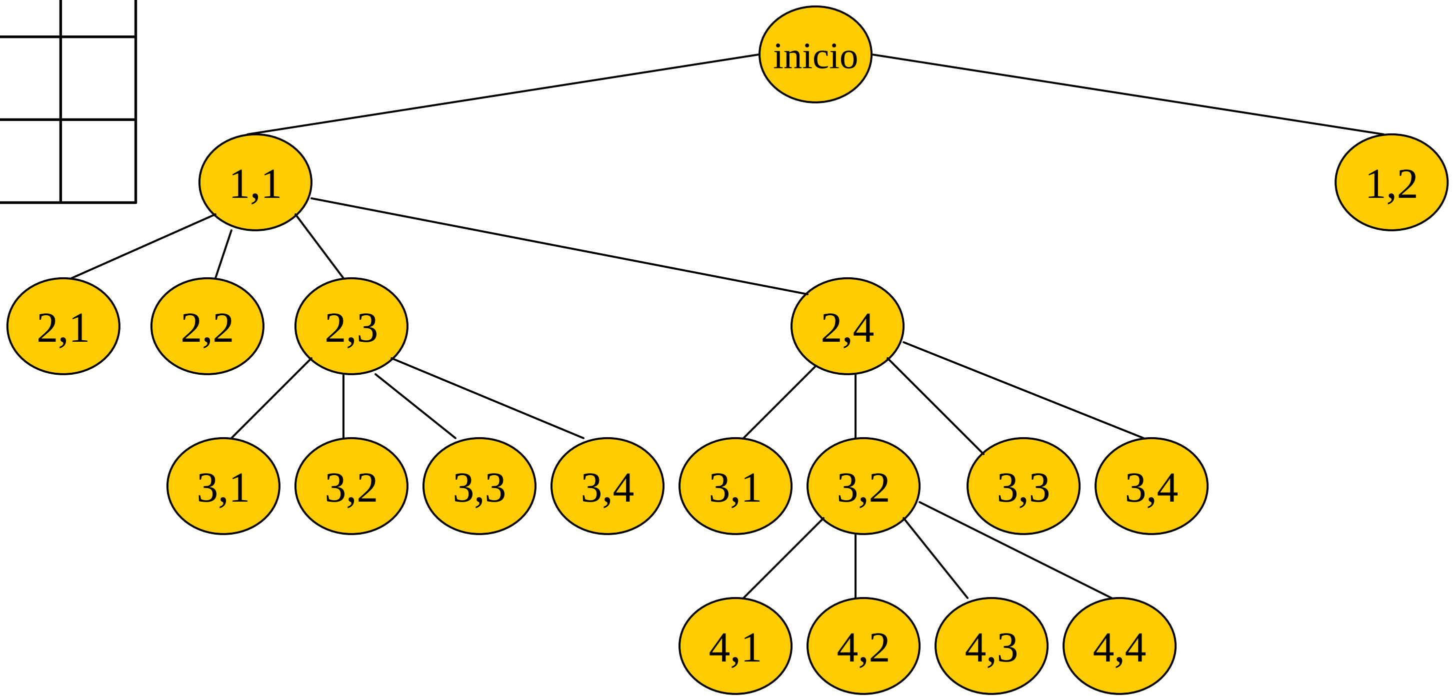
R			
x	*x*	*x*	*x*
x	*x*	*x*	*x*



NO se cumple criterio (misma columna)

Ejemplo… para $n = 4$

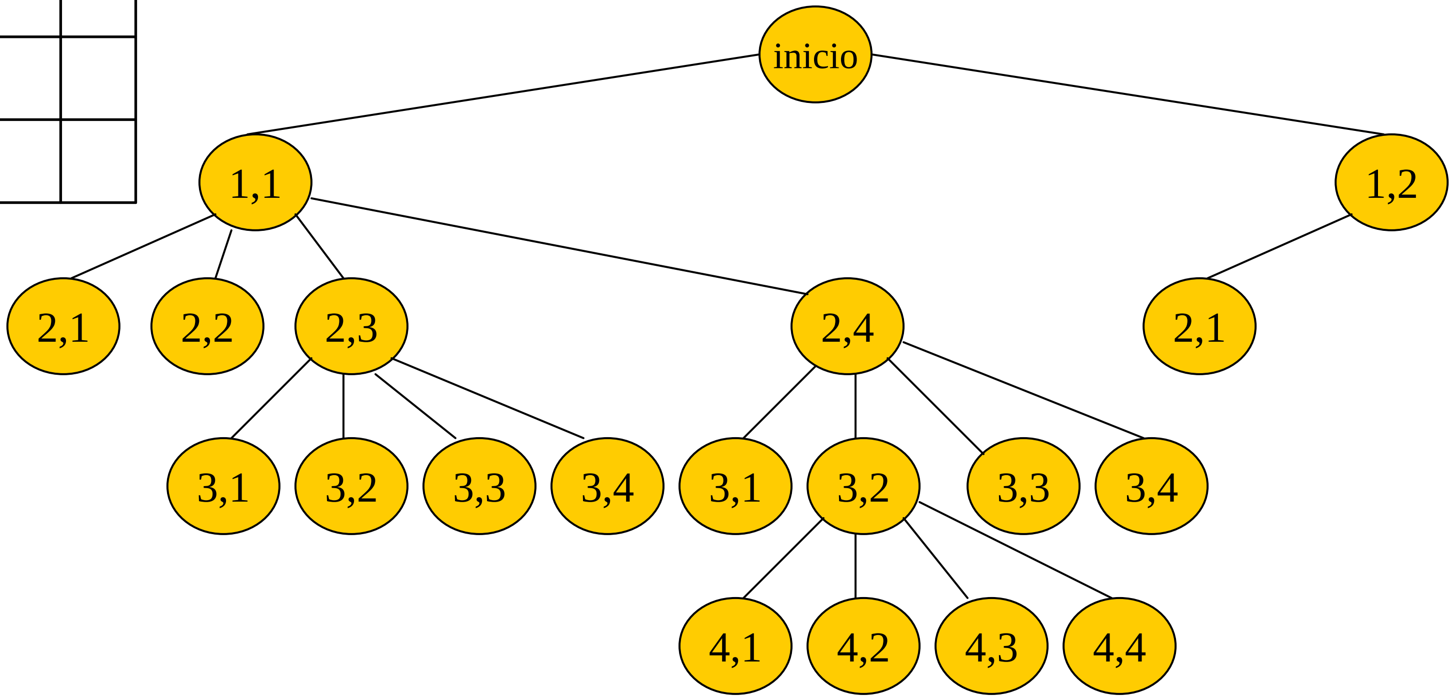
x	***R***		



OK… adelante con la búsqueda!

Ejemplo... para ***n*** = *4*

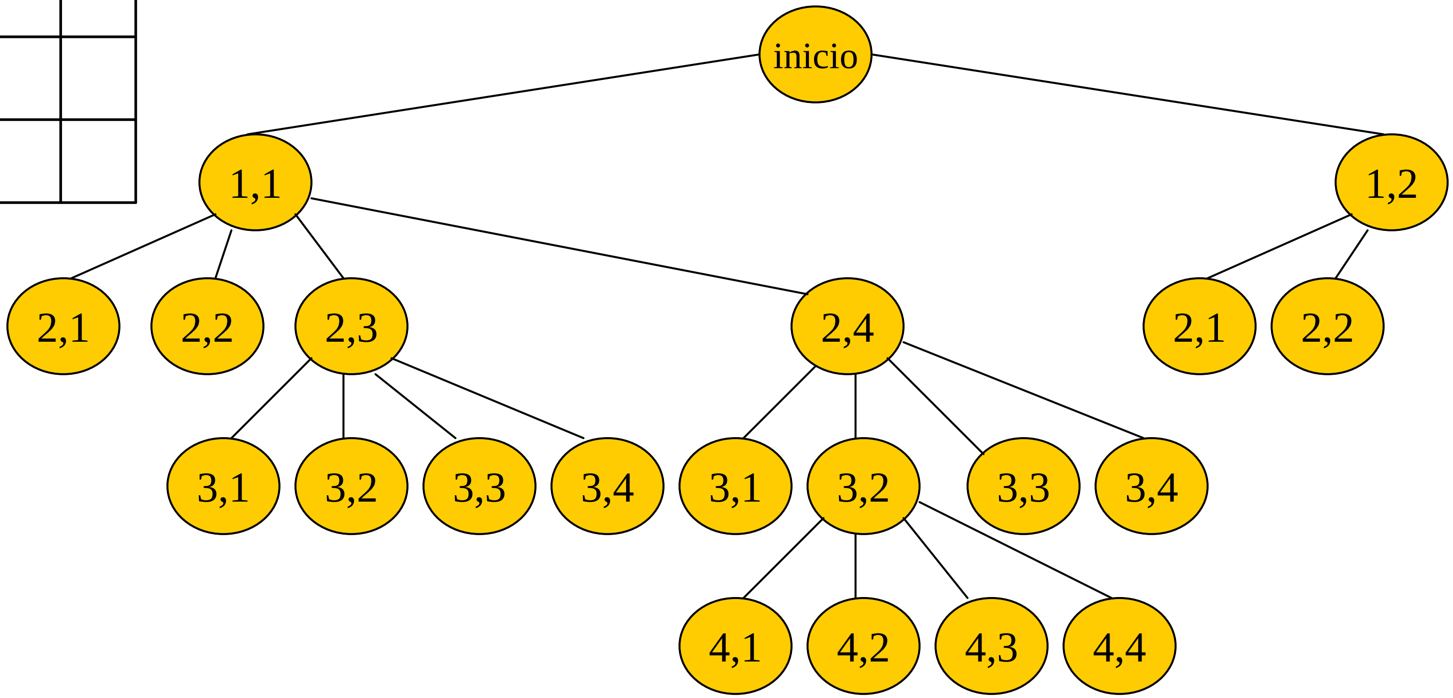
x	*R*		
x			



NO cumple criterio (misma diagonal)

Ejemplo... para n = 4

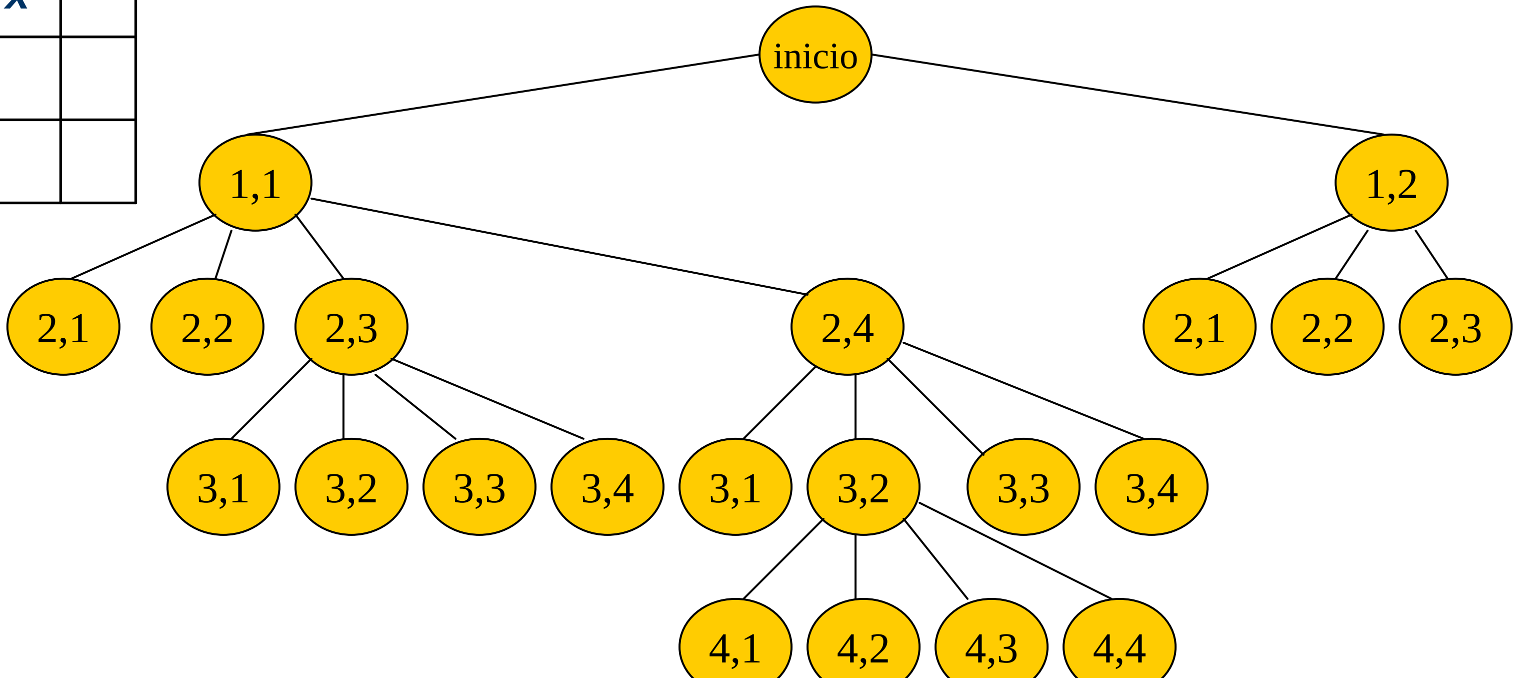
x	*R*		
x	*x*		



NO cumple criterio (misma columna)

Ejemplo… para *n* = *4*

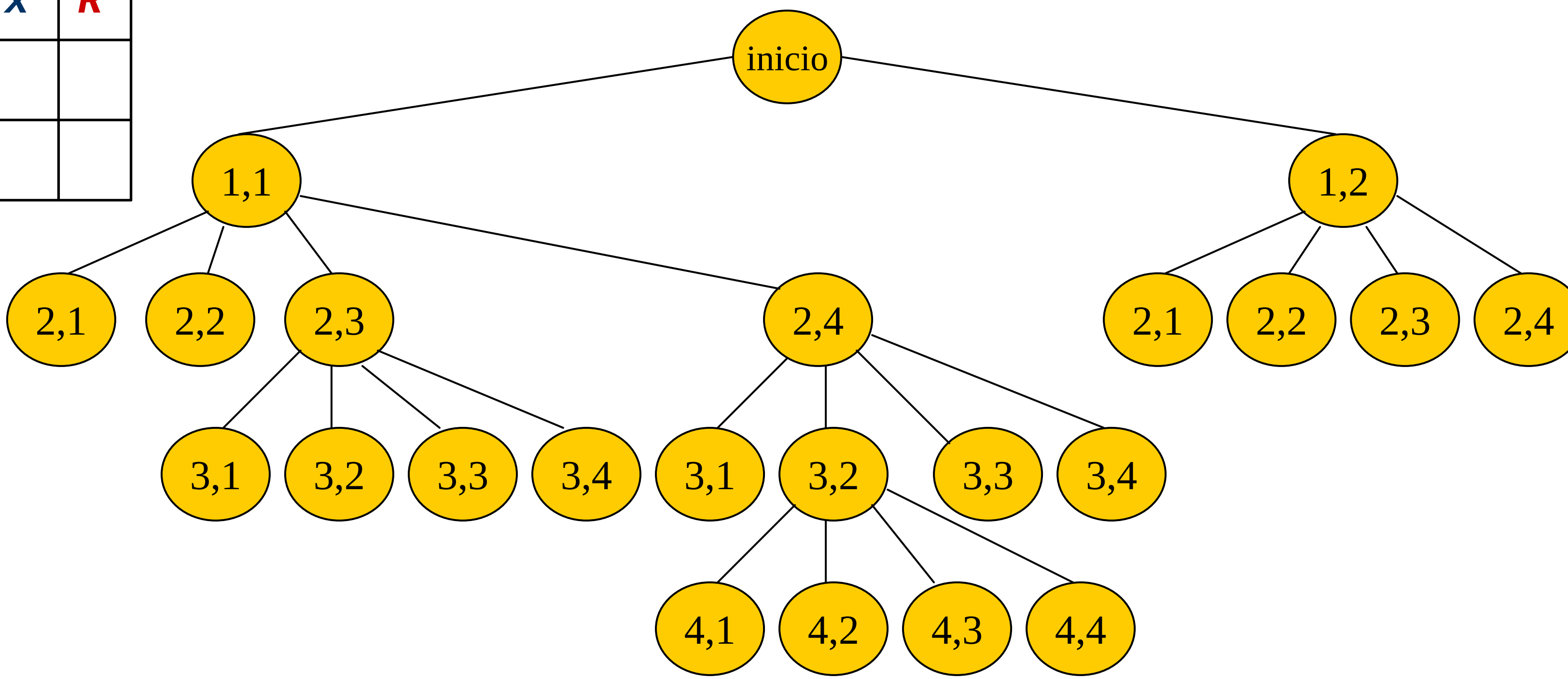
x	*R*		
x	*x*	*x*	



NO cumple criterio
(misma diagonal)

Ejemplo... para ***n*** = *4*

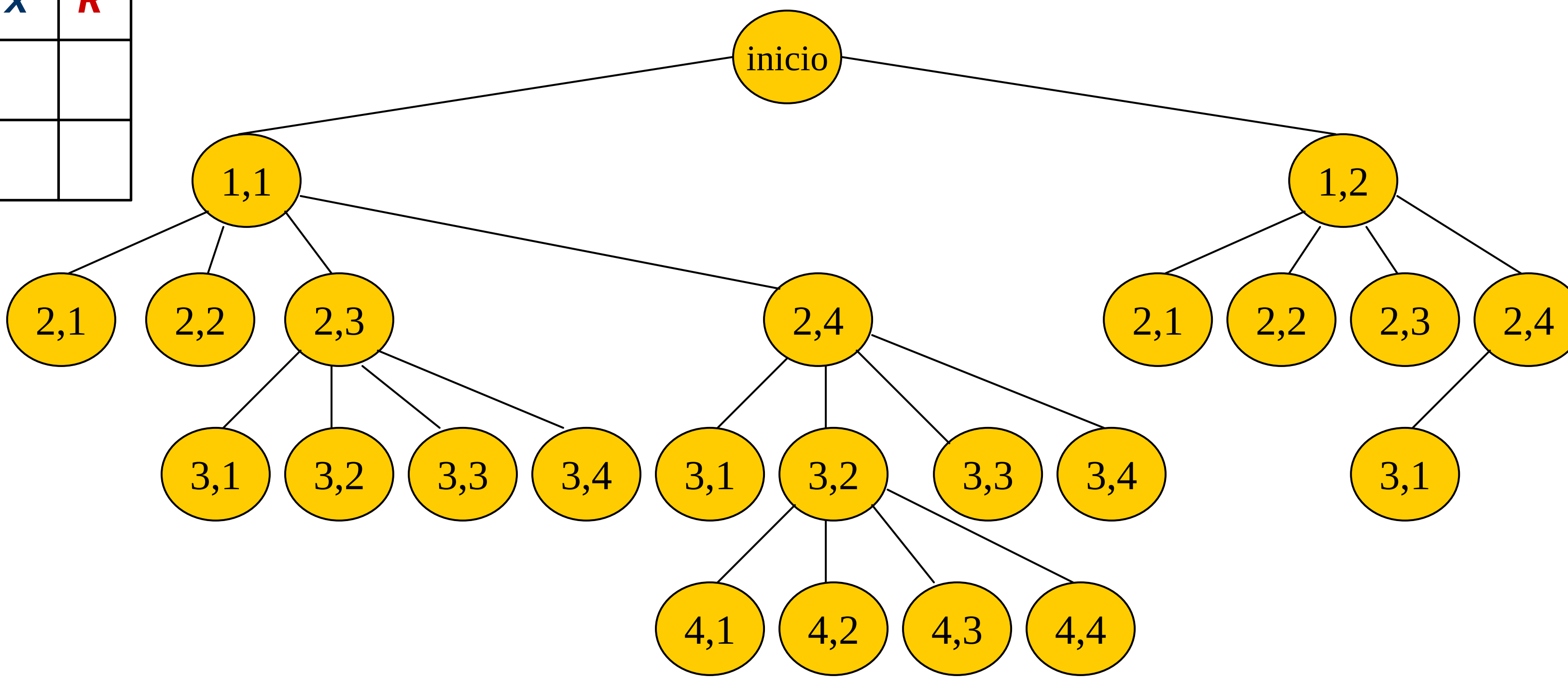
x	*R*		
x	*x*	*x*	*R*



OK... adelante con la búsqueda!

Ejemplo… para ***n*** = *4*

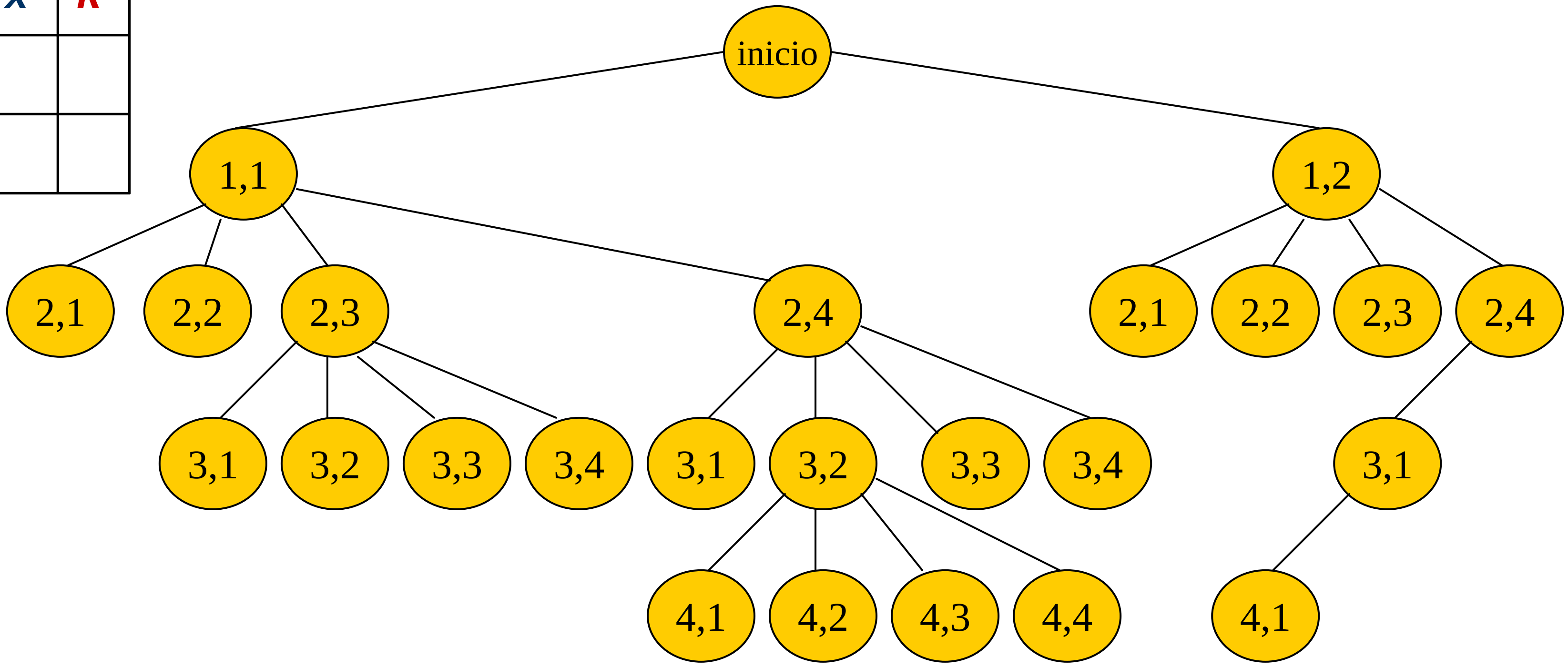
x	*R*		
x	*x*	*x*	*R*
R			



OK… adelante con la búsqueda!

Ejemplo... para *n* = *4*

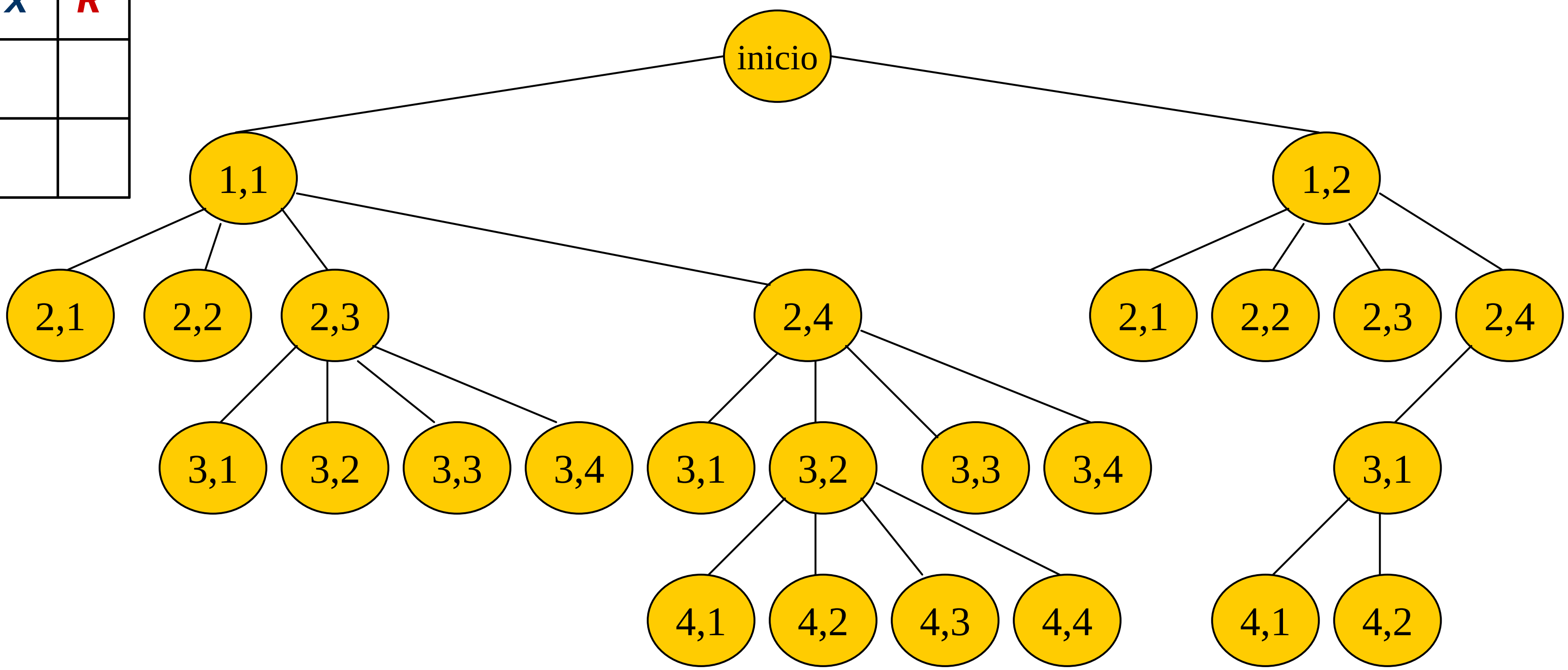
x	*R*		
x	*x*	*x*	*R*
R			
x			



NO cumple el criterio (misma columna)

Ejemplo... para ***n*** = *4*

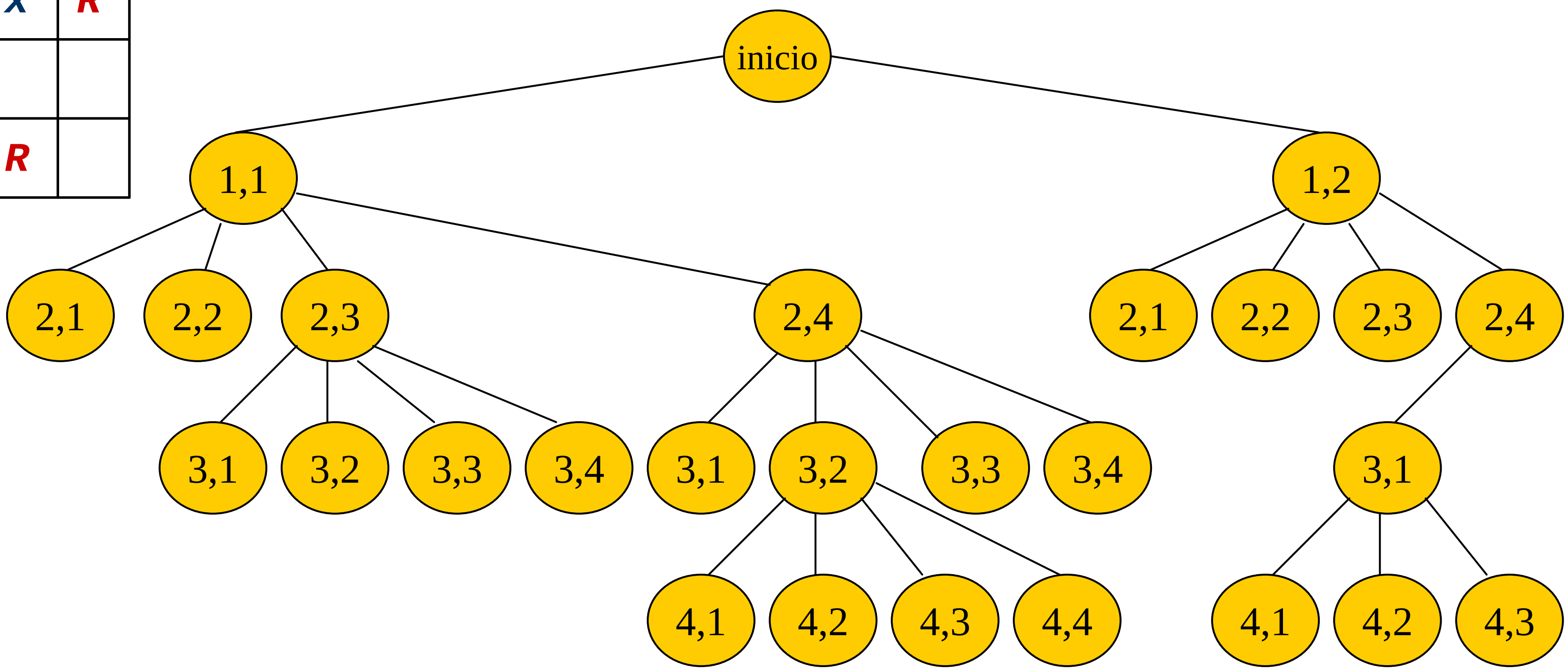
x	***R***		
x	*x*	*x*	***R***
R			
x	*x*		



NO cumple el criterio (misma columna)

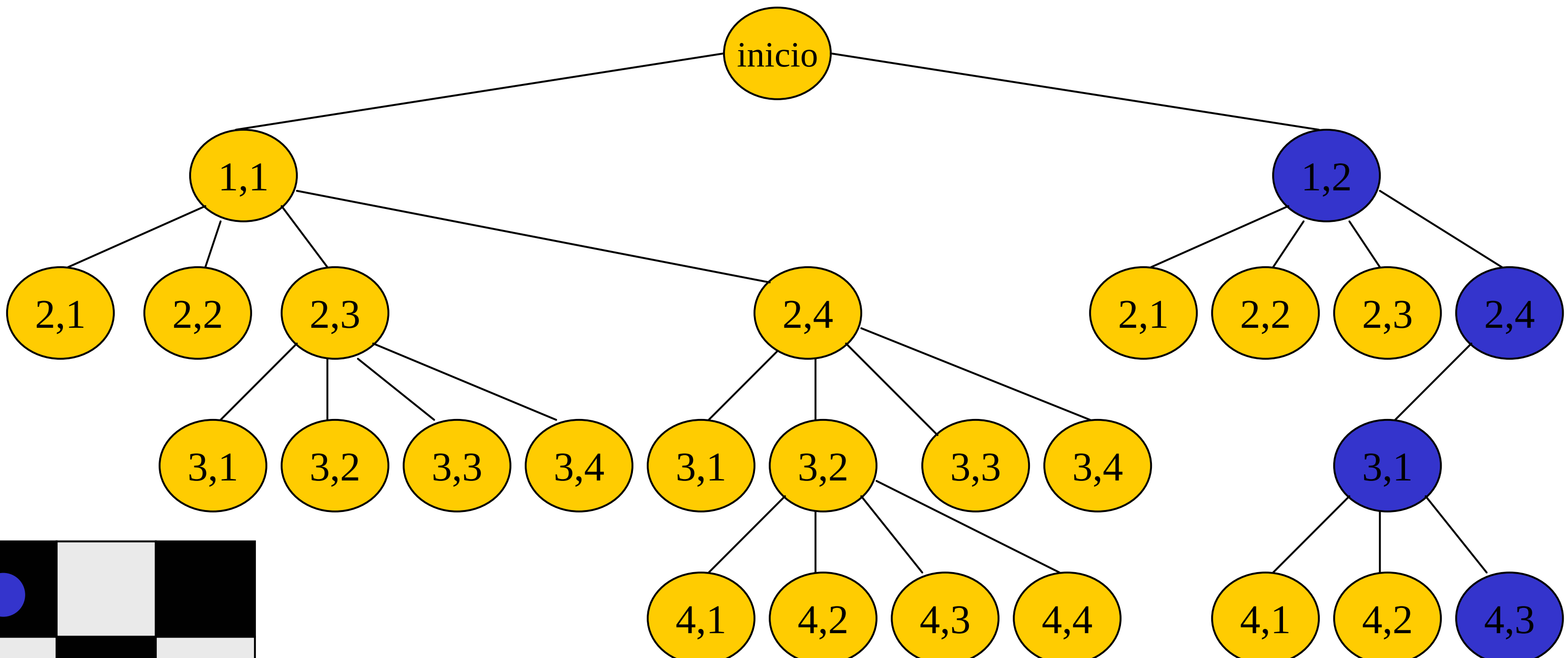
Ejemplo… para ***n*** = *4*

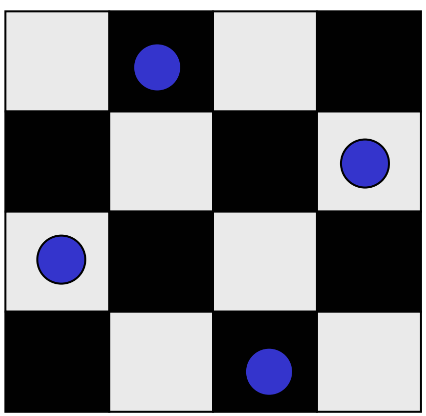
x	*R*		
x	*x*	*x*	*R*
R			
x	*x*	*R*	



¡¡OK… se encontró solución !!

Ejemplo… para ***n*** = *4*





Se comprobaron 27 nodos… Sin *backtracking* se hubieran comprobado 155 nodos

Índice

I. LA TÉCNICA BACKTRACKING

Algoritmo Backtracking

- Podemos presentar una formulación general, aunque precisa, del proceso de *backtracking*
- Supondremos que hay que encontrar todos los nodos respuesta, y no solo uno
- Sea $(x_1, x_2, \ldots, x_i)$ un camino desde la raíz hasta un nodo en el árbol de estados
- Sea $T(x_1, x_2, \ldots, x_i)$ el conjunto de todos los posibles valores x_{i+1} tales que $(x_1, x_2, \ldots, x_{i+1})$ es también un camino hacia un estado del problema
- Suponemos la existencia de funciones de acotación B_{i+1} (expresadas como predicados) tales que, $B_{i+1}(x_1, x_2, \ldots, x_{i+1})$ es falsa para un camino $(x_1, x_2, \ldots, x_{i+1})$ desde el nodo raíz hasta un estado del problema si el camino no puede extenderse para alcanzar un nodo respuesta
- Así los candidatos para la posición $i+1$ del vector solución $X(1..n)$ son aquellos valores que son generados por T y satisfacen B_{i+1}

Algoritmo Backtracking

- El Procedimiento *Backtrack*, representa el esquema general backtracking haciendo uso de *T* y B_{i+1}.

Procedimiento *BACKTRACK*(n)

{Todas las soluciones se generan en *X(1..n)* y se imprimen tan pronto como se encuentran. *T(X(1),...,X(k-1))* da todos los posibles valores de *X(k)* dado que habíamos escogido *X(1),...,X(k-1)*. El predicado $B_k(X(1),...,X(k))$ determina los elementos *X(k)* que satisfacen las restricciones implícitas}

```
Begin
  k = 1
    While k > 0 do
      If queda algún X(k) no probado tal que
              X(k) ∈T(X(1),..., X(k-1)) and Bk (X(1),..., X(k)) = true
      Then if (X(1),..., X(k)) es un camino hacia un nodo respuesta
              Then print (X(1),..., X(k))
              k = k+1
      else k = k-1
end
```

Algoritmo Backtracking recursivo

- El siguiente algoritmo, *Rbacktrack*, presenta una formulación recursiva del método, ya que como *backtracking* básicamente es un recorrido en postorden, es natural describirlo así,

Procedimiento RBACTRACK(K)

{Se supone que los primeros *k-1* valores *X(1),…,X(k-1)* del vector solución *X(1..n)* han sido asignados}

Begin

for cada *X(k)* tal que *X(k)* $\in$ *T(X(1),…,X(k-1))* and *B(X(1),…, X(k))* = *true* do

If (*X(1),…,X(k)*) es un camino hacia un nodo respuesta

Then print (*X(1),…, X(k)*)

RBACKTRACK(*K+1*)

End

Eficiencia de backtracking

- La eficiencia de los algoritmos backtracking depende básicamente de cuatro factores:
  - el tiempo necesario para generar el siguiente *X(k)*,
  - el número de *X(k)* que satisfagan las restricciones explícitas,
  - el tiempo para las funciones de acotación B_i y
  - el número de *X(k)* que satisfagan las B_i para todo *i*
- Las funciones de acotación se entienden buenas si reducen considerablemente el número de nodos que generan
- Las buenas funciones de acotación consumen mucho tiempo en evaluaciones, por lo que hay que buscar un equilibrio entre el tiempo global de computación, y la reducción del número de nodos generados.

Eficiencia de backtracking

- De los *4* factores que determinan el tiempo requerido por un algoritmo *backtracking*, sólo la cuarta, el número de nodos generados, varía de un caso a otro
- Un algoritmo backtracking en un caso podría generar sólo $O(n)$ nodos, mientras que en otro (relativamente parecido) podría generar casi todos los nodos del árbol de espacio de estados
- Si el número de nodos en el espacio solución es 2^n o $n!$, el tiempo del peor caso para el algoritmo *backtracking* sería generalmente $O(p(n)2^n)$ u $O(q(n)n!)$ respectivamente, con p y q polinomios en n
- La importancia del *backtracking* reside en su capacidad para resolver casos con grandes valores de n en muy poco tiempo
- La dificultad esta en predecir la conducta del algoritmo *backtracking* en el caso que deseemos resolver

Eficiencia de backtracking

- Podemos estimar el número de nodos que se generaran con un algoritmo *backtracking* usando el método de *Monte Carlo*
- Se trata de generar un camino aleatorio en el árbol de estados
- Sea X un nodo en ese camino aleatorio. Supongamos que X esta en el nivel i del árbol del espacio de estados
- Las funciones de acotación se usan en el nodo X para determinar el numero m_i de sus hijos que no hay que acotar. El siguiente nodo en el camino se obtiene seleccionando aleatoriamente uno de estos m_i hijos que no se han acotado
- La generación del camino termina en un nodo que sea una hoja o cuyos hijos vayan a acotarse. Usando estos m_i podemos estimar el numero total, m, de nodos en el árbol del espacio de estados que no se acotarán

Índice

II. SOLUCIONES BACKTRACKING EN DISTINTOS PROBLEMAS

1. **El Problema de las 8 Reinas**
2. **La Suma de Subconjuntos**
3. **El Problema del Viajante de Comercio**
4. **El Problema del Coloreo de un Grafo**
5. **Laberintos y Backtracking**

Solución para las 8 reinas

- Generalizamos el problema para considerar un tablero *nxn* y encontrar todas las formas de colocar n reinas que no se ataquen
- Podemos tomar $(x_1,...,x_n)$ representando una solución si x_i es la columna de la *i*-ésima fila en la que la reina *i* esta colocada
- Los x_i*'s* serán todos distintos ya que no puede haber dos reinas en la misma columna
- ¿Como comprobar que dos reinas no estén en la misma diagonal?

			(1,4)				
(2,1)		(2,3)					
	(3,2)						
(4,1)		(4,3)					
			(5,4)				(5,8)
(6,1)				(6,5)		(6,7)	
	(7,2)				(7,6)		
		(8,3)		(8,5)		(8,7)	

Solución para las 8 reinas

- Si las casillas del tablero se numeran como una matriz *A(1..n,1..n)*, cada elemento en la misma diagonal que vaya de la parte superior izquierda a la inferior derecha, tiene el mismo valor *"fila-columna"*
- También, cualquier elemento en la misma diagonal que vaya de la parte superior derecha a la inferior izquierda, tiene el mismo valor *"fila+columna"*
- Si dos reinas están colocadas en las posiciones *(i,j)* y *(k,l)*, estarán en la misma diagonal sólo si,

$$i - j = k - l \quad \text{ó} \quad i + j = k + l$$

- La primera ecuación implica que

$$j - l = i - k$$

- La segunda que

$$j - l = k - i$$

- Así, dos reinas están en la misma diagonal si y solo si

$$|j-l| = |i-k|$$

Solución para las 8 reinas

- El procedimiento COLOCA(k) devuelve verdad si la *k*-ésima reina puede colocarse en el valor actual de *X(k)*. Testea si *X(k)* es distinto de todos los valores previos *X(1),...,X(k-1)*, y si hay alguna otra reina en la misma diagonal. Su tiempo de ejecución de *O(k-1)*

Procedimiento COLOCA(K)

{*X* es un array cuyos *k* primeros valores han sido ya asignados. *ABS(r)* da el valor absoluto de *r*}

```
Begin
   For i:=1 to k-1 do
       If X(i) = X(k) or ABS(X(i)-X(k)) = ABS(i-k)
       Then return (false)
   Return (true)
end
```

Solución para las 8 reinas

Procedimiento NREINAS(N)

{Usando *backtracking* este procedimiento imprime todos los posibles emplazamientos de n reinas en un tablero *nxn* sin que se ataquen}

```
Begin
X(1) := 0, k := 1                                  {k es la fila actual}
While k > 0 do                                     {hacer para todas las filas}
   X(k) := X(k) + 1                                {mover a la siguiente columna}
   While X(k) ≤ n and not COLOCA (k) do           {puede moverse esta reina?}
         X(k) := X(k) + 1
   If X(k) ≤ n                                     {Se encontró una posición?}
      Then if k = n                                {Es una solución completa?}
              Then print (X)
           Else k := k + 1; X(k) := 0              {Ir a la siguiente fila}
   Else k := k - 1                                 {Backtrack}
end
```

Solución para la suma de subconjuntos

- Tenemos *n* números positivos distintos (usualmente llamados pesos) y queremos encontrar todas las combinaciones de estos números que sumen *M*

- Los anteriores ejemplos *2* y *3* mostraron como podríamos formular este problema usando tamaños de las tuplas fijos o variables

- Consideraremos una solución backtracking usando la estrategia del tamaño fijo de las tuplas

- En este caso el elemento *X(i)* del vector solución es uno o cero, dependiendo de si el peso *W(i)* esta incluido o no.

Solución para la suma de subconjuntos

- Generación de los hijos de cualquier nodo en el árbol:

- Para un nodo en el nivel i, el hijo de la izquierda corresponde a $X(i) = 1$, y el de la derecha a $X(i) = 0$

- Una posible elección de funciones de acotación es $B_k(X(1), \ldots, X(k)) = true$ si y solo si,

$$\sum_{1..k} W(i)X(i) + \sum_{k+1..n} W(i) \geq M$$

- Claramente $X(1), \ldots, X(k)$ no pueden conducir a un nodo respuesta si no se verifica esta condición

Solución para la suma de subconjuntos

- Las funciones de acotación pueden fortalecerse si suponemos los *W(i)'s* en orden creciente

- En este caso, $X(1),..,X(k)$ no pueden llevar a un nodo respuesta si

$$\sum_{1..k} W(i)X(i) + W(k+1) > M$$

- Por tanto las funciones de acotación que usaremos serán las definidas de la siguiente forma: $B_k(X(1), \ldots, X(k))$ es *true* si y sólo si

$$\sum_{1..k} W(i)X(i) + \sum_{k+1..n} W(i) \geq M$$

y

$$\sum_{1..k} W(i)X(i) + W(k+1) \leq M$$

Solución para la suma de subconjuntos

- Ya que nuestro algoritmo no hará uso de B_n, no necesitamos preocuparnos por la posible aparición de $W(n+1)$ en esta función

- Aunque hasta aquí hemos especificado todo lo que es necesario para usar cualquiera de los esquemas Backtracking, resultaría un algoritmo mas simple si diseñamos a la medida del problema que estemos tratando cualquiera de esos esquemas

- Esta simplificación resulta de la comprobación de que si $X(k) = 1$, entonces

$$\sum_{1..k} W(i)X(i) + \sum_{k+1..n} W(i) > M$$

Esquema de algoritmo recursivo

Procedimiento SUMASUB (*s*,*k*,*r*)

{Los valores de $X(j)$, $1 \leq j < k$, ya han sido determinados. $s = \sum_{1..k-1} W(j)X(j)$ y $r = \sum_{k..n} W(j)$. Los $W(j)$ están en orden creciente. Se supone que $W(1) \leq M$ y que $\sum_{1..n} W(i) \geq M$}

Begin

{Generación del hijo izquierdo. Nótese que $s+W(k) \leq M$ ya que $B_{k-1} = true$}

$X(k) = 1$

{4} If $s + W(k) = M$

{5} Then For $i = 1$ to k print $X(i)$

Else

{7} If $s + W(k) + W(k+1) \leq M$

Then SUMASUB ($s + W(k), k+1, r-W(k)$)

{Generación del hijo derecho y evaluación de B_k}

If $s + r - W(k) \geq M$ and $s + W(k+1) \leq M$

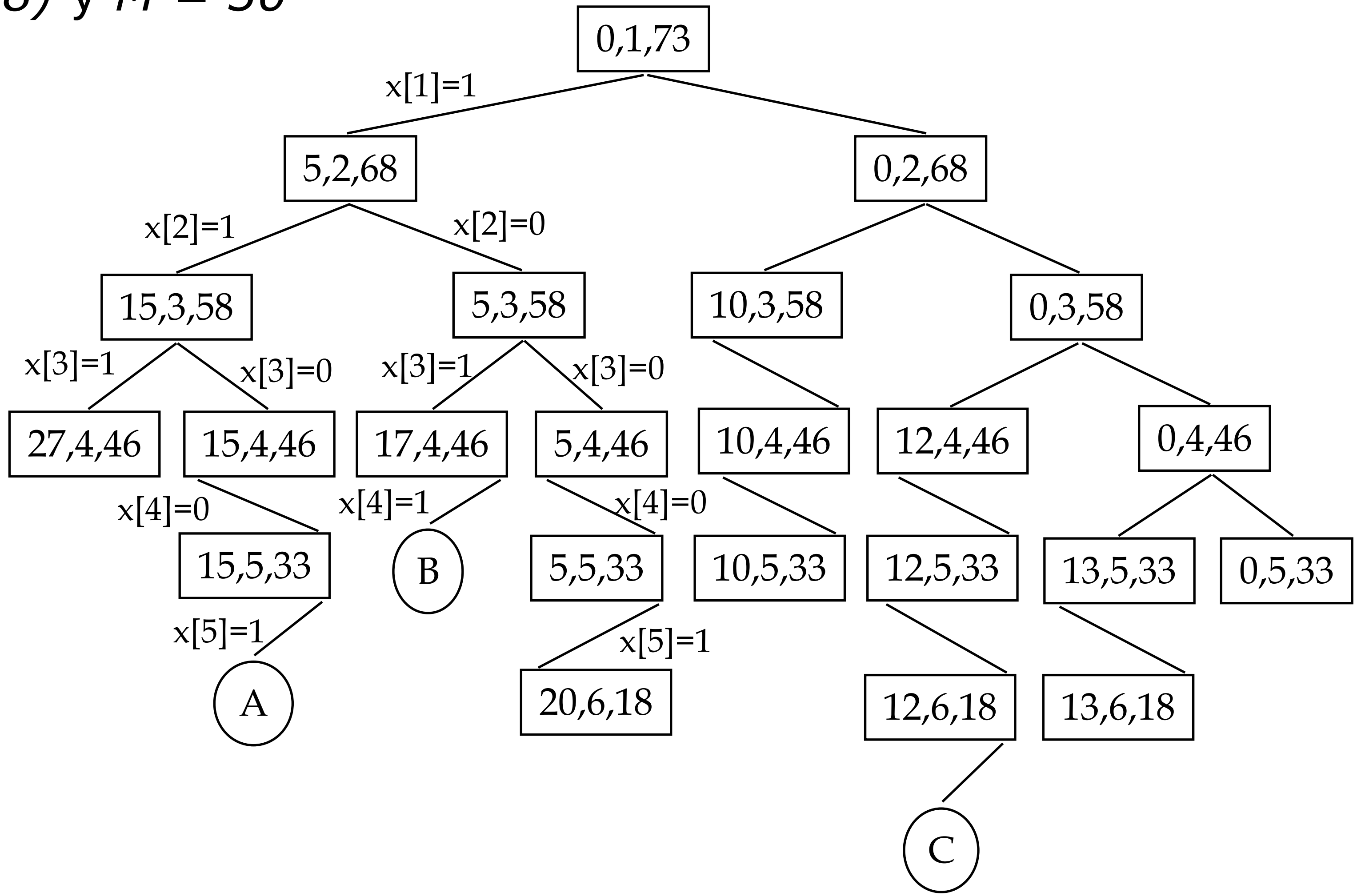
Then $X(k) = 0$

SUMASUB($s, k+1, r-w(k)$)

end

Ejemplo

Como trabaja SUMASUB para el caso en que: *W = (5, 10, 12, 13, 15, 18)* y *M = 30*



Problema de asignación de tareas

- Existen **n** personas y **n** trabajos.
- Cada persona **i** puede realizar un trabajo **j** con más o menos rendimiento: **B[i, j]**.
- **Objetivo:** asignar una tarea a cada trabajador (asignación uno-a-uno), de manera que se maximice la suma de rendimientos.

Tareas

Personas

B	1	2	3
1	4	9	1
2	7	2	3
3	6	3	5

Ejemplo 1. (P1, T1), (P2, T3), (P3, T2)

$B_{TOTAL} = 4+3+3 = 10$

Ejemplo 2. (P1, T2), (P2, T1), (P3, T3)

$B_{TOTAL} = 9+7+5 = 21$

Problema de asignación de tareas

- El problema de asignación es un problema **NP-completo** clásico.
- Otras variantes y enunciados:
  - Problema de los **matrimonios estables**.
  - Problemas con **distinto número** de tareas y personas. Ejemplo: problema de los árbitros.
  - Problemas de **asignación de recursos**: fuentes de oferta y de demanda. Cada fuente de oferta tiene una capacidad O[i] y cada fuente de demanda una D[j].
  - **Isomorfismo de grafos:** la matriz de pesos varía según la asignación realizada.

Problema de asignación de tareas

Enunciado del problema de asignación

- **Datos del problema:**
  – **n:** número de personas y de tareas disponibles.
  – **B: array** [1..n, 1..n] **de** entero. Rendimiento o beneficio de cada asignación. **B[i, j]** = beneficio de asignar a la persona **i** la tarea **j**.
- **Resultado:**
  – Realizar **n** asignaciones $\{(p_1, t_1), (p_2, t_2), ..., (p_n, t_n)\}$.
- **Formulación matemática:**

Maximizar $\sum_{i=1..n} B[p_i, t_i]$, sujeto a la restricción $p_i \neq p_j$, $t_i \neq t_j$, $\forall\ i \neq j$

proceso

Problema de asignación de tareas

1) Representación de la solución

- Mediante **pares de asignaciones**: $s = \{(p_1, t_1), (p_2, t_2), ..., (p_n, t_n)\}$, con $p_i \neq p_j$, $t_i \neq t_j$, $\forall\ i \neq j$
  - La tarea t_i es asignada a la persona p_i.
  - Árbol muy ancho. Hay que garantizar muchas restricciones. Representación no muy buena.
- Mediante **matriz de asignaciones**: $s = ((a_{11}, a_{12}, ..., a_{1n}), (a_{21}, a_{22}, ..., a_{2n}), ..., (a_{n1}, a_{n2}, ..., a_{nn}))$, con $a_{ij} \in \{0, 1\}$, y con $\sum_{i=1..n} a_{ij} = 1$, $\sum_{j=1..n} a_{ij} = 1$.
  - $a_{ij} = 1 \rightarrow$ la tarea **j** se asigna a la persona **i**
  - $a_{ij} = 0 \rightarrow$ la tarea **j** no se asigna a la persona **i**

Tareas

Personas

a	1	2	3
1	0	1	0
2	1	0	0
3	0	0	1

Problema de asignación de tareas

1) Representación de la solución

- Mediante **matriz de asignaciones**.
  - Árbol binario, pero muy profundo: n^2 niveles en el árbol.
  - También tiene muchas restricciones.
- Desde el punto de vista de las **personas:** $s = (t_1, t_2, ..., t_n)$, siendo $t_i \in \{1, ..., n\}$, con $t_i \neq t_j$, $\forall\ i \neq j$
  - $t_i \rightarrow$ número de tarea asignada a la persona **i**.
  - Da lugar a un árbol permutacional. ¿Cuánto es el número de nodos?
- Desde el punto de vista de las **tareas:** $s = (p_1, p_2, ..., p_n)$, siendo $p_i \in \{1, ..., n\}$, con $p_i \neq p_j$, $\forall\ i \neq j$
  - $p_i \rightarrow$ número de persona asignada a la tarea **i**.
  - Representación análoga (dual) a la anterior.

Problema de asignación de tareas

2) Elegir el esquema de algoritmo: caso optimización.

Backtracking (var s: array [1..n] de entero)

nivel:= 1; s:= $s_{INICIAL}$
voa:= -∞; soa:= Ø
bact:= 0

bact: Beneficio actual

repetir
  Generar (nivel, s)
  si Solución (nivel, s) AND (bact > voa) **entonces**
    voa:= bact; soa:= s
  si Criterio (nivel, s) **entonces**
    nivel:= nivel + 1
  sino
    mientras NOT MasHermanos (nivel, s) AND (nivel>0)
      hacer Retroceder (nivel, s)
  finsi
hasta nivel == 0

Problema de asignación de tareas

3) Funciones genéricas del esquema.

- **Variables:**
  - **s: array** [1..n] **de** entero: cada **s[i]** indica la tarea asignada a la persona **i**. Inicializada a 0.
  - **bact:** beneficio de la solución actual

- **Generar (nivel, s)** → Probar primero 1, luego 2, ..., n

  s[nivel]:= s[nivel] + 1

  si s[nivel]==1 **entonces** bact:= bact + B[nivel, s[nivel]]

  sino bact:= bact + B[nivel, s[nivel]] – B[nivel, s[nivel]-1]

- **Criterio (nivel, s)**

  para i:= 1, ..., nivel-1 **hacer**

      si s[nivel] == s[i] **entonces devolver** false

  finpara

  devolver true

Problema de asignación de tareas

3) Funciones genéricas del esquema.

- **Solución (nivel, s)**

  devolver (nivel==n) AND Criterio (nivel, s)

- **MasHermanos (nivel, s)**

  devolver s[nivel] < n

- **Retroceder (nivel, s)**

  bact:= bact – B[nivel, s[nivel]]

  s[nivel]:= 0

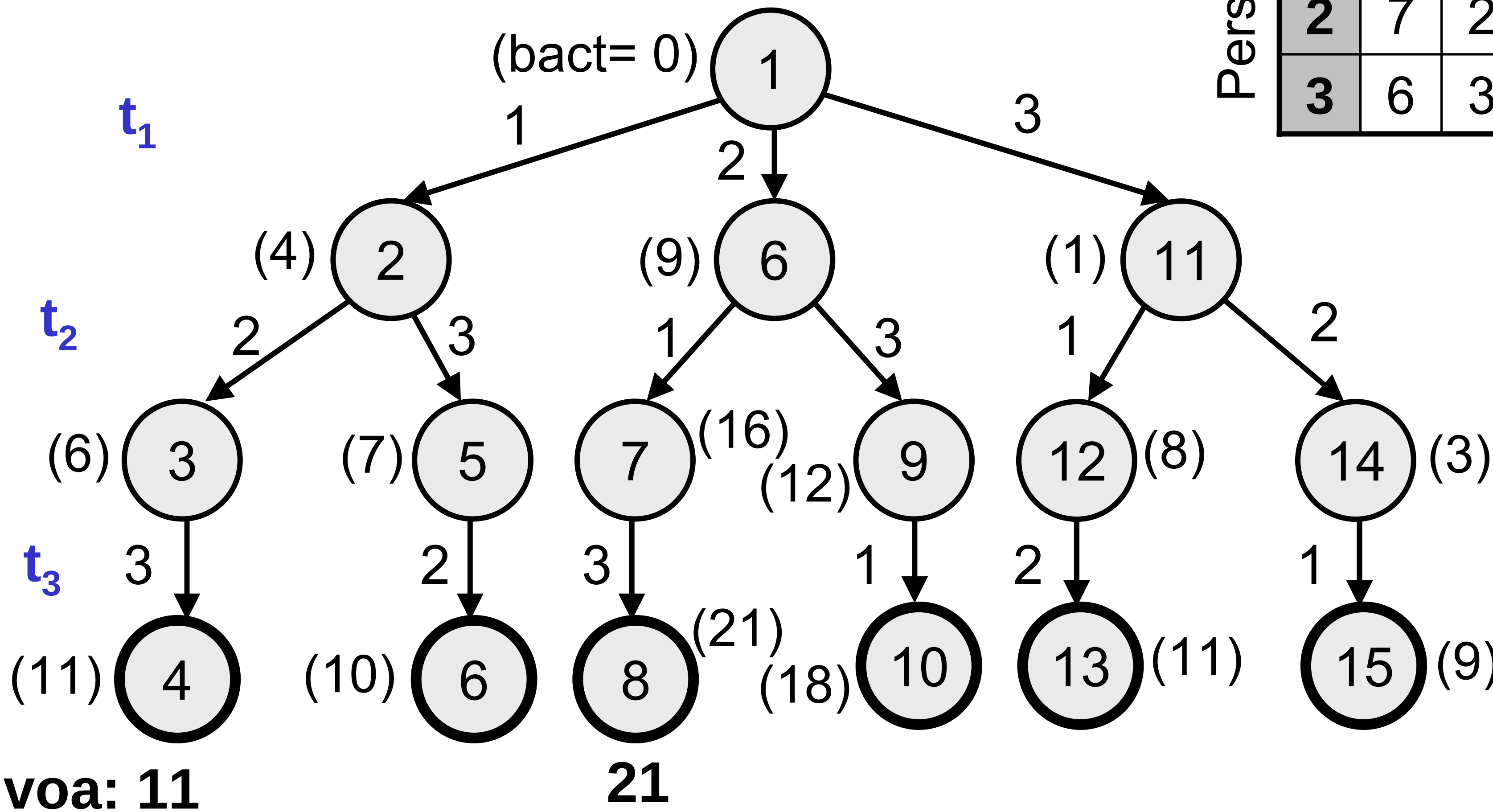
  nivel:= nivel – 1

Problema de asignación de tareas

- **Ejemplo de aplicación. n = 3**

Tareas

Personas B	1	2	3
1	4	9	1
2	7	2	3
3	6	3	5



Problema de asignación de tareas

- **Problema:** la función **Criterio** es muy lenta, repite muchas comprobaciones.
- **Solución:** usar un array que indique las tareas que están ya usadas en la asignación actual.
  - **usada: array** [0..n] **de** entero.
  - **usada[i]** indica el número de veces que es usada la tarea **i** en la planificación actual (es decir, en **s**).
  - **Inicialización: usada[i] = 0**, para todo **i**.
  - Modificar las funciones del esquema.

Problema de asignación de tareas

3) Funciones genéricas del esquema.

- **Criterio (nivel, s)**

  devolver usada[s[nivel]]==1

- **Retroceder (nivel, s)**

  bact:= bact – B[nivel, s[nivel]]

  usada[s[nivel]]--

  s[nivel]:= 0

  nivel:= nivel – 1

- **Generar (nivel, s)**

  usada[s[nivel]]--

  s[nivel]:= s[nivel] + 1

  usada[s[nivel]]++

  si s[nivel]==1 **entonces** bact:= bact + B[nivel, s[nivel]]

  sino bact:= bact + B[nivel, s[nivel]] – B[nivel, s[nivel]-1]

Problema de asignación de tareas

3) Funciones genéricas del esquema.

- Las funciones **Solución** y **MasHermanos** no se modifican.

- **Conclusiones:**
  - El algoritmo sigue siendo muy ineficiente.
  - Aunque garantiza la solución óptima...

El viajante de comercio

- Sea *G* = *(V, E)* un grafo conexo con *n* vértices. Un ciclo Hamiltoniano es un camino circular a lo largo de los *n* vértices de *G* que visita cada vértice de *G* una vez y vuelve al vértice de partida, que naturalmente es visitado dos veces

- Estamos interesados en construir un algoritmo backtracking que determine todos los ciclos Hamiltonianos de *G*, que puede ser dirigido o no (para quedarnos con el de mínimo costo)

- El vector backtracking solución $(x_1, \ldots, x_n)$ se define de modo que x_i represente el *i*-ésimo vértice visitado en el ciclo propuesto

El viajante de comercio

- Todo lo que se necesita es determinar como calcular el conjunto de posibles vértices para x_k si ya hemos elegido $x_1, \ldots, x_{k-1}$
- Si $k = 1$, entonces $X(1)$ puede ser cualquiera de los n vértices
- Para evitar imprimir el mismo ciclo n veces, exigimos que $X(1) = 1$
- Si $1 < k < n$, entonces $X(k)$ puede ser cualquier vértice v que sea distinto de $X(1), X(2),\ldots,X(k-1)$ que esté conectado por una arista a $X(k-1)$
- $X(n)$ sólo puede ser el único vértice restante y debe estar conectado a $X(n-1)$ y a $X(1)$

El viajante de comercio

- El procedimiento SiguienteValor(*k*) devuelve el siguiente vértice válido para la posición *k* o *0* si no queda ninguno

Algoritmo SiguienteValor (k)

{*x* es un array cuyos *k-1* primeros valores han sido ya asignados, *k es la posición del siguiente valor a asignar y G la matriz de adyacencia que representa el grafo*}

```
while (true)
    value = (x[k]++) mod (N+1)
    if (value = 0) then return value
    if (G[x[k-1],value]) {chequeo de distinguibilidad}
        for j = 1 to k-1  if x[j] = value then break
        if (j=k) and (k<N or (k=N and G[x[N],x[1]]))
            then return value
endWhile
```

El viajante de comercio

Procedimiento Hamiltoniano (k)

{*x* es un array cuyos *k-1* primeros valores han sido ya asignados, *k es la posición del siguiente valor a colocar (nodo en expansión o e-nodo)*}

```
while
    x[k] = SiguienteValor(k)
    if (x[k] = 0) then return
    if (k = N) then print solución
    else Hamiltoniano (k+1)
endWhile
```

Utilizando este algoritmo podemos particularizar el esquema backtracking recursivo para encontrar todos los ciclos hamiltonianos

El problema del coloreo de un grafo

- Sea G un grafo y m un número entero positivo. Queremos saber si los nodos de G pueden colorearse de tal forma que no haya dos vértices adyacentes que tengan el mismo color, y que sólo se usen m colores para esa tarea

- Este es el problema de la m-colorabilidad

- El problema de optimización de la m-colorabilidad, pregunta por el menor numero m con el que el grafo G puede colorearse. A ese entero se le denomina Número Cromático del grafo

El problema del coloreo de un grafo

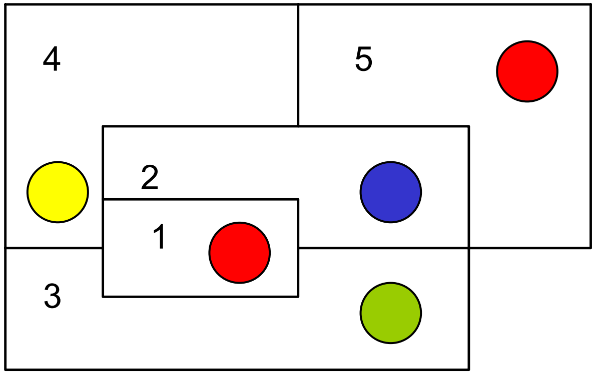
- Un grafo se llama plano si y sólo si puede pintarse en un plano de modo que ningún par de aristas se corten entre si

- Un caso especial famoso del problema de la *m*-colorabilidad es el problema de los cuatro colores para grafos planos que, dado un mapa cualquiera, consiste en saber si ese mapa podrá pintarse de manera que no haya dos zonas colindantes con el mismo color, y además pueda hacerse ese coloreo sólo con cuatro colores

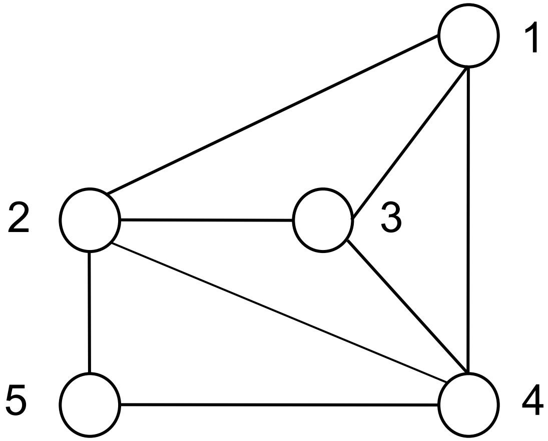
- Este problema es fácilmente traducible a la nomenclatura de grafos

El problema del coloreo de un grafo

- El mapa



puede traducirse en el siguiente grafo



El problema del coloreo de un grafo

- Representamos el grafo por su matriz de adyacencia GRAFO*(1:n, 1:n)* siendo GRAFO*(i,j)* = *true* si *(i,j)* es una arista de *G*. En otro caso GRAFO*(i,j)* = *false*

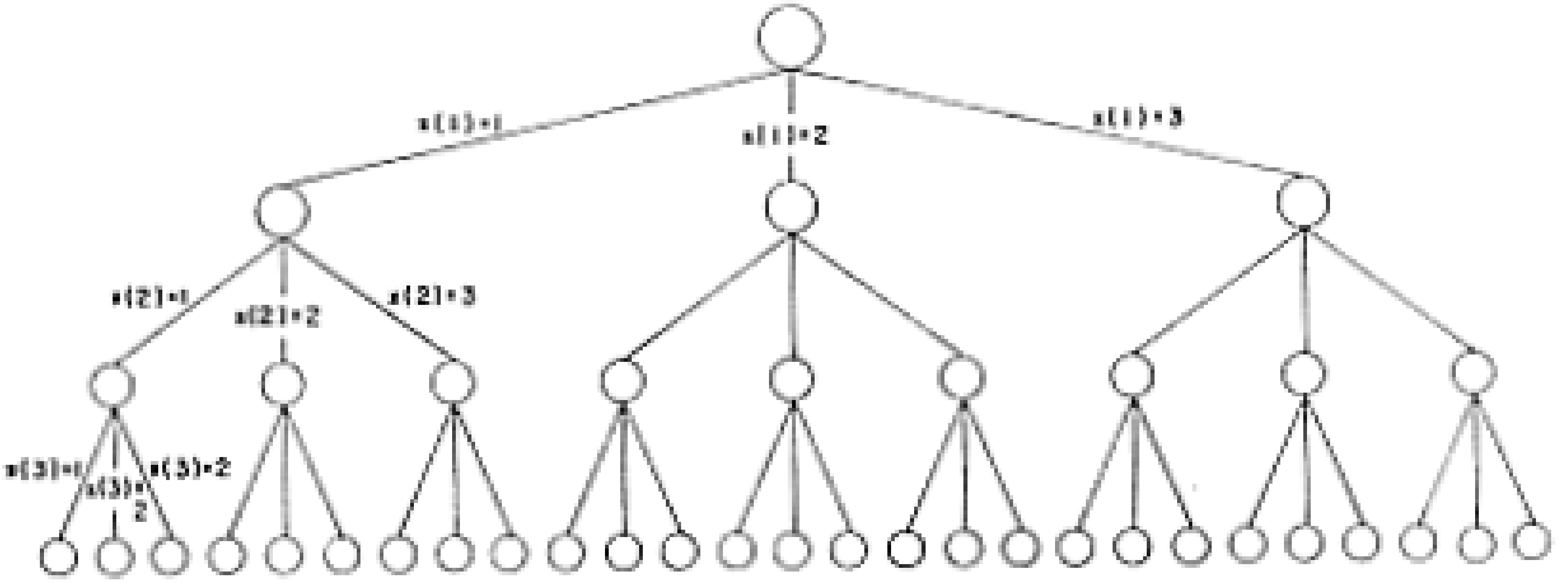
- Los colores se representan por los enteros *1,2,...,m*

- Las soluciones vendrán dadas por *n*-tuplas *(X(1),...,X(n)*, donde *X(i)* será el color del vértice *i*

- Usando la formulación recursiva del procedimiento backtracking, puede construirse un algoritmo que trabaja en un tiempo $O(nm^n)$

El problema del coloreo de un grafo

- El espacio de estados subyacente es un árbol de grado *m* y altura *n+1*, en el que cada nodo en el nivel *i* tiene *m* hijos correspondientes a las *m* posibles asignaciones para *X(i)*, $1 \leq i \leq n$, y donde los nodos en el nivel *n+1* son nodos hoja



El problema del coloreo de un grafo

Algoritmo *M*-Color (*k*)

```
while (true)
      SiguienteValor(k)
      if (color[k] = 0) then break                 (1)
      if (k = n)
            then print este coloreo                (2)
      else M-Color (k + 1)                         (3)
endWhile
```

(1) {no hay mas colores para k}
(2) {se encontró un coloreo valido para todos los nodos}
(3) {intenta colorear el siguiente nodo}

El problema del coloreo de un grafo

Algoritmo SiguienteValor (*k*)

{Devuelve los posibles colores de *X(k)* dado que *X(1)* hasta *X(k-1)* ya han sido coloreados}

```
while (true)
    color[k] = (color[k] + 1) mod (m + 1)    {ó n para asegurar}
    if (color[k] = 0) then return                    (1)
    for i = 1 to k-1
        if (conec[i,k] and color[i] = color[k])
            then break
    endfor
    if (i = k) return                                (2)
endWhile
```

(1) no hay mas colores para probar

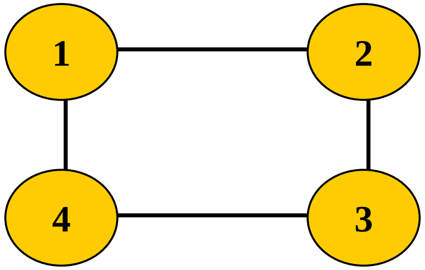
(2) Se ha encontrado un nuevo color (ningún nodo colisiona)

Eficiencia del algoritmo

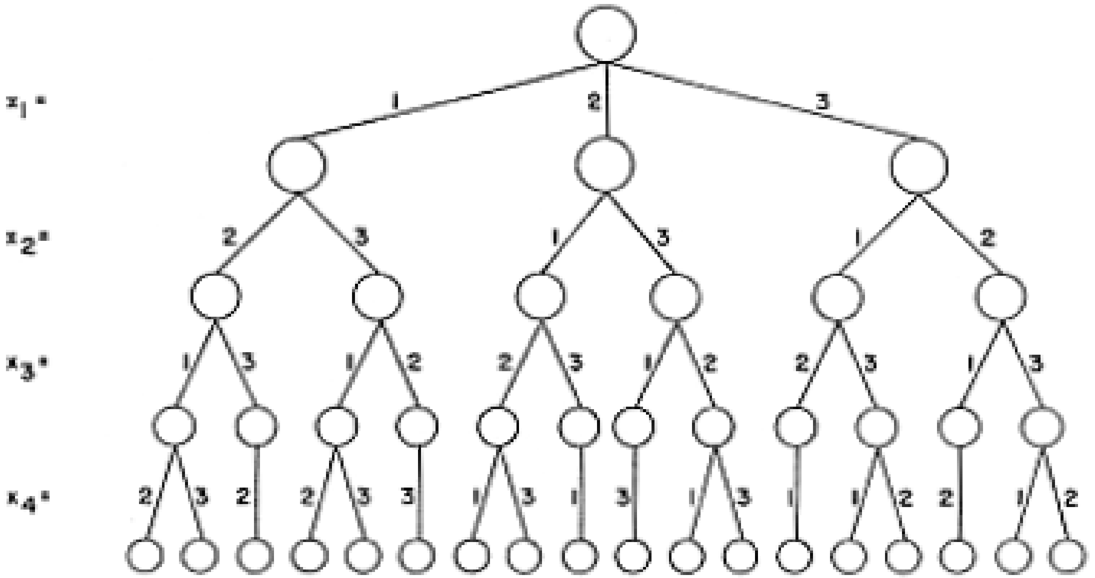
- El número de nodos internos en el espacio de estados es $\sum_{i=0..n-1} m^i$
- En cada nodo interno SiguienteValor invierte $O(nm)$ en determinar el hijo corespondiente a un coloreo legal
- El tiempo total esta acotado por $\sum_{i=1..n} m^i n = n(m^{n+1}-1)/(m-1) = O(n\ m^n)$

Ejemplo de coloreo

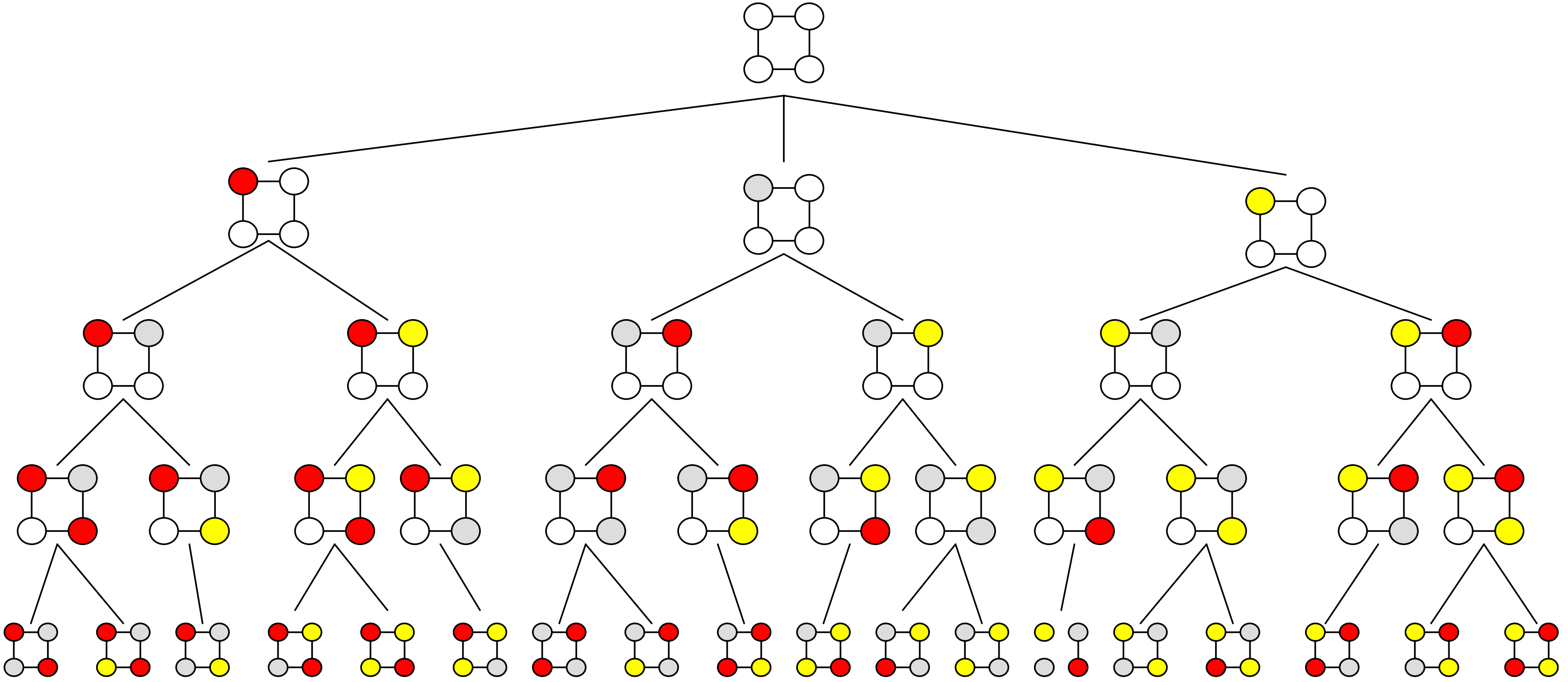
Si consideramos el siguiente grafo



El árbol que genera *M*-Color es



Otra representación del ejemplo



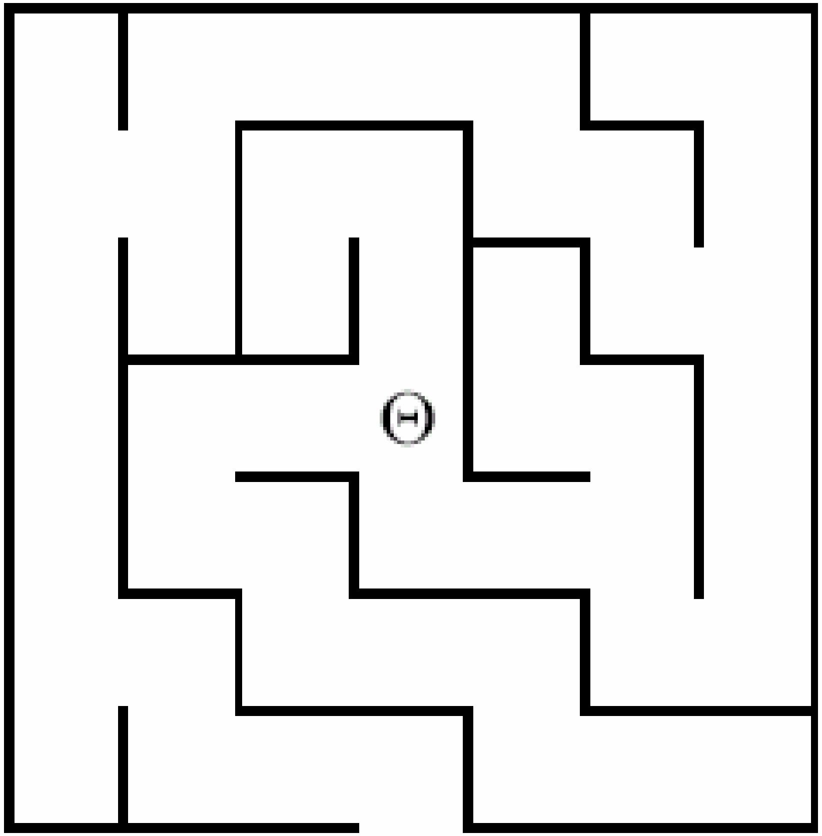
Laberintos y Backtracking

Este mosaico representa un laberinto, y esta en la Catedral de Chartres. Antes de estar allí, ya se conocía en Creta mil años antes.

También es conocido en otras culturas.

Un laberinto puede modelarse como una serie de nodos. En cada nodo hay que tomar una decisión que nos conduce a otros nodos.

Un laberinto sencillo



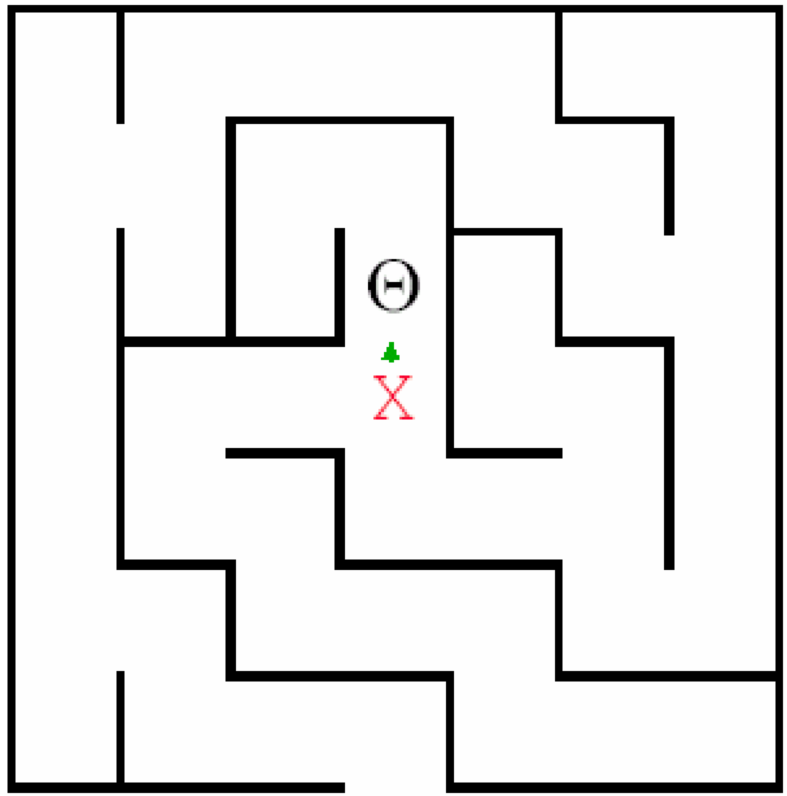
Buscar en el laberinto hasta encontrar una salida. Si no se encuentra una salida, informar de ello

Algoritmo Backtracking Modificado

Si la posición actual está fuera, devolver TRUE para indicar que hemos encontrado una solución.
Si la posición actual está marcada, devolver FALSE para indicar que este camino ya ha sido explorado.
Marcar la posición actual.

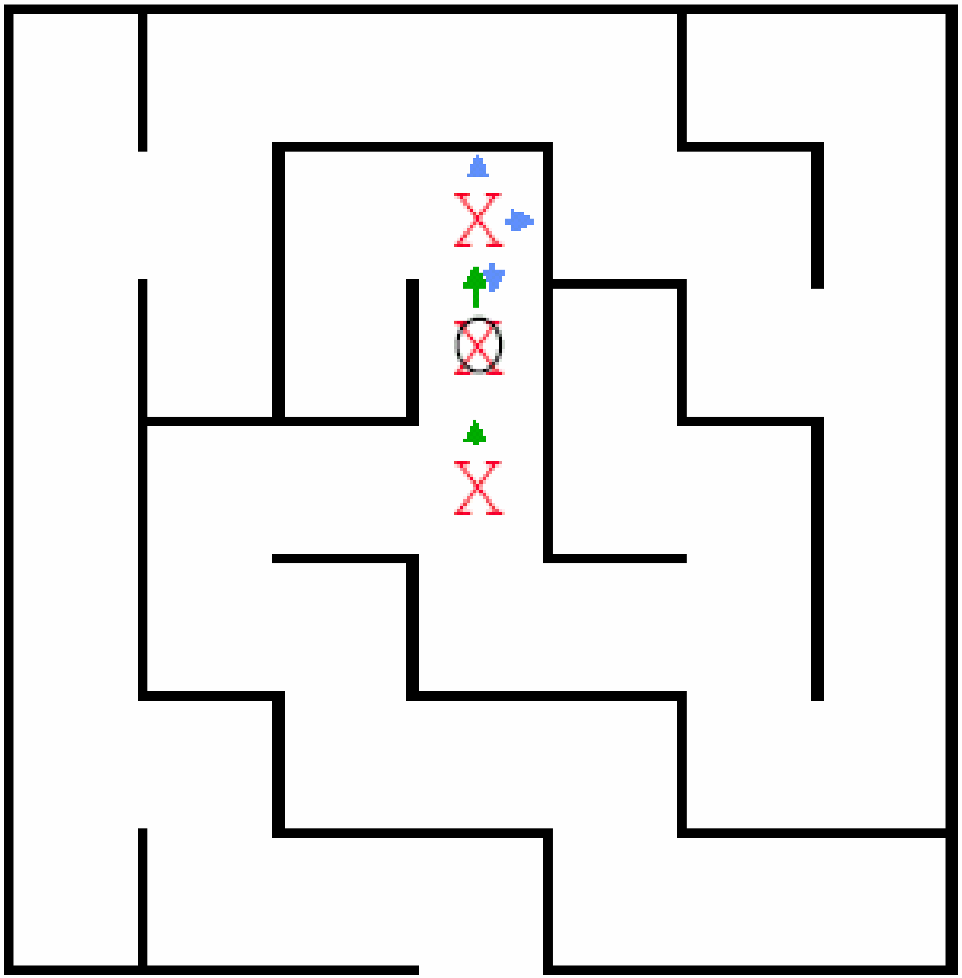
For (cada una de las 4 direcciones posibles **)**
{ **Si (** Esta dirección no está bloqueada por un muro **)**
{ Moverse un paso en la dirección indicada desde la posición actual.
Intentar resolver el laberinto desde ahí haciendo una llamada recursiva.
Si esta llamada prueba que el laberinto es resoluble, devolver TRUE para indicar este hecho.
}
}
Quitar la marca a la posición actual.
Devolver FALSE para indicar que ninguna de las 4 direcciones lleva a una solución

Backtracking en Acción



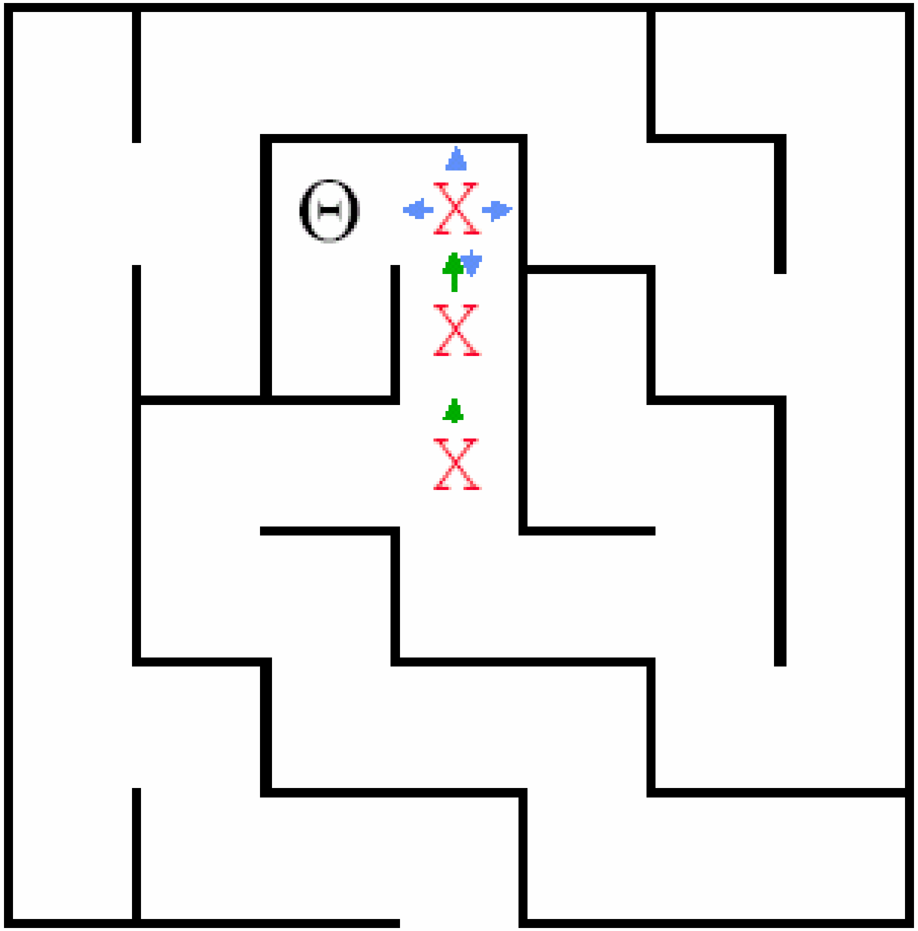
La parte crucial del algoritmo es el lazo FOR que nos lleva hacia las posibles alternativas que hay en un punto concreto. Aquí nos movemos hacia el norte.

Backtracking en Acción



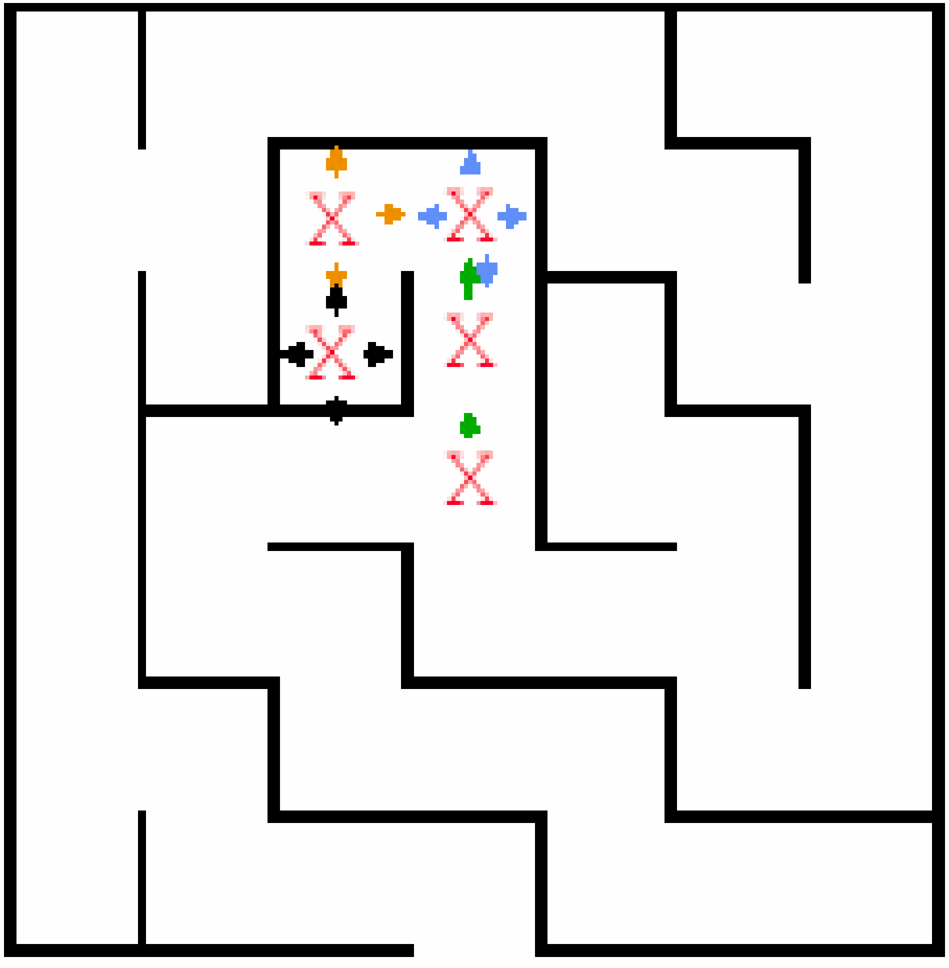
Aquí nos movemos hacia el Norte de nuevo, pero ahora la dirección Norte está bloqueada por un muro. El Este tambien está bloqueado, por lo que intentamos el Sur. Esa acción descubre que ese punto está marcado, de modo que volvemos atrás.

Backtracking en Acción



Por tanto,
el siguiente
movimiento que
podemos hacer
es hacia el Oeste

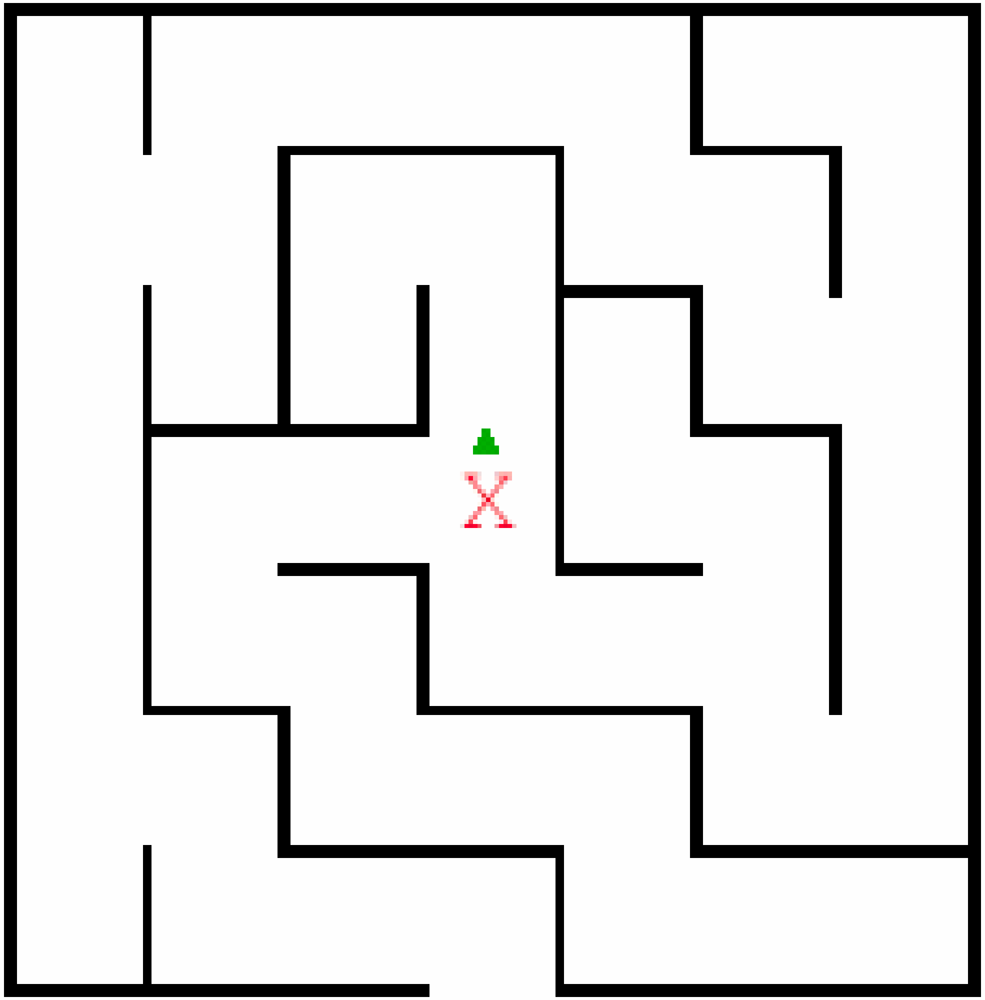
Backtracking en Acción



Este camino llega a un nodo (final) muerto

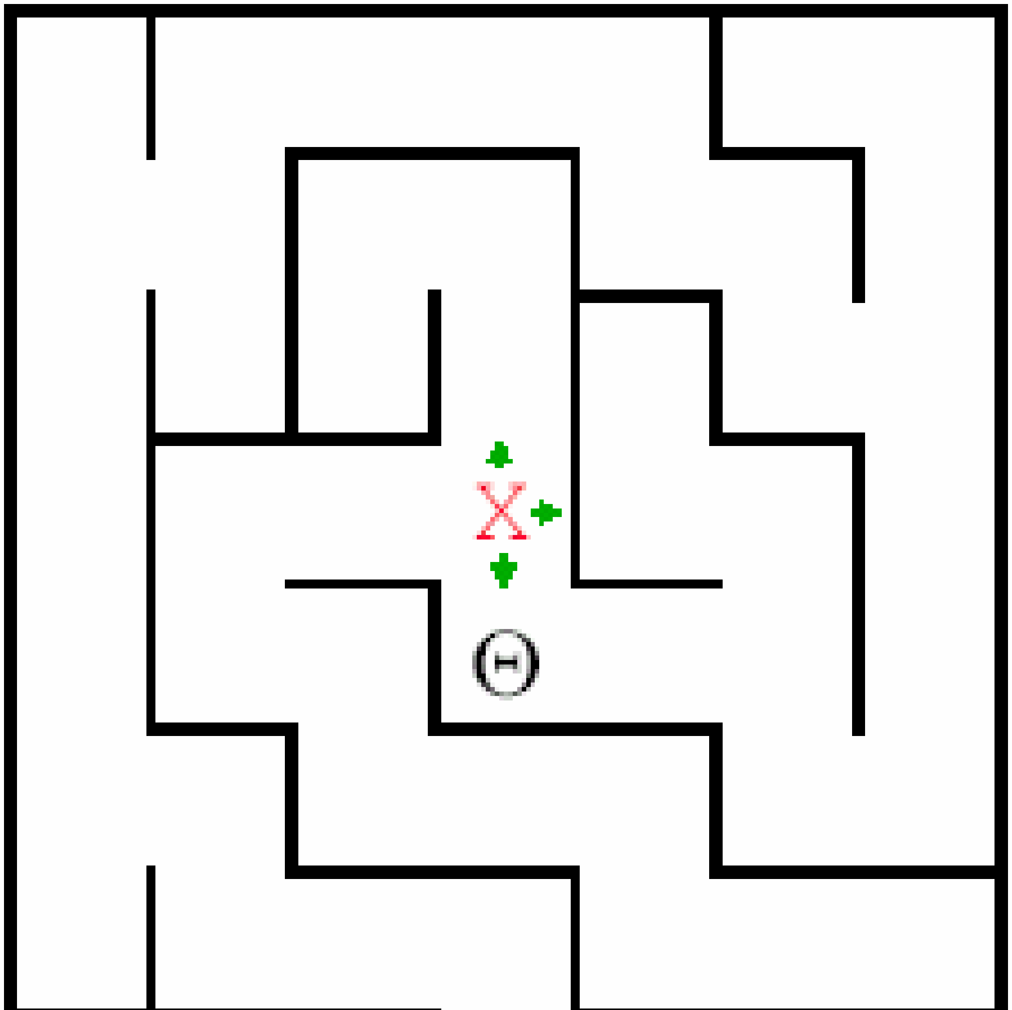
¡Por tanto, es el momento de hacer un backtrack!

Backtracking en Acción



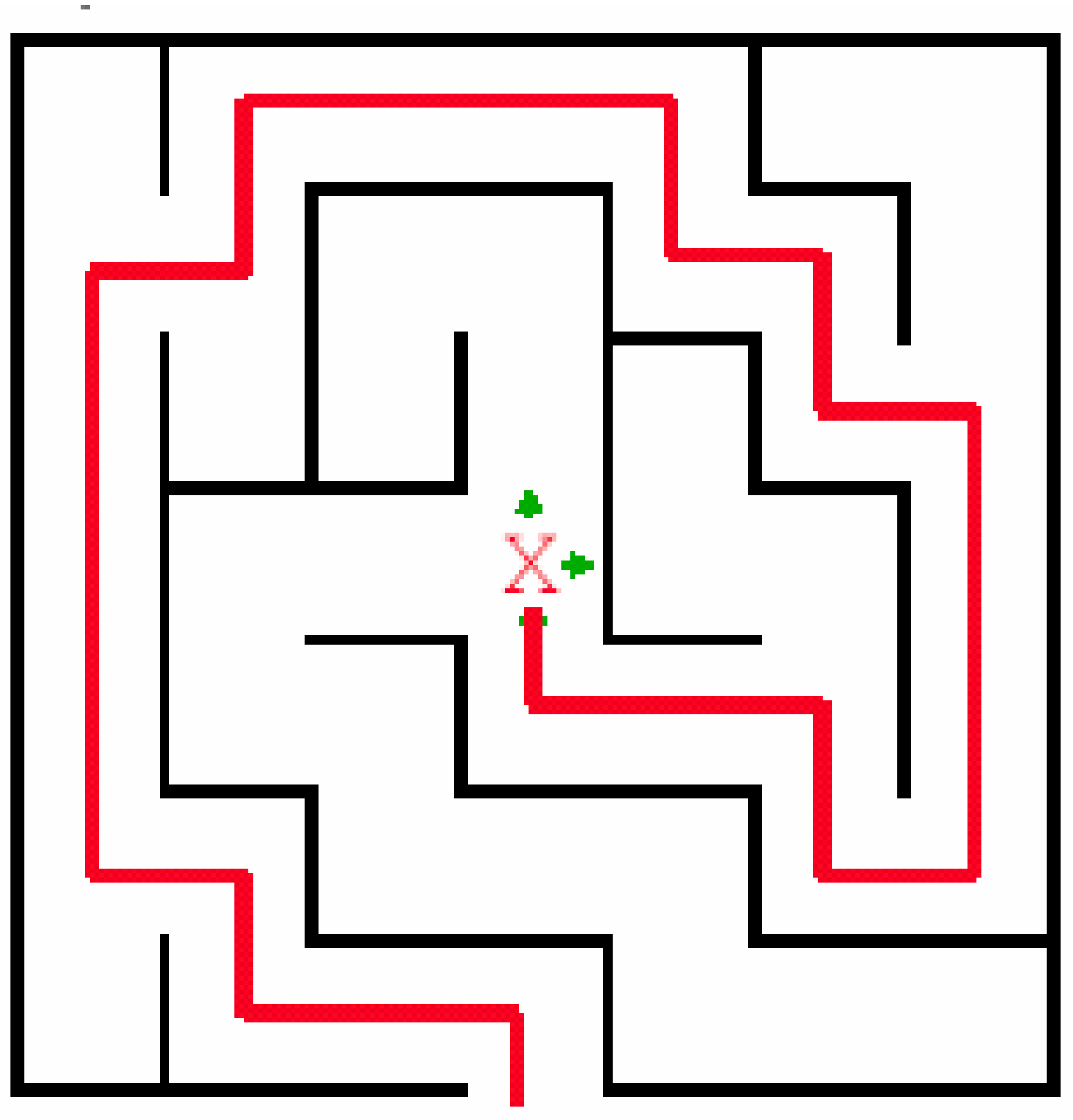
Se realizan sucesivas vueltas atrás hasta volvernos a encontrar aquí

Backtracking en Acción



Intentamos ahora el Sur

Primer camino que se encuentra



Backtracking: Conclusiones

- **Backtracking:** Recorrido exhaustivo y sistemático en un árbol de soluciones.
- **Pasos para aplicarlo:**
  - Decidir la forma del árbol.
  - Establecer el esquema del algoritmo.
  - Diseñar las funciones genéricas del esquema.
- Relativamente fácil diseñar algoritmos que encuentren soluciones óptimas pero...
- Los algoritmos de backtracking son muy ineficientes.

- **Mejoras:** mejorar los mecanismos de poda, incluir otros tipos de recorridos (no solo en profundidad)
  → Técnica de **Ramificación y Poda (Branch-Bound)**

Índice

III. MÉTODOS BRANCH-BOUND

Branch and bound

- **Branch and Bound** es una técnica muy similar a la de Backtracking, y basa su diseño en el análisis del árbol de estados de un problema:
  - Realiza un recorrido sistemático de ese árbol
  - El recorrido no tiene que ser necesariamente en profundidad
- Generalmente se aplica para resolver problemas de Optimización y para jugar juegos

Branch and bound

- Los algoritmos generados por está técnica son normalmente de orden exponencial o peor en su peor caso, pero su aplicación en casos muy grandes, ha demostrado ser eficiente (incluso más que bactracking)
- Puede ser vista como una generalización (o mejora) de la técnica de Backtracking
- Tendremos una estrategia de ramificación
- Se tratará como un aspecto importante las técnicas de poda, para eliminar nodos que no lleven a soluciones optimas
- La poda se realiza estimando en cada nodo cotas del beneficio que podemos obtener a partir del mismo

Branch and bound

- Diferencia fundamental con Backtracking:
  - En Bactracking tan pronto como se genera un nuevo hijo del nodo en curso, este hijo pasa a ser el nodo en curso
  - En BB se generan todos los hijos del nodo en curso antes de que cualquier otro nodo vivo pase a ser el nuevo nodo en curso (esta técnica no utiliza la búsqueda en profundidad)
- En consecuencia:
  - En Backtracking los únicos nodos vivos son los que están en el camino de la raíz al nodo en curso
  - En BB pueden haber más nodos vivos. Se almacenan en una estructura de datos auxiliar: **lista de nodos vivos**
- Además
  - En Bactracking el test de comprobación nos decía si era fracaso o no, mientras que en BB la cota nos sirve para podar el árbol y para saber el orden de ramificación, comenzando por las más prometedoras

Idea intutiva del Branch and Bound

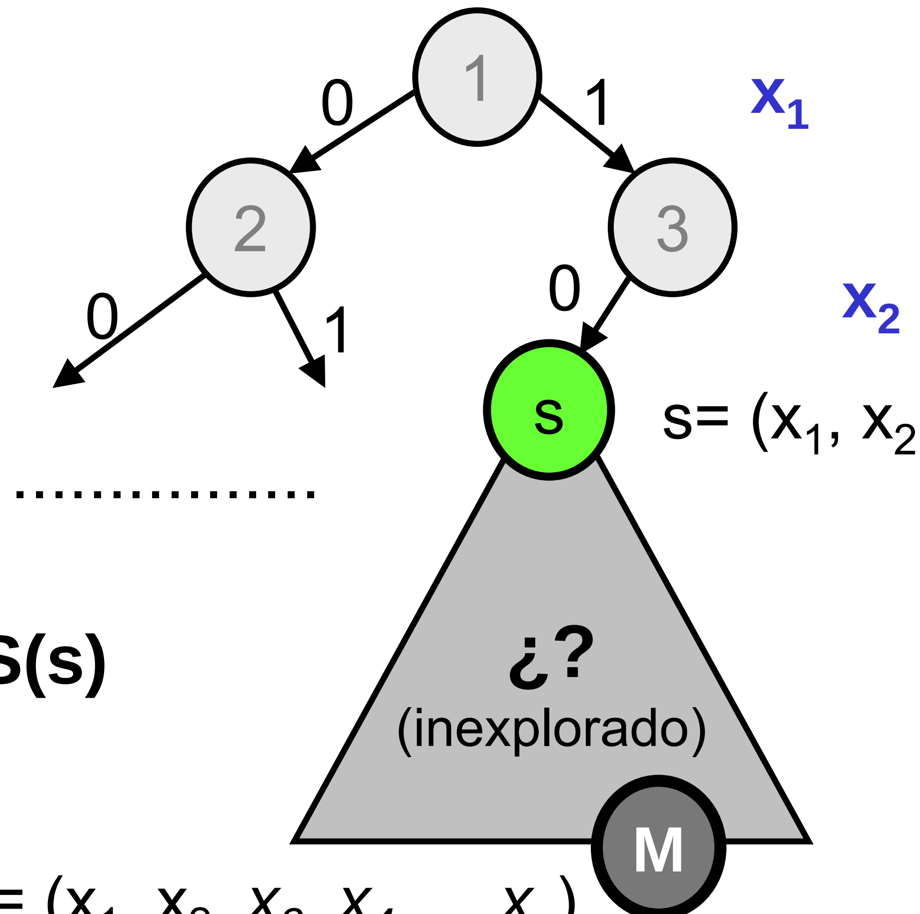
- Como decíamos anteriormente, la **ramificación y poda** (**branch and bound**) se suele utilizar en problemas de **optimización discreta** y en problemas de **juegos**.
- Puede ser vista como una **generalización** (o mejora) de la técnica de **backtracking**.
- **Similitud:**
  - Igual que backtracking, realiza un **recorrido sistemático** en un árbol de soluciones.
- **Diferencias:**
  - **Estrategia de ramificación:** el recorrido no tiene por qué ser necesariamente en profundidad.
  - **Estrategia de poda:** la poda se realiza **estimando** en cada nodo **cotas** del beneficio óptimo que podemos obtener a partir del mismo.

Idea intutiva del Branch and Bound

Estimación de cotas a partir de una solución parcial

- **Problema:** antes de explorar **s**, acotar el beneficio de la mejor solución alcanzable, **M**.

- **CI(s) ≤ Valor(M) ≤ CS(s)**



M= (x_1, x_2, x_3, x_4,..., x_n)
Valor(M) = ¿?

Idea intutiva del Branch and Bound

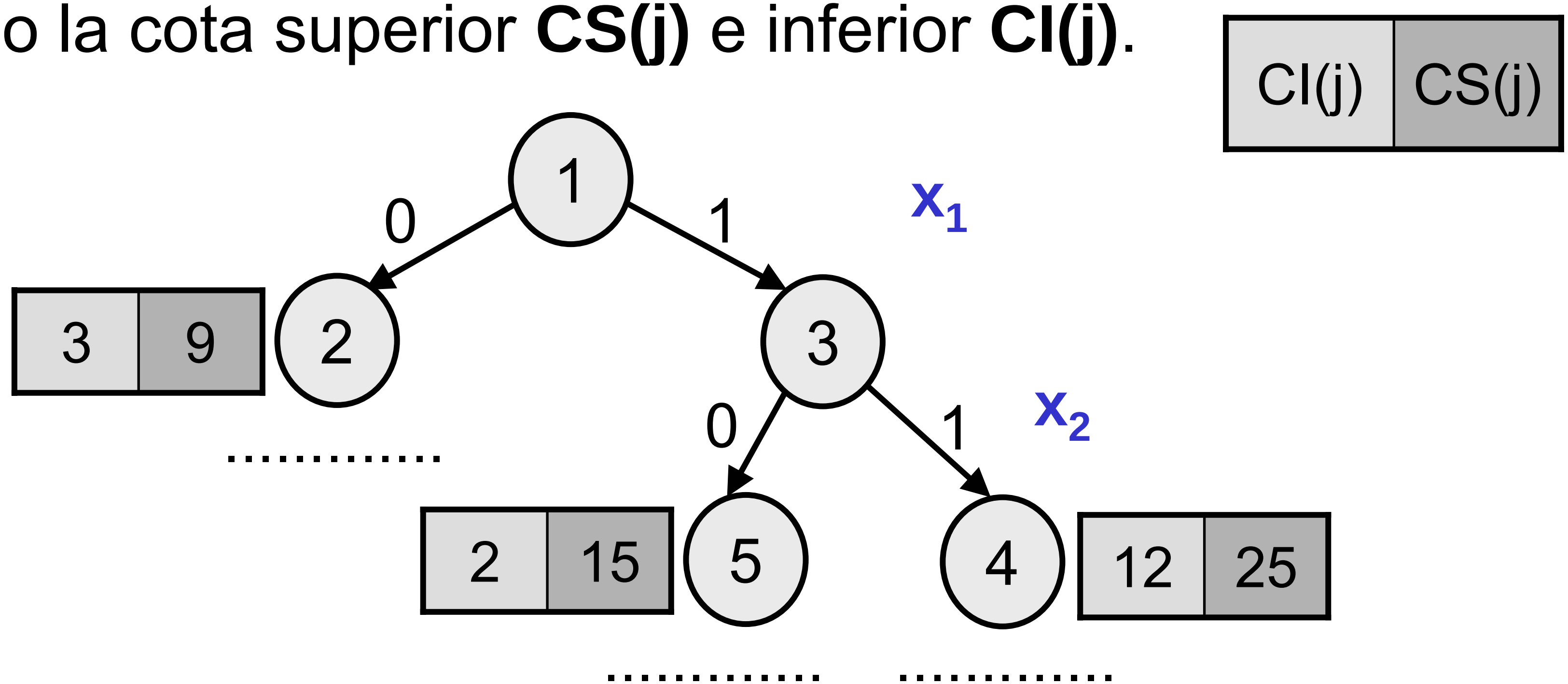
- Para cada nodo **i** tendremos:
  - **CS(i): Cota superior** del beneficio (o coste) óptimo que podemos alcanzar a partir del nodo **i**.
  - **CI(i): Cota inferior** del beneficio (o coste) óptimo que podemos alcanzar a partir del nodo **i**.
  - **BE(i): Beneficio estimado** (o coste) óptimo que se puede encontrar a partir del nodo **i**.

- Las cotas deben ser “fiables”: determinan cuándo se puede realizar una poda.
- El beneficio (o coste) estimado ayuda a decidir qué parte del árbol evaluar primero.

Idea intutiva del Branch and Bound

Estrategia de poda

- Supongamos un problema de **maximización**.
- Hemos recorrido varios nodos, estimando para cada uno la cota superior **CS(j)** e inferior **CI(j)**.



- ¿Merece la pena seguir explorando por el nodo 2?
- ¿Y por el 5?

Idea intutiva del Branch and Bound

- **Estrategia de poda (maximización).** Podar un nodo **i** si se cumple que:

  CS(i) $\leq$ **CI(j)**, para algún nodo **j** generado

  o bien

  CS(i) $\leq$ **Valor(s)**, para algún nodo **s** solución final

- **Implementación.** Usar una variable de poda **C**:
  C = max({**CI(j)** | $\forall$ **j** generado}, {**Valor(s)** | $\forall$ **s** solución final})
  - Podar **i** si: **CS(i)** $\leq$ **C**

- ¿Cómo sería para el caso de minimización?

Idea intutiva del Branch and Bound

Estrategias de ramificación

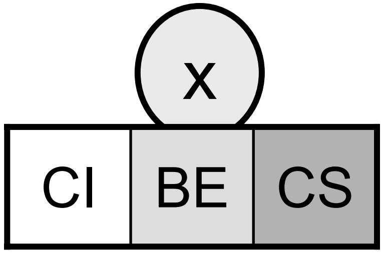
- Igual que en backtracking, hacemos un recorrido en un **árbol** de soluciones (que es **implícito**).
- **Distintos tipos de recorrido:** en profundidad, en anchura, según el beneficio estimado, etc.

- Para hacer los recorridos se utiliza una **lista de nodos vivos**.
- **Lista de nodos vivos (LNV):** contiene todos los nodos que han sido generados pero que no han sido explorados todavía. Son los nodos pendientes de tratar por el algoritmo.

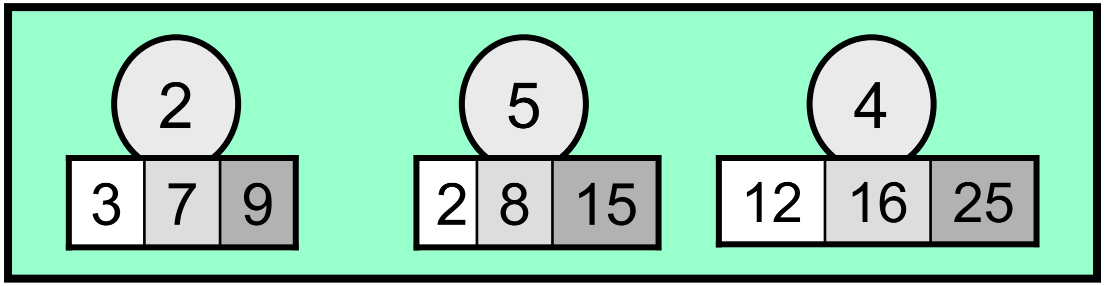
Idea intutiva del Branch and Bound

Estrategias de ramificación

- **Idea básica del algoritmo:**
  - Sacar un elemento de la lista LNV.
  - Generar sus descendientes.
  - Si no se podan, meterlos en la LNV.



LNV



- ¿En qué orden se sacan y se meten?
- Según cómo se maneje esta lista, el recorrido será de uno u otro tipo.

Idea intutiva del Branch and Bound

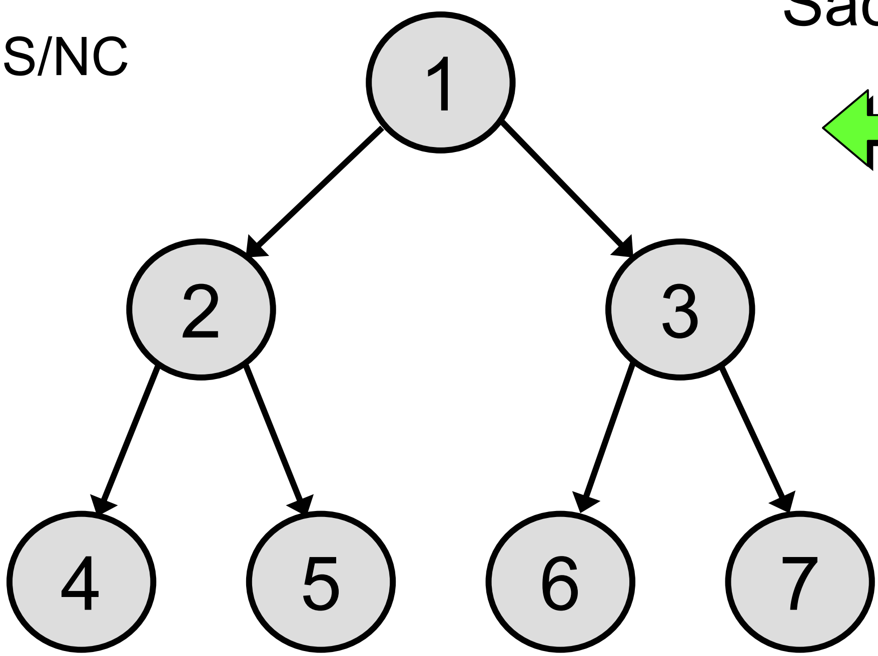
Estrategia de ramificación FIFO (First In First Out)

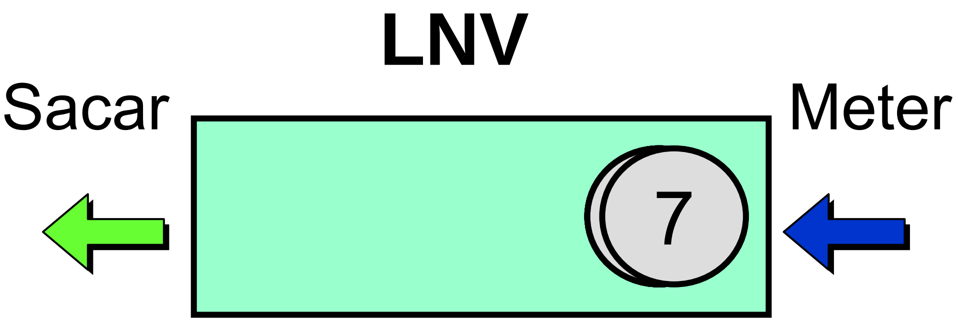
- Si se usa la estrategia FIFO, la LNV es una **cola** y el recorrido es:

a) En profundidad

b) En anchura

c) NS/NC





Idea intutiva del Branch and Bound

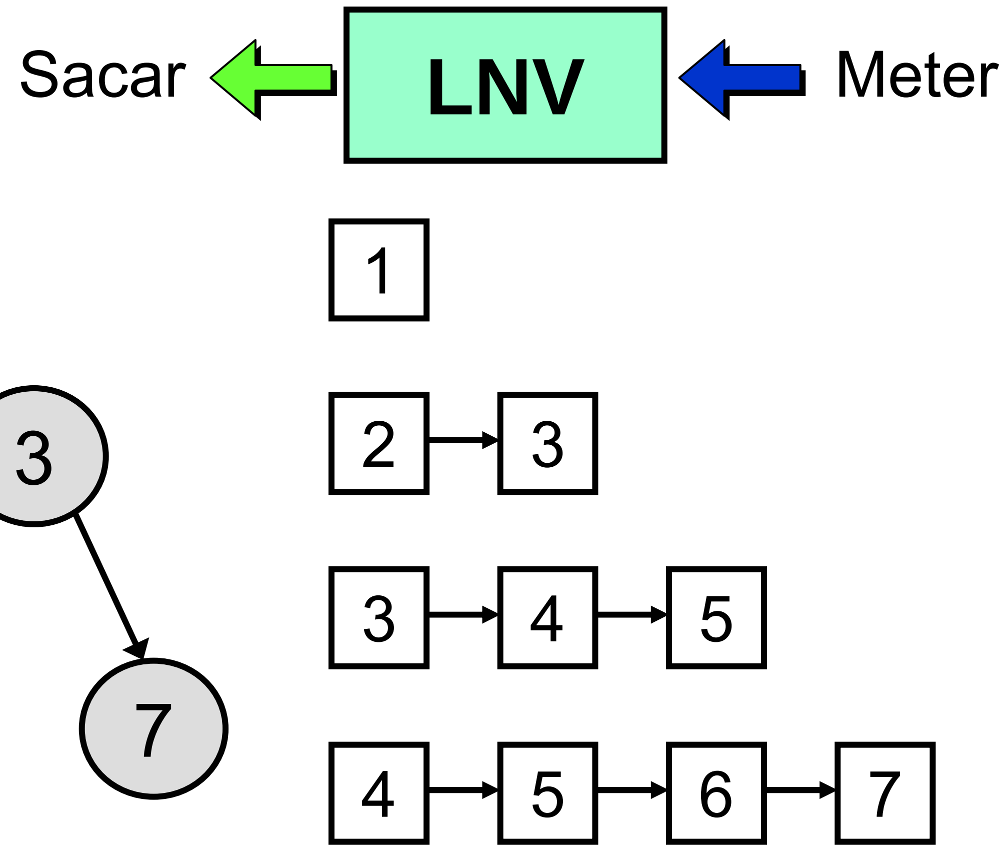
Estrategia de ramificación FIFO (First In First Out)

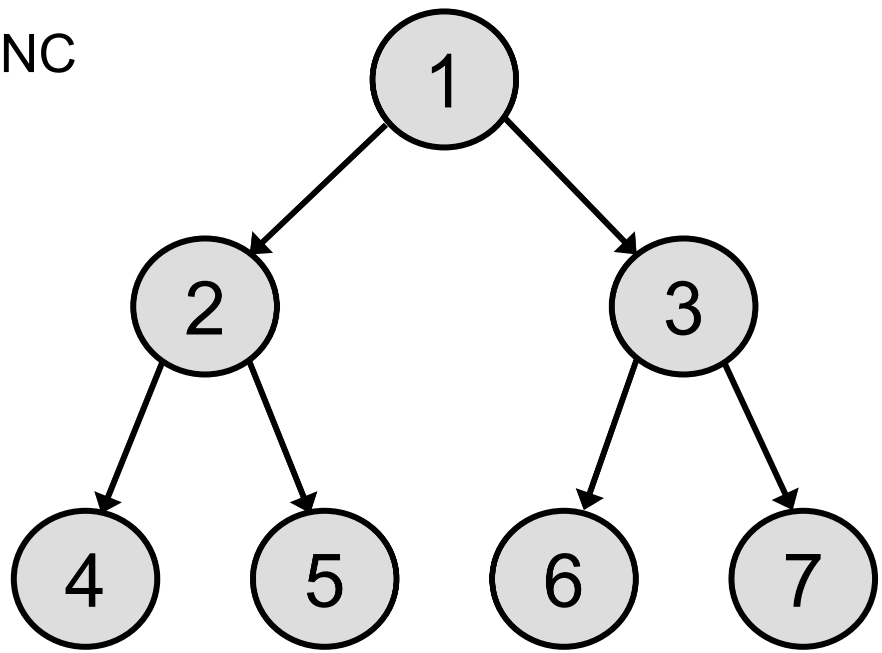
- Si se usa la estrategia FIFO, la LNV es una **cola** y el recorrido es:

a) En profundidad

b) En anchura

c) NS/NC





Idea intutiva del Branch and Bound

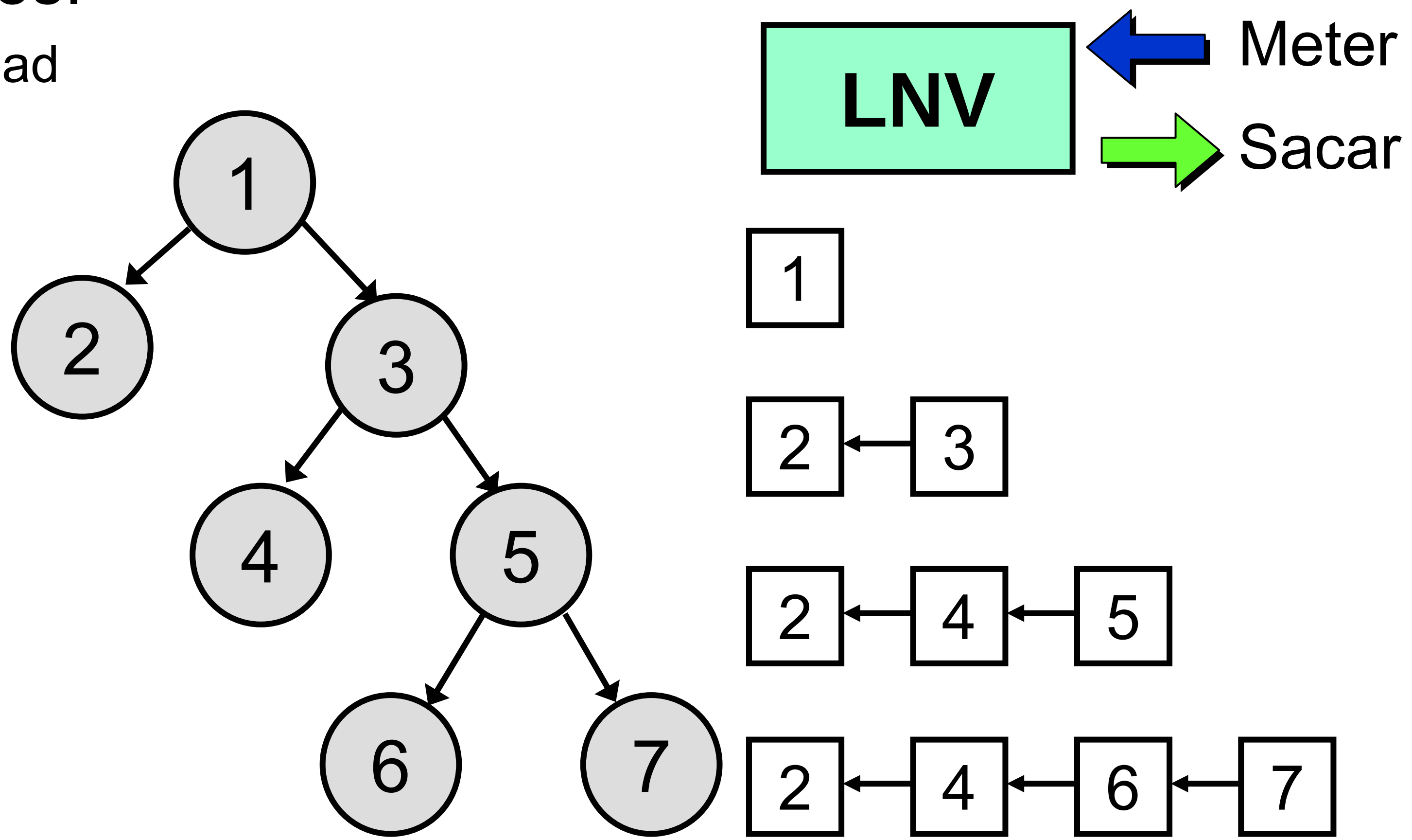
Estrategia de ramificación LIFO (Last In First Out)

- Si se usa la estrategia LIFO, la LNV es una **pila** y el recorrido es:

a) En profundidad

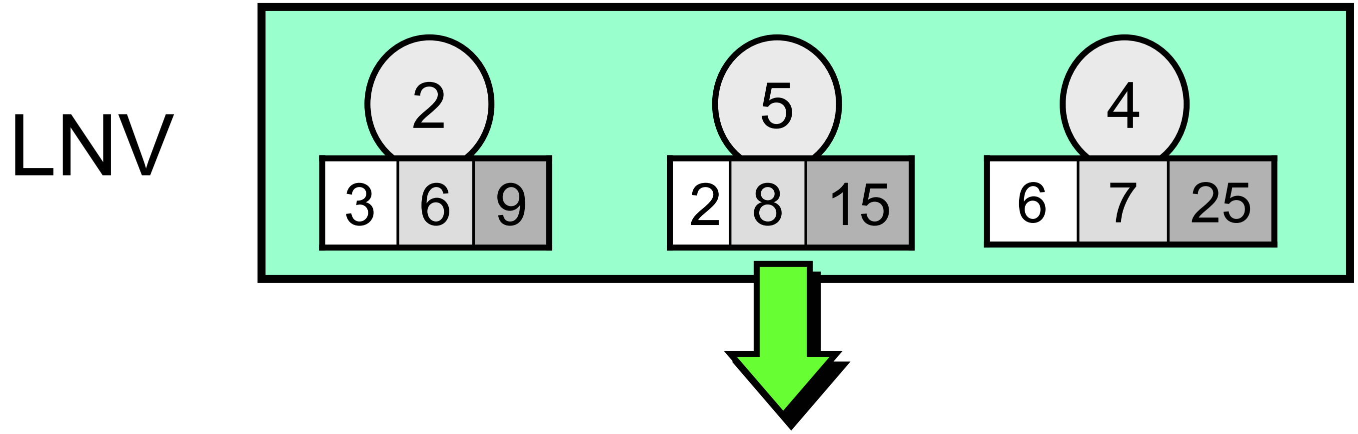
b) En anchura

c) NS/NC



Idea intutiva del Branch and Bound

- Las estrategias FIFO y LIFO realizan una búsqueda “a ciegas”, sin tener en cuenta los beneficios.
- **Usamos la estimación del beneficio:** explorar primero por los nodos con mayor valor estimado.
- **Estrategias LC (Least Cost):** Entre todos los nodos de la lista de nodos vivos, elegir el que tenga mayor beneficio (o menor coste) para explorar a continuación.



Idea intutiva del Branch and Bound

Estrategias de ramificación LC

- En caso de empate (de beneficio o coste estimado) deshacerlo usando un criterio **FIFO** ó **LIFO**.
- **Estrategia LC-FIFO:** Seleccionar de LNV el nodo que tenga mayor beneficio y en caso de empate escoger el primero que se introdujo (de los que empatan).
- **Estrategia LC-LIFO:** Seleccionar el nodo que tenga mayor beneficio y en caso de empate escoger el último que se introdujo (de los que empatan).

- ¿Cuál es mejor?
- Se diferencian si hay muchos “empates” a beneficio estimado.

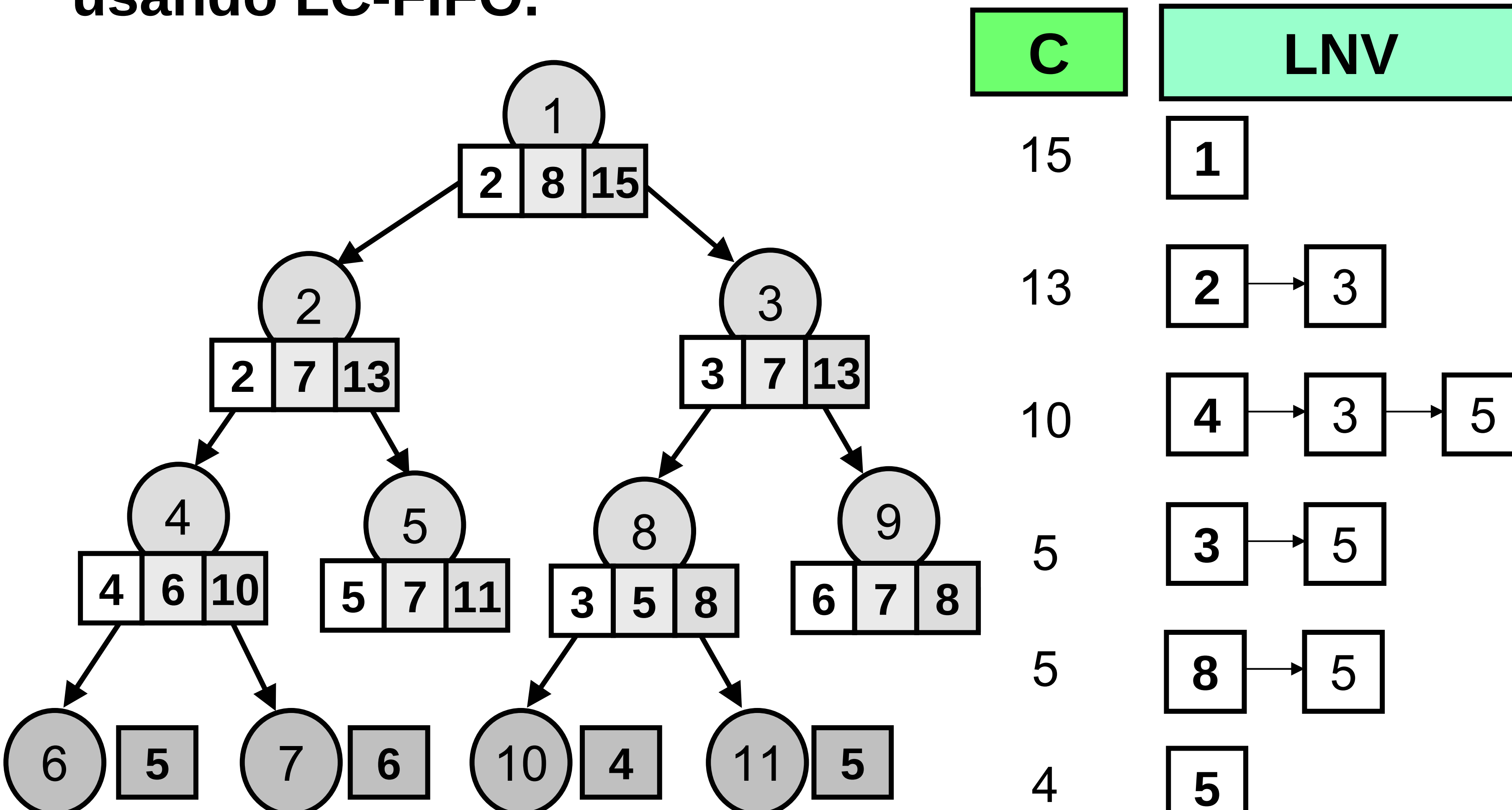
Idea intutiva del Branch and Bound

- **Resumen:**
  - En cada nodo **i** tenemos: **CI(i)**, **BE(i)** y **CS(i)**.
  - **Podar** según los valores de **CI** y **CS**.
  - **Ramificar** según los valores de **BE**.

- **Ejemplo. Recorrido con ramificación y poda, usando LC-FIFO.**
  - Suponemos un problema de minimización.
  - Para realizar la poda usamos una variable **C** = valor de la menor de las cotas superiores hasta ese momento, o de alguna solución final.
  - Si para algún nodo **i**, **CI(i)** $\geq$ **C**, entonces podar **i**.

Idea intutiva del Branch and Bound

- **Ejemplo. Recorrido con ramificación y poda, usando LC-FIFO.**



Idea intutiva del Branch and Bound

- **Esquema algorítmico de ramificación y poda.**
  - **Inicialización:** Meter la raíz en la LNV, e inicializar la variable de poda **C** de forma conveniente.
  - **Repetir** mientras no se vacíe la LNV:
    - **Sacar un nodo** de la LNV, según la estrategia de ramificación.
    - Comprobar si debe ser podado, según la **estrategia de poda**.
    - En caso contrario, **generar sus hijos**. Para cada uno:
      - Comprobar si es una **solución final** y tratarla.
      - Comprobar si debe ser **podado**.
      - En caso contrario, **meterlo en la LNV** y **actualizar C** de forma adecuada.

Idea intutiva del Branch and Bound

```
RamificacionYPoda (raiz: Nodo; var s: Nodo) // Minimización
    LNV:= {raiz}
    C:= CS(raiz)
    s:= ∅
    mientras LNV ≠ ∅ hacer
        x:= Seleccionar(LNV)        // Estrategia de ramificación
        LNV:= LNV - {x}
        si CI(x) < C entonces               // Estrategia de poda
            para cada y hijo de x hacer
                si Solución(y) AND (Valor(y)<Valor(s)) entonces
                    s:= y
                    C:= min (C, Valor(y))
                sino si NO Solución(y) AND (CI(y) < C) entonces
                    LNV:= LNV + {y}
                    C:= min (C, CS(y))
                finsi
            finpara
    finmientras
```

Idea intutiva del Branch and Bound

- **Funciones genéricas:**
  - **CI(i), CS(i), CE(i).** Cota inferior, superior y coste estimado, respectivamente.
  - **Solución(x).** Determina si **x** es una solución final válida.
  - **Valor(x)**. Valor de una solución final.
  - **Seleccionar(LNV): Nodo.** Extrae un nodo de la LNV según la estrategia de ramificación.
  - **para cada y hijo de x hacer.** Iterador para generar todos los descendientes de un nodo. Equivalente a las funciones de backtracking.

```
y:= x
mientras MasHermanos(nivel(x)+1, y) hacer
    Generar(nivel(x)+1, y)
    si Criterio(y) entonces ...
```

Idea intutiva del Branch and Bound

Algunas cuestiones

- Se comprueba el criterio de poda al meter un nodo y al sacarlo. ¿Por qué esta duplicación?
- ¿Cómo actualizar **C** si el problema es de maximizar? ¿Y cómo es la poda?
- ¿Qué información se almacena en la LNV?

LNV: Lista[Nodo]

```
tipo
   Nodo = registro
         tupla: TipoTupla      // P.ej. array [1..n] de entero
         nivel: entero
         CI, CE, CS: real
   finregistro
```

Almacenar para no recalcular. ¿Todos?

Idea intutiva del Branch and Bound

- ¿Qué pasa si para un nodo **i** tenemos que **CI(i)=CS(i)**?

- ¿Cómo calcular las cotas?

- ¿Qué pasa con las cotas si a partir de un nodo puede que no exista ninguna solución válida (factible)?

- Estas y otras cuestiones las trataremos de forma sistemática a continuación

Descripción General del Método

BB ES UN MÉTODO DE BÚSQUEDA GENERAL QUE SE APLICA CONFORME A LO SIGUIENTE:

- Explora un árbol comenzando a partir de un problema raiz (el problema original con su región factible completa)
- Entonces se aplican procedimientos de cotas inferiores y superiores al problema raiz
- Si las cotas cumplen las condiciones que se hayan establecido, habremos encontrado la solución optimal y el procedimiento termina

Descripción General del Método

- Si se encuentra una solución optimal para un subproblema, será una solución factible para el problema completo, pero no necesariamente el optimo global

- Cuando en un nodo (subproblema) la cota local es peor que el mejor valor conocido en la región, no puede existir un optimo global en el subespacio de la región factible asociada a ese nodo, y por tanto ese nodo puede ser eliminado en posteriores consideraciones

- La búsqueda sigue hasta que se examinan o “podan” todos los nodos, o hasta que se alcanza algun criterio pre-establecido sobre el mejor valor encontrado y las cotas locales de los subproblemas no resueltos

Estimadores y cotas en Branch and bound

- **Cota local:** Se calcula de forma local para cada nodo *i*. Siendo *LOptimo(i)* el coste/beneficio de la mejor solución que se podría alcanzar al expandir el nodo i, la cota local es una estimación de dicho valor que debe ser mejor o igual a *LOptimo(i)*. Cuanto más cercana sea de *LOptimo(i)* mejor será la cota y por lo tanto más se podará, pero debe haber un equilibrio entre eficiencia de cómputo y calidad de la cota.

(SE PUEDE ASEGURAR QUE NO SE ALCANZARÁ NADA MEJOR AL EXPANDIR i)

- **Cota global:** Es el valor de la mejor solución estudiada hasta el momento (o una estimación del óptimo global) y debe ser peor o igual al coste/beneficio de la solución óptima. Inicialmente se le puede asignar el valor dado por un algoritmo G*reedy*, o en su defecto el peor valor posible. Se actualiza siempre que alcancemos una solución (nodo hoja) con mejor resultado. Cuanto más cercana sea al coste/beneficio del óptimo más se podará, por lo que es importante encontrar cuanto antes soluciones buenas.

(SE PUEDE ASEGURAR QUE EL ÓPTIMO NUNCA SERÁ PEOR QUE ESTA COTA)

Estimadores y cotas en Branch and bound

- **Estimador del coste/beneficio local óptimo:** *Se calcula para cada nodo i y sirve para determinar el siguiente nodo a expandir. Es un estimador de LOptimo(i) como la cota local, pero no tiene por qué ser mejor o igual que LOptimo(i). Normalmente se utiliza la cota local como estimador, pero en el caso de que se pueda definir una medida más cercana a LOptimo(i) sin importar si es mejor o peor que LOptimo(i) podría interesar utilizar esta medida para decidir el siguiente nodo a expandir.*

- **Estrategia de poda:** *Además de podar aquellos nodos que no cumplan las restricciones implícitas (soluciones parciales no factibles) se podrán podar aquellos nodos i cuya cota local sea peor o igual que la cota global. Si sé que lo mejor que se puede alcanzar al expandir i no puede mejorar a una solución que ya se ha obtenido o se va a obtener, no es necesario expandir dicho nodo. Por la forma en la que están definidas la cota local y global se puede asegurar que no se perderá ninguna solución óptima, ya que si se cumple que,*
  - *[CotaLocal(i) mejor o igual que LOptimo(i)] y [CotaGlobal peor o igual que Óptimo],*
  - *entonces LOptimo(i) tiene que ser peor que Óptimo cuando [CotaLocal(i) peor que CotaGlobal].*

Estimadores y cotas en Branch and bound

Particularizando para problemas de minimización o maximización tenemos:

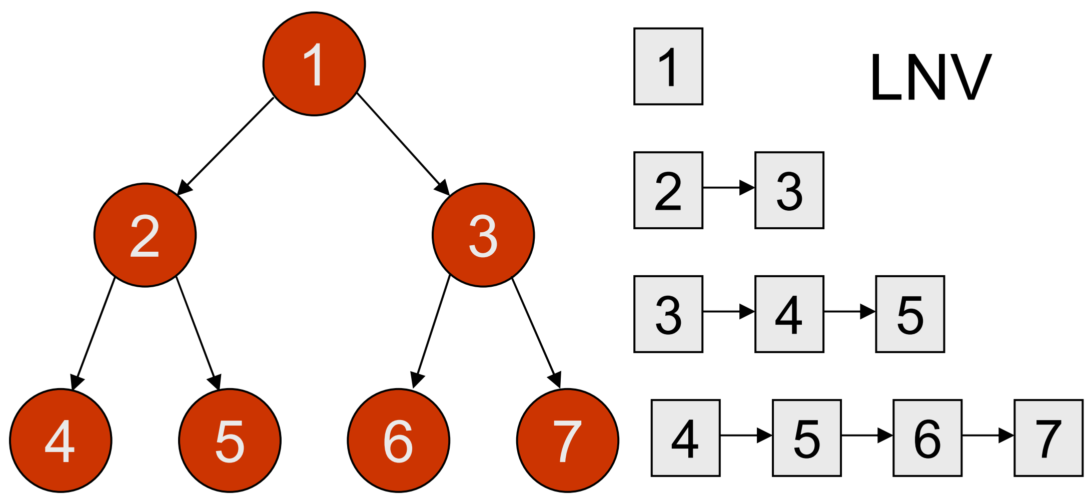
- *Minimización:*
  - *La cota local es una cota inferior CI(i) del mejor coste que se puede conseguir al expandir el nodo i, y se debe cumplir: CI(i) ≤ LOptimo(i).*
  - *La cota global es una cota superior CS del coste del óptimo global, y se debe cumplir: CS ≥ Óptimo.*
  - *En este caso se puede podar un nodo i cuando CI(i) > CS.*
- *Maximización:*
  - *La cota local es una cota superior CS(i) del máximo beneficio que se puede conseguir al expandir el nodo i, y se debe cumplir:*

    CS(i) ≥ LOptimo(i).
  - *La cota global es una cota inferior CI del beneficio del óptimo global, y se debe cumplir: CI ≤ Óptimo.*
  - *En este caso se puede podar un nodo i cuando CS(i) < CI.*

Estrategias de ramificación

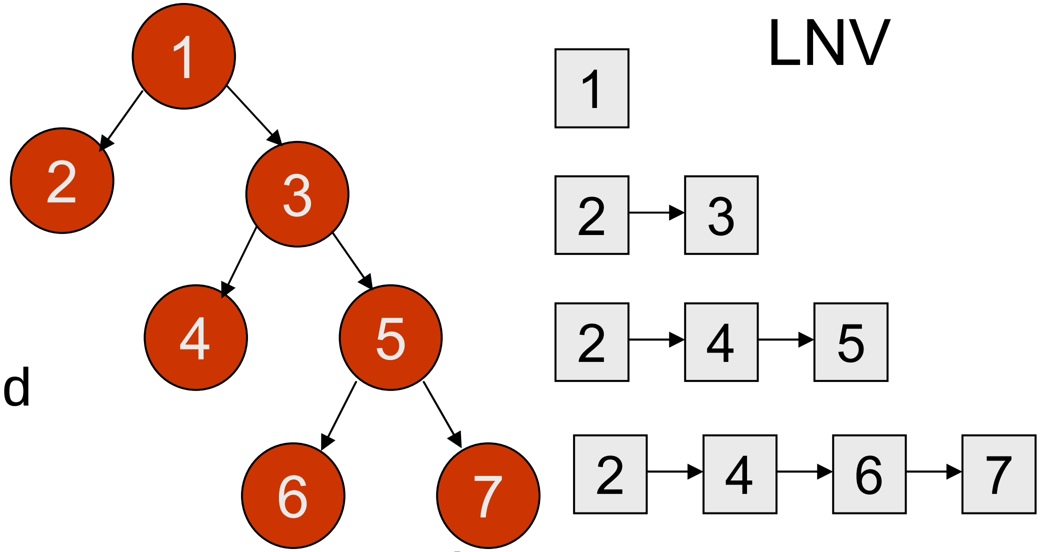
- Normalmente el árbol de soluciones es implícito, no se almacena en ningún lugar
- Para hacer el recorrido se utiliza una **lista de nodos vivos**
- **Lista de nodos vivos:** contiene todos los nodos que han sido generados pero que no han sido explorados todavía. Son los nodos pendientes de tratar por el algoritmo
- Según cómo sea la lista, el recorrido será de uno u otro tipo.

ESTRATEGIA FIFO (First In First Out)
La lista de nodos vivos es una cola
El recorrido es en anchura



Estrategias de ramificación

ESTRATEGIA LIFO
(Last In First Out)
La lista de nodos vivos
es una pila
El recorrido es en profundidad



- Las estrategias FIFO y LIFO realizan una búsqueda "a ciegas", sin tener en cuenta los beneficios
- Usando la estimación del beneficio, entonces será mejor buscar primero por los nodos con mayor valor estimado
- **ESTRATEGIAS LC (Least Cost):**
  - Entre todos los nodos de la lista de nodos vivos, elegir el que tenga mayor beneficio (o menor coste) para explorar a continuación

Estrategias de ramificación LC

- En caso de empate (de beneficio o coste estimado) deshacerlo usando un criterio FIFO ó LIFO:
  - **Estrategia LC-FIFO:** Seleccionar de la LNV el que tenga mayor beneficio y en caso de empate escoger el primero que se introdujo (de los que empatan)
  - **Estrategia LC-LIFO:** Seleccionar de la LNV el que tenga mayor beneficio y en caso de empate escoger el último que se introdujo (de los que empatan)

- En cada nodo podemos tener: cota inferior de coste, coste estimado y cota superior del coste

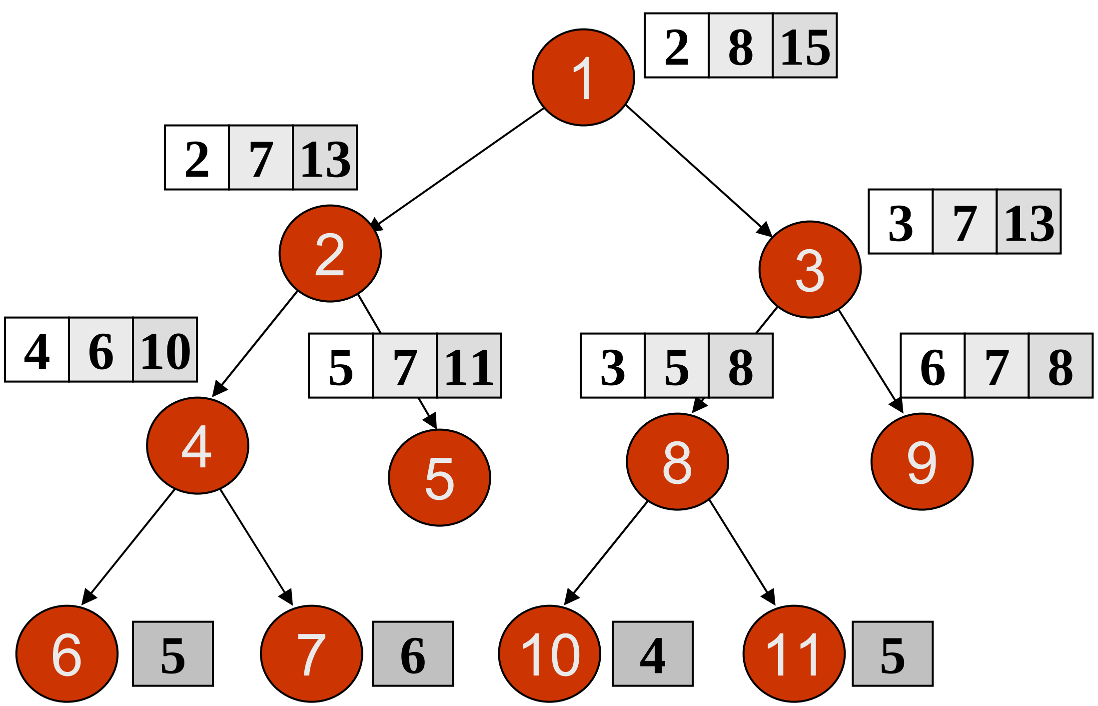
- Podar según los valores de las cotas

- Ramificar según los costes estimados

BB: El Método General

Ejemplo. Branch and Bound usando LC-FIFO:

- Supongamos un problema de minimización, y que tenemos el caso 1 (a partir de un nodo siempre existe alguna solución)
- Para realizar la poda usaremos una variable ***C*** = valor de la menor de las cotas superiores hasta ese momento (o de alguna solución final)
- Si para algún nodo *i*, *CI(i)* $\geq$ *C*, entonces podar *i*



C	LNV
15	1
13	2 → 3
10	4 → 3 → 5
5	3 → 5
5	8 → 5
4	5

BB: El Método General

Algunas cuestiones

- Sólo se comprueba el criterio de poda cuando se introduce o se saca un nodo de la lista de nodos vivos
- Si un descendiente de un nodo es una solución final entonces no se introduce en la lista de nodos vivos. Se comprueba si esa solución es mejor que la actual, y se actualiza ***C*** y el valor de la mejor solución óptima de forma adecuada

- ¿Qué pasa si a partir de un nodo solución pueden haber otras soluciones?
- ¿Cómo debe ser actualizada la variable ***C*** (variable de poda) si el problema es de maximización?
- ¿Cómo será la poda, para cada uno de los casos anteriores?
- ¿Cuándo acaba el algoritmo?
- ¿Cómo calcular las cotas?

BB: El Método General

- **Esquema general.** Problema de minimización, suponiendo el caso en que existe solución a partir de cualquier nodo

```
RamificacionYPoda (NodoRaiz: tipo_nodo; var s: tipo_solucion);
    LNV:= {NodoRaiz};
    C, s := CS (NodoRaiz); {Primera solución y cota asociada}
    Mientras LNV ≠ ∅ hacer
        x:= Seleccionar (LNV); {Según un criterio FIFO, LIFO, LC-FIFO ó LC-LIFO}
        LNV:= LNV - {x};
        Si CI (x) <= C entonces          { Si no se cumple se poda x }
            Para cada y hijo de x hacer
                Si y es una solución final mejor que s entonces
                    s:= y;
                    C:= Coste (y);
                Sino si y no es solución final y (CI(y) <= C) entonces
                    LNV:= LNV + {y};
                    Ctmp, s_tmp := CS (y);
                    si (Ctmp < C)
                        C:= Ctmp;
                        s := s_tmp;
            FinPara;
    FinMientras;
```

BB: Análisis de tiempos de ejecución

- El tiempo de ejecución depende de:
  - **Número de nodos recorridos**: depende de la efectividad de la poda
  - **Tiempo gastado en cada nodo**: tiempo de hacer las estimaciones de coste y tiempo de manejo de la lista de nodos vivos
- En el peor caso, el tiempo es igual que el de un algoritmo con back-tracking (ó peor si tenemos en cuenta el tiempo que requiere la LNV)
- En el caso promedio se suelen obtener mejoras respecto a backtracking
- ¿Cómo hacer que un algoritmo BB sea más eficiente?
  - **Hacer estimaciones de costo muy precisas**: Se realiza una poda exhaustiva del árbol. Se recorren menos nodos pero se gasta mucho tiempo en realizar las estimaciones
  - **Hacer estimaciones de costo poco precisas**: Se gasta poco tiempo en cada nodo, pero el número de nodos puede ser muy elevado. No se hace mucha poda
- Se debe buscar un equilibrio entre la exactitud de las cotas y el tiempo en calcularlas

Índice

IV. SOLUCIONES BRANCK-BOUND EN DISTINTOS PROBLEMAS

1. El Problema de la Mochila 0/1

2. El Problema del Viajante de Comercio

El Problema de la Mochila 0/1

- **Diseño del algoritmo** BB:
  - Definir una representación de la solución. A partir de un nodo, cómo se obtienen sus descendientes
  - Dar una manera de calcular el valor de las cotas y la estimación del beneficio
  - Definir la estrategia de ramificación y de poda

- **Representación de la solución:**
  - **Mediante un árbol binario**: $(s_1, s_2, \ldots, s_n)$, con $s_i = (0, 1)$
    Hijos de un nodo $(s_1, s_2, \ldots, s_k)$:
    $$(s_1, \ldots, s_k, 0) \text{ y } (s_1, \ldots, s_k, 1)$$

  - **Mediante un árbol combinatorio**: $(s_1, s_2, \ldots, s_m)$ donde $m \leq n$ y $s_i \in \{1, 2, \ldots, n\}$
    Hijos de un nodo $(s_1, .., s_k)$:
    $$(s_1, .., s_k, s_k+1), (s_1, \ldots, s_k, s_k+2), \ldots, (s_1, \ldots, s_k, n)$$

El Problema de la Mochila 0/1

- **Cálculo de cotas:**
  - **Cota inferior**: Beneficio que se obtendría sólo con los objetos incluidos hasta ese nodo
  - **Estimación del beneficio**: A la solución actual, sumar el beneficio de incluir los objetos enteros que quepan, utilizando avance rápido. Suponemos que los objetos están ordenados por orden decreciente de v_i/w_i
  - **Cota superior**: Resolver el problema de la mochila continuo a partir de ese nodo (con un algoritmo ~~greedy~~), y quedarse con la parte entera.

- **Ejemplo**. $n = 4$, $M = 7$, $v = (2, 3, 4, 5)$, $w = (1, 2, 3, 4)$

  Nodo actual: $(1, 1)$ $(1, 2)$

  Hijos: $(1, 1, 0), (1, 1, 1)$ $(1, 2, 3), (1, 2, 4)$

  Cota inferior: $CI = v_1 + v_2 = 2 + 3 = 5$

  Estimación del beneficio: $EB = CI + v_3 = 5 + 4 = 9$

  Cota superior: $CS = CI + \lfloor MochilaGreedy(3, 4) \rfloor = 5 + \lfloor 4 + 5/4 \rfloor = 10$

El Problema de la Mochila 0/1

- **Forma de realizar la poda:**
  - En una variable ***C*** guardar el valor de la mayor cota inferior hasta ese momento dado
  - Si para un nodo, su cota superior es menor o igual que ***C*** entonces se puede podar ese nodo

- **Estrategia de ramificación:**
  - Puesto que tenemos una estimación del coste, usar una estrategia ***LC***: explorar primero las ramas con mayor valor esperado *(MB)*
  - ¿*LC-FIFO* ó *LC-LIFO*? Usaremos la *LC-LIFO*: en caso de empate seguir por la rama más profunda (*MB-LIFO*)

El Problema de la Mochila 0/1

```
Mochila01BB (v, w: array [1..n] of integer; M: integer; var s: nodo);
    inic:= NodoInicial (v, w, M);
    C:= inic.CI;
    LNV:= {inic};
    s.v_act:= -∞;
    Mientras LNV ≠ ∅ hacer
        x:= Seleccionar (LNV);            {Según el criterio MB-LIFO}
        LNV:= LNV - {x};
        Si  x.CS > C  Entonces            {Si no se cumple se poda x}
            Para  i:= 0, 1  Hacer
                y:= Generar (x, i, v, w, M);
                Si (y.nivel = n)  Y  (y.v_act > s.v_act)  Entonces
                    s:= y;
                    C:= max (C, s.v_act );
                Sino Si  (y.nivel < n)  Y  (y.CS > C)  Entonces
                    LNV:= LNV + {y};
                    C:= max (C, y.CI );
            FinPara;
        FinSi;
    FinMientras;
```

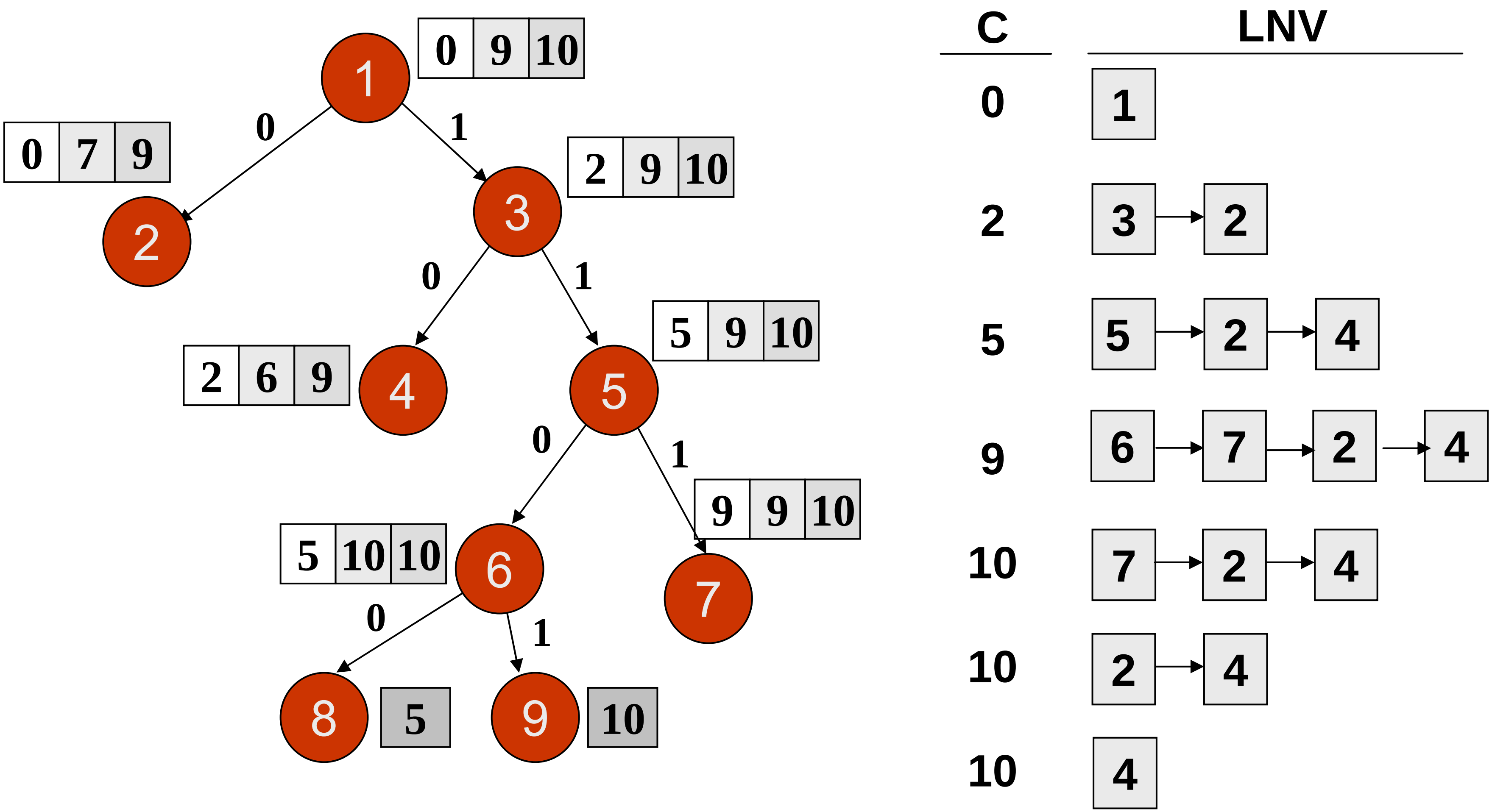
El Problema de la Mochila 0/1

```
NodoInicial (v, w: array [1..n] of integer; M: integer) : nodo;
    res.CI:= 0;
    res.CS:= ⌊MochilaVoraz (1, M, v, w)⌋;
    res.BE:= Mochila01Voraz (1, M, v, w);
    res.nivel:= 0;
    res.v_act:= 0;          res.w_act:= 0;
    Devolver res;

Generar (x: nodo; i: (0, 1); v,w: array [1..n] of int; M: int): nodo;
    res.tupla:= x.tupla;
    res.nivel:= x.nivel + 1;
    res.tupla[res.nivel]:= i;
    Si  i = 0  Entonces  res.v_act:= x.v_act;  res.w_act:= x.w_act;
    Sino  res.v_act:= x.v_act + v[res.nivel]; res.w_act:= x.w_act +
                        w[res.nivel];
    res.CI:= res.v_act;
    res.BE:= res.CI + Mochila01Voraz (res.nivel+1, M - res.w_act, v, w);
    res.CS:= res.CI + ⌊MochilaVoraz (res.nivel+1, M - res.w_act, v, w)⌋;
    Si res.w_act > M Entonces   {Sobrepasa el peso M: descartar el nodo}
        res.CI:= res.CS:= res.BE:= -∞;
    Devolver res;
```

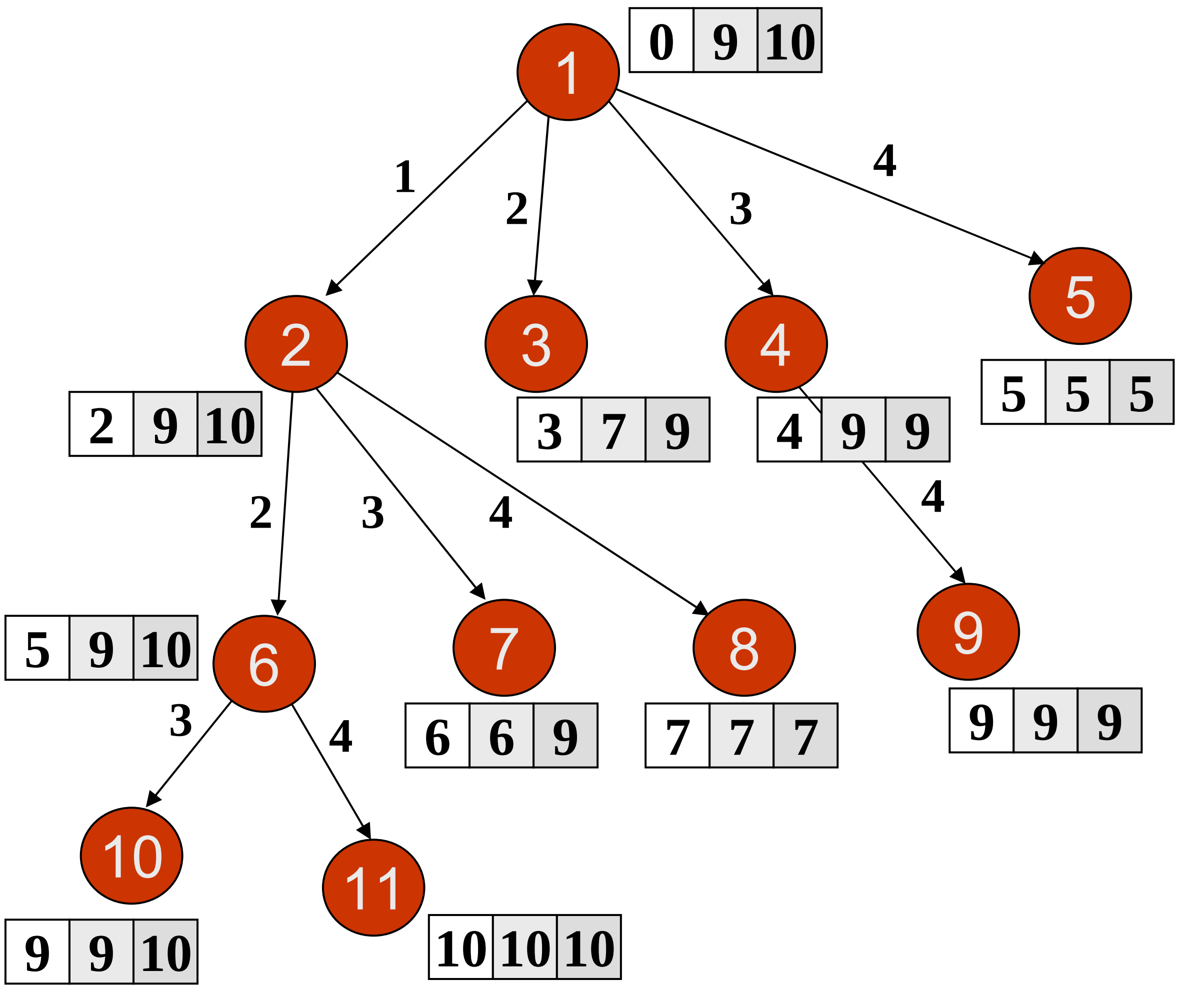
El Problema de la Mochila 0/1

- **Ejemplo**. *n = 4, M = 7, v = (2, 3, 4, 5), w = (1, 2, 3, 4)*



El Problema de la Mochila 0/1

- **Ejemplo**. Utilizando un árbol combinatorio y *LC-FIFO*, *n = 4*, *M = 7*, *v = (2, 3, 4, 5)*, *w = (1, 2, 3, 4)*



C	LNV
0	1
5	2 → 4 → 3
7	4 → 6 → 3 → 7
9	6 → 3 → 7
10	3 → 7
10	7

Recordemos...

Aplicación de ramificación y poda (proceso metódico):

1) Definir la representación de la solución. A partir de un nodo, cómo se obtienen sus descendientes.
2) Dar una manera de calcular el valor de las cotas y la estimación del beneficio.
3) Definir la estrategia de ramificación y de poda.
4) Diseñar el esquema del algoritmo.

El problema de asignación de tareas

Enunciado del problema de asignación

- **Datos del problema:**
  - **n:** número de personas y de tareas disponibles.
  - **B: array** [1..n, 1..n] **de** entero. Rendimiento o beneficio de cada asignación. **B[i, j]** = beneficio de asignar a la persona **i** la tarea **j**.
- **Resultado:**
  - Realizar **n** asignaciones $\{(p_1, t_1), (p_2, t_2), ..., (p_n, t_n)\}$.
- **Formulación matemática:**

Maximizar $\sum_{i=1..n} B[p_i, t_i]$, sujeto a

la restricción $p_i \neq p_j$, $t_i \neq t_j$, $\forall$ $i \neq j$

Tareas

Personas

B	1	2	3
1	5	6	4
2	3	8	2
3	6	5	1

El problema de asignación de tareas

1) Representación de la solución

- Desde el punto de vista de las **personas:**
  $s = (t_1, t_2, ..., t_n)$, siendo $t_i \in \{1, ..., n\}$, con $t_i \neq t_j$, $\forall\ i \neq j$
  - **t_i** → número de tarea asignada a la persona **i**.

tipo
  Nodo = **registro**
    tupla: **array** [1..n] **de** entero
    nivel: entero
    bact: entero
    CI, BE, CS: entero
  finregistro

1.a) ¿Cómo es el nodo raíz?

1.b) ¿Cómo generar los hijos de un nodo?

1.c) ¿Cómo es la función Solución(x: Nodo): booleano?

El problema de asignación de tareas

1.a) Nodo raíz

```
raiz.nivel:= 0
raiz.bact:= 0
```

1.b) Para cada y hijo de un nodo x

```
para i:= 1, ..., n hacer
    y.nivel:= x.nivel+1
    y.tupla:= x.tupla
    si Usada(x, i) entonces break
    y.tupla[y.nivel]:= i
    y.bact:= x.bact + B[y.nivel, i]
    .....

operación Usada(m: Nodo; t: entero): booleano
    para i:= 1,..., m.nivel hacer
            si m.tupla[i]==t entonces devolver TRUE
    devolver FALSE
```

El problema de asignación de tareas

- **Otra posibilidad:** almacenar las tareas usadas en el nodo.
  tipo
  Nodo = **registro**
  tupla: **array** [1..n] **de** entero
  nivel: entero
  bact: entero
  usadas: **array** [1..n] **de** booleano
  CI, BE, CS: entero
  finregistro

- **Resultado:** se tarda menos tiempo pero se usa más memoria.

1.c) Función Solución(x: Nodo): booleano

devolver x.nivel==n

El problema de asignación de tareas

2) Cálculo de las funciones CI(x), CS(x), BE(x)

Tareas

Personas

B	1	2	3
1	5	6	4
2	3	8	2
3	6	5	1

2) Posibilidad 1. Estimaciones triviales:

- **CI.** Beneficio acumulado hasta ese momento: x.CI:= x.bact
- **CS.** CI más suponer las restantes asignaciones con el máximo global: x.CS:= x.bact + (n-x.nivel)*max(B[·,·])
- **BE.** La media de las cotas: x.BE:= (x.CI+x.CS)/2

El problema de asignación de tareas

2) Posibilidad 2. Estimaciones precisas:

- **CI.** Resolver el problema usando un algoritmo voraz.
  x.CI:= x.bact + AsignaciónVoraz(x)

- **AsignaciónVoraz(x):** Asignar a cada persona la tarea libre con más beneficio.
  operación AsignaciónVoraz(m: Nodo): entero

```
        bacum:= 0
        para i:= m.nivel+1, ..., n hacer
                k:= argmax_{∀j∈{1..n}, m.usadas[j]==FALSE} B[i, j]
                m.usadas[k]:= TRUE
                bacum:= bacum + B[i, k]
        finpara
        devolver bacum
```

El problema de asignación de tareas

2) Posibilidad 2. Estimaciones precisas:

- **CS.** Asignar las tareas con mayor beneficio (aunque se repitan).
  x.CS:= x.bact + MáximosTareas(x)

  operación MáximosTareas(m: Nodo): entero

  bacum:= 0

  para i:= m.nivel+1, ..., n **hacer**

  k:= $\text{argmax}_{\substack{\forall j \in \{1..n\} \\ m.usadas[j]==FALSE}}$ B[i, j]

  bacum:= bacum + B[i, k]

  finpara

  devolver bacum

- **BE.** Tomar la media: x.BE:= (x.CI+x.CS)/2

El problema de asignación de tareas

2) Cálculo de las funciones CI(x), CS(x), BE(x)

- **Cuestión clave:** ¿podemos garantizar que la solución óptima a partir de **x** estará entre **CI(x)** y **CS(x)**?

- **Ejemplo. n**= 3. ¿Cuánto serían **CI(raíz)**, **CS(raíz)** y **BE(raíz)**? ¿Cuál es la solución óptima del problema?

Tareas

Personas

B	**1**	**2**	**3**
1	5	6	4
2	3	8	2
3	6	5	1

El problema de asignación de tareas

3) Estrategia de ramificación y de poda

3.a) Estrategia de poda

- **Variable de poda C:** valor de la mayor cota inferior o solución final del problema.
- **Condición de poda:** podar **i** si: **i.CS ≤ C**

3.b) Estrategia de ramificación

- **Usar una estrategia MB-LIFO:** explorar primero los nodos con mayor BE y en caso de empate seguir por la rama más profunda.

El problema de asignación de tareas

4) Esquema del algoritmo. (Exactamente el mismo que antes)

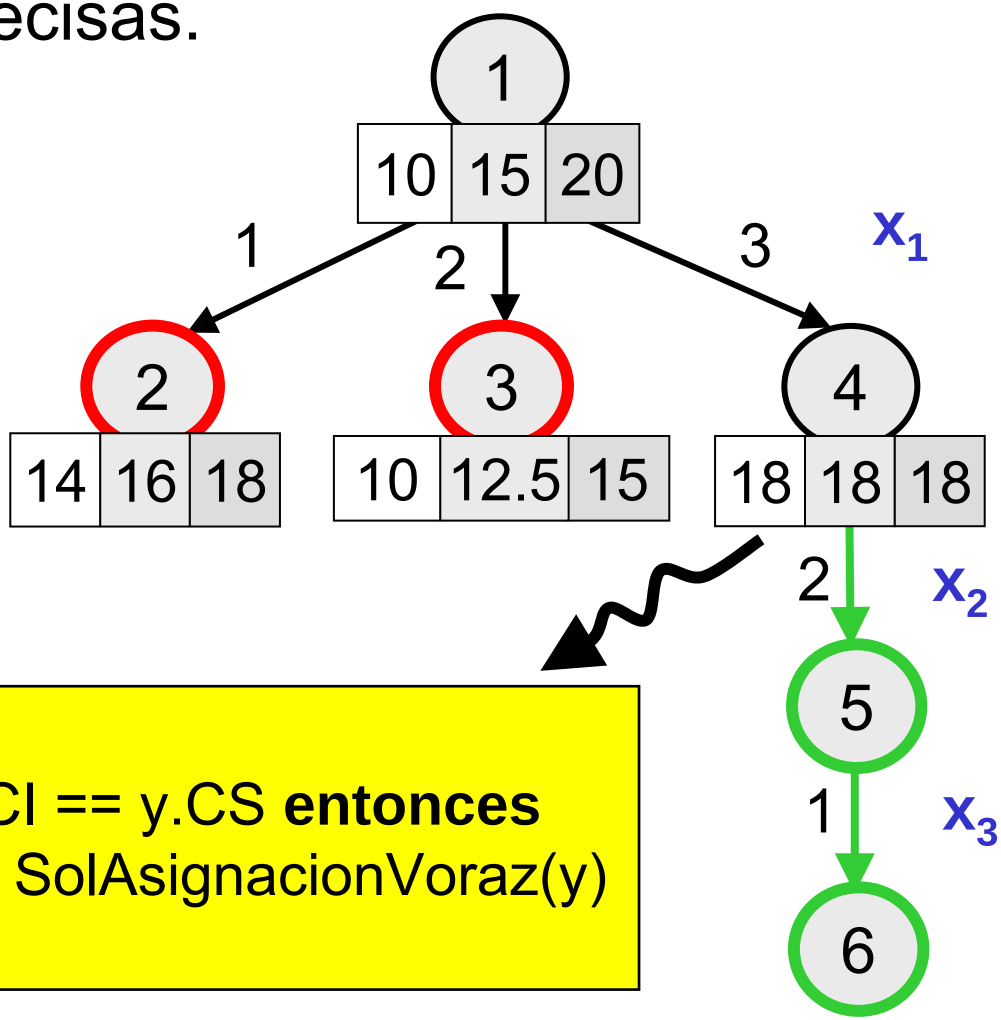
```
AsignaciónRyP (n: ent; B: array[1..n,1..n] de ent; var s: Nodo)
   LNV:= {raiz}
   C:= raiz.CI
   s:= ∅
   mientras LNV ≠ ∅ hacer
         x:= Seleccionar(LNV)       // Estrategia MB-LIFO
         LNV:= LNV - {x}
         si x.CS > C entonces             // Estrategia de poda
            para cada y hijo de x hacer
               si Solución(y) AND (y.bact > s.bact) entonces
                  s:= y
                  C:= max (C, y.bact)
               sino si NO Solución(y) AND (y.CS > C) entonces
                  LNV:= LNV + {y}
                  C:= max (C, y.CI)
               finsi
            finpara
   finmientras
```

El problema de asignación de tareas

- **Ejemplo. n=** 3. Estimaciones precisas.

Tareas / Personas

B	1	2	3
1	5	6	4
2	3	8	2
3	6	5	1



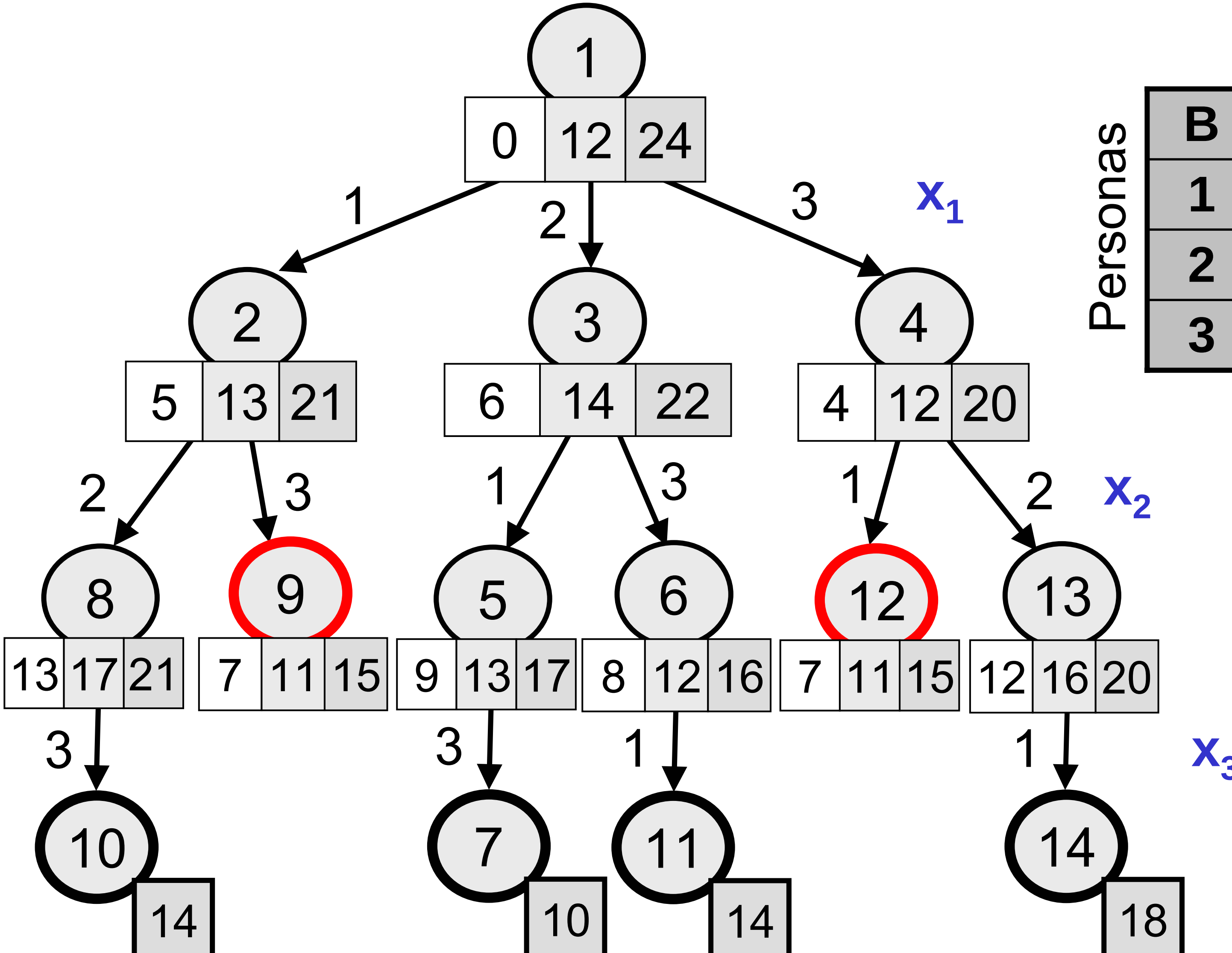
...
si y.CI == y.CS **entonces**
  y:= SolAsignacionVoraz(y)
...

s= (3, 2, 1)

C	LNV
10	1
18	2, 3
18	3

El problema de asignación de tareas

- **Ejemplo. n**= 3. Usando las estimaciones triviales.



Tareas

Personas

B	1	2	3
1	5	6	4
2	3	8	2
3	6	5	1

s= (3, 2, 1)

El problema de asignación de tareas

- Con estimaciones precisas: 4 nodos generados.
- Con estimaciones triviales: 14 nodos generados.

- ¿Conviene gastar más tiempo en hacer estimaciones más precisas?

- ¿Cuánto es el tiempo de ejecución en el peor caso?
- Estimaciones triviales: **O(1)**
- Estimaciones precisas: **O(n(n-nivel))**

El Problema del Viajante de Comercio

- Este problema fue resuelto con Programación Dinámica, obteniendo un algoritmo de orden ***$O(n^2 2^n)$...***
- Para un '*n*' grande, el algoritmo es ineficiente...
- Branch and bound se adapta para solucionarlo

Recordatorio:

- Encontrar un recorrido de longitud mínima para una persona que tiene que visitar varias ciudades y volver al punto de partida, conocida la distancia existente entre cada dos ciudades.
- Es decir, dado un grafo dirigido con arcos de longitud no negativa, se trata de encontrar un circuito de longitud mínima que comience y termine en el mismo vértice y pase exactamente una vez por cada uno de los vértices restantes

El Problema del Viajante de Comercio

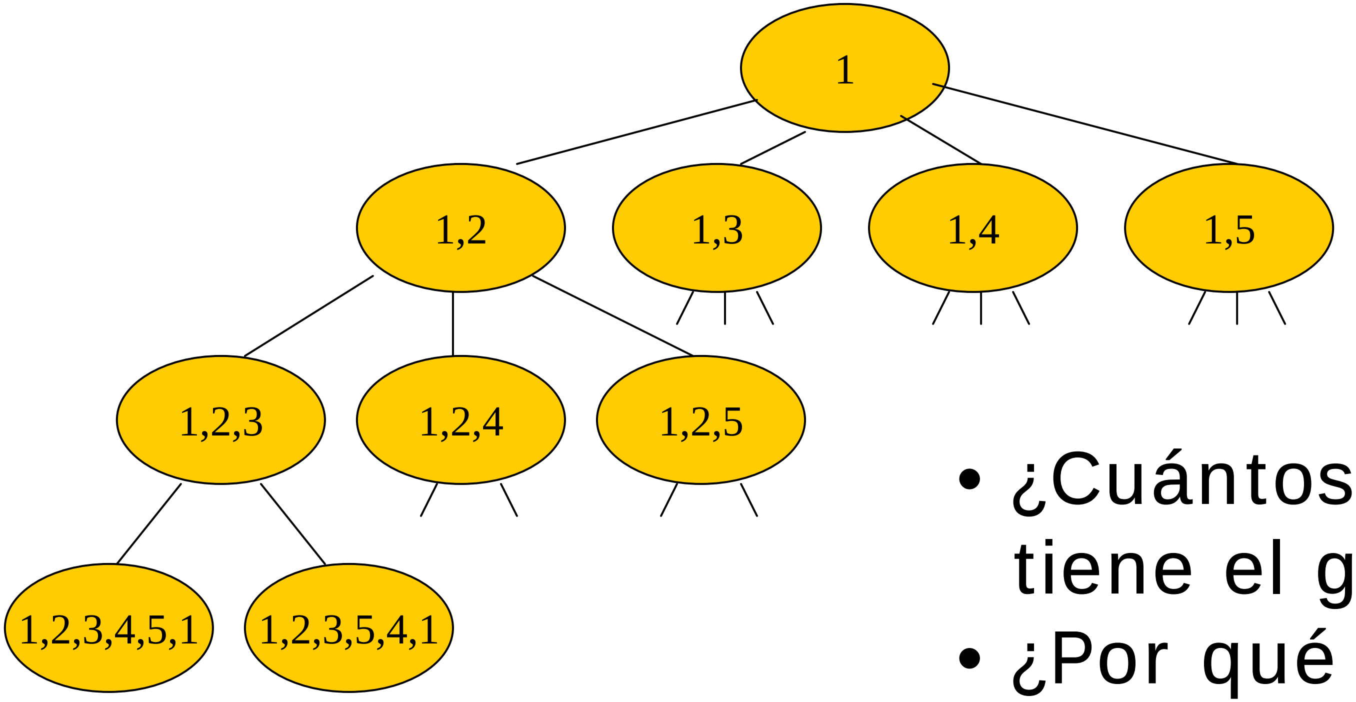
- Formalización:
  - Sean $G = (V,A)$ un grafo dirigido, $V=\{1,2,\ldots,n\}$,
  - $D[i,j]$ la longitud de $(i,j) \in A$, $D[i,j]=\infty$ si no existe el arco (i,j)
  - El circuito buscado empieza en el vértice *1*
  - Candidatos:
    $E = \{ 1,X,1 \mid X \text{ es una permutación de } (2,3,\ldots,n) \}$
    $|E| = (n-1)!$
  - Soluciones factibles:
    $E = \{ 1,X,1 \mid X = x_1,x_2,\ldots,x_{n-1}, \text{ es una permutación de } (2,3,\ldots,n) \text{ tal que } (i_j,i_{j+1}) \in A,\ 0<j<n,\ (1, x_1) \in A\ ,\ (x_{n-1},1) \in A\}$
  - Función objetivo:
    $F(X)=D[1,x_1]+D[x_1, x_2] + D[x_2, x_3]+\ldots+D[x_{n-2}, x_{n-1}]+D[x_n,1]$

El Problema del Viajante de Comercio

- **Recordatorio**: Ciclo en el grafo en el que TODOS los vértices del grafo se visitan sólo una vez al menor costo

- Arbol de búsqueda de soluciones:
  - La raíz del árbol (nivel 0) es el vértice de inicio del ciclo
  - En el nivel 1 se consideran TODOS los vértices menos el inicial
  - En el nivel 2 se consideran TODOS los vértices menos los 2 que ya fueron visitados
  - Y así sucesivamente hasta el nivel ‘n-1’ que incluirá al vértice que no ha sido visitado

Ejemplo



- ¿Cuántos vértices tiene el grafo?
- ¿Por qué no se requiere el último nivel en el árbol?

Análisis del problema con Branch and Bound

- Criterio de selección para expandir un nodo del árbol de búsqueda de soluciones:
  - Un vértice en el nivel *i* del árbol, debe ser adyacente al vértice en el nivel *i-1* del camino correspondiente en el árbol
  - Puesto que es un problema de Minimización, si el costo posible a acumular al expandir el nodo i, es menor al mejor costo acumulado hasta ese momento, vale la pena expandir el nodo, si no, el camino se deja de explorar ahí…

Estimación del costo posible a acumular

- Si se sabe cuáles son los vértices que faltan por visitar
- Cada vértice faltante, tiene arcos de salida hacia otros vértices
- El mejor costo, será el del arco que tenga el valor menor
- Esta información se puede obtener del renglón correspondiente al vértice en la matriz de adyacencias (excluyendo los ceros)
- La sumatoria de los mejores arcos de cada vértice faltante, más el costo del camino ya acumulado, es una estimación válida para tomar decisiones respecto a las podas en el árbol

Ejemplo

- Dada la siguiente matriz de adyacencias, ¿cuál es el costo mínimo posible de visitar todos los nodos una sola vez?

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix} \begin{array}{l} \longrightarrow \text{Mínimo} = 4 \\ \longrightarrow \text{Mínimo} = 7 \\ \longrightarrow \text{Mínimo} = 4 \\ \longrightarrow \text{Mínimo} = 2 \\ \longrightarrow \text{Mínimo} = 4 \end{array}$$

TOTAL = 21

Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$

1

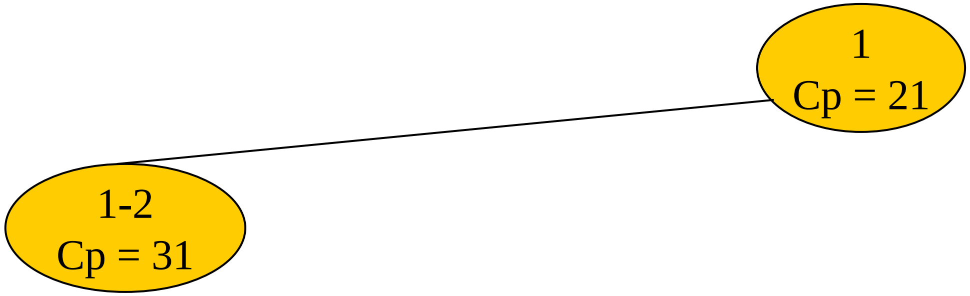
Cp = 21

Costo máximo = ∞

Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$

Costo máximo = ∞



Cálculo del Costo posible:

Acumulado de 1-2 : **14**

Más mínimo de 2-3, 2-4 y 2-5: **7**

Más mínimo de 3-1, 3-4 y 3-5: **4**

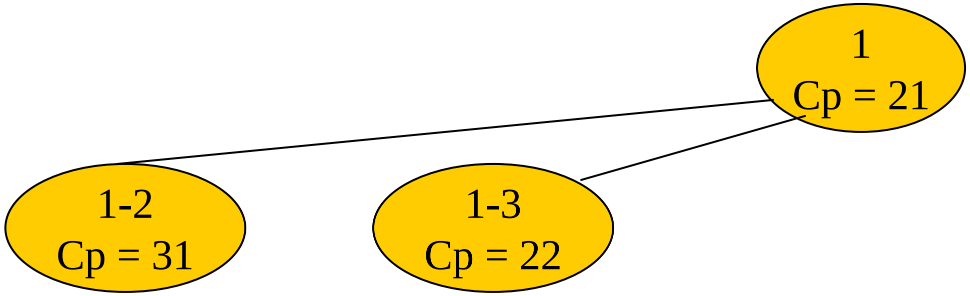
Más mínimo de 4-1, 4-3 y 4-5: **2**

Más mínimo de 5-1, 5-3 y 5-4: **4**

TOTAL = 31

Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$



Costo máximo = ∞

<u>Cálculo del Costo posible:</u>

Acumulado de 1-3 : **4**

Más mínimo de 3-2, 3-4 y 3-5: **5**

Más mínimo de 2-1, 2-4 y 2-5: **7**

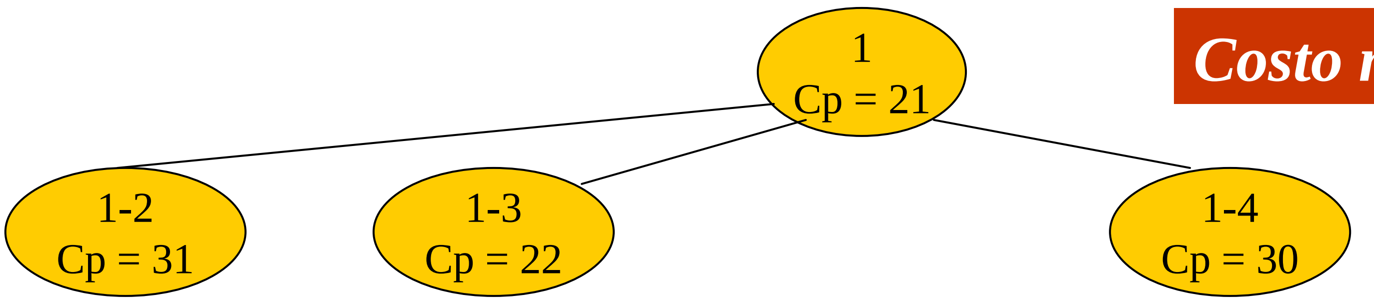
Más mínimo de 4-1, 4-3 y 4-5: **2**

Más mínimo de 5-1, 5-3 y 5-4: **4**

TOTAL = 22

Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$



Costo máximo = ∞

Cálculo del Costo posible:

Acumulado de 1-4 : **10**

Más mínimo de 4-2, 4-3 y 4-5: **2**

Más mínimo de 3-1, 3-2 y 3-5: **4**

Más mínimo de 2-1, 2-3 y 2-5: **7**

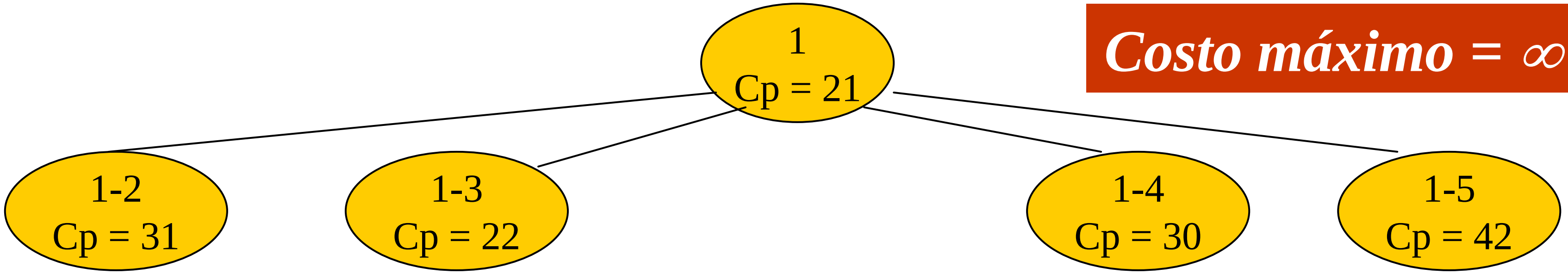
Más mínimo de 5-1, 5-2 y 5-3: **7**

TOTAL = 30

Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$

Costo máximo = ∞



¿Cuál es el mejor nodo para expandir?

Cálculo del Costo posible:

Acumulado de 1-5 : **20**

Más mínimo de 5-2, 5-3 y 5-4: **4**

Más mínimo de 4-1, 4-2 y 4-3: **7**

Más mínimo de 3-1, 3-2 y 3-4: **4**

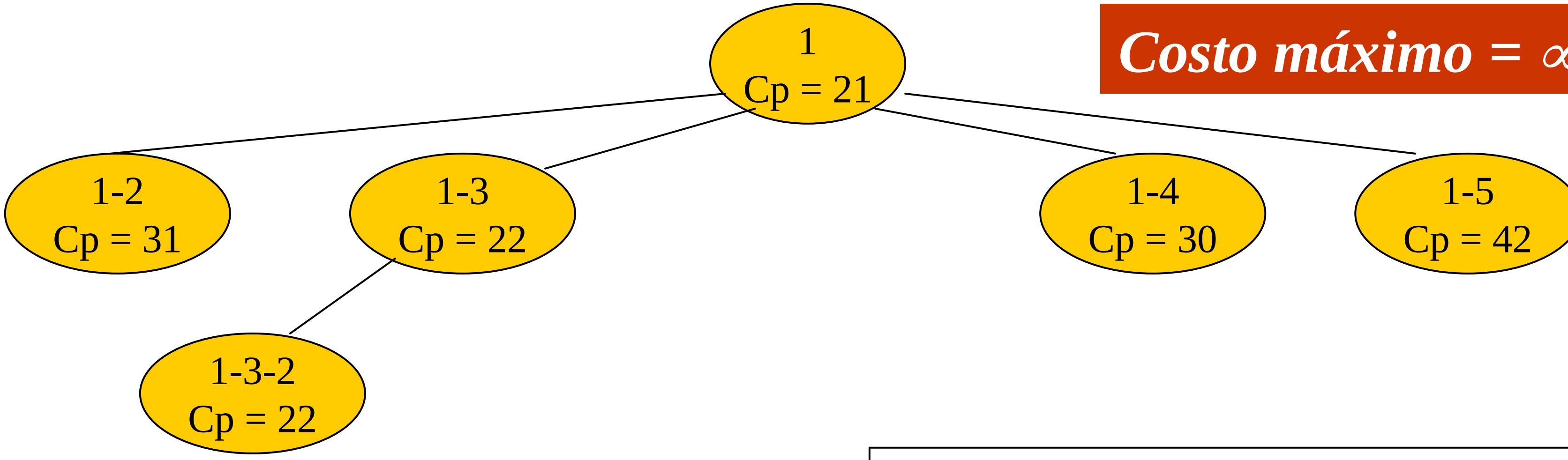
Más mínimo de 2-1, 2-3 y 2-4: **7**

TOTAL = 42

Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$

Costo máximo = ∞



<u>Cálculo del Costo posible:</u>

Acumulado de 1-3-2 : **9**

Más mínimo de 2-4 y 2-5: **7**

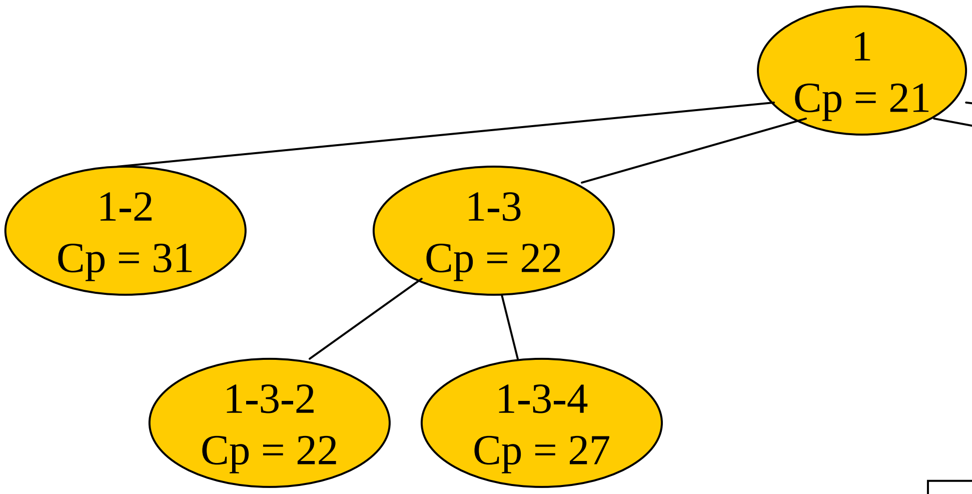
Más mínimo de 4-1 y 4-5: **2**

Más mínimo de 5-1 y 5-4: **4**

TOTAL = 22

Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$



Costo máximo = ∞

Cálculo del Costo posible:

Acumulado de 1-3-4 : **11**

Más mínimo de 4-2 y 4-5: **2**

Más mínimo de 2-1 y 2-5: **7**

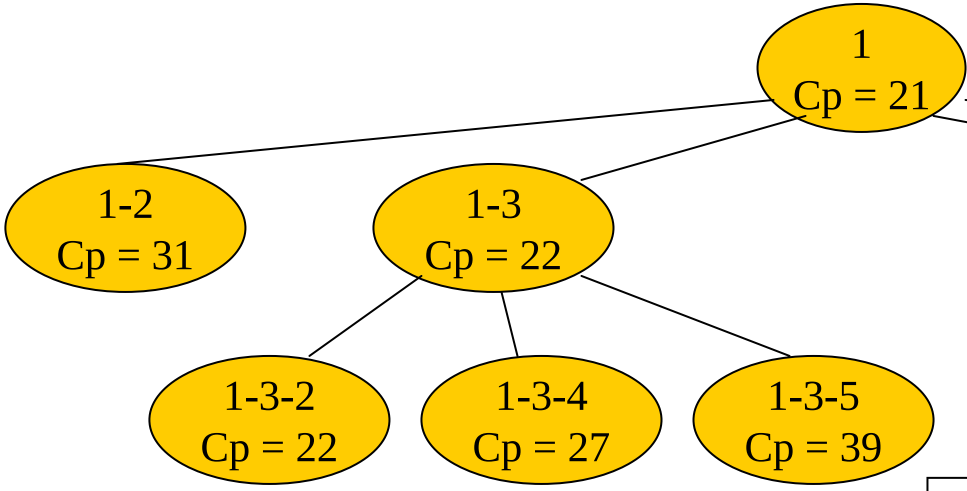
Más mínimo de 5-1 y 5-2: **7**

TOTAL = 27

Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0

Costo máximo = ∞



Cálculo del Costo posible:

Acumulado de 1-3-5 : **20**

Más mínimo de 5-2 y 5-4: **4**

Más mínimo de 2-1 y 2-4: **8**

Más mínimo de 4-1 y 4-2: **7**

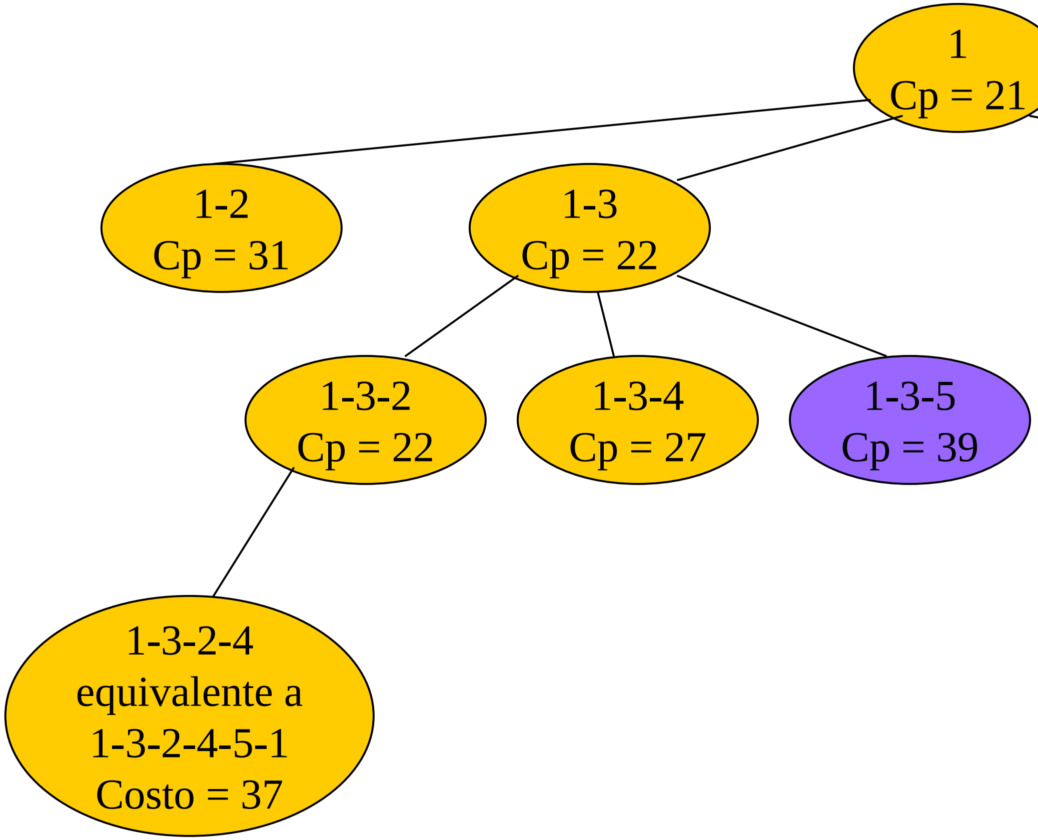
TOTAL = 39

¿Cuál es el mejor nodo para expandir?

Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$

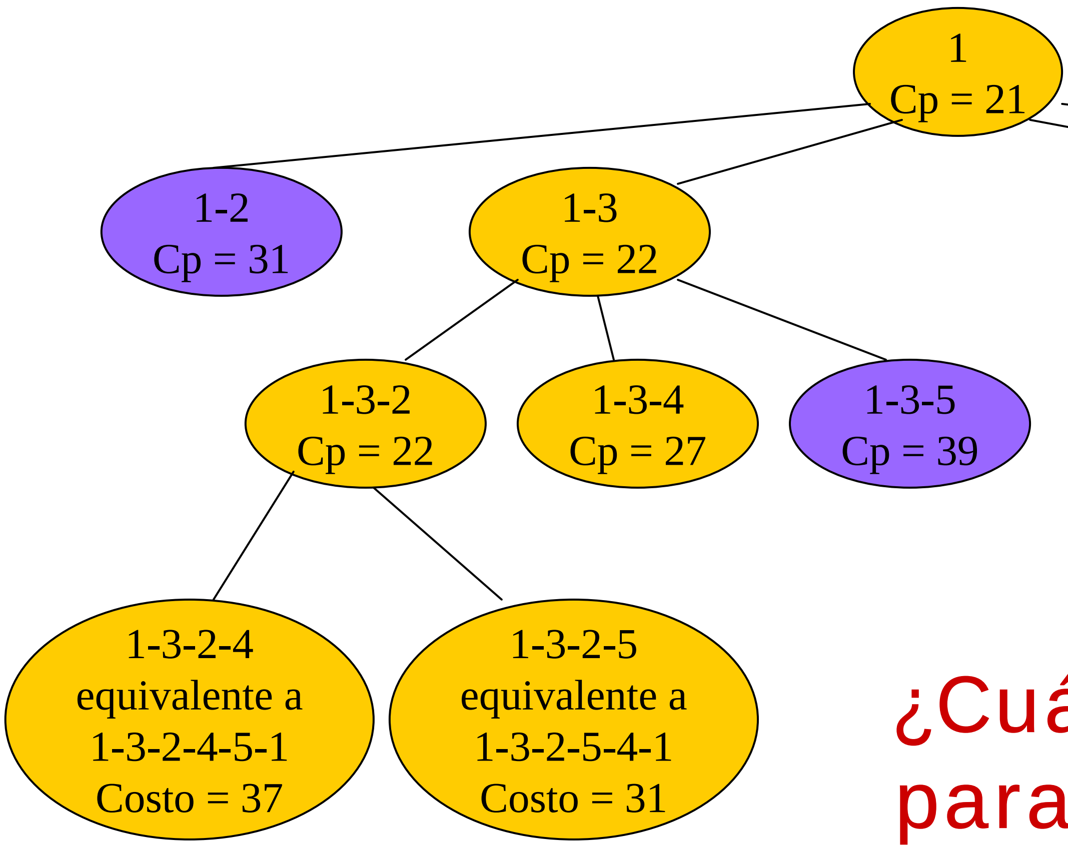
Costo máximo = 37



Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$

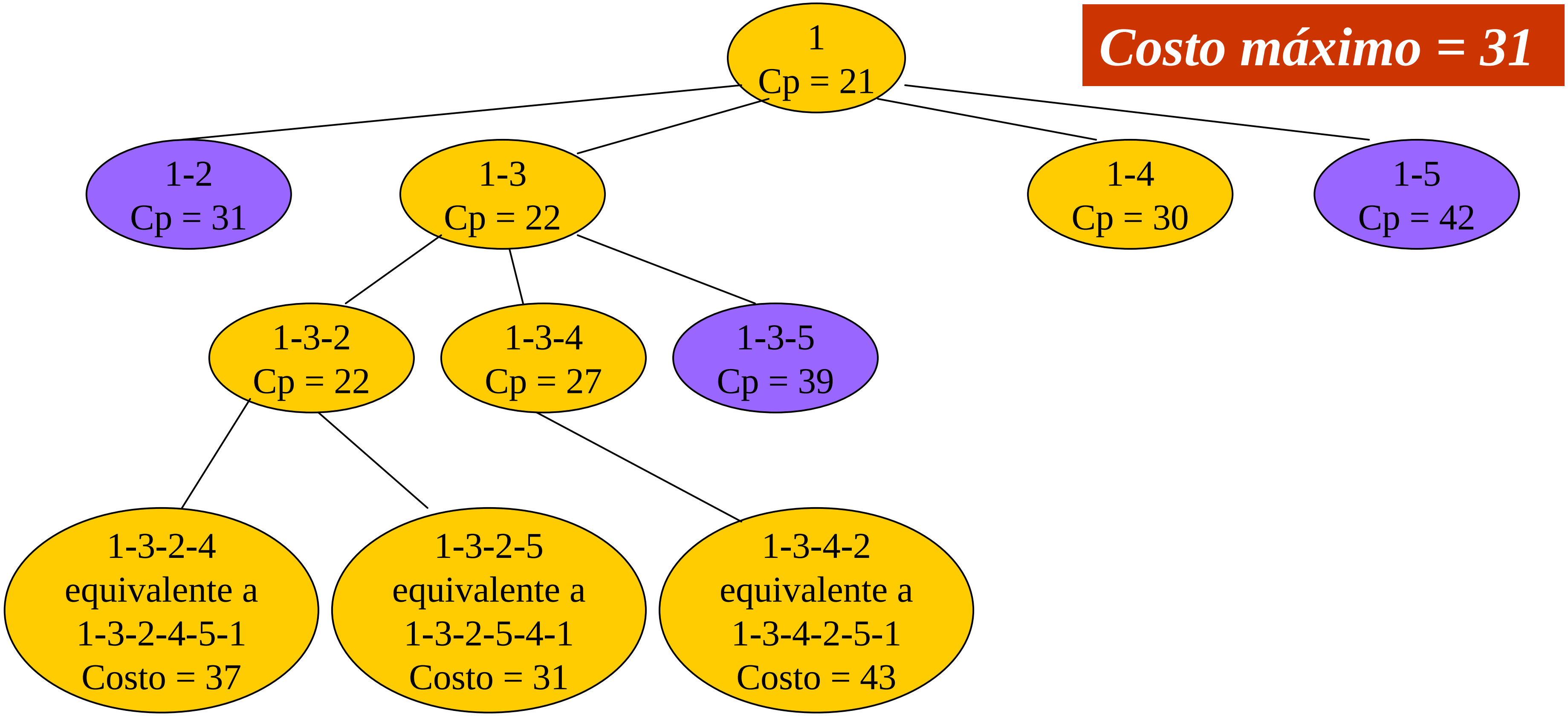
Costo máximo = 31



¿Cuál es el mejor nodo para expandir?

Ejemplo

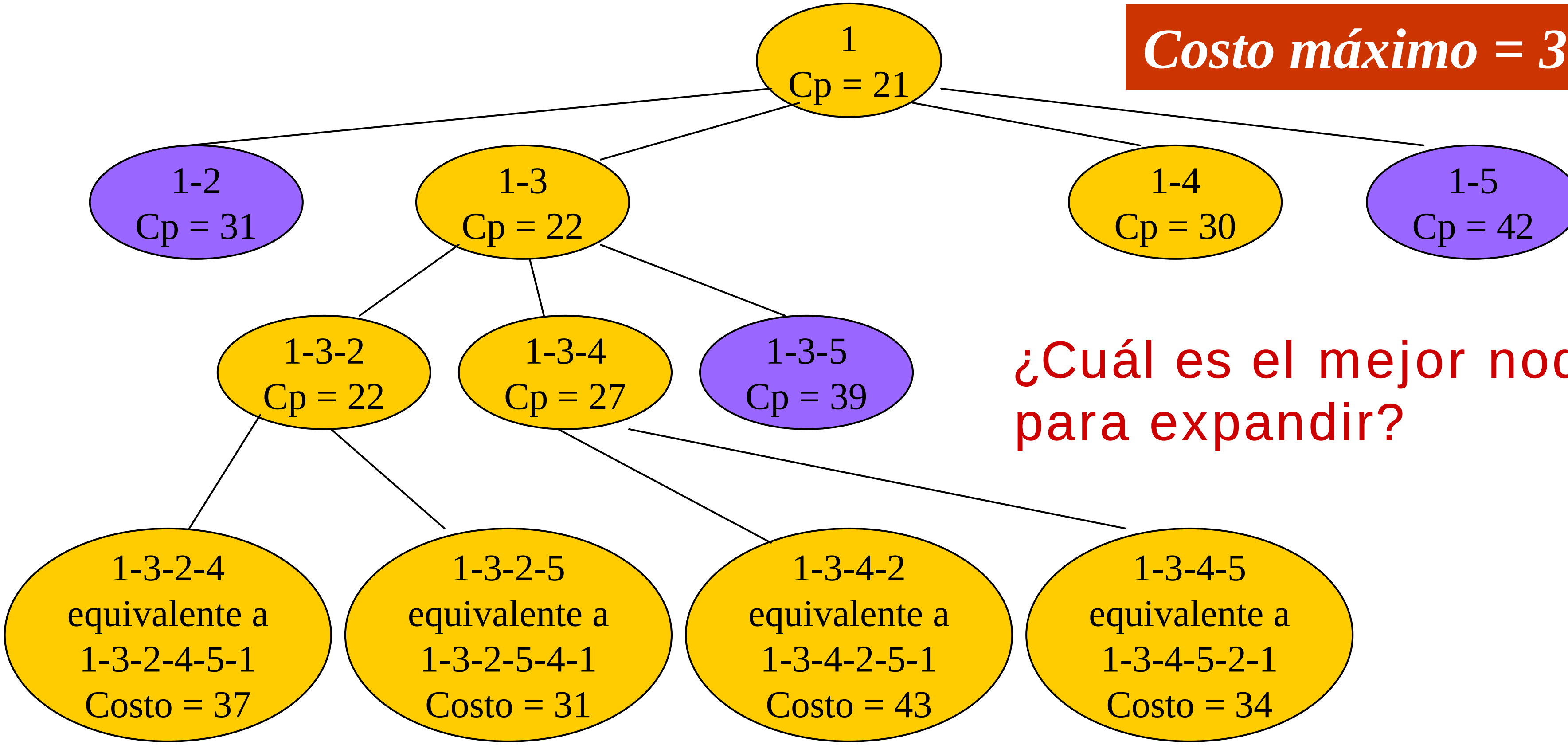
$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$



Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$

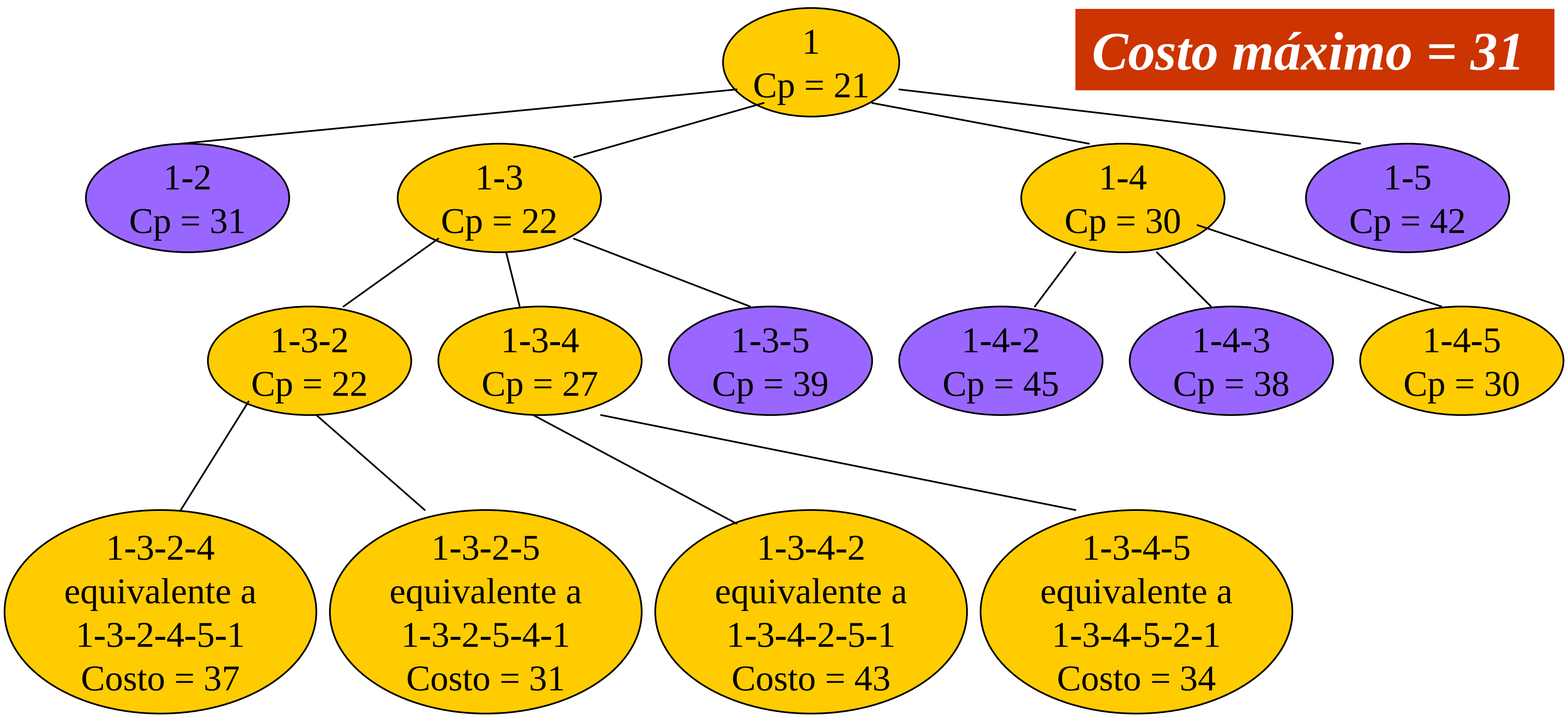
Costo máximo = 31



¿Cuál es el mejor nodo para expandir?

Ejemplo

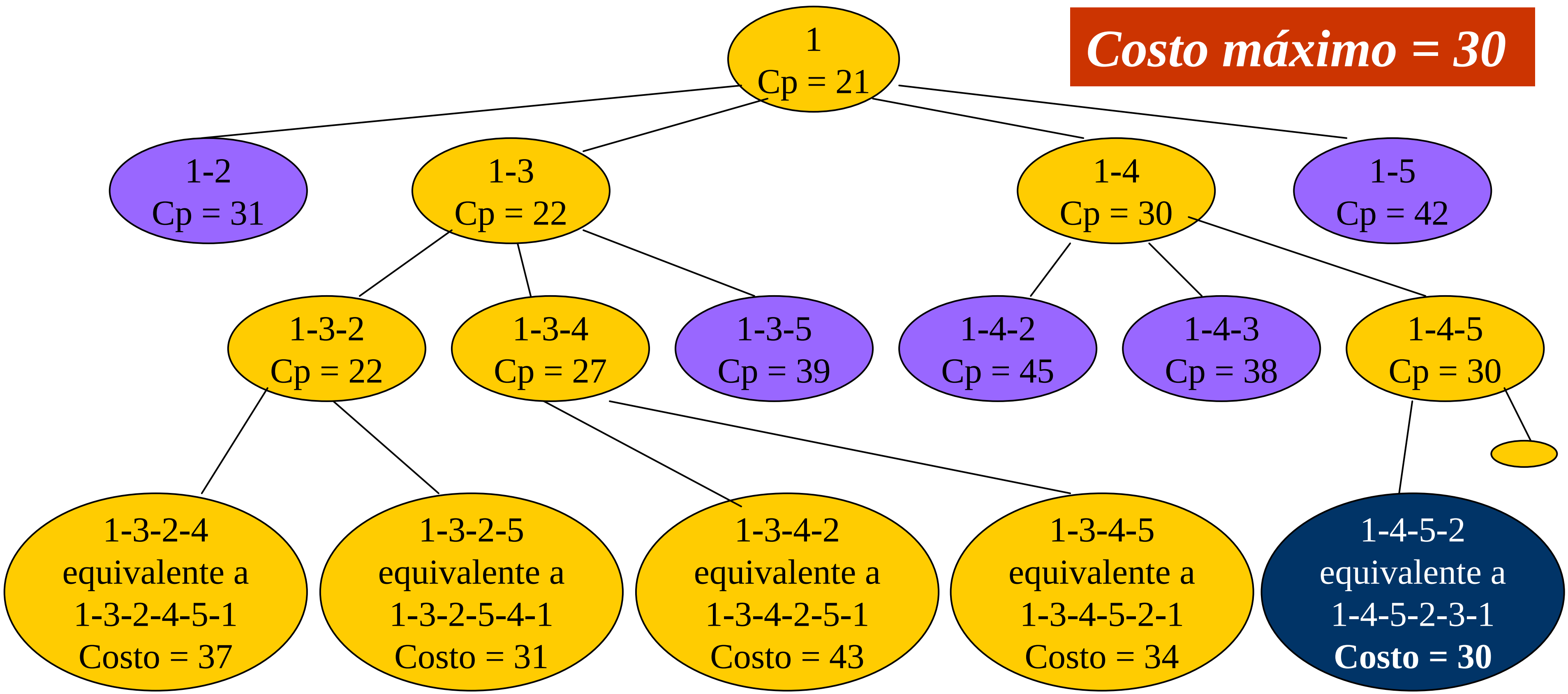
$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$



¿Cuál es el mejor nodo para expandir?

Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$



El Problema del Viajante de Comercio (Conclusión final)

- Branch and bound ofrece una opción más de solución del problema del Viajante de Comercio
- Sin embargo, NO asegura tener un buen comportamiento en cuanto a eficiencia, ya que en el peor caso tiene un tiempo exponencial...
- El problema puede ser resuelto con algoritmos heurísticos: SA, AG, TS, …

Ramificación y poda: **Conclusiones**

Ramificación y poda: mejora y generalización de la técnica de backtracking.

- **Idea básica.** Recorrido implícito en árbol de soluciones:
  - Distintas estrategias de ramificación.
  - Estrategias LC: explorar primero las ramas más prometedoras.
  - Poda basada en acotar el beneficio a partir de un nodo: CI, CS.
- **Estimación de cotas:** aspecto clave en RyP. Utilizar algoritmos de avance rápido.
- **Compromiso tiempo-exactitud.** Más tiempo → mejores cotas. Menos tiempo → menos poda.

Algorítmica