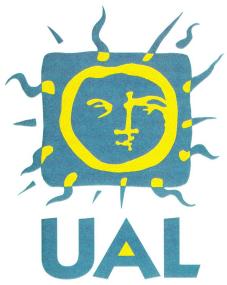
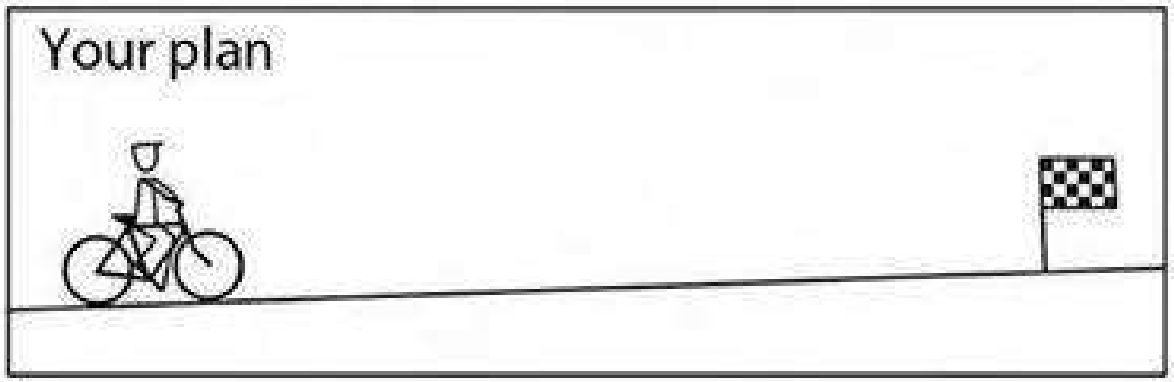
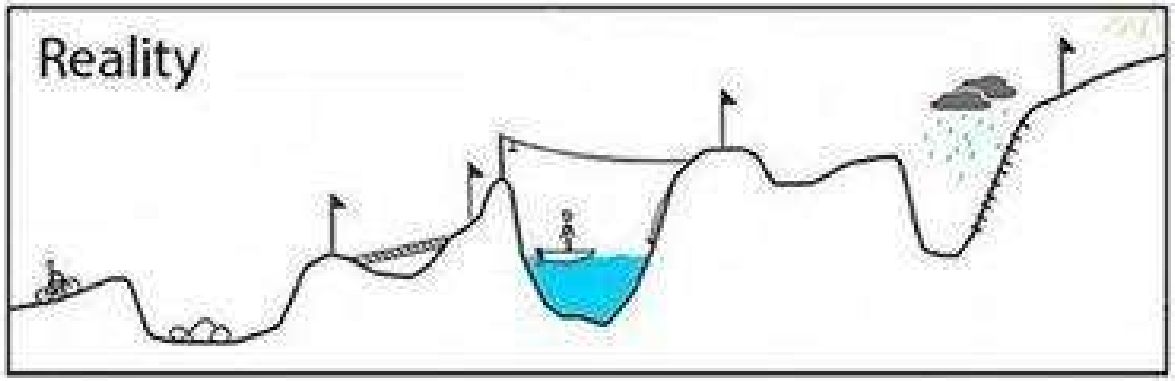


Tema 5 – Backtracking.

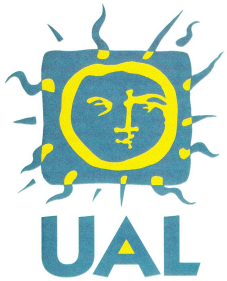
- *5.1. Introducción.*
- *5.2 Esquema general.*
- *5.3 Análisis de tiempos de ejecución.*
- *5.4 Ejemplos:*
  - *5.4.1 Búsqueda exhaustiva en grafos.*
  - *5.4.2 El problema de la Mochila 0/1.*
  - *5.4.3 El problema del viajante.*
  - *5.5.4 Planificación de tareas.*
  - *5.5.5 Otros ejemplos.*

1 – Introducción.





Algoritmos de retroceso
Corrección sobre el camino



1 – Introducción.

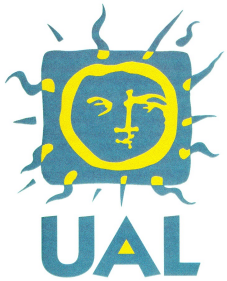
Caracterización de los problemas a resolver con backtracking:

Los problemas cuya resolución vamos a abordar, tienen unas características que permiten trabajar de la siguiente forma:

- Buscamos una solución, la mejor solución o el conjunto de todas las soluciones que satisfacen ciertas condiciones.
- Cada solución es el resultado de una secuencia de decisiones.
- Existe una función objetivo que debe ser satisfecha por cada selección, u optimizada si sólo queremos la mejor.

- Si se conoce un criterio óptimo de selección en cada decisión:

Técnica *greedy*



1 – Introducción.

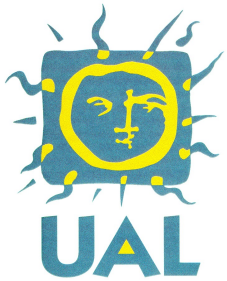
Caracterización de los problemas a resolver con backtracking:

Los problemas cuya resolución vamos a abordar, tienen unas características que permiten trabajar de la siguiente forma:

- Buscamos una solución, la mejor solución o el conjunto de todas las soluciones que satisfacen ciertas condiciones.
- Cada solución es el resultado de una secuencia de decisiones.
- Existe una función objetivo que debe ser satisfecha por cada selección, u optimizada si sólo queremos la mejor.

- Si se cumple principio de optimalidad de Bellman (relajado) y se pueden estructurar las soluciones parciales en una tabla aceptable:

Técnica *Programación Dinámica*



1 – Introducción.

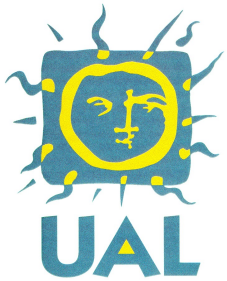
Caracterización de los problemas a resolver con backtracking:

Los problemas cuya resolución vamos a abordar, tienen unas características que permiten trabajar de la siguiente forma:

- Buscamos una solución, la mejor solución o el conjunto de todas las soluciones que satisfacen ciertas condiciones.
- Cada solución es el resultado de una secuencia de decisiones.
- Existe una función objetivo que debe ser satisfecha por cada selección, u optimizada si sólo queremos la mejor.

-Si no podemos resolver este tipo de problemas utilizando las técnicas anteriores (greedy o dynamic programming):

TENEMOS QUE BUSCAR SOLUCIONES Técnica *de Backtracking*



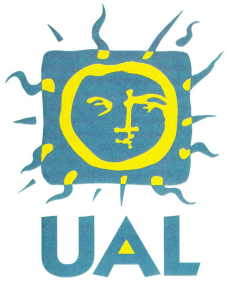
1 – Introducción.

Caracterización de los problemas a resolver con bactracking:

La solución de nuestro problema supone tomar una serie de decisiones, pero:

- ✓No disponemos de suficiente información como para saber cuál elegir a priori.
- ✓Cada decisión nos lleva a un nuevo conjunto de decisiones.
- ✓Alguna secuencia de decisiones (y puede que más de una) puede solucionar nuestro problema.
- ✓Puede haber secuencias de decisiones (a menudo muchas) que no son solución (nos llevan a caminos sin salida).

En esta situación, sería deseable poder explorar una secuencia de decisiones, y si vemos que no nos lleva a la solución, poder volver atrás y cambiarla.

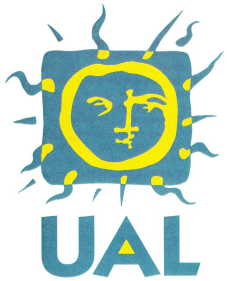


1 – Introducción.

Caracterización de la estrategia backtracking:

Algoritmos con *backtracking* (algoritmos con *vuelta atrás*, o con *retroceso*).

- Búsqueda de la/una solución, mediante una exploración sistemática de las posibles soluciones del problema.
- Se explora una secuencia de decisiones, y si vemos que no nos lleva a la solución, se vuelve atrás y cambiarla.
- No siguen unas reglas para la búsqueda de la solución, simplemente prueban todo lo posible, hasta encontrar la solución o encontrar que no existe solución al problema.
- La búsqueda se separa en varias búsquedas parciales o subtareas (secuencia de decisiones).
- Las subtareas suelen incluir más subtareas, por lo que el tratamiento general de estos algoritmos es de naturaleza recursiva.



1 – Introducción.

Caracterización formal de los problemas resolubles con backtracking:

- La solución se construye por etapas, es decir, puede expresarse en la forma de una n-tupla, $(x_1, x_2, x_3, \ldots, x_n)$, donde cada x_i pertenece a un conjunto finito S_i, y representa la decisión tomada en la etapa i-ésima, de entre un conjunto finito de alternativas.

- El problema se puede formular como la búsqueda de aquella n-tupla que optimiza (maximiza o minimiza), o simplemente satisface, un determinado criterio (predicado) $P(x_1, ..., x_n)$. Se podrían buscar también todas las n-tuplas que satisfacen el criterio.

Puede ser problemas de optimización, localización, juegos, ... Suelen ser problemas complejos, en los que la única forma de encontrar la solución es analizando (todas) las posibilidades → ritmo de crecimiento exponencial

1 – Introducción.

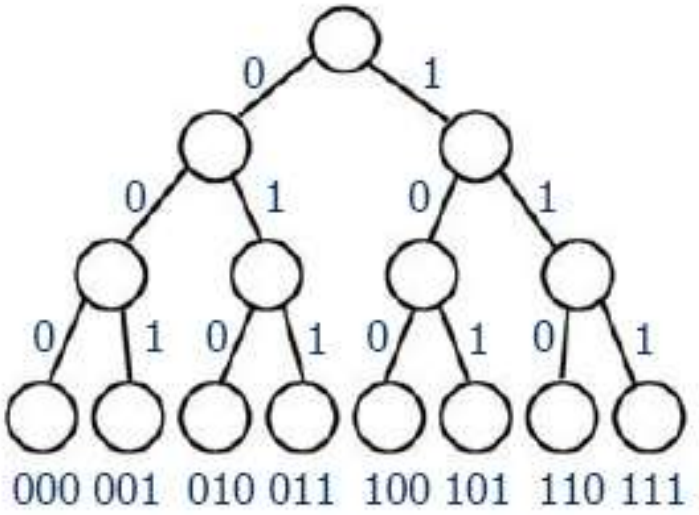
Resolución por fuerza bruta:

En principio, podemos solucionar nuestro problema probando todas las combinaciones $(x_1, x_2, x_3, \ldots, x_n)$, para ver cuál optimiza (o cumple) P.

Habrá un número de n-tuplas candidatas igual a: $\prod_{i=1}^{n} |S_i|$

Ejemplo: Generando todas las posibles combinaciones de n bits, podemos resolver el problema de la mochila 0/1 para n objetos.

$T(n) = 2\,T(n-1) + 1 \in O(2^n)$

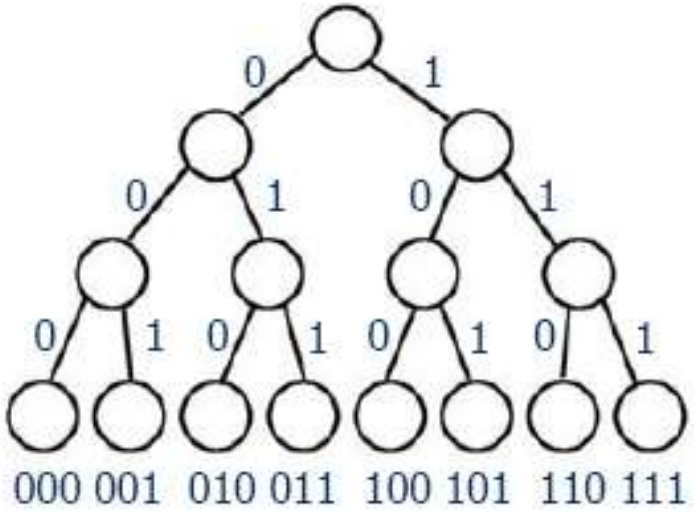


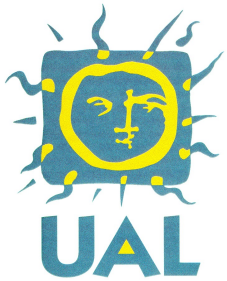
1 – Introducción.

Resolución por fuerza bruta:

Ejemplo: Generando todas las posibles combinaciones de n bits, podemos resolver el problema de la mochila 0/1 para n objetos.

```
método combinaciones_binarias (entero[] V, int pos)
  si (pos == V.size)
        procesa_combinación(V);
  sino
        V[pos]=0;
        combinaciones_binarias(V,pos+1)
        V[pos]=1
        combinaciones_binarias(V,pos+1)
  fin_si
fmétodo
```





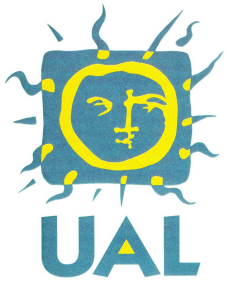
1 – Introducción.

Resolución por fuerza bruta: análisis de la propuesta

✓ Implícitamente, 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 del árbol, cuyas hojas corresponden a posibles soluciones del problema.

✓ Se exploran todas las opciones hasta encontrar la solución. Se analizan opciones que a veces ya sabemos no llevan a la solución.



1 – Introducción.

Resolución por fuerza bruta: análisis de la propuesta

Mejora posible:

- Eliminar la necesidad de llegar hasta todas las hojas del árbol.

- Si, para un nodo interno del árbol, podemos asegurar que no alcanzamos una solución (esto es, no nos lleva a nodos hoja útiles), entonces podemos obviar esa rama completa del árbol, y realizar una vuelta atrás (backtracking), para pasar a explorar otra rama.

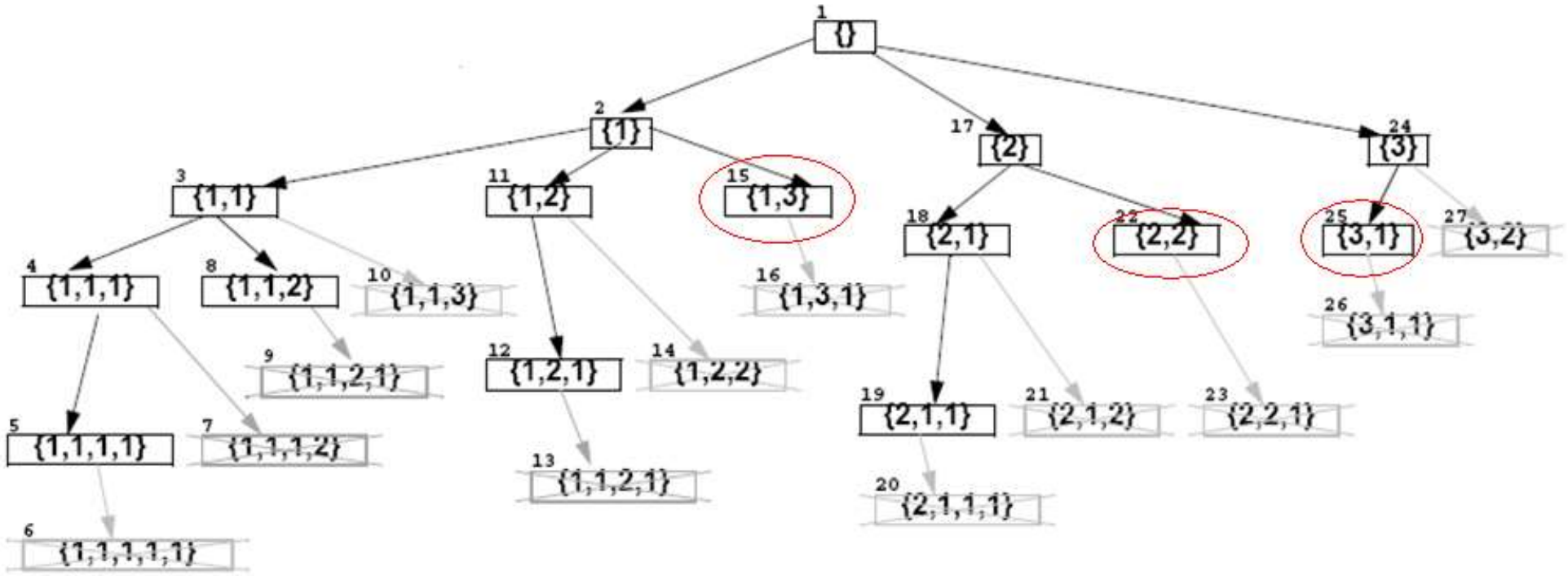
- La ventaja es que alcanzamos antes la solución.

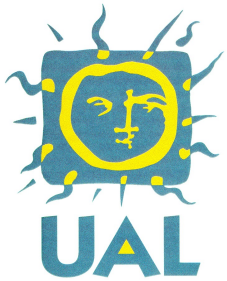
1 – Introducción.

Ejemplo (informal) de la mejora al añadir backtracking

Cambio de monedas:

Monedas: {v1 = 1, v2 = 2, v3 =3} Cambiar: 4





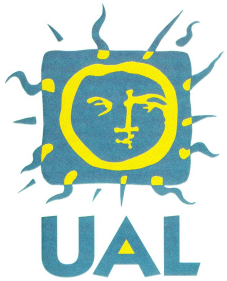
1 – Introducción.

Diferencias con otras técnicas:

- En los algoritmos *greedy*, se construye la solución aprovechando la posibilidad de calcularla a trozos: un candidato nunca se descarta una vez elegido. Con backtracking, sin embargo, la elección de un candidato en una etapa no es irrevocable. De hecho, se añaden y quitan elementos, hasta probar todas las combinaciones.

- El tipo de problemas en los que aplicaremos backtracking, no se puede descomponer en subproblemas independientes, por lo que la técnica *divide y vencerás* no resulta aplicable.

- En *programación dinámica*, cuando todos los subproblemas se resuelven directamente, el estudio directo de todos los casos es en realidad "Backtracking". En PD se almacenan los resultados para no tener que recalcular, se utiliza Backtracking cuando no se puede hacer.



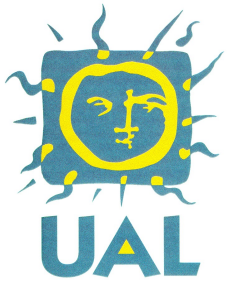
1 – Introducción.

Elementos propios de los algoritmos de backtracking:

Solución: $(x_1, x_2, x_3, \ldots, x_n)$, $x_i \in S_i$.

Espacio de soluciones de tamaño $\Pi |S_i|$. Los algoritmos con backtracking determinan las soluciones del problema buscando sistemáticamente en el espacio de soluciones del mismo. Esta búsqueda se puede representar en el *árbol de soluciones*, asociado al espacio de soluciones del problema.

Solución parcial: $(x_1, x_2, \ldots, x_k, ?, ?, ..., ?)$, $x_i \in S_i$. Vector solución para el que aún no se han establecido todas sus componentes. En el backtracking, se forma cada tupla de manera progresiva, elemento a elemento, comprobando, para cada elemento x_i añadido a la tupla, que $(x_1, \ldots, x_i)$ puede conducir a una tupla completa satisfactoria.



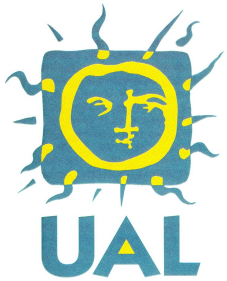
1 – Introducción.

Elementos propios de los algoritmos de backtracking:

Función de poda/acotación (también denominada función objetivo parcial, predicado acotador, o test de factibilidad): $P(x_1, \ldots, x_i)$. Función que nos permite identificar cuándo una solución parcial conduce o no a una solución del problema.

✓ Diferencia entre fuerza bruta y backtracking: Si se comprueba que $(x_1, \ldots, x_i)$ no puede conducir a ninguna solución, se evita formar las $|S_{i+1}| \cdot \ldots \cdot |S_n|$ tuplas que comienzan por $(x_1, \ldots, x_i)$.

En la función de acotación se utilizan las restricciones propias del problema.



1 – Introducción.

Elementos propios de los algoritmos de backtracking:

Restricciones asociadas a los problemas, para saber si una n-tupla es solución:

- Restricciones *explícitas*: Restringen los posibles valores de las variables x_i (todas las tuplas que las satisfacen, definen el espacio de soluciones del problema, de tamaño $\Pi\ |S_i|$).

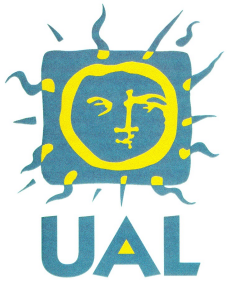
Ejemplos:

$x_i \geq 0 \Rightarrow S_i = \{\text{números reales no negativos}\}$

$x_i \in \{0,1\} \Rightarrow S_i = \{0, 1\}$

$l_i \leq x_i \leq u_i \Rightarrow S_i = \{a: l_i \leq a \leq u_i\}$

- Restricciones *implícitas*: Establecen relaciones entre las variables x_i (determinan las n-tuplas que satisfacen el criterio $P(x_1, ..., x_n)$, y nos indican cuándo una solución parcial nos puede llevar a una solución).



1 – Introducción.

Terminología relativa al espacio de soluciones:

✓ El espacio de soluciones está formado por el conjunto de tuplas que satisfacen las restricciones explícitas, y se puede estructurar como un *árbol de exploración*: en cada nivel se toma la decisión de la etapa correspondiente.

✓ *Nodo estado* (o *nodo problema*): Cada uno de los nodos del árbol de exploración. El camino desde la raíz hasta el nodo, representa una tupla parcial o completa, que satisface las restricciones explícitas.

✓ *Nodo solución*: Nodo estado para el cual, el camino desde la raíz hasta el nodo, representa una tupla completa que satisface las restricciones implícitas del problema.

1 – Introducción.

Terminología relativa a la 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* (o *nodo fracaso*): Estado del problema que ya ha sido generado y, o bien se ha podado, o bien ya se han generado todos sus descendientes.

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

- *Nodo prometedor*: Nodo que no es solución pero desde el que todavía podría ser posible llegar a la solución.

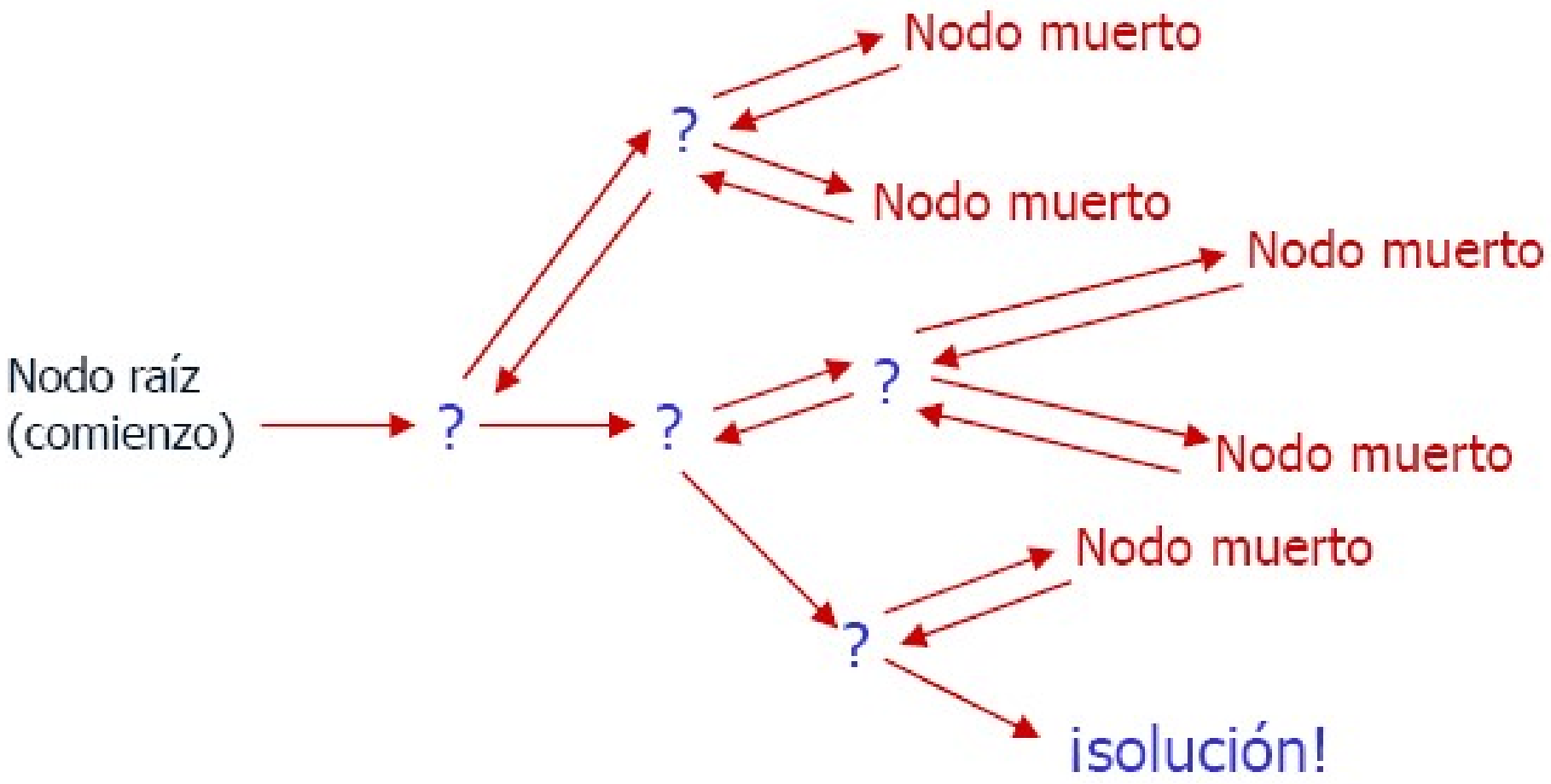
1 – Introducción.

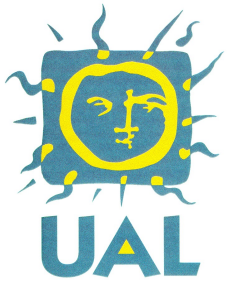
Mecanismo de generación de estados:

- Para generar todos los estados de un problema, comenzamos con un nodo raíz, a partir del cual se generan otros nodos.
- Al ir generando estados del problema, mantenemos una lista de nodos vivos.
- Se usan funciones de acotación para "matar" nodos vivos sin tener que generar todos sus nodos hijos.
- Existen distintas formas de generar los estados de un problema, en función de cómo exploremos el árbol de estados. En la técnica de backtracking, esta exploración se realiza en *profundidad*:
  - Tan pronto como un nuevo hijo (N) del E-nodo en curso (A) ha sido generado, este hijo se convierte en un nuevo E-nodo.
  - A se convertirá de nuevo en E-nodo cuando el subárbol N haya sido explorado por completo.

1 – Introducción.

Mecanismo de generación de estados:





1 – Introducción.

Resumen características algoritmos de backtracking:

- Constituyen una forma sistemática de recorrer todo el espacio de soluciones.

- Dependiendo de si buscamos una solución cualquiera, la óptima, o todas las posibles, el algoritmo se detiene una vez encontrada la primera solución, o continúa buscando el resto de soluciones.

- Estos algoritmos no crean ni gestionan el árbol explícitamente; se crea implícitamente con las llamadas *recursivas* al algoritmo.

2 – Esquema general.

Elementos básicos del esquema:

Una solución se puede expresar como una n-tupla $(x_1, \ldots, x_n)$, que satisface unas restricciones, y tal vez optimiza una función objetivo.

En cada momento, el algoritmo se encontrará en cierto nivel k, con una solución parcial $(x_1, \ldots, x_k)$. En ese punto:

- Si se puede añadir un nuevo elemento x_{k+1} a la solución, se genera la nueva solución y se avanza al nivel k+1.

- Si no, se prueban otros valores para x_k.

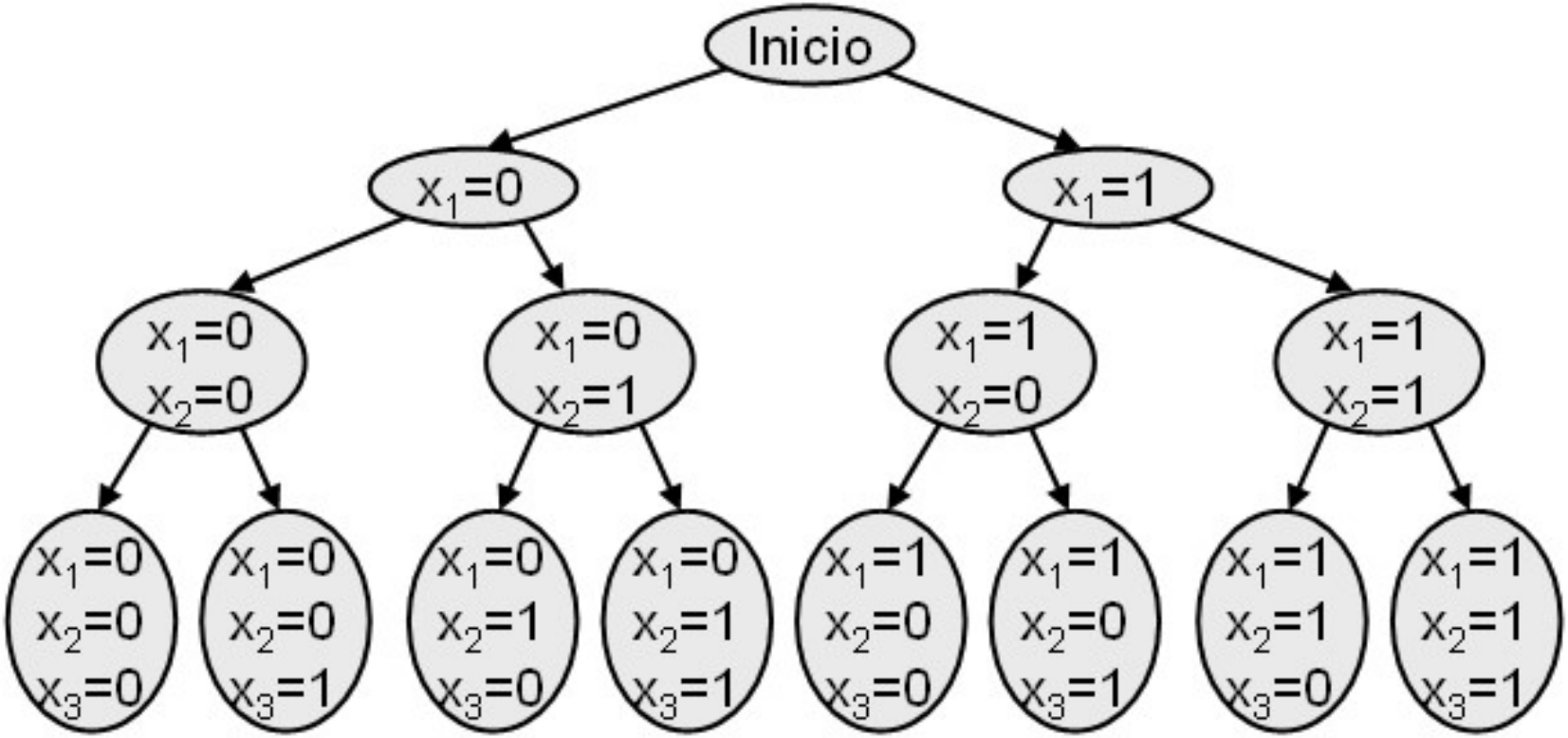
- Si no existe ningún valor posible por probar, entonces se retrocede al nivel anterior, k-1.

- Se sigue hasta que la solución parcial sea una solución completa del problema, o hasta que no queden más posibilidades por probar.

2 – Esquema general.

Elementos básicos del esquema:

El resultado es equivalente a hacer un recorrido en profundidad en el árbol de soluciones:



2 – Esquema 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.

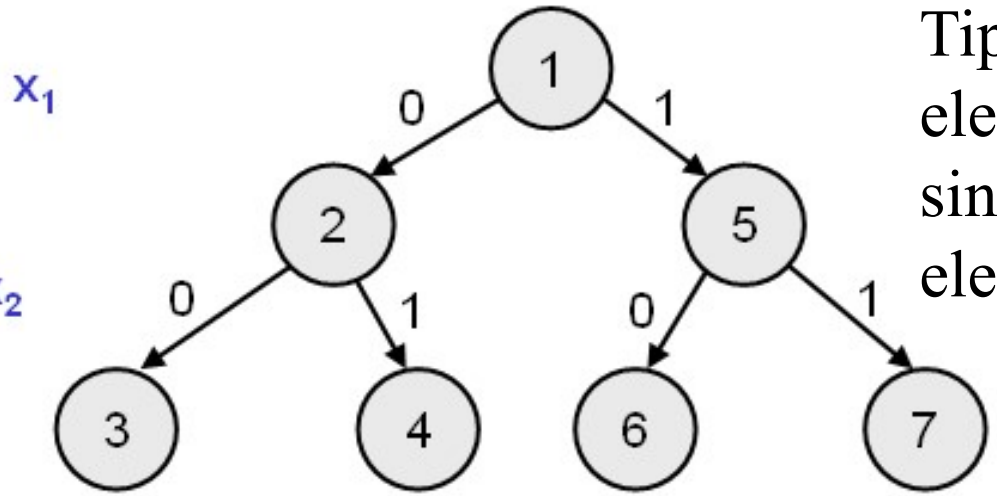
✓ La primera decisión para aplicar backtracking: ¿cómo es la forma del árbol?

✓Preguntas relacionadas: ¿Cómo es la representación de la solución al problema? ¿Qué significa cada valor de la tupla solución $(x_1, \ldots, x_n)$?

2 – Esquema general.

Tipos comunes de árboles de backtracking: binario

$s = (x_1, x_2, \ldots, x_n)$, con $x_i \in \{0, 1\}$



Tipo de problemas: Elegir ciertos elementos de entre un conjunto, sin importar el orden de los elementos.

Ejemplos:

- Problema de la mochila 0/1.
- Elegir un subconjunto de elementos de entre una colección que cumpla una condición.

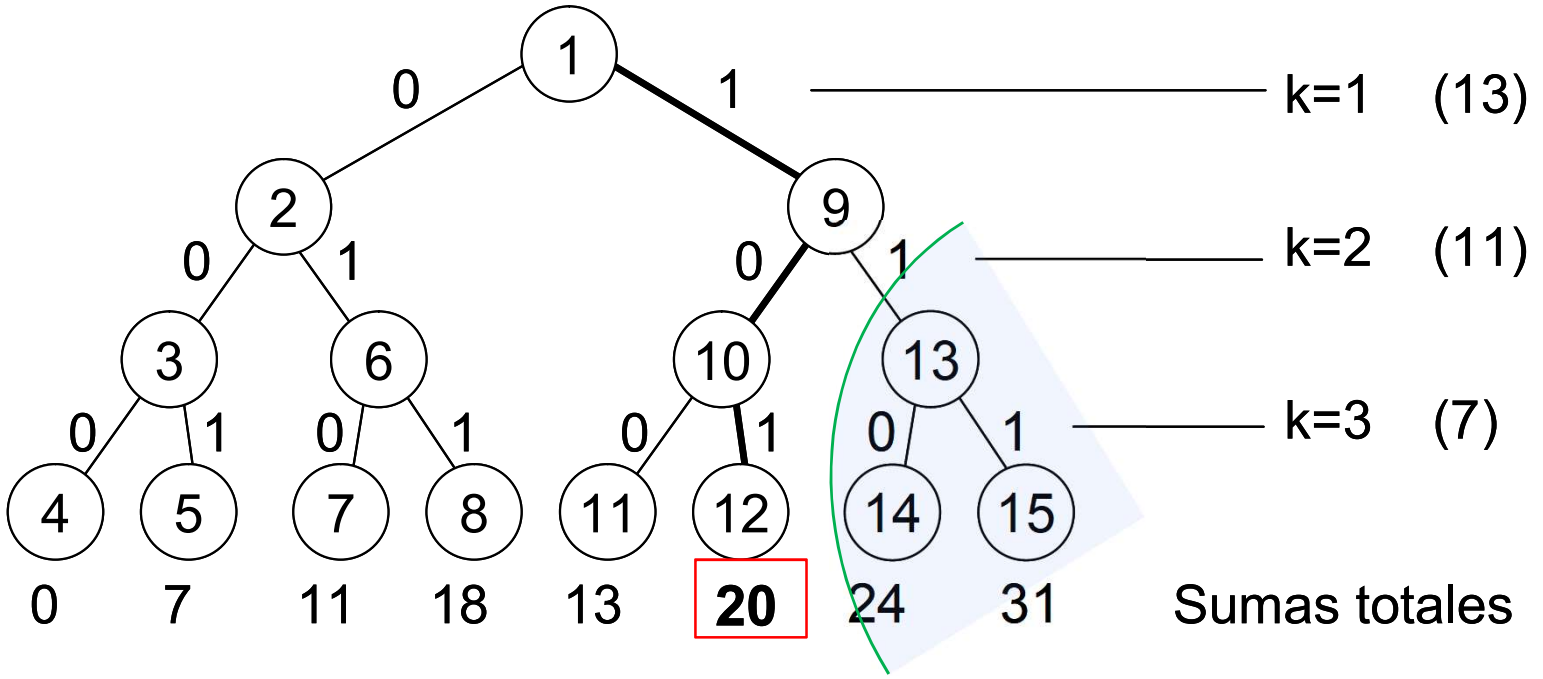
Encontrar un subconjunto de {13, 11, 7} que sume exactamente 20.

2 – Esquema general.

Tipos comunes de árboles de backtracking: binario, ejemplo

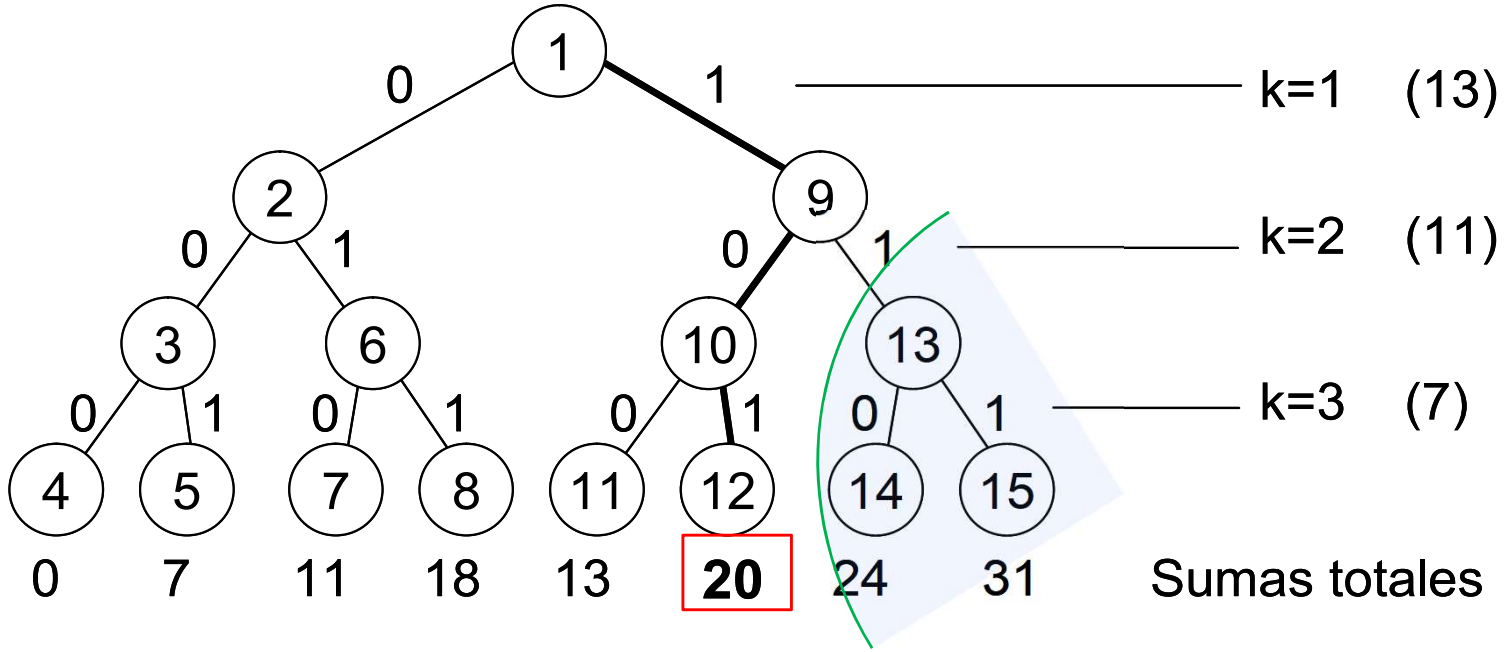
Ejemplos: Encontrar un subconjunto de {13, 11, 7} que sume exactamente 20.

Si se usa un árbol binario, en cada nivel i se decide si el elemento i está o no en la solución. Representación de la solución: (x_1, x_2, x_3), donde $x_i = \{0, 1\}$.



2 – Esquema general.

Tipos comunes de árboles de backtracking: binario, ejemplo

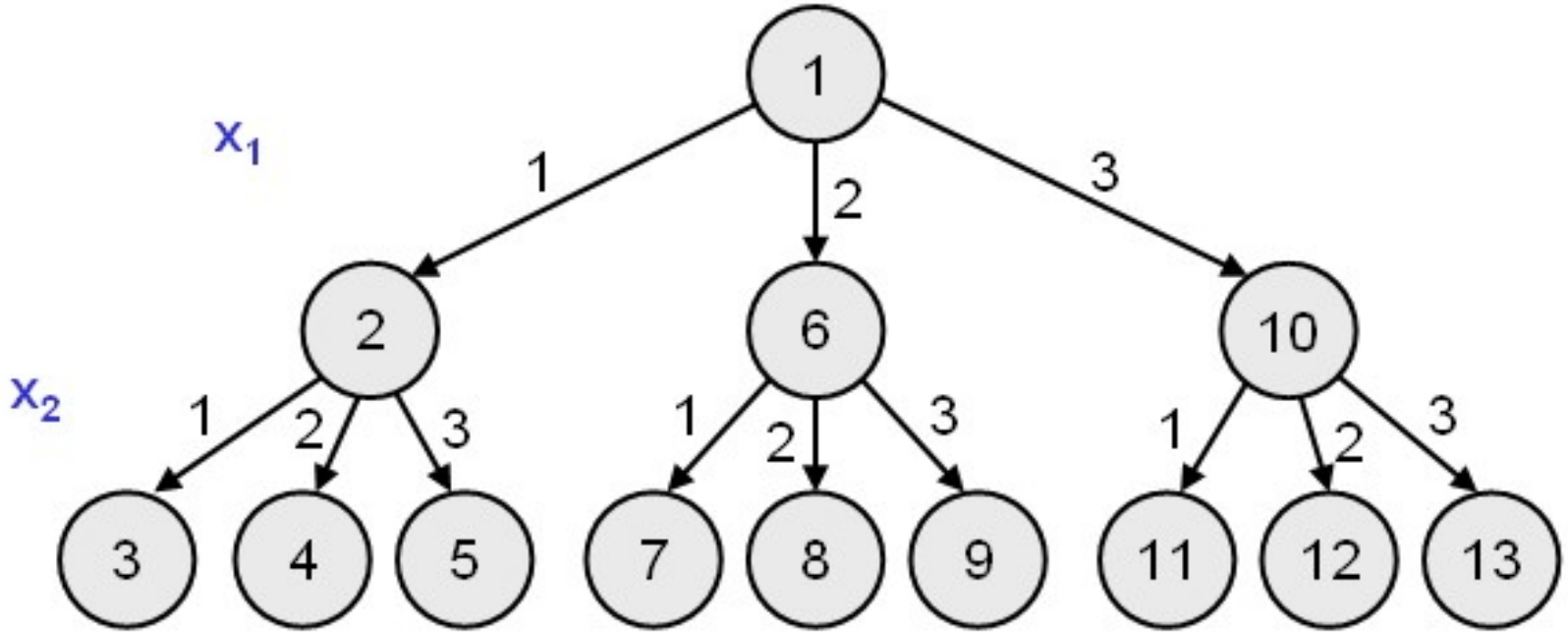


- Cada nodo representa un paso del algoritmo, una solución parcial en cada momento dado.
- El árbol indica un orden de ejecución (recorrido en profundidad) pero no se almacena en ningún lugar.
- Una solución es un nodo hoja con valor de suma 20.
- Mejora: En cada nodo llevamos el valor de la suma hasta ese punto. Si el valor es mayor que 20: retroceder al nivel anterior.

2 – Esquema general.

Tipos comunes de árboles de backtracking: árboles *k-arios*

$s = (x_1, x_2, \ldots, x_n)$, con $x_i \in \{1, ..., k\}$



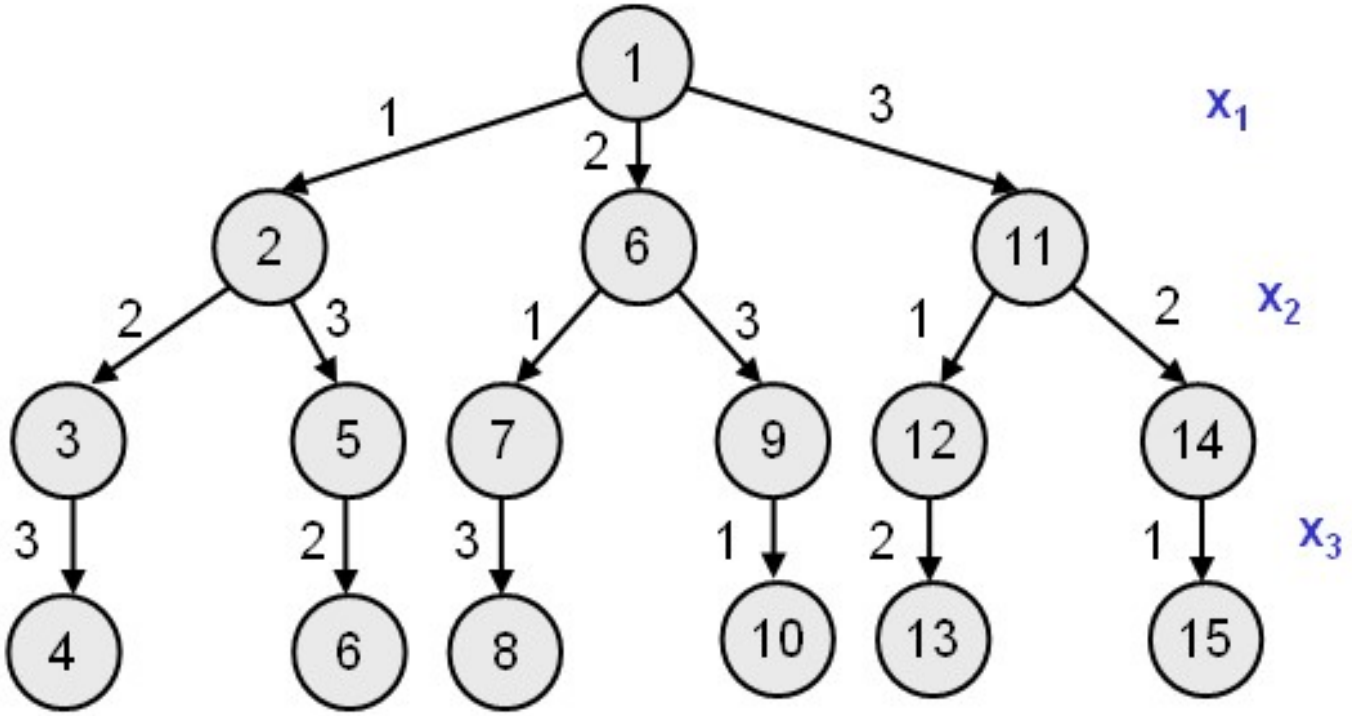
Tipo de problemas: varias opciones para cada x_i.

+ Problema del cambio de monedas.
+ Problema de las n reinas.

2 – Esquema general.

Tipos comunes de árboles de backtracking: permutacionales

$s = (x_1, x_2, \ldots, x_n)$, con $x_i \in \{1, ..., n\}$ y $x_i \neq x_j$



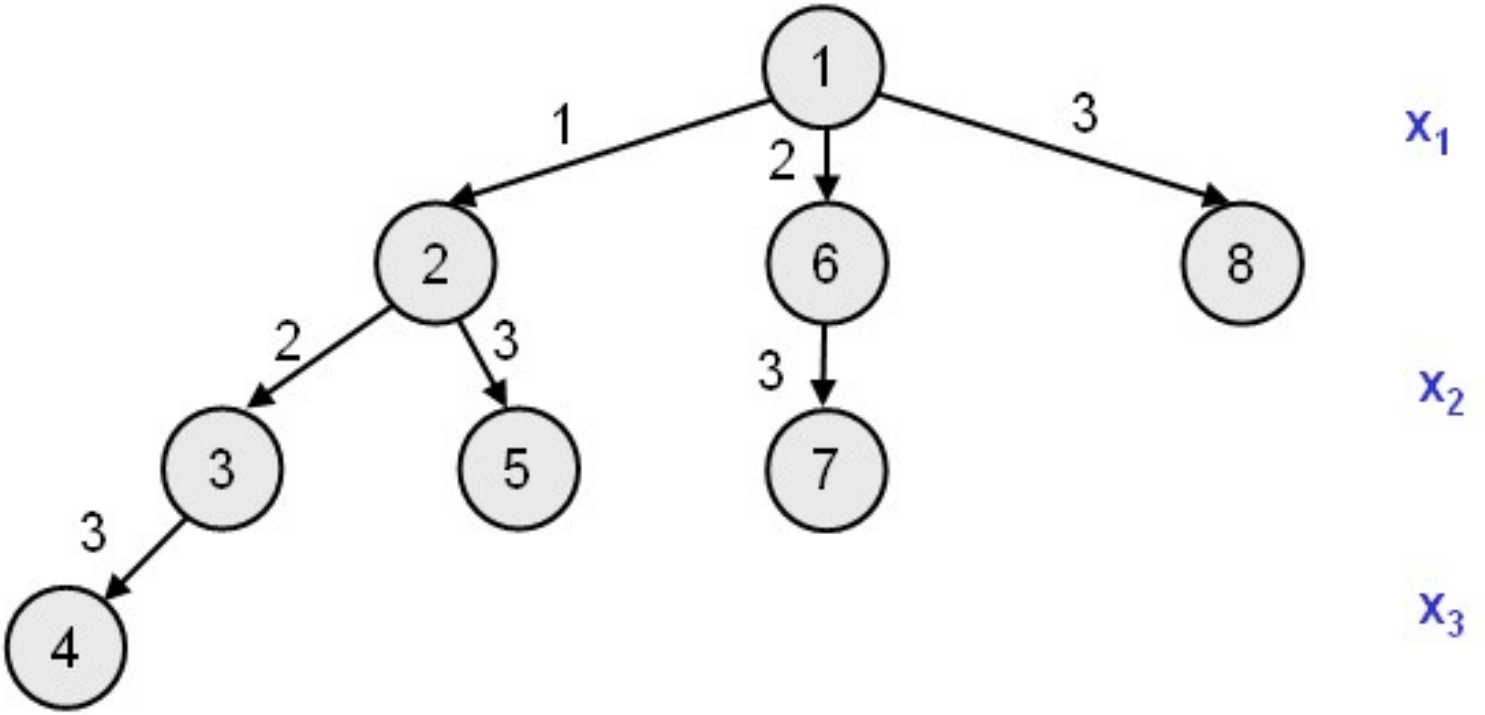
Tipo de problemas: Los x_i no se pueden repetir.

- Generar todas las permutaciones de (1, ..., n).
- Asignar n trabajos a n personas, asignación uno-a-uno.

2 – Esquema general.

Tipos comunes de árboles de backtracking: combinatorios

$s = (x_1, x_2, \ldots, x_m)$, con $m \leq n$, $x_i \in \{1, ..., n\}$ y $x_i < x_{i+1}$



Tipo de problemas: Los mismos que con árboles binarios.

Binario: (0, 1, 0, 1, 0, 0, 1) → Combinatorio: (2, 4, 7)

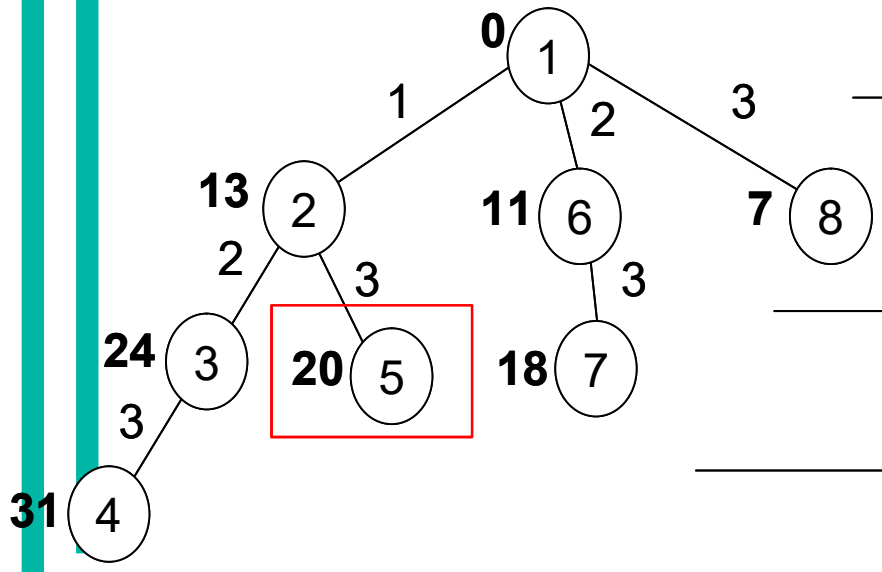
2 – Esquema general.

Tipos comunes de árboles de backtracking: combinatorios

$s = (x_1, x_2, \ldots, x_m)$, con $m \leq n$, $x_i \in \{1, ..., n\}$ y $x_i < x_{i+1}$

Ejemplo de correspondencia binario - combinatorio

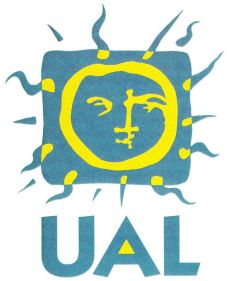
Subconjunto del conjunto {13, 11, 7} que sume exactamente 20.



- En cada nivel i se decide qué elemento se añade (1, 2 ó 3) de los restantes.
- Representación de la solución: $(s_1, ..., s_m)$, con $m \leq n$ y $s_i \in \{1, 2, 3\}$.

Cada nodo es una posible solución. Será válida si la suma es 20.

El recorrido es también en profundidad.



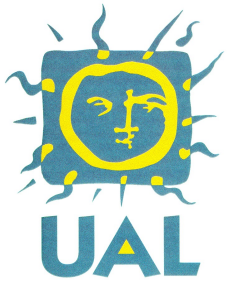
2 – Esquema general.

Cuestiones a resolver antes de desarrollar solución backtracking

- ¿Qué tipo de árbol es adecuado para el problema?
  - ✓¿Cómo es la representación de la solución?
  - ✓¿Cómo es la tupla solución? ¿Qué indica cada 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.

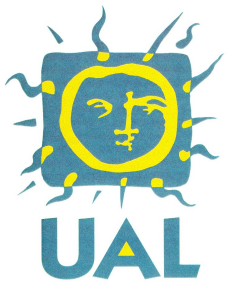
- ¿Qué ramas se pueden descartar por no conducir a soluciones del problema?
  - ✓Poda por restricciones del problema.
  - ✓Poda según el criterio de la función objetivo.



2 – Esquema general.

Pasos en la construcción del algoritmo con backtracking

- Organizar el espacio de soluciones posibles en forma de árbol, para facilitar la búsqueda.
- Fijar la descomposición en etapas.
- Recorrer el árbol:
  - Nodo solución → Identificar nodos que puedan dar lugar a soluciones.
  - Nodo problema → Nodos que pueden dar lugar a fragmentos de solución.
  - Nodo fracaso → Nodos cuyos descendientes no van a ser solución.
- Al encontrar un nodo fracaso, retroceder a su antecesor.
- Si un nodo problema, al explorar, se ve que es fracaso, retroceder.
- Si un posible nodo solución se comprueba que no lo es, retroceder.

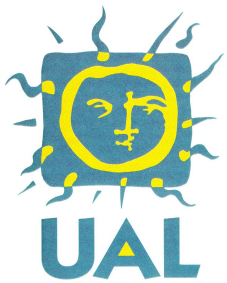


2 – Esquema general.

Esquema general (no recursivo) de solución con backtracking

```
Método Backtracking (..): Tuplasolucion
        nivel = 1
        s = s.Inicial
        fin = false
        repetir
            Generar (nivel, s)
            si Solución (nivel, s) entonces
                        fin = true                    // Encontrada solución
            sino si Criterio (nivel, s) entonces
                        nivel = nivel + 1
            sino mientras NO MasHermanos(nivel, s) hacer
                        Retroceder (nivel, s)
            fsi
        hasta fin
        retornar s
```

Problema de satisfacción de restricciones: buscamos cualquier solución que cumpla cierta propiedad (Solución), y se supone que existe alguna.



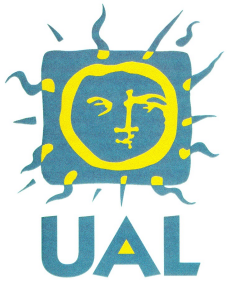
2 – Esquema general.

Esquema general (no recursivo) de solución con backtracking

Problema de satisfacción de restricciones: buscamos cualquier solución que cumpla cierta propiedad (Solución), y se supone que existe alguna.

Variables:

- ✓ s: Almacena la solución parcial hasta cierto punto.
- ✓ sINICIAL: Valor de inicialización.
- ✓ nivel: Indica el nivel actual en el que se encuentra el algoritmo.
- ✓ fin: Valdrá true cuando hayamos encontrado alguna solución.
- ✓ valorActual: variables temporales con el valor actual (beneficio, peso, etc.) de la tupla solución.



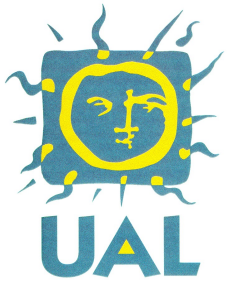
2 – Esquema general.

Esquema general (no recursivo) de solución con backtracking

Problema de satisfacción de restricciones: buscamos cualquier solución que cumpla cierta propiedad (Solución), y se supone que existe alguna.

Métodos:

1. Generar (nivel, s): Genera el siguiente hermano, o el primero, para el nivel actual. Devuelve el siguiente valor a añadir a la solución parcial actual (depende de la solución parcial y del nivel).
2. Solución (nivel, s): Comprueba si la tupla (s[1], ..., s[nivel]) es una solución válida para el problema.
3. 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).



2 – Esquema general.

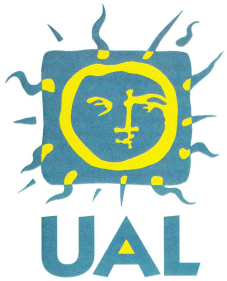
Esquema general (no recursivo) de solución con backtracking

Problema de satisfacción de restricciones: buscamos cualquier solución que cumpla cierta propiedad (Solución), y se supone que existe alguna.

Métodos:

4. MasHermanos (nivel, s): Devuelve true si hay más hermanos del nodo actual que todavía no han sido generados.

5. 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.



2 – Esquema general.

Esquema general (no recursivo) de solución con backtracking

Problema de satisfacción de restricciones: buscamos cualquier solución que cumpla cierta propiedad (Solución), y se supone que existe alguna.

Ejemplo: subconjunto del conjunto T= $\{t_1, t_2, ..., t_n\}$ que sume exactamente P (sabemos que hay una solución).

- Árbol: binario

- Variables:

  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)
  sINICIAL: (-1, -1, ..., -1)
  fin: Valdrá true cuando se haya encontrado solución.
  tact: Suma acumulada hasta ahora (inicialmente 0).

2 – Esquema general.

Esquema general (no recursivo) de solución con backtracking

Ejemplo: subconjunto del conjunto T= $\{t_1, t_2, ..., t_n\}$ que sume exactamente P (sabemos que hay una solución).

- Algoritmo: El esquema anterior.

- Métodos:

  + Generar (nivel, s)
    s[nivel] = s[nivel] + 1
    si s[nivel]=1 entonces tact = tact + tnivel

  + Solución (nivel, s)
    devolver (nivel=n) Y (tact=P)

  + Criterio (nivel, s)
    devolver (nivel < n) Y (tact ≤ P)

2 – Esquema general.

Esquema general (no recursivo) de solución con backtracking

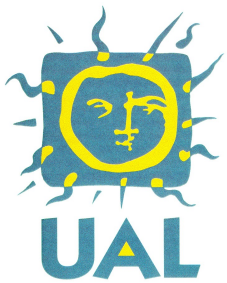
Ejemplo: subconjunto del conjunto T= $\{t_1, t_2, ..., t_n\}$ que sume exactamente P (sabemos que hay una solución).

- Algoritmo: El esquema anterior.

- Métodos:

    + MasHermanos (nivel, s)
        devolver s[nivel] < 1

    + Retroceder (nivel, s)
        tact = tact – tnivel*s[nivel]
        s[nivel] = -1
        nivel = nivel – 1



2 – Esquema general.

Variaciones del esquema general (no recursivo) de solución con backtracking: posibilidad de no solución

```
Método Backtracking_pns (..): Tuplasolucion
        nivel = 1
        s = s.Inicial
        fin = false
        repetir
            Generar (nivel, s)
            si Solución (nivel, s) entonces
                        fin = true                    // Encontrada solución
            sino si Criterio (nivel, s) entonces
                        nivel = nivel + 1
            sino mientras NO MasHermanos(nivel, s) Y (nivel>0)  hacer
                        Retroceder (nivel, s)
            fsi
        hasta fin OR (nivel=0)
        retornar s
```

2 – Esquema general.

Variaciones del esquema general (no recursivo) de solución con backtracking: se desean todas las soluciones

```
Método Backtracking_ts (..): Tuplasolucion[]
        nivel = 1;  s = s.Inicial;   fin = false
        repetir
            Generar (nivel, s)
            si Solución (nivel, s) entonces
                        Almacenar (nivel, s)     // Encontrada una solución
            fin_si                    //  A veces la solución es intermedia
            si Criterio (nivel, s) entonces        // A veces no hay que seguir
                        nivel = nivel + 1
            sino mientras NO MasHermanos(nivel, s) Y (nivel>0)  hacer
                        Retroceder (nivel, s)
            fsi
        hasta (nivel=0)
        retornar soluciones                  // Las soluciones almacenadas
fmétodo
```

2 – Esquema general.

Variaciones del esquema general (no recursivo) de solución con backtracking: optimización – se desea optimizar una función -

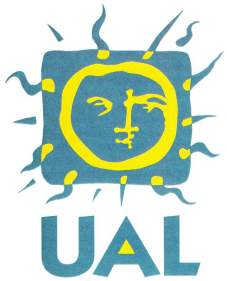
```
Método Backtracking_so (..): Tuplasolucion
        nivel = 1;  s = s.Inicial;   fin = false
        voa = -∞; soa = Ø          // voa: valor óptimo actual;
        repetir                    // soa: solución óptima actual
            Generar (nivel, s)
            si Solución (nivel, s) Y Valor(s) > voa  entonces
                        voa = Valor(s); soa = s
            fin_si                 //  A veces la solución es intermedia
            si Criterio (nivel, s) entonces        // A veces no hay que seguir
                        nivel = nivel + 1
            sino mientras NO MasHermanos(nivel, s) Y (nivel>0)  hacer
                        Retroceder (nivel, s)
            fsi
        hasta (nivel=0)
        retornar soa               // La mejor solución almacenada
```

2 – Esquema general.

Variaciones del esquema general (no recursivo) de solución con backtracking: optimización – se desea optimizar una función -

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:
    - > Opción 1:
        - +Valor(s)
            - devolver s[1] + s[2] + ... + s[n]
        - + Todas las demás no cambian.
    - > Opción 2: Incluir variable, vact: nº elementos tupla actual.
        - + Inicialización (añadir): vact = 0
        - + Generar (añadir): vact = vact + s[nivel]
        - + Retroceder (añadir): vact = vact – s[nivel]



2 – Esquema general.

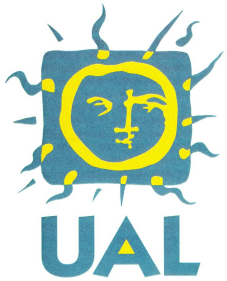
Esquema general recursivo de solución con backtracking

✓La solución con backtracking se encuentra añadiendo elementos a una solución parcial, que se implementa recorriendo un árbol.

✓Los árboles son estructuras intrínsecamente recursivas, cuyo manejo requiere casi siempre recursión, en especial en lo que se refiere a sus recorridos.

✓Se puede lograr una implementación sencilla con procedimientos recursivos.

✓ La búsqueda se realiza utilizando en el recorrido del árbol una *función de factibilidad*, que nos indica si desde el punto en que estamos, es decir al añadir un elemento, es posible encontrar la solución (no lo asegura.)



2 – Esquema general.

Esquema general recursivo de solución con backtracking

✓La búsqueda se realiza utilizando en el recorrido del árbol una *función de factibilidad*, que nos indica si desde el punto en que estamos, es decir al añadir un elemento, es posible encontrar la solución (no lo asegura.)

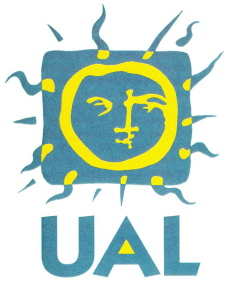
- Cualquier combinación con un elemento no factible se convierte a su vez en combinación no factible.
- Cuando se encuentra un elemento no factible, se eliminan todas las combinaciones que lo contengan, es decir, cuando se llega a un nodo fracaso, hay que deshacer la última decisión tomada para optar por la siguiente alternativa.

2 – Esquema general.

Esquema general recursivo de solución con backtracking

```
Método Backtracking_R (etapa): solucion;
    IniciarOpciones; éxito =FALSE
    repetir
            SeleccionarNuevaOpcion;
            si Aceptable entonces
                AnotarOpcion;          // Se incorpora la opción a la solución
                si SolucionIncompleta entonces
                        Backtracking_R(etapa_siguiente);
                        si NO exito entonces
                            CancelarAnotacion
                        fsi
                sino                          // solucion completa
                        exito:=TRUE
                fsi
            fsi
    hasta (exito) O (UltimaOpcion)
    retornar solucion
fmétodo
```

Obtención de UNA solución



2 – Esquema general.

Esquema general recursivo de solución con backtracking

Elementos de la solución:

Obtención de UNA solución

+ Generación de descendientes, en donde para cada nodo generamos sus descendientes con posibilidad de solución. A este paso se le denomina expansión, ramificación o bifurcación. (Bucle repetir)

+ Para cada uno de estos descendientes, hemos de aplicar lo que denominamos prueba de fracaso. (Aceptable) Comprobamos que con lo que tenemos aún podemos llegar a la solución.

+ Caso de que sea aceptable este nodo, aplicaremos la prueba de solución que comprueba si el nodo que es posible solución efectivamente lo es. (No SoluciónImcompleta.)

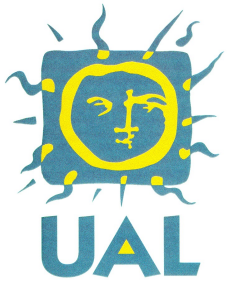
La vuelta atrás se realiza gracias al proceso de recursión (explora en profundidad), y el proceso de anotar y cancelar anotaciones (avance y retroceso).

2 – Esquema general.

Esquema general recursivo de solución con backtracking

```
Método Backtracking_Rts (etapa): soluciones[];
    IniciarOpciones; éxito =FALSE
    repetir
            SeleccionarNuevaOpcion;
            si Aceptable entonces
                AnotarOpcion;          // Se incorpora la opción a la solución
                si SolucionIncompleta entonces
                        Backtracking_Rts(etapa_siguiente);
                        si NO exito entonces
                           CancelarAnotacion
                        fsi
                sino                          // incorporamos solución
                        IncluirSolucion
                fsi
            fsi
    hasta (UltimaOpcion)
    retorna soluciones
fmétodo
```

Obtención de TODAS las soluciones



2 – Esquema general.

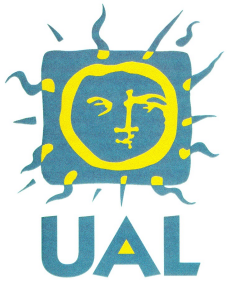
Conclusiones sobre el esquema general esquema backtracking

+ 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 e incluir otros tipos de recorridos (no solo en profundidad) → Técnica de Ramificación y Poda.



3 – Análisis de tiempos de ejecución.

Consideraciones generales:

+ La eficiencia de estos algoritmos depende de:

  - ❑ El número de nodos que haya que explorar para cada caso, lo cual es imposible de calcular a priori de forma exacta.
  - ❑ El tiempo requerido para tratar cada nodo, que viene dado por el coste de las funciones.

+ Sea n el número de etapas necesarias para construir la solución (esto es, la longitud máxima de la tupla solución, o la profundidad del árbol de exploración) y [0..v-1] el rango de valores para la decisión correspondiente a cada etapa. Se trata y construye un árbol k-ario.

3 – Análisis de tiempos de ejecución.

Consideraciones generales:

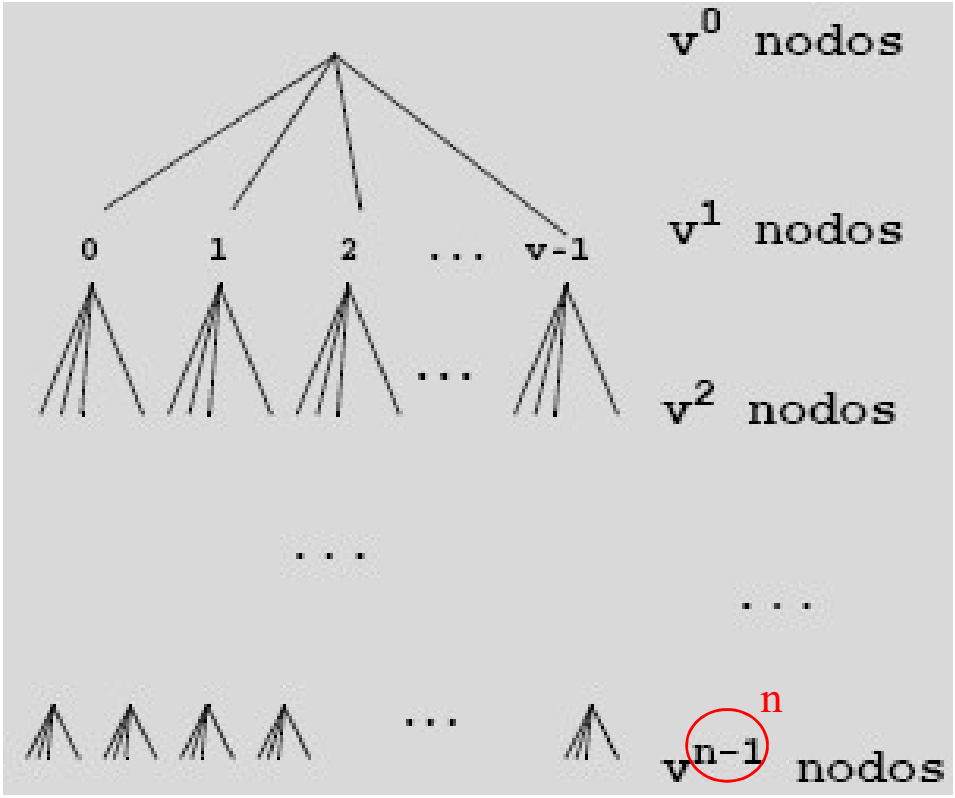
+ Mejor caso: se encuentra la solución en el primer recorrido en profundidad. Solo se tienen que generar y analizar n nodos:

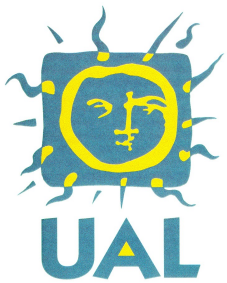
O(n) - Pendiente del coste de los métodos.

+ Peor caso: se ha de explorar el árbol completo. Se puede evaluar la cota superior (pendiente coste métodos internos):

$$\text{nodos} = \sum_{i=0}^{n} v^i \approx v^n$$

$O(v^n)$ cota superior





3 – Análisis de tiempos de ejecución.

Ejemplo con árboles binarios o combinatorios:

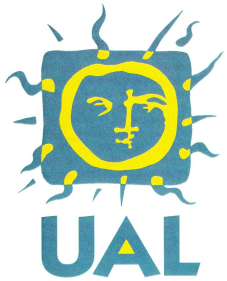
Ejemplo del problema de la elección de elementos de un subconjunto (selección de valores que suman algo, mochila 0/1,…). Se construyen árboles binarios y el número de nodos a construir es:

$$numNodos(n) = 2 + 2^2 + 2^3 + ... + 2^n = 2^{n+1} - 2 \qquad **$$

De igual forma, si se utilizan los árboles de tipo combinatorio, se puede ver que el número de nodos es:

$$numNodos(n) = n! \qquad \text{(a construir n!-1)}$$

En el peor caso, si el número de nodos es 2^n o n!, el tiempo de ejecución será de orden O (p(n)2^n) u O (q(n)n!), respectivamente, con p y q polinomios en n (que representan el coste de los métodos internos).



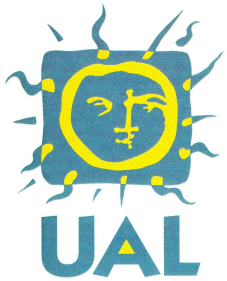
3 – Análisis de tiempos de ejecución.

Análisis de la aplicabilidad de la estrategia con Backtracking:

+ En general, los algoritmos con backtracking dan lugar a tiempos de órdenes factoriales o exponenciales → No usar si existen otras alternativas más rápidas.

+ El uso de restricciones, tanto implícitas como explícitas (que permiten declarar un nodo como fracasado y no explorar más), trata de reducir este tiempo, pero en muchos casos no es suficiente para conseguir algoritmos tratables, cuyos tiempos de ejecución sean de orden de complejidad razonable.

+ Cuando se busca una solución (no todas ni un óptimo). Podemos considerar la posibilidad de distintas formas de ir generando los nodos del árbol, para lograr esto sólo hemos de ir variando el orden en el que se generan los descendientes de un nodo, de manera que trate de ser lo más apropiado a nuestra estrategia.



3 – Análisis de tiempos de ejecución.

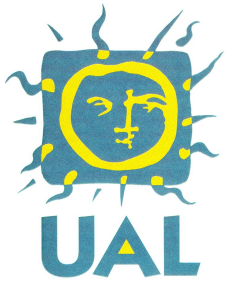
Análisis de la aplicabilidad de la estrategia con Backtracking:

+ El test de factibilidad (o chequeo) permite ver de que tipo es cada nodo … y por tanto si podemos abandonar la exploración en ese punto o no.

+ Los test de chequeo van a determinar la eficiencia de un algoritmo con backtracking.

Eficiencia en memoria:

Los algoritmos con backtracking tienen asociados unos requisitos de memoria, propia del esquema backtracking, O(n), dados por la máxima profundidad de las llamadas recursivas.



3 – Análisis de tiempos de ejecución.

Recomendaciones para una buena eficiencia (relativa):

1. Buena elección del conjunto de posibles soluciones → la menor posible.
2. Intentar aplicar el test solución antes de llegar a un nodo solución → detectar lo antes posible si estamos en un Nodo fracaso, así evitamos explorar su descendencia pues es trabajo inútil.
3. El test debe ser aplicable con la menor cantidad de información.
4. El test debe ser ligero para que no necesite tiempo extra, que supondría una complicación si además no detecta muchos nodos fracaso.
5. Regla general: test sencillos; los test sofisticados para situaciones desesperadas con árboles gigantescos.

4 – Ejemplos: Búsqueda exhaustiva en grafos.

Planteamiento del problema:

+ Dado un grafo conexo. Se parte de un nodo dado y se visitan los vértices del grafo de manera ordenada y sistemática, pasando de un vértice a otro a través de las aristas del grafo.

  Se utiliza cuando queremos hacer alguna operación con todos los nodos del árbol (en nuestro caso diremos que ha sido visitado) o localizar el nodo que optimiza cierta condición (se añade índice.)

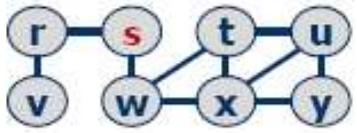
+ Tipos de recorridos:

  ○ Búsqueda en profundidad: Equivalente al recorrido en preorden de un árbol. Este supone aplicar la estratégia backtracking.

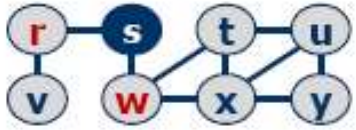
  ○ Búsqueda en anchura: Equivalente al recorrido de un árbol por niveles. Este sigue un esquema simple de ramificación.

4 – Ejemplos: Búsqueda exhaustiva en grafos.

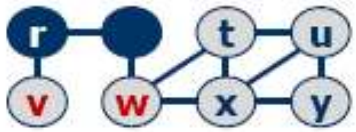
Recorrido en profundidad: ejemplo



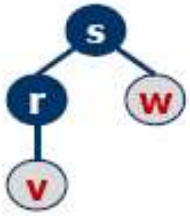
Pila S={}



Pila S={s}



Pila S={r,s}



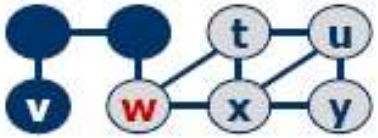
En la pila S, tenemos el camino activo.

Los nodos visitables salen de analizar las aristas que conectan el último nodo seleccionado con nodos no seleccionados.

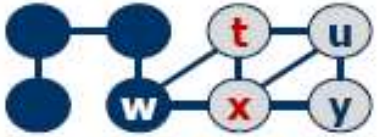
Debemos tener almacenado el conjunto de nodos visitados y no visitados.

4 – Ejemplos: Búsqueda exhaustiva en grafos.

Recorrido en profundidad: ejemplo

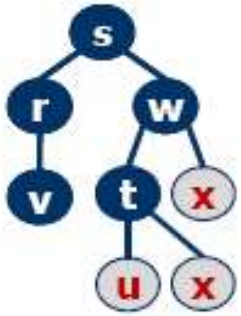


Pila S={v,r,s}



Pila S={w,s}

Pila S={t,w,s}

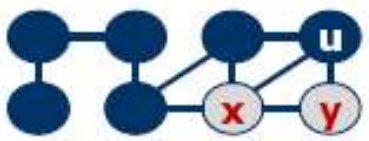


El backtracking está implícito en el hecho de volver a los nodos no visitados cuando se termina un hilo.

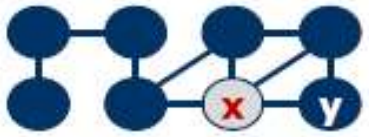
La pila S termina dando el recorrido que nos lleva al último nodo por visitar.

4 – Ejemplos: Búsqueda exhaustiva en grafos.

Recorrido en profundidad: ejemplo

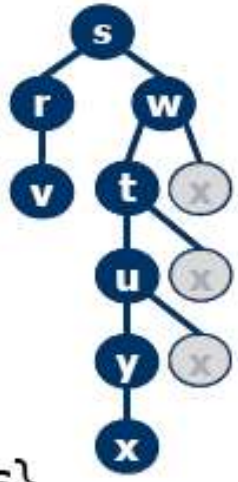


Pila S={u,t,w,s}



Pila S={y,u,t,w,s}

Pila S={x,y,u,t,w,s}

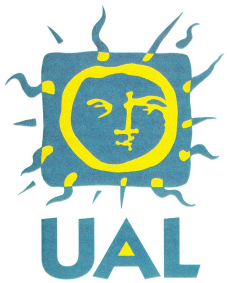


La pila S termina dando el recorrido que nos lleva al último nodo por visitar.

Lo importante es visitar todos los nodos… ese es el verdadero objetivo del algoritmo.

El recorrido no es único: depende del vértice inicial y del orden de visita de los vértices adyacentes.

El orden de visita de unos nodos puede interpretarse como un árbol: árbol de expansión en profundidad asociado al grafo.



4 – Ejemplos: Búsqueda exhaustiva en grafos.

Esquema algorítmico de la solución:

```
método DFS_Lanzador (Grafo G(V,E))        // DFS Depth-First-Search
        para (i=0; i<V.length; i++)
                visitado[i] = false
        fpara
        para (i=0; i<V.length; i++)
                si (NO visitado[i])
                        DFS(G,i)
                fsi
        fpara
fmétodo
```

```
Método DFS (Grafo G(V,E), int i)
        visitado[i] = true;
        para cada (v[j] adyacente a v[i])
                si (NO visitado[j])
                        DFS(G,j)
                fsi
        fparaCada
fmétodo
```

La eficiencia es O(|V|+|E|) si usamos la representación basada en listas de adyacencia.

4 – Ejemplos: Búsqueda exhaustiva en grafos.

Modo de trabajo:

- ✓ Se comienza visitando un nodo cualquiera.
- ✓ Se recorre en profundidad la componente conexa que cuelga de cada sucesor (se examinan los caminos hasta que se llega a nodos ya visitados o sin sucesores).
- ✓Si después de haber visitado todos los descendientes del primer nodo (él mismo, sus sucesores, los sucesores de sus sucesores, …) todavía quedan nodos por visitar, se repite el proceso a partir de cualquiera de estos nodos no visitados.

Aplicaciones con variaciones:

- ❑ Analizar la robustez de una red de computadores representada por un grafo.
- ❑ Examinar si un grafo dirigido tiene ciclos, antes de aplicar sobre él cualquier algoritmo que exija que sea acíclico. (Requiere modificación.)

4 – Ejemplos: El problema de la Mochila 0/1.

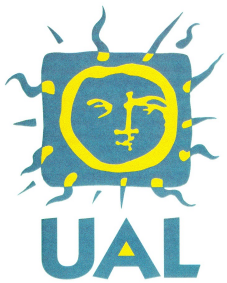
Planteamiento del problema:

+ Datos: n: número de objetos disponibles.
W: capacidad de la mochila.
$w = (w_1, w_2, ..., w_n)$ pesos de los objetos.
$b = (b_1, b_2, ..., b_n)$ beneficios de los objetos.

+ Solución: $\{x_1,..,x_i,...,x_n\} / x_i \in \{0,1\}$.

+ Objetivo: Seleccionar los objetos que nos garantizan un beneficio máximo, pero con un peso global menor o igual que W.

Maximizar $(\Sigma\, x_i \cdot b_i) / \Sigma\, x_i \cdot w_i \leq W$ con $i=1..n$, $x_i \in \{0, 1\}$.



4 – Ejemplos: El problema de la Mochila 0/1.

Aplicación de backtracking (proceso metódico):

1. Determinar cómo es la forma del árbol de backtracking ↔ cómo es la representación de la solución.

2. Elegir el esquema de algoritmo adecuado, adaptándolo en caso necesario.

3. Diseñar las funciones genéricas para la aplicación concreta: según la forma del árbol y las características del problema.

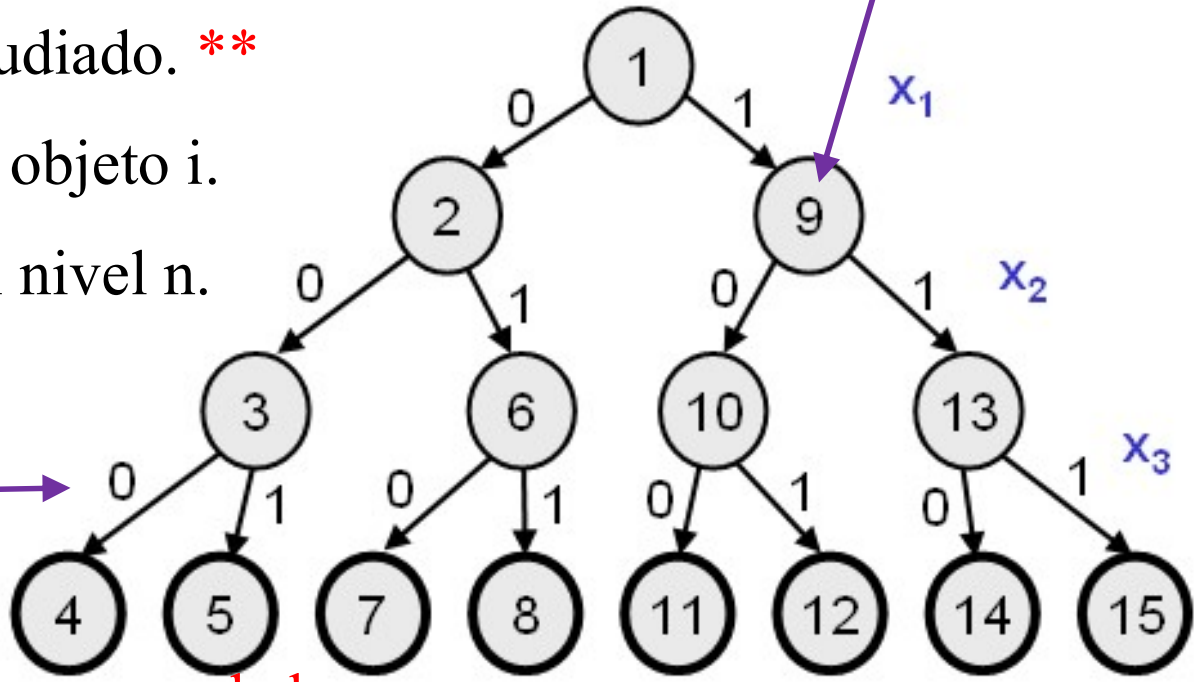
4. Posibles mejoras: usar variables locales con valores acumulados, hacer más podas del árbol, etc.

4 – Ejemplos: El problema de la Mochila 0/1.

Representación de la solución: (1) Árbol binario

- $s = (x_1, x_2, ..., x_n)$, con $x_i \in \{0,1\}$ **
- $x_i = 0 \rightarrow$ No se coge el objeto i.
- $x_i = 1 \rightarrow$ Sí se coge el objeto i.
- $x_i = -1 \rightarrow$ Objeto i no estudiado. **
- En el nivel i se estudia el objeto i.
- Las soluciones están en el nivel n.

El número representa el orden en que se recorren los nodos.



Cada nivel representa un objeto.
(0 no elegir, 1-elegir)

Luego los nodos almacenan el peso acumulado.

4 – Ejemplos: El problema de la Mochila 0/1.

Representación de la solución: (2) Árbol combinatorio

$s= (x_1, x_2, ..., x_m) / m \leq n, x_i \in \{1, ..., n\}$ y $x_i < x_{i+1}$

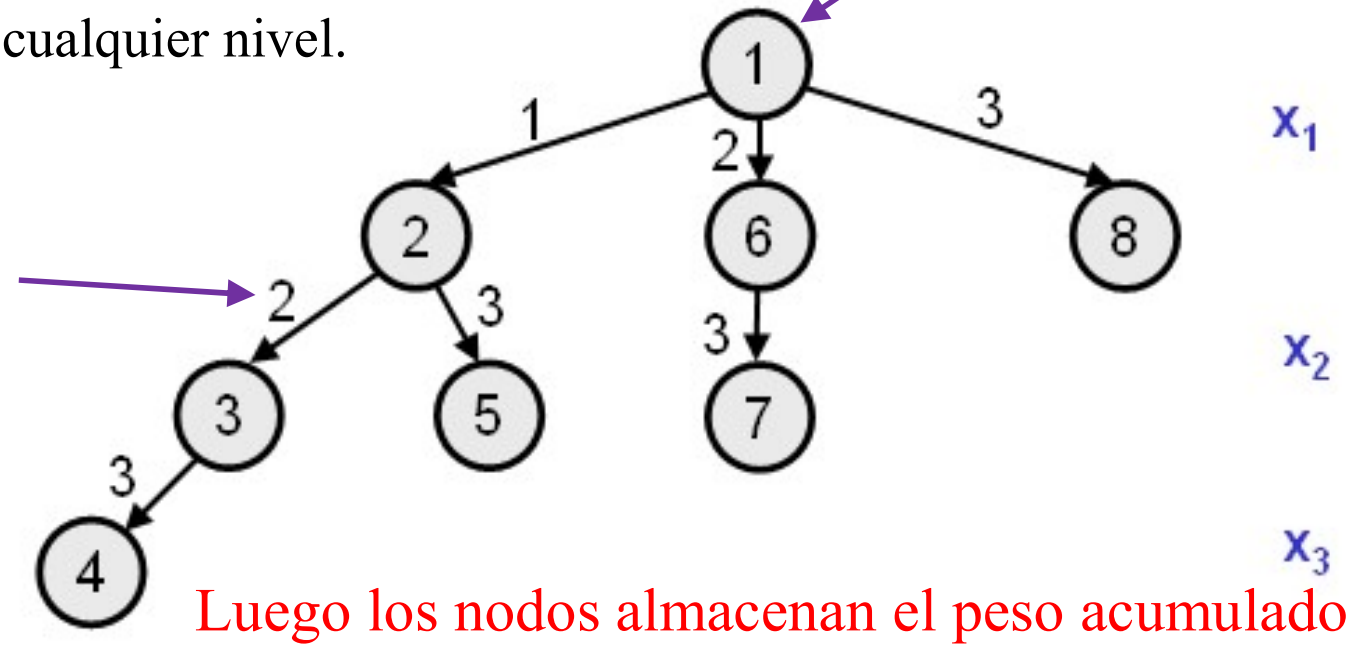
$x_i \rightarrow$ Número de objeto escogido.

$m \rightarrow$ Número total de objetos escogidos.

Las soluciones están en cualquier nivel.

El número de nodo representa el orden en que se recorren los nodos.

El número de arista representa elegir el objeto con ese número.



Luego los nodos almacenan el peso acumulado.

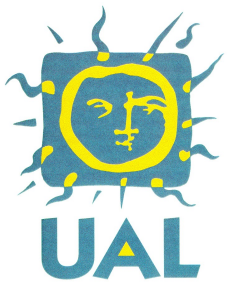
4 – Ejemplos: El problema de la Mochila 0/1.

Esquema algorítmico: iterativo / optimización

```
Método Backtracking (real[] b,p): Tuplasolucion
        nivel = 1;  s = s.Inicial;
        voa = -∞; soa = Ø          // voa, soa: valor, solución óptimos actuales
        pact=0; bact=0             // peso actual, beneficio altual
        repetir
            Generar (nivel, s)
            si Solución (nivel, s) Y (bact > voa)  entonces
                        voa = Valor(s); soa = s
            sino si Criterio (nivel, s) entonces
                        nivel = nivel + 1
            sino
                    mientras NO MasHermanos(nivel, s) Y (nivel>0)  hacer
                        Retroceder (nivel, s)
            fsi
        hasta (nivel=0)
        retornar soa               // La mejor solución almacenada
```

Árbol binario

fmétodo



4 – Ejemplos: El problema de la Mochila 0/1.

Esquema algorítmico: optimización

- Para calcular el peso y el beneficio en cada nodo podemos usar variables locales pact, bact, que guardarán el peso y el beneficio acumulado.
- El array de soluciones será s: array [1..n] of –1,0,1.
  - ✓s[i] = 1, 0. Se añade o no se añade el objeto i.
  - ✓s[i] = -1. No se ha considerado el objeto i (valor inicialización).
- Al ser un problema de optimización no acabamos hasta haber recorrido todos los nodos. Acabar cuando nivel = 0 (volvemos al nodo raíz).
- En cada momento llevamos la mejor solución hasta un nodo. Si encontramos una solución nueva, comprobar si es mejor que la solución actual.
- Variable voa, con el valor de la mejor solución hasta este nodo y soa con los objetos que la componen.

4 – Ejemplos: El problema de la Mochila 0/1.

Funciones genéricas del esquema:

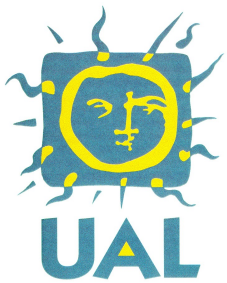
- Generar (nivel, s) → Probar primero 0 y luego 1

s[nivel]= s[nivel] + 1

	si s[nivel]=1 entonces
pact= pact + p[nivel]*s[nivel]	pact= pact + p[nivel]
bact= bact + b[nivel]*s[nivel]	bact= bact + b[nivel]
	finsi

- Solución (nivel, s) → Indica los nodos hoja que cumplen la restricción de peso.

devolver (nivel=n) AND (pact ≤ W)



4 – Ejemplos: El problema de la Mochila 0/1.

Funciones genéricas del esquema:

- Criterio (nivel, s) → Indicará si se cumple la restricción de peso y no estamos en el último nivel.

devolver (nivel<n) AND (pact ≤ W)

- MasHermanos (nivel, s)

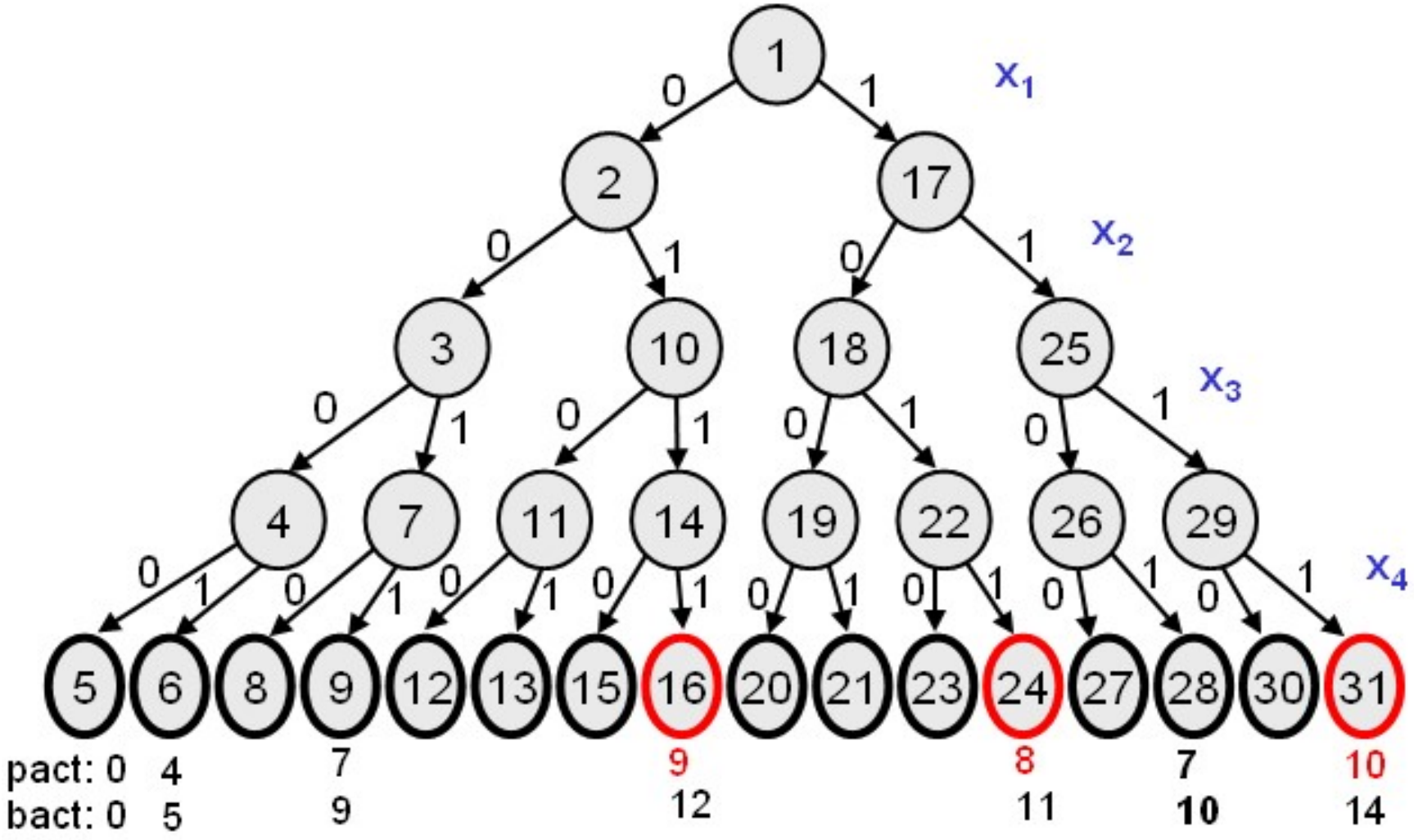
devolver s[nivel] < 1

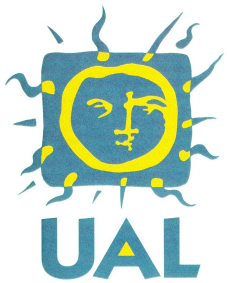
- Retroceder (nivel, s)

pact= pact – p[nivel]*s[nivel]
bact= bact – b[nivel]*s[nivel]
s[nivel]= -1
nivel= nivel – 1

4 – Ejemplos: El problema de la Mochila 0/1.

Ejemplo: n = 4; W = 7; b = (2, 3, 4, 5); w = (1, 2, 3, 4)





4 – Ejemplos: El problema de la Mochila 0/1.

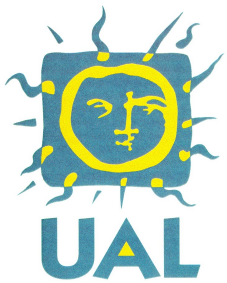
Análisis de la solución:

✓ El algoritmo resuelve el problema, encontrando la solución óptima, pero es muy ineficiente.

✓ Orden de complejidad del algoritmo: Número de nodos generado= $2^{n+1}-1$. El algoritmo es de $O(2^n)$.

✓ Problema adicional: en el ejemplo, se generan todos los nodos posibles, no hay ninguna poda. La función Criterio es siempre cierta (excepto para algunos nodos hoja).

✓ Solución: Mejorar la poda con una función Criterio más restrictiva.



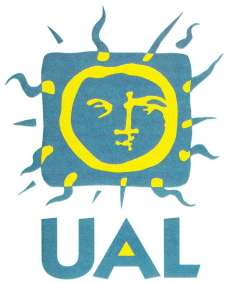
4 – Ejemplos: El problema de la Mochila 0/1.

Mejoras del método Criterio con condiciones de poda:

+ Se podría incluir una poda según el criterio de optimización:
  - ❑ Poda según el criterio de peso: si el peso actual es mayor que W podar el nodo.
  - ❑ Poda según el criterio de optimización: si el beneficio actual no puede mejorar el voa, podar el nodo.

+ A pesar de esta poda, en el peor caso, el orden de complejidad sigue siendo un $O(2^n)$.

+ En promedio, se espera que la poda elimine muchos nodos, reduciendo el tiempo total. Pero el tiempo sigue siendo muy malo.



4 – Ejemplos: El problema de la Mochila 0/1.

Mejoras estructurales:

+ Utilizar el concepto de densidad de valor:
  - ❑ Para cada nodo, generar primero el valor 1 y luego el valor 0 (en lugar de primero 0 y luego 1). Idea: es de esperar que la solución de la mochila 0/1 sea "parecida" a la de la mochila no 0/1.
  - ❑ Ordenamos los objetos por b_i/w_i entonces tendremos una solución con 1 en las primeras posiciones, para los valores con mayor densidad de valor (mejoramos así la poda por no alcanzar el voa en nodos posteriores.)

+ Usar un árbol combinatorio.

4 – Ejemplos: El problema del viajante.

Planteamiento del problema:

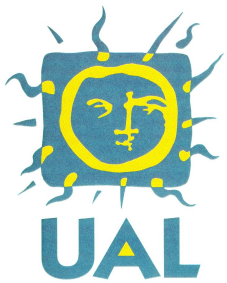
- Buscamos determinar todos los ciclos hamiltonianos de un grafo conexo G = <V, A> con n vértices.
- Un ciclo hamiltoniano es un camino que recorre los n vértices del grafo, visitando una sola vez cada vértice, hasta finalizar en el vértice de partida.
- Vector solución: $(x_1, ..., x_n)$ tal que x_i representa el i-ésimo vértice visitado en el ciclo propuesto.
- Todo lo que se necesita es determinar el conjunto de posibles vértices x_k una vez elegidos $x_1, ..., x_{k-1}$.

4 – Ejemplos: El problema del viajante.

Condiciones de la implementación:

- Si k=1, x[1] puede ser cualquiera de los n vértices, aunque para evitar encontrar el mismo ciclo n veces, exigiremos que x[1] = 1.
- Si 1 < k < n, entonces x[k] puede ser cualquier vértice v que sea distinto de los vértices x[1], x[2], ..., x[k-1] y que esté conectado mediante una arista a x[k-1].
- x[n] sólo puede ser el único vértice restante y debe estar conectado tanto a x[n-1] como a x[1].

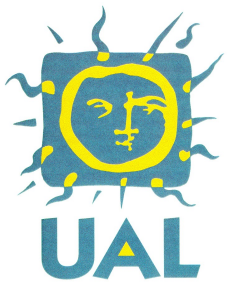
- Se utiliza un árbol permutacional (2).
- El método criterio evalúa la disponibilidad de aristas para continuar (2) (3).



4 – Ejemplos: El problema del viajante.

Esquema algorítmico: métodos valor y hamiltoniano

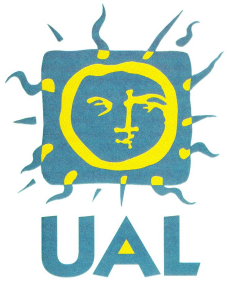
```
Método siguienteValor (k): int          // siguiente vértice válido, 0 si ninguno
// x[1..k-1] vértices ya asignados, G[ ] [ ] matriz de adyacencia grafo
    x[k] = 0     // Inicialización
    hacer
            x[k]++;
            si (G[x[k-1]][x[k]] && x[k] ∉ x[1..k-1] &&
                                    (k<N || (k==N && G [x[k]] [x[1]] ))) entonces
                    return x[k];
    mientras (x[k]<N);
    x[k]=0
    return 0
fmétodo
```



4 – Ejemplos: El problema del viajante.

Esquema algorítmico: métodos valor y hamiltoniano

```
método hamiltoniano (k)  // x[1..k-1] vértices ya asignados
    si (k==N) entonces
        almacenar x[1..N]    // Una solución
    sino
        hacer
            x[k] = siguienteValor(k);
            si (x[k]!=0)
                hamiltoniano(k+1);
        mientras (x[k]!=0);
    fsi}
fmétodo
```



4 – Ejemplos: El problema del viajante.

Comentarios de la solución:

- Este algoritmo encuentra “todos” los ciclos hamiltonianos, tendríamos que evaluar cuál es el de coste mínimo. Sustituimos almacenar por evaluar el coste respecto al que tenemos almacenado en este momento y lo sustituimos si es menor. Se inicializa el coste del coste por $+\infty$.

- Si buscamos el óptimo, se puede mejorar el criterio teniendo el cuenta el costo ya acumulado frente al de la última solución almacenada.

- Se disponía de una heurística voraz muy eficiente, pero sub-óptima para este problema.

- También disponíamos de una solución con programación dinámica y otra de fuerza bruta.

4 – Ejemplos: Planificación de tareas.

Planteamiento del problema:

Vamos a considerar la siguiente situación:

- ✓ 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**

Problema NP Completo

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

4 – Ejemplos: Planificación de tareas.

Planteamiento de la solución:

- Datos del problema:
  - n: número de personas y de tareas disponibles.
  - B array [1..n, 1..n] de enteros, contiene el rendimiento o beneficio de cada asignación: B[i, j] = beneficio de asignar a la persona i la tarea j.

- Resultado:

  Realizar n asignaciones $\{(p_1, t_1), (p_2, t_2), ..., (p_n, t_n)\}$.

  ** a discutir **

Formulación matemática:

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

4 – Ejemplos: Planificación de tareas.

Planteamiento de la solución: representación

1. Pares de asignaciones:

$$s = \{(p_1, t_1), (p_2, t_2), ..., (p_n, t_n)\}, \text{ con } p_i \neq p_j,\ t_i \neq t_j,\ \forall\ i \neq j$$

- La tarea t_i es asignada a la persona p_i.

- Árbol muy ancho (k-ario).

- Hay que garantizar muchas restricciones.

- Representación no muy buena.

4 – Ejemplos: Planificación de tareas.

Planteamiento de la solución: representación

2. Matriz de asignaciones:

$s = ((a_{11}, a_{12}, ..., a_{1n}), (a_{21}, a_{22}, ..., a_{2n}), ..., (a_{n1}, a_{n2}, ..., a_{nn})) / a_{ij} \in \{0,1\}$

con $\sum_{i=1..n} a_{ij} = 1$. $\sum_{j=1..n} a_{ij} = 1$.

Representando:

$a_{ij} = 1 \rightarrow$ 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	1	2	3
1	**0**	**1**	**0**
2	**1**	**0**	**0**
3	**0**	**0**	**1**

- Árbol binario, pero muy profundo: n^2 niveles en el árbol.
- También tiene muchas restricciones.

4 – Ejemplos: Planificación de tareas.

Planteamiento de la solución: representación

3. Vector de asignaciones desde el punto de vista de las personas:

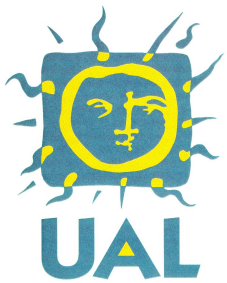
$s = (t_1, t_2, ..., t_n)$, siendo $t_i \in \{1, ..., n\}$, con $t_i \neq t_j$, $\forall\ i \neq j$

- t_i es el número de tarea asignada a la persona i.
- Da lugar a un árbol permutacional.

4. Vector de asignaciones desde el punto de vista de las tareas:

$s = (p_1, p_2, ..., p_n)$, siendo $p_i \in \{1, ..., n\}$, con $p_i \neq p_j$, $\forall\ i \neq j$

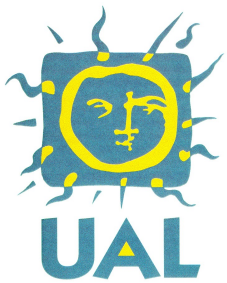
- p_i es el número de persona asignada a la tarea i.
- Representación análoga (dual) a la anterior.



4 – Ejemplos: Planificación de tareas.

Esquema algorítmico: optimización

```
Método Backtracking_so (..): Tuplasolucion
        nivel = 1;  s = s.Inicial; bact=0    // bact: beneficio actual
        voa = -∞; soa = Ø          // voa: valor óptimo actual;
        repetir                    // soa: solución óptima actual
            Generar (nivel, s)  // Actualiza bact
            si Solución (nivel, s) and bact > voa  entonces
                        voa = bact; soa = s
            fin_si                 //  A veces la solución es intermedia
            si Criterio (nivel, s) entonces        // A veces no hay que seguir
                        nivel = nivel + 1
            sino mientras NO MasHermanos(nivel, s) Y (nivel>0)  hacer
                        Retroceder (nivel, s)
            fsi
        hasta (nivel=0)
        retornar soa               // La mejor solución almacenada
```



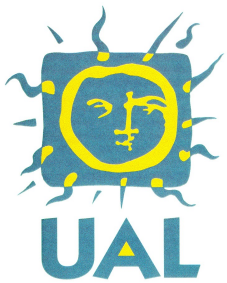
4 – Ejemplos: Planificación de tareas.

Elementos básicos 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]
fsi
```



4 – Ejemplos: Planificación de tareas.

Elementos básicos del esquema:

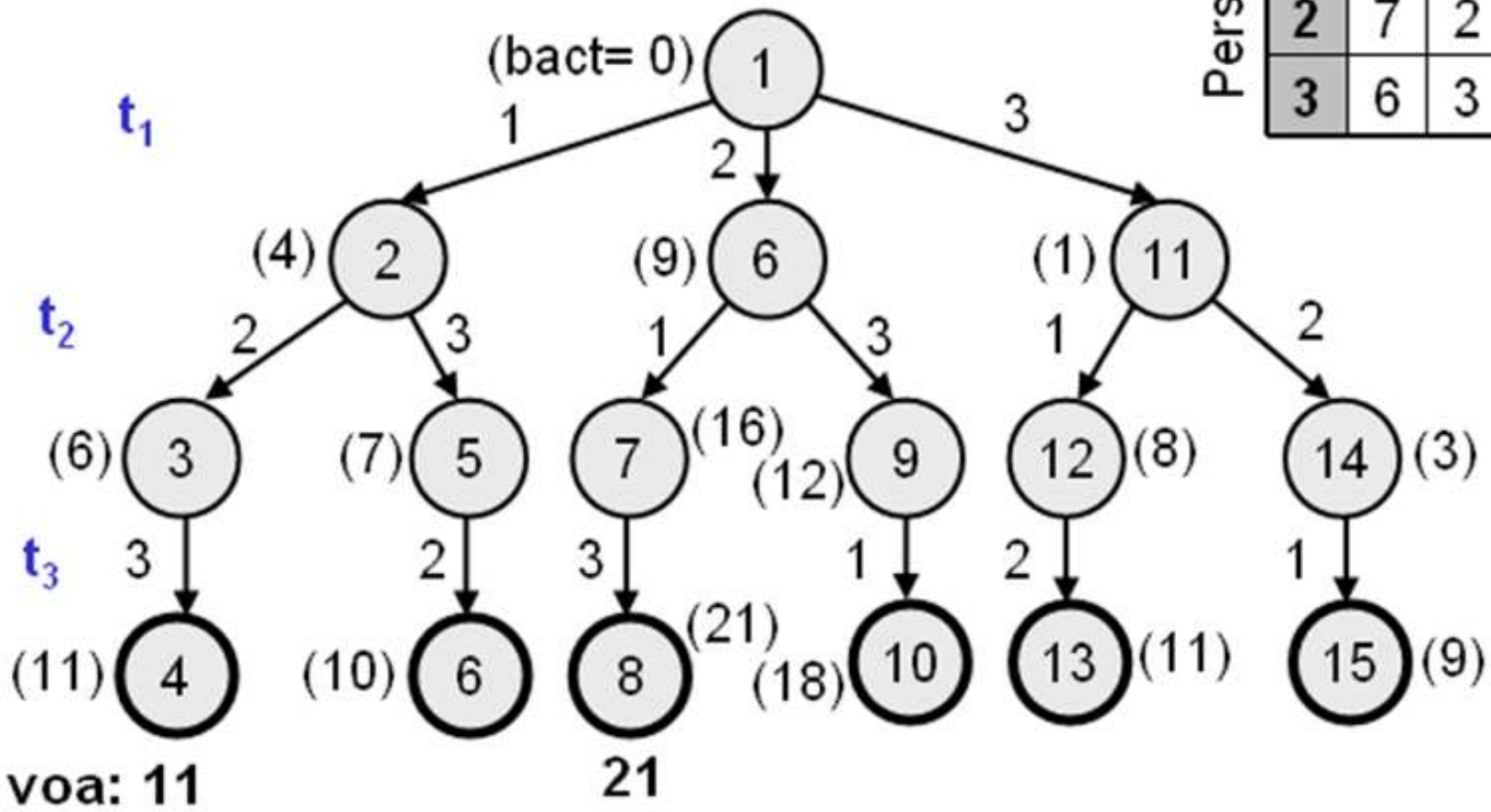
```
+ Criterio (nivel, s)
        para i= 1, ..., nivel-1 hacer
                si s[nivel] = s[i] entonces devolver false
        finpara
        devolver true

+ 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
```

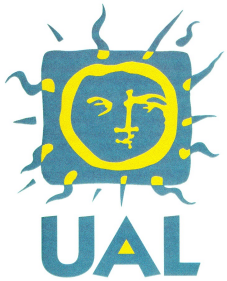
4 – Ejemplos: Planificación de tareas.

Ejemplo de aplicación: n =3

Tareas

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





4 – Ejemplos: Planificación de tareas.

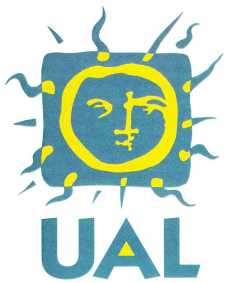
Análisis de la solución:

+ 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.



4 – Ejemplos: Planificación de tareas.

Análisis de la solución: cambios para mejora criterio

+ 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

Los métodos Solución y MasHermanos no cambian.

+ Generar (nivel, s)

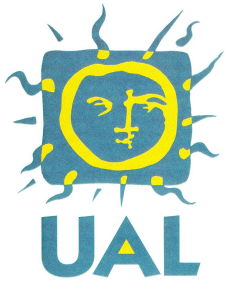
usada[s[nivel]]--

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]



4 – Ejemplos: Planificación de tareas.

Conclusiones:

- Se mejora la eficiencia (ver como cambia de la versión 1 a la 2, trabajo autónomo.)

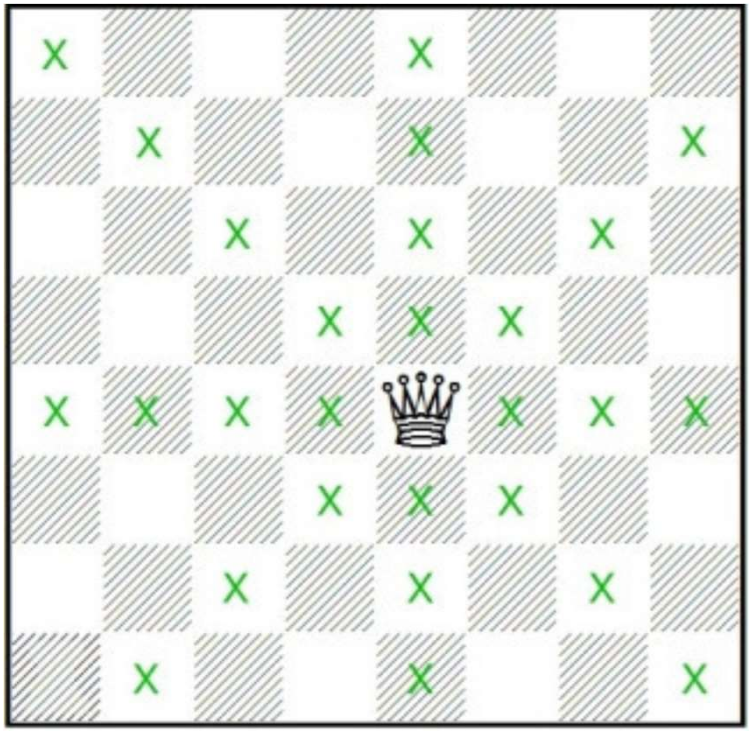
- A pesar de ello sigue siendo poco eficiente.

- Se podría intentar mejorar mediante una poda que utilice el criterio de optimización…. No siendo necesario llegar al final de cada rama.

4 – Ejemplos: Otros ejemplos.

El problema de las n reinas: origen 8 reinas

Se trata de colocar ocho reinas sobre un tablero de ajedrez, de tal forma que ninguna amenace (pueda comerse) a otra. Una reina amenaza a otra pieza que esté en la misma columna, fila o cualquiera de las dos diagonales.



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

Puesto que no puede haber más de una reina por fila, podemos replantear el problema: Colocar una reina en cada fila del tablero de forma que no se amenacen. En este caso, para ver si dos reinas se amenazan, basta con ver si comparten columna o diagonal.

4 – Ejemplos: Otros ejemplos.

El problema de las n reinas: origen 8 reinas

- Como cada reina debe estar en una fila diferente, sin pérdida 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 $(x_1, ..., x_8)$ en las que x_i indica la columna en la que se coloca la reina i.

- El problema verifica las condiciones necesarias para que pueda ser abordable utilizando Vuelta Atrás:
  + La solución se puede representar con una n-tupla.
  + Las componentes x_i se eligen una a una de entre un conjunto finito de valores (las columnas de la tabla).

4 – Ejemplos: Otros ejemplos.

El problema de las n reinas: origen 8 reinas

+ Espacio de soluciones:

$$\text{Tamaño } |S_i|^8 = 8^8 = 2^{24} = 16M$$

+ Restricciones *explícitas*:

$$S_i = \{1, 2, 3, 4, 5, 6, 7, 8\},\ 1 \leq i \leq 8$$

(Limita las opciones a las permutaciones, tamaño (8!) = 40320)

+ Restricciones *implícitas*:

- Ningún par (x_i, x_j) con $x_i = x_j$ (todas las reinas deben estar en columnas diferentes).
- Ningún par (x_i, x_j) con $|j-i| = |x_j-x_i|$ (todas las reinas deben estar en diagonales diferentes).

4 – Ejemplos: Otros ejemplos.

El problema de las n reinas: origen 8 reinas

- Aplicando las restricciones, en cada etapa k vamos a generar las k-tuplas con posibilidades de ser solución o parte de ella.

- Los nodos del árbol de profundidad k en la n-tupla que se va construyendo, si se verifican las restricciones, son k-prometedores. Si no se verifican, es un nodo fracaso.

- De esta manera, iremos generando el árbol.

- Eficiencia:

    Nº de posibles permutaciones: 8!

    Comprobación de restricciones nivel k: O(k)

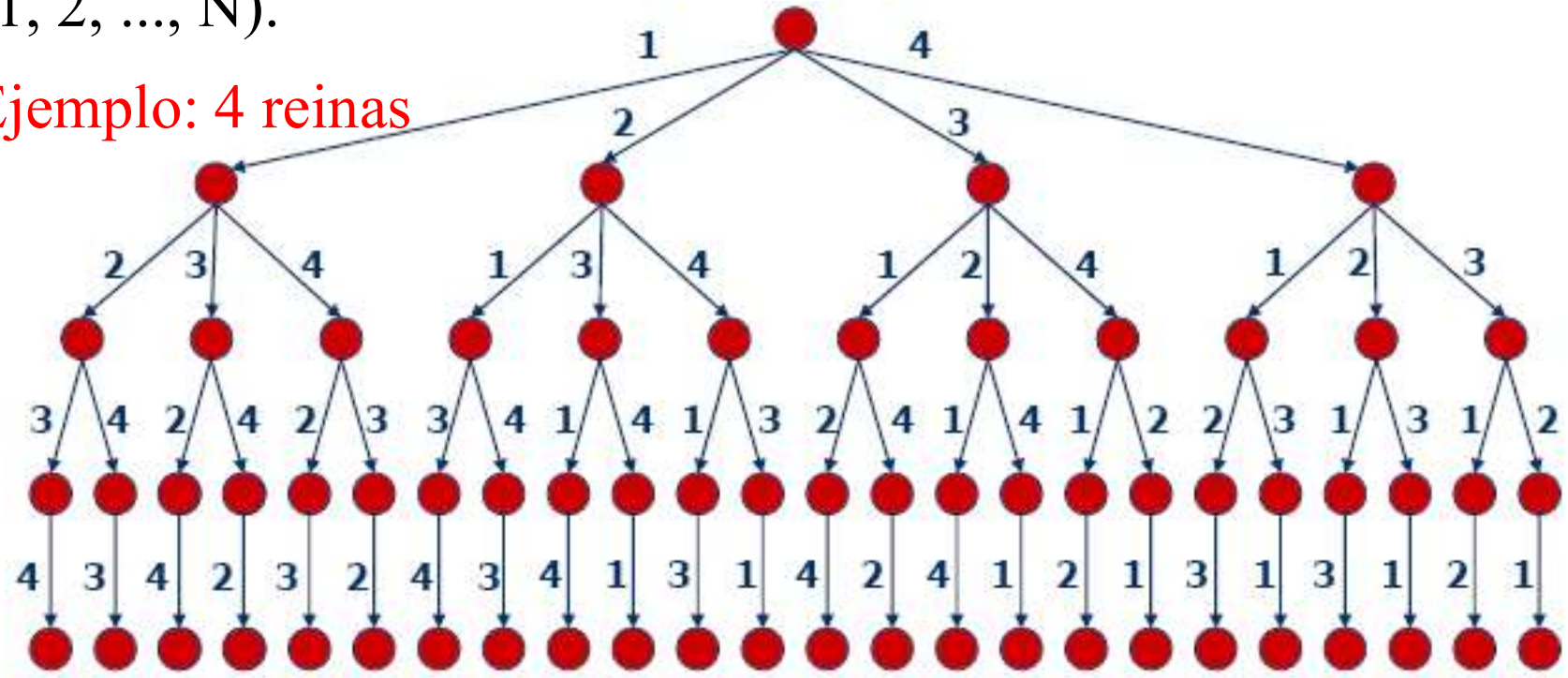
4 – Ejemplos: Otros ejemplos.

El problema de las n reinas: generalización del caso 8

Formulación del problema: Colocar N reinas en un tablero NxN, de modo se ataquen entre ellas.

- El espacio de soluciones consiste en las N! permutaciones de la N-tupla (1, 2, ..., N).

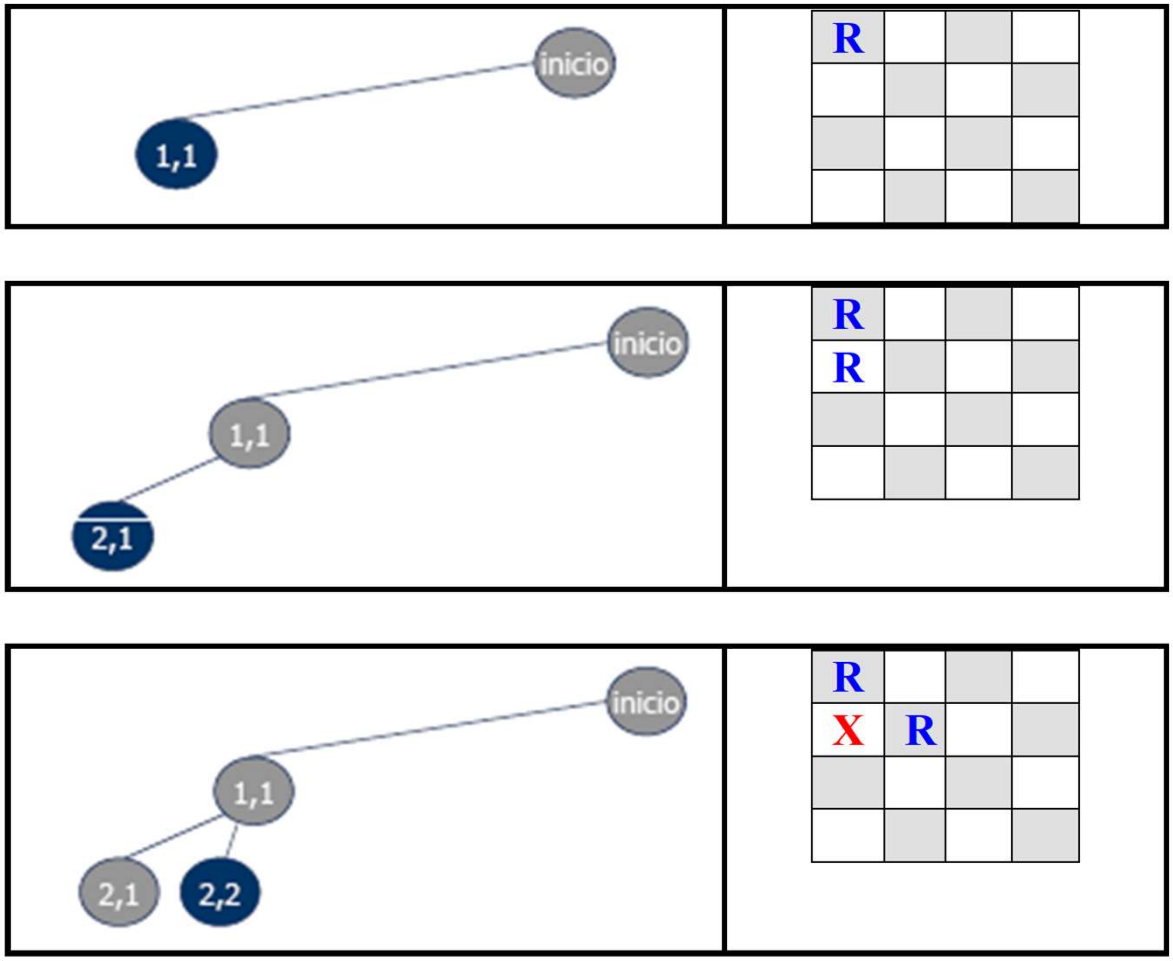
Ejemplo: 4 reinas



$T(n) \in \theta(n \cdot n!) = \theta(n!) \rightarrow$ NP-completo

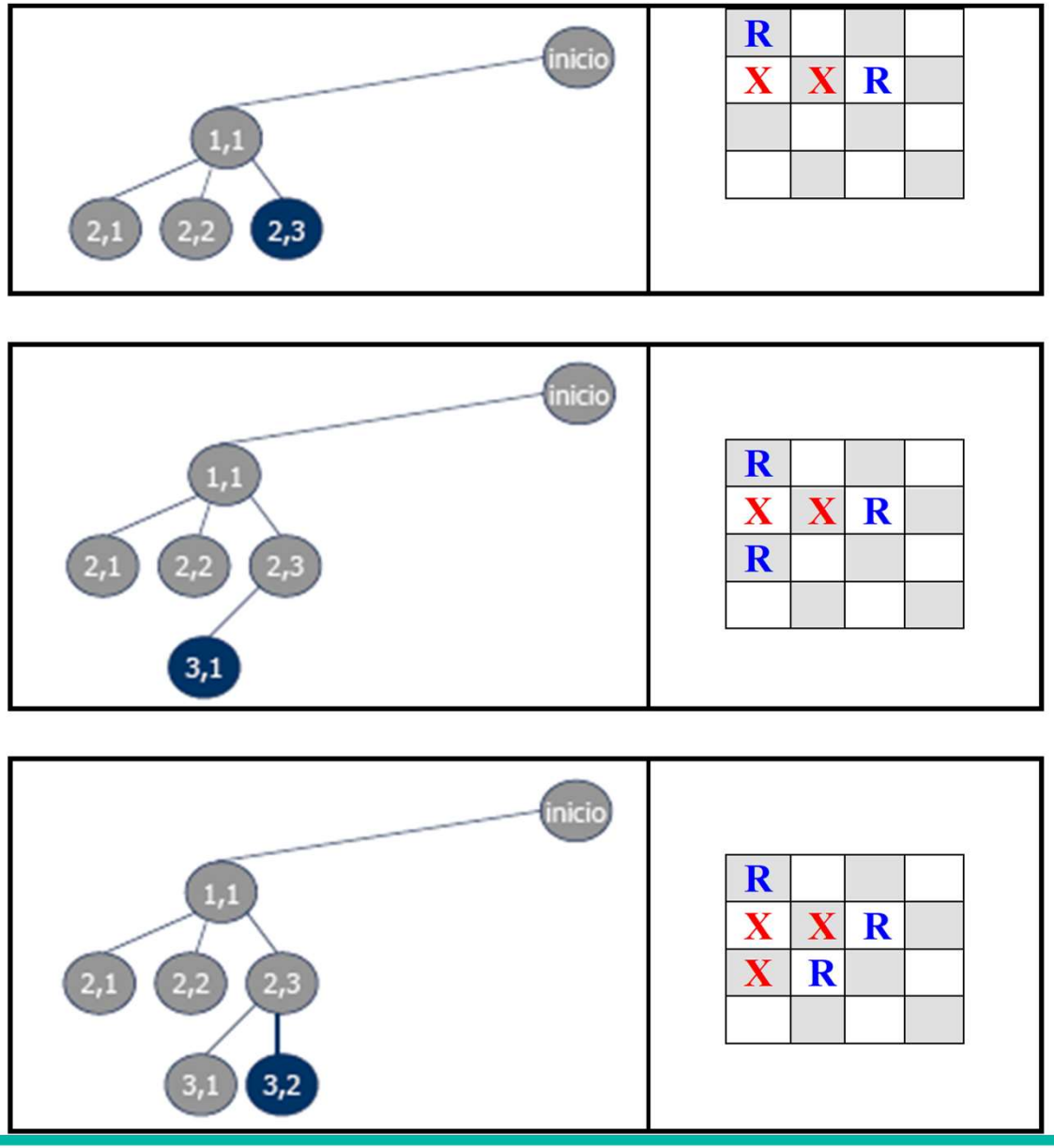
4 – Ejemplos: Otros ejemplos.

El problema de las n reinas. Ejemplo 4. Generación de estados



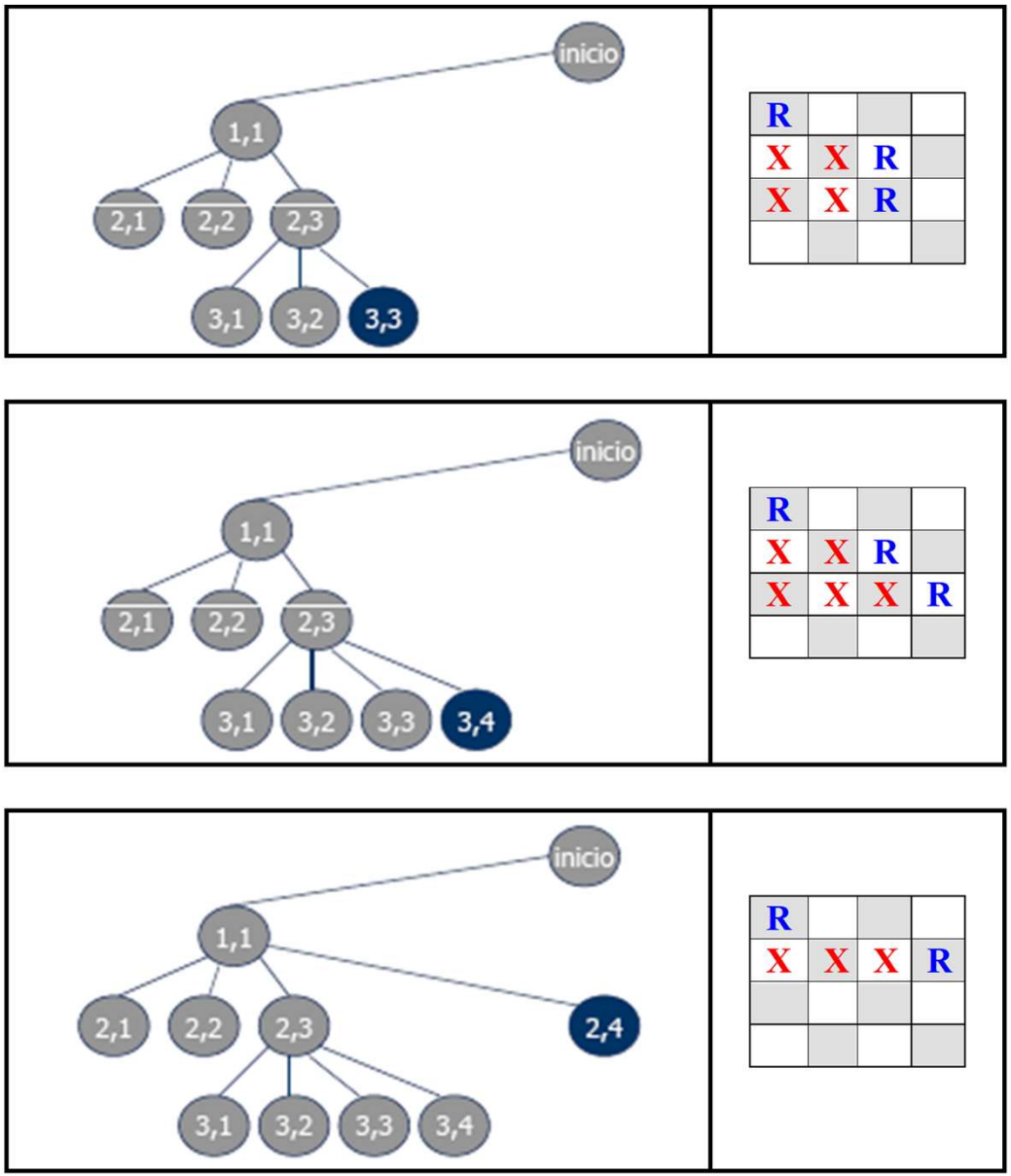
4 – Ejemplos: Otros ejemplos.

El problema de las n reinas. Ejemplo 4. Generación de estados



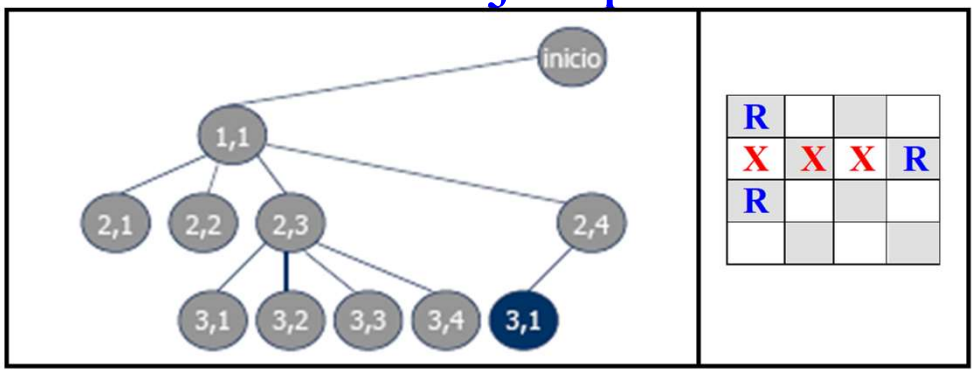
4 – Ejemplos: Otros ejemplos.

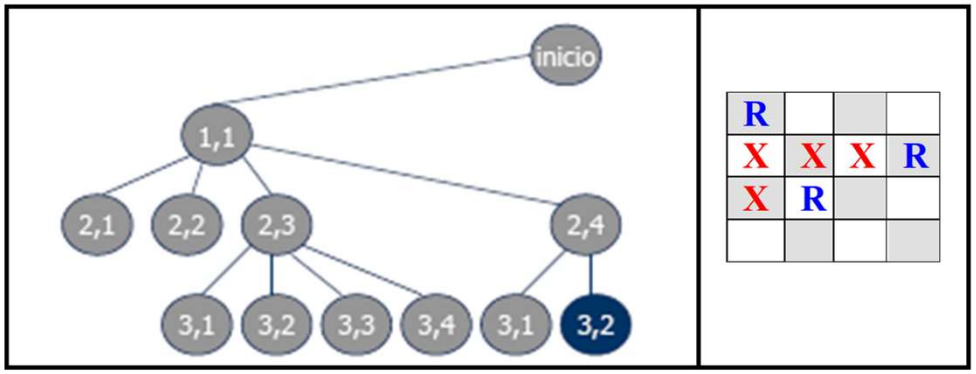
El problema de las n reinas. Ejemplo 4. Generación de estados

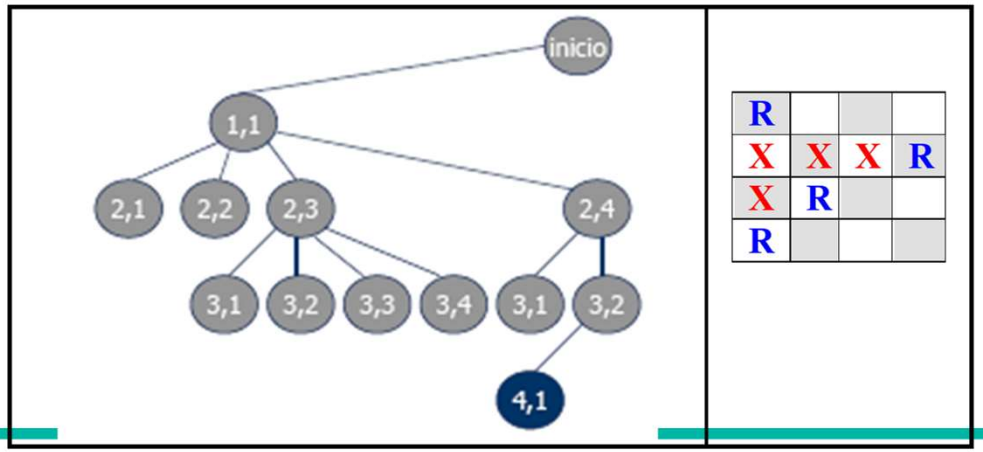


4 – Ejemplos: Otros ejemplos.

El problema de las n reinas. Ejemplo 4. Generación de estados

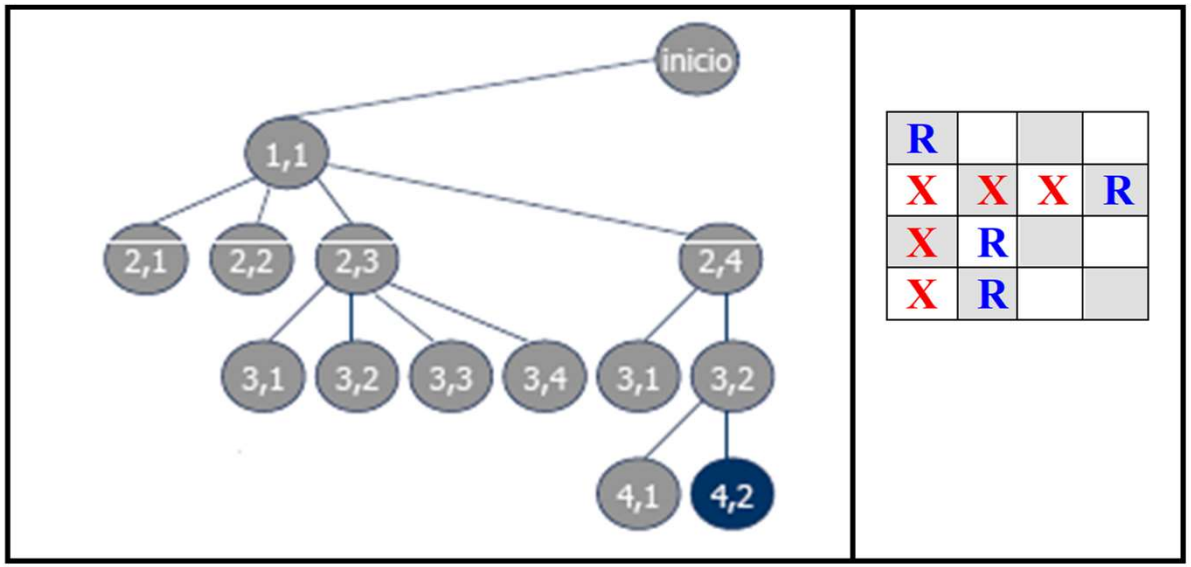


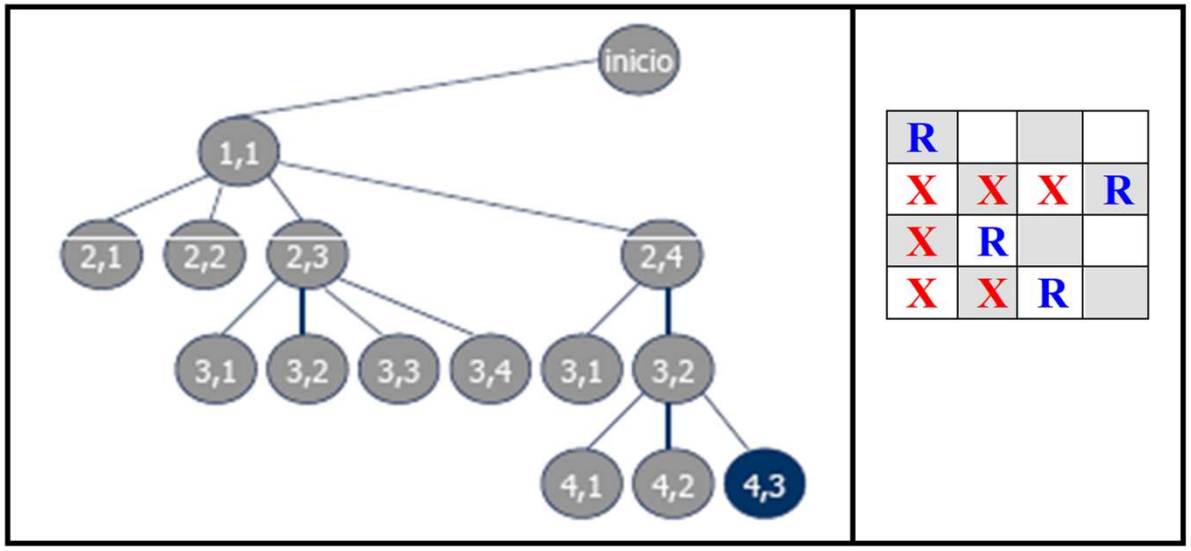




4 – Ejemplos: Otros ejemplos.

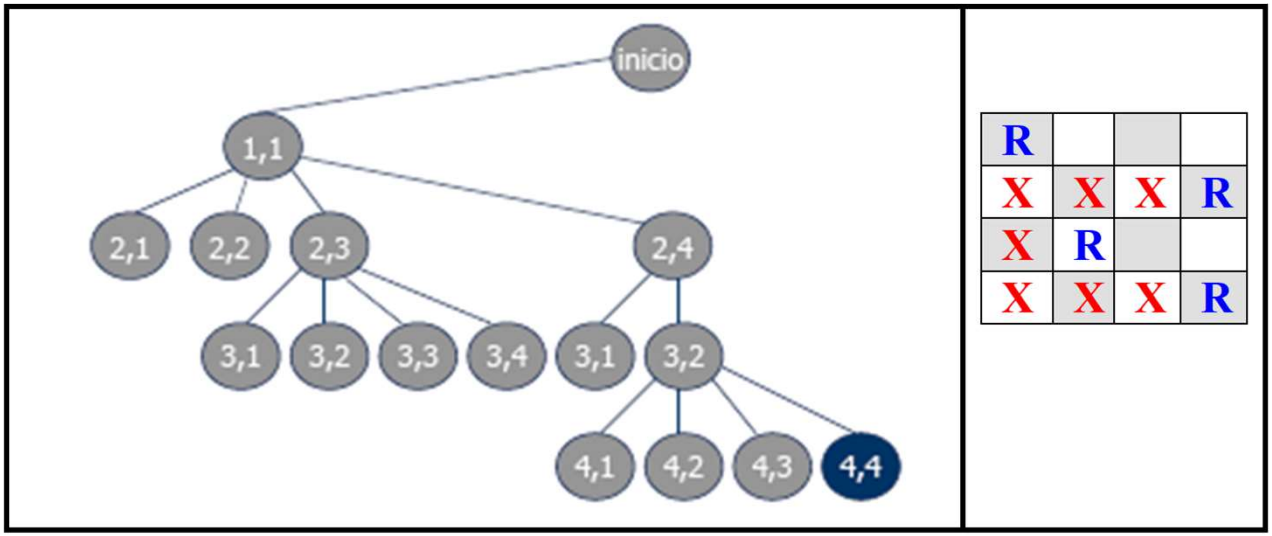
El problema de las n reinas. Ejemplo 4. Generación de estados

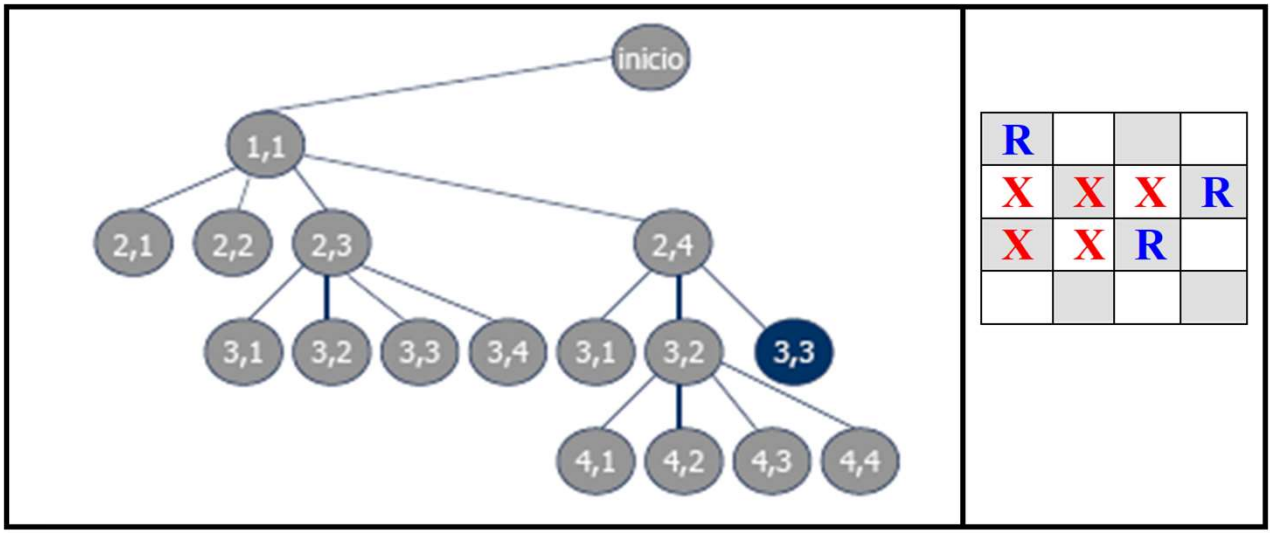




4 – Ejemplos: Otros ejemplos.

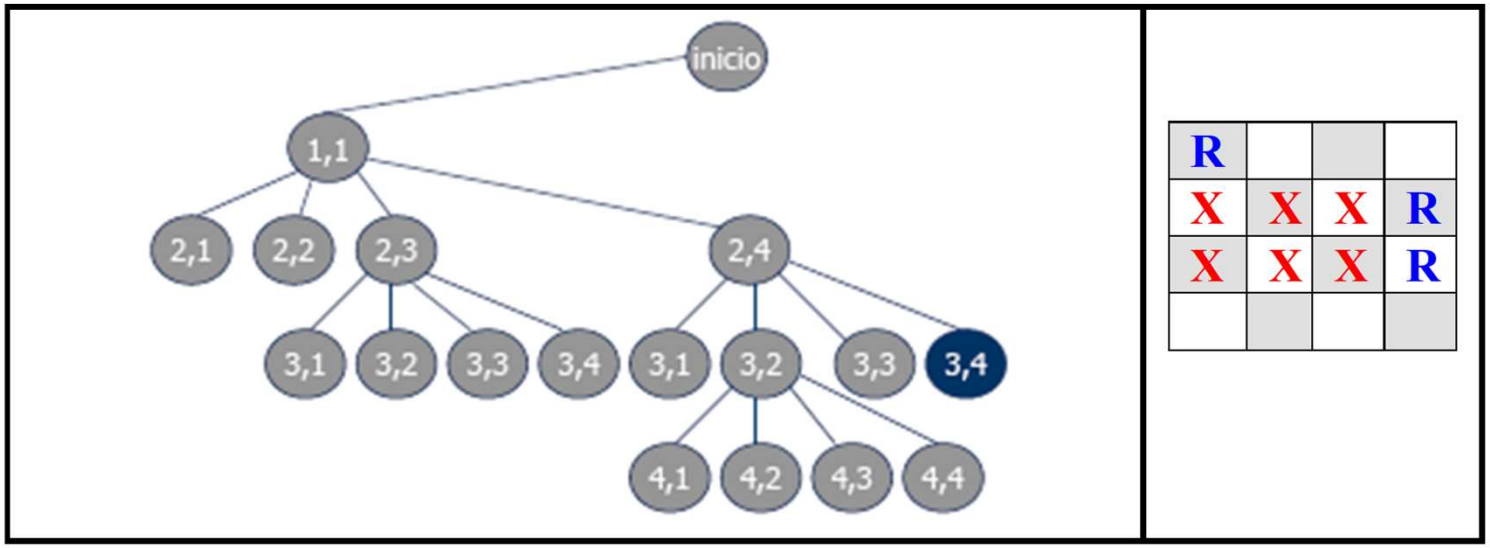
El problema de las n reinas. Ejemplo 4. Generación de estados

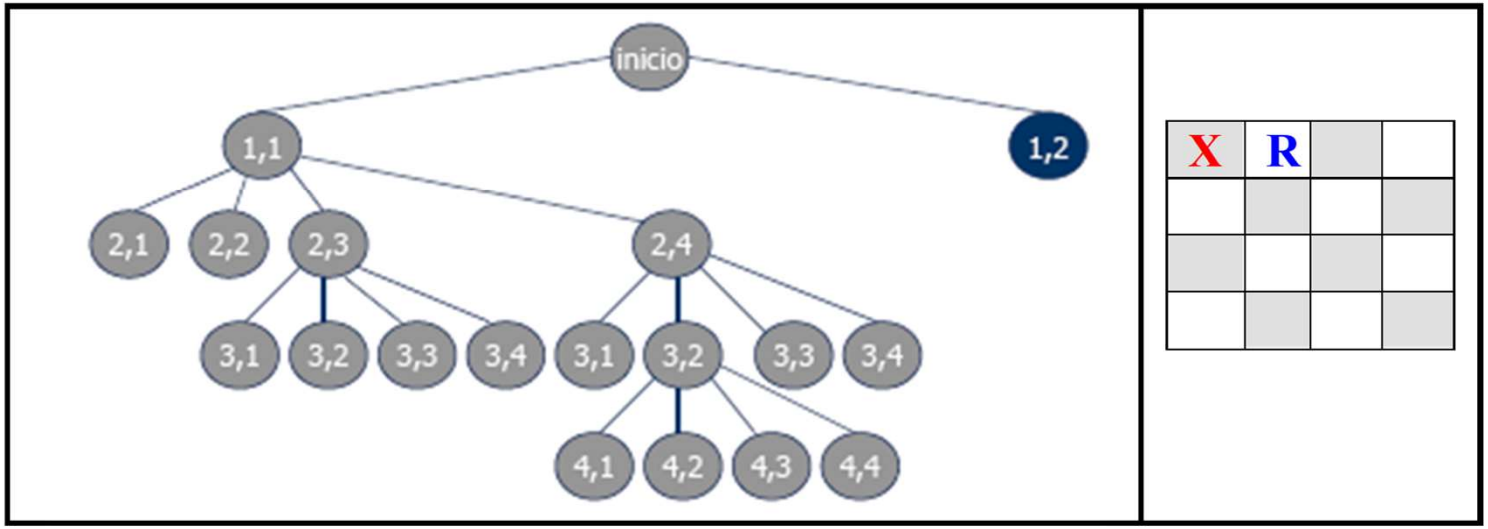




4 – Ejemplos: Otros ejemplos.

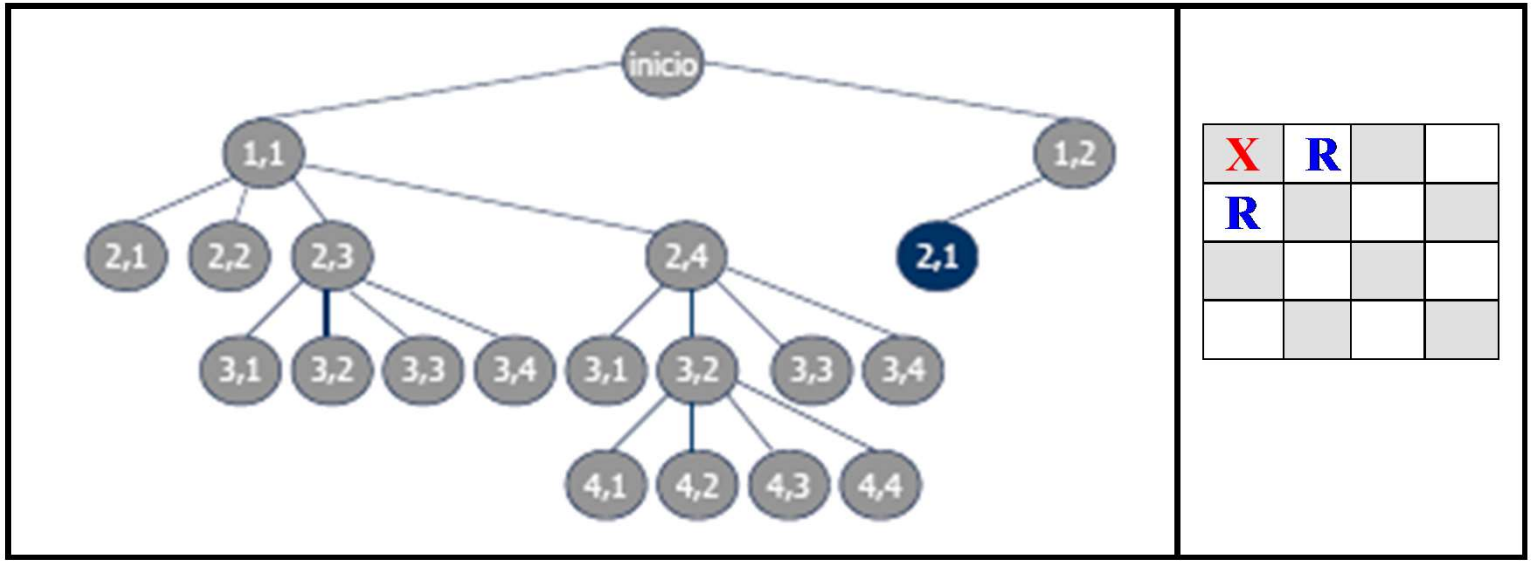
El problema de las n reinas. Ejemplo 4. Generación de estados

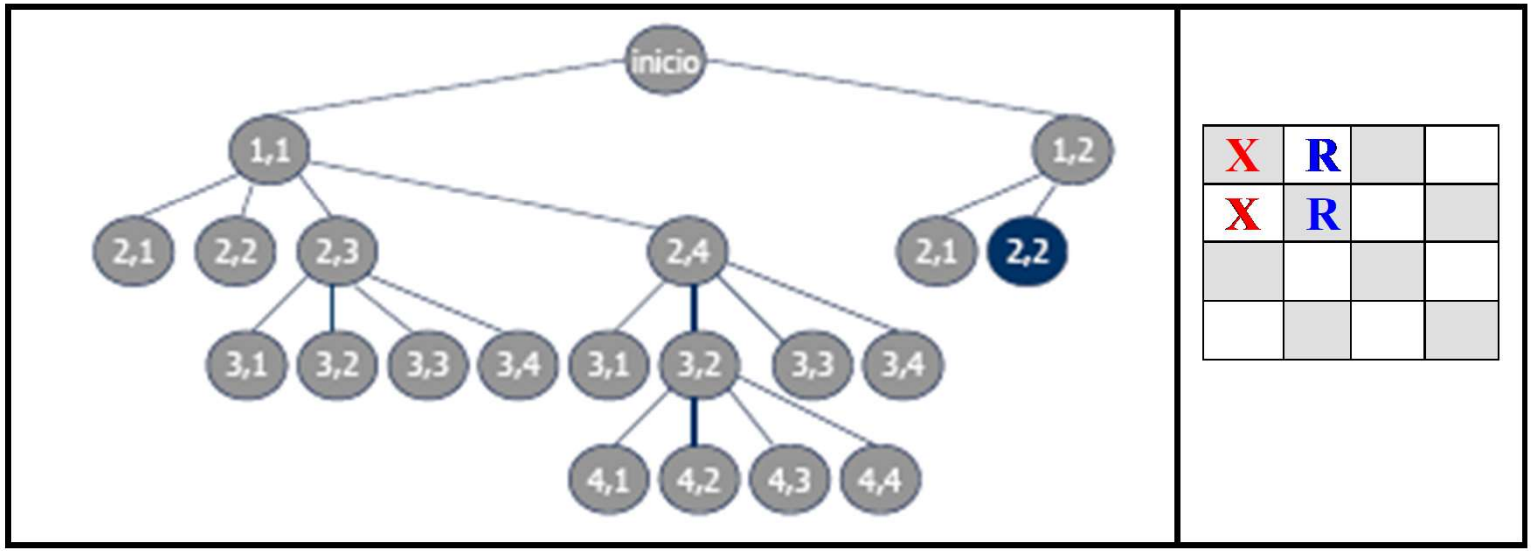




4 – Ejemplos: Otros ejemplos.

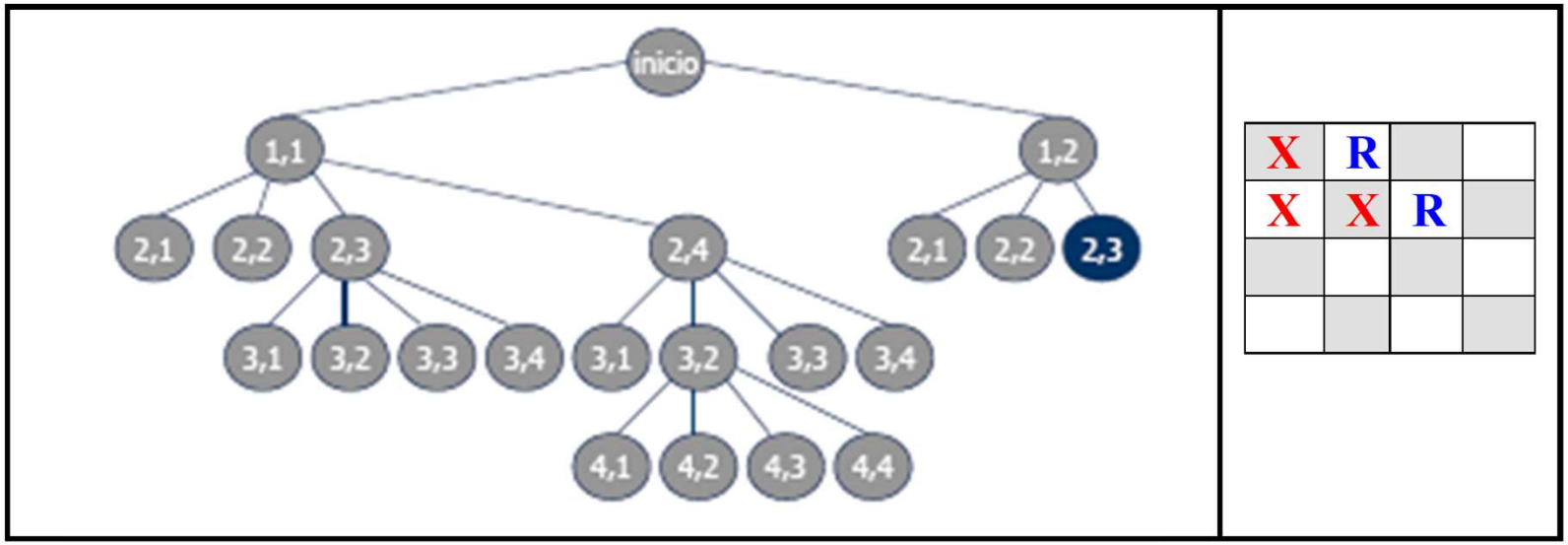
El problema de las n reinas. Ejemplo 4. Generación de estados

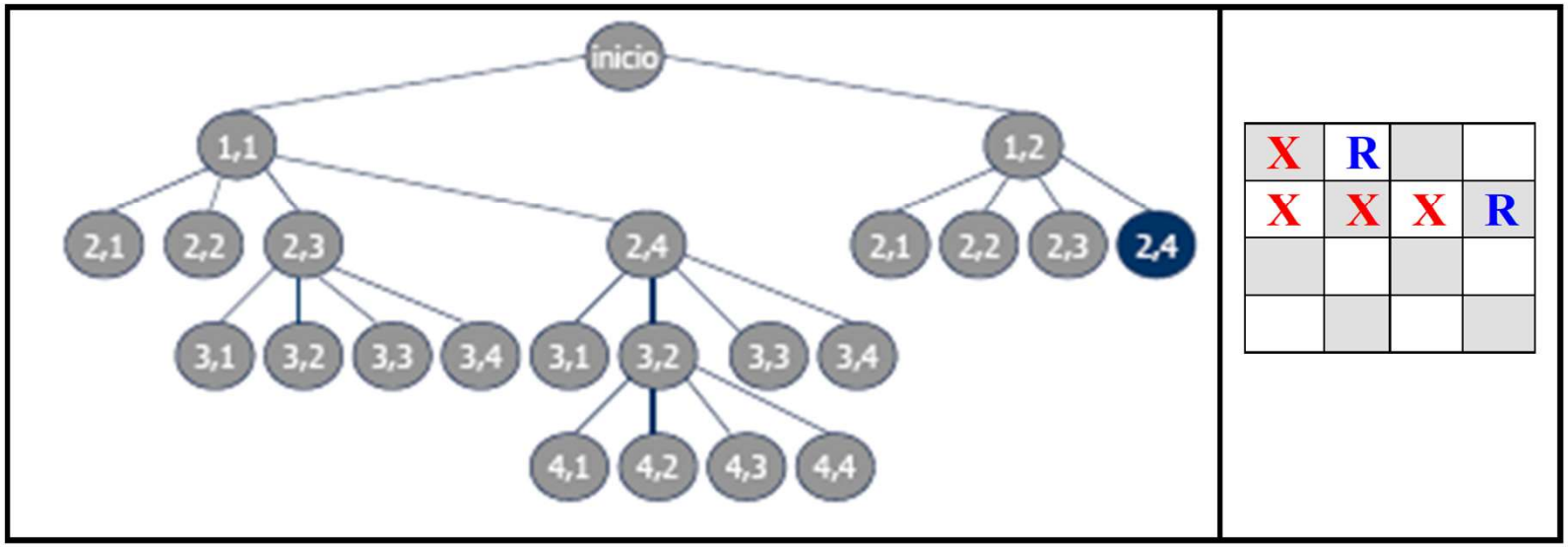




4 – Ejemplos: Otros ejemplos.

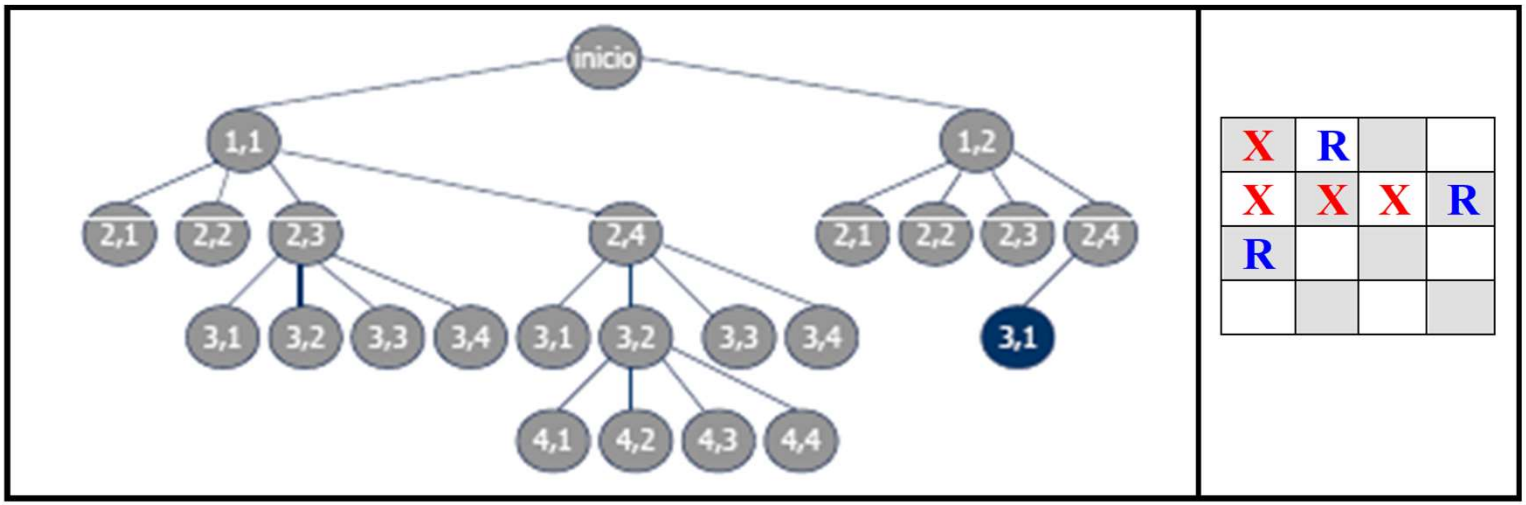
El problema de las n reinas. Ejemplo 4. Generación de estados

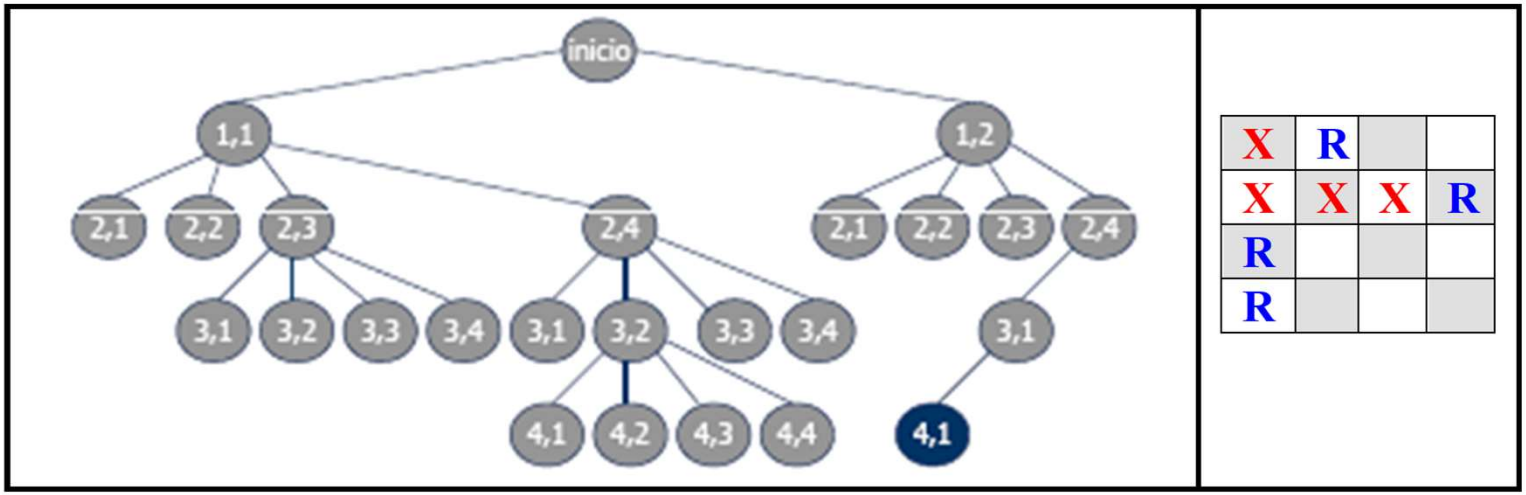




4 – Ejemplos: Otros ejemplos.

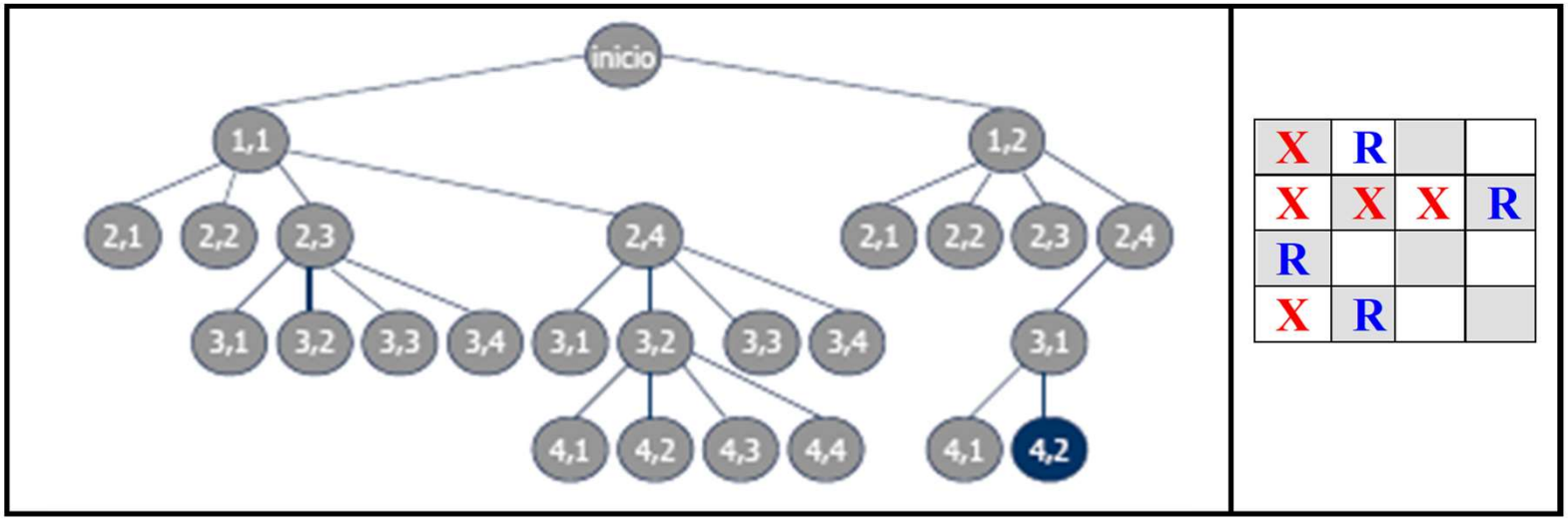
El problema de las n reinas. Ejemplo 4. Generación de estados

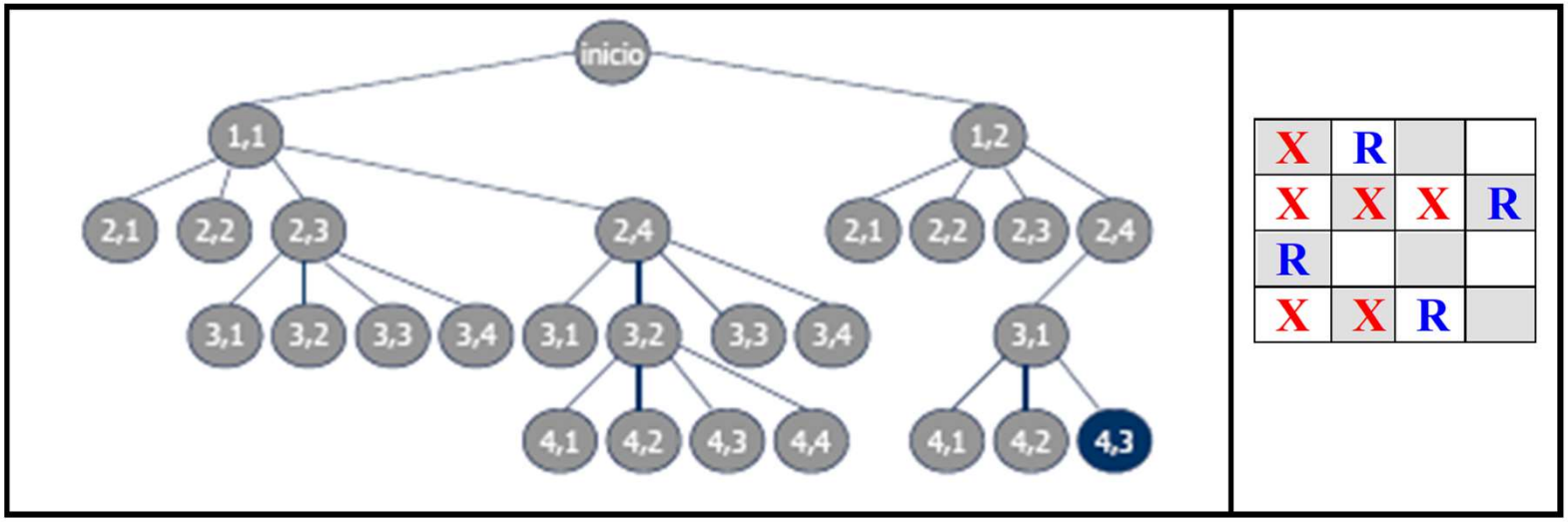




4 – Ejemplos: Otros ejemplos.

El problema de las n reinas. Ejemplo 4. Generación de estados





Solución. Se han probado 27 nodos, sin poda (criterio) serían 155

4 – Ejemplos: Otros ejemplos.

El problema de las n reinas: representación de estados

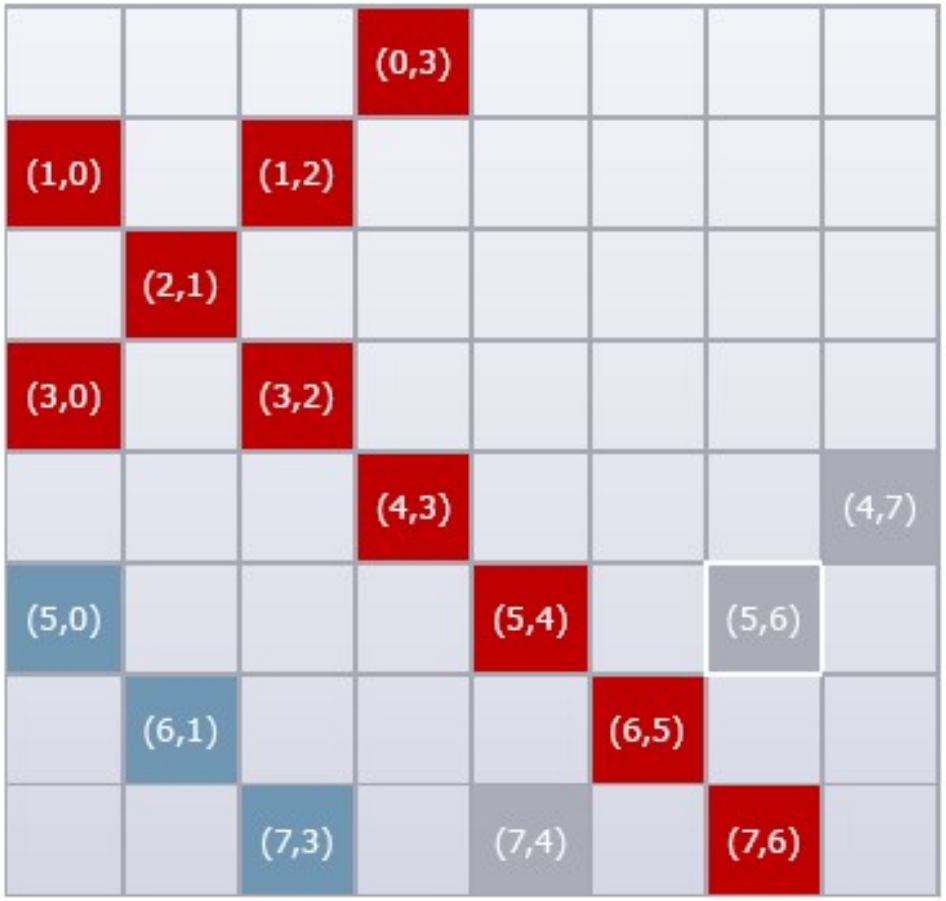
- Tablero NxN en el que colocar N reinas que no se ataquen.
- Solución: $(x_0, x_2, x_3, \ldots, x_{n-1})$, donde x_i es la columna de la i-ésima fila en la que se coloca la reina i (fila i).
- Restricciones implícitas: $x_i \in \{0..1\}$.
- Restricciones explícitas: No puede haber dos reinas en la misma columna ni en la misma diagonal.
  - Distinta columna: Todos los x_i diferentes.
  - Distinta diagonal: Las reinas (i,j) y (k,l) están en la misma diagonal si i-j=k-l o bien i+j=k+l, lo que se puede resumir en:

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

En términos de los x_i, tendremos: $|x_k - x_i| = |k-i|$.

4 – Ejemplos: Otros ejemplos.

El problema de las n reinas: representación de estados



$|x_k - x_i| = |k - i|$

- El espacio de soluciones se reduce a n! elementos, teniendo en cuenta que todas ellas han de ser permutaciones de (1, 2, …, n), dado que todas las x_i han de ser distintas.

4 – Ejemplos: Otros ejemplos.

El problema de las n reinas: esquemas algorítmicos

```
// Inicialmente, k=0 y reinas[i]=-1.
método NReinas (int reinas[], int n, int k)
    si (k == n)       // Solución (no quedan reinas por colocar)
        Escribir(reinas, n);
    sino              // Aún quedan reinas por colocar (k<n)
        for (reinas[k]=0; reinas[k]<n; reinas[k]++)
            si (comprobar(reinas,n,k))
                NReinas(reinas, n, k+1);
            fsi
        ffor
    fsi
fmétodo
```

Implementación recursiva mostrando todas las soluciones

```
Método comprobar (int reinas[], int n, int k): boolean
    int i;
    for (i=0; i<k; i++)
        si ( ( reinas[i]==reinas[k] ) ||
             ( abs(k-i) == abs(reinas[k]-reinas[i]) ) )
                return false;
        fsi
    ffor
    return true;
fMetodo
```

4 – Ejemplos: Otros ejemplos.

El problema de las n reinas: esquemas algorítmicos

Implementación recursiva mostrando la primera solución

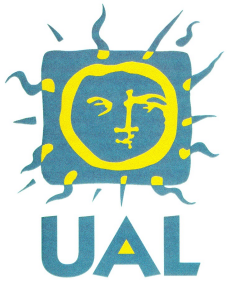
```
// Inicialmente, k=0 y reinas[i]=-1.
método NReinas (int reinas[], int n, int k): boolean
    boolean ok = false
    si (k == n)       // Solución (no quedan reinas por colocar)
         ok = true
         Escribir(reinas, n)
    sino              // Aún quedan reinas por colocar (k<n)
        mientras ((reinas[k]<n-1) && !ok))
            si (comprobar(reinas,n,k))
                ok = NReinas(reinas, n, k+1)
            fsi
        fmientras
    fsi
    return ok
fmétodo
```

4 – Ejemplos: Otros ejemplos.

El problema de las n reinas: esquemas algorítmicos

```
método NReinas (int reinas[], int n)
    int k=0;                    // Fila actual = k
    desde(int i=0; i<n; i++) reinas[i]= -1;   // Configuración inicial
    mientras (k>=0)
        reinas[k]++;        // Colocar reina k en la siguiente columna…
        mientras ((reinas[k]<n) && !comprobar(reinas,n,k))
            reinas[k]++;
        fmientras
        si (k<n) // Reina colocada
            si (k == n-1)
                Escribir(reinas,n);          // Solución
            sino
                k++; reinas[k] = -1          // Siguiente reina
            fsi
        sino
            k--;            // Marcha atrás
        fsi
    fmientras
fmétodo
```

Implementación iterativa mostrando todas las soluciones



4 – Ejemplos: Otros ejemplos.

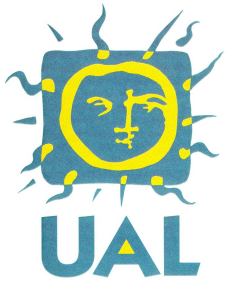
El problema de las n reinas: evaluación eficiencia

Eficiencia temporal (recursivo – todas las soluciones):

- longitud de la solución: N (número de reinas)
- alternativas en cada etapa: N (número de columnas)
- *eficiencia (cota superior)*: $O(N^N)$

Evaluación de la eficiencia real (media):

- La evaluación es compleja, ya que el tiempo de ejecución en cada nodo no es constante y el número de nodos generado es difícil de predecir.
- La cota superior está muy alejada del número real de nodos generados.
- Solución: Estimación de la eficiencia por probabilidad. Hacemos un cálculo aproximado del número de nodos esperado.



4 – Ejemplos: Otros ejemplos.

El problema de las n reinas: evaluación eficiencia

Estimación de la eficiencia por probabilidad:

- Generamos varias permutaciones de (1, 2, ..., n), de forma aleatoria.
- Para cada una calcular el nivel al que llegaría (aplicando la función Criterio), y el número de nodos máximo para ese nivel.
- Hacer una media del número de nodos.

4 – Ejemplos: Otros ejemplos.

Coloreado (m-colorabilidad) de un grafo

Sean **G** un grafo y **m** un entero positivo.

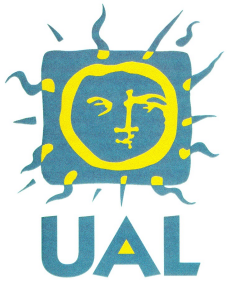
Problema: ¿Los nodos de G pueden colorearse de forma que no haya dos adyacentes con el mismo color, usando sólo m colores?

➢*Menor* m con que G puede colorearse: su *número cromático*.

Problema de ***decisión***: Dados G y m, ¿es G m-coloreable?
Problema de ***optimización***: Dado G, ¿cuál es su número cromático?

Vector solución: $(x_1, ..., x_n)$ tal que x_i representa el color del vértice i, con $x_i \in \{1..m\}$.

El espacio de estados subyacente es un árbol de **grado m** y **altura n+1**, en el que cada nodo del nivel i tiene m hijos (las m posibles asignaciones para x_i) y en el que los nodos del nivel n+1 son nodos hoja.



4 – Ejemplos: Otros ejemplos.

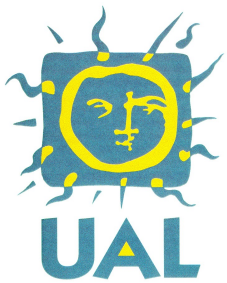
Coloreado (m-colorabilidad) de un grafo. Implementación.

```
// x[1..k-1] almacena los colores asignados a los vértices 1..k-1
// Inicialmente, x[i] = 0
método colorea (m, k)
        si (k=N)
                escribir (x[1..N]) // Solución con m colores
        sino
                hacer
                        x[k] = siguienteValor(m,k)
                        si (x[k]<>0)
                                colorea(m, k+1)
                        fsi
                mientras (x[k]<>0)
        fsi
fmétodo
```

4 – Ejemplos: Otros ejemplos.

Coloreado (m-colorabilidad) de un grafo. Implementación.

```
// siguienteValor(k): siguiente color válido para nodo k (0 si no quedan).
método siguienteValor (m,k):entero
    x[k]= x[k]+1                    // Siguiente color
    mientras (x[k]<=m) hacer
        conflictos = 0
        para i desde 0 hasta k-1 hacer
            si ( G[i][k]<>0 Y x[i]=x[k] )
                conflictos=conflictos+1  //Vértice adyacente del mismo color
            fsi
        fpara
        si (conflictos == 0)
                return x[k]
        sino
                x[k]=x[k]+1
        fsi
    fmientras
    return 0      // Ya se han probado los m colores permitidos.
fmétodo
```



4 – Ejemplos: Otros ejemplos.

Coloreado (m-colorabilidad) de un grafo. Eficiencia.

Nodos internos del árbol de estados: $\mathbf{\Sigma_{i=0..n-1}\ m^i}$

En cada nodo interno, se llama a la función siguienteValor(m, k), que es de orden **O(n·m)**.

El **tiempo total de ejecución** del algoritmo está acotado por:

$$\mathbf{\Sigma_{i=1..n}\ n \cdot m^i = n(m^{n+1}-1)/(m-1) \in O(n \cdot m^n)}$$

Por tanto, el algoritmo es $\mathbf{O(n \cdot m^n)}$.

Hay que recordar que existe una **heurística voraz** muy eficiente.

4 – Ejemplos: Otros ejemplos.

Coloreado (m-colorabilidad) de un grafo.

Un subproblema clásico:

Dado un mapa, ¿pueden pintarse sus regiones, autonomías, países, etc. de tal forma que no haya dos regiones adyacentes de igual color y no se empleen más de 4 colores?

- Cada región se modela con un nodo, y si dos regiones son adyacentes, sus correspondientes nodos se conectan con un arco.

- Desde hace muchos años, se sabía que 5 colores eran suficientes para pintar cualquier mapa, pero no se había encontrado ningún mapa que requiriera más de 4.

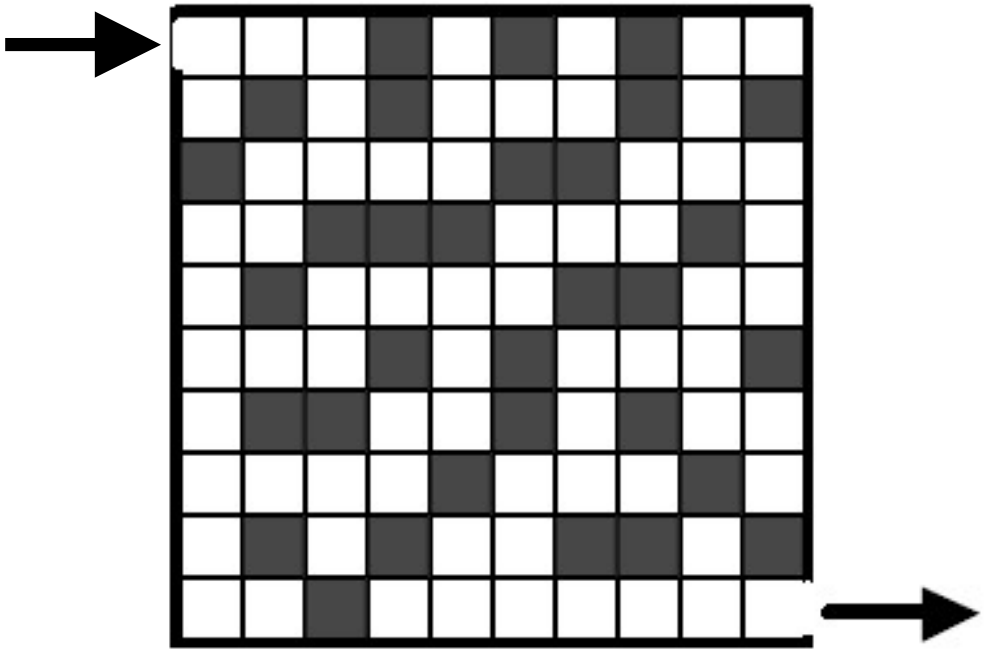
- Se ha demostrado ya que 4 colores siempre son suficientes.

4 – Ejemplos: Otros ejemplos.

Salida de un laberinto.

Laberinto: Matriz de n·n, de casillas *libres* u *ocupadas* por pared.

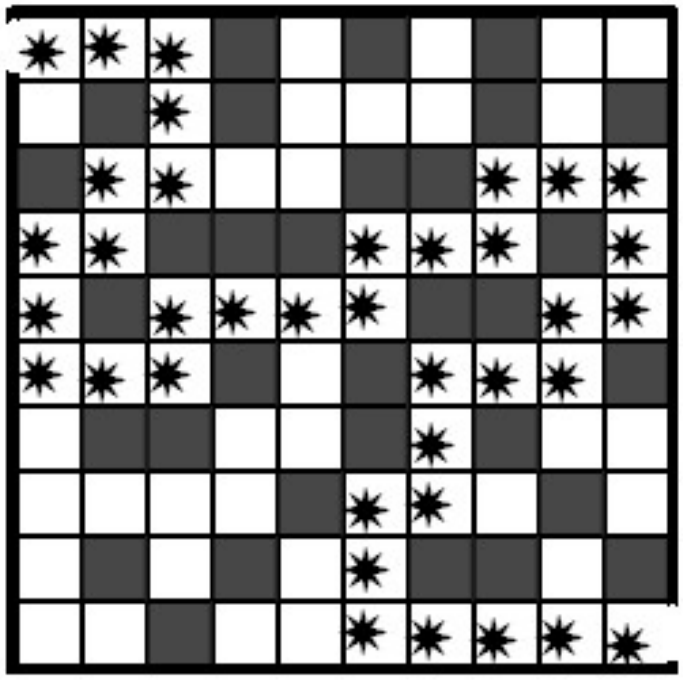
- Es posible pasar de una casilla a otra sólo en vertical u horizontal.
- Se debe ir de la casilla (1, 1) a la (n, n). – Este es un caso especial –



4 – Ejemplos: Otros ejemplos.

Salida de un laberinto.

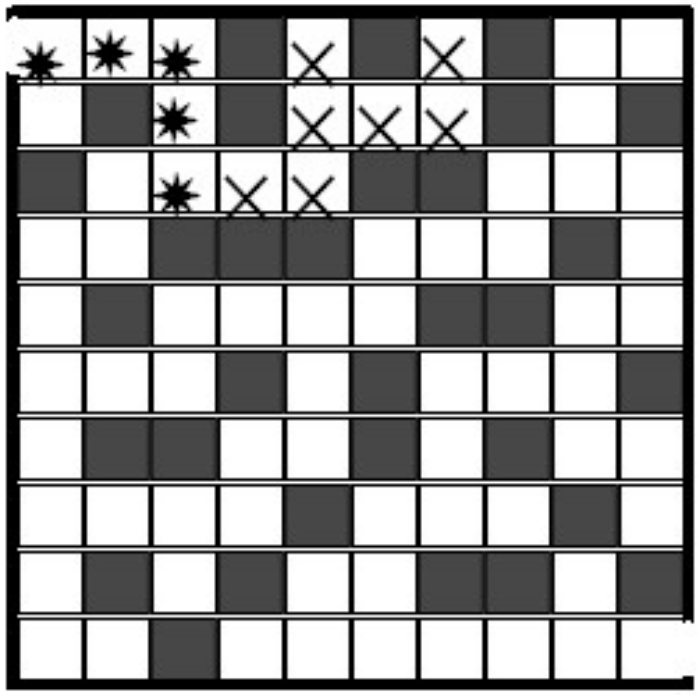
Algoritmo con **backtracking**, donde se marcará en la misma matriz del laberinto un camino solución (si existe):

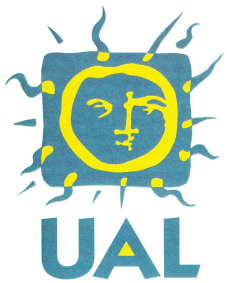


4 – Ejemplos: Otros ejemplos.

Salida de un laberinto.

- Si por un camino recorrido se llega a una casilla desde la que es imposible encontrar una solución, hay que volver atrás y buscar otro camino.
- Además hay que marcar las casillas por donde ya se ha pasado para evitar meterse varias veces en el mismo callejón sin salida, dar vueltas alrededor de columnas…





4 – Ejemplos: Otros ejemplos.

Salida de un laberinto.

Tipos de datos:

casilla = (libre, pared, camino, imposible)

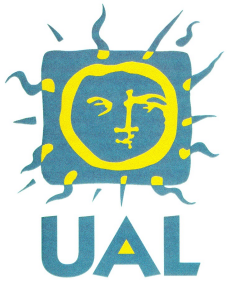
laberinto = vector[1..n,1..n] de casilla

Solución con **backtracking**:

```
método buscarCamino(laberinto lab): booleano
        return ensayar(1, 1, lab)
fmétodo
```

4 – Ejemplos: Otros ejemplos. Salida de un laberinto.

```
método ensayar(x,y:entero; lab:laberinto): booleano
  boolean encon                          // indica si encontrado camino
  si ((x<1) ∨ (x>n) ∨ (y<1) ∨ (y>n))     // pos. fuera del laberinto
      return falso
  sino
    si (lab[x,y] ≠ libre)                // posición no libre, no se puede seguir.
          return falso
    sino
          lab[x,y]=camino
          si ((x=n) ∧ (y=n))             //se ha encontrado una solución
                  return true
         sino
             encon = ensayar(x+1,y,lab)
             si !encon entonces ensayar(x,y+1,lab,encontrado) fsi
             si !encon entonces ensayar(x-1,y,lab,encontrado) fsi
             si !encon entonces ensayar(x,y-1,lab,encontrado) fsi
             si !encon entonces lab[x,y]=imposible fsi  // Marca no salida
         fsi
    fsi
  fsi
fmétodo
```



4 – Ejemplos: Otros ejemplos.

Salida de un laberinto. Eficiencia.

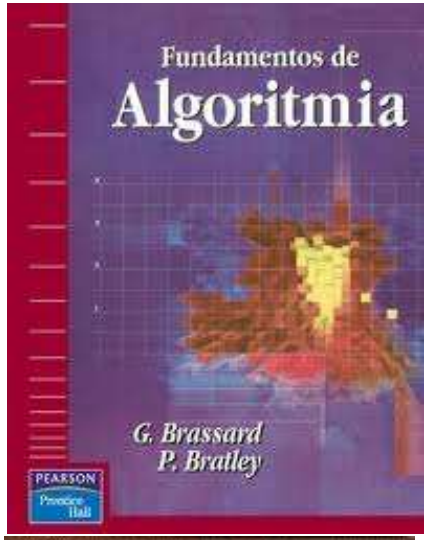
La eficiencia **temporal** depende del número de posiciones del laberinto que sean parte del camino (C):

- Máxima longitud de la solución: C.
- Alternativas en cada etapa: 3 (como máximo, menos en la 1ª que podrían ser 2 – para nuestro caso especial de salida de 1,1 –).
- Eficiencia: $O(3^C)$.
- C está acotado por el tamaño del laberinto (filas×columnas).

Eficiencia **espacial**:

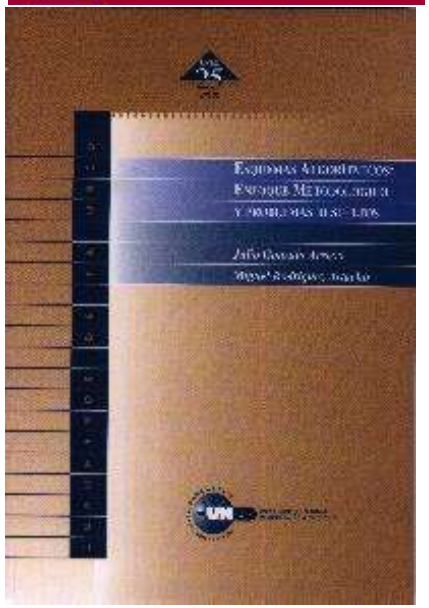
- Laberinto (tabla filas×columnas): O(filas×columnas)
- Variables auxiliares: O(1)
- Como máximo C llamadas recursivas anidadas: O(C)
- Eiciencia: O(filas×columnas)

Material a estudiar



Fundamentos de algoritmia
G. Brassard, T. Bratley
Prentice Hall, D.L. 2004

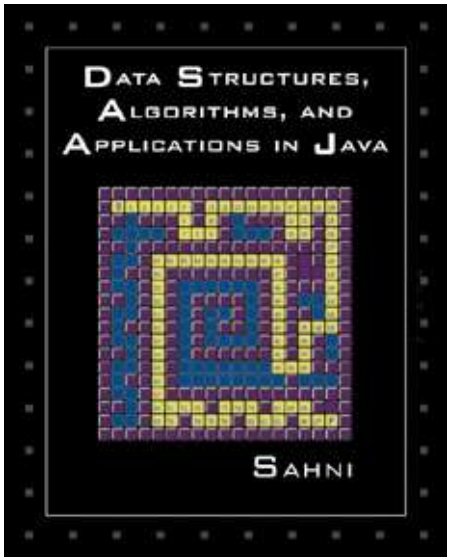
Biblioteca: 519 BRA fun
Capítulo 9 (apartados 9-1 a 9-6.)

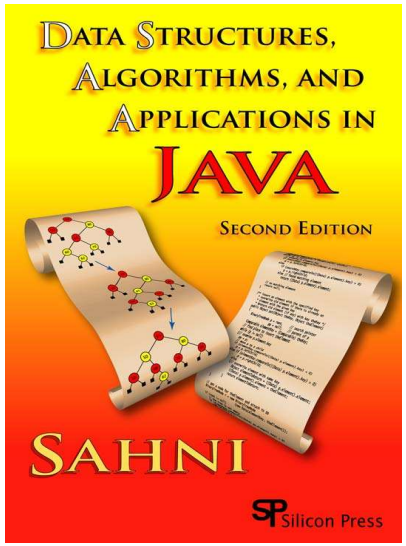


Esquemas algorítmicos. Enfoque metodológico y problemas resueltos
Julio Gonzalo Arroyo y Miguel Rodríguez Artacho)
ISBN: 84-362-3622-X

Biblioteca: 519 GON esq
Capítulo 4 (sobre todo para ejemplos.)

Material a estudiar





Data Structures, Algorithms, and Applications in Java

Sartaj Sahni

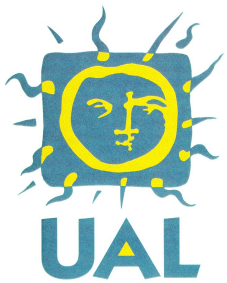
McGraw Hill, 2000 / Silicom Press, 2005

Biblioteca: Agotado – Esperando 2ª edición. Capítulo 21.

Ejercicios recomendados - Libres

1. Construir un algoritmo que compruebe si un grafo es cíclico, utilizando como punto de partida el esquema algorítmico de búsqueda exhaustiva en grafos.

- Presentar el esquema algorítmico detallado (pseudocódigo).
- Seleccionar las estructuras de datos adecuadas para almacenar los datos.
- Implementar el correspondiente código en Java.
- Evaluar experimentalmente la eficiencia utilizando diversos juegos de prueba.



Ejercicios recomendados - Libres

2. Modificar el algoritmo de asignación de tareas teniendo en cuenta los siguientes factores:
   a/ No todos los empleados pueden realizar cualquier tarea (necesitan de una cualificación o permiso especial).
   b/ Puede haber más tareas que personas, se elige aquella combinación que ofrece un mejor rendimiento (global), dejando pendientes las que menor beneficio generen (en conjunto.)

- Presentar el esquema algorítmico detallado (pseudocódigo).
- Seleccionar las estructuras de datos adecuadas para almacenar los datos.
- Implementar el correspondiente código en Java.
- Evaluar experimentalmente la eficiencia utilizando diversos juegos de prueba.

Ejercicios recomendados - Libres

3. Modificar el algoritmo de salida de un laberinto, de forma que el tablero sea rectangular (n x m), punto de salida sea cualquier punto del tablero y el de salida sea cualquier punto del borde del tablero.

- Presentar el esquema algorítmico detallado (pseudocódigo).
- Seleccionar las estructuras de datos adecuadas para almacenar los datos.
- Implementar el correspondiente código en Java.
- Evaluar experimentalmente la eficiencia utilizando diversos juegos de prueba.