Estrategias algorítmicas

Tema 3(III)

Algorítmica y Modelos de Computación

1. Introducción.
2. Algoritmos divide y vencerás.
3. Algoritmos voraces.
4. Programación dinámica.
5. Algoritmos Bactracking (vuelta atrás).
6. Ramificación y poda.

5. Algoritmos vuelta atrás (Bactracking).

1. Introducción. Características generales.
2. Esquema general.
3. Análisis de tiempos de ejecución.
4. Ejemplos de aplicación.

   4.1. Problema de la mochila 0-1.

   4.2. Problema de la asignación.

   4.3. Resolución de juegos.

5. Algoritmos Bactracking. Introducción. Características generales.

- El **backtracking** (o método de **retroceso** o **vuelta atrás**) es una técnica general de resolución de problemas aplicable en problemas de **optimización**, **juegos** y otros tipos.
- Las técnicas vistas hasta ahora intentan construir la solución basándose en ciertas propiedades de esta. Sin embargo, ciertos problemas no pueden solucionarse con ninguna de las técnicas anteriores: la única manera de resolver estos problemas es a través de un **estudio exhaustivo** de un conjunto de posibles soluciones.
- La técnica de backtracking permite realizar este estudio exhaustivo
- El **backtracking** realiza una **búsqueda exhaustiva y sistemática en el espacio de soluciones**. Por ello, suele resultar muy ineficiente.
- Se puede entender como “opuesto” a avance rápido:
  - **Avance rápido:** añadir elementos a la solución y no deshacer ninguna decisión tomada.
  - **Backtracking:** añadir y quitar todos los elementos. **Probar todas las combinaciones.**
- Cada **solución es el resultado de una secuencia de decisiones**
  - Pero a diferencia del método voraz, las decisiones pueden deshacerse ya sea porque no lleven a una solución o porque se quieran explorar todas las soluciones (para obtener la solución óptima)

5. Algoritmos Bactracking. Características generales.

- Existe una **función objetivo** que debe ser satisfecha u optimizada por cada selección
- Las etapas por las que pasa el algoritmo se pueden representar mediante un **árbol de expansión** (o ***árbol del espacio de estados***).
- El árbol de expansión no se construye realmente, sino que esta implícito en la ejecución del algoritmo
- Cada **nivel** del árbol representa una etapa de la secuencia de decisiones
- Una **solución** se puede expresar como una tupla: $(\mathbf{x_1, x_2, ..., x_n})$, satisfaciendo unas restricciones y tal vez optimizando cierta función objetivo.
- En cada momento, el algoritmo se encontrará en cierto nivel **k**, con una solución parcial $(x_1, ..., x_k)$.
  - Si se puede añadir un nuevo elemento a la solución $\mathbf{x_{k+1}}$, se genera y se avanza al nivel **k+1**.
  - Si no, se prueban otros valores para $\mathbf{x_k}$.
  - Si no existe ningún valor posible por probar, entonces se retrocede al nivel anterior **k-1**.
  - Se sigue hasta que la solución parcial sea una solución completa del problema, o hasta que no queden más posibilidades por probar.

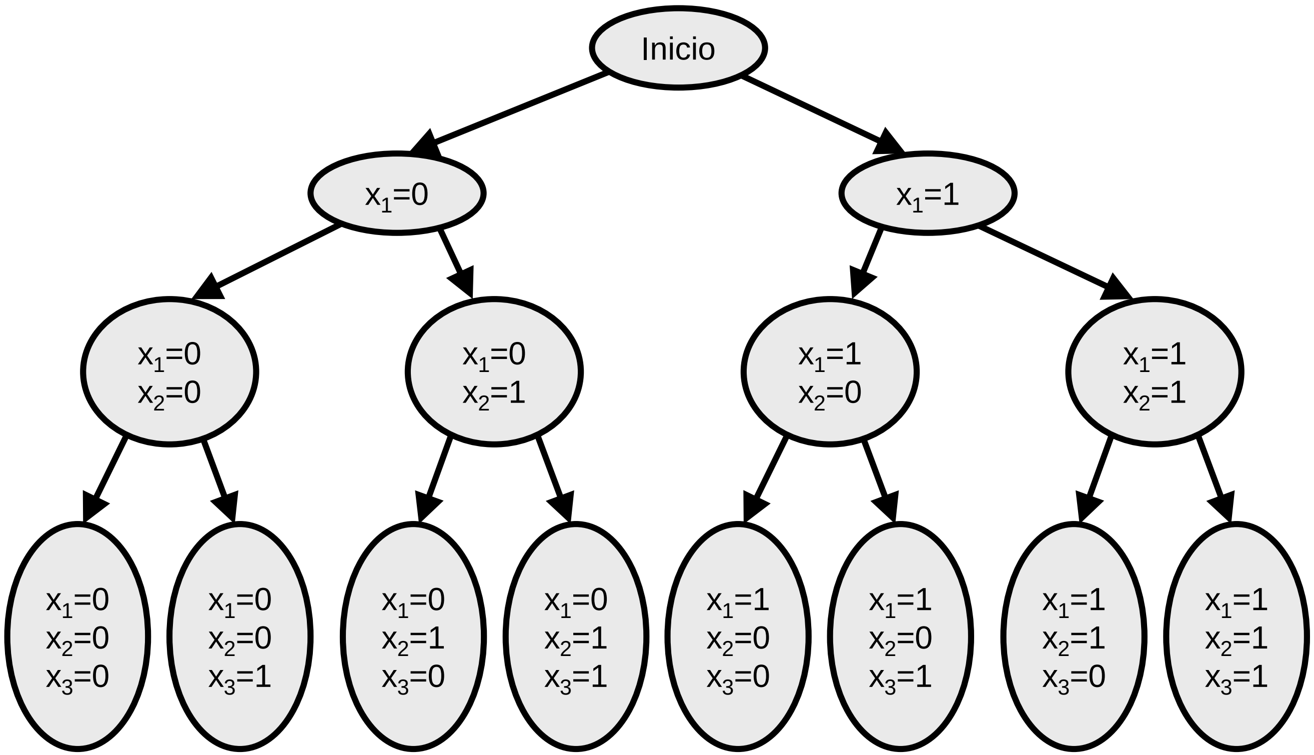
5. Algoritmos Bactracking. Características generales.

- Las **soluciones** del problema se pueden representar como una n-tupla :

  $(\mathbf{x_1, x_2, ..., x_n})$
- El **objetivo** consiste en encontrar soluciones factibles.
- La idea consiste en **construir el vector solución elemento a elemento** usando una **función factible** modificada para estimar si una solución parcial o incompleta tiene alguna posibilidad de éxito.
- Cada una de las tuplas $(\mathbf{x_1, x_2, ..., x_i}; ?)$ donde $i \leq n$ se denomina un **estado** y denota un conjunto de soluciones.
- Un estado puede ser:
  - estado **terminal** o solución: describe un conjunto con un solo elemento;
  - estado **no terminal** o solución parcial: representa implícitamente un conjunto de varias soluciones;
- Se dice que un **estado no terminal** es **factible** o prometedor cuando no se puede descartar que contenga alguna solución factible.
- El conjunto de estados se organiza formando un **árbol de estados**.

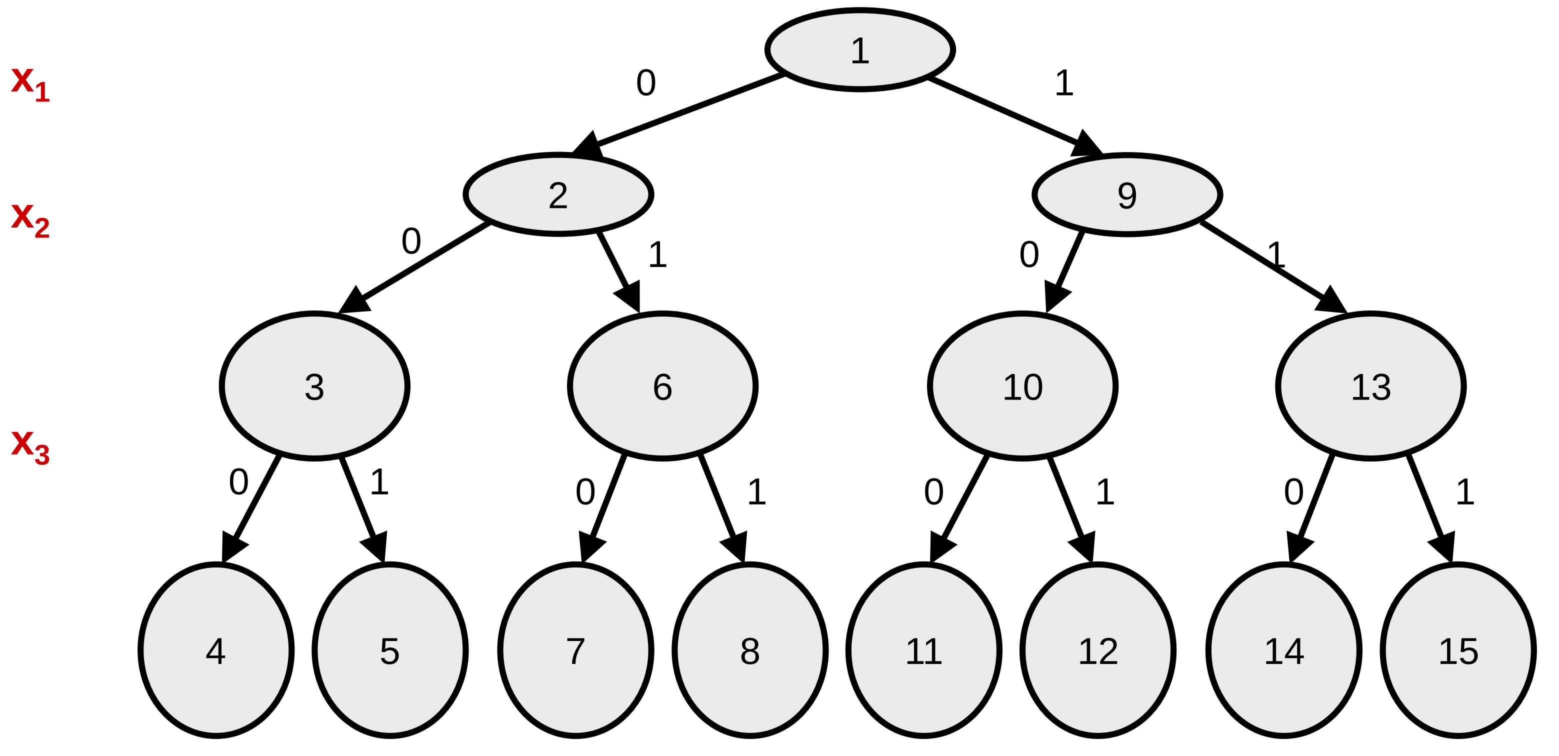
5. Algoritmos Bactracking. Características generales.

- El resultado es equivalente a hacer un **recorrido en profundidad en el árbol de soluciones** (árbol de expansión o árbol del espacio de estados).



5. Algoritmos Bactracking. Características generales.

- **Representación simplificada del árbol.**



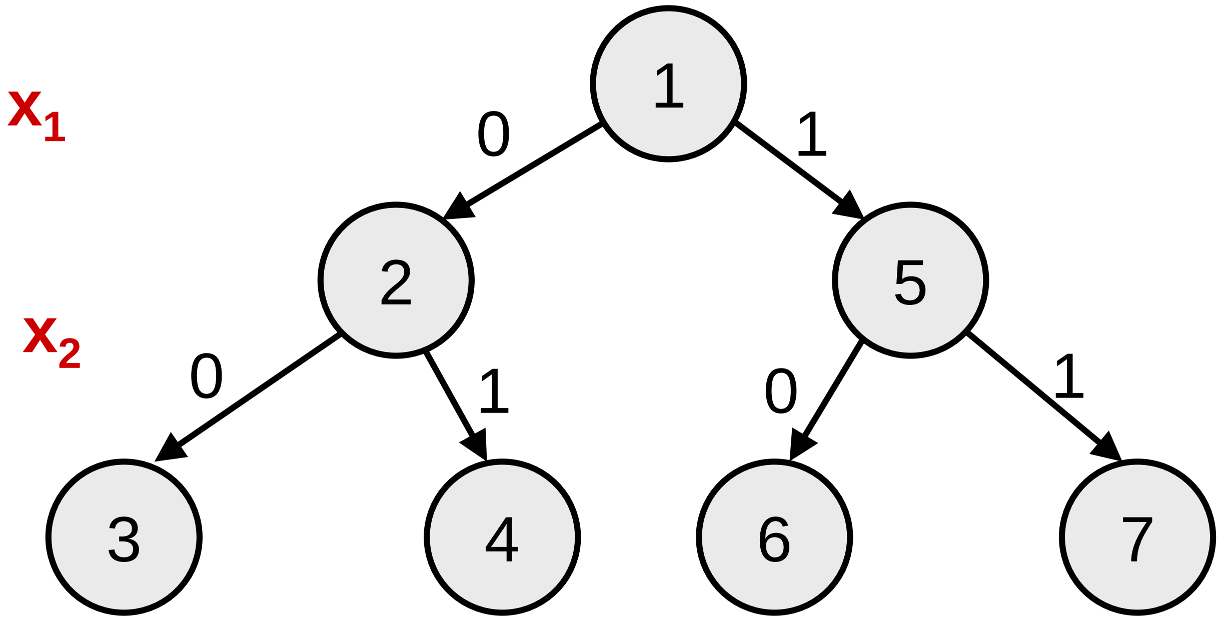
5. Algoritmos Bactracking. Características generales.

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

NOTA: En todos los algoritmos de recorrido de grafos supondremos que el grafo está implementado con listas de adyacencia.

5. Algoritmos Bactracking. Características generales.

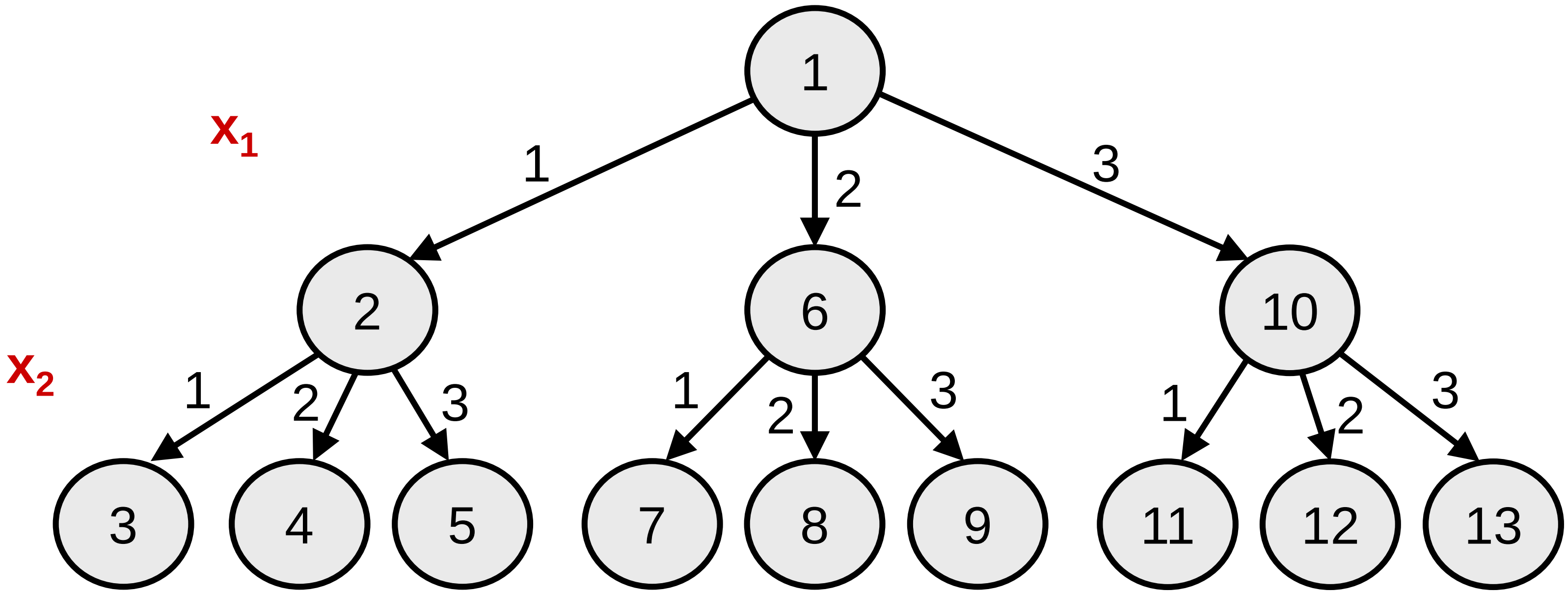
- **Árboles binarios: s**= $(x_1, x_2, ..., x_n)$, con $x_i \in \{0, 1\}$



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

5. Algoritmos Bactracking. Características generales.

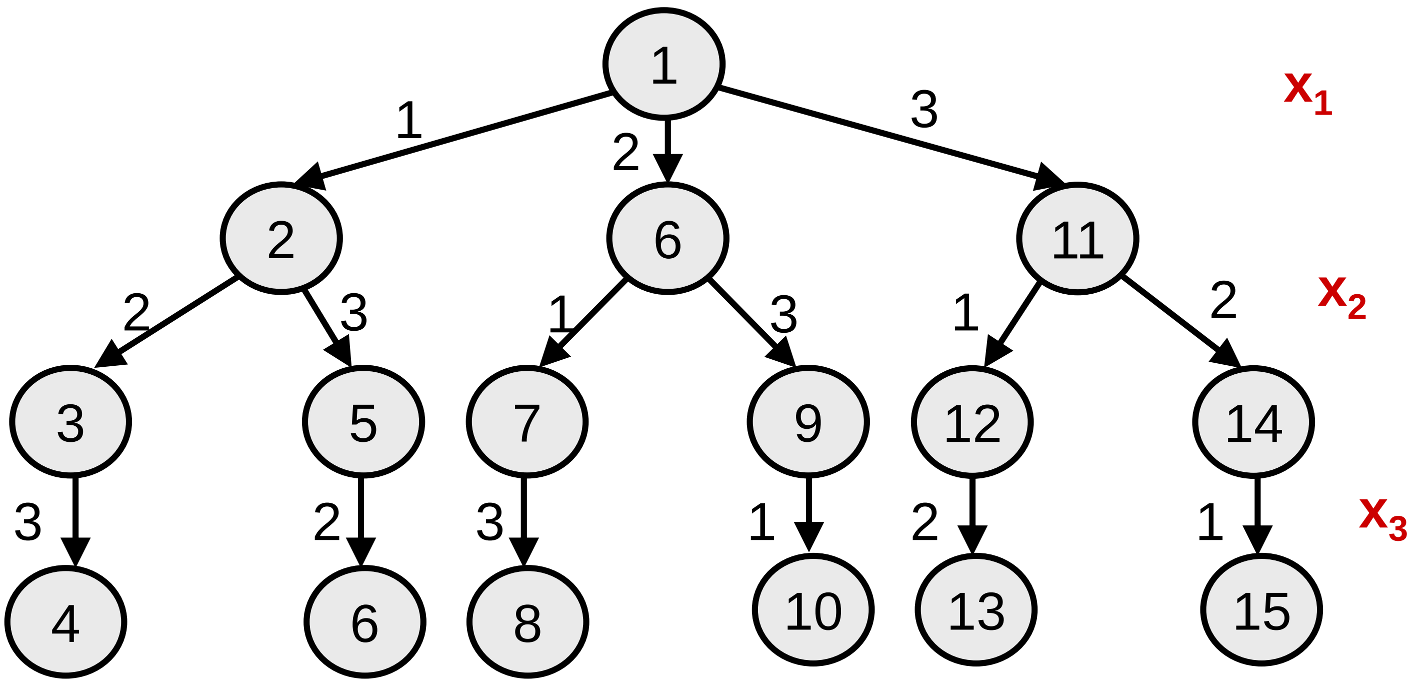
- **Árboles k-arios: s**= $(x_1, x_2, ..., x_n)$, con $x_i \in \{1,..,k\}$



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

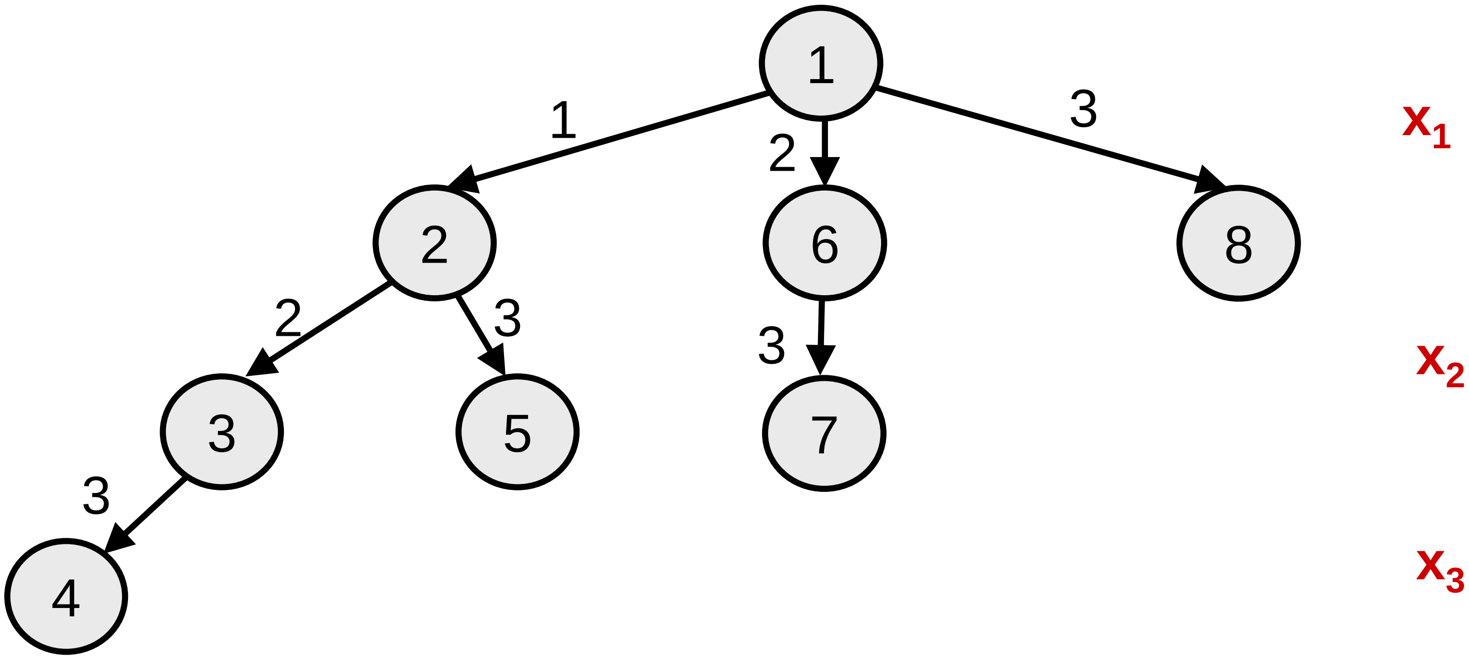
5. Algoritmos Bactracking. Características generales.

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



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

- **Árboles combinatorios: s**= $(x_1, x_2, ..., x_m)$, 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, 0) → Combinatorio: (2, 4, 7)

5. Algoritmos Bactracking. Características generales. Ejemplo de backtracking.

- **Ejemplo:** Diseñar un algoritmo que permita obtener un subconjunto de números dentro del conjunto *datos*= {17, 11, 3} cuya suma sea 20

1. **Representación de la solución**:

   1.1. Tupla o vector de 3 elementos $[x_1, x_2, x_3]$ con $x_i \in \{0, 1\}$

   - Restricciones explicitas: indican que valores pueden tomar los componentes de la solución
     - **x_i = 0** indica que el elemento i **no** esta en la solución
     - **x_i = 1** indica que el elemento i **si** esta en la solución

   - La solución parcial debe cumplir que $\sum_{i=1}^{3} x_i datos_i \le 20$

   - Restricciones implícitas: indican que tuplas pueden dar lugar a soluciones válidas

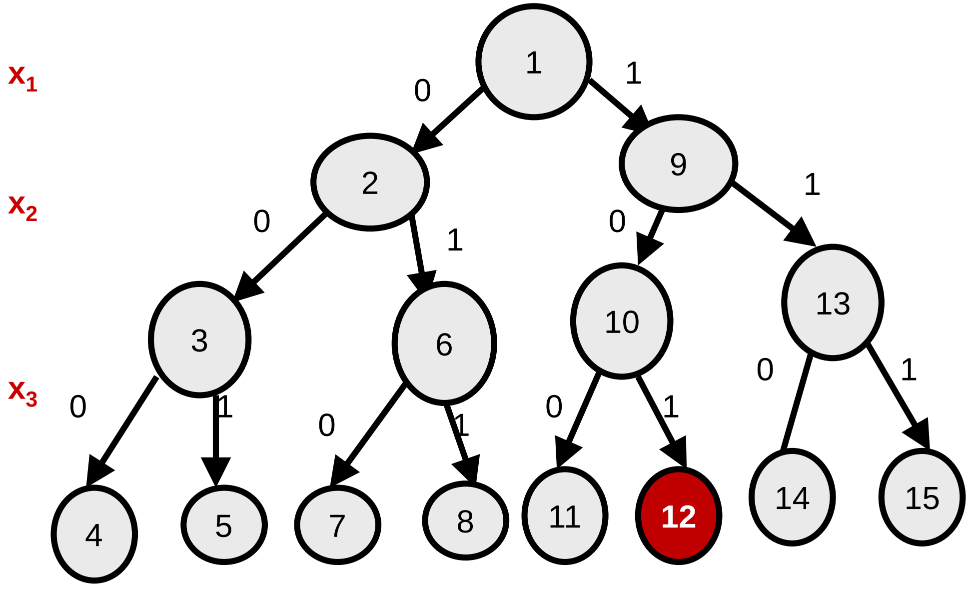
   - La solución debe cumplir el siguiente objetivo: $\sum_{i=1}^{3} x_i datos_i = 20$

   - Para este problema existe una única solución: x= [1, 0, 1]

5. Algoritmos Bactracking. Características generales. Ejemplo.

2. Árbol de expansión. Para la representación elegida existen dos formas posibles:

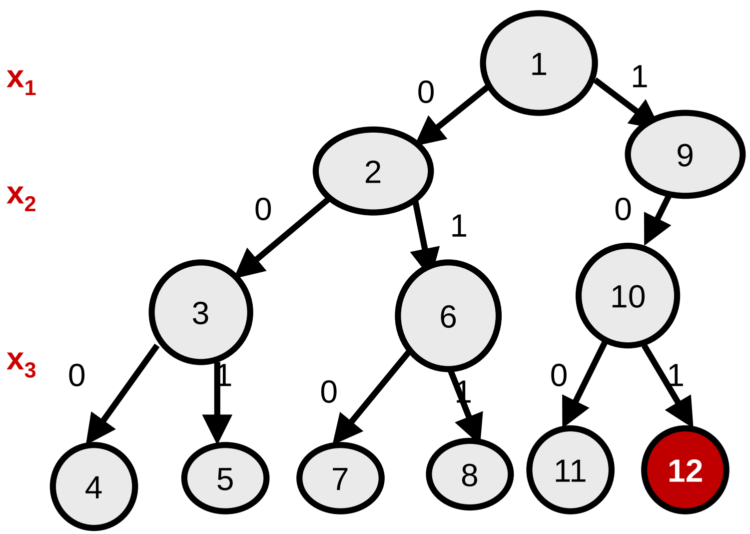
2.1. Árbol binario $\Rightarrow$ Generar todas las combinaciones posibles y escoger aquellas que sean solución



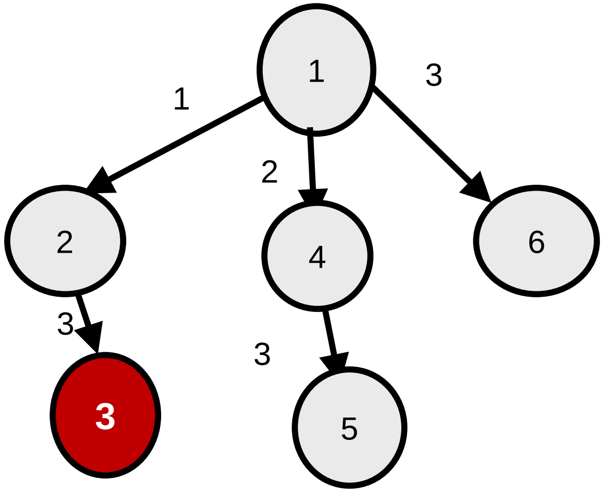
- Cada camino de la raíz a las hojas define una posible solución
- El árbol de expansión tiene 2^3 hojas (8 posibles soluciones)
- Se generan tuplas que no son soluciones $\Rightarrow$ ineficiente

2. Árbol de expansión. (cont.)

2.2. Árbol binario utilizando la técnica de backtracking ⇒ a medida que se construye la tupla, se comprueba si esta puede llegar a ser una solución al problema. En caso negativo, se ignora y se vuelve al estado anterior.



- **1.2.** Otra posible **representación de la solución** con árbol combinatorio:
  - Tupla o vector de a lo sumo 3 elementos ordenados con valores entre 1 y 3 (los índices de los elementos del conjunto anterior)
    - 1 indica que el elemento 1 (con valor 17) esta en la solución
    - 2 indica que el elemento 2 (con valor 11) esta en la solución
    - 3 indica que el elemento 3 (con valor 3) esta en la solución
  - Para este problema existe una única solución: [1, 3]



5. Algoritmos Bactracking. Esquema general.

Cuestiones a resolver antes de programar:

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

5. Algoritmos Bactracking. Esquema general.

- La **técnica de backtracking** es un **recorrido en profundidad (preorden) del árbol de expansión**

1. En cada momento el algoritmo se encontrara en un cierto **nivel K**
2. En el nivel K se tiene una **solución parcial (x_1, ..., x_k),**
3. Se comprueba si se puede **añadir** un nuevo elemento x_{k+1} a la solución
   - En caso afirmativo, **(x_1, ..., x_{k+1})** es prometedor $\Rightarrow$ se genera la solución parcial **(x_1, ..., x_{k+1})** y se avanza al nivel **K+1**
   - En otro caso $\Rightarrow$ se prueban otros valores de x_k
4. Si ya no existen mas valores para x_k , se retrocede (se **vuelve atrás**-backtrack) al nivel anterior **K-1**
5. El algoritmo continúa hasta que la solución parcial sea una solución completa del algoritmo, o hasta que no queden mas posibilidades

Esquema general de backtracking recursivo

```
función backtrackingRec (solucion[1..n], etapa)
  /* valores es un vector [1..opciones]  */
  IniciarValores(valores,etapa)
  repetir
     nuevovalor := SeleccionarNuevoValor(valores)
     si Alcanzable(nuevovalor) entonces
         AnotarNuevoValor(solucion,nuevovalor)
         si SolucionIncompleta(solucion) entonces
            backtrackingRec(solucion,siguienteEtapa)
         sino si EsSolucion(solucion) entonces
            escribir(solucion)
         fsi
         Desanotar(solucion)
     fsi
  hasta UltimoValor(valores)
ffunción
```

Funciones que aparecen en el esquema recursivo

- **IniciarValores(valores)**
  - Genera todas las opciones del nivel donde se encuentra
- **SeleccionarNuevoValor**(valores)
  - Considera un nuevo valor de los posibles
- **Alcanzable(nuevovalor)**
  - Comprueba si la opción nuevovalor puede forma parte de la solución
- **AnotarNuevoValor(solucion,nuevovalor)**
  - Anota en solución el valor nuevovalor
- **EsSolucion(solucion)**
  - Indica si solución es una solución para el problema
- **Desanotar(solucion)**
  - Elimina la ultima anotación en el vector solución
- **UltimoValor(valores)**
  - Indica si ya no quedan mas nodos por expandir

5. Algoritmos Bactracking. Esquema general NO recursivo

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

```
procedimiento backtracking(var solucion[1..n])
  nivel:=1
  solucion:= solucionINICIAL
  fin:=false
  repetir
    Generar(nivel,solucion)/*solucion[nivel]←generar(nivel,solucion)*/
    si EsSolucion(nivel,solucion) entonces
       fin:=true
    sino
       si Criterio(nivel, solucion) entonces
          nivel:=nivel + 1
       sino mientras not(HayMasHermanos(nivel,solucion)) hacer
              Retroceder(nivel,solucion)
           fmientras
       fsi
    fsi
  hasta fin=true
fprocedimiento
```

5. Algoritmos Bactracking. Esquema general NO recursivo

Variables:

- **s**: Almacena la solución parcial hasta cierto punto.
- $\mathbf{s_{INICIAL}}$: Valor de inicialización.
- **nivel**: Indica el nivel actual en el que se encuentra el algoritmo.
- **fin**: Valdrá **true** cuando hayamos encontrado alguna solución.
- Además, suele ser común utilizar **variables temporales** con el valor actual (beneficio, peso, etc.) de la tupla solución.

Funciones que aparecen en el esquema no recursivo:

- **`Generar(nivel,s)`**
  - Genera el siguiente hermano, o el primero, para el **nivel** actual. Devuelve el valor a añadir a la solución parcial actual. Ejemplo: Generar(1, [0, **-**, **-**]) = 1
- **`EsSolucion(nivel,s)`**
  - Comprueba si la solución calculada hasta el momento(tupla (s[1], ...,s[nivel])) es una solución válida para el problema. Ejemplo: EsSolucion([0,0, 1]) = false
- **`Criterio(nivel,s)`**
  - Comprueba si a partir de la solución parcial actual (s[1], ..., s[nivel]) se puede alcanzar una solución válida. En otro caso se rechazarán todos los descendientes (**poda**). Ejemplos: Criterio(2,[0,0, **-**]) = true; Criterio(2,[1,1;**-**]) = false
- **`HayMasHermanos(nivel,s))`**
  - Devuelve el valor true si el nodo actual tiene hermanos que aun no han sido generados. Ejemplo: HayMasHermanos(2, [1, 1, **-**]) = false
- **`Retroceder(nivel,s))`**
  - Retrocede un nivel en el árbol de soluciones. Disminuye en 1 el valor del **nivel** y y posiblemente tendrá que actualizar la solución actual, quitando los elementos retrocedidos.

5. Algoritmos Bactracking. Esquema general. **Ejemplo (1/3)**

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

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

5. Algoritmos Bactracking. Esquema general. **Ejemplo (2/3)**

- **Funciones**:
  - **Generar (nivel, s)**

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

    si s[nivel]==1 **entonces** tact:= tact + t_{nivel}
  - **EsSolución (nivel, s)**

    devolver (nivel==n) Y (tact==P)
  - **Criterio (nivel, s)**

    devolver (nivel<n) Y (tact≤P)
  - **HayMasHermanos (nivel, s)**

    devolver s[nivel] < 1
  - **Retroceder (nivel, s)**

    tact:= tact – t_{nivel}*s[nivel]

    s[nivel]:= -1

    nivel:= nivel – 1

- **Algoritmo:** (el mismo que el esquema general)

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

5. Algoritmos Bactracking. Variaciones del Esquema general.

- **Variaciones** del esquema general:

1. Si no es seguro que exista una solución
2. Si se quiere almacenar todas las soluciones (no sólo una)
3. Si el problema es de optimización (maximizar o minimizar)

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

Backtracking (var s: TuplaSolución)

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

Para poder generar todo el árbol de backtracking

5. Algoritmos Bactracking. Variaciones del Esquema general: **Caso 2º.**

- **Caso 2)** Se quiere almacenar todas las soluciones.

```
Backtracking (var s: TuplaSolución)
    nivel:= 1
    s:= s_INICIAL
    fin:= false
    repetir
        Generar (nivel, s)
        si EsSolución (nivel, s) entonces
            Almacenar (nivel, s)
        si Criterio (nivel, s) entonces
                nivel:= nivel + 1
        sino
                mientras NOT HayMasHermanos (nivel, s) AND (nivel>0)
                        hacer Retroceder (nivel, s)
        finsi
    hasta nivel==0
```

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

5. Algoritmos Bactracking. Variaciones del Esquema general: **Caso 3º.**

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

Backtracking (var s: TuplaSolución)

nivel:= 1

s:= $s_{INICIAL}$

voa:= -∞; soa:= Ø

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

repetir

Generar (nivel, s)

si EsSolución (nivel, s) **AND Valor(s) > voa entonces**

voa:= Valor(s); soa:= s

si Criterio (nivel, s) **entonces**

nivel:= nivel + 1

sino

mientras NOT HayMasHermanos (nivel, s) **AND (nivel>0)**

hacer Retroceder (nivel, s)

finsi

hasta **nivel==0**

 Variaciones del Esquema general: **Ejemplo.**

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

- **Funciones**:
  - **Valor(s)**

    devolver s[1] + s[2] + ... + s[n]
  - Todo lo demás no cambia !!

- **Otra posibilidad:** incluir una nueva variable:

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

5. Algoritmos Bactracking. Análisis de tiempos de ejecución.

Observaciones

- La representación de las soluciones determina la forma del árbol de expansión
  - Cantidad de descendientes de un nodo
  - Profundidad del árbol
  - Cantidad de nodos del árbol
- La representación de las soluciones determina, como consecuencia, la eficiencia del algoritmo ya que el tiempo de ejecución depende del número de nodos generados
- El árbol tendrá (como mucho) tantos niveles como valores tenga la secuencia solución
- En cada nodo se debe poder determinar:
  - Si es solución o posible solución del problema
  - Si tiene hermanos sin generar
  - Si a partir de este nodo se puede llegar a una solución

5. Algoritmos Bactracking. Análisis de tiempos de ejecución.

- En general se obtienen ordenes de complejidad exponencial y factorial
- El orden de complejidad depende del número de nodos generados y del tiempo requerido para cada nodo (que podemos considerar constante)
- Si la solución es de la forma $(\mathbf{x_1, x_2, \ldots, x_n})$, donde $\mathbf{x_i}$ admite $\mathbf{m_i}$ valores
- En el caso peor, se generaran todas las posibles combinaciones para cada $\mathbf{x_i}$

Nivel 1	**m_1 nodos**
Nivel 2	$\mathbf{m_1} * \mathbf{m_2}$ nodos
…..	….
Nivel n	$\mathbf{m_1} * \mathbf{m_2} *\ldots * \mathbf{m_n}$ nodos

- Para el ejemplo planteado al principio, $\mathbf{m_i} = 2$

$$\boldsymbol{T(n) = 2 + 2^2 + 2^3 + \ldots + 2^n = 2^{n+1} - 2}$$

Tiempo exponencial

- Cada caso depende de como se realice la poda del árbol, y de la instancia del problema

5. Algoritmos Bactracking. Análisis de tiempos de ejecución.

- Para el problema de calcular todas las permutaciones de (1, 2,…, **n**)
- Se representa la solución como una tupla $\{x_1, x_2, ..., x_n\}$,
- Restricciones explícitas: $x_i \in \{i, ..., n\}$,
- En el nivel 1, hay *n* posibilidades, en el nivel 2 *n* - 1

Nivel 1	***n* nodos**
Nivel 2	$n*(n-1)$ nodos
…..	….
Nivel n	$n*(n-1)*\ldots*1$ nodos

- **Tiempo factorial**

$$T(n) = n + n*(n-1) + \ldots + n! \in O(n!)$$

5. Algoritmos Bactracking. Análisis de tiempos de ejecución.

Resumen

- Normalmente, el tiempo de ejecución se puede obtener multiplicando dos factores:
  - Número de nodos del árbol.
  - Tiempo de ejecución de cada nodo.

  siempre que el tiempo en cada nodo sea del mismo orden.
- Las podas eliminan nodos a estudiar, pero su efecto suele ser más impredecible.
- En general, los algoritmos de backtracking dan lugar a tiempos de órdenes factoriales o exponenciales $\Rightarrow$ No usar si existen otras alternativas más rápidas.

- "**Mochila 0-1**". Como el problema de la mochila, pero los objetos **no** se pueden partir (se cogen enteros o nada)
- **Datos del problema:**
  - **n**: número de objetos disponibles.
  - **M**: capacidad de la mochila.
  - $\mathbf{p = (p_1, p_2, ..., p_n)}$ pesos de los objetos.
  - $\mathbf{b = (b_1, b_2, ..., b_n)}$ beneficios de los objetos.
- **Formulación matemática:**

Maximizar $\sum_{i=1}^{n} x_i b_i$ sujeto a la **restricción** $\sum_{i=1}^{n} x_i p_i \leq M$ con $x_i \in \{1,0\}$

- Veremos la solución con los esquemas:
  1. Backtracking recursivo
  2. Backtracking NO recursivo

1. Implementación con backtracking recursivo.

- La solución se puede representar, con un árbol binario, como una tupla

  s= {x_1, x_2, ..., x_n}
- Restricciones explicitas: $x_i \in \{0,1\}$
  - $x_i = 0 \rightarrow$ el objeto ***i*** no se introduce en la mochila
  - $x_i = 1 \rightarrow$ el objeto ***i*** no se introduce en la mochila
- Restricciones implícitas: $\sum_{i=1}^{n} x_i p_i \leq M$
- El objetivo es maximizar la función $\sum_{i=1}^{n} x_i b_i$
- Se almacenará una **solución parcial** que se irá actualizando al encontrar una nueva solución con mayor beneficio
- Solo los nodos terminales del árbol de expansión pueden ser solución al problema
- La función `alcanzable(Criterio)` comprueba que los pesos acumulados hasta el momento no excedan la capacidad de la mochila. Esta función permite la **poda** de nodos

5. **Backtracking recursivo**. Ejemplo. 1_Problema de la mochila 0/1.

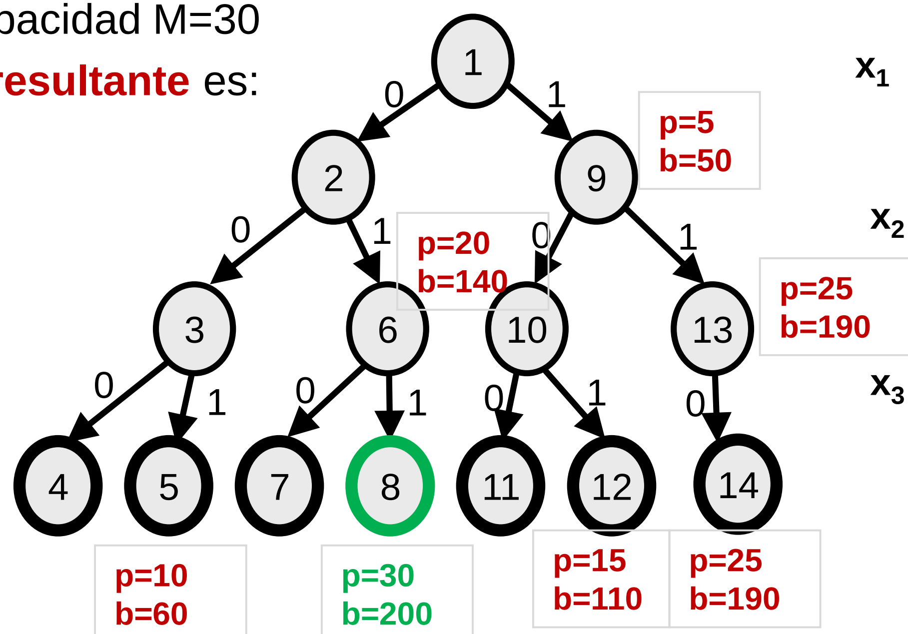
```
/*elem[1..n] es un vector de estructuras con dos campos: beneficio y peso*/
proc mochila(elem[1..n],solAct[1..n],sol[1..n],benActIni,ben,pesoActIni,etapa)
  para obj:= 0 hasta 1 hacer
    solAct[etapa]:=obj
    benAct:= benActIni + obj*elem[etapa].beneficio
    pesoAct:= pesoActIni + obj*elem[etapa].peso
    si (pesoAct ≤ M) entonces
      si etapa = n entonces
        si benAct > ben entonces
          sol:=solAct /* Se asigna el vector completo */
          ben:= benAct
        fsi
      sino
        mochila(elem,solAct,sol,benAct,ben,pesoAct,etapa+1) /* recursión */
      fsi
    fsi
  fpara
fproc
```

- El procedimiento que invoca al algoritmo es:

```
proc invoca_mochila(elem[1..n],sol[1..n])
  crear solAct[1..n]
  mochila(elem,solAct,sol,0,-∞,0,1)
fproc
```

- **Ejemplo**: Con una mochila de capacidad M=30 y los siguientes objetos, el **árbol resultante** es:

objeto	A	B	C
peso	5	20	10
valor	50	140	60



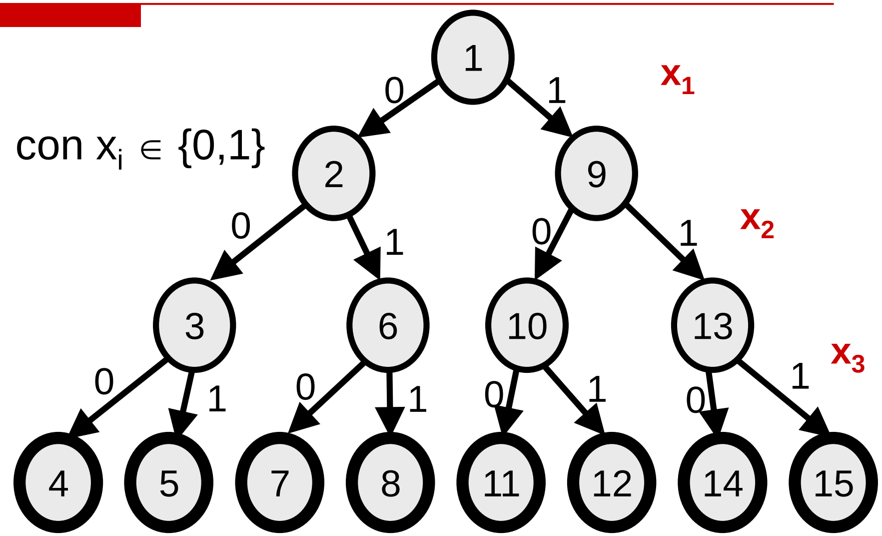
2. Implementación con backtracking NO recursivo:

- Aplicación de backtracking NO recursivo (proceso metódico):

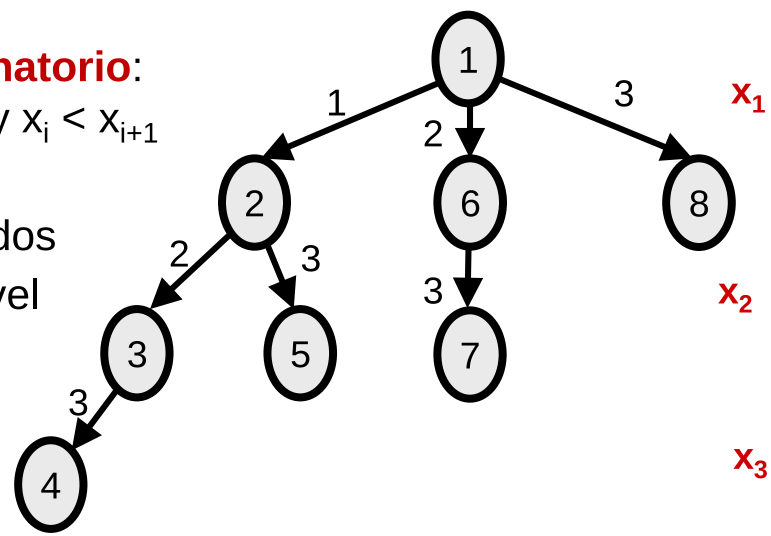
  1. Determinar cómo es la forma del árbol de backtracking ⇔ cómo es la representación de la solución.
  2. Elegir el esquema de algoritmo adecuado, adaptándolo en caso necesario.
  3. Diseñar las funciones genéricas para la aplicación concreta: según la forma del árbol y las características del problema.
  4. Posibles mejoras: usar variables locales con “valores acumulados”, hacer más podas del árbol, etc.

1. Representación de la solución.

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



- También es posible usar un **árbol combinatorio**:
  s= (x_1, x_2, ..., x_m), con $m \leq n$, $x_i \in \{1,..,n\}$ y $x_i < x_{i+1}$
  - x_i → Número de objeto escogido
  - m → Número total de objetos escogidos
  - Las soluciones están en cualquier nivel



2. Elegir el esquema de algoritmo: caso optimización.

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

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

pact: Peso actual
bact: Beneficio actual

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

3. Funciones genéricas del esquema.

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

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

  pact:= pact + p[nivel]*s[nivel]

  bact:= bact + b[nivel]*s[nivel]

  si s[nivel]==1 **entonces**

  pact:= pact + p[nivel]

  bact:= bact + b[nivel]

  finsi

- **EsSolución (nivel, s)**

  devolver (nivel==n) AND (pact≤M)

- **Criterio (nivel, s)**

  devolver (nivel<n) AND (pact≤M)

- **HayMasHermanos (nivel, s)**

  devolver s[nivel] < 1

- **Retroceder (nivel, s)**

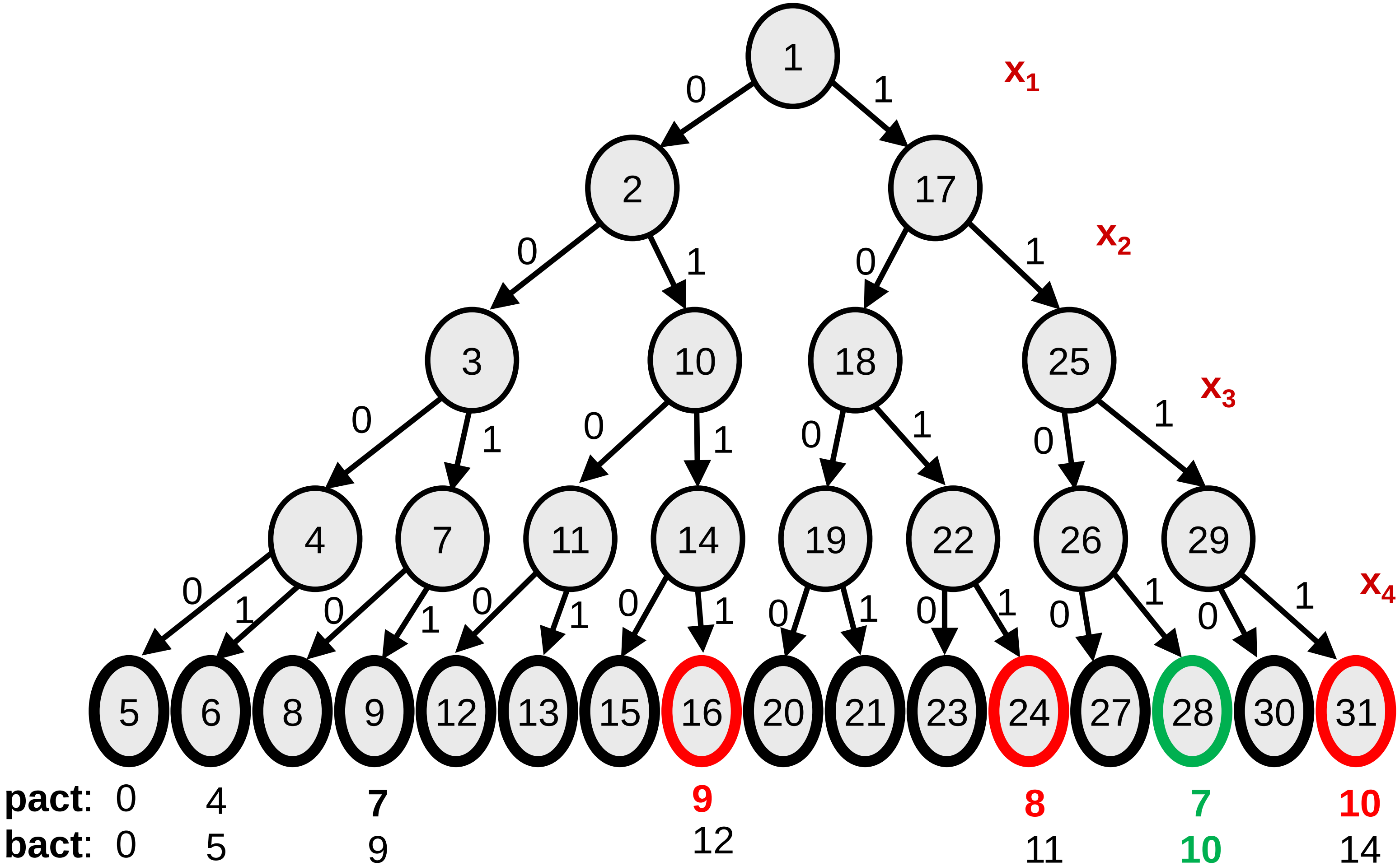
  pact:= pact – p[nivel]*s[nivel]

  bact:= bact – b[nivel]*s[nivel]

  s[nivel]:= -1

  nivel:= nivel – 1

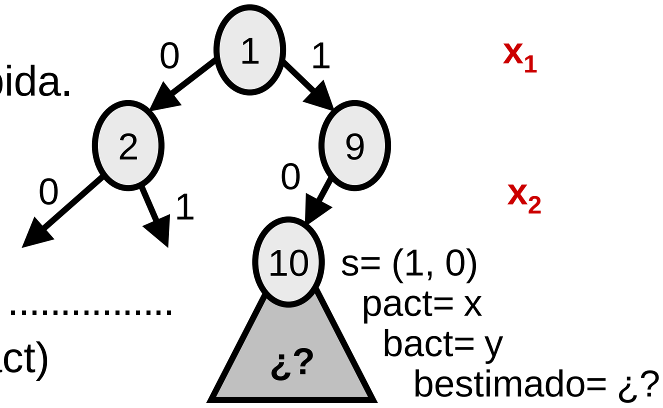
- **Ejemplo:** n = 4; M = 7; b = (2, 3, 4, 5); p = (1, 2, 3, 4)



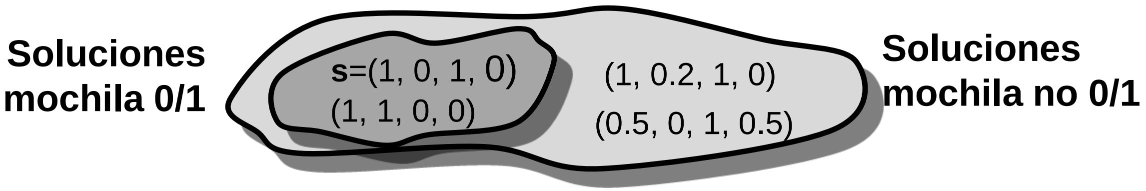
5. Algoritmos Bactracking. Ejemplos de aplicación. 1_Problema de la mochila 0/1.

- El algoritmo resuelve el problema, encontrando la solución óptima pero:
  - Es muy ineficiente. ¿Cuánto es el orden de complejidad?
  - **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.
- Incluir una poda según el criterio de optimización.
  - **Poda según el criterio de peso:** si el peso actual es mayor que M podar el nodo.
  - **Poda según el criterio de optimización:** si el beneficio actual no puede mejorar el **voa** podar el nodo.

- ¿Cómo calcular una cota superior del beneficio que se puede obtener a partir del nodo actual, es decir $(x_1, ..., x_k)$?
- La estimación debe poder realizarse de forma rápida.



- La **estimación del beneficio** para el nivel y nodo actual será: bestimado:= bact + Estimacion (nivel+1, n, M - pact)
- **Estimacion (k, n, Q)**: Estimar una cota superior para el problema de la mochila 0/1, usando los objetos **k..n**, con capacidad máxima **Q**.
- ¿Cómo? **Idea:** el resultado del problema de la mochila (no 0/1) es una cota superior válida para el problema de la mochila 0/1.
- **Demostración:**



- Sea **s** la solución óptima de la mochila 0/1. **s** es válida para la mochila no 0/1. Por lo tanto, la solución óptima de la mochila no 0/1 será **s** o mayor.

- **Estimacion (k, n, Q)**: Aplicar el algoritmo voraz para el problema de la mochila, con los objetos **k..n**. Si los beneficios son enteros, nos podemos quedar con la parte entera por abajo del resultado anterior.
- ¿Qué otras partes se deben modificar?
- **Criterio (nivel, s)**

  si (pact > M) OR (nivel == n) **entonces devolver** FALSO

  sino

  bestimado:= bact + $\lfloor$MochilaVoraz (nivel+1, n, M - pact)$\rfloor$

  devolver bestimado > voa

  finsi
- En el algoritmo principal:

  .....

  mientras (NOT HayMasHermanos (nivel, s) OR

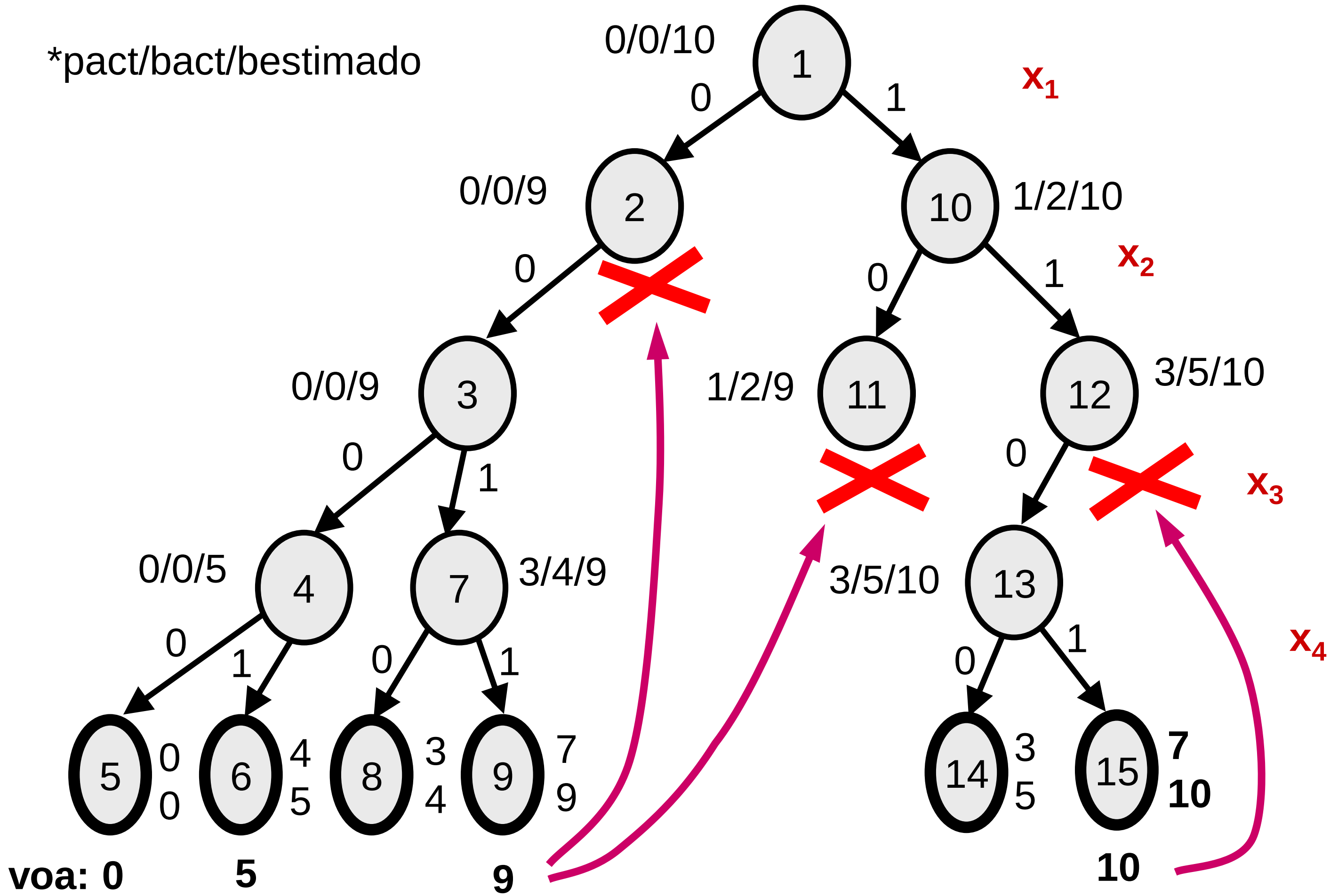
  NOT Criterio (nivel, s)) AND (nivel > 0) **hacer**

  Retroceder (nivel, s)

  .....

- **Ejemplo:** n = 4; M = 7; b = (2, 3, 4, 5); p = (1, 2, 3, 4) ; b/p=(2, 1.5, 1.3, 1.25)
  bestimado:= bact + ⌊MochilaVoraz (nivel+1, n, M - pact)⌋

*pact/bact/bestimado



- Se eliminan nodos pero a costa de aumentar el tiempo de ejecución en cada nodo.
- ¿Cuál será el tiempo de ejecución total?
- Suponiendo los objetos ordenados por b_i/p_i...
- Tiempo de la función **Criterio** en el nivel **i** (en el peor caso) es

  $\boldsymbol{T}_{\textbf{Criterio}}$ =1 + Tiempo de la función **MochilaVoraz** = $\boldsymbol{1 + n - i}$
- **Idea intuitiva.** Tiempo en el peor caso (suponiendo todos los nodos): Número de nodos **O**(2^n) ***** Tiempo de cada nodo(función criterio) **O**(n).
- ¿Tiempo: **O**($n \cdot 2^n$)? **NO**

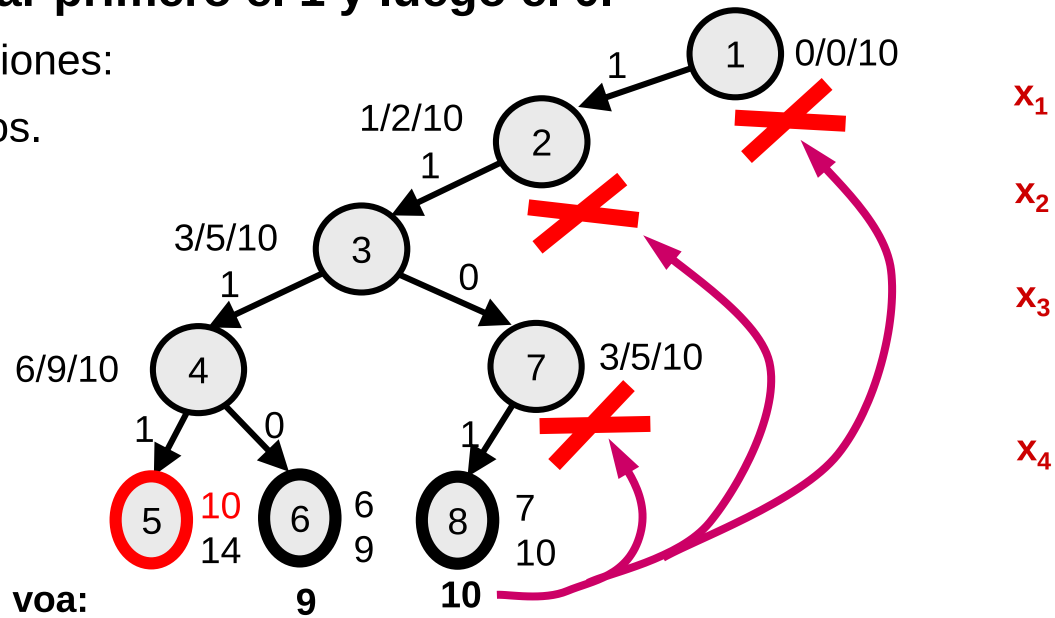
$$t(n) = \sum_{i=1}^{n} 2^i \cdot (n - i + 1) = (n+1)\sum_{i=1}^{n} 2^i - \sum_{i=1}^{n} i \cdot 2^i = 2 \cdot 2^{n+1} - 2n - 4$$

Conclusiones:

- El cálculo intuitivo no es correcto.
- 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. ¿Cómo mejorarlo?
- Posibilidades:
  - 1. Generar primero el 1 y luego el 0.
  - 2. Usar un árbol combinatorio.
  - ...

- **Modificación 1ª: Generar primero el 1 y luego el 0.**
- **Ejercicio:** Cambiar las funciones:

  Generar y HayMasHermanos.
- **Ejemplo:** n = 4; M = 7
  b = (2, 3, 4, 5)
  p = (1, 2, 3, 4)

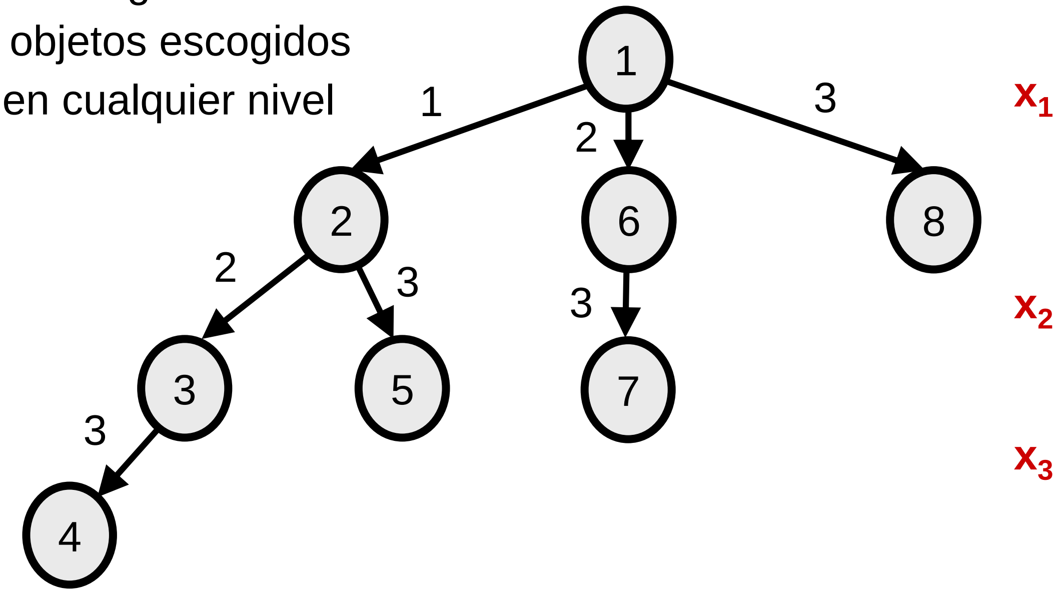


- En este caso es mejor la estrategia “primero el 1”, pero ¿y en general?
- Si la solución óptima es de la forma s = (1, 1, 1, X, X, 0, 0, 0) entonces se alcanza antes la solución generando primero 1 (y luego 0).
- Si es de la forma s = (0, 0, 0, X, X, 1, 1, 1) será mejor empezar por 0.
- **Idea:** es de esperar que la solución de la mochila 0/1 sea “parecida” a la de la mochila **no** 0/1. Si ordenamos los objetos por b_i/p_i entonces tendremos una solución del primer tipo.

- **Modificación 2ª: Usar un árbol combinatorio.**
- **Representación de la solución:**

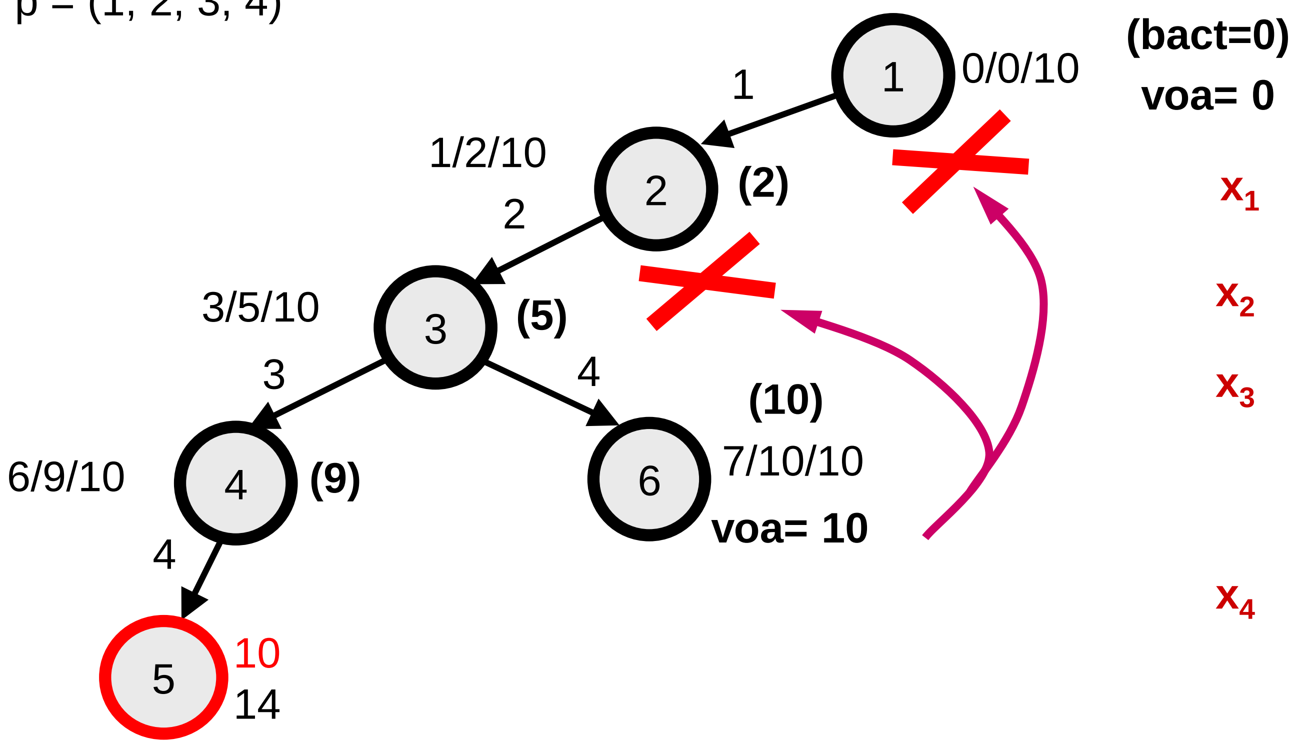
  $\mathbf{s= (x_1, x_2, ..., x_m)}$, con $m \leq n$, $x_i \in \{1,..,n\}$ y $x_i < x_{i+1}$

  - x_i → Número de objeto escogido
  - m → Número total de objetos escogidos
  - Las soluciones están en cualquier nivel



- **Ejercicio:** Cambiar la implementación para generar este árbol.
  - Esquema del algoritmo: nos vale el mismo.
  - Modificar las funciones Generar, Solución, Criterio y HayMasHermanos.

- **Ejemplo:** n = 4; M = 7
  b = (2, 3, 4, 5)
  p = (1, 2, 3, 4)



- **Resultado:** conseguimos reducir el número de nodos.
- ¿Mejorará el tiempo de ejecución y el orden de complejidad?

5. Algoritmos Bactracking. Ejemplos. 2_Problema de asignación de tareas.

- Existen **n** empleados y **n** tareas a realizar.
- Se representa mediante una tabla M de tamaño nxn, **M[i, j]**, el coste de realizar la tarea j por el empleado i , para i , j = 1,…, n.
- **Objetivo:** asignar una tarea a cada trabajador (asignación uno-a-uno), de manera que se **minimice el coste total**.

Tareas

Empleados

M	1	2	3
1	3	5	1
2	10	10	1
3	8	5	5

- **Ejemplo 1.** s= {T1,T3,T2} /* (E1,T1), (E2,T3), (E3,T2) */

  M_{TOTAL}= 3+1+5= **9**
- **Ejemplo 2.** s= {T2,T1,T3} /* (E1,T2), (E2,T1), (E3,T3)*/

  M_{TOTAL}= 5+10+5= **30**

> ➢ El **problema de asignación** es un problema **NP-completo** clásico.

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

- **Datos del problema:**
  - **n:** número de empleados y de tareas disponibles.
  - **M: array** [1..n, 1..n] **de** entero. Coste (rendimiento) de cada asignación. **M[i, j]** = coste de realizar la tarea **j** por el empleado **i**.
  - El problema consiste en asignar a cada operario i una tarea j de forma que se minimice el coste total.
- La **solución** se puede representar como una tupla

  $\mathbf{s= \{x_1, x_2, ..., x_n\}}$
- Restricciones explicitas: $x_i \in \{1,\ldots n\}$
  - $\mathbf{x_i}$ es la **tarea** asignada al ***i-ésimo*** empleado
- Restricciones implícitas: $\mathbf{x_i \neq x_j}$, $\forall$ **i≠j**
- El objetivo es minimizar la función $\sum_{i=1}^{n} M[i, x_i]$
- Por cada solución que encuentre el algoritmo, se anotara su coste y se comparara con el coste de la mejor solución encontrada hasta el momento

```
procTareas(M[1..n,1..n],XAct[1..n],mejorX[1...n],costeAcIni,coste,etapa)
  XAct[etapa]:= 0
  repetir
    XAct[etapa] := XAct[etapa] + 1
      si TareaNoAsignada(XAct,etapa) entonces
        costeAc := costeAcIni + M[etapa,XAct[etapa]]
        si (costeAc ≤  coste) entonces
          si etapa < n entonces
            tareas(M,XAct,mejorX,costeAc,coste,etapa+1)
          sino
            mejorX :=  XAct
            coste :=  costeAc
          fsi
        fsi
      fsi
  hasta XAct[etapa]=n
fproc
```

- El código de TareaNoAsignada es el siguiente:

```
función TareaNoAsignada(Asignadas, n)
  para i := 1 hasta n-1 hacer
    si Asignadas[i] = Asignadas[n] entonces
      devolver falso
    fsi
  fpara
  devolver cierto
ffun
```

- El procedimiento que invoca al algoritmo es:

```
proc invoca_tareas(M[1..n,1..n],X[1..n],C)
   crear XAct[1..n]
   C:= ∞
   Tareas(M,XAct,X,0,C,1)
fproc
```

5. Algoritmos Bactracking. Ejemplos. 2_Problema de asignación de tareas.

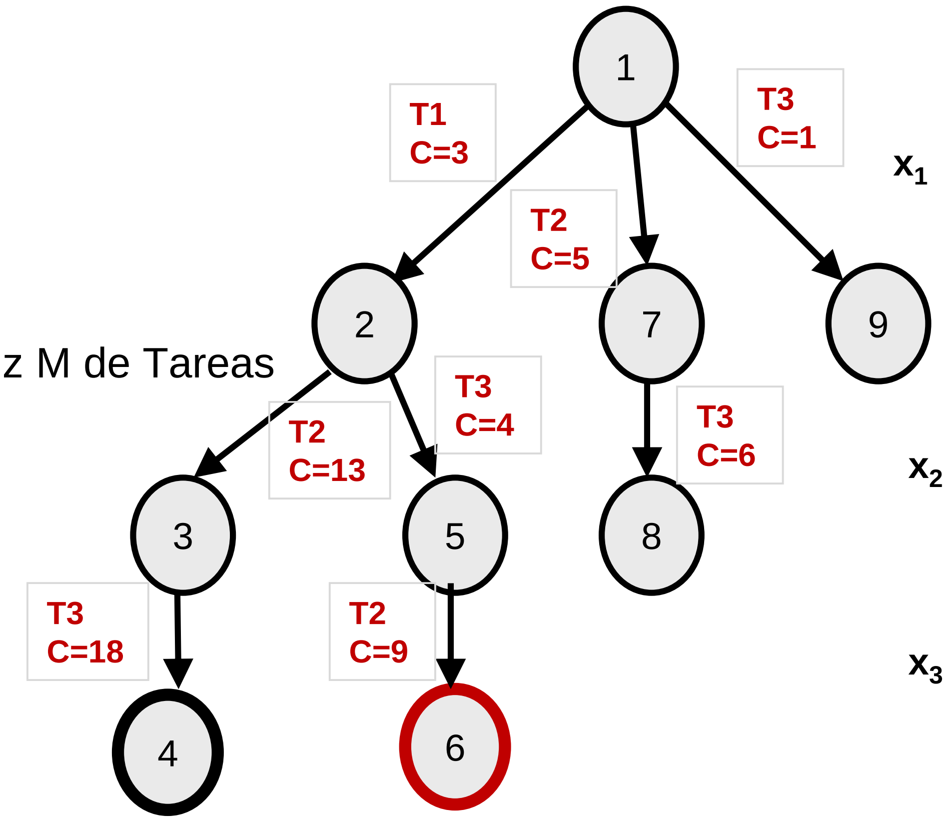
- **Ejemplo**:

Tareas (columnas) / Empleados (filas)

M	1	2	3
1	3	5	1
2	10	10	1
3	8	5	5

- **Secuencia de llamadas** para la matriz M de Tareas

```
tareas([0,0,0], [0,0,0], 0, ∞, 1)
    tareas([1,0,0], [0,0,0], 3, ∞, 2)
       tareas([1,2,0], [0,0,0], 13, ∞, 3)
       tareas([1,3,0], [1,2,3], 4, 18, 3)
    tareas([2,0,0], [1,3,2], 5, 9, 2)
       tareas([2,3,0], [1,3,2], 6, 9, 3)
    tareas([3,0,0], [1,3,2], 1, 9, 2)
```

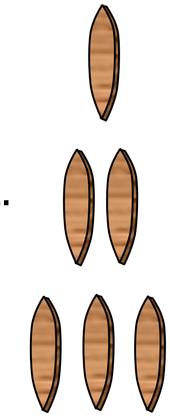


- El algoritmo realiza podas en el árbol de expansión eliminando aquellos nodos que no van a llevar a la solución optima
- **Ejercicio**: Realizar el ejemplo con el esquema NO recursivo

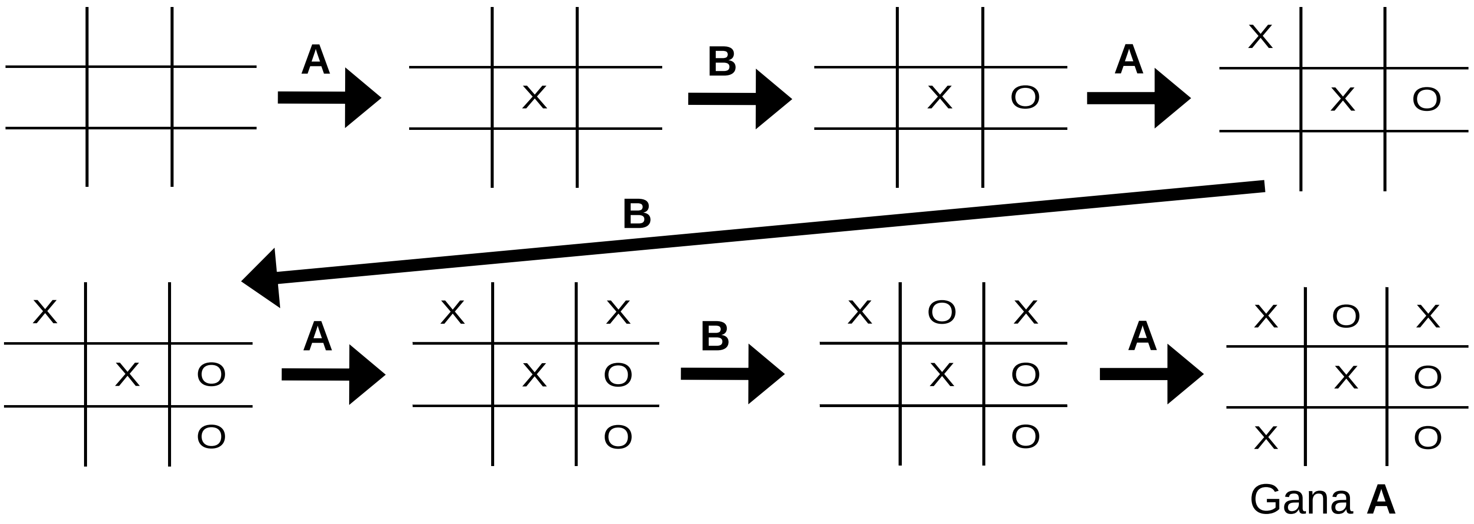
5. Algoritmos Bactracking. Ejemplos de aplicación. **3_Resolución de juegos.**

- La idea de backtracking (recorrido exhaustivo del árbol de un problema) se puede aplicar en **problemas de juegos**.
  - **Objetivo final:** decidir el movimiento óptimo que debe realizar el jugador que empieza moviendo.
- **Características (juegos de inteligencia):**
  - En el juego participan dos jugadores, **A** y **B**, que mueven alternativamente (primero **A** y luego **B**).
  - En cada movimiento un jugador puede elegir entre un número finito de posibilidades.
  - El resultado del juego puede ser: gana **A**, gana **B** o hay empate. El **objetivo** de los dos jugadores es **ganar**.
  - Supondremos juegos en los que no influye el azar.
  - **Ejemplos.** Las tres en raya, las damas, el ajedrez, el NIM, el juego de los palillos, etc.

- **Ejemplo.** El juego de los palillos.
  - Tres filas de palillos (en general **n**).
  - Cada jugador debe quitar uno o varios palillos, pero siempre de la misma fila.
  - Pierde el que quite el último palillo.



- **Ejemplo.** El juego de las tres en raya.



- Una partida es una **secuencia** de movimientos.
- Si representamos todas las partidas (todos los posibles movimientos) tenemos **un árbol.**

Árboles de juegos:

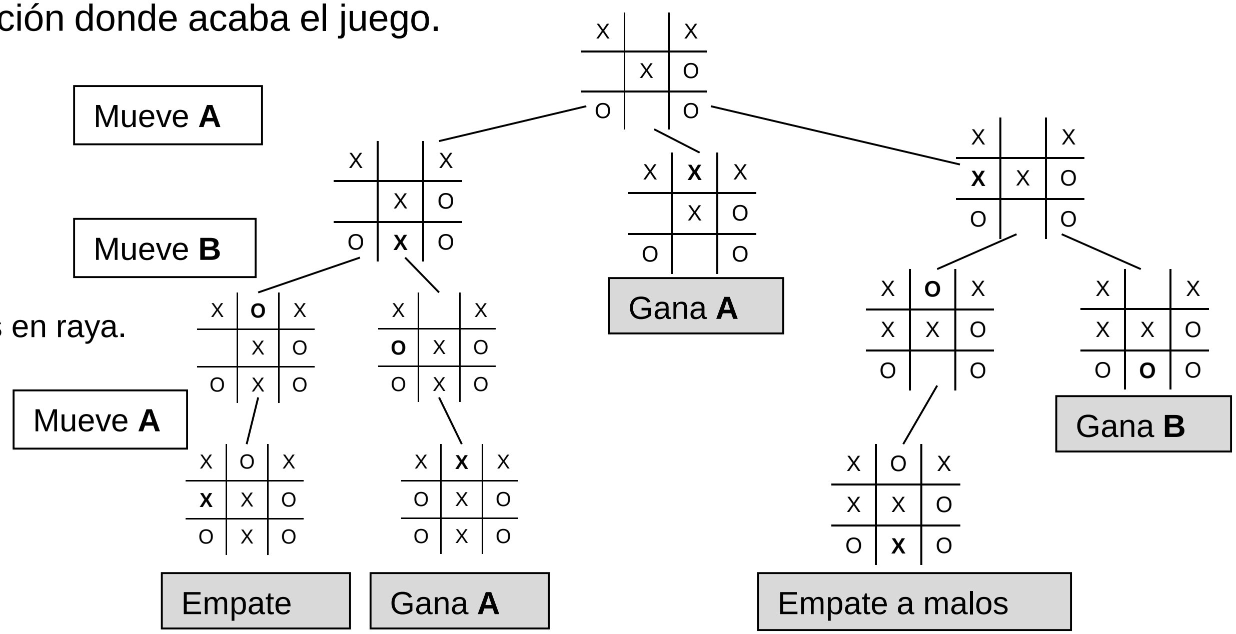
- Cada nodo del árbol representa un posible estado del juego.
- La **raíz** representa el comienzo de una partida.
- Los **descendientes** de un nodo dado son los movimientos posibles de cada jugador.
- En el nivel 1 mueve el jugador **A**.
- En el nivel 2 mueve el jugador **B**.
- En el nivel 3 mueve el jugador **A**.
- En el nivel 4 mueve el jugador **B**.
- ...
- Una hoja es una situación donde acaba el juego.

Ejemplo.

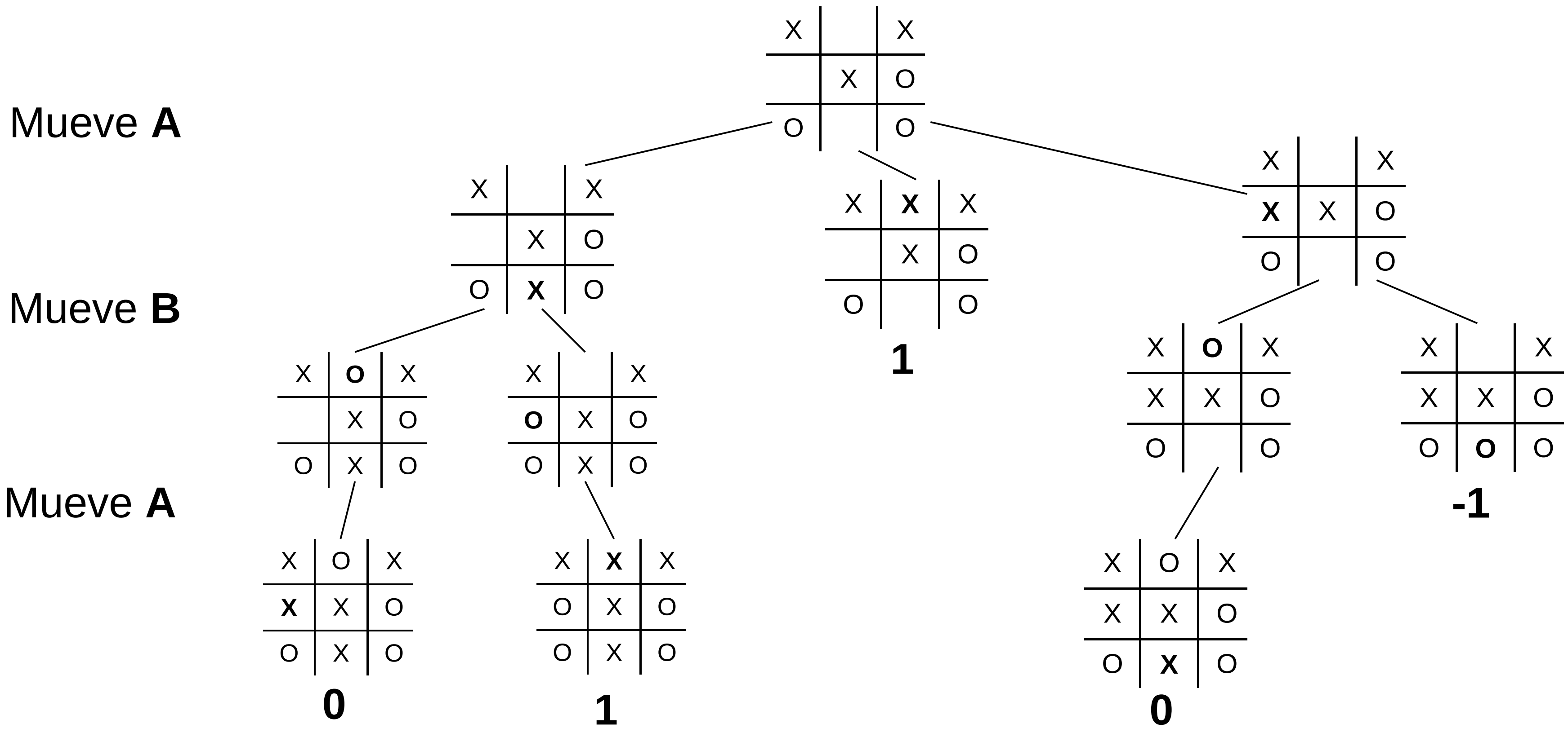
Parte del árbol de juego tres en raya.

A → X

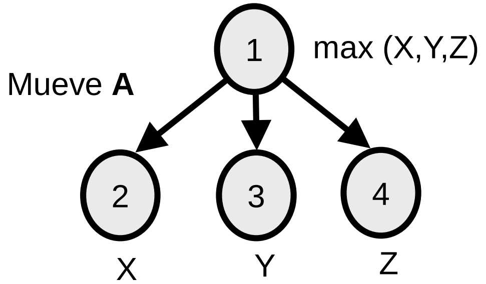
B → O

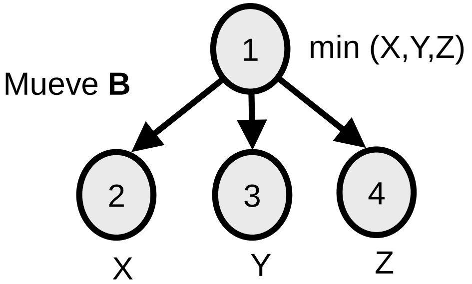


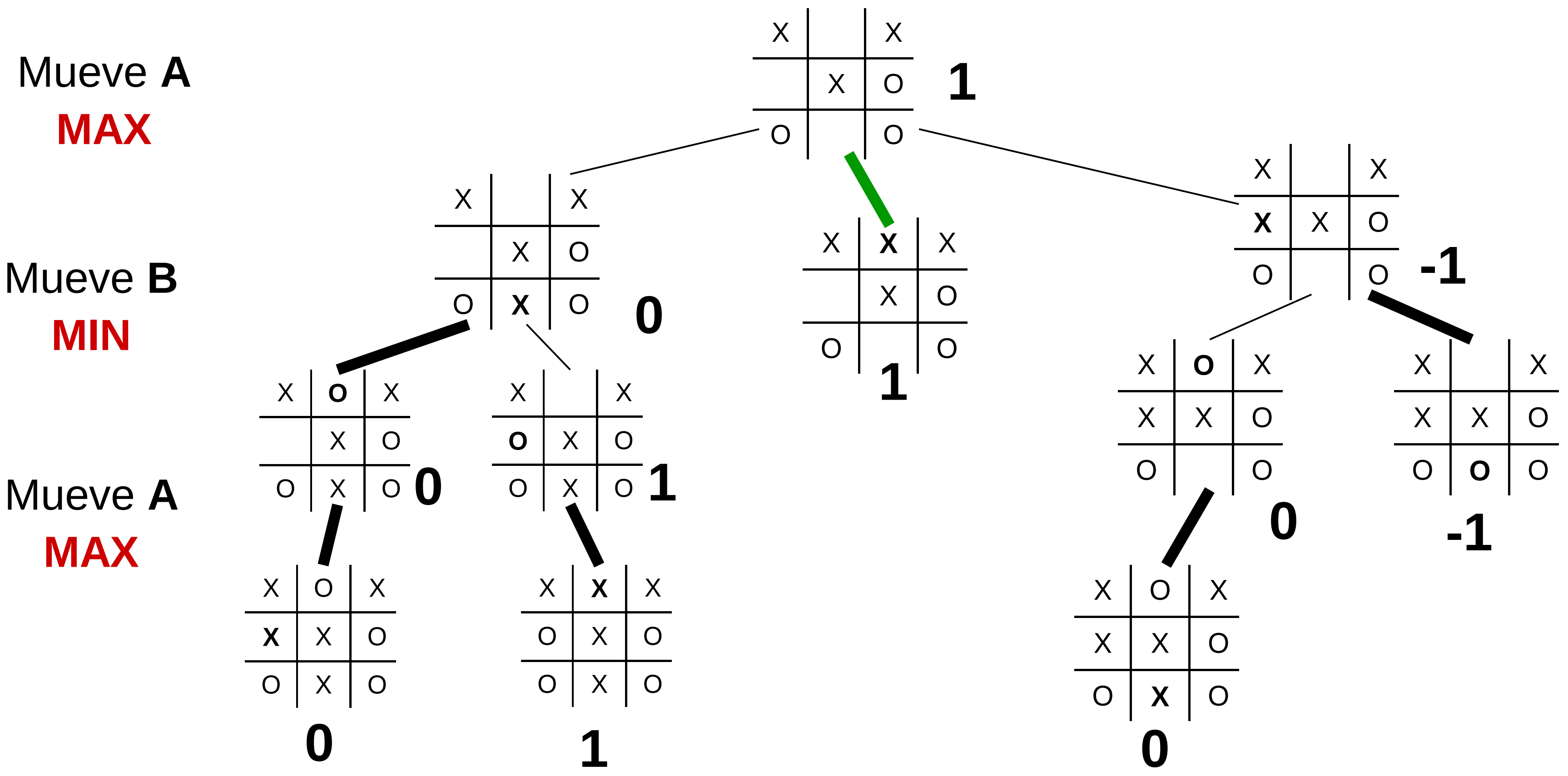
- Etiquetamos cada hoja con un número, que valdrá:
  - **1** → Si el juego finaliza con victoria de **A.**
  - **-1** → Si acaba con victoria de **B.**
  - **0** → Si se produce un empate.



- El objetivo para **A**: encontrar un camino en el árbol que le lleve hasta una hoja con valor 1.
- Pero, ¿qué pasa si a partir de la situación inicial no se llega a un nodo hoja con valor 1?
  - En los movimientos de **B**, el jugador **B** intentará llegar a hojas con valor -1 (ó en caso de no existir, de valor 0).
  - En los movimientos de **A**, el jugador **A** intentará llegar a hojas con valor 1 (ó en caso de no existir, de valor 0).
- De esta manera se define una forma de propagar el valor de los hijos hacia los padres: **estrategia minimax**.
- **Estrategia minimax.** Los valores de las hojas se **propagan** a los padres de la siguiente forma:
  - En los movimientos de **A**, el valor del nodo padre será el **máximo** de los valores de los nodos hijos.
  - En los movimientos de **B**, el valor del nodo padre será el **mínimo** de los valores de los nodos hijos.
  - Se repite hasta llegar al nodo raíz (situación de partida).







- **Movimiento óptimo:** aquel que conduzca al máximo. O si el primer nivel es un MIN, el que conduzca al mínimo.

- En general, tendremos una **función de utilidad**.
- **Función de utilidad:** para cada nodo hoja devuelve un valor numérico, indicando cómo de buena es esa situación para el jugador **A**.
- Si el árbol del juego es muy grande o infinito (por ejemplo, en el ajedrez) entonces la función de utilidad debe poder aplicarse sobre situaciones no terminales.
- En ese caso, la función de utilidad es una medida heurística: cómo es de prometedora la situación para **A**.

- **Proceso de resolución de juegos:**
  - Generar el árbol de juego hasta un nivel determinado. ¿Cuánto?
  - Aplicar la función de utilidad a los nodos hoja.
  - Propagar los valores de utilidad hasta la raíz, usando la **estrategia minimax**:
    - En los movimientos impares tomar el máximo de los hijos.
    - En los movimientos pares tomar el mínimo de los hijos.
  - **Solución final:** escoger el movimiento indicado por el hijo de la raíz con mayor valor.

- **Implementación:** Usar un backtracking recursivo.
- Backtracking → el recorrido será en profundidad.

- **Implementación de la estrategia minimax.**

```
BuscaMinimax (B: TipoTablero; modo: (MAX, MIN)) : real
si EsHoja(B) entonces
    devolver Utilidad (B)
sino
    si modo == MAX entonces valoract:= -∞
    sino valoract:= ∞
    para cada hijo C del tablero B hacer
        si modo == MAX entonces
            valoract:= max (valoract, BuscaMinimax(C, MIN))
        sino
            valoract:= min (valoract, BuscaMinimax(C, MAX))
        finsi
    finpara
    devolver valoract
finsi
```

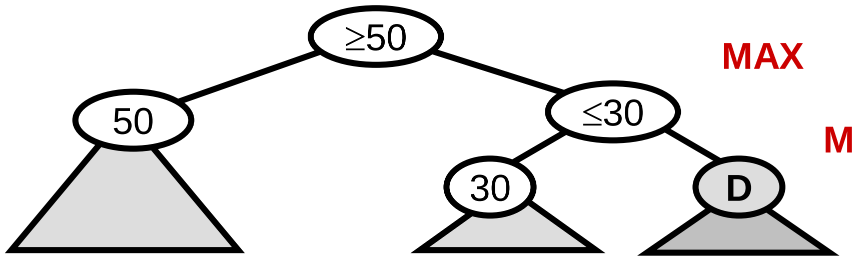
- **Tipos de datos:**
  - **TipoTablero:** Representación del estado del juego en un momento dado.
- **Funciones genéricas:**
  - **EsHoja (B):** Indica si el nodo es una situación terminal, o si estamos en el nivel máximo.
  - **Utilidad (B, modo):** Devuelve el valor de la función de utilidad para el tablero **B** en el **modo** indicado.
  - **para cada hijo C del tablero B:** Iterador para generar todos los movimientos a partir de una situación de partida **B**.

- **NOTA:** Faltaría devolver también el movimiento óptimo.

- **Ejemplo.** El juego de los palillos.

- Sobre los árboles de juegos se puede aplicar un tipo propio de poda, conocida como poda **alfa-beta**.
- **Poda Alfa:**

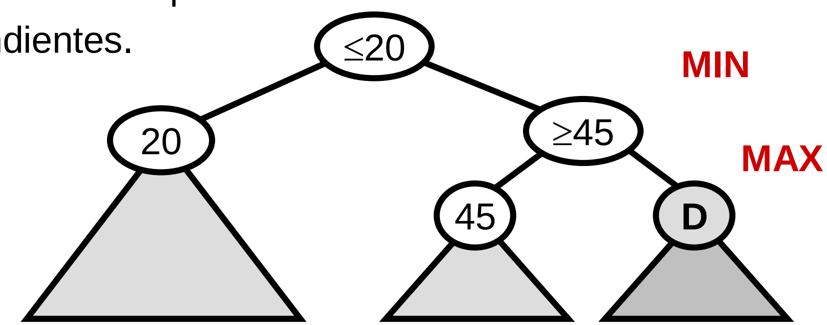
  Supongamos que en cierto momento de la evaluación llegamos a la siguiente situación.



  - Haya lo que haya en **D**, nunca estará el movimiento óptimo.
  - **Conclusión:** podar el nodo **D** y sus descendientes.

- **Poda Beta:**

  Supongamos que en cierto momento de la evaluación llegamos a la siguiente situación.
  - Haya lo que haya en **D**, nunca estará el movimiento óptimo.
  - **Conclusión:** podar el nodo **D** y sus descendientes.



- **Implementación estrategia minimax, con poda alfa-beta.**

```
BuscaMinimax (B: TipoTablero; valorPadre: real; modo: (MAX, MIN)) : real
    si EsHoja(B) entonces
        devolver Utilidad (B)
    sino
        si modo == MAX entonces valoract:= -∞
        sino valoract:= ∞
        para cada hijo C del tablero B hacer
            si modo == MAX entonces
                valoract:= max (valoract, BuscaMinimax(C, valoract, MIN))
{ P. beta}      si valoract ≥ valorPadre entonces salir del para
            sino
                valoract:= min (valoract, BuscaMinimax(C, valoract, MAX))
{ P. alfa}      si valoract ≤ valorPadre entonces salir del para
            finsi
        finpara
        devolver valoract
    finsi
```

5. Algoritmos Bactracking. Conclusiones.

- **Backtracking:** Recorrido exhaustivo y sistemático en un árbol de soluciones.
- **Pasos para aplicarlo:**
  - Decidir la forma del árbol.
  - Establecer el esquema del algoritmo.
  - Diseñar las funciones genéricas del esquema.
- Relativamente fácil diseñar algoritmos que encuentren soluciones óptimas pero los algoritmos de backtracking son muy ineficientes.
- **Mejoras:** mejorar los mecanismos de poda, incluir otros tipos de recorridos (no solo en profundidad)

  $\Rightarrow$ Técnica de **Ramificación y Poda.**