

Programa de teoría

Parte I. Estructuras de Datos.

1. Abstracciones y especificaciones.
2. Conjuntos y diccionarios.
3. Representación de conjuntos mediante árboles.
4. Grafos.

Parte II. Algorítmica.

1. Análisis de algoritmos.
2. Divide y vencerás.
3. Algoritmos voraces.
4. Programación dinámica.

→ 5. Backtracking.

6. Ramificación y poda.

PARTE II: ALGORÍTMICA

Tema 5. Backtracking.

- 5.1. Método general.
- 5.2. Análisis de tiempos de ejecución.
- 5.3. Ejemplos de aplicación.
 - 5.3.1. Problema de la mochila 0/1.
 - 5.3.2. Problema de la asignación.
 - 5.3.3. Resolución de juegos.

5.1. Método general.

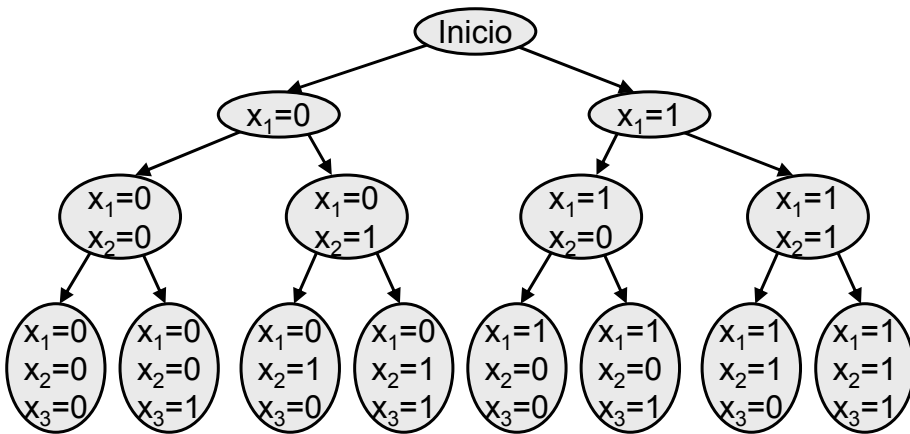
- El **backtracking** (o método de **retroceso** o **vuelta atrás**) es una técnica general de resolución de problemas, aplicable en problemas de **optimización**, **juegos** y otros tipos.
- El **backtracking** realiza una búsqueda exhaustiva y sistemática en el espacio de soluciones. Por ello, suele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

5.1. Método general.

- Una **solución** se puede expresar como una tupla: (x_1, x_2, \dots, 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, \dots, 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.

5.1. Método general.

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



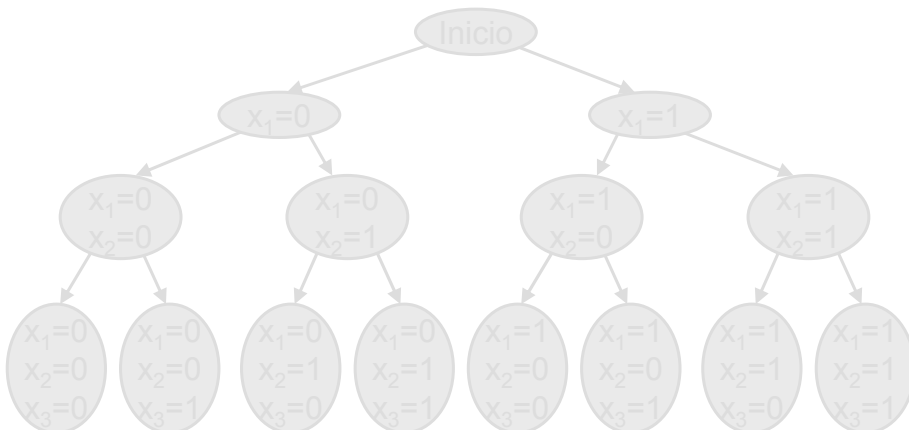
A.E.D.

Tema 5. Backtracking.

5

5.1. Método general.

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



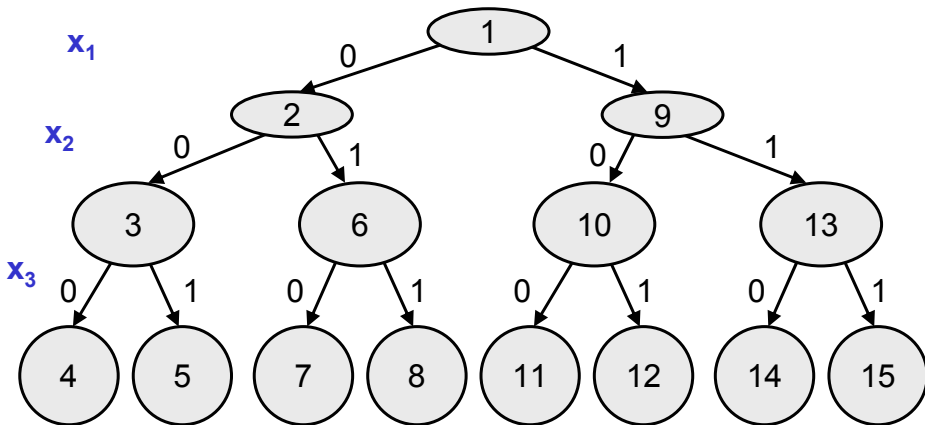
A.E.D.

Tema 5. Backtracking.

6

5.1. Método general.

- Representación simplificada del árbol.



5.1. Método general.

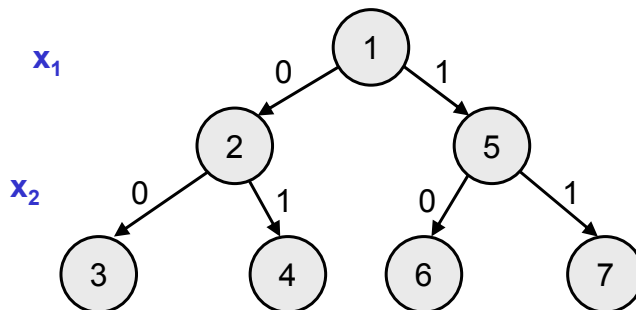
- **Árboles de backtracking:**
 - El árbol es simplemente una forma de representar la ejecución del algoritmo.
 - Es **implícito**, no almacenado (no necesariamente).
 - El recorrido es en **profundidad**, normalmente de izquierda a derecha.
 - La primera decisión para aplicar backtracking: ¿cómo es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

5.1. Método general.

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

5.1. Método general.

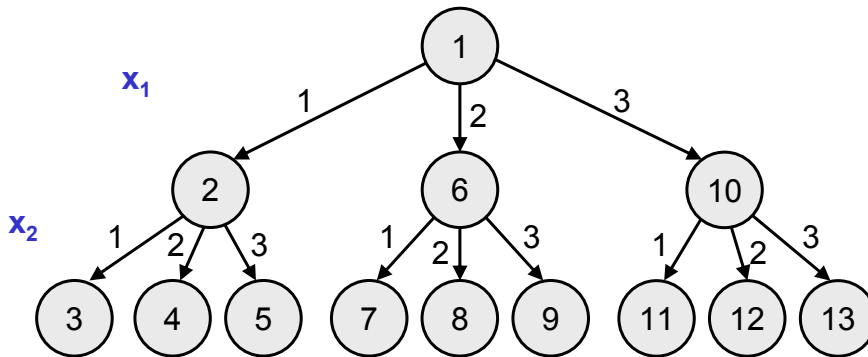
- **Árboles binarios:** $s = (x_1, x_2, \dots, x_n)$, con $x_i \in \{0, 1\}$



- **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.

5.1. Método general.

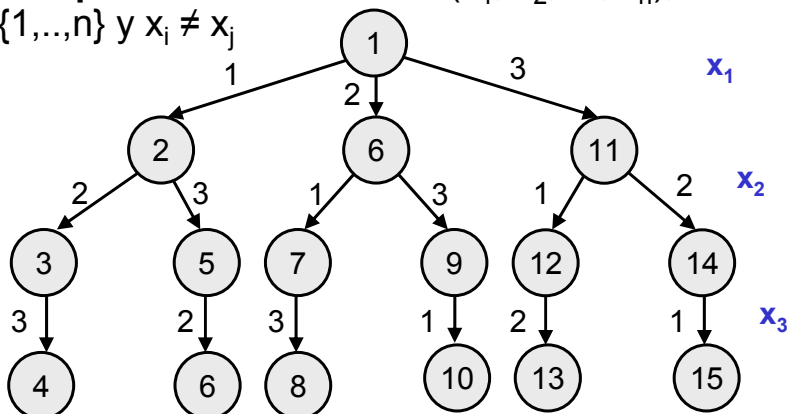
- **Árboles k-arios:** $s = (x_1, x_2, \dots, x_n)$, con $x_i \in \{1, \dots, k\}$



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

5.1. Método general.

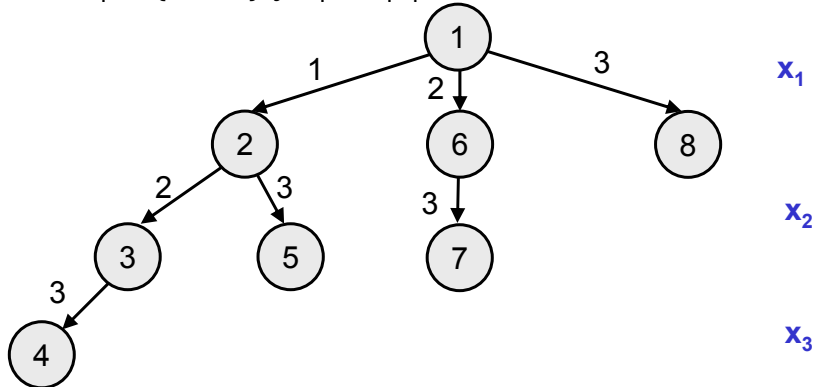
- **Árboles permutacionales:** $s = (x_1, x_2, \dots, x_n)$, con $x_i \in \{1, \dots, n\}$ y $x_i \neq x_j$



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

5.1. Método general.

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



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



5.1. Método general.

Cuestiones a resolver antes de programar:

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

5.1. Método general.

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

Backtracking (var s: TuplaSolución)

nivel:= 1

s:= s_{INICIAL}

fin:= false

repetir

Generar (nivel, s)

 si **Solución (nivel, s)** entonces

 fin:= true

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

 nivel:= nivel + 1

 sino mientras NOT **MasHermanos (nivel, s)** hacer

Retroceder (nivel, s)

hasta fin



5.1. Método general.

• Variables:

- **s**: Almacena la solución parcial hasta cierto punto.
- **s_{INICIAL}**: Valor de inicialización.
- **nivel**: Indica el nivel actual en el que se encuentra el algoritmo.
- **fin**: Valdrá **true** cuando hayamos encontrado alguna solución.

• Funciones:

- **Generar (nivel, s)**: Genera el siguiente hermano, o el primero, para el **nivel** actual.
- **Solución (nivel, s)**: Comprueba si la tupla (s[1], ..., s[nivel]) es una solución válida para el problema.

5.1. Método general.

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

5.1. Método general.

- **Ejemplo de problema:** Encontrar un subconjunto del conjunto $T = \{t_1, t_2, \dots, t_n\}$ que sume exactamente P .
- **Variables:**
 - Representación de la solución con un árbol binario.
 - **s:** array $[1..n]$ de $\{-1, 0, 1\}$
 - $s[i] = 0 \rightarrow$ el número i -ésimo no se utiliza
 - $s[i] = 1 \rightarrow$ el número i -ésimo sí se utiliza
 - $s[i] = -1 \rightarrow$ valor de inicialización (número i -ésimo no estudiado)
 - **s_{INICIAL}:** $(-1, -1, \dots, -1)$
 - **fin:** Valdrá **true** cuando se haya encontrado solución.
 - **tact:** Suma acumulada hasta ahora (inicialmente 0).

5.1. Método general.

Funciones:

- **Generar (nivel, s)**
s[nivel]:= s[nivel] + 1
si s[nivel]==1 entonces tact:= tact + t_{nivel}
- **Solución (nivel, s)**
devolver (nivel==n) Y (tact==P)
- **Criterio (nivel, s)**
devolver (nivel<n) Y (tact≤P)
- **MasHermanos (nivel, s)**
devolver s[nivel] < 1
- **Retroceder (nivel, s)**
tact:= tact – t_{nivel}*s[nivel]
s[nivel]:= -1
nivel:= nivel – 1

5.1. Método general.

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

Backtracking (var s: TuplaSolución)

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

5.1. Método general.

Variaciones del esquema general:

- 1) ¿Y si no es seguro que exista una solución?
- 2) ¿Y si queremos almacenar todas las soluciones (no sólo una)?
- 3) ¿Y si el problema es de optimización (maximizar o minimizar)?

5.1. Método general.

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

Backtracking (var s: TuplaSolución)

nivel:= 1

s:= s_{INICIAL}

fin:= false

repetir

Generar (nivel, s)

si Solución (nivel, s) **entonces**

fin:= true

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

nivel:= nivel + 1

sino

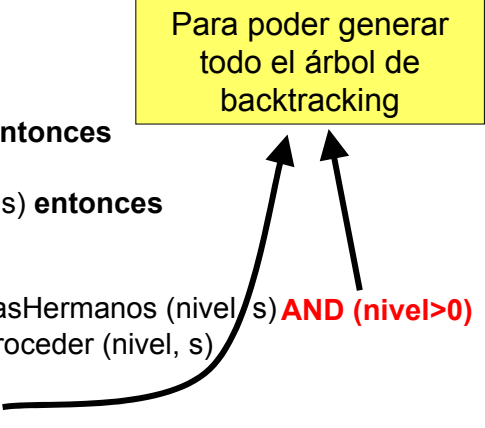
mientras NOT MasHermanos (nivel, s) **AND** (nivel>0)

hacer Retroceder (nivel, s)

finsi

hasta fin **OR** (nivel==0)

Para poder generar
todo el árbol de
backtracking



5.1. Método general.

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

Backtracking (var s: TuplaSolución)

nivel:= 1

S:= S_{INICIAL}

fin:= false

repetir

Generar (nivel, s)

si Solución (nivel, s) **entonces**

Almacenar (nivel, s)

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

nivel:= nivel + 1

sino

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

hacer Retroceder (nivel, s)

finsi

hasta nivel==0

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

5.1. Método general.

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

Backtracking (var s: TuplaSolución)

nivel:= 1

S:= S_{INICIAL}

voa:= -∞; soa:= ∅

repetir

Generar (nivel, s)

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

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

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

nivel:= nivel + 1

sino

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

hacer Retroceder (nivel, s)

finsi

hasta nivel==0

voa: valor óptimo actual
soa: solución óptima actual

5.1. Método general.

- **Ejemplo de problema:** Encontrar un subconjunto del conjunto $T = \{t_1, t_2, \dots, t_n\}$ que sume exactamente P , usando el menor número posible de elementos.
- **Funciones:**
 - **Valor(s)**
 devolver $s[1] + s[2] + \dots + s[n]$
 - ¡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): $vact := vact - s[nivel]$

5.2. Análisis de tiempos de ejecución.

- 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.
- **Ejercicio:** ¿Cuántos nodos se generan en un árbol binario, k-ario, permutacional y combinatorio?
- 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 → No usar si existen otras alternativas más rápidas.

5.3. Ejemplos de aplicación.

5.3.1. Problema de la 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.
 - **p** = (**p**₁, **p**₂, ..., **p**_n) pesos de los objetos.
 - **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 $\sum_{i=1..n} x_i p_i \leq M$, y $x_i \in \{0, 1\}$

5.3.1. Problema de la mochila 0/1.

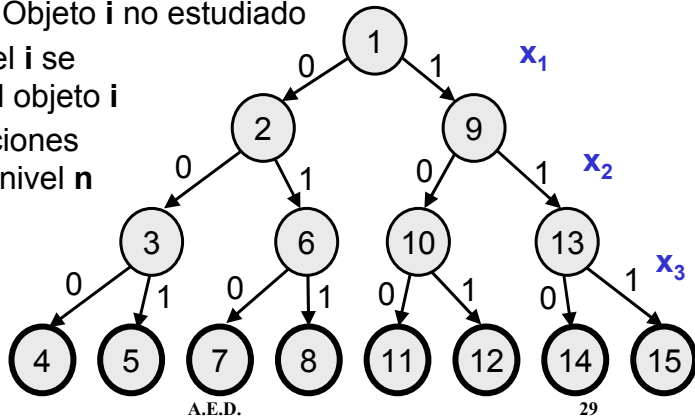
Aplicación de backtracking (proceso metódico):

- 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.

5.3.1. Problema de la mochila 0/1.

1) Representación de la solución.

- Con un árbol binario: $\mathbf{s} = (x_1, x_2, \dots, x_n)$, 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 nivel n

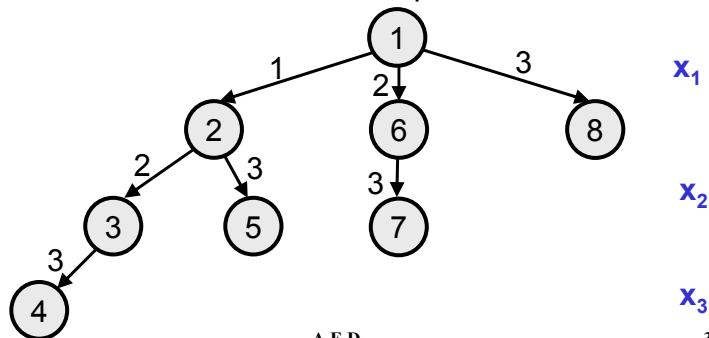


A.E.D.
Tema 5. Backtracking.

5.3.1. Problema de la mochila 0/1.

1) Representación de la solución.

- También es posible usar un árbol combinatorio:
 - $\mathbf{s} = (x_1, x_2, \dots, x_m)$, con $m \leq n$, $x_i \in \{1, \dots, n\}$ 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



A.E.D.
Tema 5. Backtracking.

5.3.1. Problema de la mochila 0/1.

2) Elegir el esquema de algoritmo: caso optimización.

Consejo: No reinventar la rueda.

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

nivel:= 1; s:= S_{INICIAL}

voa:= -∞; soa:= ∅

pact:= 0; bact:= 0

repetir

 Generar (nivel, s)

si Solución (nivel, s) AND (bact > voa) **entonces**

 voa:= bact; soa:= s

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

 nivel:= nivel + 1

sino

mientras NOT MasHermanos (nivel, s) AND (nivel>0)

hacer Retroceder (nivel, s)

finsi

hasta nivel == 0

pact: Peso actual

bact: Beneficio actual

A.E.D.

Tema 5. Backtracking.

31

5.3.1. Problema de la mochila 0/1.

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

- **Solución (nivel, s)**

devolver (nivel==n) AND (pact≤M)

- **Criterio (nivel, s)**

devolver (nivel<n) AND (pact≤M)

- **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

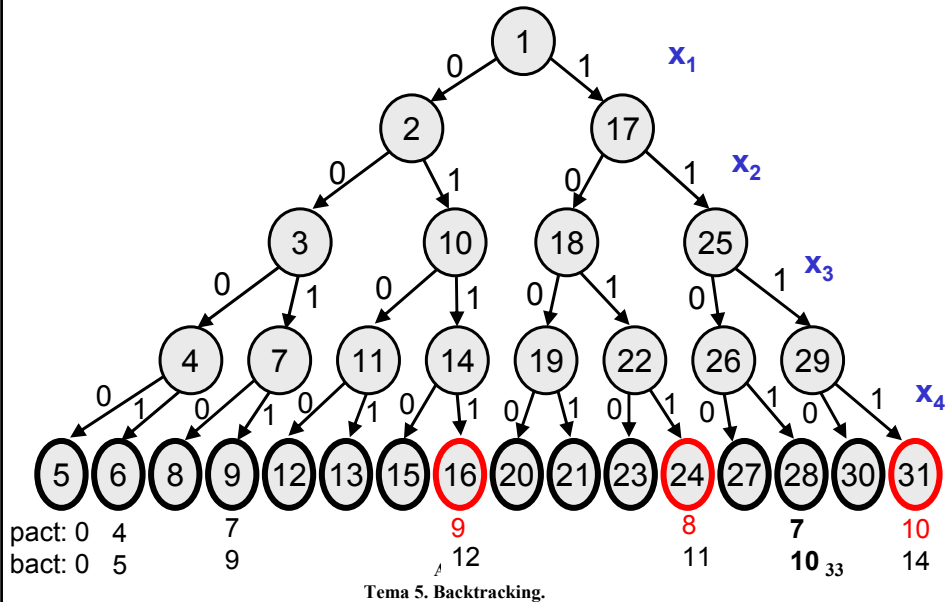
A.E.D.

Tema 5. Backtracking.

32

5.3.1. Problema de la mochila 0/1.

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

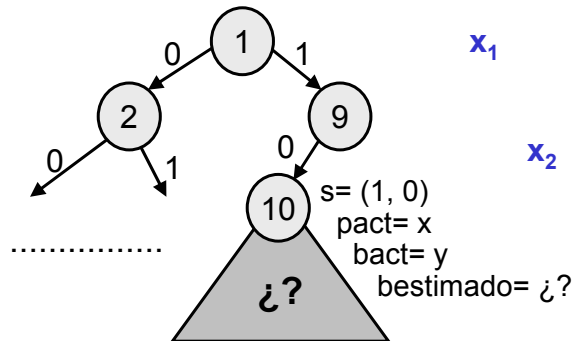


5.3.1. Problema de la mochila 0/1.

- 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.

5.3.1. Problema de la mochila 0/1.

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



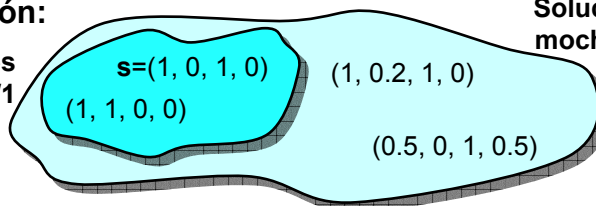
- La estimación del beneficio para el nivel y nodo actual será:
 $\text{bestimado} := \text{bact} + \text{Estimacion}(\text{nivel}+1, n, M - \text{pact})$

5.3.1. Problema de la mochila 0/1.

- **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:**

Soluciones
mochila 0/1



Soluciones
mochila no
0/1

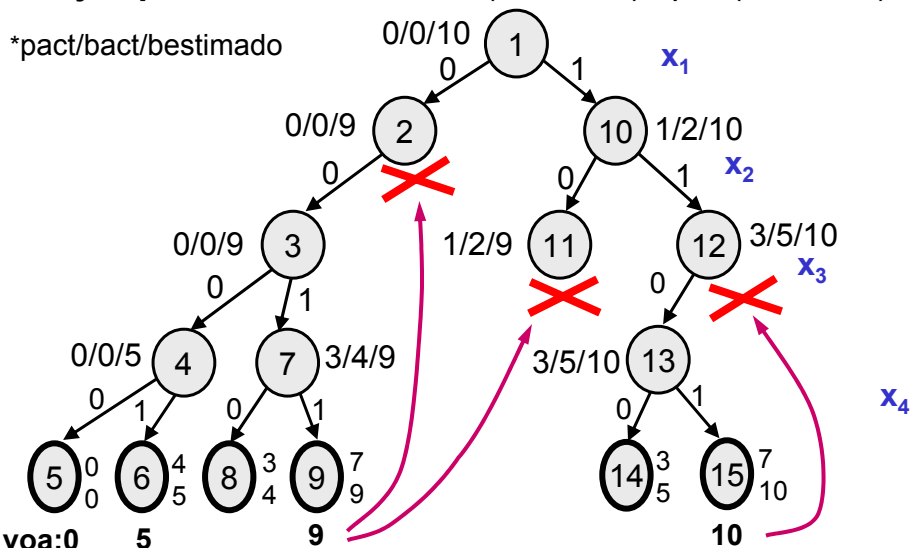
- Sea s la solución óptima de la mochila 0/1. s es válida para la mochila no 0/1. Por lo tanto, la solución óptima de la mochila no 0/1 será s o mayor.

5.3.1. Problema de la mochila 0/1.

- **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 + $\lfloor \text{MochilaVoraz}(\text{nivel}+1, n, M - \text{pact}) \rfloor$
 - devolver** bestimado > voa
 - finsi
- En el algoritmo principal:
 - ...
 - mientras** (NOT MasHermanos (nivel, s) OR NOT Criterio (nivel, s)) AND (nivel > 0) **hacer** Retroceder (nivel, s)
 - ...

5.3.1. Problema de la mochila 0/1.

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



5.3.1. Problema de la mochila 0/1.

- 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) = 1 + Tiempo de la función **MochilaVoraz** = 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

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

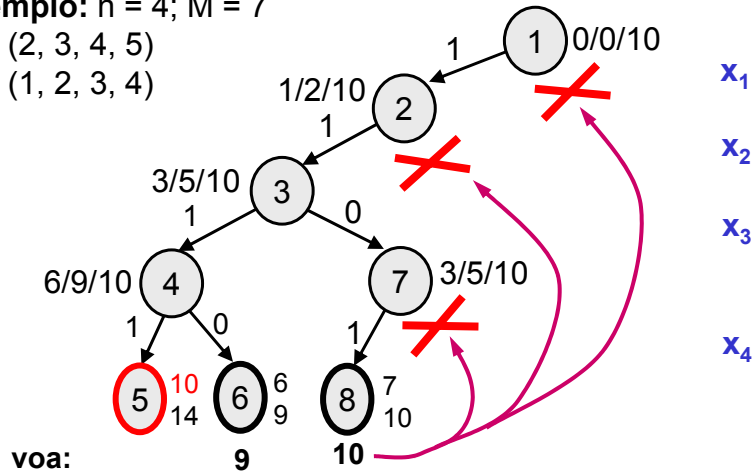
5.3.1. Problema de la mochila 0/1.

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:
 - Generar primero el 1 y luego el 0.
 - Usar un árbol combinatorio.
 - ...

5.3.1. Problema de la mochila 0/1.

- **Modificación:** Generar primero el 1 y luego el 0.
- **Ejercicio:** Cambiar las funciones Generar y MasHermanos.
- **Ejemplo:** $n = 4$; $M = 7$
 $b = (2, 3, 4, 5)$
 $p = (1, 2, 3, 4)$



A.E.D.
Tema 5. Backtracking.

41

5.3.1. Problema de la mochila 0/1.

- 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.

A.E.D.
Tema 5. Backtracking.

42

5.3.1. Problema de la mochila 0/1.

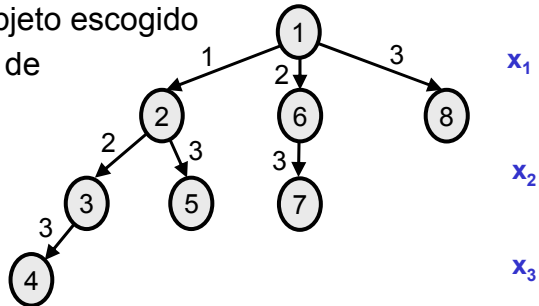
- **Modificación:** Usar un árbol combinatorio.
- **Representación de la solución:**

$s = (x_1, x_2, \dots, x_m)$, con $m \leq n$, $x_i \in \{1, \dots, n\}$ 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



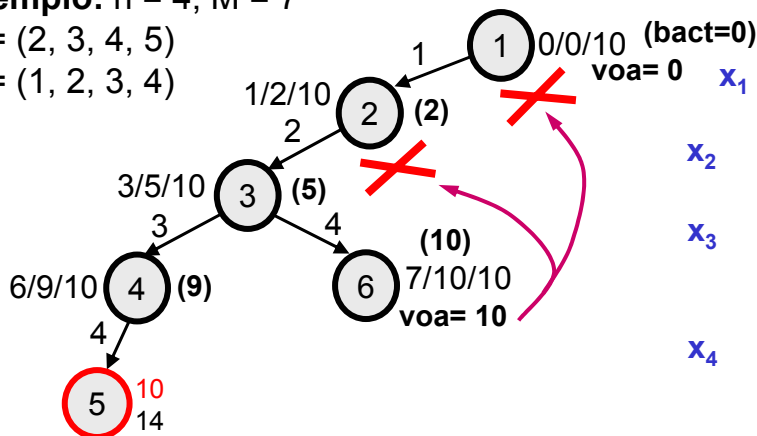
- **Ejercicio:** Cambiar la implementación para generar este árbol.
 - Esquema del algoritmo: nos vale el mismo.
 - Modificar las funciones Generar, Solución, Criterio y MasHermanos.

5.3.1. Problema de la mochila 0/1.

- **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?

5.3.2. Problema de asignación.

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

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

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

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

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

5.3.2. Problema de asignación.

- 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.

5.3.2. Problema de asignación.

Enunciado del problema de asignación

- **Datos del problema:**

- **n**: número de personas y de tareas disponibles.
- **B**: array [1..n, 1..n] de entero. Rendimiento o beneficio de cada asignación. **B[i, j]** = beneficio de asignar a la persona *i* la tarea *j*.

- **Resultado:**

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

- **Formulación matemática:**

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

[proceso](#)

5.3.2. Problema de asignación.

1) Representación de la solución

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

- $a_{ij} = 1 \rightarrow$ la tarea *j* se asigna a la persona *i*
- $a_{ij} = 0 \rightarrow$ la tarea *j* no se asigna a la persona *i*

		Tareas			
		a	1	2	3
Personas	1	0	1	0	
	2	1	0	0	
	3	0	0	1	

5.3.2. Problema de asignación.

1) Representación de la solución

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

5.3.2. Problema de asignación.

2) Elegir el esquema de algoritmo: caso optimización.

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

nivel:= 1; s:= s_{INICIAL}

voa:= $-\infty$; soa:= \emptyset

bact:= 0

bact: Beneficio actual

repetir

 Generar (nivel, s)

si Solución (nivel, s) AND (bact > voa) **entonces**

 voa:= bact; soa:= s

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

 nivel:= nivel + 1

sino

mientras NOT MasHermanos (nivel, s) AND (nivel>0)

hacer Retroceder (nivel, s)

fin

hasta nivel == 0

5.3.2. Problema de asignación.

3) Funciones genéricas del esquema.

- **Variabes:**
 - **s:** array [1..n] de entero: cada **s[i]** indica la tarea asignada a la persona **i**. Inicializada a 0.
 - **bact:** beneficio de la solución actual
- **Generar (nivel, s)** → Probar primero 1, luego 2, ..., n
s[nivel]:= s[nivel] + 1
si s[nivel]==1 **entonces** bact:= bact + B[nivel, s[nivel]]
sino bact:= bact + B[nivel, s[nivel]] – B[nivel, s[nivel]-1]
- **Criterio (nivel, s)**
para i:= 1, ..., nivel-1 **hacer**
 si s[nivel] == s[i] **entonces** devolver false
finpara
devolver true

5.3.2. Problema de asignación.

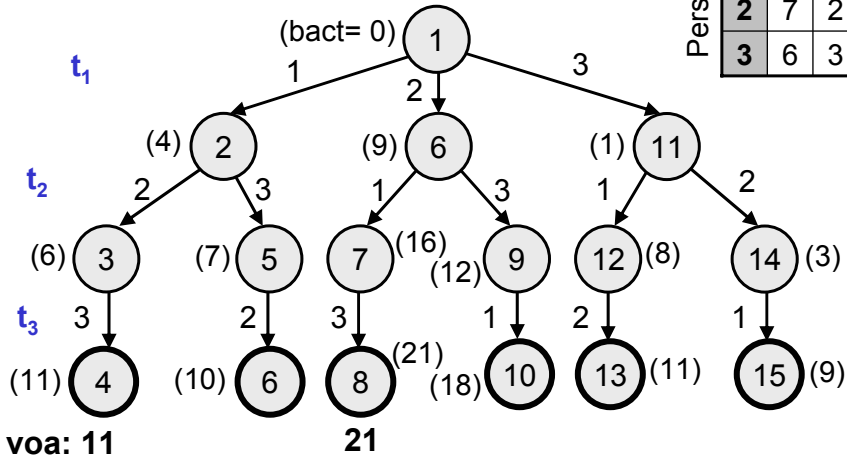
3) Funciones genéricas del esquema.

- **Solución (nivel, s)**
 devolver (nivel==n) AND Criterio (nivel, s)
- **MasHermanos (nivel, s)**
 devolver s[nivel] < n
- **Retroceder (nivel, s)**
 bact:= bact – B[nivel, s[nivel]]
 s[nivel]:= 0
 nivel:= nivel – 1

5.3.2. Problema de asignación. Tareas

- Ejemplo de aplicación. $n = 3$

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



5.3.2. Problema de asignación.

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

5.3.2. Problema de asignación.

3) Funciones genéricas del esquema.

- **Criterio (nivel, s)**
devolver usada[s[nivel]]==1
- **Retroceder (nivel, s)**
bact:= bact – B[nivel, s[nivel]]
usada[s[nivel]]--
s[nivel]:= 0
nivel:= nivel – 1
- **Generar (nivel, s)**
usada[s[nivel]]--
s[nivel]:= s[nivel] + 1
usada[s[nivel]]++
si s[nivel]==1 **entonces** bact:= bact + B[nivel, s[nivel]]
sino bact:= bact + B[nivel, s[nivel]] – B[nivel, s[nivel]-1]

5.3.2. Problema de asignación.

3) Funciones genéricas del esquema.

- Las funciones **Solución** y **MasHermanos** no se modifican.
- ¿Cuál es ahora el orden de complejidad del algoritmo?
- **Conclusiones:**
 - El algoritmo sigue siendo muy ineficiente.
 - Aunque garantiza la solución óptima...
 - ¿Cómo mejorar el tiempo?
 - Aplicar una poda según el criterio de optimización...



5.3.3. Resolución de juegos.

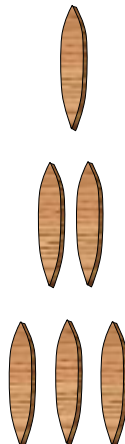
- La idea de backtracking (recorrido exhaustivo del árbol de un problema) se puede aplicar en **problemas de juegos**.
- **Objetivo final:** decidir el movimiento óptimo que debe realizar el jugador que empieza moviendo.

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.

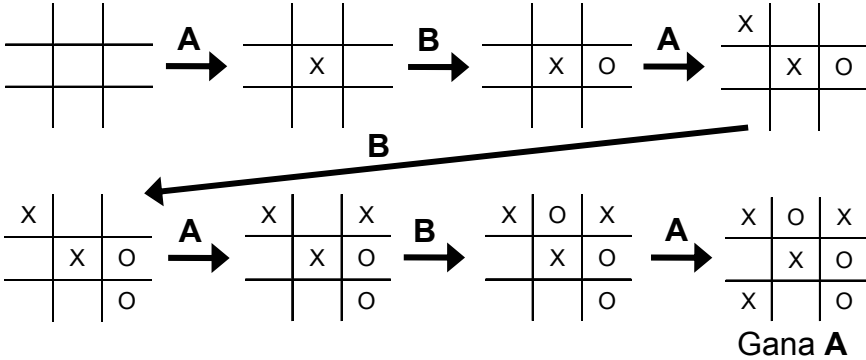
5.3.3. Resolución de juegos.

- **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.



5.3.3. Resolución de juegos.

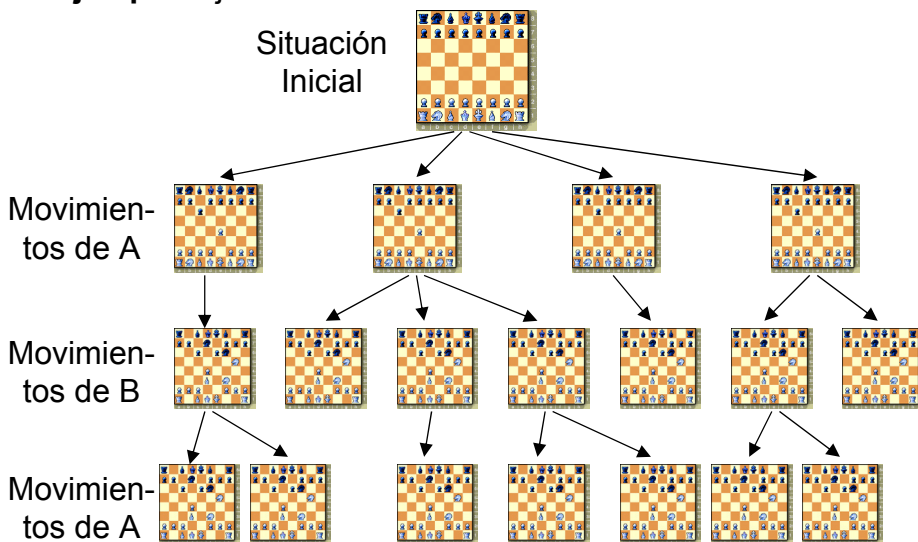
- **Ejemplo.** El juego de las tres en raya.



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

5.3.3. Resolución de juegos.

- **Ejemplo.** Ajedrez.



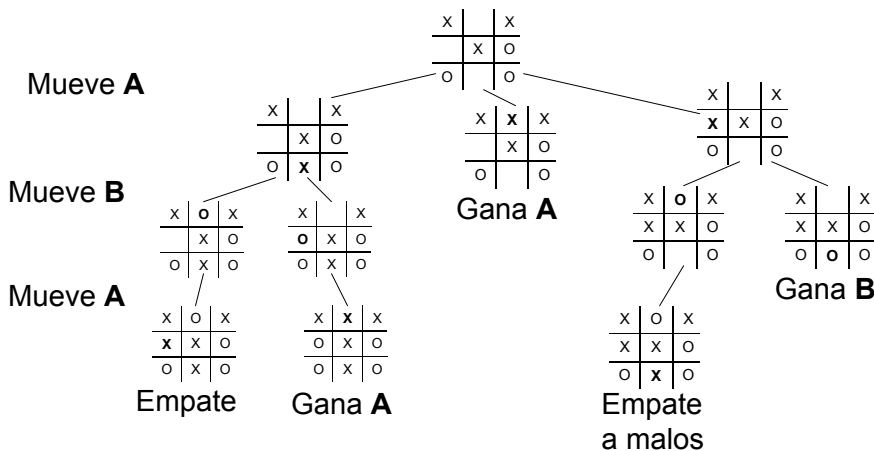
5.3.3. Resolución de juegos.

- **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.

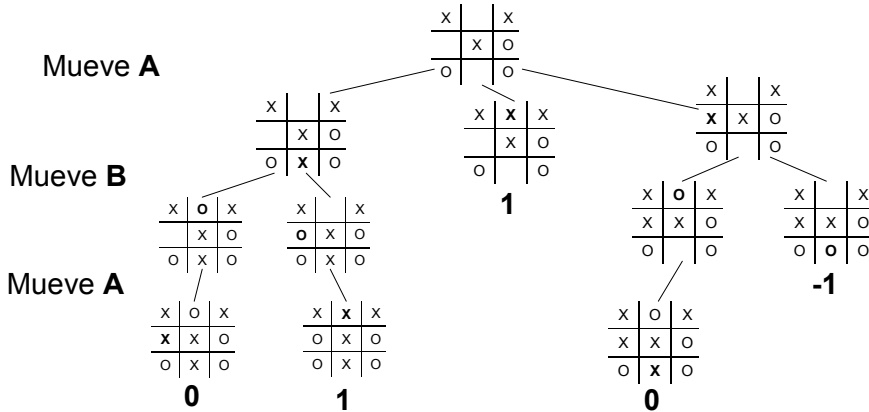
5.3.3. Resolución de juegos.

- **Ejemplo.** Parte del árbol de juego de las tres en raya.
A → X B → O



5.3.3. Resolución de juegos.

- 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.



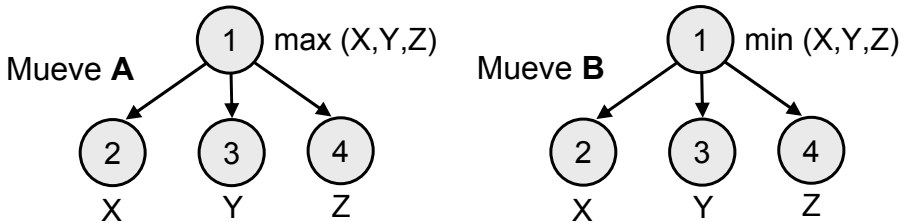
5.3.3. Resolución de juegos.

- 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**.

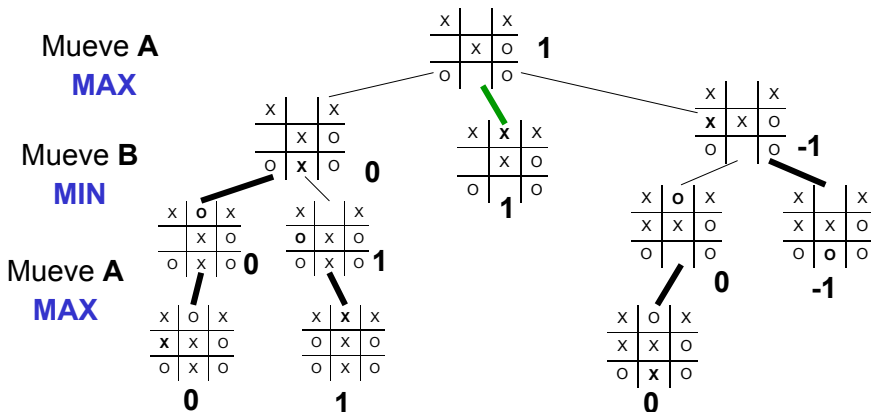
5.3.3. Resolución de juegos.

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).



5.3.3. Resolución de juegos.



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

5.3.3. Resolución de juegos.

- 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**.



A.E.D.

Tema 5. Backtracking.

67

5.3.3. Resolución de juegos.

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

A.E.D.

Tema 5. Backtracking.

68

5.3.3. Resolución de juegos.

- **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:= $-\infty$

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

A.E.D.

Tema 5. Backtracking.

69

5.3.3. Resolución de juegos.

- **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**.

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

A.E.D.

Tema 5. Backtracking.

70

5.3.3. Resolución de juegos.

- **Ejemplo.** El juego de los palillos.
- Representación del tablero:
tipo TipoTablero = array [1..n] de entero



- **Funciones:**

EsHoja (B)

devolver $N\text{Pal}(B) \leq 1$

Utilidad (B, modo)

si $(N\text{Pal}(B) == 0 \text{ AND } \text{modo} == \text{MAX})$ OR

$(N\text{Pal}(B) == 1 \text{ AND } \text{modo} == \text{MIN})$ entonces devolver 1

sino devolver -1

para cada hijo C del tablero B

para $i := 1, \dots, n$ hacer

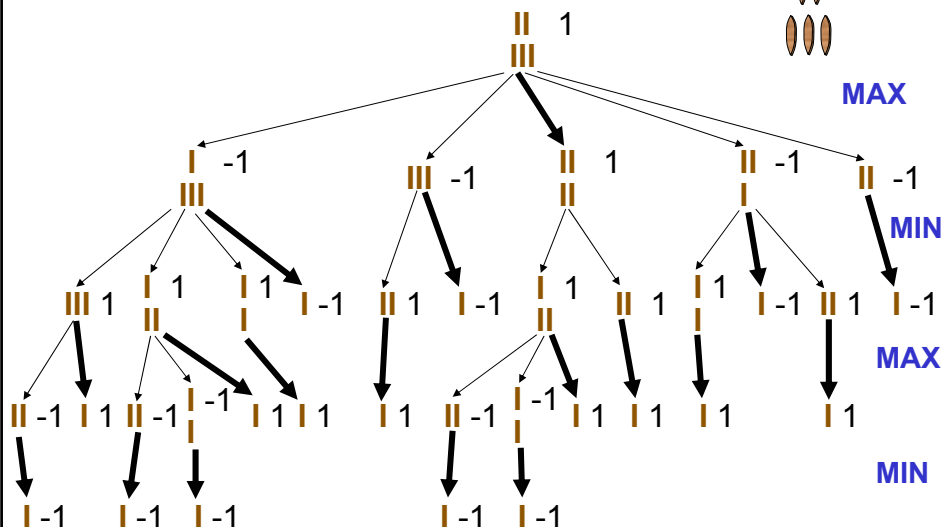
para $j := 1, \dots, B[i]$ hacer

C := B

C[i] := C[i] - j

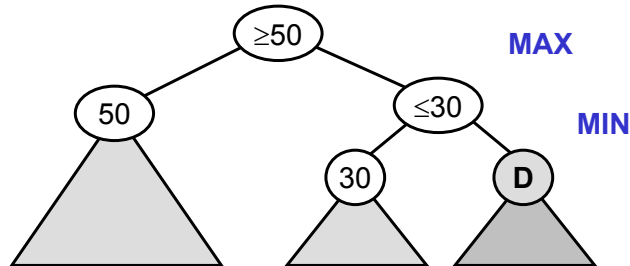
5.3.3. Resolución de juegos.

- **Ejemplo.** El juego de los palillos. $B = (2, 3)$



5.3.3. Resolución de juegos.

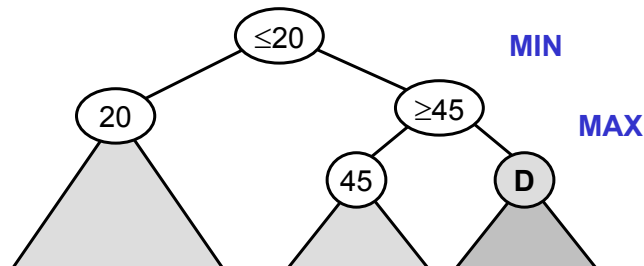
- Sobre los árboles de juegos se puede aplicar un tipo propio de poda, conocida como poda **alfa-beta**.
- **Poda Alfa:**
Supongamos que en cierto momento de la evaluación llegamos a la siguiente situación.



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

5.3.3. Resolución de juegos.

- **Poda Beta:**
Supongamos que en cierto momento de la evaluación llegamos a la siguiente situación.



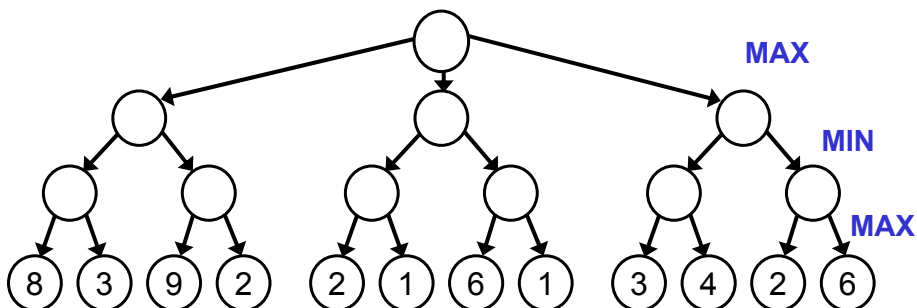
- Haya lo que haya en **D**, nunca estará el movimiento óptimo.
- **Conclusión:** podar el nodo **D** y sus descendientes.

5.3.3. Resolución de juegos.

- 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:= $-\infty$
 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 \geq valorPadre entonces salir del para**
 sino
 valoract:= min (valoract, BuscaMinimax(C, valoract, MAX))
 { P. alfa} **si valoract \leq valorPadre entonces salir del para**
 finsi
 finpara
 devolver valoract
finsi

5.3.3. Resolución de juegos.

- ¿Cómo sería la llamada inicial al procedimiento **BuscaMinimax**?
- **Ejemplo.** Aplicar la estrategia minimax con poda alfa-beta al siguiente árbol de juego.



5. Backtracking.

Conclusiones:

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