PARTE II: ALGORÍTMICA

Tema 7. Ramificación y poda.

7.1. Método general.

- La **ramificación y poda** (**branch and bound**) se suele utilizar en problemas de **optimización discreta** y en problemas de **juegos**.
- Puede ser vista como una **generalización** (o mejora) de la técnica de **backtracking**:
  - La ramificación y poda realiza un **recorrido sistemático** en un árbol de soluciones.
  - El recorrido no tiene porqué ser necesariamente en profundidad. Tendremos una **estrategia de ramificación**.
  - Se tratará como un aspecto importante las **técnicas de poda**, para eliminar nodos que no lleven a soluciones optimas.
  - La poda se realiza **estimando** en cada nodo **cotas** del beneficio que podemos obtener a partir del mismo.

7.1. Método general.

- Para cada nodo **i** tendremos:
  - **Cota superior** (**CS(i)**) y **Cota inferior** (**CI(i)**) del beneficio (o coste) óptimo que podemos alcanzar a partir de ese nodo. ® Determinan cuándo se puede realizar una poda.
  - **Estimación del beneficio (o coste)** óptimo que se puede encontrar a partir de ese nodo. Puede ser una media de las anteriores. ® Ayuda a decidir qué parte del árbol evaluar primero.

Estrategia de poda

- Supongamos un problema de **maximización**.
- Hemos recorrido varios nodos 1..n, estimando para cada uno la cota superior **CS (j)** e inferior **CI (j)**, respectivamente, para **j** entre 1 y n.

7.1. Método general.

- **CASO 1.** Si a partir de un nodo siempre podemos obtener alguna solución válida, entonces **podar un nodo i si**:

  CS (i) £ CI (j), para algún nodo j generado.

- **Ejemplo:** problema de la mochila, utilizando un árbol binario.
  - A partir de **a**, podemos encontrar un beneficio máximo de CS(a)=4.
  - A partir de **b**, tenemos garantizado un beneficio mínimo de CI(b)=5.
  - Podemos podar el nodo **a**, sin perder ninguna solución óptima.

- **CASO 2.** Si a partir de un nodo puede que no lleguemos a una solución válida, entonces podemos **podar un nodo i si**:

  CS (i) £ Beneficio (j), para algún j, solución final (factible).

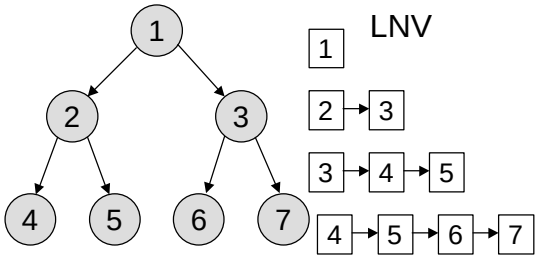
- **Ejemplo:** problema de las reinas. A partir de una solución parcial, no está garantizado que exista alguna solución.

7.1. Método general.

Estrategias de ramificación

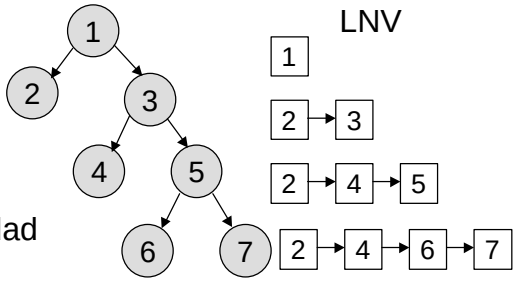
- Normalmente el árbol de soluciones es implícito, no se almacena en ningún lugar.
- Para hacer el recorrido se utiliza una **lista de nodos vivos**.
- **Lista de nodos vivos:** contiene todos los nodos que han sido generados pero que no han sido explorados todavía. Son los nodos pendientes de tratar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

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



7.1. Método general.

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



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

7.1. Método general.

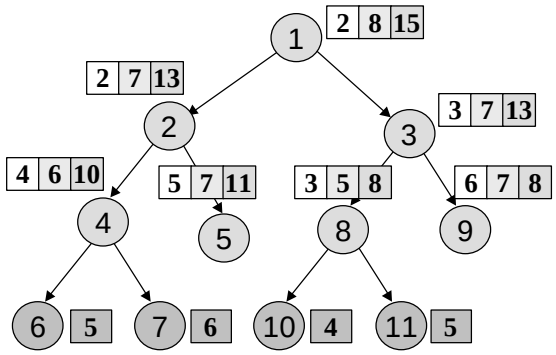
Estrategias de ramificación LC

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

- En cada nodo tenemos: cota inferior de coste, coste estimado y cota superior del coste.
- Podar según los valores de las cotas.
- Ramificar según los costes estimados.

7.1. Método general.

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



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

7.1. Método general.

Algunas cuestiones

- Sólo se comprueba el criterio de poda cuando se introduce o se saca un nodo de la lista de nodos vivos.
- Si un descendiente de un nodo es una solución final entonces no se introduce en la lista de nodos vivos. Se comprueba si esa solución es mejor que la actual, y se actualiza **C** y el valor de la mejor solución óptima de forma adecuada.
- ¿Qué pasa si a partir de un nodo solución pueden haber otras soluciones (ejemplo: árbol combinatorio)?
- ¿Cómo debe ser actualizada la variable **C** (variable de poda) si el problema es de maximización, o si tenemos el caso 2 (a partir de un nodo puede que no exista ninguna solución)?
- ¿Cómo será la poda, para cada uno de los casos anteriores?
- ¿Qué pasa si para un nodo **i** tenemos que **CI(i)** = **CS(i)**?
- ¿Cuándo acaba el algoritmo?
- ¿Cómo calcular las cotas?

7.1. Método general.

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

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

7.2. Análisis de tiempos de ejecución.

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

7.3. Ejemplos de aplicación.

7.3.1. Problema de la mochila 0/1.

- **Diseño del algoritmo** de ramificación y poda:
  - Definir una representación de la solución. A partir de un nodo, cómo se obtienen sus descendientes.
  - Dar una manera de calcular el valor de las cotas y la estimación del beneficio.
  - Definir la estrategia de ramificación y de poda.
- **Representación de la solución:**
  - **Mediante un árbol binario**: $(s_1, s_2, ..., s_n)$, con $s_i = (0, 1)$.
    Hijos de un nodo $(s_1, s_2, ..., s_k)$: $(s_1, ..., s_k, 0)$ y $(s_1, ..., s_k, 1)$.
  - **Mediante un árbol combinatorio**: $(s_1, s_2, ..., s_m)$ donde $m \leq n$ y $s_i \in \{1, 2, ..., n\}$.
    Hijos de un nodo $(s_1, .., s_k)$:
    $(s_1, .., s_k, s_k+1)$, $(s_1, ..., s_k, s_k+2)$, ..., $(s_1, ..., s_k, n)$

7.3.1. Problema de la mochila 0/1.

- **Cálculo de cotas:**
  - **Cota inferior**: Beneficio que se obtendría incluyendo sólo los objetos incluidos hasta ese nodo.
  - **Estimación del beneficio**: A la solución actual, sumar el beneficio de incluir los objetos enteros que quepan, utilizando avance rápido. Suponemos que los objetos están ordenados por orden decreciente de v_i/w_i.
  - **Cota superior**: Resolver el problema de la mochila no 0/1 a partir de ese nodo (con un algoritmo voraz), y quedarse con la parte entera. (Es la misma que la usada en backtracking.)

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

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

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

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

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

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

7.3.1. Problema de la mochila 0/1.

- **Forma de realizar la poda:**
  - En una variable **C** guardar el valor de la mayor cota inferior hasta ese momento dado.
  - Si para un nodo, su cota superior es menor o igual que **C** entonces se puede podar ese nodo.
- **Estrategia de ramificación:**
  - Puesto que tenemos una estimación del coste, usar una estrategia **LC**: explorar primero las ramas con mayor valor esperado **(MB)**.
  - ¿LC-FIFO ó LC-LIFO? Usaremos la LC-LIFO: en caso de empate seguir por la rama más profunda. (MB-LIFO)

```
type nodo = record
              v_act, w_act: integer;
              CI, BE, CS: integer;
              tupla: array [1..n] of integer;
              nivel: integer;
            end;
```

7.3.1. Problema de la mochila 0/1.

```
Mochila01RyP (v, w: array [1..n] of integer; M: integer; var s: nodo);
    inic:= NodoInicial (v, w, M);
    C:= inic.CI;
    LNV:= {inic};
    s.v_act:= -∞;
    Mientras LNV ≠ ∅ hacer
        x:= Seleccionar (LNV);              { Según el criterio MB-LIFO }
        LNV:= LNV - {x};
        Si  x.CS > C  Entonces              { Si no se cumple se poda x }
            Para  i:= 0, 1  Hacer
                y:= Generar (x, i, v, w, M);
                Si (y.nivel = n)  Y  (y.v_act > s.v_act)  Entonces
                    s:= y;
                    C:= max (C, s.v_act );
                Sino Si  (y.nivel < n)  Y  (y.CS > C)  Entonces
                    LNV:= LNV + {y};
                    C:= max (C, y.CI );
            FinPara;
        FinSi;
    FinMientras;
```

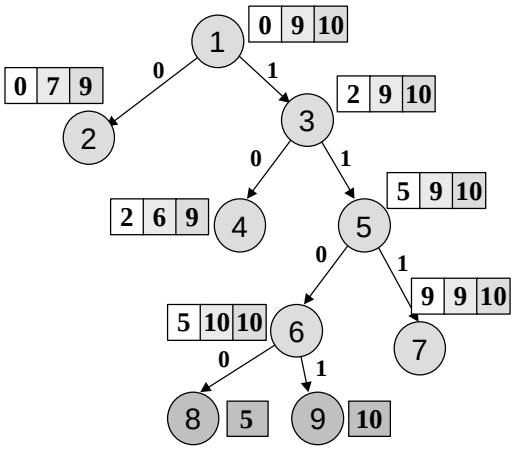

7.3.1. Problema de la mochila 0/1.

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

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

7.3.1. Problema de la mochila 0/1.

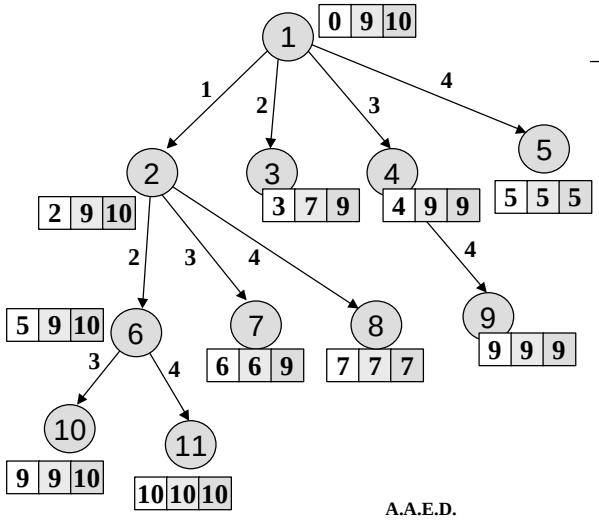
- **Ejemplo**. n = 4, M = 7, v = (2, 3, 4, 5), w = (1, 2, 3, 4)



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

7.3.1. Problema de la mochila 0/1.

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



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

7.3.2. Secuenciamiento de trabajos.

- Es una generalización del problema de la planificación con plazo fijo (del tema 4, algoritmos voraces).
- Tenemos **n** trabajos y un solo procesador.
- **Cada trabajo i tiene:**
  - **Duración: t_i**
  - **Plazo de ejecución: d_i**
  - **Penalización si no se ejecuta: p_i**
- Dado un trabajo, o bien empieza a ejecutarse dentro de su plazo **d_i** (requiriendo un tiempo **t_i**) o bien no se ejecuta, produciendo una penalización **p_i**.
- **Objetivo:** hacer una planificación de las tareas de forma que se minimice la penalización de las tareas no ejecutadas.

7.3.2. Secuenciamiento de trabajos.

Diseño de la solución

- Igual que en el algoritmo voraz, dado un conjunto de tareas el orden de ejecución será en orden creciente de plazo, d_i.
- **Representación de la solución:**
  - Representación **binaria**: (s_1, s_2, ..., s_n), con s_i = (0, 1).
- **Cálculo de cotas:**
  - **Cota inferior:** penalización de los trabajos asignados con valor 0 hasta este nodo (son los trabajos que no se ejecutan).
  - **Cota superior:** la cota inferior más la penalización de los trabajos no considerados hasta este momento.
  - **Coste estimado:** podemos aproximarlo con la media de las dos cotas anteriores.

7.3.2. Secuenciamiento de trabajos.

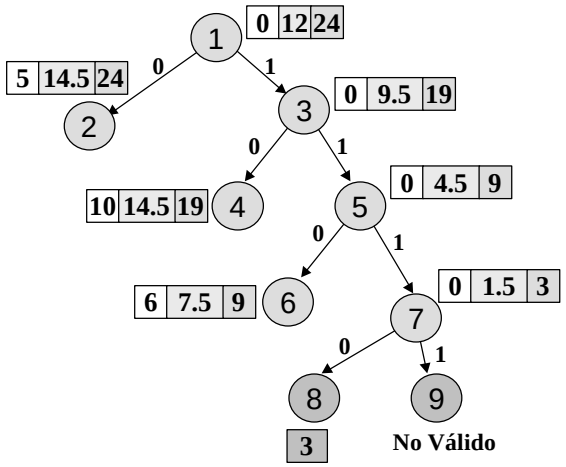
- **Ejemplo.** n = 4, p = (5, 10, 6, 3), d = (1, 3, 2, 1),
  t = (1, 2, 1, 1)

  Nodo actual: (1, 0, 1)
  Cota inferior: CI = p_2 = 10
  Cota superior: CS = CI + p_4 = 10 + 3 = 13
  Estimación del coste: CE = (CI + CS)/2 = (10+13)/2 = 11.5

- **Forma de realizar la poda:**
  - El problema es de minimización y a partir de cada nodo existe al menos una solución. **C**: valor de la menor cota superior hasta ese punto.
  - Podar si la cota inferior es mayor que **C**.
- **Estrategia de ramificación.**
  - Supondremos LC-FIFO.

7.3.2. Secuenciamiento de trabajos.

- **Ejemplo.** n = 4, p = (5, 10, 6, 3), d = (1, 3, 2, 1), t = (1, 2, 1, 1)
  Estrategia LC-FIFO



C	LNV
24	1
19	3 → 2
9	5 → 2
3	7 → 2
3	2

7.3.2. Secuenciamiento de trabajos.

- También sería posible usar árboles combinatorios.
- ¿Es buena la forma de calcular las cotas?
- **Otro posible cálculo de cotas: s** = (s_1, s_2, ..., s_k)
  - **Cota superior:** el coste actual de **s** más el coste de resolver el problema usando un algoritmo voraz, con las tareas (s_{k+1}, ..., s_n) y los "huecos" disponibles.
  - **Cota inferior:** el coste actual de **s** más el coste de resolver el problema usando un algoritmo voraz, con las tareas (s_{k+1}, ..., s_n) y los "huecos" disponibles, suponiendo los plazos (d_{k+1}, ..., d_n) iguales al plazo máximo.
  - **Coste estimado:** la media de las dos cotas anteriores.

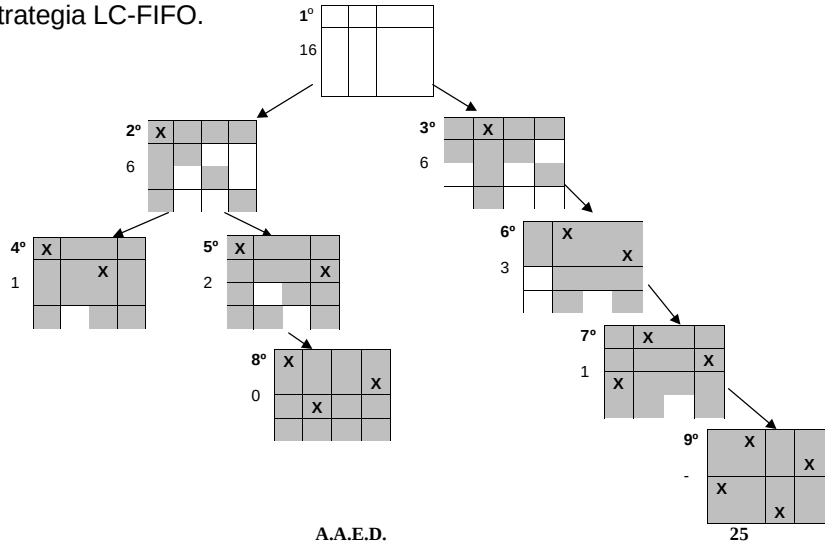
7.3.3. Problema de las n reinas.

- **Problema:** Dado un tablero de ajedrez de tamaño **n**x**n**, encontrar una forma de colocar **n** reinas, sin que ninguna de ellas se coma a otra.
- No es un problema de optimización, pero podemos usar ramificación y poda para mejorar la búsqueda.
- **Estimación del beneficio**: se usará para indicar qué ramas son las más prometedoras. Tendremos alguna estrategia LC.
- **Cota inferior y superior**: no se usan, se supondrán con valor -∞ e +∞, respectivamente. → No se realizará ninguna poda.
- Se acabará el proceso al encontrar una solución.
- La medida de lo **prometedora** que es una situación del tablero es una medida **heurística**.
- **Ejemplo**: Una configuración será mejor cuantas más casillas hayan no alcanzables por las reinas colocadas en el mismo.

  Beneficio de un tablero = Nº de casillas libres en el tablero

7.3.3. Problema de las n reinas.

- Recorrido del árbol en el problema de las 4 reinas.
  Estimación del beneficio = Número de casillas libres.
  Estrategia LC-FIFO.



7.3.3. Problema de las n reinas.

- **Problema**: cuando estamos llegando al final, el número de casillas es muy pequeño, y la estimación favorece a nodos de nivel superior.
- **Solución**: mejorar la heurística de estimación de beneficios.

- **Posibilidad 1**: Para favorecer que hayan casillas libres en las filas inferiores, multiplicar el número de casillas en cada nivel por el número de ese nivel.

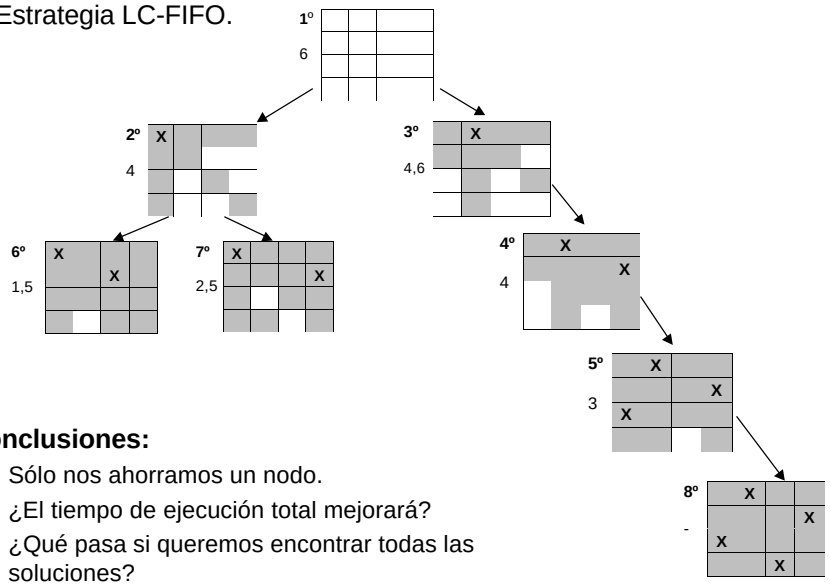
  $Beneficio_2 = \sum$ Nº casillas libres en cada fila*Nº de fila

- **Posibilidad 2**: Tener en cuenta el nivel por el que vamos. Si el nivel es próximo a n, estamos cerca de una solución. Si es próximo a 1, estamos cerca de la raíz.

  $Beneficio_3 = Beneficio_2$ /Nº de reinas que quedan por colocar

7.3.3. Problema de las n reinas.

- Recorrido del árbol en el problema de las 4 reinas, utilizando $Beneficio_3$ y Estrategia LC-FIFO.



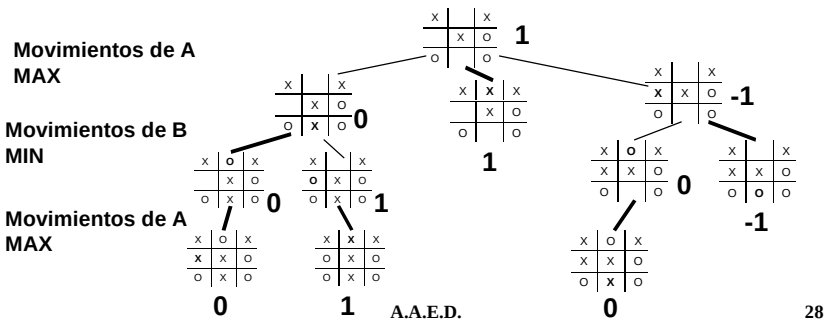
- **Conclusiones:**
  - Sólo nos ahorramos un nodo.
  - ¿El tiempo de ejecución total mejorará?
  - ¿Qué pasa si queremos encontrar todas las soluciones?

7.3.4. Resolución de juegos. Poda alfa-beta.

- Consideraremos juegos con dos jugadores, **A** y **B**, que mueven alternativamente (primero A y luego B), con el objetivo de ganar.

Utilizando Backtracking

- **Función de utilidad:** para cada nodo hoja devuelve un valor numérico, indicando cómo de buena es esa situación para el jugador A.
- **Estrategia minimax:**
  - Movimientos de **A**: el nodo padre tendrá el **máximo** de los valores de los hijos.
  - Movimientos de **B**: el nodo padre tendrá el **mínimo** de los valores de los hijos.
  - Se repite (recorriendo en profundidad) hasta llegar al nodo raíz.

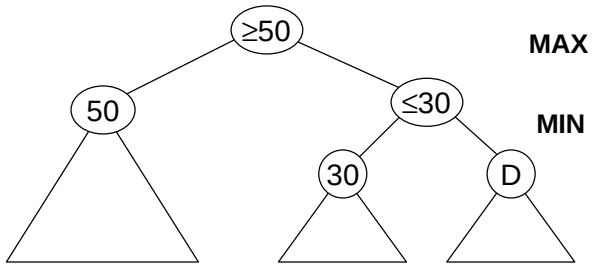


7.3.4. Resolución de juegos. Poda alfa-beta.

- Podemos mejorarlo utilizando **ramificación y poda**:
  - **Ramificación**: Realizar estimaciones del beneficio para explorar primero por los movimientos más prometedores.
  - **Poda**: Eliminar movimientos que no conduzcan a soluciones mejores.
- La propagación de valores en un árbol de juegos sigue la estrategia **minimax**. ¿Cómo realizar la poda en este caso?

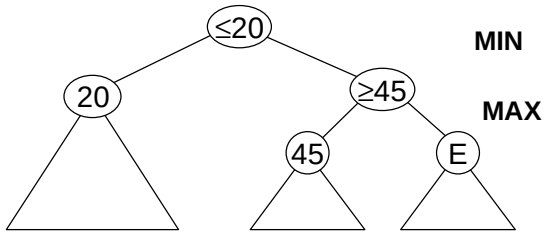
Poda alfa-beta

- En un punto de la evaluación llegamos a la siguiente situación.

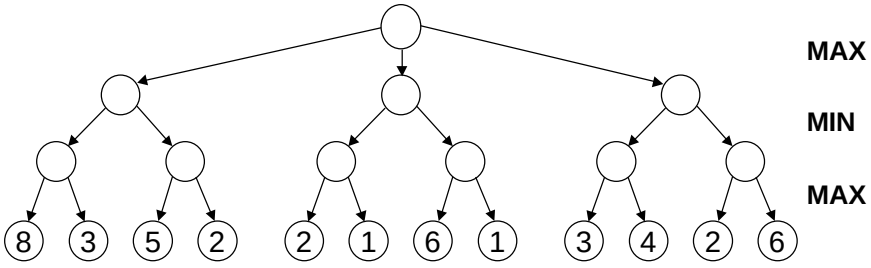


- **Poda alfa:** Podemos descartar el nodo D y sus descendientes.

7.3.4. Resolución de juegos. Poda alfa-beta.



- **Poda beta:** Podemos descartar el nodo E y sus descendientes.
- **Ejemplo**. Aplicar un recorrido en profundidad con poda alfa-beta.



7.3.4. Resolución de juegos. Poda alfa-beta.

- **Implementación recursiva.**

```
MinimaxAB (B: tipo_tablero; valor_padre: real; modo: (MAX, MIN)) : real;
    if B es una hoja then
        Devolver Utilidad (B);
    else begin
        if modo = MAX then
            valor_act:= -∞;
        else
            valor_act:= +∞;
        for cada hijo C del tablero B do
            if modo = MAX then
                valor_act:= max (valor_act, MinimaxAB (C, valor_act, MIN));
{ Poda beta }       if valor_act ³ valor_padre then
                        Salir del for, descartando los demás hijos de B;
            else { modo = MIN }
                valor_act:= min (valor_act, MinimaxAB (C, valor_act, MAX));
{ Poda alfa }       if valor_act £ valor_padre then
                        Salir del for, descartando los demás hijos de B;
        Devolver valor_act;
    end;
```

- **Llamada inicial:** MinimaxAB (TableroInicial, +∞, MAX);