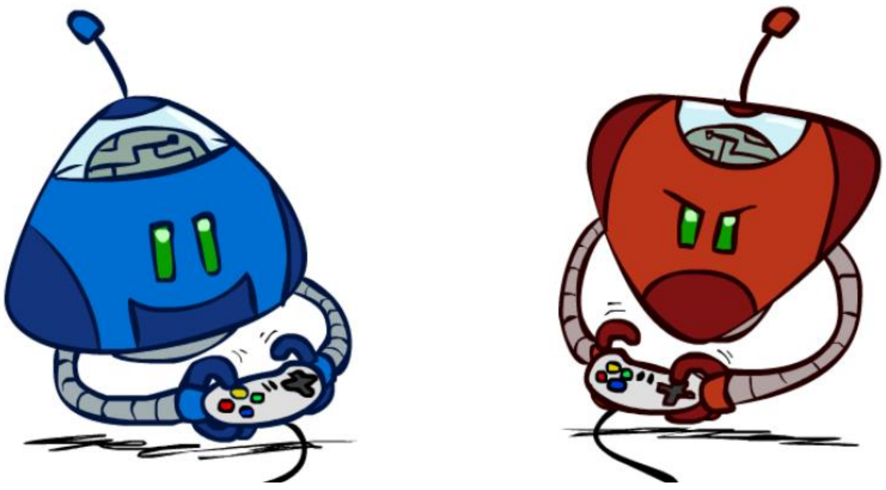
Grado en ingeniería informática
Inteligencia Artificial 2020/2021

2.3. Búsqueda con adversarios



Universidad Autónoma de Madrid

Búsqueda con Adversarios: Juegos

- Métodos informados o heurísticos
  - Introducción
  - Búsqueda primero el mejor
  - Algoritmos de mejora iterativa
  - Búsqueda con adversarios
    - Introducción
    - Minimax
    - Poda alfa-beta
    - Consideraciones prácticas

Lecturas :

- CAPÍTULO 6 de Russell & Norvig
- CAPÍTULO 12 de Nilsson

Algunas figuras de http://aima.cs.berkeley.edu/

Introducción

- **Historia:**
  - "The theory of Games and Economic Behavior", 1944 John von Neumann, Oskar Morgenstern.
  - Ideas:
    - Estrategia óptima para un jugador: Zermelo (1912), Von Neumann (1928)
    - Limitaciones de recursos (evaluación aproximada de la función de utilidad): Konrad Zuse (1945), Norbert Wiener (1948), Claude Shannon (1950)
    - Primera propuesta para el ajedrez: Alan Turing (1951).
    - Aprendizaje en el juego de damas: Arthur Samuel (1952-57)
    - Poda del árbol de juego: MacCarthy (1956)
- **Entorno multiagente** + **competición**

  Los agentes tienen objetivos diferentes ⇒ búsqueda con adversarios.
    - **Idealización** en la que los jugadores con **objetivos opuestos** alternan **turnos** en los que realizan acciones (movimientos o jugadas).
    - Los agentes sólo pueden realizar "**movimientos permitidos**" (tal como definen las reglas del juego).
    - Los movimientos se eligen de acuerdo a una **estrategia** que especifica un movimiento por cada posible movimiento del oponente.
    - El juego termina cuando uno de los agentes alcanza su objetivo (cuantificado mediante una **función de utilidad**).

Tipos de juegos

- Criterios de clasificación:
  - **Número de jugadores**
    - Dos jugadores (ej. backgammon, ajedrez, damas)
    - Multijugador: estrategias mixtas (competitivas / cooperativas, ej. alianzas temporales).
  - Propiedades de la **función de utilidad**
    - **Suma-cero**: La suma de utilidades de los agentes es cero, independientemente del resultado del juego (ej. ajedrez, damas)
    - **Suma constante**: La suma de utilidades de los agentes es constante, independientemente del resultado del juego (equivalente a juegos de suma-cero: normalización de la utilidad)
    - **Suma variable**: No son de suma cero. Pueden tener estrategias óptimas complejas, que a veces involucran colaboración (ej. monopoly, backgammon)
  - **Información** de la que disponen los jugadores
    - **Información perfecta** (ej. ajedrez, damas, go)
    - **Información parcial** (ej. casi todos los juegos de cartas)
  - Elementos de **azar**
    - **Deterministas** (ej. ajedrez, damas, go)
    - **Estocásticos** (ej. backgammon)
  - Tiempo ilimitado / limitado (ej. ajedrez con reloj).
  - Movimientos ilimitados / limitados (ej. ajedrez con máximo de 40 movimientos).

Búsqueda con adversarios

- **Problema de búsqueda:**
  - **Estado inicial**.
  - **Función sucesor**:
    (move estado-actual) → estado-sucesor
  - **Test terminal**: determina si el estado del juego es un estado terminal (es decir, el juego ha concluido)
  - **Utilidad (función objetivo o de pago)**: Valoración numérica de los estados terminales.

  Árbol del juego: Estado inicial + movimientos legales alternados.

- Consideremos el siguiente juego sencillo:

  Dos jugadores realizan movimientos alternados hasta que uno de ellos gana (el otro pierde) o hay un empate.

  Cada jugador tiene un modelo perfecto del entorno determinista y de los efectos que producen los movimientos legales.

  Puede haber limitaciones computacionales / temporales a los movimientos de los agentes.

  - Dos agentes con objetivos opuestos que alternan jugadas
  - Información perfecta
  - Determinista
  - Juego de suma cero o suma constante

Búsqueda con adversario

- Un árbol de juego es una representación explícita de todas las posibles secuencias de jugadas en una partida
  - El nodo raíz corresponde al estado inicial de la partida.
  - Los jugadores alternan sus movimientos
  - Para generar el siguiente nivel a partir de un nodo dado, se genera tantos nodos hijos como posibles movimientos tenga el jugador que tiene el turno en el nodo considerado.
  - Las hojas corresponden a estados terminales (fin de partida)
  - Un camino desde la raíz (el estado inicial de la partida) hasta una hoja representa una partida completa
- Los algoritmos de búsqueda vistos hasta ahora no sirven
- El problema ya no es encontrar un camino en el árbol de juego (puesto que esto depende de los movimientos futuros que hará el oponente), sino decidir el mejor movimiento dado el estado actual del juego

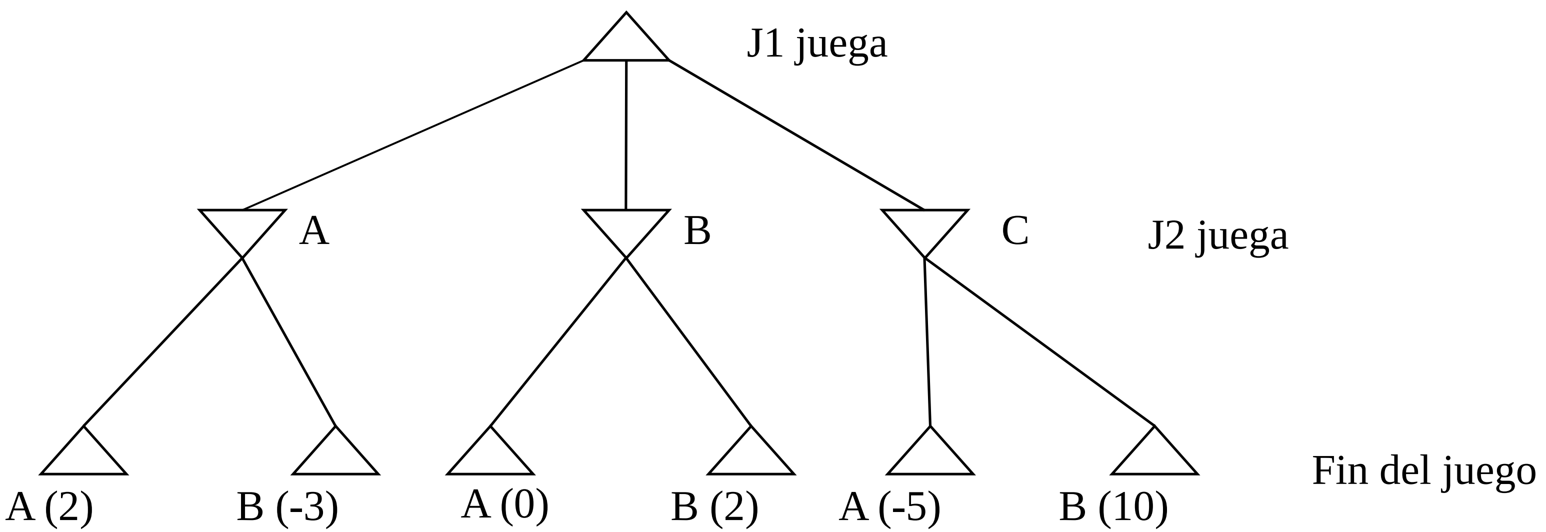
Representación de juegos

- Matriz de balance final: Valor de la función de utilidad para cada jugador, dadas las acciones de los otros jugadores

		J2	
		A	B
J1	A	2	-3
	B	0	2
	C	-5	10

J2 paga a J1

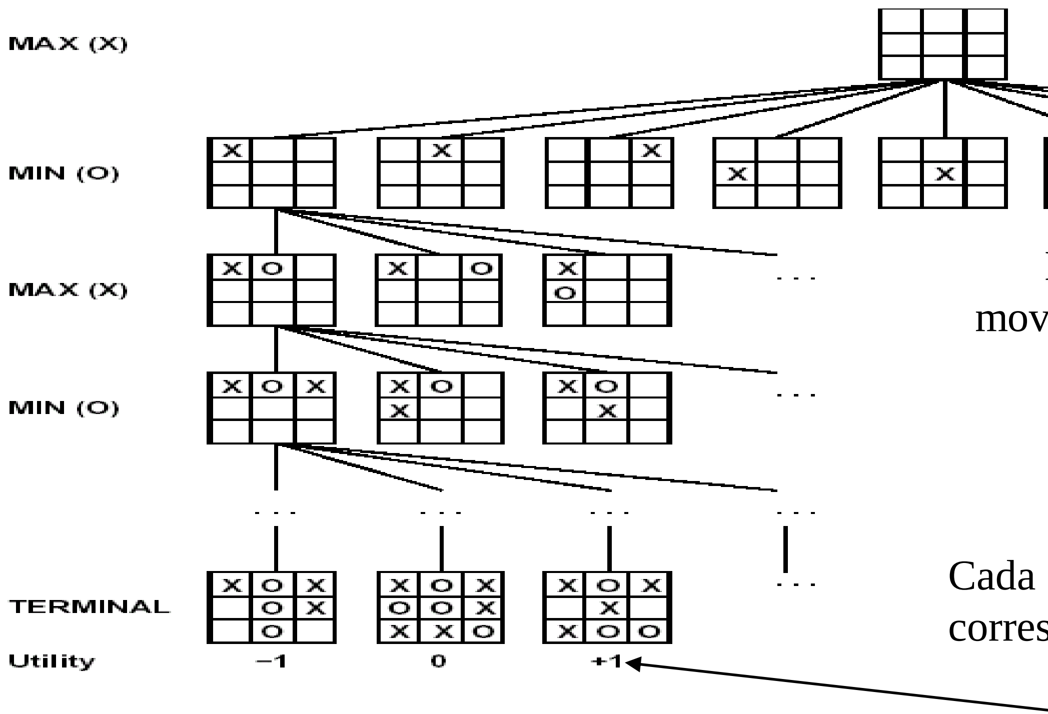
- Árbol del juego



Árbol de juego para tres en raya

- Estado inicial: Tablero 3x3 vacío
- Movimientos: Poner una ficha (MAX: x, MIN: o) en una de las casillas vacías.
- Test terminal: 3 fichas del mismo jugador están alineadas

Inicialmente MAX puede poner una ficha en cualquiera de las 9 casillas vacías.



MAX (x) y MIN (o) alternan movimientos hasta que se alcanza un estado terminal

Cada estado terminal está etiquetado por el valor correspondiente de la función de utilidad

Estrategias óptimas

- Consideremos un juego con dos jugadores: **MAX** y **MIN**.
  - **MAX mueve primero**.
  - **MAX y MIN alternan sus movimientos**: En el árbol del juego, los nodos de profundidad par (impar) corresponden a MAX (MIN).
  - Una **dupla** de profundidad **k** en el árbol del juego corresponde a los nodos del árbol del juego de profundidades 2k y 2k+1.
- Descripción formal del juego:
  - Estado inicial: Configuración inicial del tablero + identidad del primer jugador.
  - Función sucesor: Sucesores(n)
  - Test para determinar si un estado es terminal (fin de partida): Terminal(n).
  - Función de utilidad: Utilidad(n) es el valor numérico asignado al estado terminal n desde el punto de vista de MAX.
- **Estrategia óptima para MAX**: Estrategia que obtiene un resultado al menos tan bueno como cualquier otra estrategia, asumiendo que MIN juega de manera óptima.
- **Estrategia minimax**:

  Usar el **valor minimax** de un nodo para guiar la búsqueda: **Utilidad de un nodo** (desde el punto de vista de MAX) **asumiendo que ambos jugadores juegan de manera óptima.**

$$\text{minimax}(n) = \begin{cases} \text{Utilidad}(n) & \text{si n es un nodo terminal.} \\ \max\{\text{minimax}(s);\ s \in \text{sucesores}(n)\} & \text{si n es un nodo MAX.} \\ \min\{\text{minimax}(s);\ s \in \text{sucesores}(n)\} & \text{si n es un nodo MIN.} \end{cases}$$

El algoritmo minimax

```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game
    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))
```

```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
    return v
```

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do v ← MIN(v, MAX-VALUE(s))
    return v
```

- **Completo** sólo si **el árbol del juego es finito** (puede haber estrategias óptimas finitas para árboles infinitos)
- **Óptima sólo si el oponente es óptimo** (si el oponente es subóptimo, es posible aprovechar sus puntos débiles para encontrar estrategias mejores. Peligroso).
- **Complejidad temporal exponencial** $O(b^m)$;

  m = profundidad máxima del árbol del juego
- **Complejidad espacial**: **Lineal** si se usa **búsqueda-primero-en-profundidad** O(b·m)

Minimax

- El jugador que tiene el turno es MAX. El oponente es MIN
- Son nodos MAX (MIN) aquéllos en los que tiene el turno MAX (MIN)
  - El nodo raíz es un nodo MAX (profundidad 0).
  - Los nodos de profundidad **par** son nodos **MAX**
  - Los nodos de profundidad **impar** son nodos **MIN**
- A un nodo terminal se le asigna un valor de utilidad desde el punto de vista de MAX.

  Ej. Ajedrez, codificado como un juego de
  suma cero: gana MAX (+1); tablas (0); gana MIN (-1)
  suma constante: gana MAX (2); tablas (1); gana MIN (0)
- El valor MINIMAX de un nodo se obtiene propagando los valores de la función de utilidad correspondientes a los nodos terminales hacia la raíz del árbol:
  - **Valor MINIMAX de un nodo MAX = máximo de los valores MINIMAX de sus hijos.**

    MAX intenta maximizar su ventaja eligiendo el mejor movimiento posible en su turno
  - **Valor MINIMAX de un nodo MIN = mínimo de los valores MINIMAX de sus hijos.**

    MIN intenta minimizar la puntuación de MAX eligiendo el movimiento que más perjudica MAX
- La secuencia de jugadas resultante es la estrategia MINIMAX.

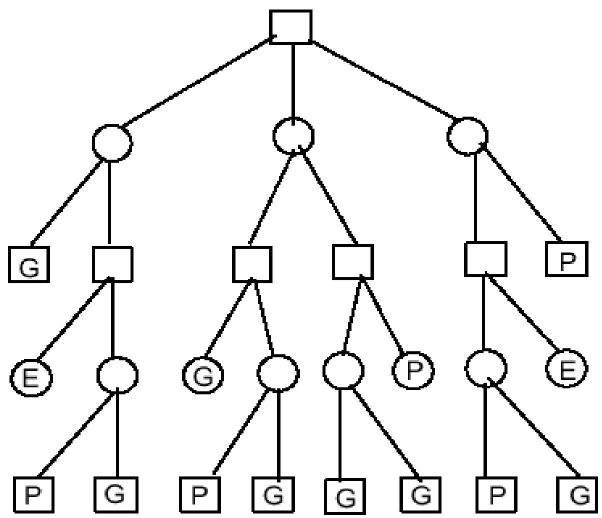
Minimax

- El valor MINIMAX del nodo raíz corresponde al mejor valor de la función de utilidad que MAX puede alcanzar en la partida suponiendo que el oponente es óptimo.
  - Aunque el resultado sea el óptimo, puede que MAX no pueda ganar la partida.

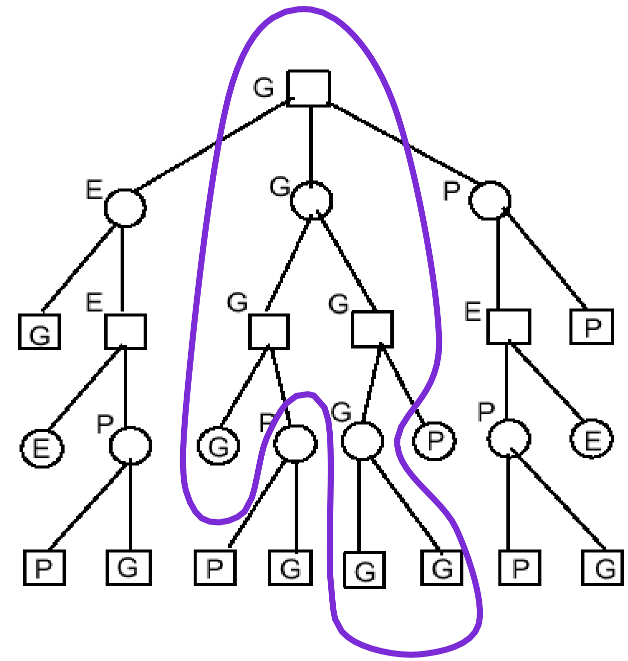
- Resolver un árbol de juego significa encontrar el valor MINIMAX del nodo raíz mediante el algoritmo MINIMAX.

Ejemplo

Árbol de juego sin resolver



Árbol de juego resuelto



Árbol ganador para MAX

G = 1, E = 0, P = -1

Nodos MAX: cuadrados

Nodos MIN: círculos

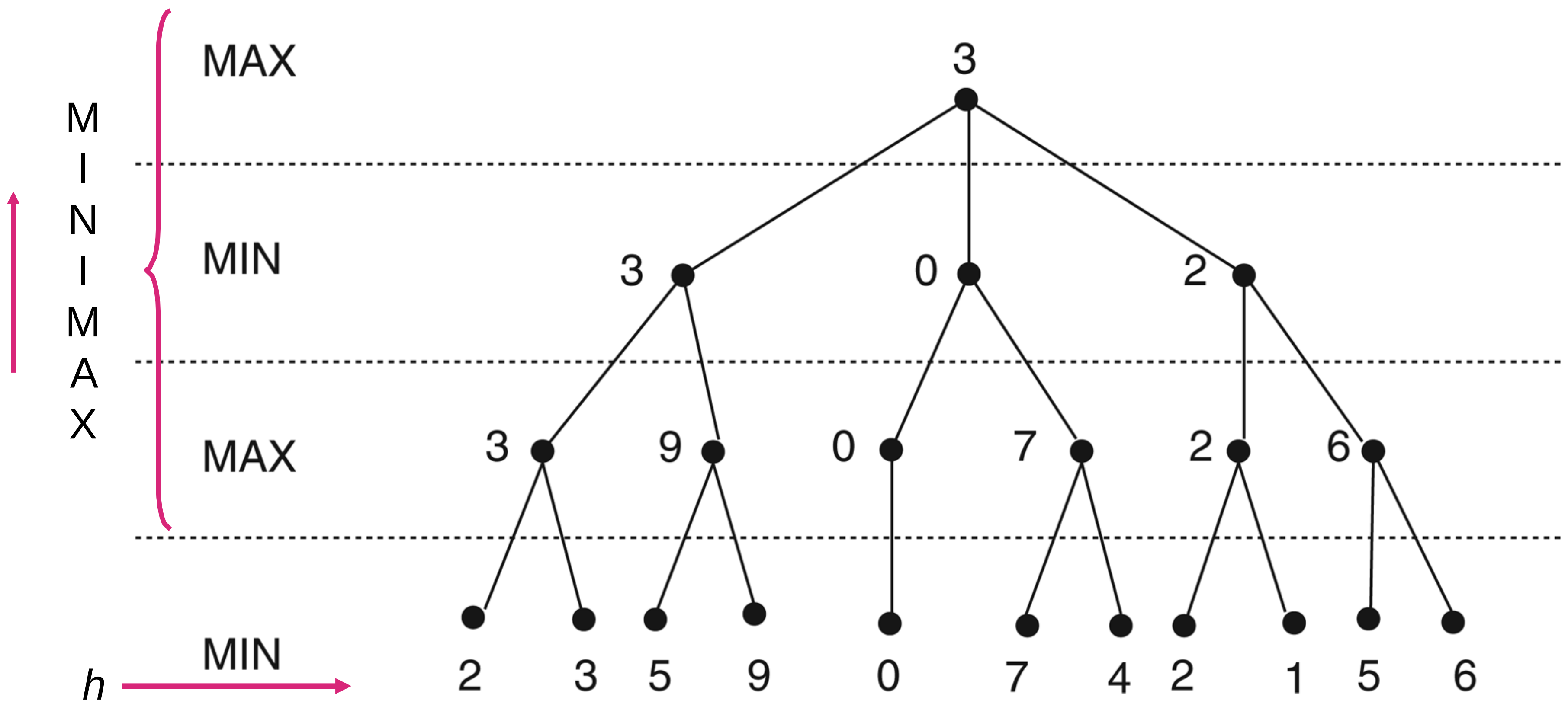
Minimax

- El árbol de juego se genera mediante **búsqueda primero en profundidad**
  - La complejidad espacial del algoritmo MINIMAX es lineal en la profundidad máxima del árbol (m) y en el factor de ramificación (b): O(bm)

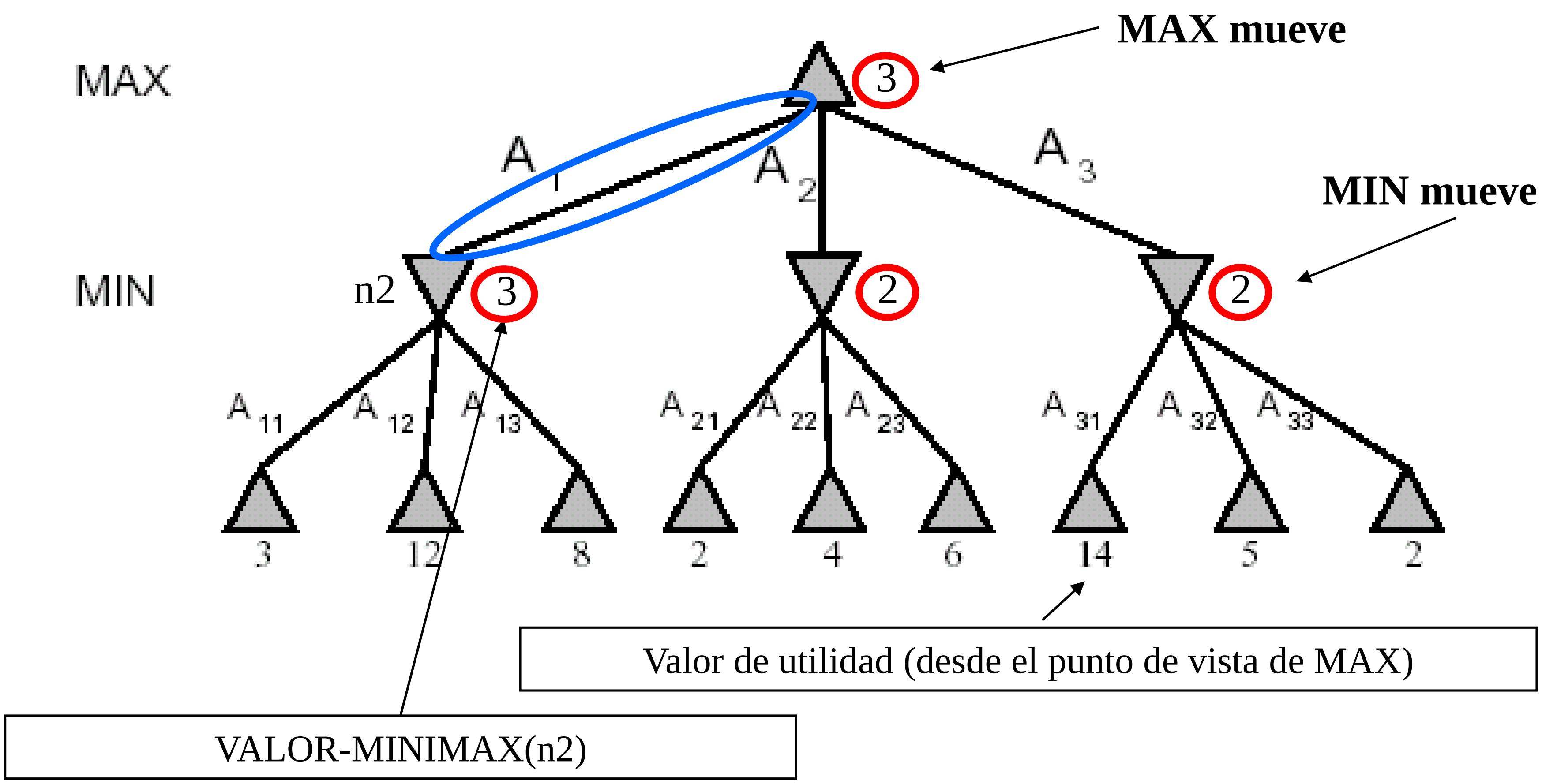
    No es necesario mantener el árbol completo en memoria
  - La complejidad temporal del algoritmo MINIMAX es exponencial: $O(b^m)$
    - El método de etiquetado descrito requiere generar el árbol de juego completo.
    - En la práctica, para la mayoría de los juegos, desarrollar el árbol de juego de manera completa es una tarea impracticable
      - Damas: El árbol completo tiene aproximadamente 10^{40} nodos
        - Suponiendo que se generan Generar el árbol completo requeriría 10^{21} siglos (3 billones nodos/seg.)
      - Ajedrez: unos 10^{120} nodos y unos 10^{101} siglos
    - Incluso si un adivino nos proporcionase un árbol etiquetado sería muy costoso almacenarlo o recorrerlo para encontrar la solucion
  - En la práctica se realiza una prepoda del árbol: Se establece un límite de profundidad de la búsqueda y se etiquetan los nodos terminales (no necesariamente correspondientes a finales de partida) mediante el valor de una función de evaluación (idealmente, relacionado de manera monótona con el valor minimax).

Ejemplo 1: valor minimax de 3 capas

Cálculo del valor minimax mirando hacia delante 3 capas (niveles)



Procedimiento minimax: ejemplo 2



- El mejor movimiento para MAX es A_1 (maximiza la utilidad)
- El mejor contra-movimiento para MIN es A_{11} (minimiza la utilidad)

Decisiones imperfectas

- **Problema**:
  - Normalmente la **función de utilidad** es **demasiado costosa** de computar.
  - Con **recursos limitados** o juegos con árboles infinitos es imposible realizar una búsqueda completa.
- **Solución:** Búsqueda con horizonte limitado
  - Definir una **función de evaluación** heurística **eval(n)**, que es una **estimación** de la verdadera función de utilidad **utilidad(n)**.
  - Usar un **test de corte** para determinar cuándo parar la búsqueda y computar **eval(n)**.
  - **Propiedades de eval(n):**
    - **eval(n)** debería ordenar los nodos terminales en el mismo orden que **utilidad(n)**.
    - **eval(n)** debería estar muy correlacionada con **minimax(n)** en nodos no terminales.
    - **eval(n)** debería ser sencilla de computar.

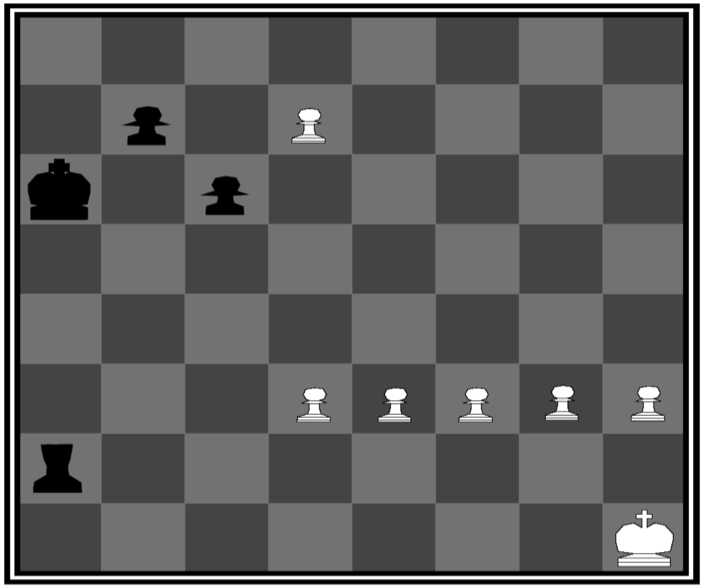
Búsqueda con horizonte limitado

Cambiar en búsqueda minimax:

"If test-terminal(estado) then return utilidad(estado)" $\Rightarrow$

"If test-corte(estado, profundidad) then return eval(estado)"

- Búsqueda primero-en-profundidad limitada en profundidad: Parar la búsqueda a un valor fijado de profundidad.
- Búsqueda primero-en-profundidad con profundidad iterativa: Más robusto si hay limitaciones en el tiempo.
- **Efecto del horizonte**: Desastre/éxito puede estar justo tras el horizonte de búsqueda.
  - No parar la búsqueda en posiciones "activas". Parar sólo si el estado es **inactivo**.
  - **Extensiones singulares**: Explorar posiciones más profundas que son claramente ventajosas.
  - **Poda hacia delante**: Quitar ramas de búsqueda que son claramente inferiores (peligro: podemos perdernos un sacrificio genial).



Mueven las negras

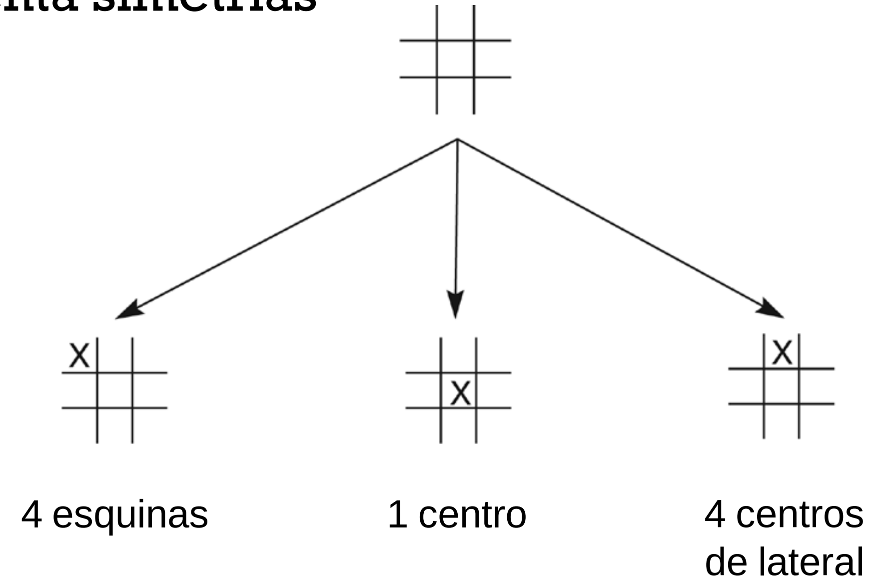
La función de evaluación

- En los nodos terminales (de final de partida), el valor de la función de evaluación debe coincidir con el de la función de utilidad.
- En nodos no terminales (ej. si se ha establecido un límite de profundidad en la búsqueda), es una estimación del valor minimax en dicho nodo.
- La función de evaluación *(heurística)* tiene en cuenta distintas características del estado del juego:
  - Número de piezas propias y del contrario (ventaja de uno sobre el otro)
  - Ponderación de la importancia de las piezas en ajedrez
  - Posiciones de las piezas en el tablero
  - Puntos débiles (peón aislado, etc.)
  - Características posicionales (protección del rey, capacidad de maniobra, control del centro del tablero, etc.)
- El valor de la función de evaluación debe poder calcularse de manera eficiente: Cuanto más tiempo se emplee en el cómputo de dicho valor, menos tiempo se puede dedicar a la búsqueda.

  Es necesario alcanzar un compromiso entre la calidad de la estimación y el coste computacional necesario para obtener dicha estimación.

Ejemplo: las 3 en raya

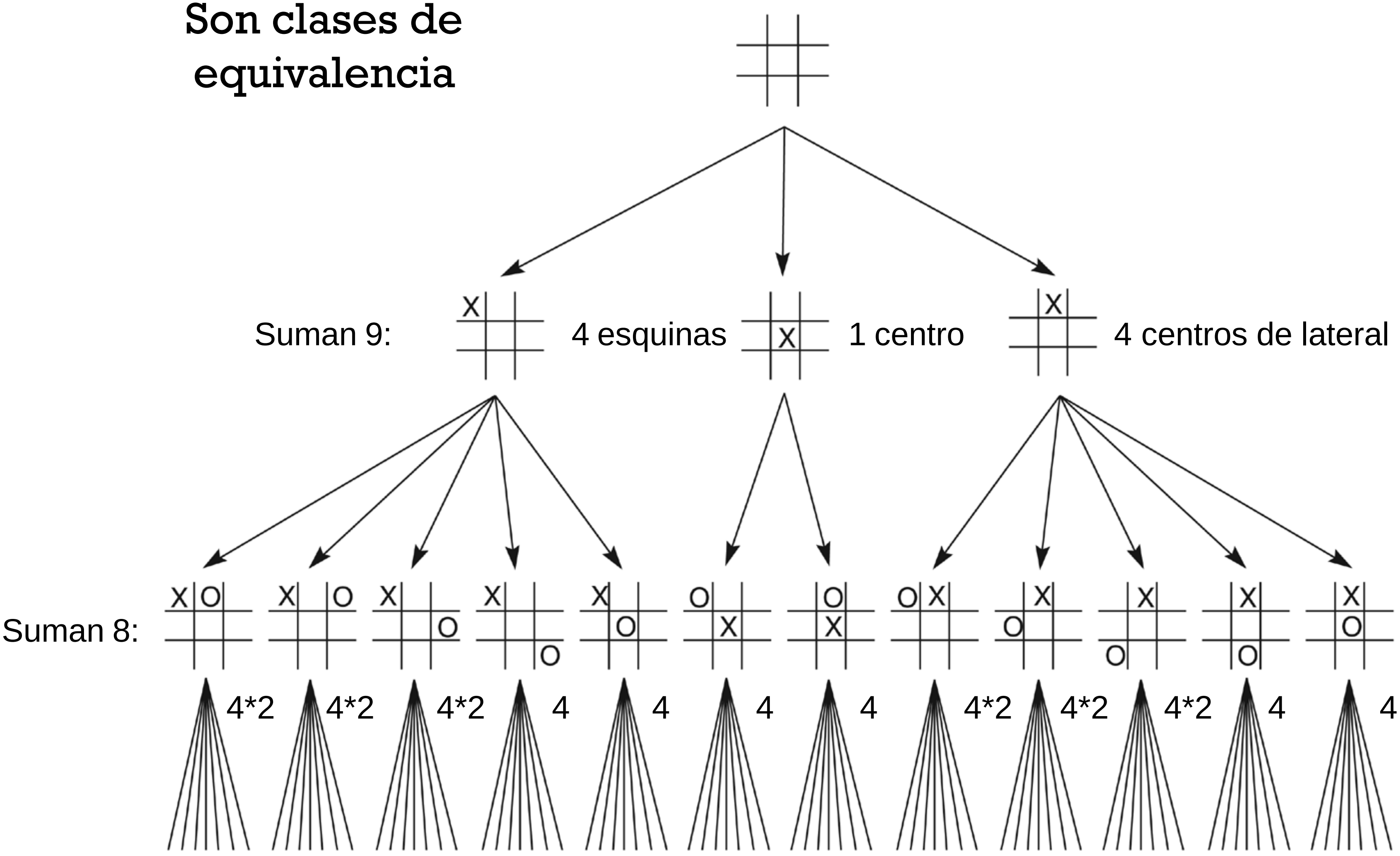
- Representación directa:
  - 9 movimientos iniciales, 8 posibles segundos movimientos, ...
  - Muchos operadores y "estados" distintos (configuraciones de tablero)
- Representación que tiene en cuenta simetrías
  - 3 movimientos iniciales
    - esquina
    - centro del tablero
    - centro de un lateral



  - Las 6 configuraciones restantes son equivalentes a alguna de las consideradas por simetría.
  - Reduce el espacio de estados y, por lo tanto, la complejidad de la búsqueda

Ejemplo: reducción simétrica en las 3 en raya

Son clases de equivalencia



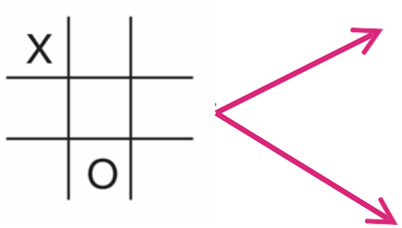
Ejemplo: función de evaluación para el tres en raya

<u>Función de utilidad</u>:

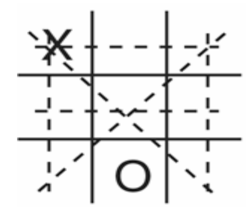
- $-\infty$ si en el estado terminal gana MIN (y, por lo tanto, pierde MAX)
- $+\infty$ si en el estado terminal gana MAX (y, por lo tanto, pierde MIN)

<u>Función de evaluación</u>: *eval(n) = M(n) – O(n)*

- *M(n)* = n° de posibles líneas ganadoras de MAX *(contando vacías)*
- *O(n)* = n° de posibles líneas ganadoras de MIN *(contando vacías)*

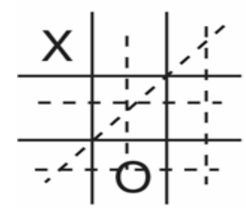


X tiene 6 posibles líneas ganadoras

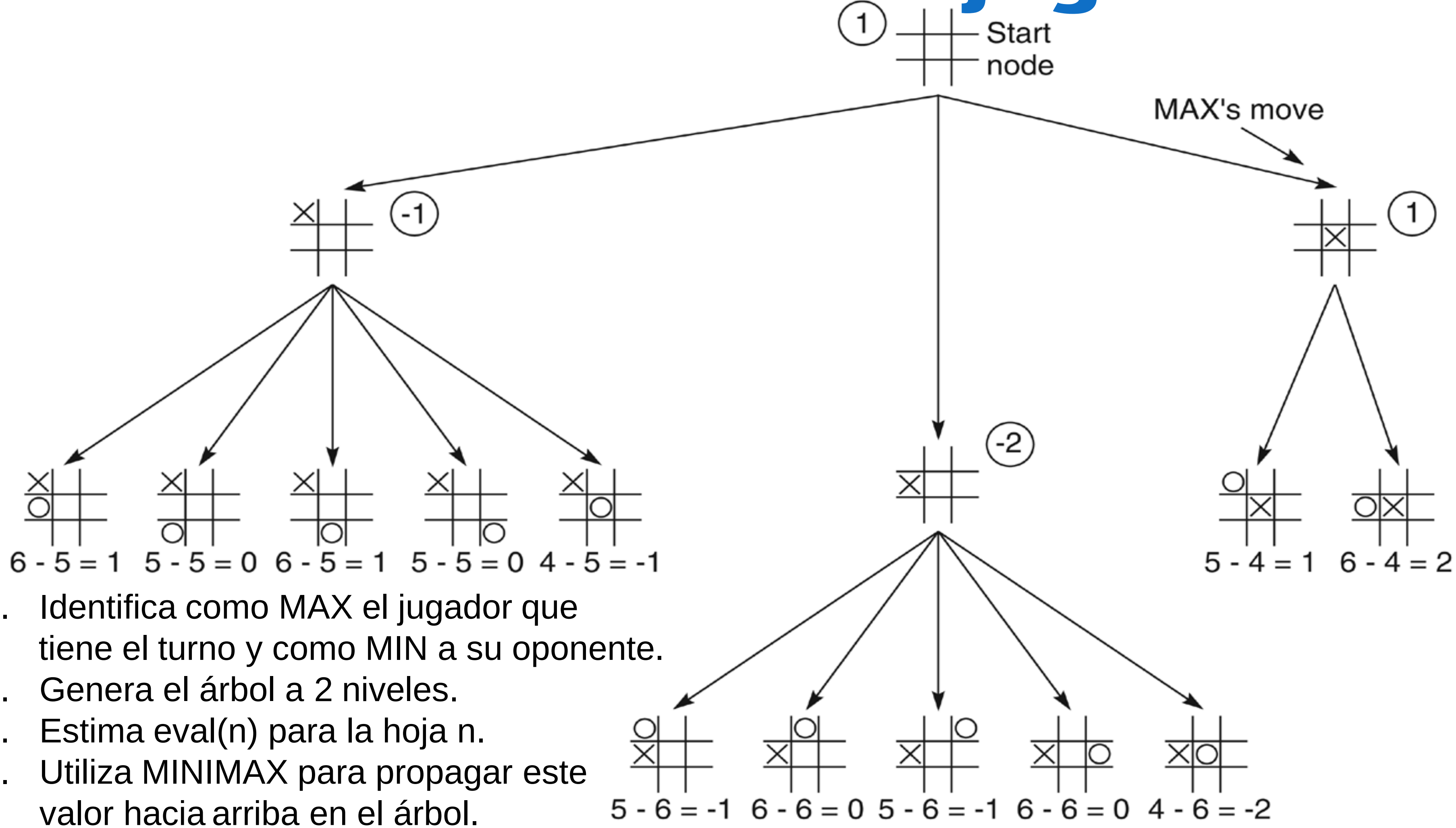


O tiene 5 posibles líneas ganadoras

h(n) = 6 – 5 = 1



MINIMAX con límite de profundidad 2: Movimiento inicial del jugador 'x'

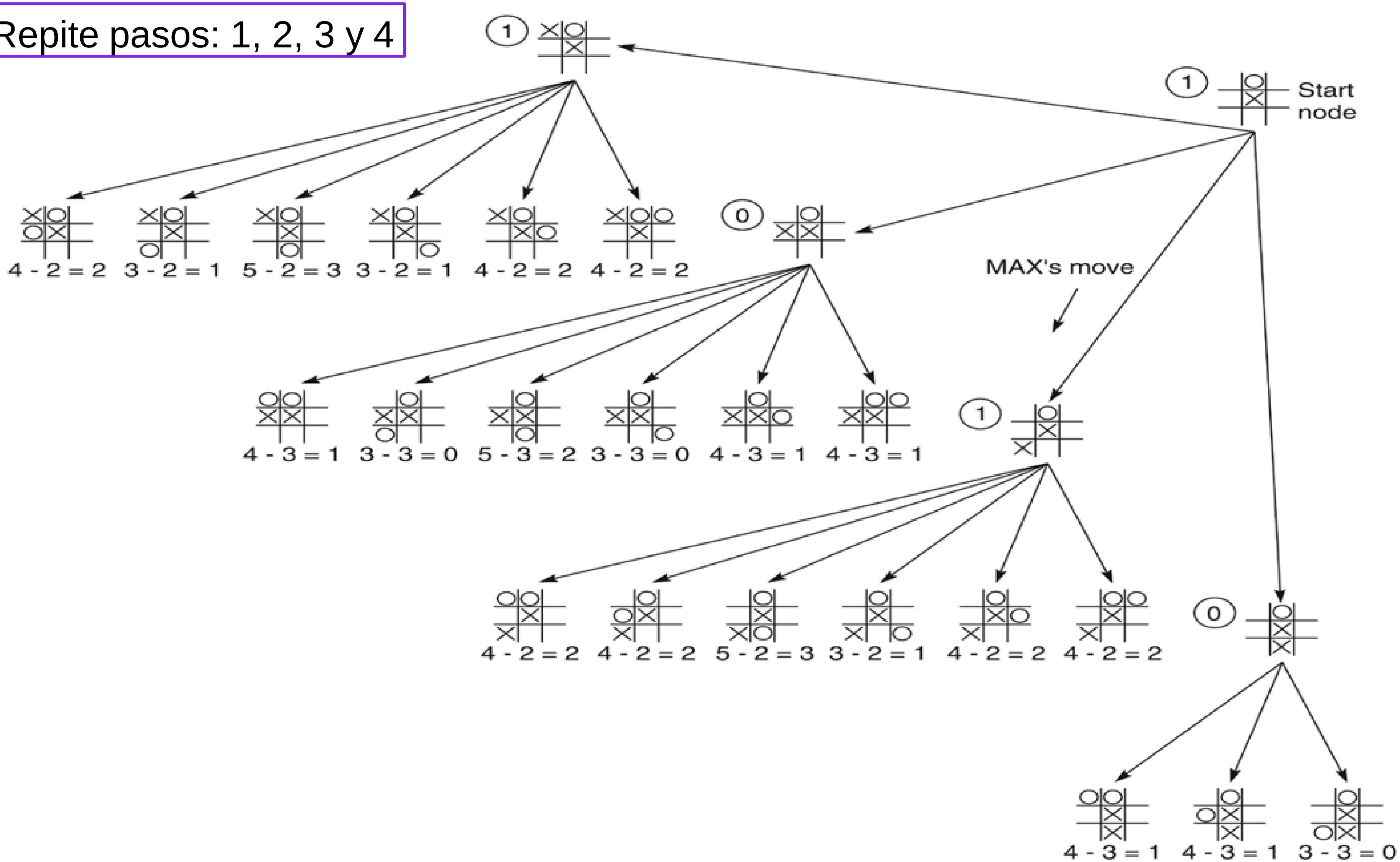


1. Identifica como MAX el jugador que tiene el turno y como MIN a su oponente.
2. Genera el árbol a 2 niveles.
3. Estima eval(n) para la hoja n.
4. Utiliza MINIMAX para propagar este valor hacia arriba en el árbol.

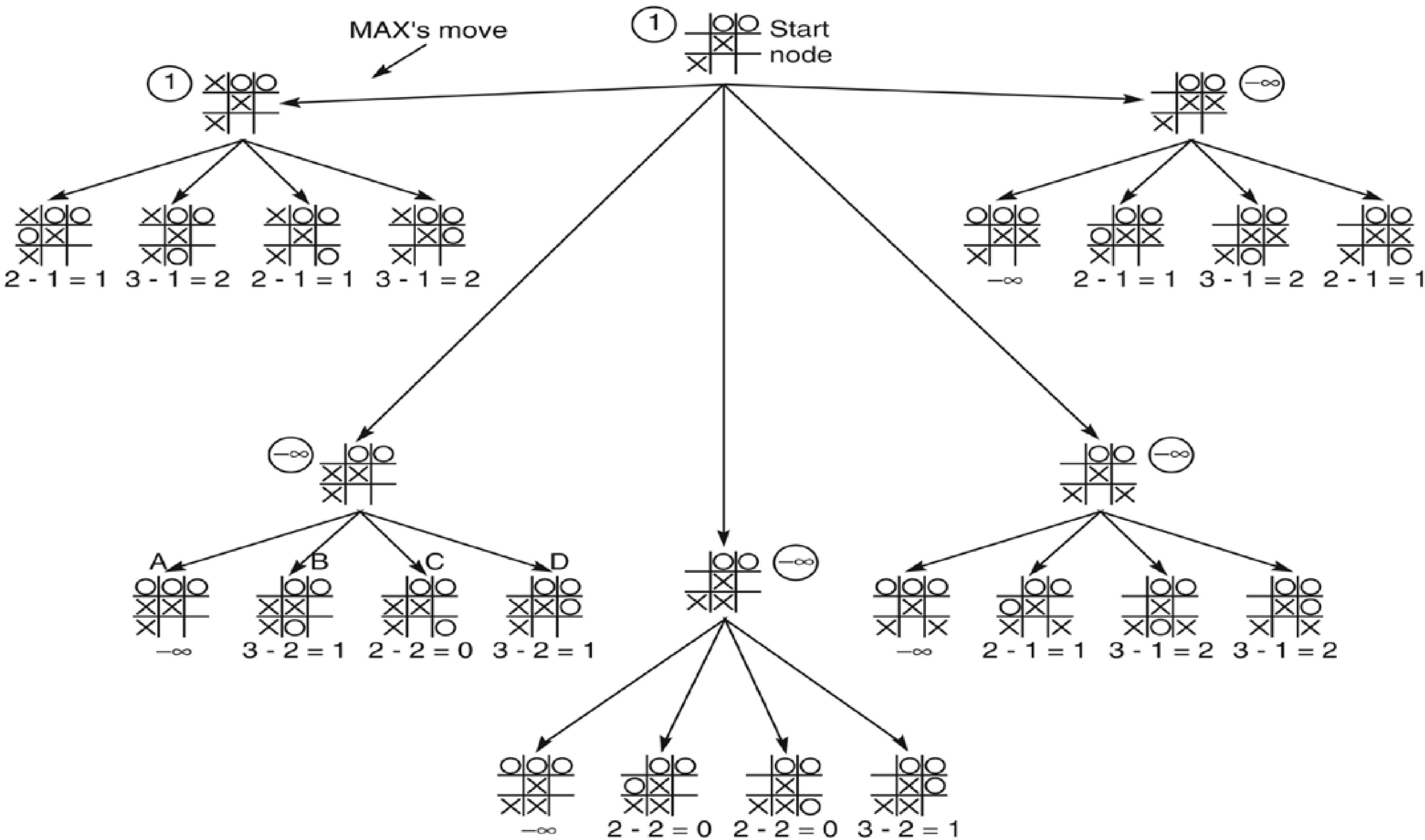
1. MAX identifica el mejor movimiento y lo realiza (se modifica el estado de la partida).
2. Repite el paso 1 alternando jugadores (hay cambio de turno) hasta fin de partida.

MINIMAX con límite de profundidad 2: Segundo movimiento del jugador 'x'

Repite pasos: 1, 2, 3 y 4



MINIMAX con límite de profundidad 2: Tercer movimiento del jugador 'x'



Funciones de evaluación

¿Cómo construimos funciones de evaluación buenas? Calcular el **valor esperado** de la utilidad estimando las "probabilidades" de diferentes resultados finales desde la posición actual.

$$\text{utilidad\_esperada(n)} = \sum_{\text{resultados}} \text{Probabilidad(n} \rightarrow \text{resultado)} \times \text{utilidad(resultado)}$$

Ej., 50% probabilidades de ganar (utilidad 1)
25% probabilidades de perder (utilidad -1)
25% probabilidades de empatar (utilidad 0)
f=0.50*1 + 0.25*(-1) + 0.25*0 = 0.25

- **Evaluación basada en características**:
  Codificar el estado actual por un conjunto de **características** $\{f_1(n), f_2(n), \ldots, f_K(n)\}$.

$$\text{eval(n)} = F[f_1(n), f_2(n), \ldots, f_K(n)]$$

$$ej., \text{eval(n)} = \sum_{i=1}^{K} w_i f_i(n)$$

F utiliza conocimiento del dominio, que puede ser

- Proporcionado por expertos (ej. en ajedrez Deep Blue de IBM, 1997)
- **Aprendido** de la experiencia. Por ejemplo puede ser una red neuronal entrenada mediante aprendizaje por refuerzo a partir de partidas que el ordenador juega contra sí mismo (ej. en ajedrez, alphaZero de DeepMind, 2018)

Ajedrez

Tiene aproximadamente 10^{40} nodos, b = 35.

- Asumamos que cada sucesor puede generarse en 1 μs (10^{-6} s)
  - Una exploración completa tomaría $3 \cdot 10^{26}$ años (la edad del universo es ~ 10^{10} años).
  - Si asumimos una limitación temporal de 1 minuto por movimiento, sólo podremos realizar una exploración completa hasta profundidad 5.
  - Con poda alfa-beta + ordenamiento perfecto, podemos realizar una exploración completa hasta profundidad 10.

  - Función de evaluación simple (función lineal pesada) basada en el **valor material**:

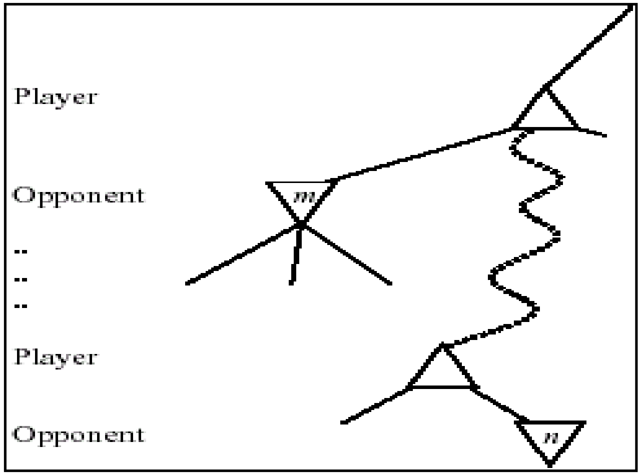
    inicialmente MAX = blancas;

    peón = 1; caballo y alfil = 3; torre = 5; reina = 9.

    f = (n-peones-blancos)*1 + (n-alfiles-blancos)*3 +...

    - (n-peones-negros)*1 - (n-alfiles-negros)*3 -...
  - Funciones de evaluación más complejas toman en cuenta características cualitativas como el "control del centro", "buena posición del rey", "buena estructura de peones".
  - Uso de librerías de movimientos (aperturas, movimientos finales)

Poda alfa-beta



- Observación: Para calcular el valor minimax de un nodo muchas veces no es necesario explorar exhaustivamente.
  El algoritmo realiza una búsqueda primero-en-profundidad desde un nodo dado.
  Si durante dicha búsqueda se encuentran los nodos ***m*** y ***n*** en diferentes subárboles, y el nodo ***m*** es mejor que el ***n***, entonces, asumiendo decisiones óptimas, nunca se llegará al nodo ***n*** en el juego actual.
- Tener en cuenta **en cada nodo un intervalo** $[\alpha, \beta]$ que contiene el valor minimax del nodo, y **actualizar** los límites del intervalo **según va avanzando la búsqueda**
  - α es el valor de la **mejor** alternativa **para MAX** encontrada hasta el momento (es decir, la de **mayor** valor)
    - $\Rightarrow$ <u>los valores de α nunca descienden en un nodo MAX.</u>
  - β es el valor de la **mejor** alternativa **para MIN** encontrada hasta el momento (es decir, la de **menor** valor)
    - $\Rightarrow$ <u>los valores de β nunca aumentan en un nodo MIN.</u>
- **<u>Reglas para parar la búsqueda</u>**:
  - **α-cutoff**: Parar la búsqueda en un nodo MIN tal que $\beta \leq \alpha$ para cualquiera de sus antecesores MAX.
  - **β -cutoff**: Parar la búsqueda en un nodo MAX tal que $\alpha \geq \beta$ de cualquiera de sus antecesores MIN.

Minimax + poda alfa-beta + búsqueda en profundidad

- El **valor α** representa una **cota inferior para el valor MINIMAX en un nodo MAX** (lo peor que le podría ir a MAX). Se **inicializa** a ***min-utility***. **Nunca decrece.**
- El **valor β** representa una **cota superior para el valor MINIMAX en un nodo MIN** (lo mejor que le podría ir a MIN). Se **inicializa** a ***max-utility***. **Nunca crece.**

1. **Inicialización en el nodo raíz: [α, β]← [*min-utility*, *max-utility*]**

   En caso de que no se conozcan los valores mínimo (min-utility) y máximo (max-utility) que puede alcanzar la función de utilidad para el juego considerado, se utilizará la inicialización [α, β] ← [$-\infty$, $+\infty$]

2. Desde la raíz, los valores de [α, β] son propagados de nodo padre a nodo hijo mediante un recorrido en el árbol primero en profundidad hasta que el nodo hijo
   - sea terminal: α o $\beta =$ *utility*(n)
   - se encuentre en el límite de profundidad preestablecido: α o $\beta =$ *eval*(n)

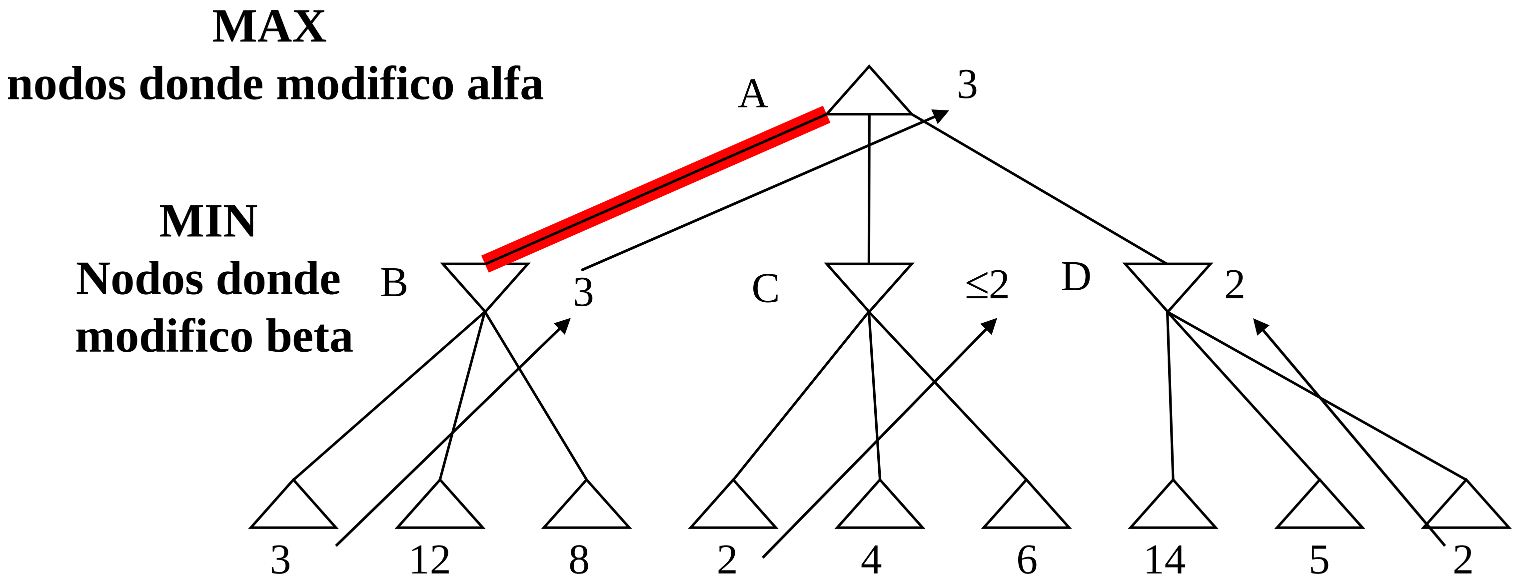
3. En el proceso de backtracking (propagación de información desde las hojas hacia la raíz en el árbol de búsqueda, se actualizan los intervalos de los nodos internos del siguiente modo:
   - **En un nodo MAX se actualiza el valor α:**

     α ← **máximo (α , β**’s **de sucesores de MAX generados hasta ese momento)**.
   - **En un nodo MIN se actualiza el valor β:**

     β ← **mínimo (β , α**’s **de sucesores de MIN generados hasta ese momento)**

4. **Poda**: Si, como resultado de la **actualización de los valores de α, β,** se cumple $\alpha \geq \beta$, se **realiza poda en ese nodo** y **no se prosigue la búsqueda** para el resto de sucesores aún no explorados de dicho nodo.

Ejemplo: poda alfa-beta



Poda alfa-beta

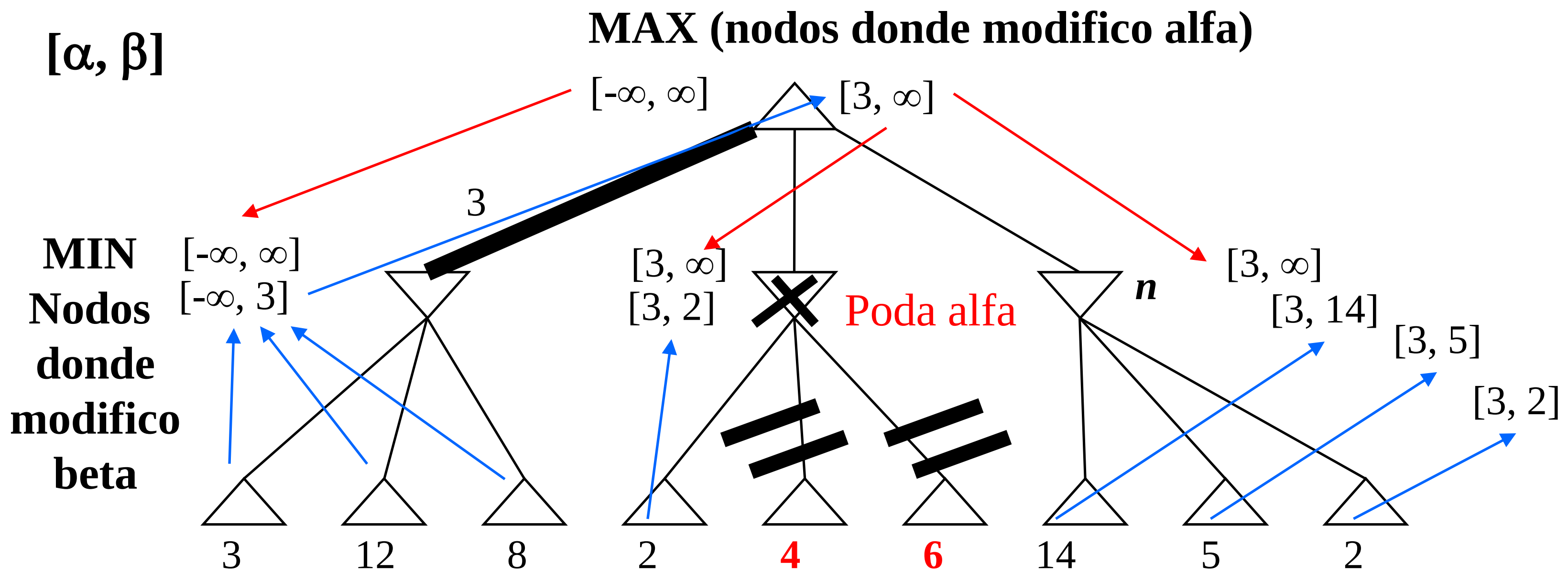
```
function ALPHA-BETA-DECISION(state) returns an action
   return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

function MAX-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           α, the value of the best alternative for  MAX along the path to state
           β, the value of the best alternative for  MIN along the path to state
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s, α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v

function MIN-VALUE(state, α, β) returns a utility value
   same as MAX-VALUE but with roles of α, β reversed
```

- **La poda no afecta al resultado final.**
- **La eficacia de la poda depende del orden** en la búsqueda: Es buena si los movimientos buenos se exploran primero.
  - **Caso peor: No hay mejora**
  - **Ordenación aleatoria**: ***$O(b^{3d/4})$*** $\Rightarrow$ $b^* = b^{3/4}$ (Pearl, 1984)
  - **Ordenación perfecta: $O(b^{d/2})$** $\Rightarrow$ $b^* = b^{1/2}$.

    Donald E. Knuth; Ronald W. Moore; *An analysis of alpha-beta pruning*. Artificial Intelligence 6(4); 293-326 (1975)

    El uso de heurísticas sencillas lleva a menudo a b* cercano al óptimo (ej. examinar primero los movimientos que fueron los mejor considerados en el anterior turno)

Poda alfa-beta: Ejemplo, I



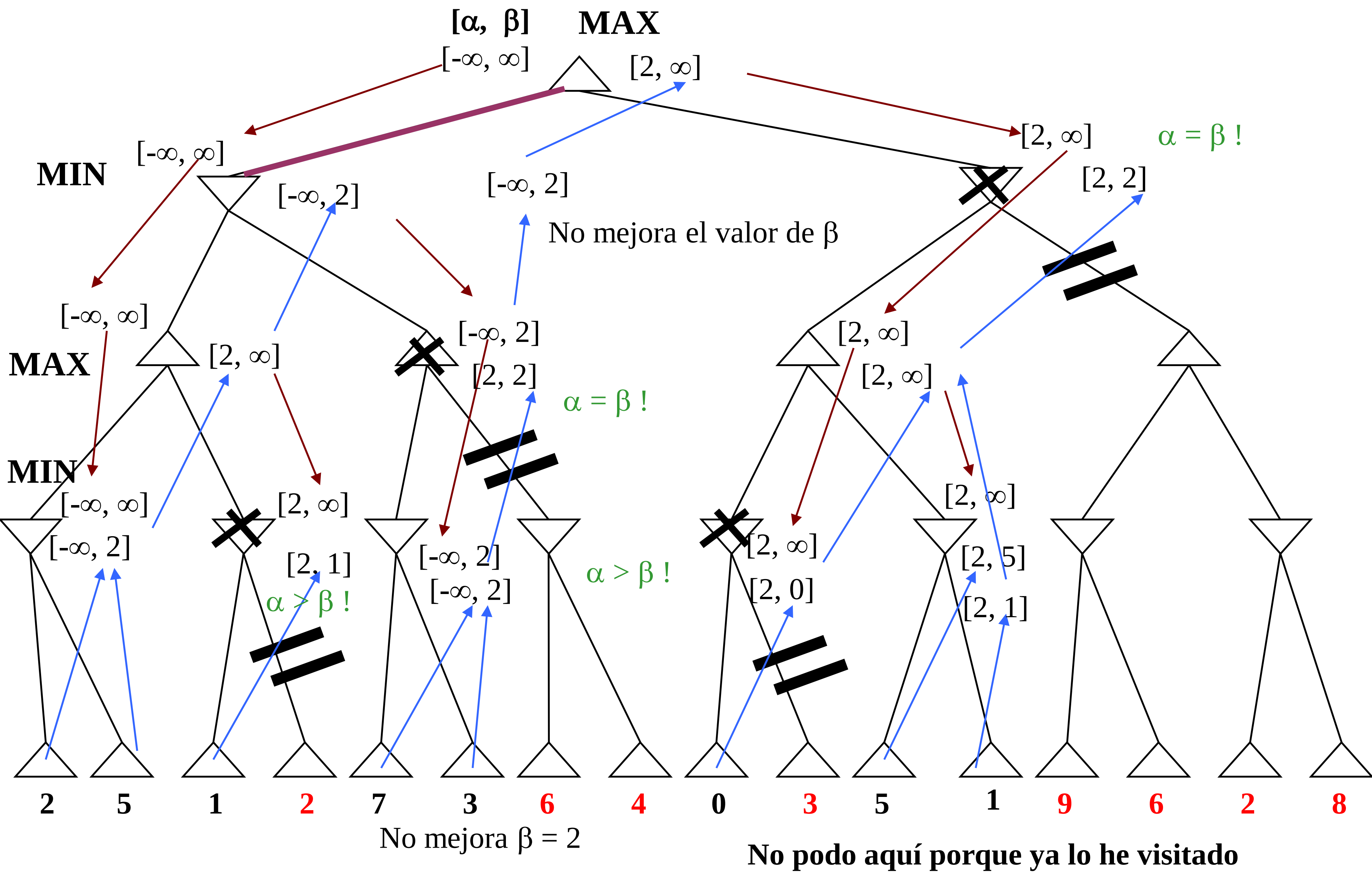
El orden en la búsqueda es importante para la eficacia de la poda

Algoritmo de poda

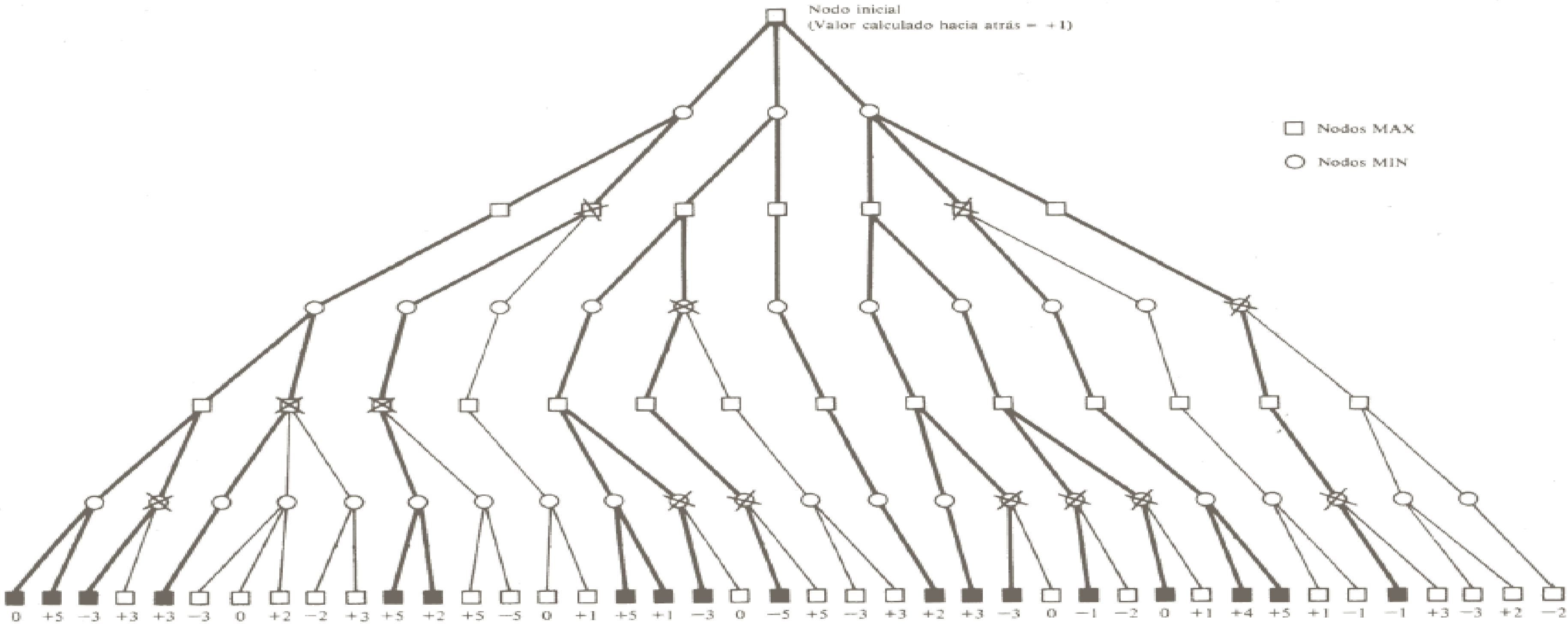
En nodo MAX: $\alpha \leftarrow \max(\alpha, \beta\text{'s de sucesores})$

En nodo MIN: $\beta \leftarrow \min(\beta, \alpha\text{'s de sucesores})$

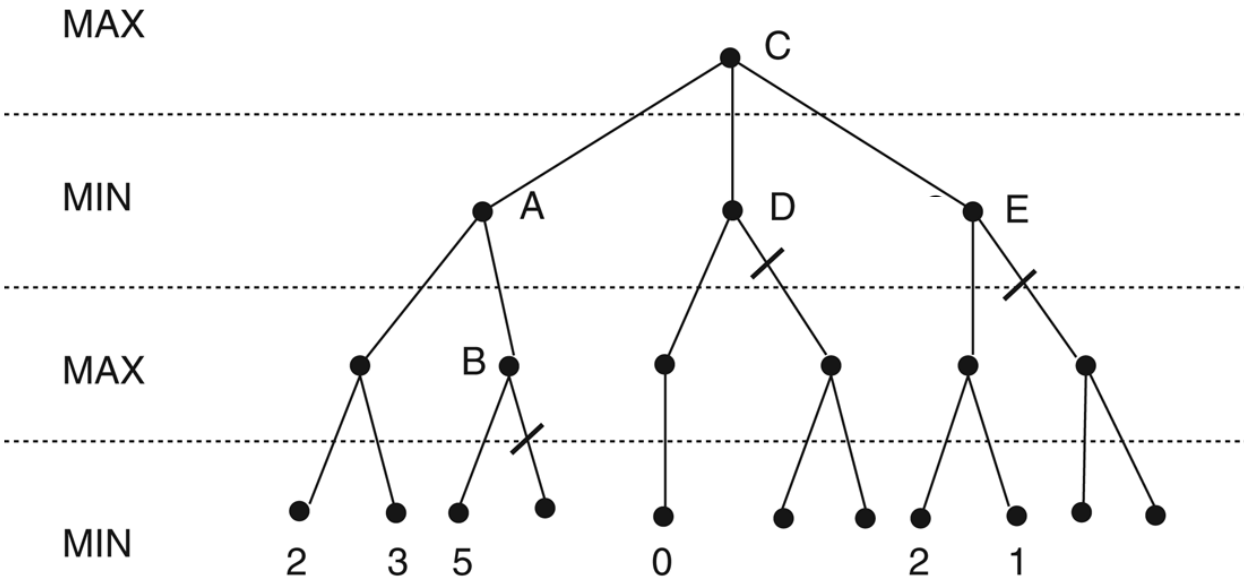
Poda alfa-beta: Ejemplo, II



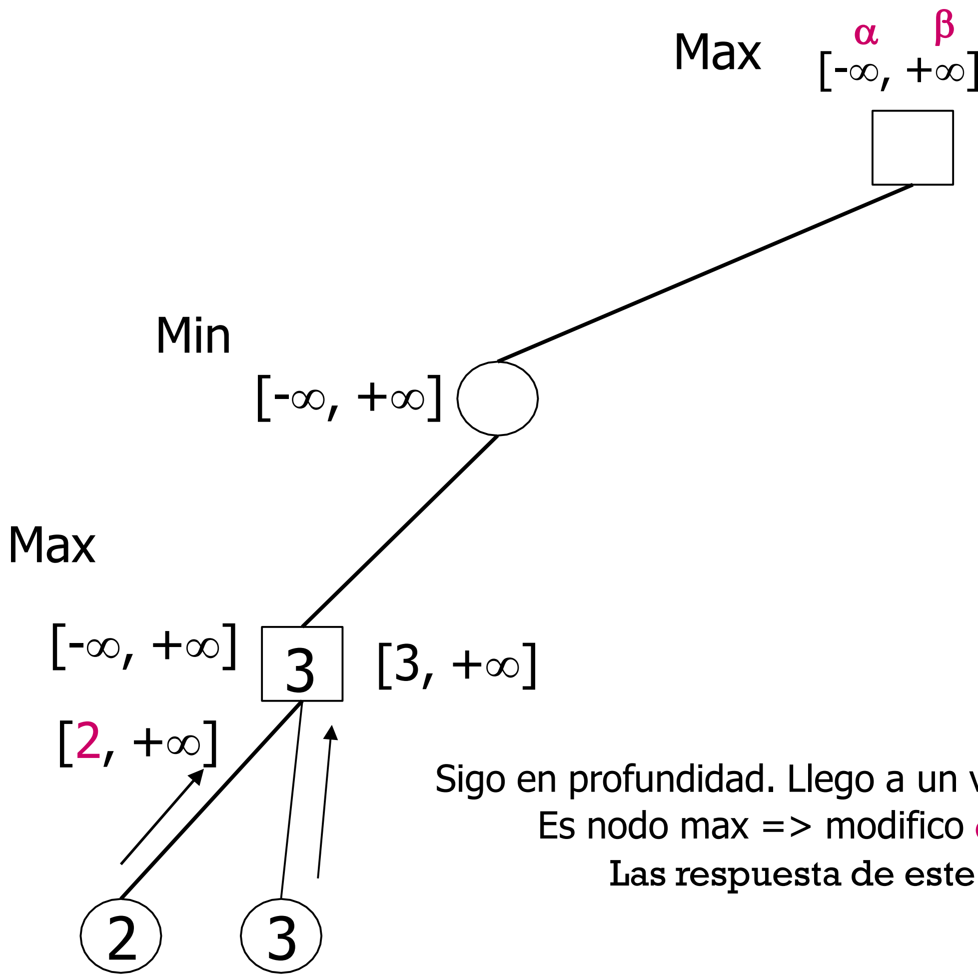
Poda alfa-beta: Ejemplo, III



Ejemplo IV



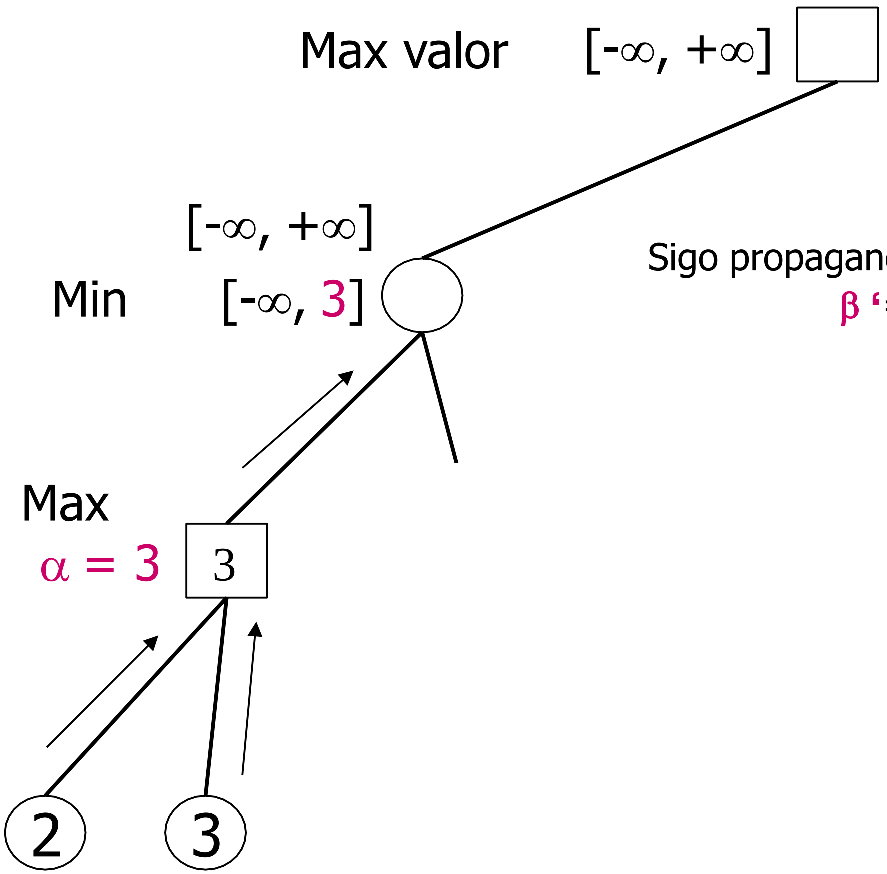
Ejemplo paso a paso



Sigo en profundidad. Llego a un valor. Propago hacia arriba.
Es nodo max => modifico α. α ' = max (2,3) =3.
Las respuesta de este nodo MAX es 3

Llego a un valor. Propago hacia arriba.
Es nodo max => modifico α. α ' = max (-∞,2) = 2.

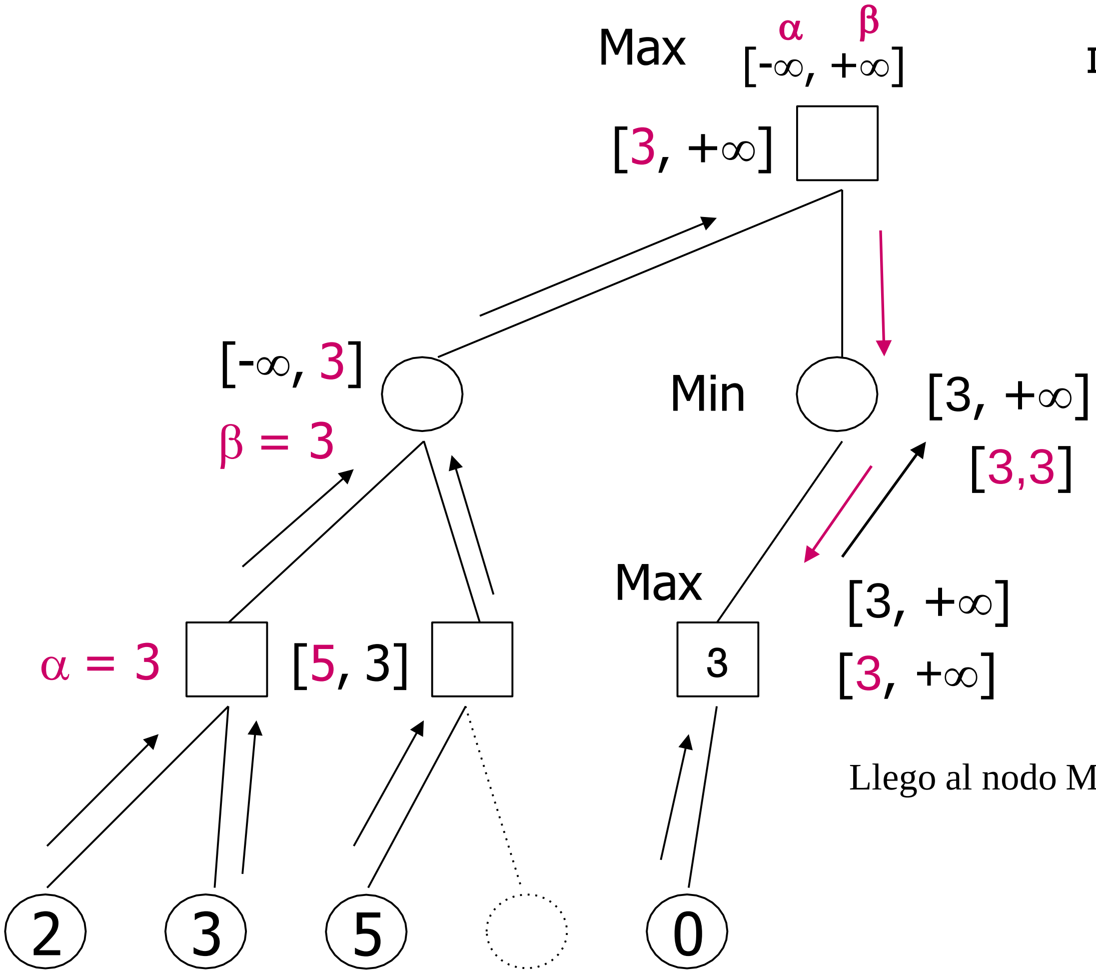
Ejemplo paso a paso



Sigo propagando hacia arriba. Llego a nodo min => modifico β

β '= min (β ,valor sucesor) = min (+∞, 3)

Ejemplo paso a paso

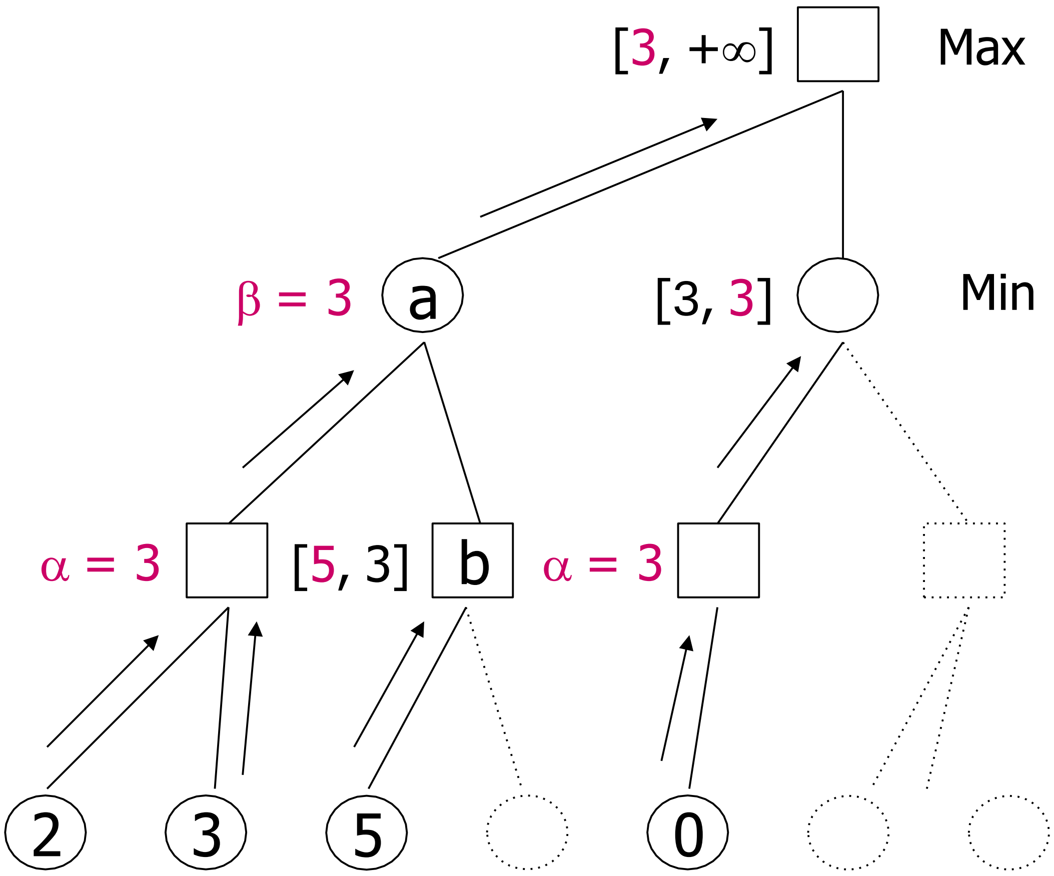


Llego al nodo MAX. Modifico α. α ' = max (-∞,3) =3
Propago hacia abajo

Llego a nodo min => modifico β
β '= min (β ,valor sucesor) = min (+∞, 3) = 3

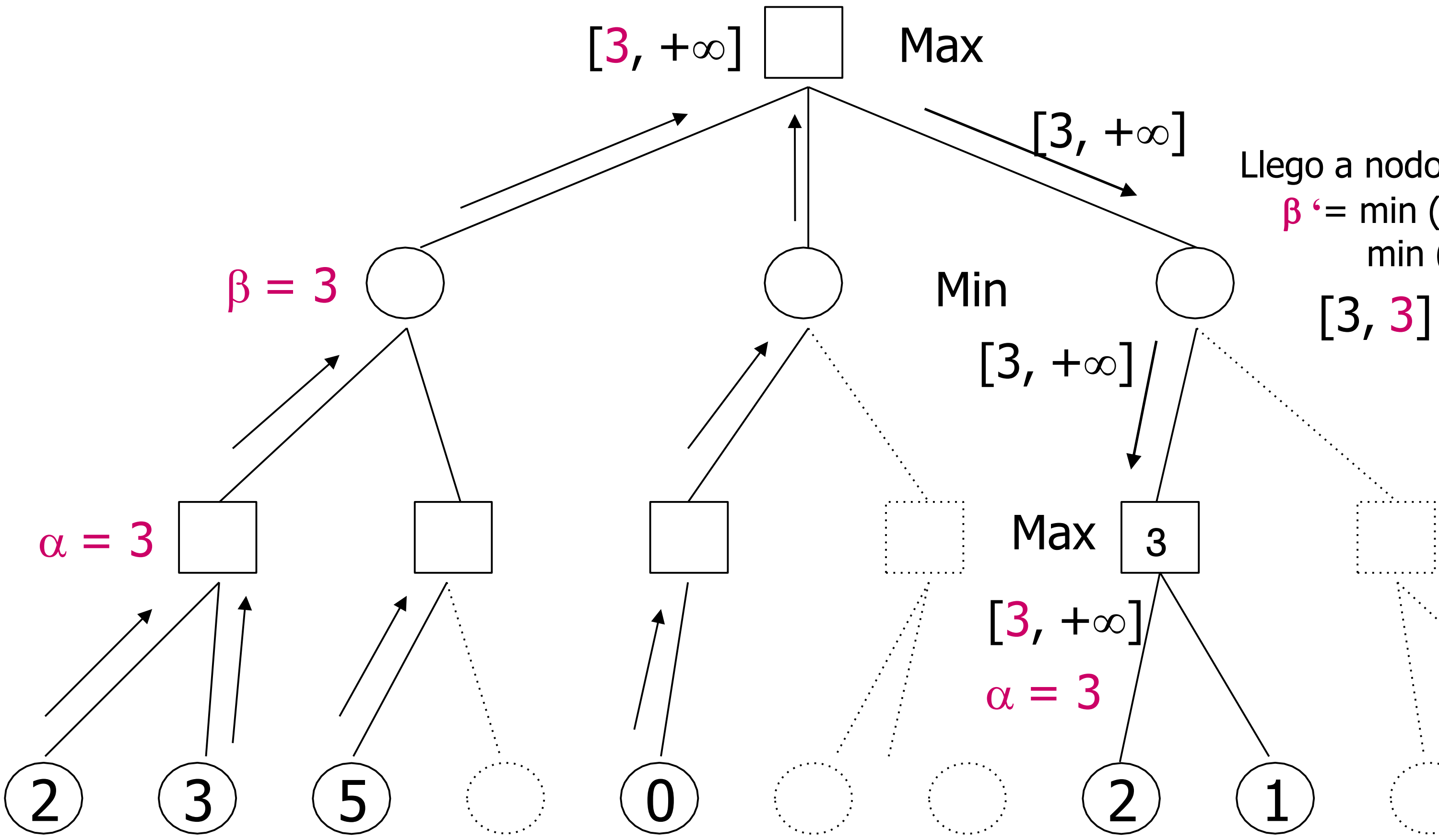
Llego al nodo MAX. Modifico α. α ' = max (3,0) =3

Ejemplo paso a paso



poda alfa. Estoy en nodo min y $\beta <= \alpha$

Ejemplo paso a paso



Llego a nodo min => modifico β
β ‘= min (β ,valor sucesor) = min (+∞, 3) = 3

Propago hacia abajo
Llego al nodo MAX. Modifico α. α ‘ = max (3,2) =3. Sigo max (3,1)= 3

poda alfa
Estoy en nodo min y β <= α

Propiedades de la poda alfa-beta

- Este algoritmo es de tipo ramificación y poda:

  Garantiza encontrar el mismo mejor movimiento (u otro equivalente con el mismo valor) que minimax, pero de forma más eficiente

- Si el mejor nodo límite es el generado primero, las podas serán máximas y habrá que generar y evaluar el mínimo número de nodos
  - Influye mucho el orden de generación de los sucesores
  - Suponiendo una búsqueda en profundidad con un límite de profundidad l, Utilizando ordenación óptima alfa-beta tiene que examinar sólo $O(b^{l/2})$ nodos para escoger el mejor movimiento, en vez de $O(b^{l})$ con minimax
    - El factor de ramificación efectivo sería $b^{1/2}$ en lugar de b
    - Con los mismos recursos computacionales, permite realizar búsquedas con el mismo coste temporal con un límite de profundidad que es el doble de la correspondiente búsqueda sin poda (= tiempo)

Juegos de azar

- Introducción de un **proceso aleatorio** como un **agente** adicional
  - Turno de MAX = movimiento de RAND + movimiento de MAX
  - Turno de MIN = movimiento de RAND + movimiento de MIN

  Movimiento de RAND = tirar una moneda, un dado, etc.

- El árbol del juego tiene nodos MAX + nodos MIN + nodos RAND

- Valor Expectiminimax:

$$\text{expectiminimax}(n)\begin{cases} \text{utilidad}(n) & \text{si n es un nodo terminal} \\ \max\{\text{expectiminimax}(s);\ s \in \text{ sucesores}(n)\} & \text{si n es un nodo MAX.} \\ \min\{\text{expectiminimax}(s);\ s \in \text{ sucesores}(n)\} & \text{si n es un nodo MIN.} \\ \sum_{s \in \text{ sucesores}(n)} p(s)\ \text{expectiminimax}(s) & \text{si n es un nodo RAND.} \end{cases}$$

  - IMPORTANTE: Para que funcione el expectiminimax, la función de evaluación debería ser una **transformación lineal positiva** de la utilidad esperada del estado.

  - La complejidad temporal es exponencial: $O(b^l \cdot n^l)$.
    b = factor de ramificación de nodos MAX / MIN.
    n = factor de ramificación de los nodos de RAND.
    l = límite en la búsqueda en profundidad.

  - Los nodos de RAND también pueden ser podados.