

Inteligencia Artificial

Búsqueda entre adversarios

Primavera 2007

profesor: Luigi Ceccaroni



Juegos

- En los **entornos multiagente (cooperativos o competitivos)**, cualquier agente tiene que considerar las acciones de otros agentes.
- La imprevisibilidad de estos otros agentes puede introducir muchas **contingencias** en el proceso de resolución de problemas.
- Los entornos competitivos, en los cuales los objetivos de los agentes están en conflicto, dan ocasión a problemas de **búsqueda entre adversarios**, a menudo conocidos como **juegos**.

Juegos

- La **teoría matemática de juegos**, una rama de la economía, ve a cualquier entorno multiagente como un juego.
- En IA, los “juegos” son una clase más especializada, que los teóricos llaman **juegos**:
 - **de suma cero**
 - de dos jugadores (jugador MAX, jugador MIN)
 - por turnos
 - deterministas
 - **de información perfecta** (ajedrez, damas, tres en raya...) vs. **información imperfecta** (poker, stratego, bridge...)

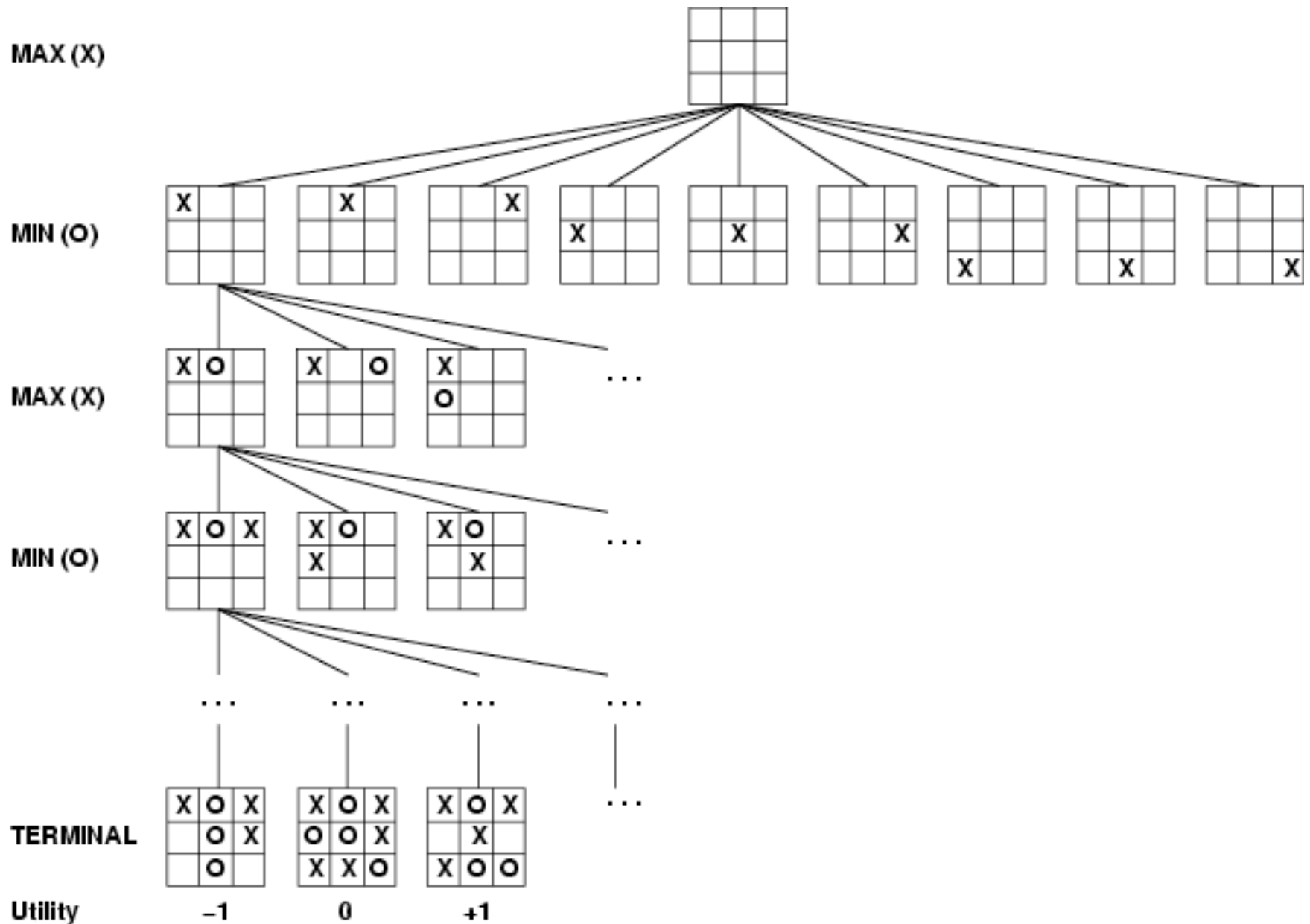
Juegos

- Los juegos son interesantes *porque* son demasiado difíciles de resolver.
- El ajedrez, por ejemplo, tiene un factor de ramificación promedio de 35 y los juegos van a menudo a 50 movimientos por cada jugador:
 - grafo de búsqueda: aproximadamente 10^{40} nodos distintos
 - árbol de búsqueda: 35^{100} o 10^{154}
- Los juegos, como el mundo real, requieren la capacidad de tomar *alguna* decisión (la jugada) cuando es infactible calcular la decisión *óptima*.

Decisiones óptimas en juegos

- Un juego puede definirse formalmente como una clase de problemas de búsqueda con los componentes siguientes:
 - El **estado inicial**
 - Una **función sucesor**, que devuelve una lista de pares (*movimiento, estado*)
 - Un **test terminal**, que determina cuándo termina el juego (por estructura o propiedades o función utilidad)
 - Una **función utilidad**

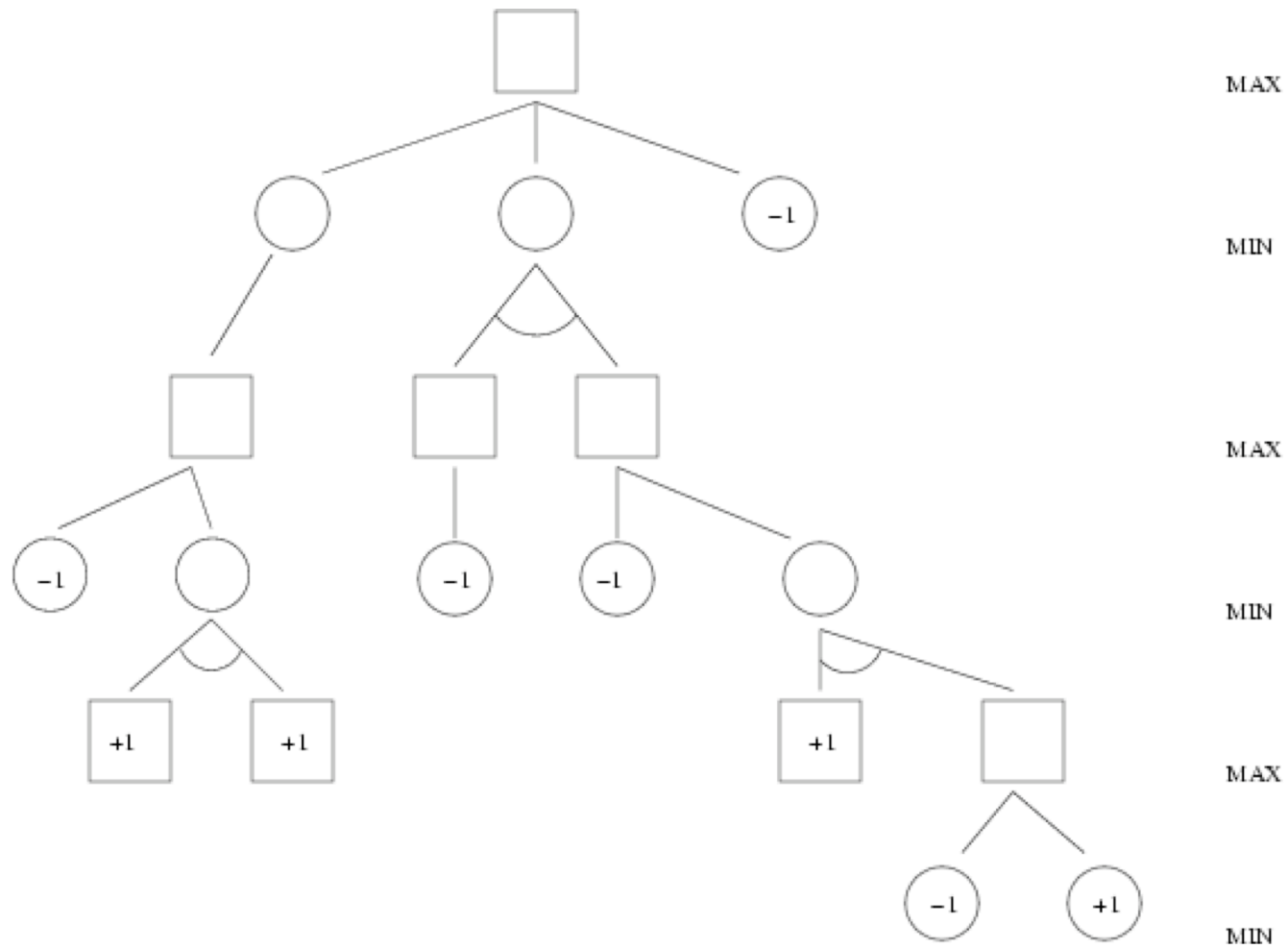
Búsqueda entre adversarios



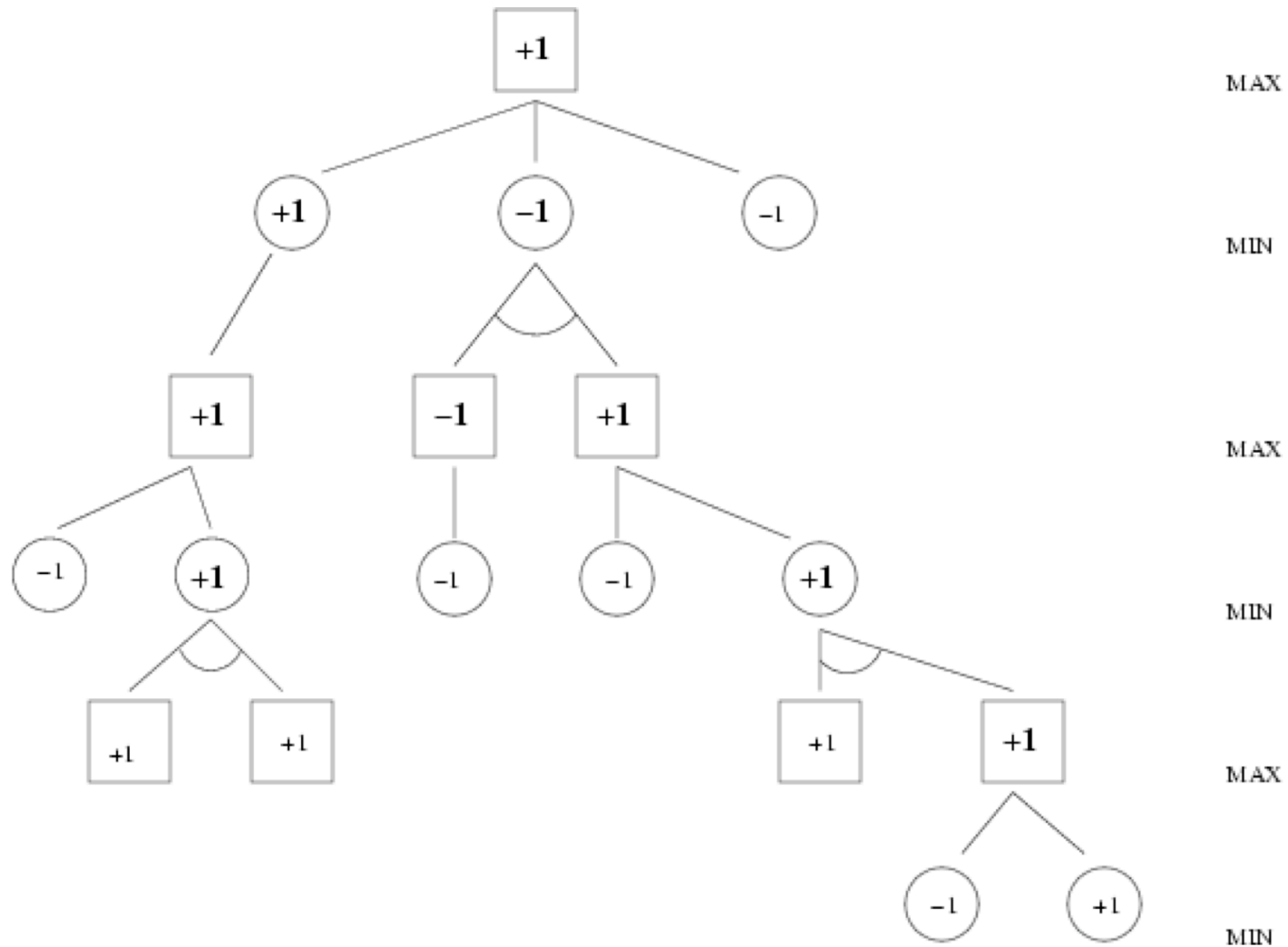
Búsqueda entre adversarios

- **Aproximación trivial:** generar todo el árbol de jugadas.
- Se etiquetan las jugadas terminales, dependiendo de si gana MAX o MIN, con un valor de *utilidad* de, por ejemplo, “+1” o “-1”.
- El objetivo es encontrar un conjunto de movimientos accesible que dé como ganador a MAX.
- Se propagan los valores de las jugadas terminales de las hojas hasta la raíz.
- Incluso un juego simple como *tic-tac-toe* es demasiado complejo para dibujar el árbol de juegos entero.

Búsqueda entre adversarios



Búsqueda entre adversarios

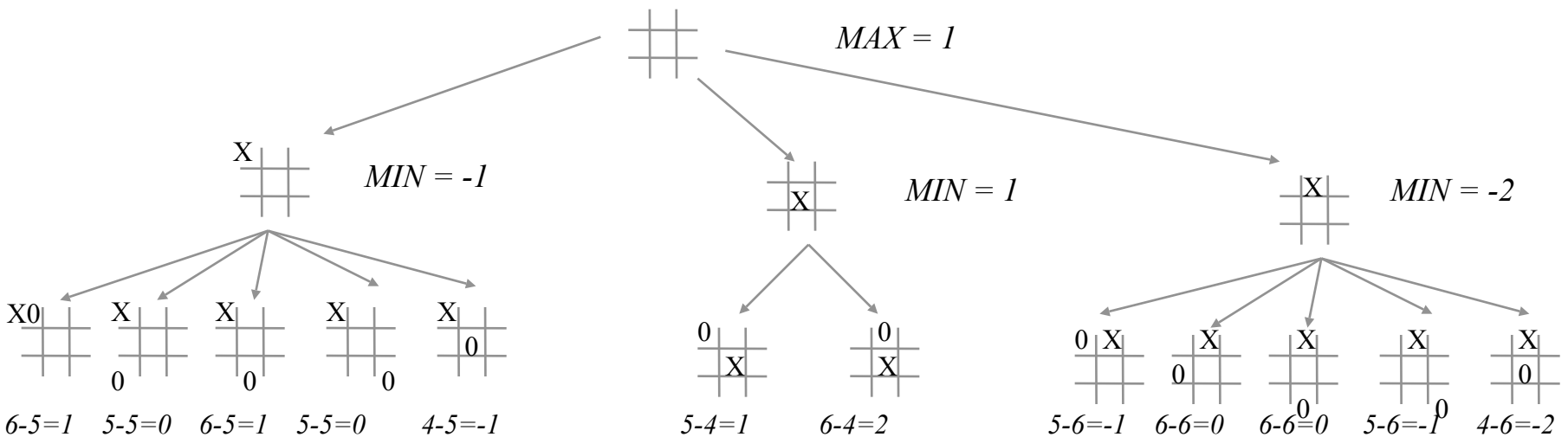


Búsqueda entre adversarios

- **Aproximación heurística:** definir una función que nos indique lo cerca que estamos de una jugada ganadora (o perdedora).
- En esta función interviene información del dominio.
- Esta función no representa ningún coste, ni es una distancia en pasos.
- Por convención:
 - las jugadas ganadoras se evalúan a “ $+\infty$ ”
 - las jugadas perdedoras se evalúan a “ $-\infty$ ”
- El algoritmo busca con profundidad limitada.
- Cada nueva decisión por parte del adversario implicará repetir parte de la búsqueda.

Ejemplo: *tic-tac-toe*

- e (función utilidad) = número de filas, columnas y diagonales completas disponibles para MAX - número de filas, columnas y diagonales completas disponibles para MIN
- MAX juega con X y desea maximizar e
- MIN juega con 0 y desea minimizar e
- Valores absolutos altos de e : buena posición para el que tiene que mover
- Controlar las simetrías
- Utilizar una profundidad de parada (en el ejemplo: 2)

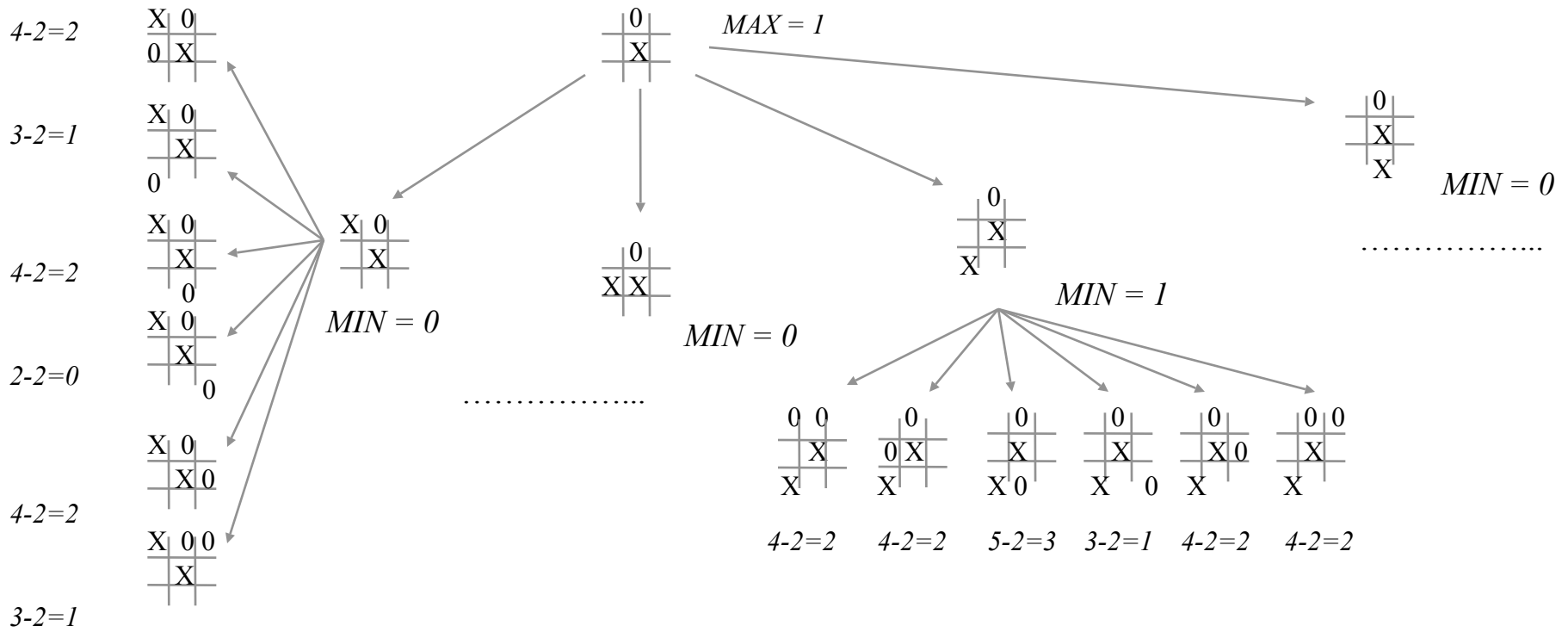


La mejor jugada de MAX es pues

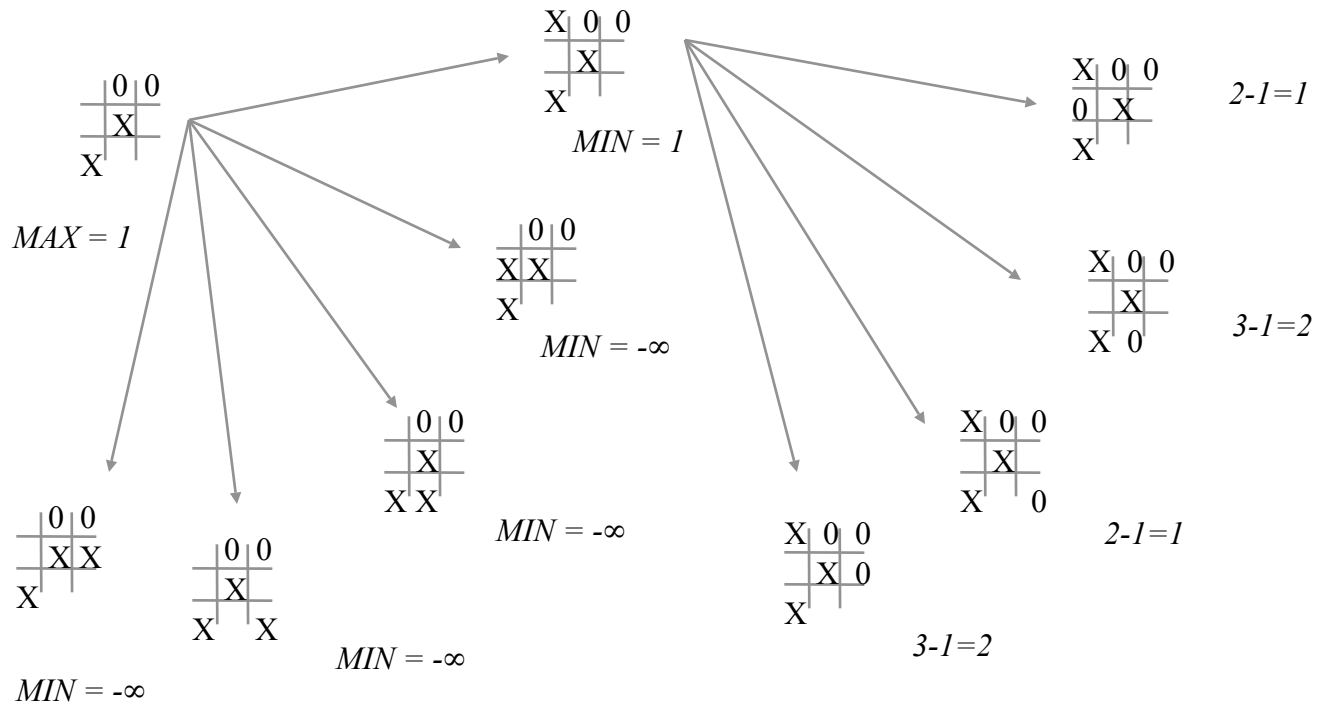
	X	

 tras lo cual MIN podría jugar

		X

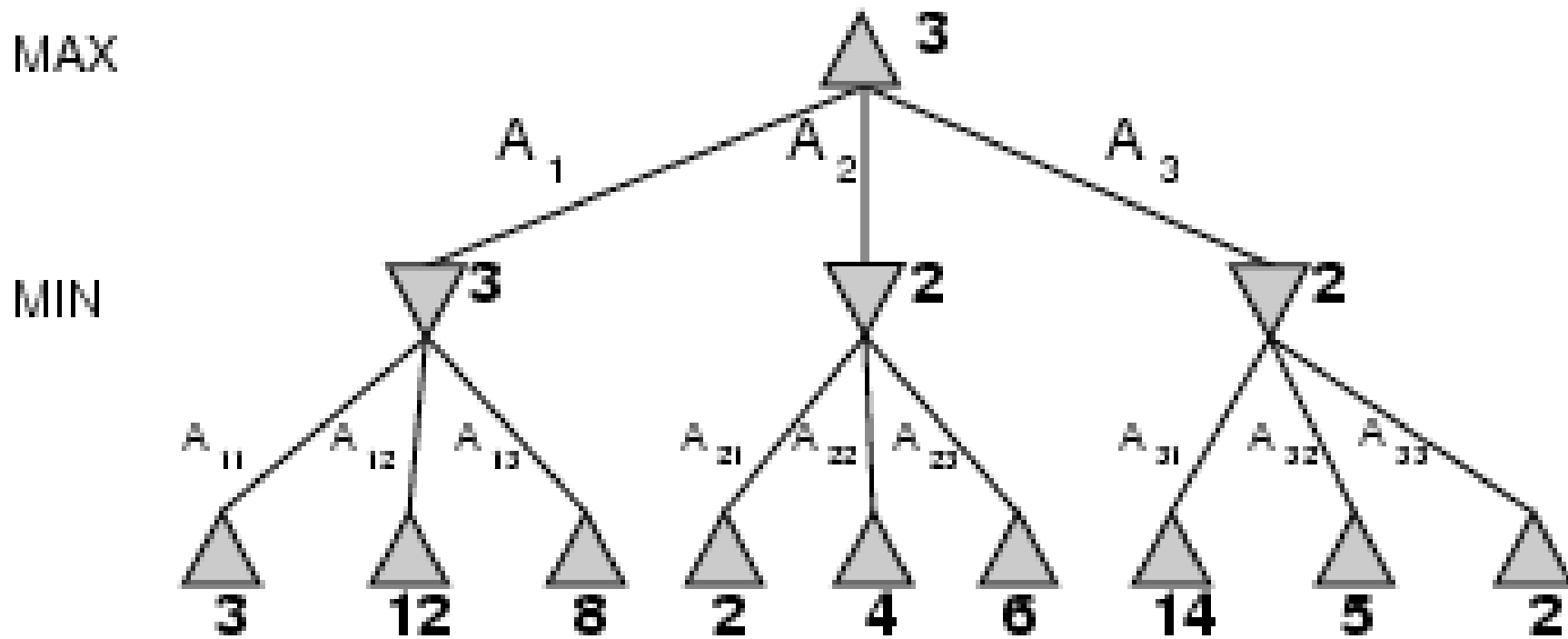


La mejor jugada de MAX es pues $\begin{array}{|c|c|c|} \hline & 0 & \\ \hline & X & \\ \hline X & & \\ \hline \end{array}$ tras lo cual MIN podría jugar $\begin{array}{|c|c|c|} \hline & 0 & 0 \\ \hline & X & \\ \hline X & & \\ \hline \end{array}$



La mejor jugada de MAX es pues $\begin{array}{|c|c|c|} \hline X & 0 & 0 \\ \hline X & & \\ \hline X & & \\ \hline \end{array}$

Búsqueda entre adversarios: **minimax**

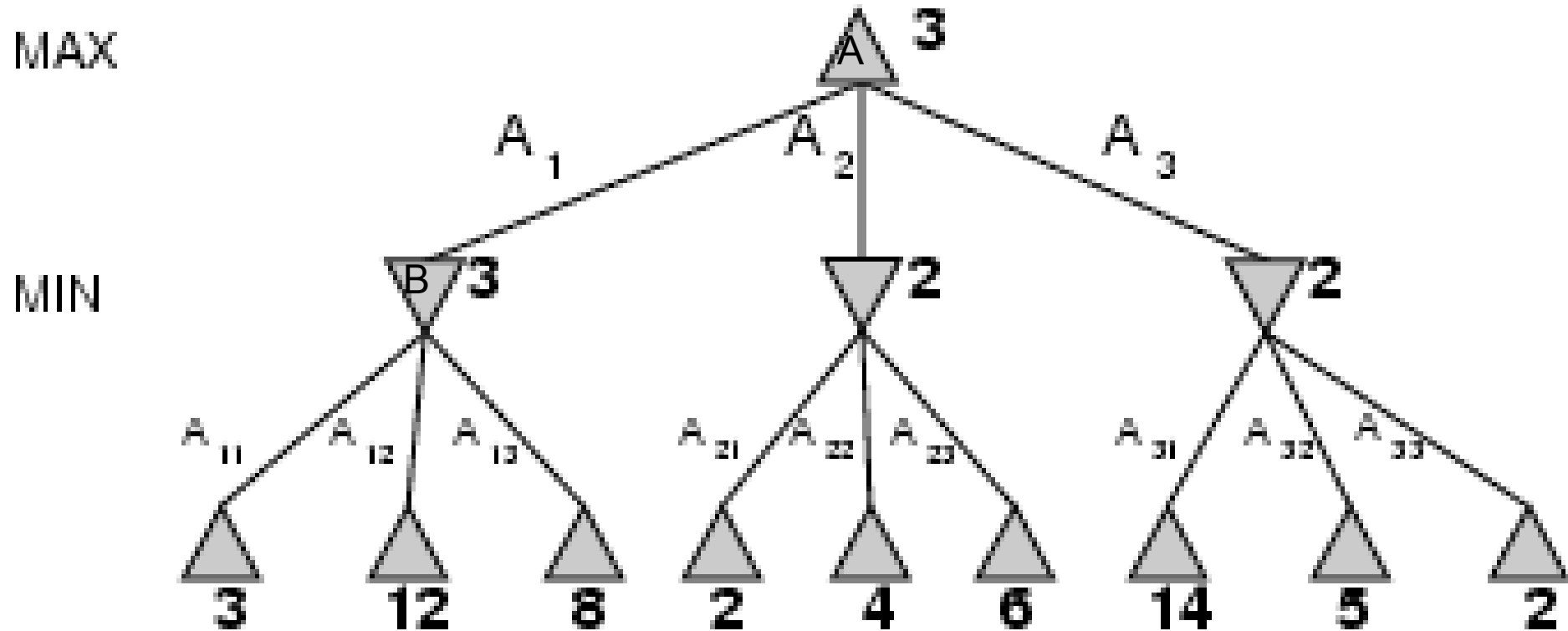


- Valor-Minimax(n): utilidad para MAX de estar en el estado n asumiendo que ambos jugadores jueguen óptimamente:
 - Utilidad(n), si n es un estado terminal
 - $\max_{s \in \text{Sucesores}(n)} \text{Valor-Minimax}(s)$, si n es un estado MAX
 - $\min_{s \in \text{Sucesores}(n)} \text{Valor-Minimax}(s)$, si n es un estado MIN

Algoritmo minimax (1)

- Calcula la decisión minimax del estado actual.
- Usa un cálculo simple recurrente de los valores minimax de cada estado sucesor.
- La recursión avanza hacia las hojas del árbol.
- Los valores minimax **retroceden** por el árbol cuando la recursión se va deshaciendo.

Algoritmo minimax (2)



- El algoritmo primero va hacia abajo a los tres nodos izquierdos y utiliza la función Utilidad para descubrir que sus valores son 3, 12 y 8.
- Entonces toma el mínimo de estos valores, 3, y lo devuelve como el valor del nodo *B*.

Algoritmo minimax (3)

- Realiza una exploración primero en profundidad completa del árbol de juegos.
- Si la profundidad máxima del árbol es m , y hay b movimientos legales en cada punto, entonces la complejidad :
 - en tiempo es $O(b^m)$;
 - en espacio es
 - $O(bm)$ si se generan todos los sucesores a la vez;
 - $O(m)$ si se generan los sucesores uno por uno.
- Juegos reales: los costos de tiempo son inaceptables, pero este algoritmo sirve como base para el primer análisis matemático y para algoritmos más prácticos.

Algoritmo minimax (4)

función Decisión-Minimax(*estado*) **devuelve** una *acción*

variables de entrada: *estado*, estado actual del juego

$v \leftarrow \text{Max-Valor}(\textit{estado})$

devolver la *acción* de Sucesores(*estado*) con valor v

función Max-Valor(*estado*) **devuelve** un *valor utilidad*

si Test-Terminal(*estado*) **entonces devolver** Utilidad (*estado*)

$v \leftarrow -\infty$

para un s en Sucesores(*estado*) **hacer**

$v \leftarrow \text{Max}(v, \text{Min-Valor}(s))$

devolver v

función Min-Valor(*estado*) **devuelve** un *valor utilidad*

si Test-Terminal(*estado*) **entonces devolver** Utilidad (*estado*)

$v \leftarrow \infty$

para un s en Sucesores(*estado*) **hacer**

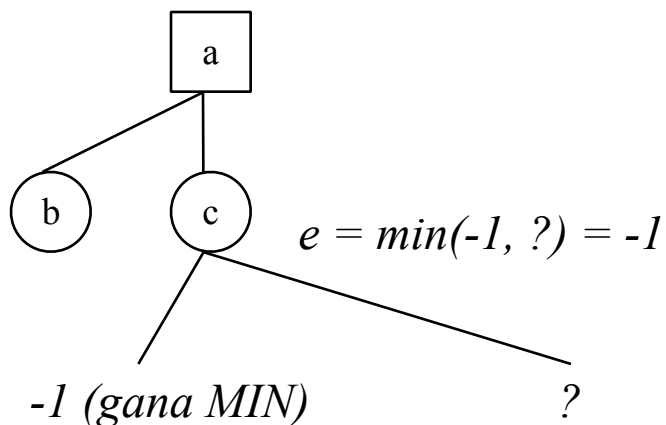
$v \leftarrow \text{Min}(v, \text{Max-Valor}(s))$

devolver v

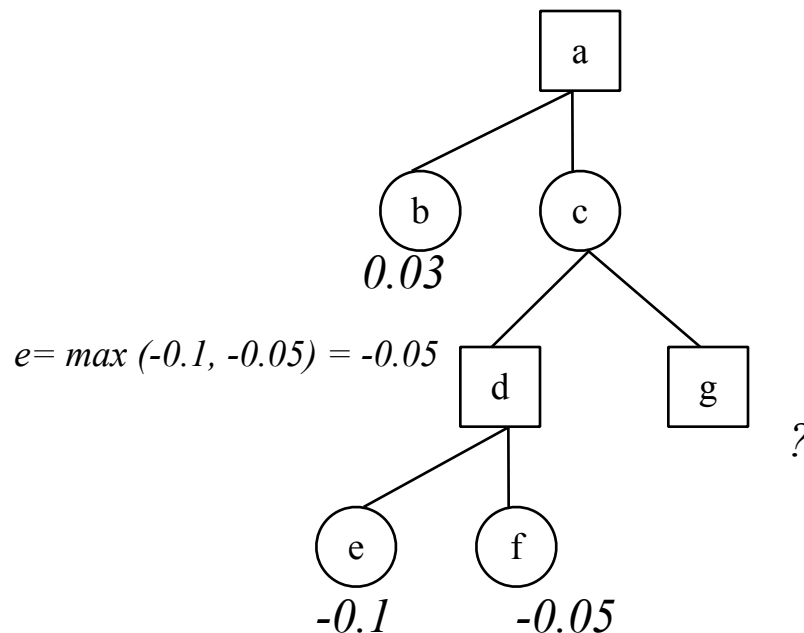
Poda alfa-beta

- Problema de la búsqueda minimax: el número de estados que tiene que examinar es exponencial con el número de movimientos.
- El exponente no se puede eliminar, pero se puede dividir en la mitad.
- Es posible calcular la decisión minimax correcta sin mirar todos los nodos en el árbol.
- La **poda alfa-beta** permite eliminar partes grandes del árbol, sin influir en la decisión final.

Minimax con poda α - β

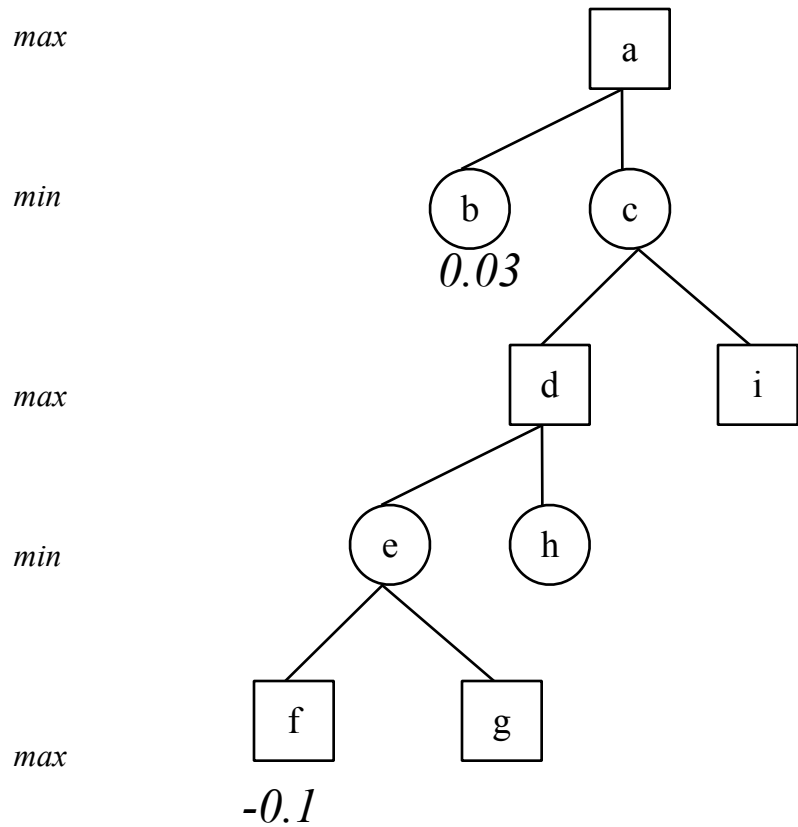


No tiene sentido seguir buscando los otros descendientes de c.



En c: $e = \min(-0.05, v(g))$
por lo tanto en a: $e = \max(0.03, \min(-0.05, v(g))) = 0.03$
Se pueden pues podar los nodos bajo g;
no aportan nada.

- El valor de la raíz y la decisión minimax son independientes de los valores de las hojas podadas.



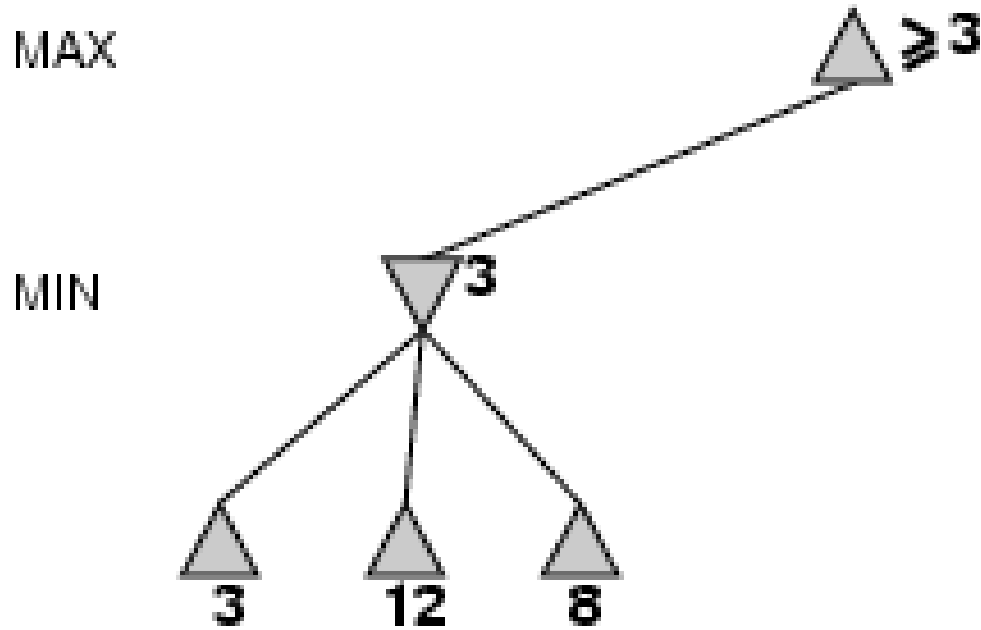
$e(e) = \min(-0.1, v(g))$
 Como la rama b ya da un 0.03, cualquier cosa peor no sirve
 \Rightarrow No hay que explorar g
 $e(d) = \max(e(e), h)$
 \Rightarrow Sí hay que explorar h
 ...

- La búsqueda minimax es primero en profundidad: en cualquier momento sólo se consideran los nodos a lo largo de un camino del árbol.

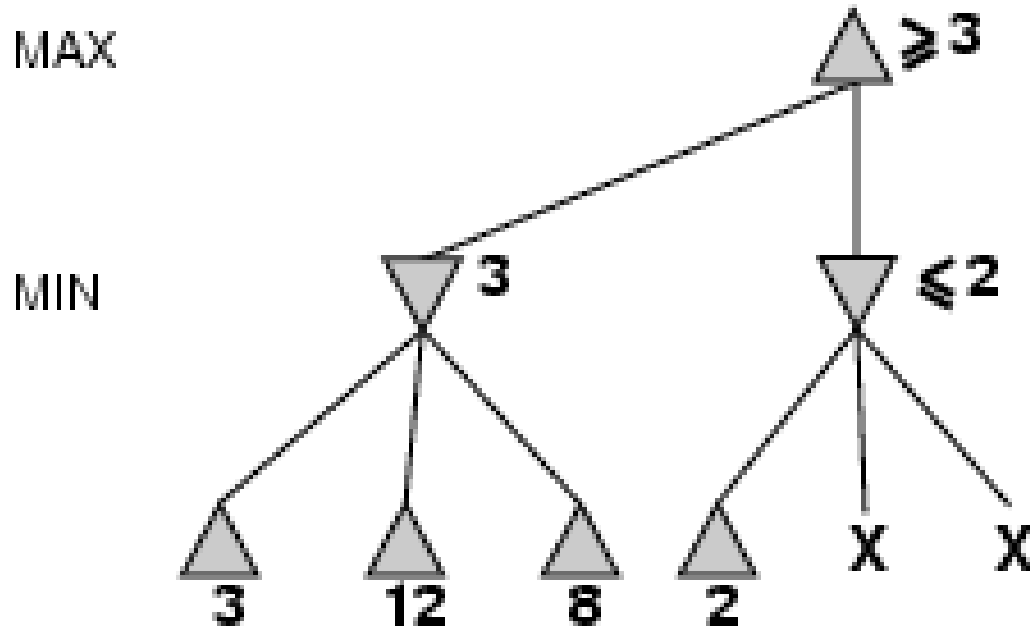
Poda alfa-beta

- Los dos parámetros alfa y beta describen los límites sobre los valores que aparecen a lo largo del camino:
 - α = el valor de la mejor opción (el más alto) que se ha encontrado hasta el momento en cualquier punto del camino, para MAX
 - β = el valor de la mejor opción (el más bajo) que se ha encontrado hasta el momento en cualquier punto del camino, para MIN
- La búsqueda alfa-beta actualiza el valor de α y β según se va recorriendo el árbol y termina la recursión cuando encuentra un nodo peor que el actual valor α o β correspondiente.

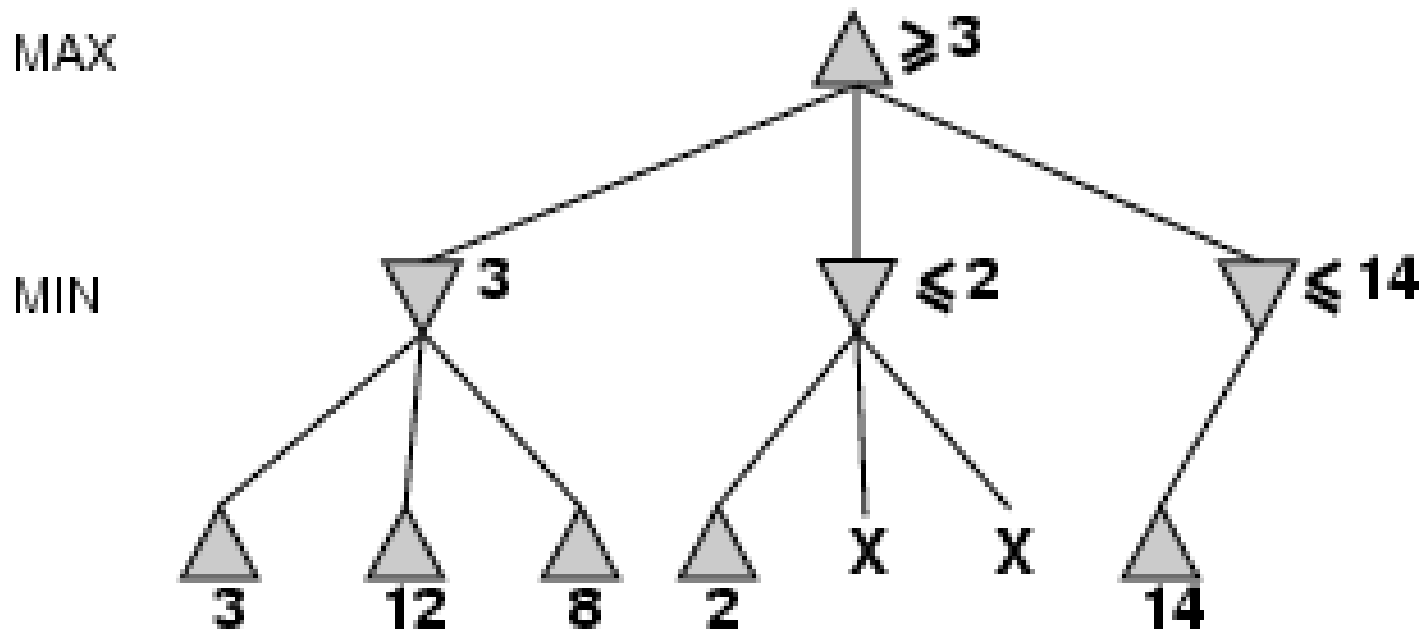
Poda alfa-beta: ejemplo



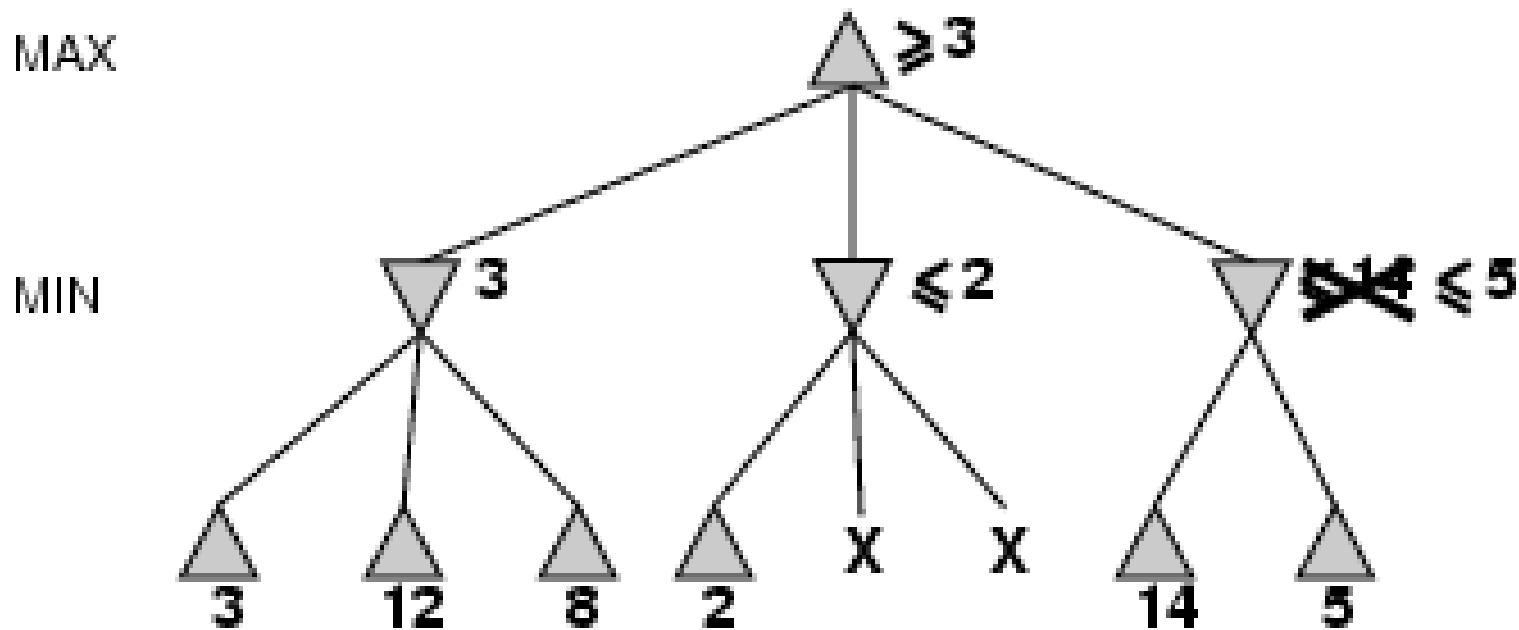
Poda alfa-beta: ejemplo



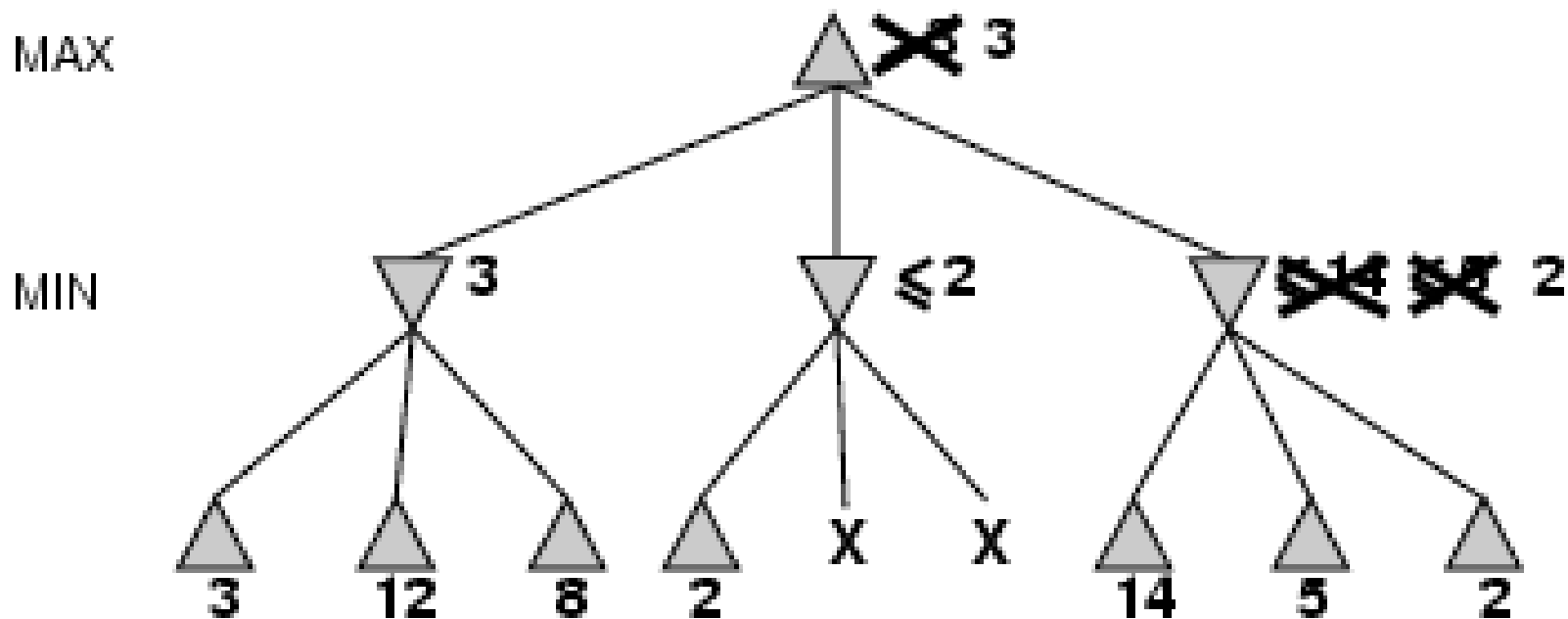
Poda alfa-beta: ejemplo

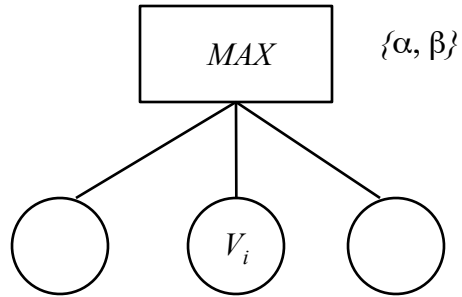


Poda alfa-beta: ejemplo



Poda alfa-beta: ejemplo

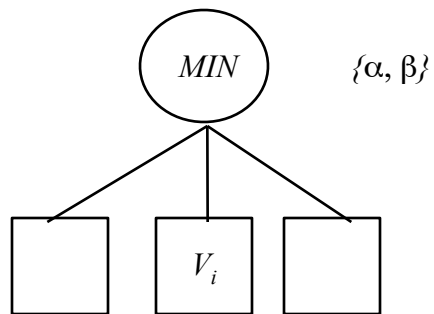




Si $V_i > \alpha$ modificar α

Si $V_i \geq \beta$ poda β

Retornar α



Si $V_i < \beta$ modificar β

Si $V_i \leq \alpha$ poda α

Retornar β

Las cotas α y β se transmiten de padres a hijos de 1 en 1 y en el orden de visita de los nodos.

Minimax con poda α - β

Funcion valorMax (g, α , β) **retorna** entero

Si estado_terminal(g) **entonces**

retorna(evaluacion(g))

si no

Para cada mov **en** movs_posibles(g)

α =max(α ,valorMin(aplicar(mov,g), α , β))

si $\alpha \geq \beta$ **entonces** retorna(β)

fPara

retorna(α)

fsi

fFuncion

Funcion valorMin (g, α , β) **retorna** entero

Si estado_terminal(g) **entonces**

retorna(evaluacion(g))

si no

Para cada mov **en** movs_posibles(g)

β =min(β ,valorMax(aplicar(mov,g), α , β))

si $\alpha \geq \beta$ **entonces** retorna(α)

fPara

retorna(β)

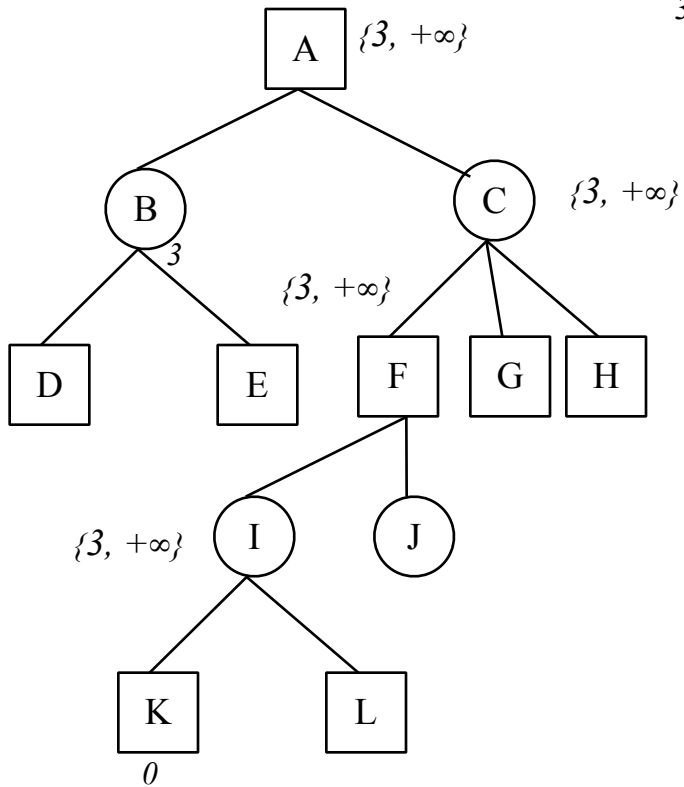
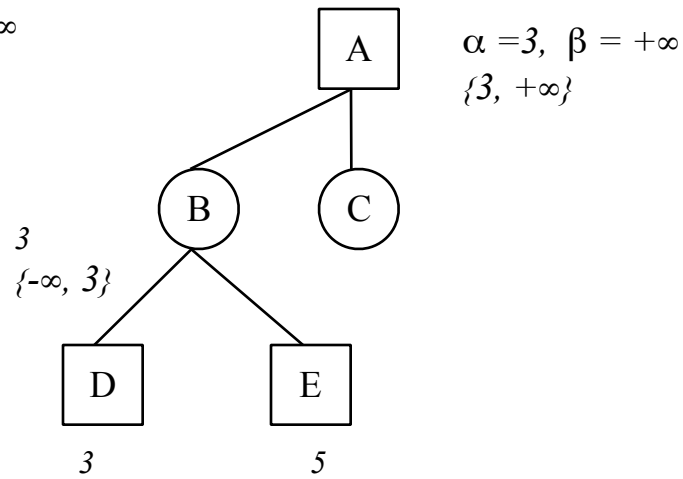
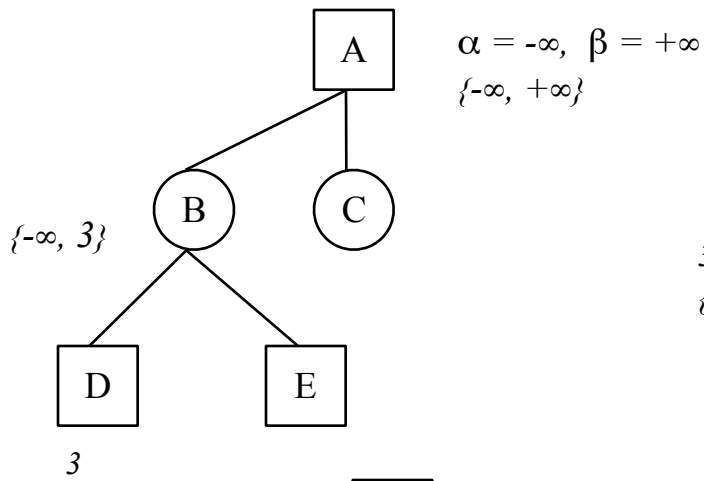
fsi

fFuncion

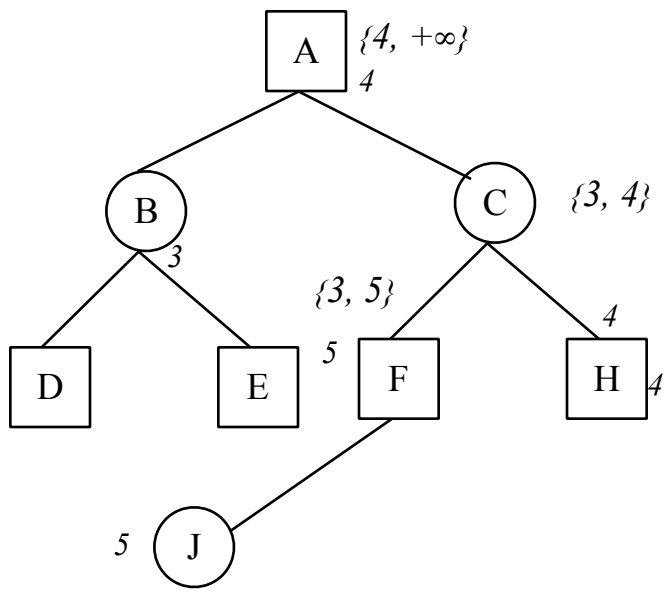
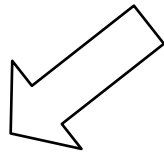
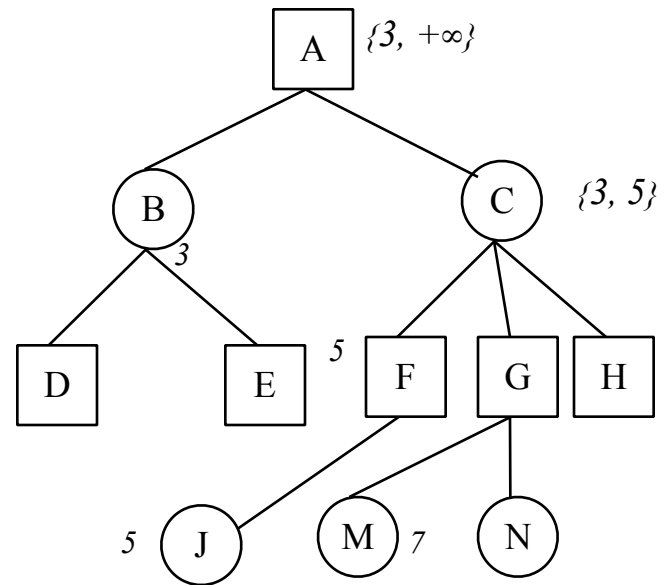
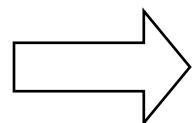
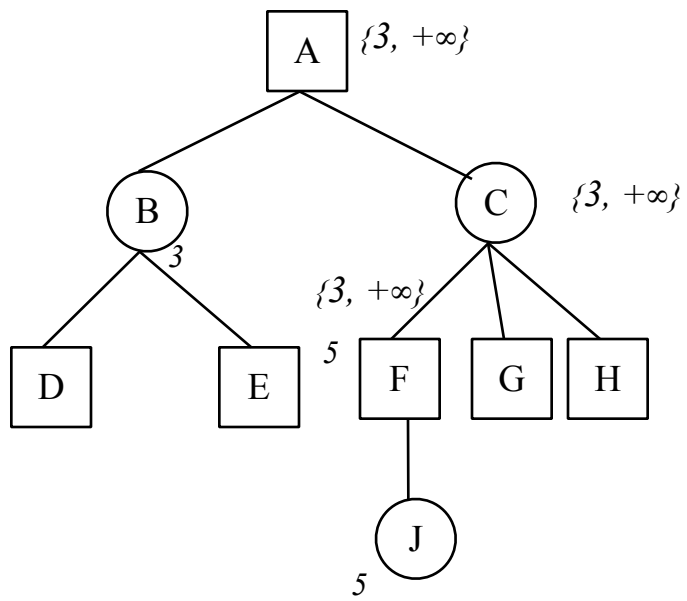
El recorrido se inicia llamando a la función valorMax con $\alpha=-\infty$ $\beta=+\infty$.

En la función valorMax α es el valor que se actualiza.

En la función valorMin β es el valor que se actualiza.



*Se puede podar I
ya que es un nodo min y
el valor de $v(K) = 0$ es $< \alpha = 3$*



Podemos podar G pues es un nodo max y el valor de M (7) > $\beta = 5$