Tema 2.
Resolución de problemas mediante búsqueda

Búsqueda con Adversarios: Juegos

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 de 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)
    - Primer programa de ajedrez: Alan Turing (1951).
    - Aprendizaje: Arthur Samuel (1952-57)
    - Poda del árbol de juego: MacCarthy (1956)
- **<u>Entorno multiagente</u> + <u>competición</u>**

  Los agentes tienen objetivos diferentes $\Rightarrow$ búsqueda con adversarios.
    - **Idealización** en la que los jugadores con **objetivos divergentes** alternan **turnos** en los que realizan acciones.
    - Los agentes sólo pueden realizar “**movimientos legales**” (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 (medido por 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
  - Información perfecta
  - Determinista
  - Juego de suma-cero

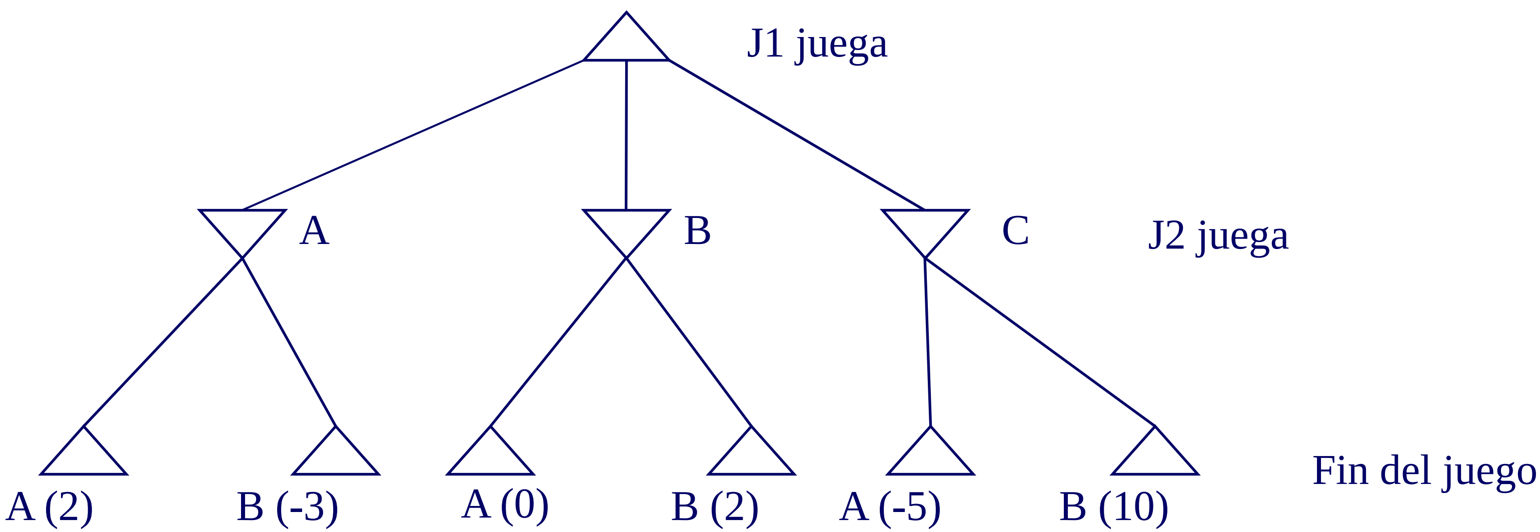
Representación de juegos

- 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

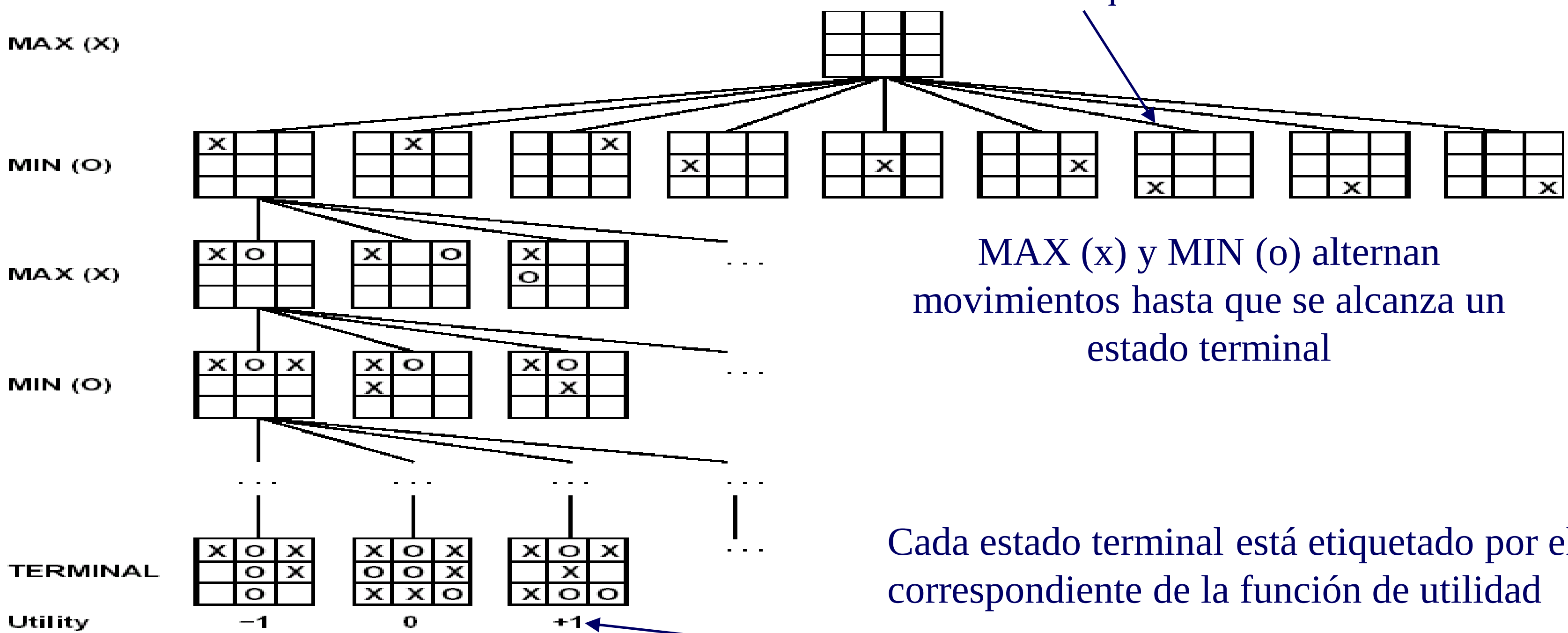
- Árbol del juego



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:
  - <u>Estado inicial</u>: Configuración inicial del tablero + identidad del primer jugador.
  - <u>Función sucesor</u>: Sucesores(n)
  - <u>Test terminal</u>: Terminal(n).
  - <u>Función de utilidad</u>: Utilidad(n), sólo si n es un nodo terminal.
- **Estrategia óptima para MAX**: Estrategia que con un resultado al menos tan bueno como cualquier otra estrategia, asumiendo que MIN es un oponente infalible.
- **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 óptimamente** desde ahí hasta el fin del juego

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

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))
```

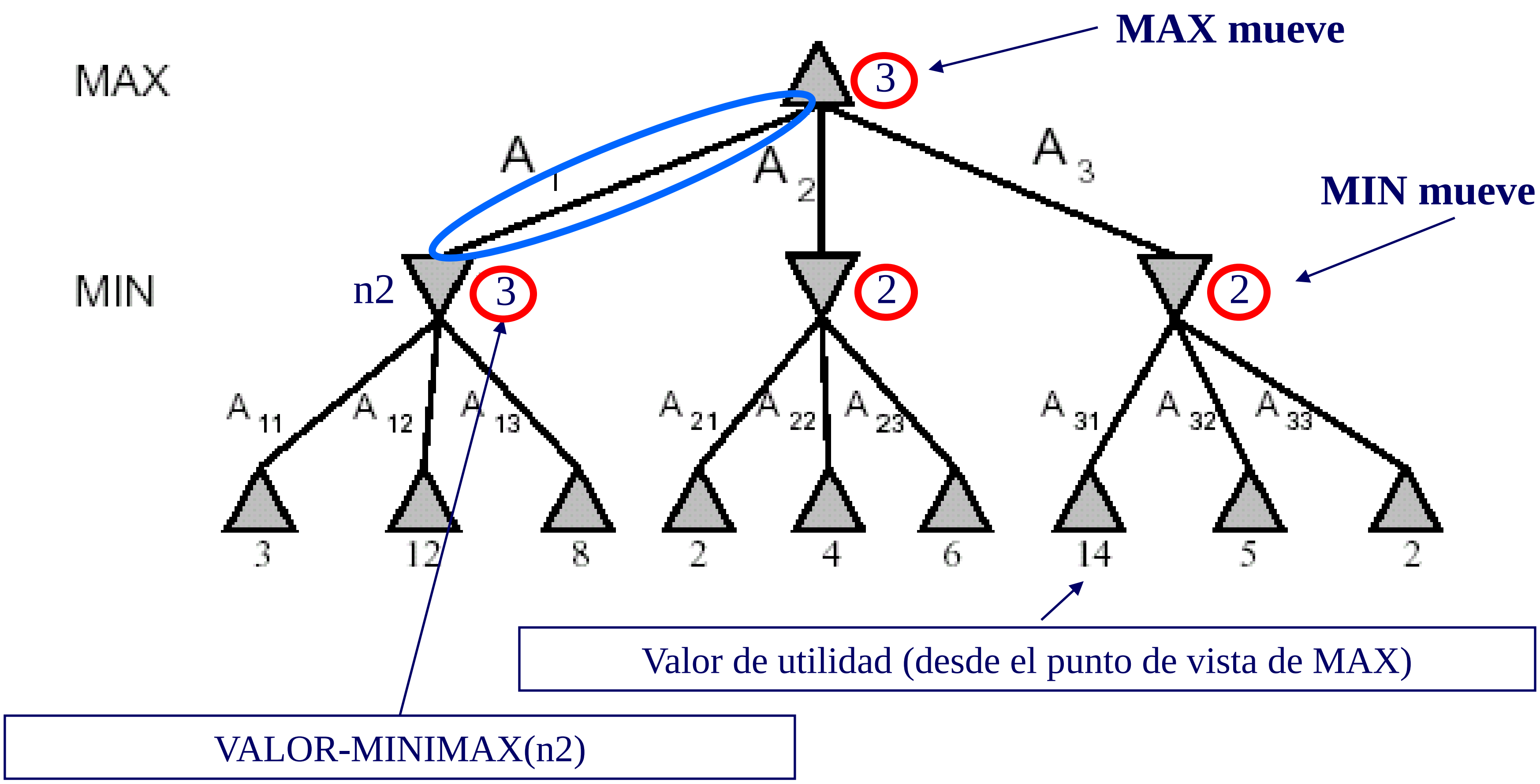
```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
    return v
```

```
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** (notar que puede haber estrategias óptimas finitas para árboles infinitos)
- **Óptima sólo si el oponente es óptimo** (si el oponente es subóptimo, podemos usar sus debilidades 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** $O(b \cdot m)$

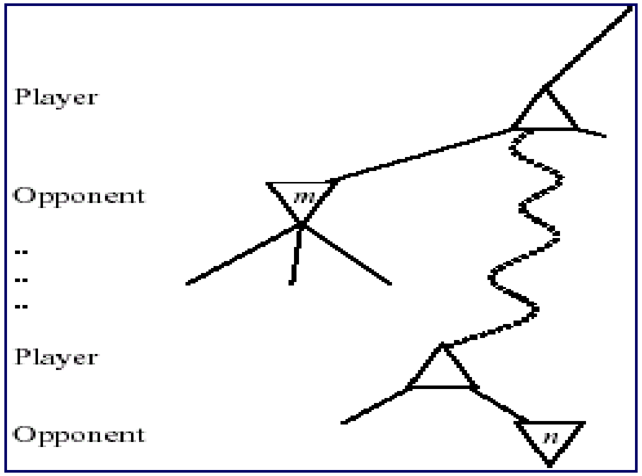
Procedimiento minimax: un ejemplo



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

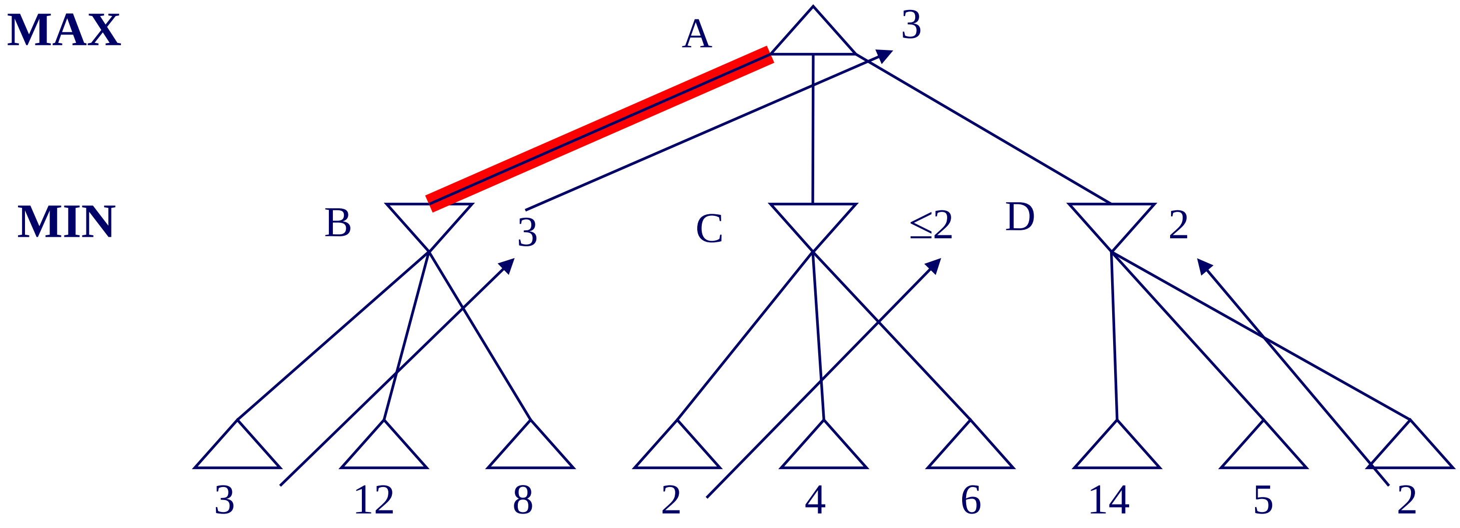
Poda alfa-beta

- 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** [α, β] 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 cuyo valor de $\beta \leq$ valor de α de cualquiera de sus antecesores MAX.
  - **β -cutoff**: Parar la búsqueda en un nodo MAX cuyo valor de $\alpha \geq$ valor de β de cualquiera de sus antecesores MIN.

Ejemplo: poda alfa-beta



Poda alfa-beta

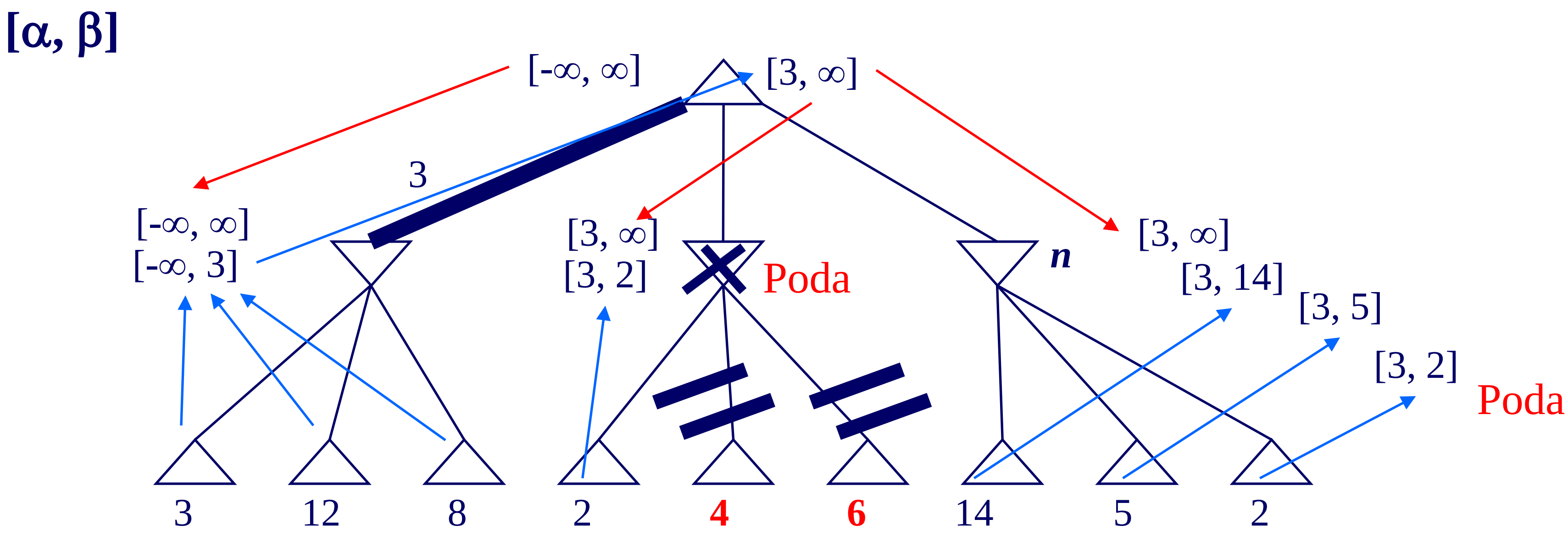
```
function ALPHA-BETA-DECISION(state) returns an action
   return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))
------------------------------------------------------------------------
function MAX-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           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: $O(b^{m/2})$** $\Rightarrow$ $b^* = b^{1/2}$.

    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



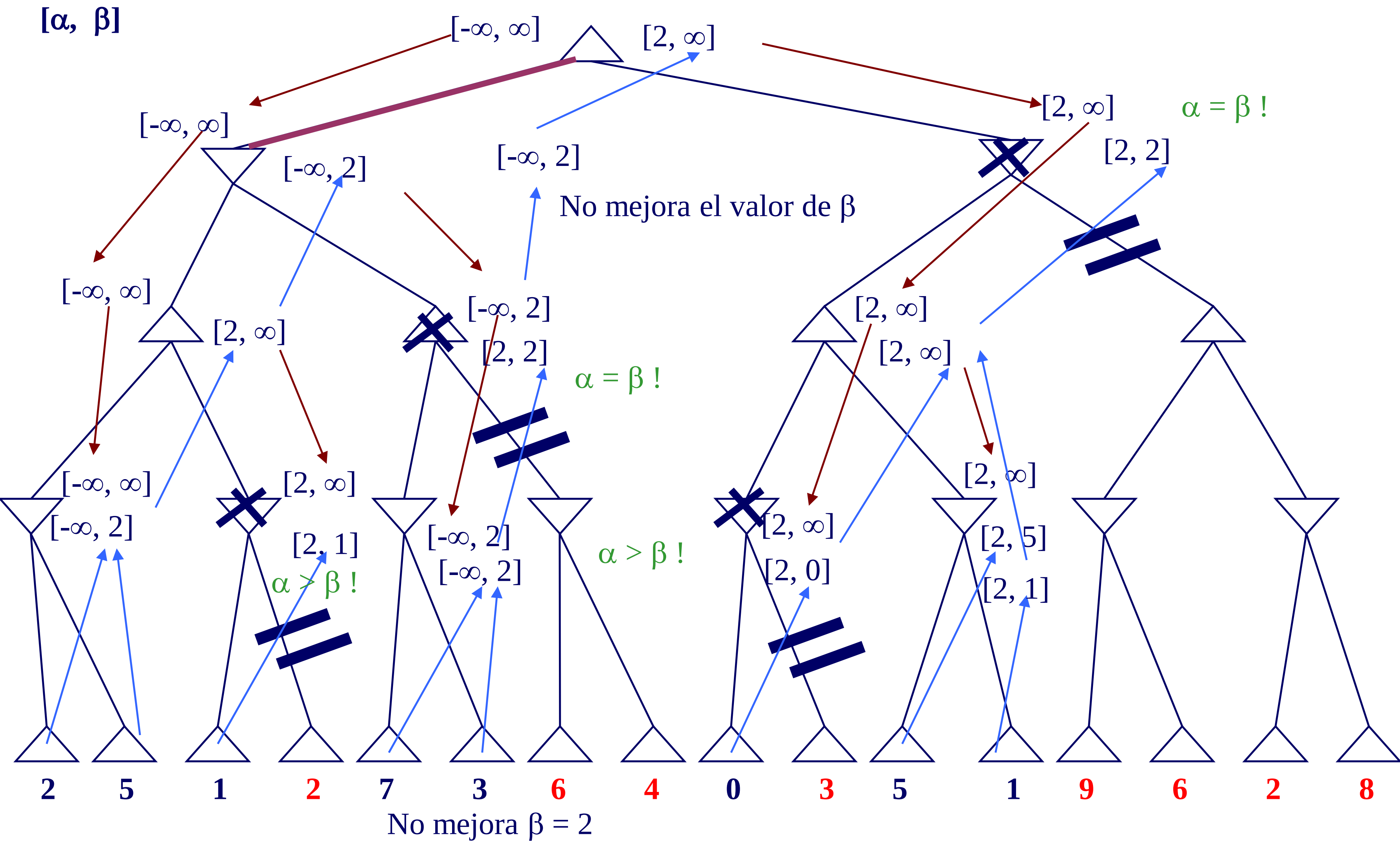
El orden en la búsqueda es importante para la eficacia de la poda

Algoritmo de poda

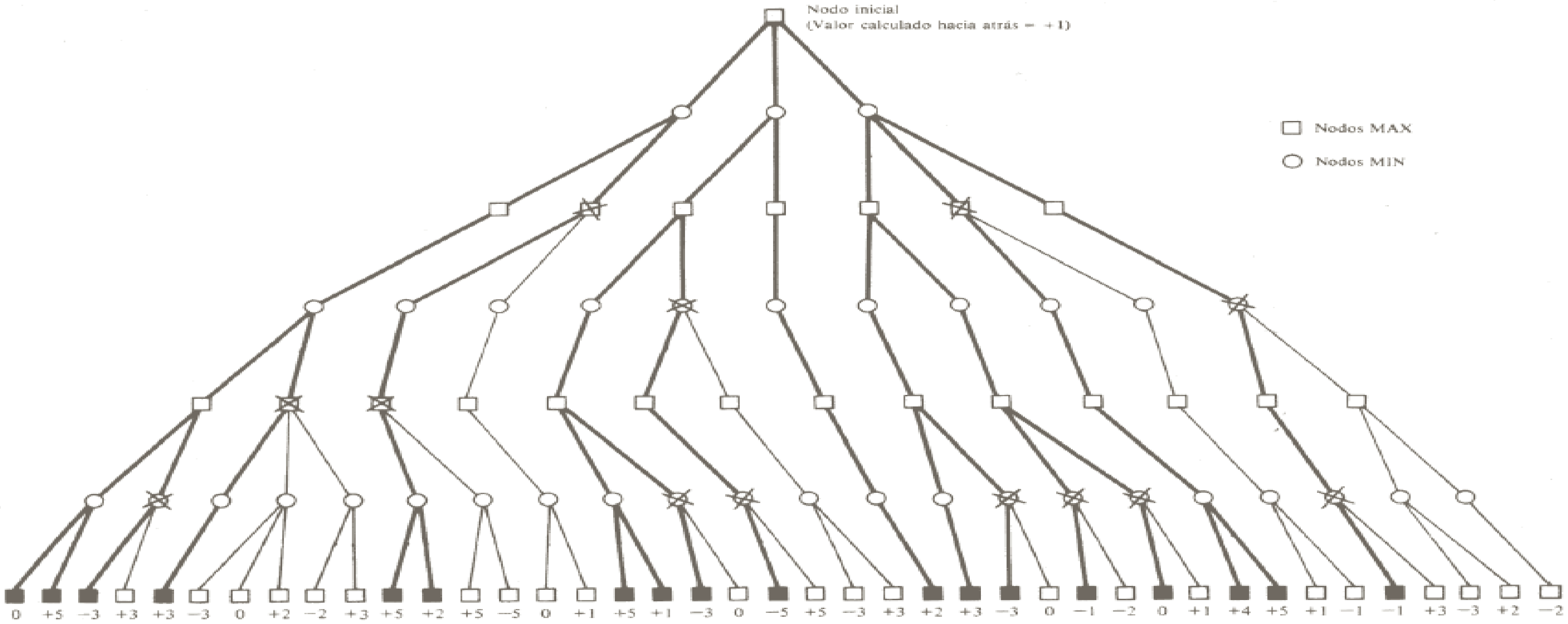
En nodo MAX: $\alpha \leftarrow \max(\alpha, \beta\text{'s de sucesores})$

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

Poda alfa-beta: Ejemplo, II



Poda alfa-beta: Ejemplo, III



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.

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**:

  Caracterizar al estado actual por un conjunto de **características** { $f_1(n)$, $f_2(n)$,…, $f_K(n)$ }.

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

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

F contiene conocimiento experto.

F puede ser **aprendido** (aprendizaje automático) de la experiencia (ej. se puede dejar al ordenador jugar contra sí mismo).

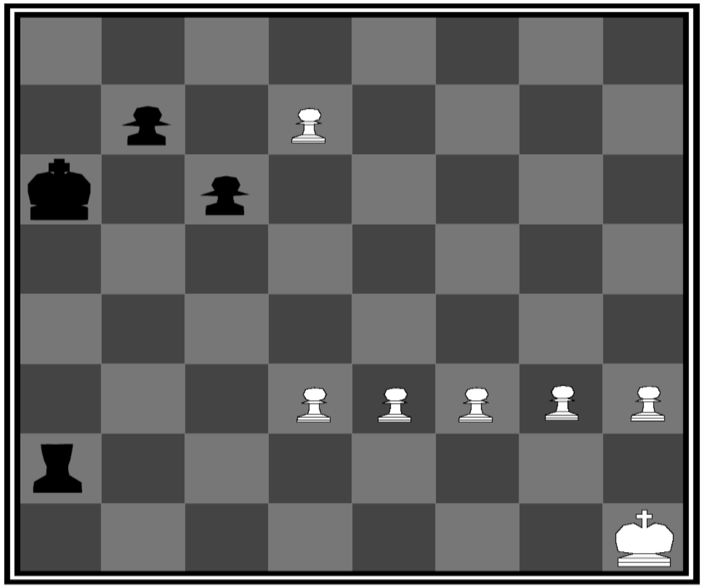
Búsqueda de horizonte limitado

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.
- <u>**Efecto del horizonte**</u>: 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

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.

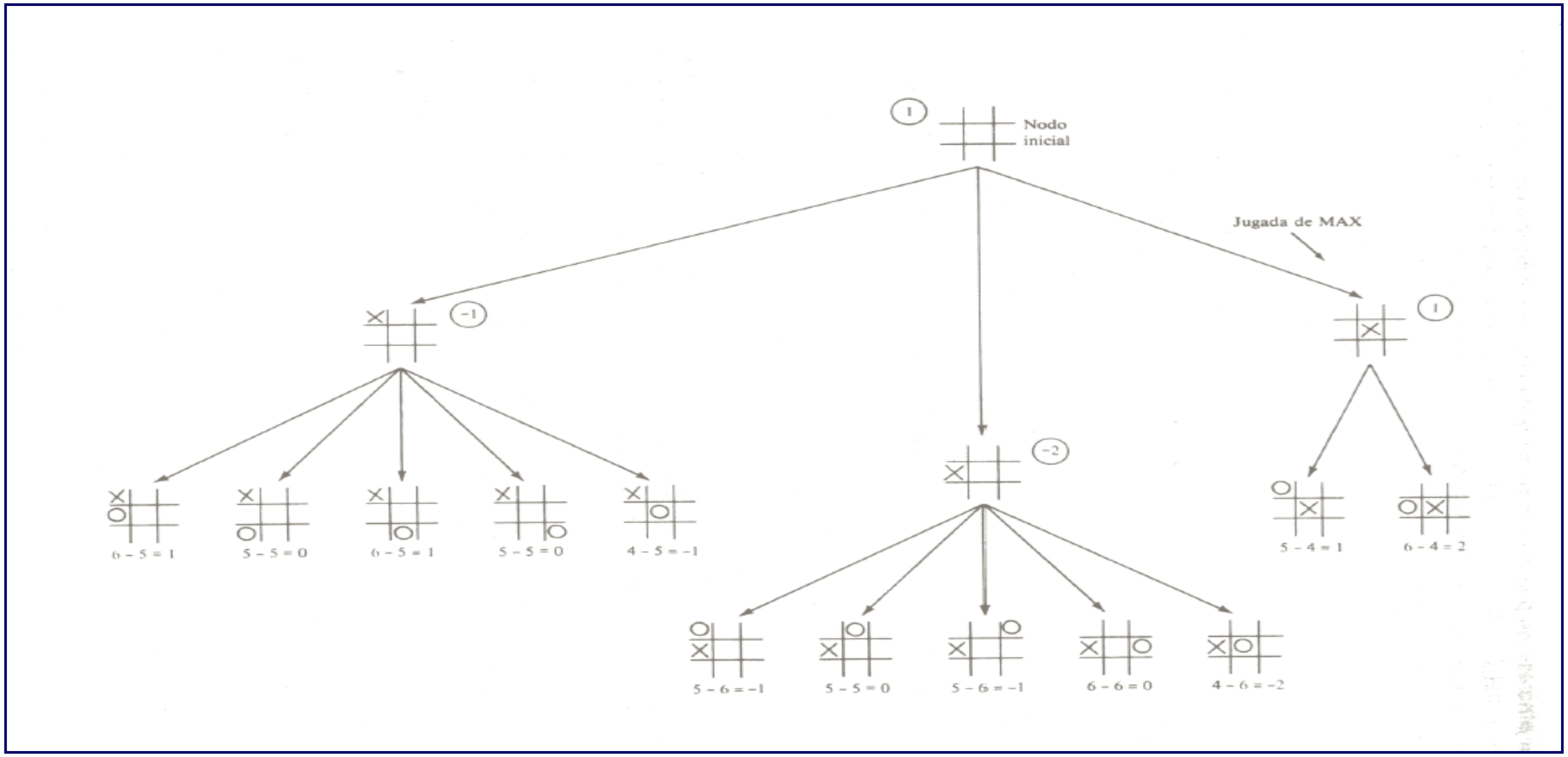
  f = (n-peones-blancos)*1 + (n-alfiles-blancos)*3 +...

  - (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)

Tres en raya

eval(n)	líneas-abiertas-MAX(n) – líneas-abiertas-MIN(n)	n no es un estado terminal
	+ ∞	Si gana MAX
	- ∞	Si gana MIN

líneas-abiertas-MAX: número de filas/columnas/diagonales abiertas para MAX.
líneas-abiertas-MIN: número de filas/columnas/diagonales abiertas para MIN.



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: $O(b^d \cdot n^d)$.
    b = factor de ramificación de nodos MAX / MIN.
    n = factor de ramificación de los nodos de RAND.
    d = límite en la búsqueda en profundidad.

  - Los nodos de RAND pueden ser podados.