## Ayudantía 6

# Búsqueda adversaria

**Por Daniel Toribio y Ignacio Villanueva**

6 de mayo de 2024

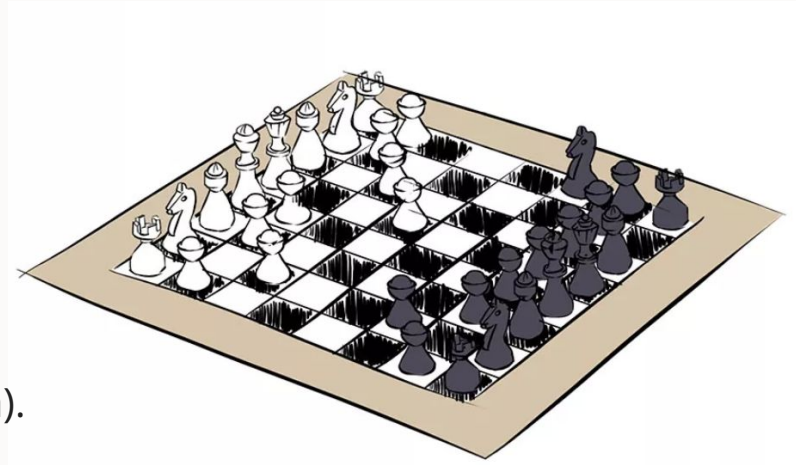
## Antes de comenzar...

# Menti!

# Tipos de problemas (Juegos)

Veremos juegos de **dos jugadores**, con **turnos**, **información perfecta** y **suma cero.**

- **Información perfecta**: Tenemos toda la información del tablero (estado) del juego.
- **Suma cero**: Lo que es bueno para mí, es malo para mi oponente en igual medida (y vice versa).



# Búsqueda

Podemos considerar un juego como un **espacio de búsqueda**:

- **Tablero** = nodo/estado
- **Jugadas** = conexiones/acciones

El **objetivo** será un **tablero donde ganamos** (estado objetivo):

- Gato: Tres fichas (nuestras) en línea
- Conecta-4: Cuatro fichas (nuestras) en línea
- Ajedrez: Jaque-mate al rey del oponente

# Adversario

Para cada estado que revisemos en la búsqueda, tenemos que considerar si es **mi turno, o el turno de mi adversario.**

La búsqueda puede presentarse como una simulación del juego entre dos jugadores: Max y Min:

Si es mi turno: **Max**

- Elijo la mejor jugada que tengo disponible, para **max**imizar el puntaje

Si es el turno de mi oponente: **Min**

- Asumo que mi oponente siempre toma las decisiones óptimas (para él) entonces elige la opción que **min**imiza mi puntaje (la mejor jugada para mi oponente es la que me “hace peor”).

# Adversario

***Observación***

Yo **no sé** cómo va a jugar mi adversario, pero si **asumo que es un genio** (siempre juega lo que es mejor para él/peor para mí) estaré preparada para lo peor.

Luego si mi adversario toma decisiones subóptimas durante el juego, sólo es mejor para mí.

# ¿Cómo defino la mejor jugada desde un tablero?

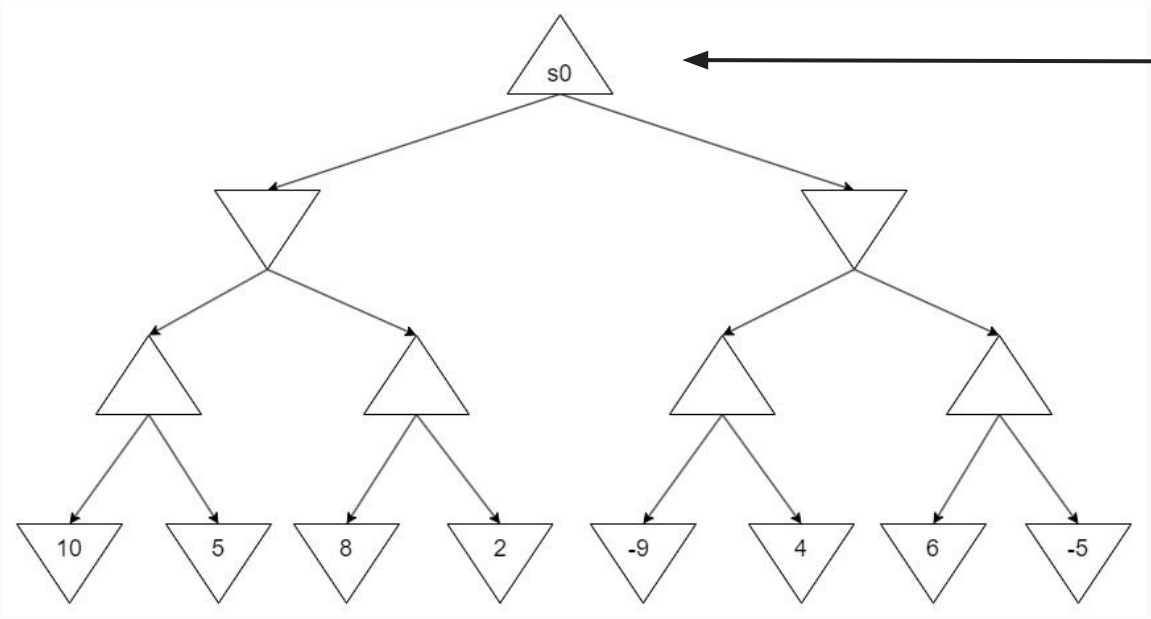
Puntuamos el tablero en base al valor *minimax*.

Valor del tablero para Max

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

Para jugar de manera óptima, elegimos la acción asociada al valor *minimax*

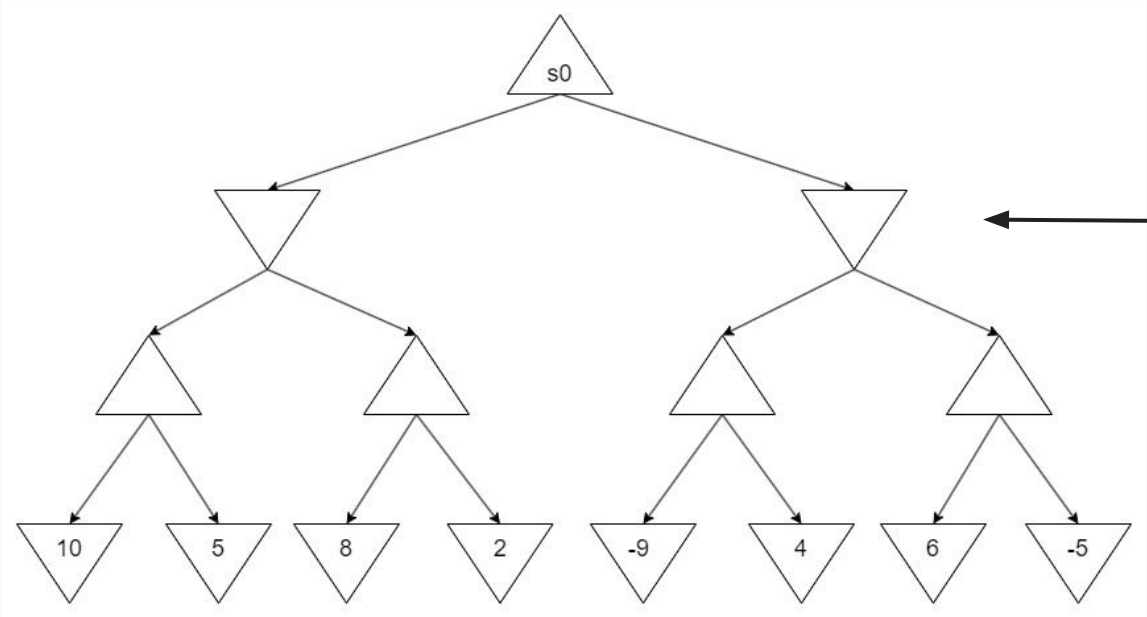
# Cálculo del valor MiniMax




Turno de Max

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

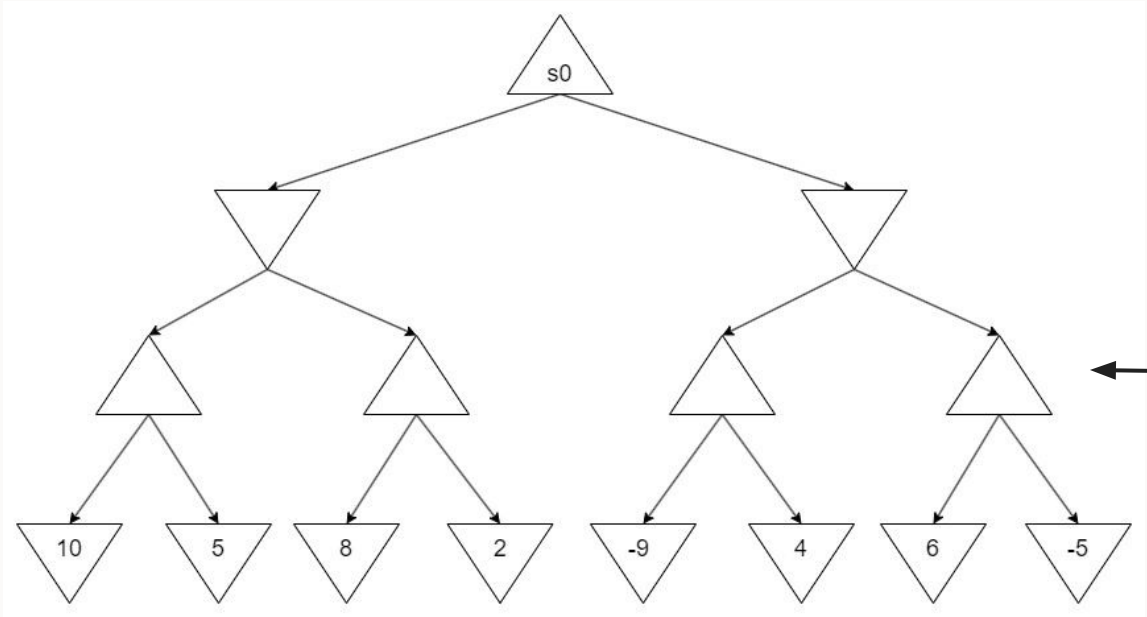
# Cálculo del valor MiniMax




Turno de Min

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

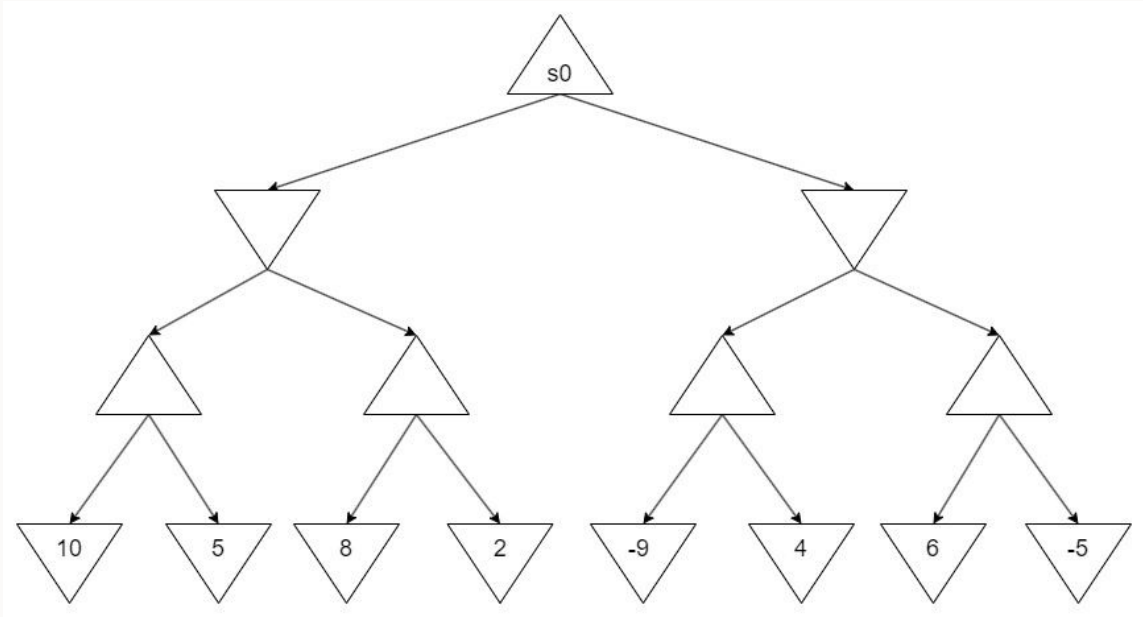
# Cálculo del valor MiniMax




Turno de Max

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

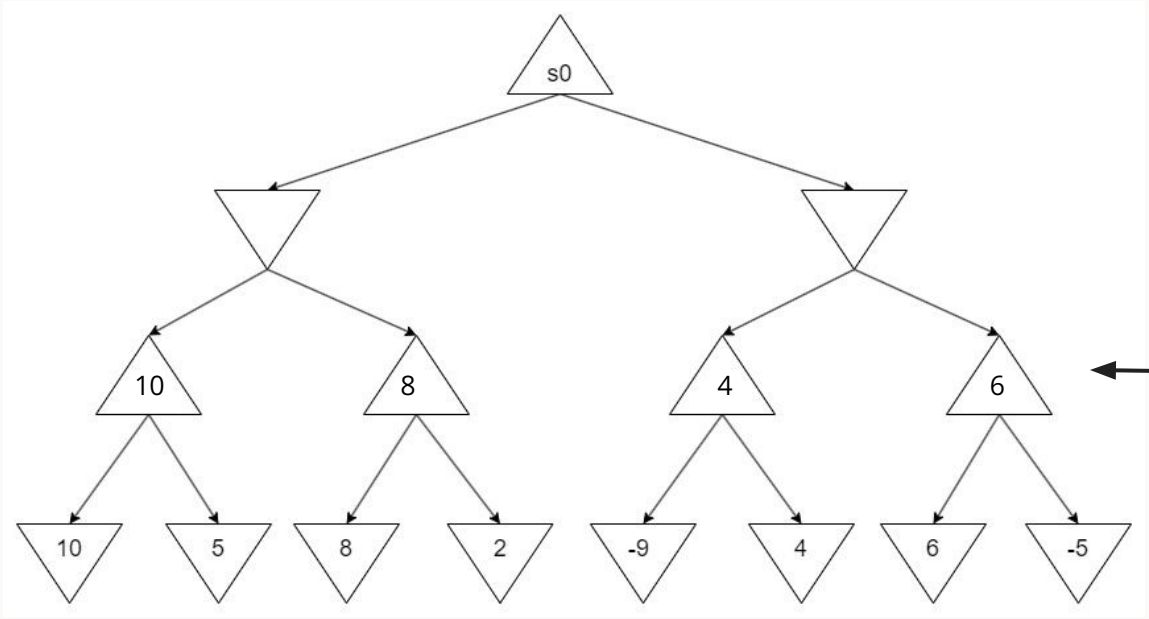
# Cálculo del valor MiniMax




Estado terminal

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

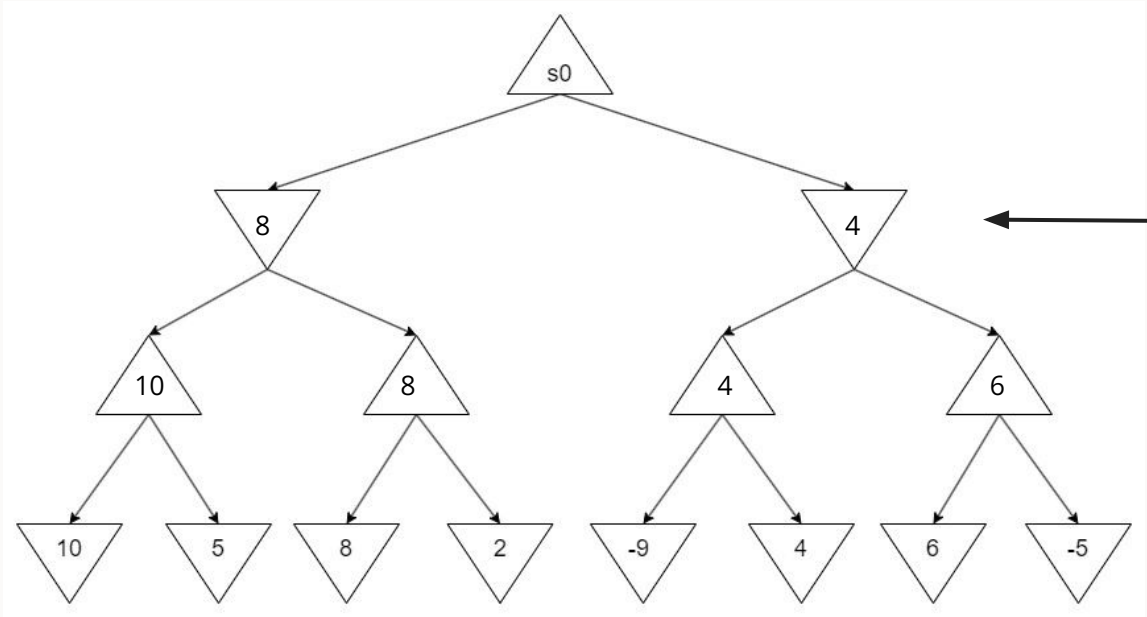
# Cálculo del valor MiniMax




Turno de Max

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

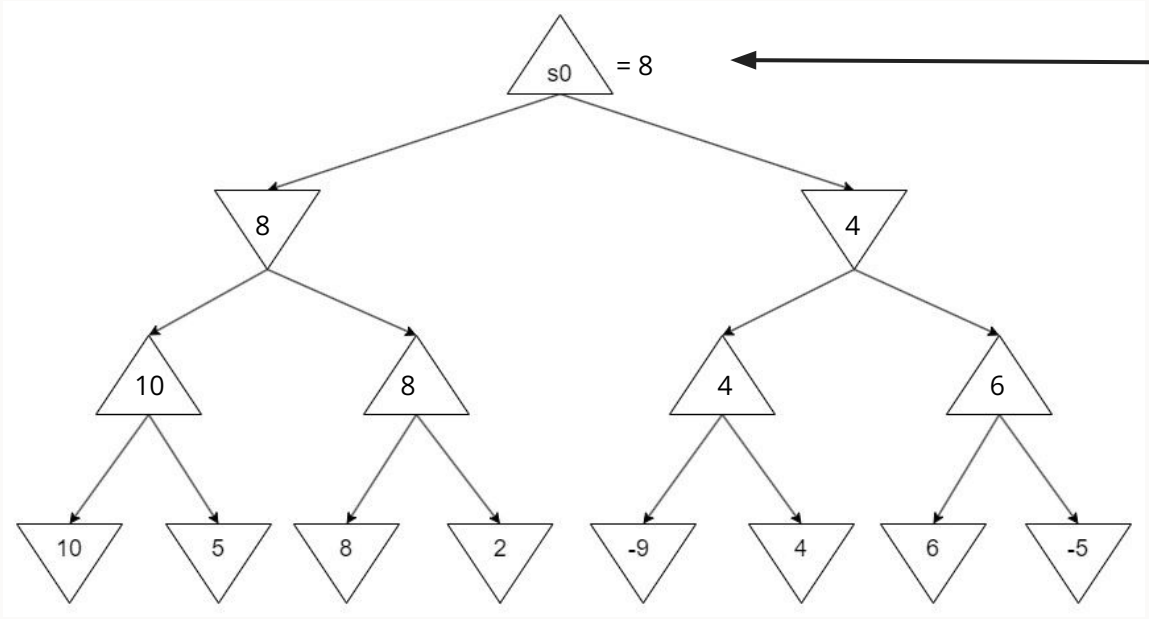
# Cálculo del valor MiniMax




Turno de Min

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

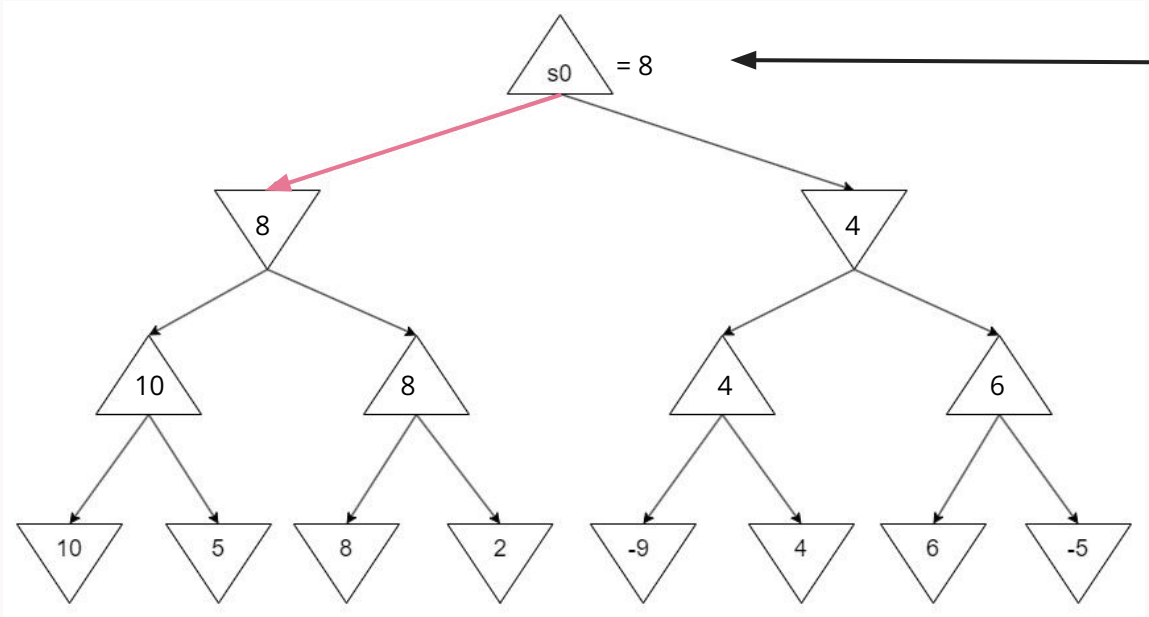
# Cálculo del valor MiniMax




Turno de Max
(retorno de MiniMax)

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

# Cálculo del valor MiniMax




Elijo la acción que me lleva al estado asociado al 8 (hijo izquierdo)

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

# Búsqueda MiniMax

Es un algoritmo que recibe un tablero y retorna la acción asociada al valor minimax de dicho tablero

```
function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

# Menti!

# Rendimiento MiniMax

La búsqueda MiniMax corre DFS **sobre el árbol de estados completo**:

→ Para el ajedrez, esto es explorar $10^{44}$ estados

Para mejorar el rendimiento podemos:

- **Acotar** la profundidad del árbol de búsqueda
- **Podar** ramas del árbol de búsqueda

# Acotar profundidad del árbol de búsqueda

## Función de evaluación

- Es como una **heurística** pero en el contexto de búsqueda adversaria
- **Asigna puntajes a estados no terminales**, lo que permite limitar a un máximo la altura de un árbol

→ Básicamente, hace una estimación sobre “quién va ganando”

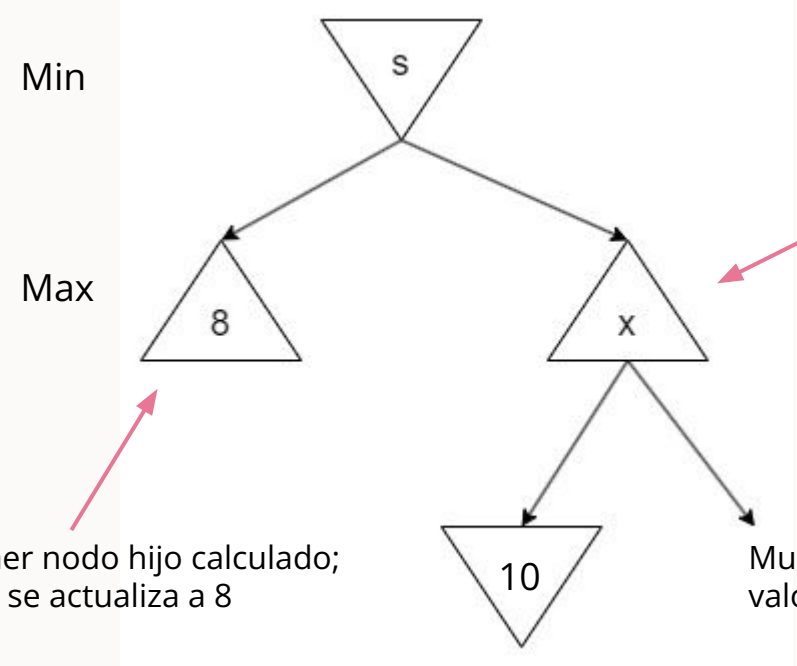
# Poda Alpha-Beta

Podemos podar buena parte del árbol si guardamos dos parámetros adicionales en cada llamada a la búsqueda MiniMax:

- Alpha: cota **inferior** del valor de un nodo
- Beta: cota **superior** del valor de un nodo

Tener estas cotas nos indica si es necesario calcular un nodo.

# Poda Alpha-Beta - Cota superior Beta

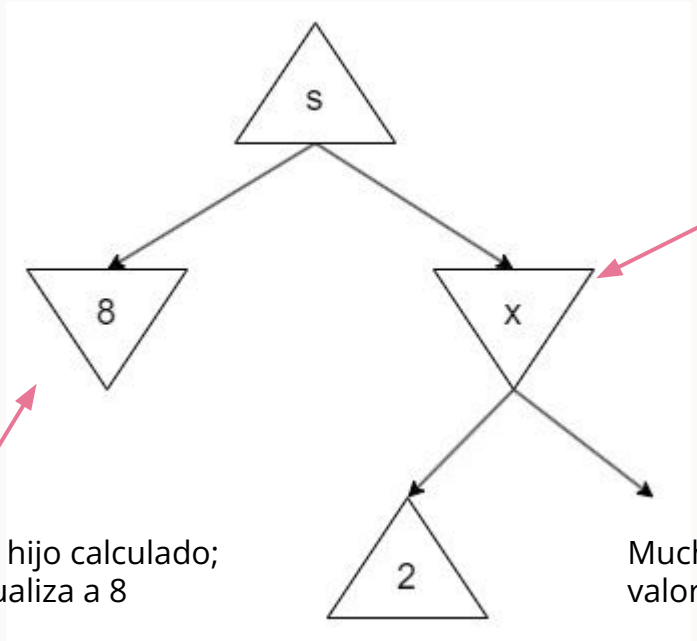



Sabemos que x ≥ 10 apenas verificamos el valor del primer nodo hijo. Como nuestro beta es igual a 8, sabemos que el nodo *s* **no** optará por el valor *x*. Luego podemos omitir el cálculo de los nodos hijos aún no calculados y llegar a la misma elección (podamos el sub-árbol)

Primer nodo hijo calculado; beta se actualiza a 8

Muchos nodos hijos cuyo valor no conocemos

# Poda Alpha-Beta - Cota inferior Alpha




Sabemos que x ≤ 2 apenas verificamos el valor del primer nodo hijo. Como nuestro alpha es igual a 8, sabemos que el nodo *s* **no** optará por el valor *x*. Luego podemos omitir el cálculo de los nodos hijos aún no calculados y llegar a la misma elección (podamos el sub-árbol)

Primer nodo hijo calculado; alpha se actualiza a 8

Muchos nodos hijos cuyo valor no conocemos

# Poda Alpha-Beta

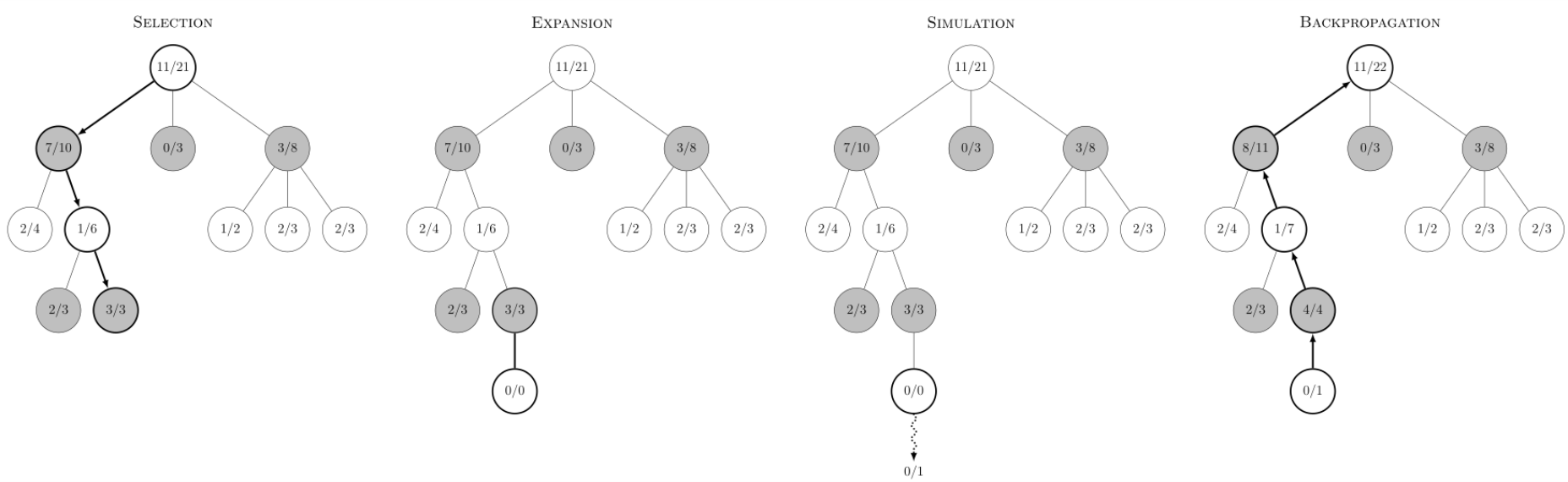
```
function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

# Menti!

# Monte Carlo tree search

- Es otro algoritmo de búsqueda con adversario.
- Se usa en problemas en que es **difícil definir una buena función de evaluación** y/o en problemas con un **alto factor de ramificación**.
- El valor de un estado se **estima** como el promedio de los **valores obtenidos** (ratio partidas ganadas/partidas totales) tras varias **simulaciones completas del juego** (partidas finalizadas).
- Para seleccionar qué acción tomamos en un estado, usamos **políticas de selección** como UCB1.
- Se basa en **4 etapas**: Selección, Expansión, Simulación, Retropropagación

# Monte Carlo tree search




By Robert Moss - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=88889583

Ayudantía 6

# Búsqueda adversaria

Por Daniel Toribio y Ignacio Villanueva

6 de mayo de 2024