

# *Inteligencia Artificial*

## Búsqueda adversarial y Árboles de juegos

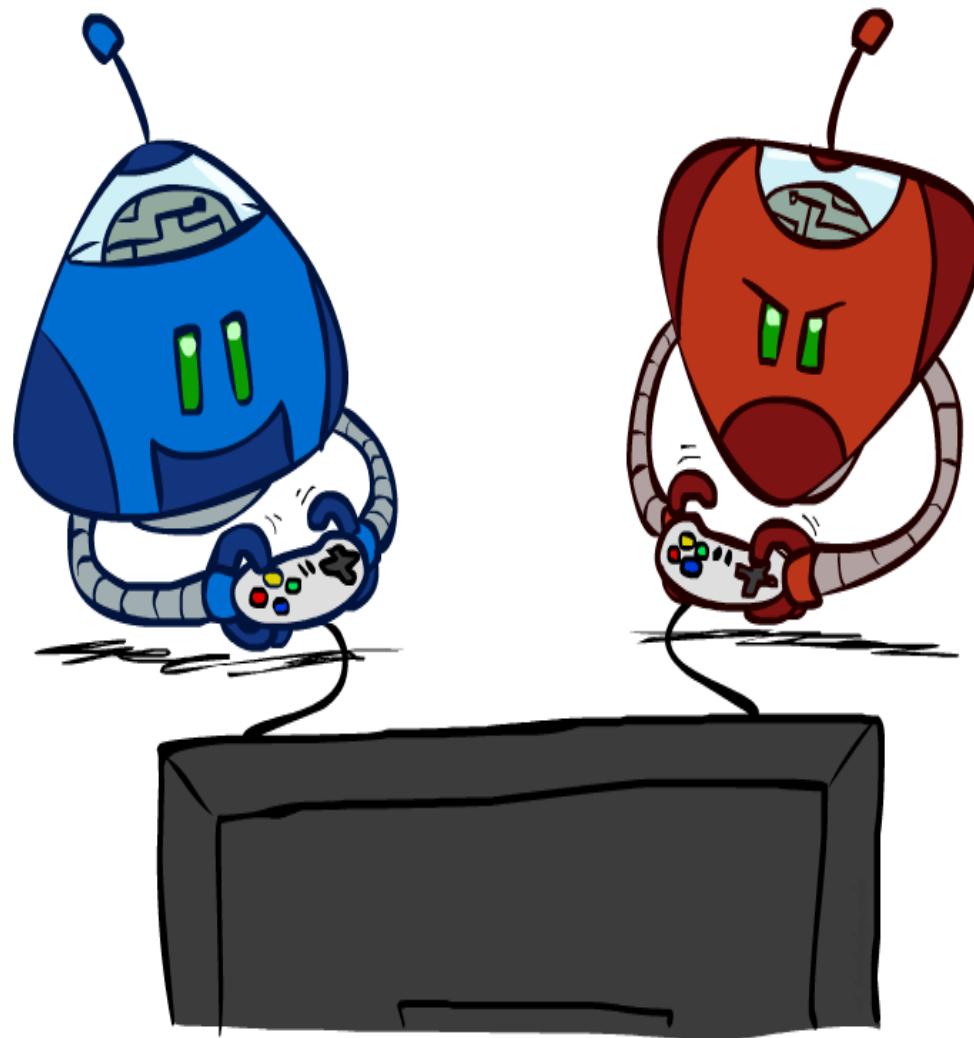
Aitziber Atutxa

[transparencias de Koldo Gojenola y Ekaitz Jauregi adaptadas de Berkeley: Dan Klein, Pieter Abbeel]

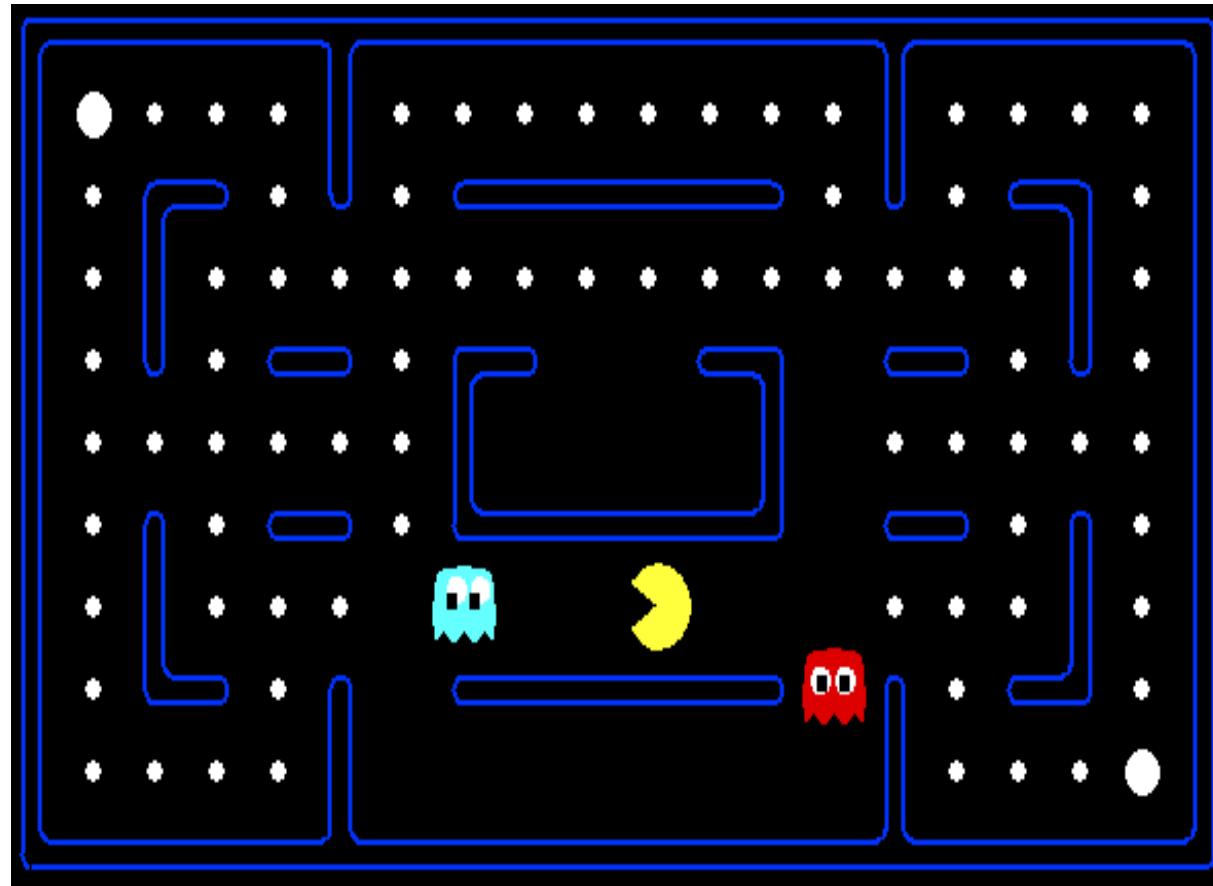


Universidad  
del País Vasco

Euskal Herriko  
Unibertsitatea

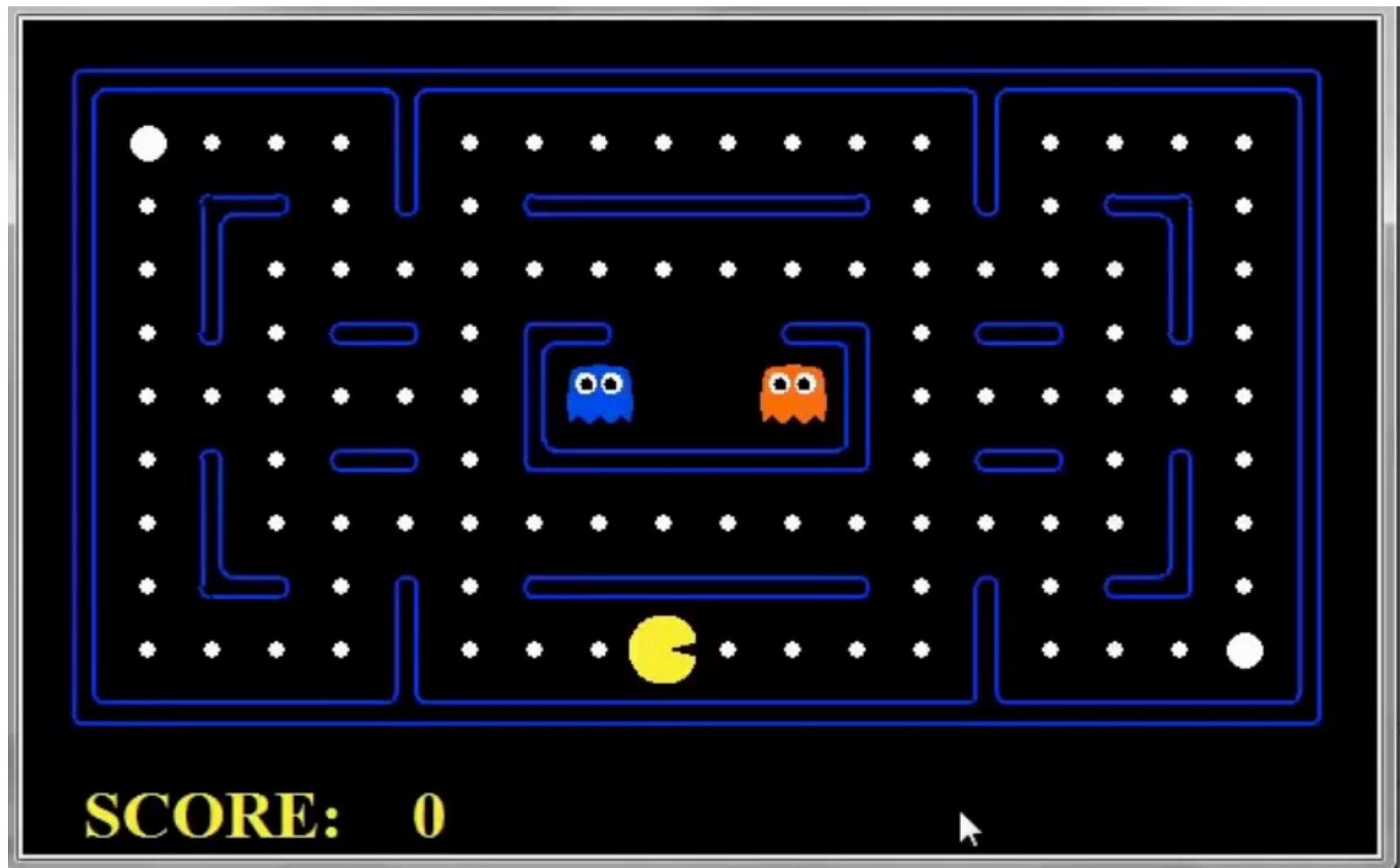


# Comportamiento y computación

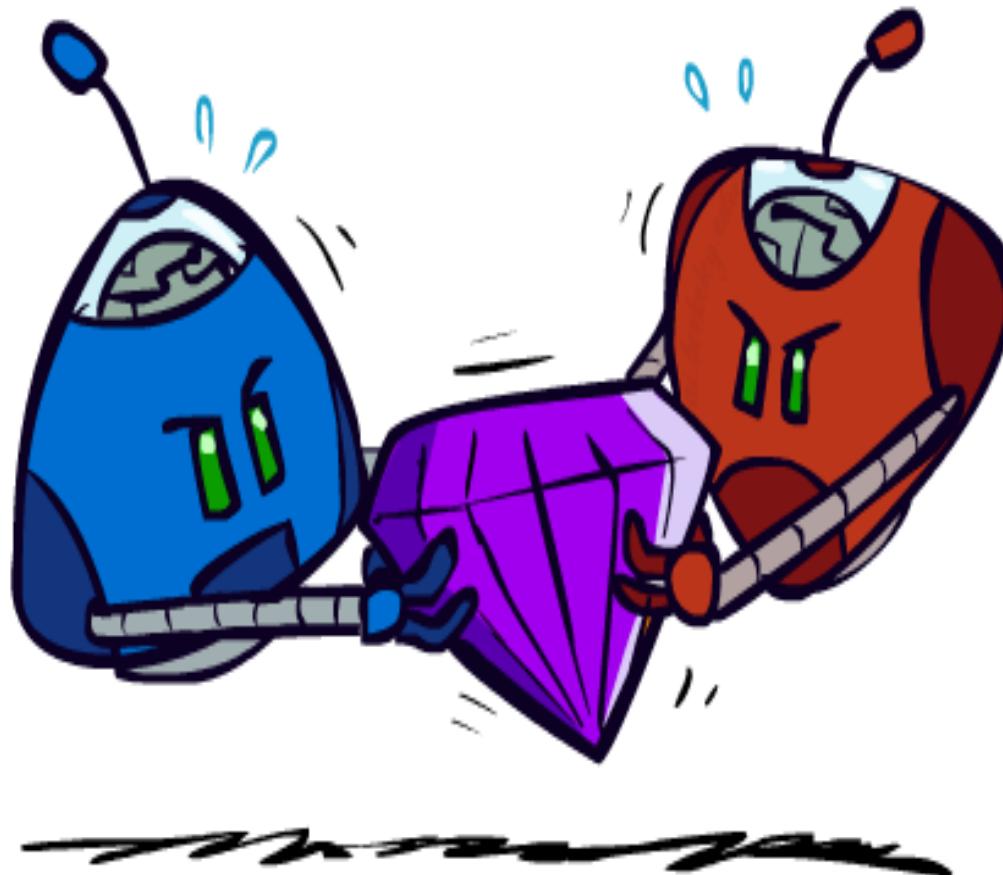


[Demo: mystery pacman (L6D1)]

# Video Demo Mystery Pacman

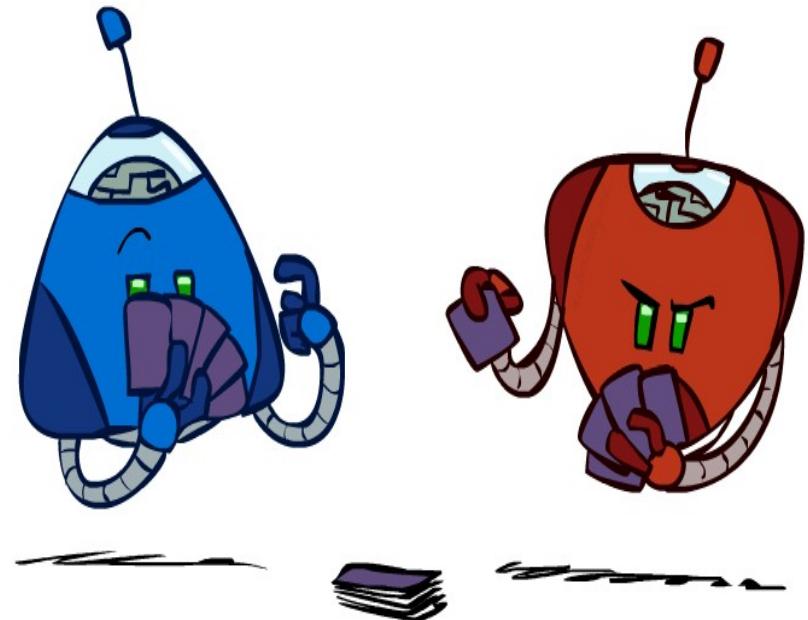


# Juegos Adversariales



# Tipos de juegos

- ¡Hay muchas clases de juegos!
- A tener en cuenta:
  - ¿Determinístico o estocástico?
  - ¿Uno, dos o más jugadores?
  - ¿Suma cero?
  - ¿Información perfecta (podemos ver el estado)?
- Queremos algoritmos para calcular una estrategia (política o policy) que recomiende un movimiento desde cada estado

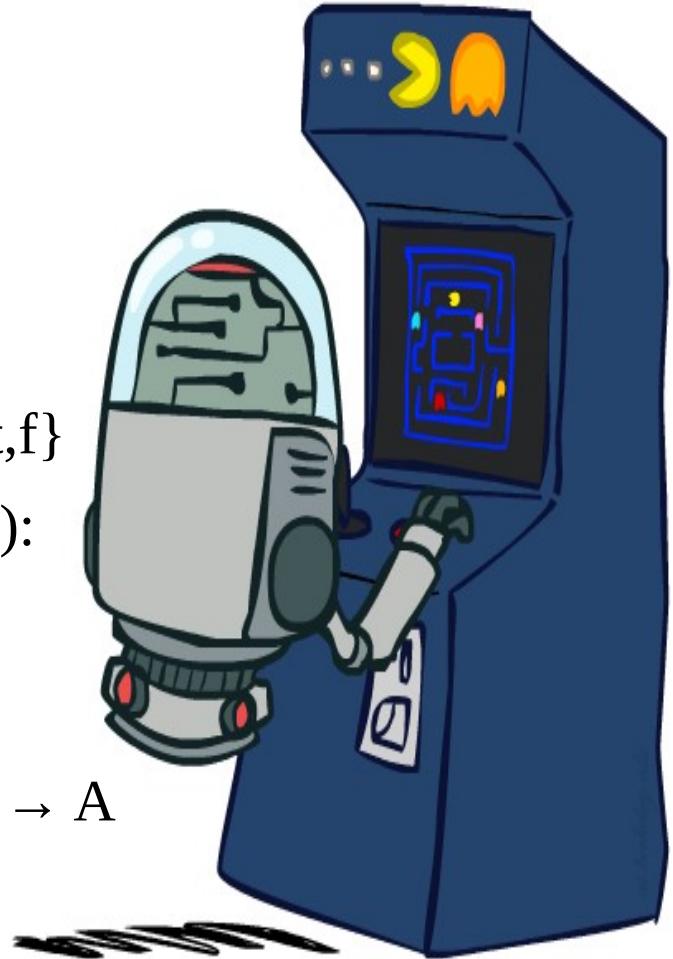


# Juegos determinísticos

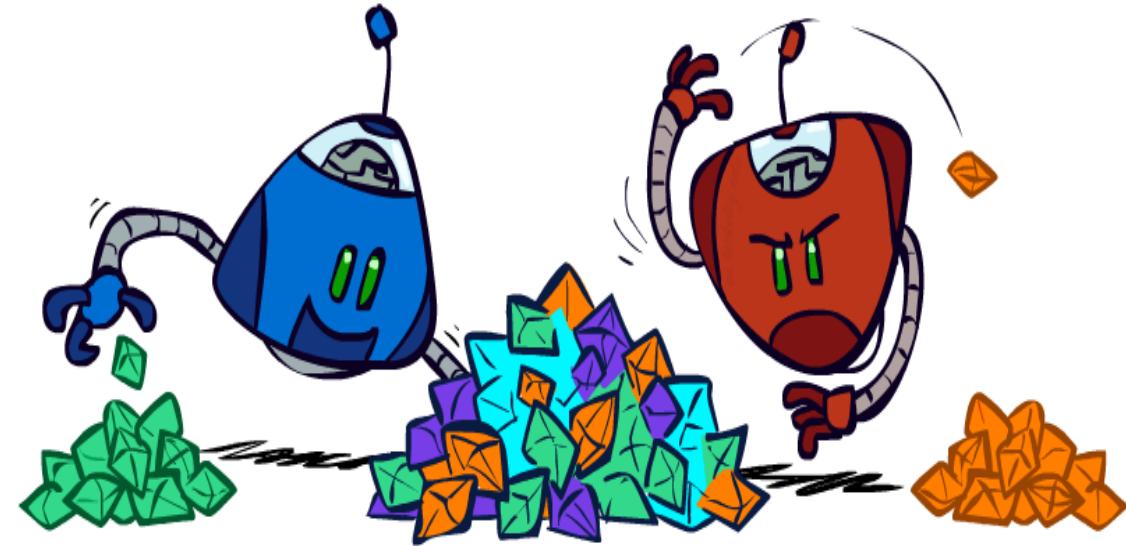
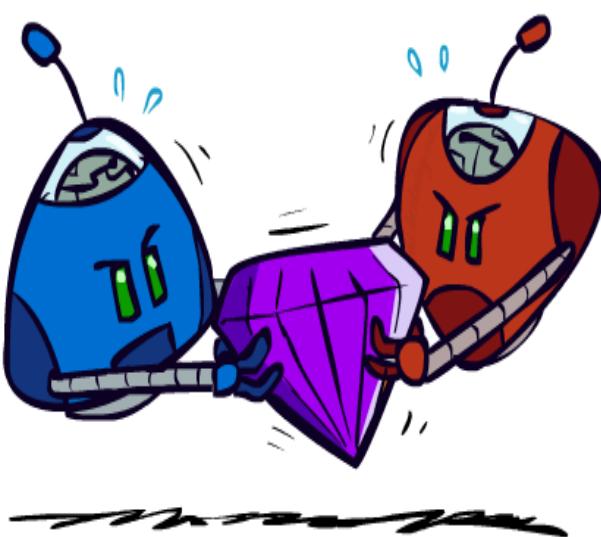
➤ Muchas formalizaciones posibles, una es:

- Estados:  $S$  (inicio en  $s_0$ )
- Jugadores:  $P=\{1\dots N\}$  (normalmente a turnos)
- Acciones:  $A$  (puede depender del jugador / estado)
- Función de transición:  $S \times A \rightarrow S$
- Test de terminación (estado objetivo o final):  $S \rightarrow \{t,f\}$
- Función de utilidad para terminal (Terminal Utilities):
- ¿Qué valor tiene este estado final
- para un jugador?  $S \times P \rightarrow R$

➤ La solución para un jugador es una política (**policy**):  $S \rightarrow A$



# Juegos de suma cero



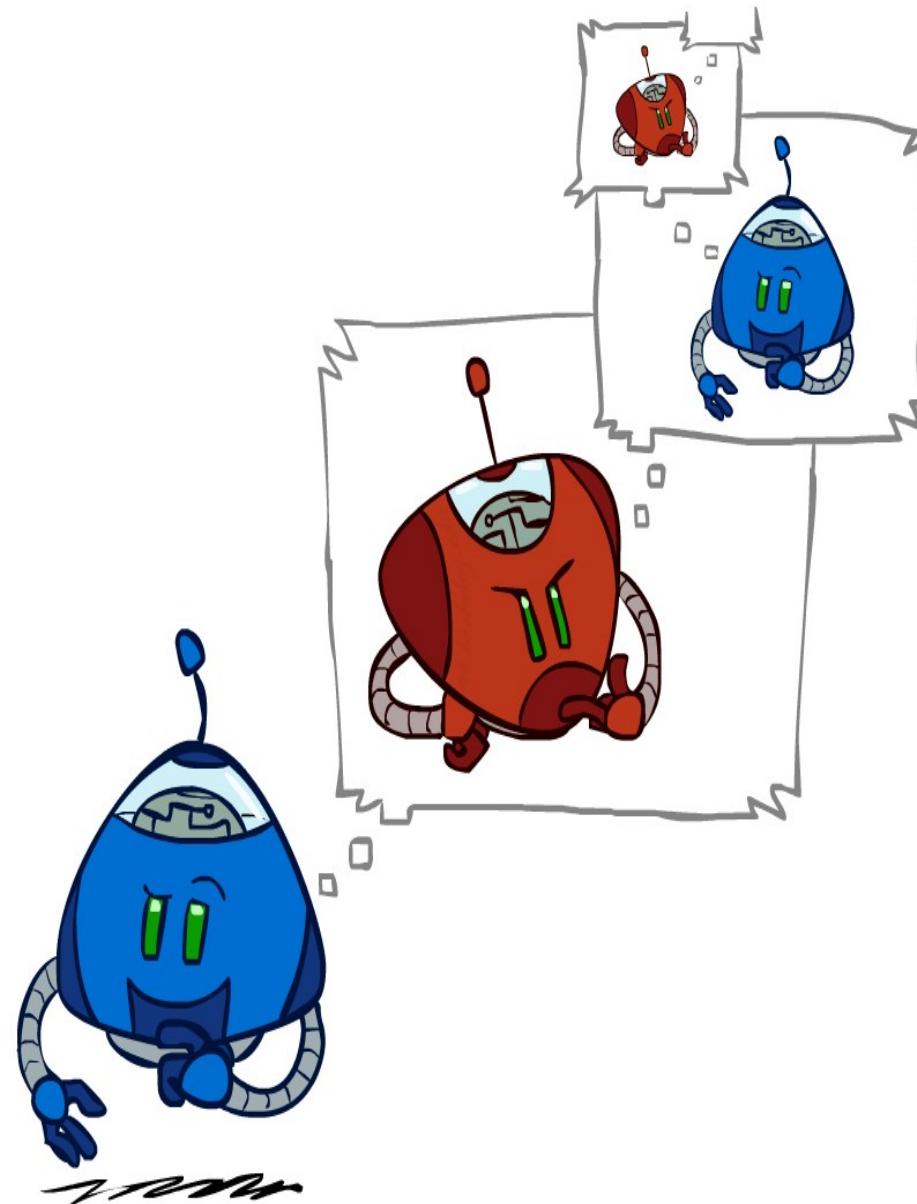
## ➤ Juegos de suma cero

- Los agentes tienen utilidades opuestas (valores)
- Podemos pensar en un único valor que uno maximiza y el otro minimiza
- Adversarial, competición pura

## ➤ Juegos Generales

- Los agentes tienen utilidades independientes (valores)
- Cooperación, indiferencia, competición, y más, todo es posible
- Más después

# Búsqueda Adversarial

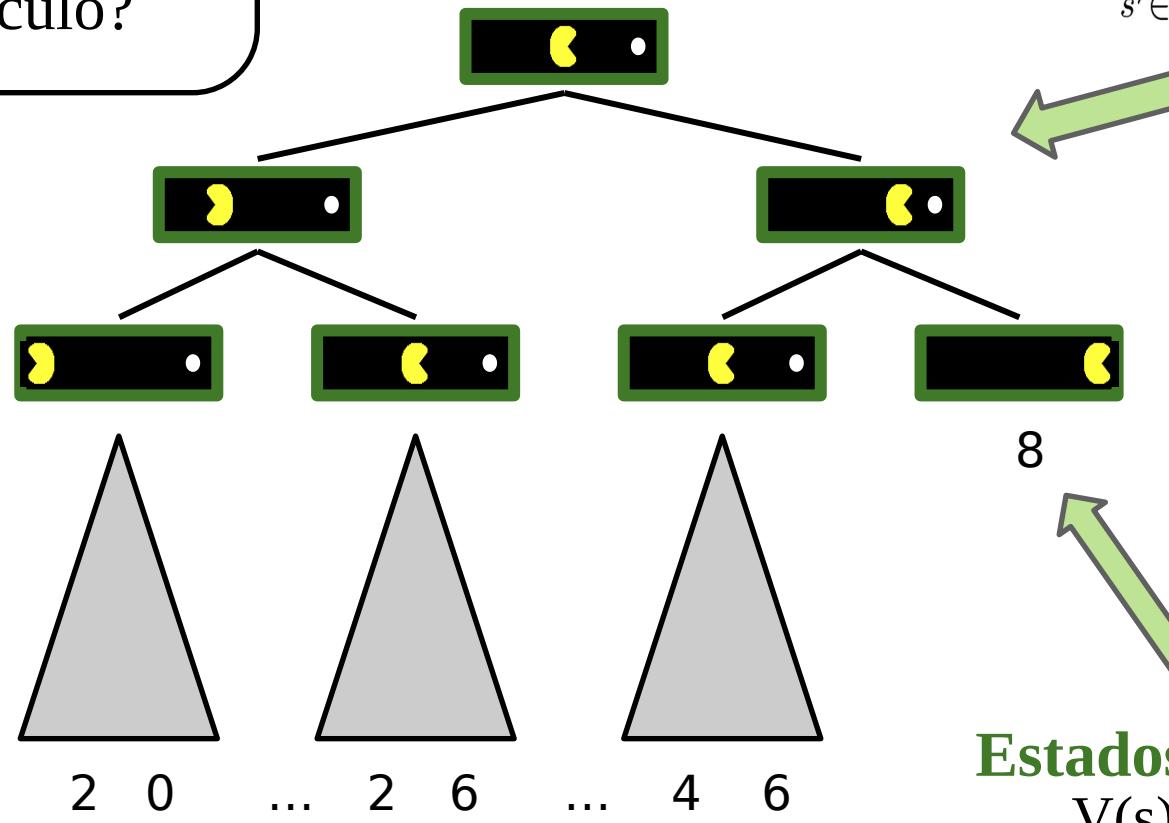


# Valor de un estado

Valor de un estado: El mejor resultado (utility) desde ese estado.  
¿Cómo lo calculo?

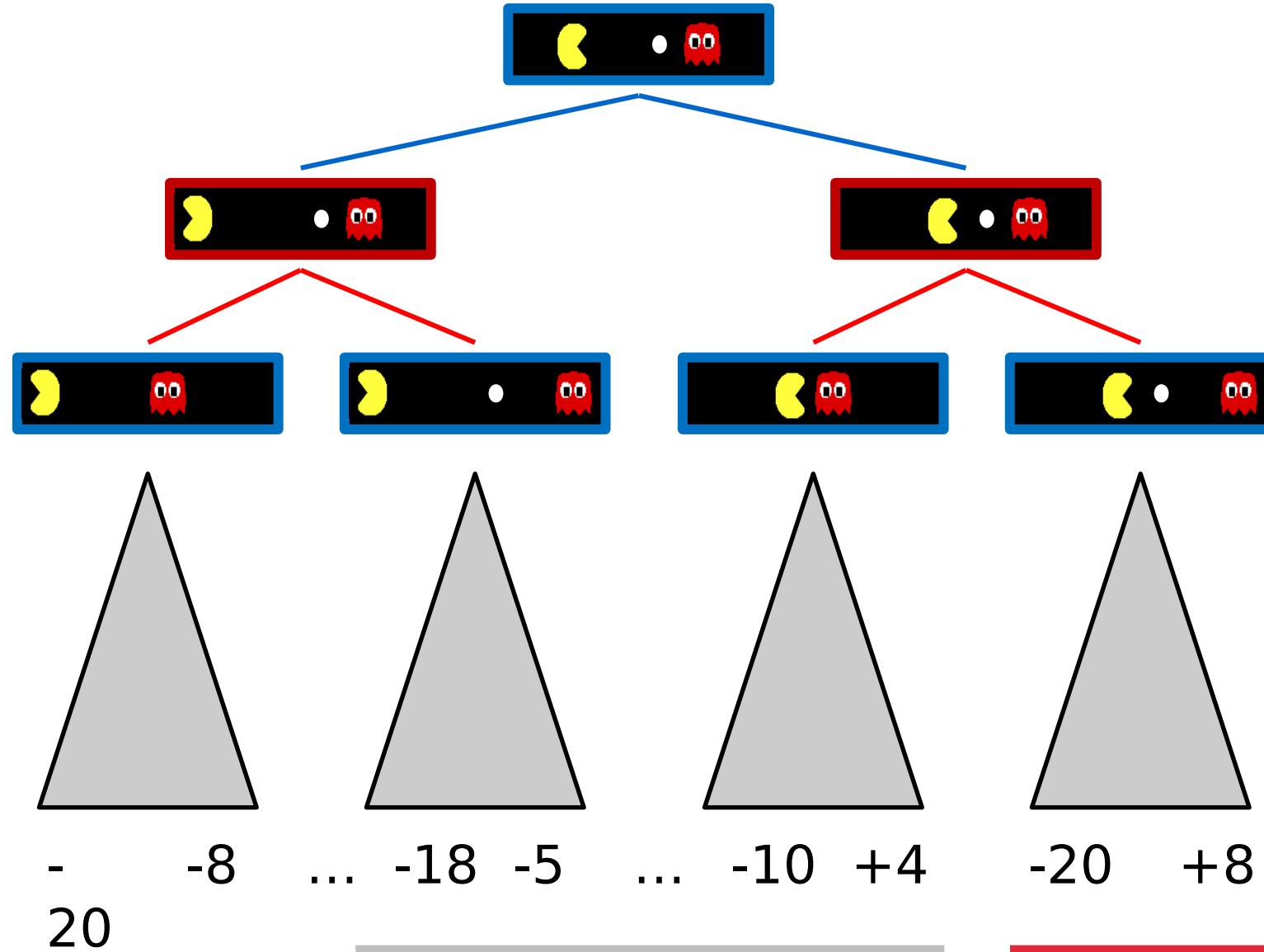
Estados no terminales:

$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



Estados terminales:  
 $V(s) = \text{conocido}$

# Árboles de estados Adversariales



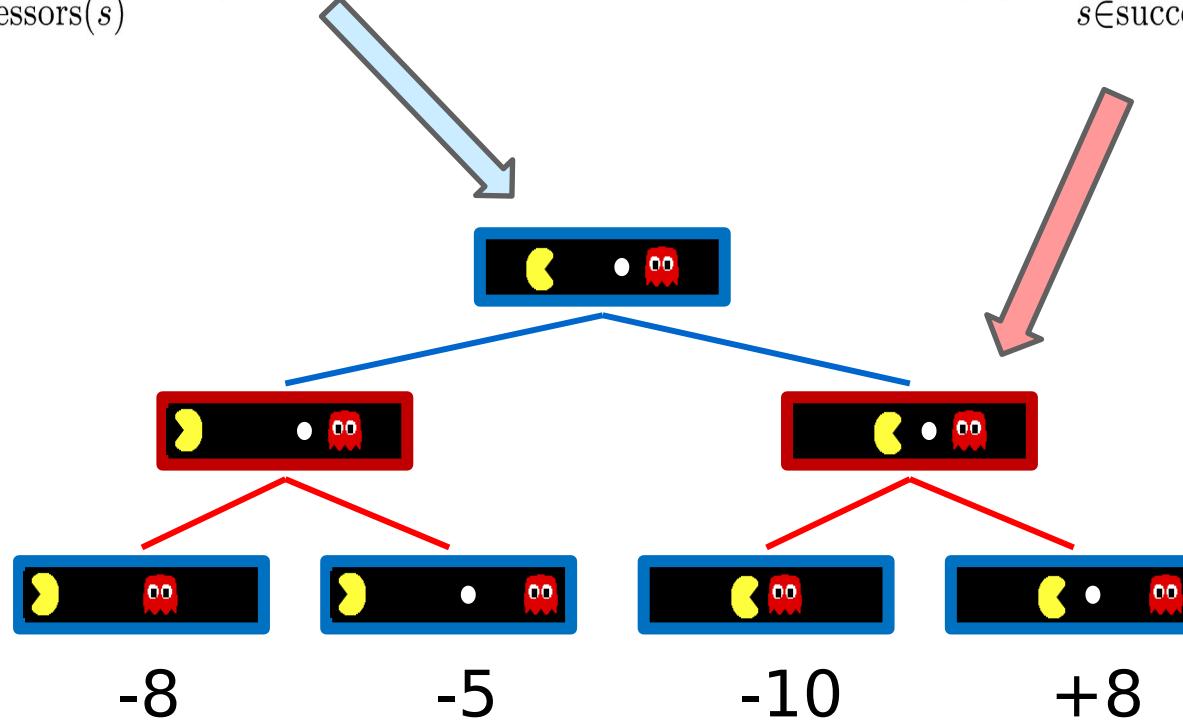
# Valores Minimax

Estados bajo el control del agente:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

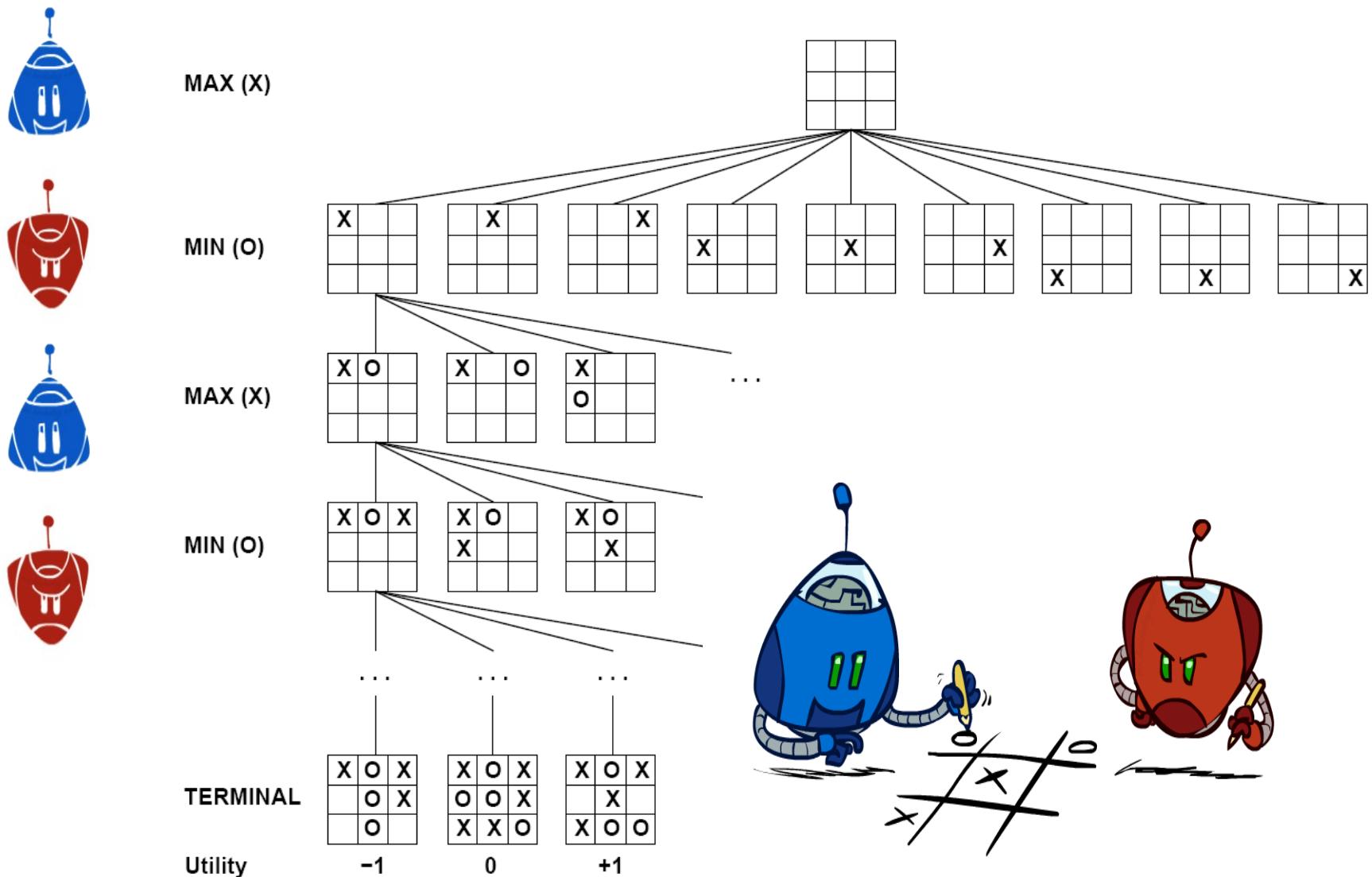
Estados bajo el control del oponente:

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Estados terminales:  
 $V(s) = \text{conocido}$

# Árbol de juegos de Tic-Tac-Toe



# Búsqueda Adversarial (Minimax)

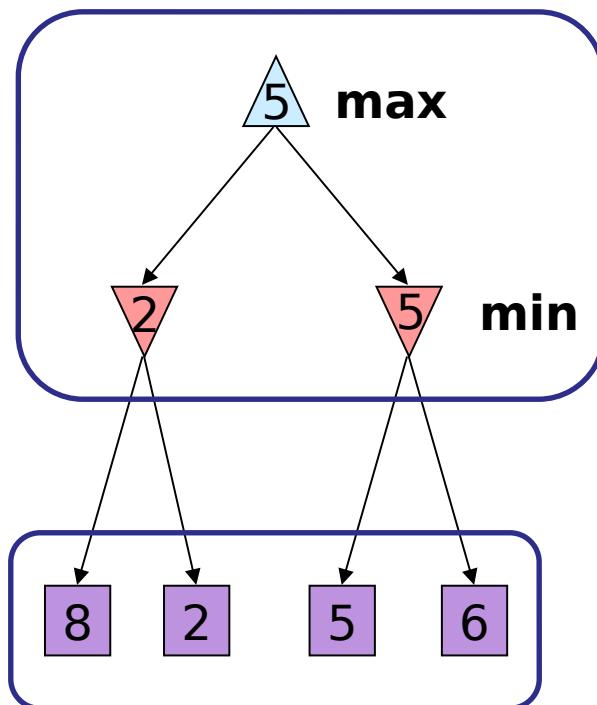
## ➤ Juegos determinísticos, de suma cero:

- Tic-tac-toe, ajedrez, damas
- Un jugador maximiza el resultado
- El otro minimiza el resultado

## ➤ Búsqueda Minimax:

- Árbol de búsqueda en un espacio de estados
- Los jugadores alternan turnos
- Se calcula el **valor minimax** de cada nodo: el máximo resultado (utility) posible contra un adversario racional (óptimo)

**Valores Minimax:**  
calculados **recursivamente**



**Valores terminales:**  
parte del juego

# Implementación de Minimax

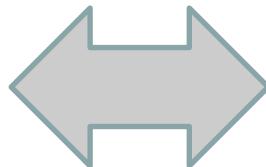
```
def max-value(state):
```

    initialize v = -∞

    for each successor of state:

        v = max(v, min-value(successor))

    return v



```
def min-value(state):
```

    initialize v = +∞

    for each successor of state:

        v = min(v, max-value(successor))

    return v

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

El valor v empieza con  
-∞ y va aumentando

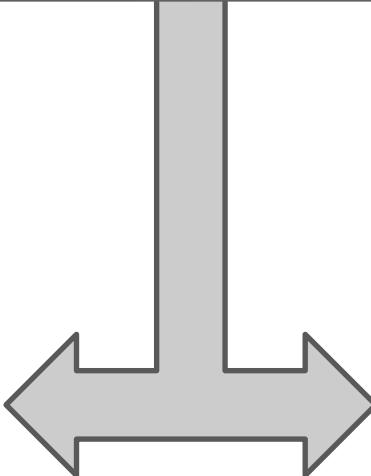
El valor v empieza con  
+∞ y va disminuyendo

# Implementación de Minimax (Dispatch)

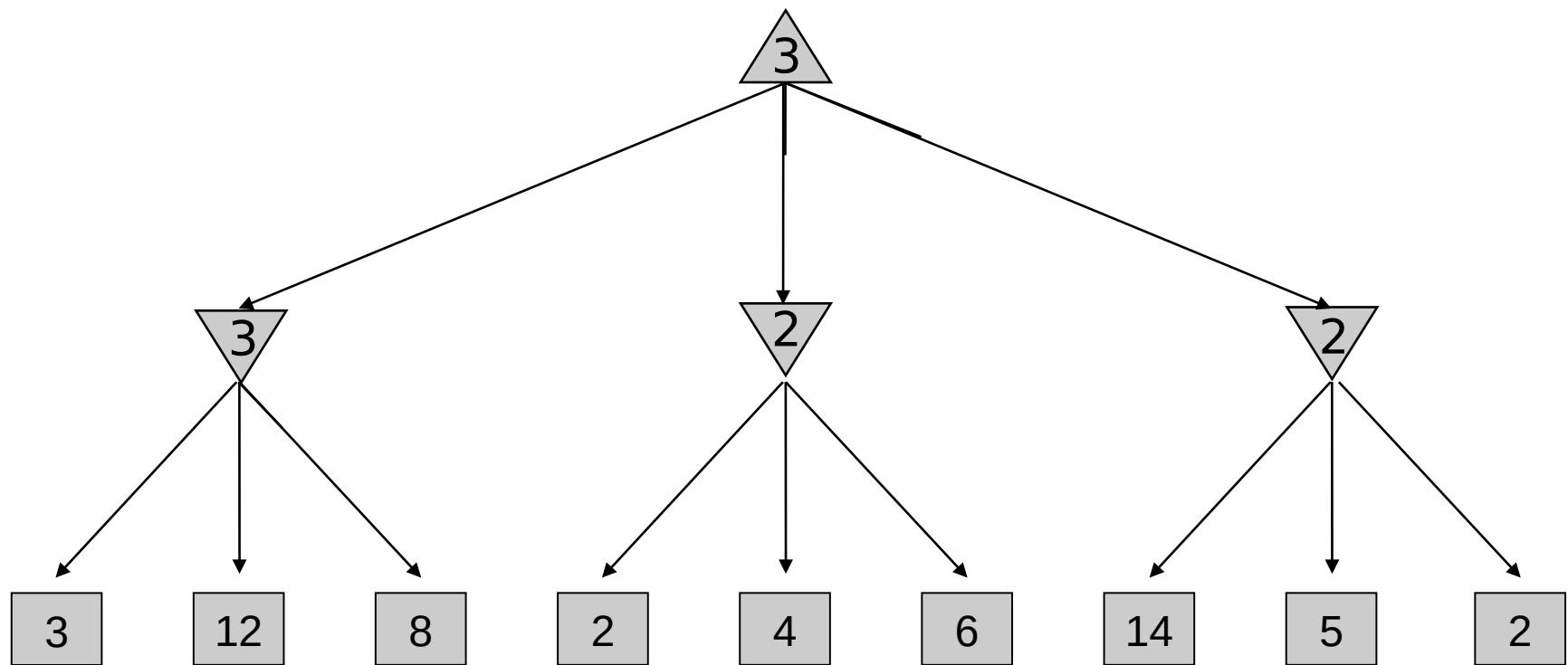
```
def value(state):  
    if the state is a terminal state: return the state's utility  
    if the next agent is MAX: return max-value(state)  
    if the next agent is MIN: return min-value(state)
```

```
def max-value(state):  
    initialize v = -∞  
    for each successor of state:  
        v = max(v,  
                 value(successor))  
    return v
```

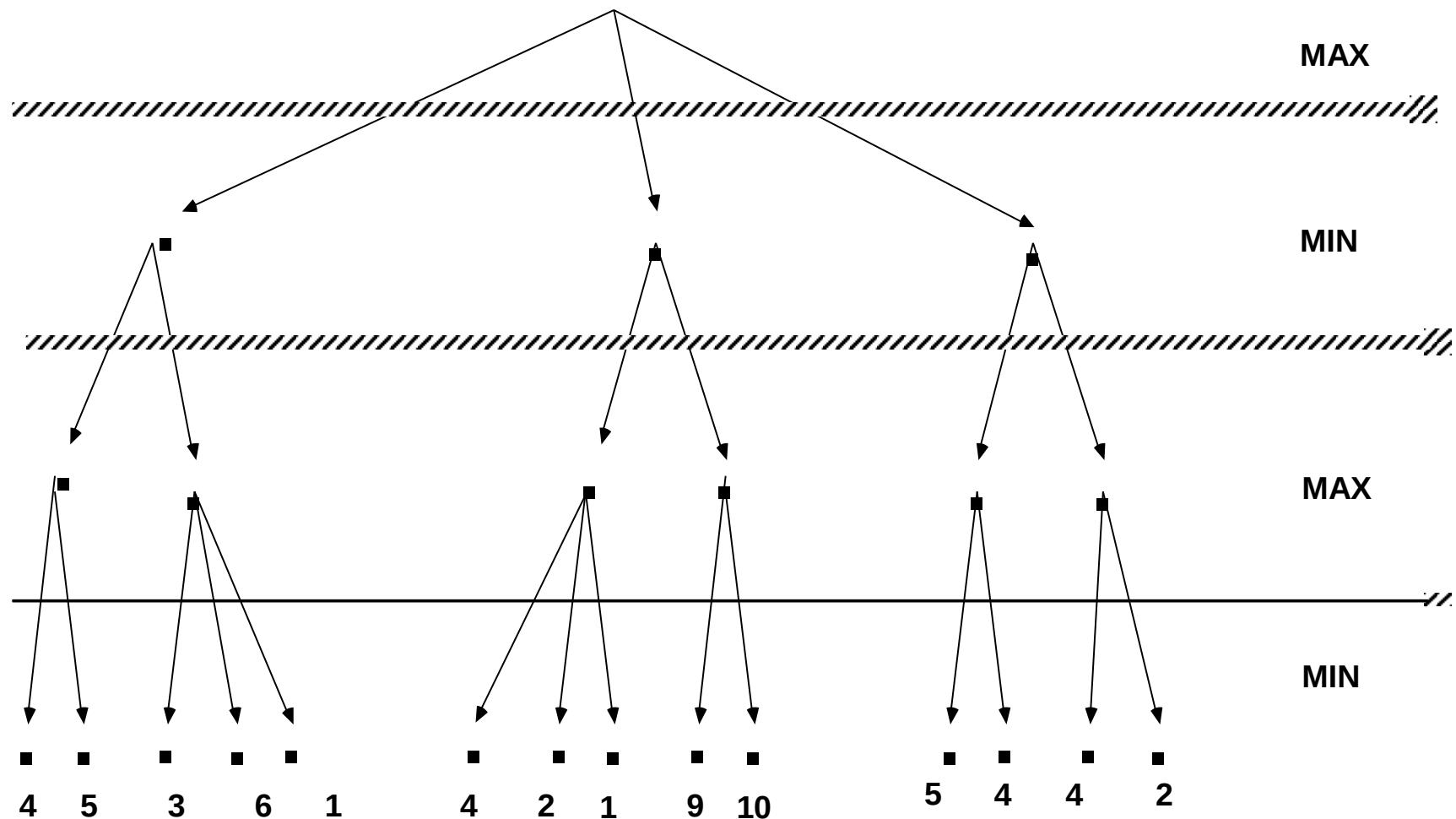
```
def min-value(state):  
    initialize v = +∞  
    for each successor of state:  
        v = min(v,  
                 value(successor))  
    return v
```



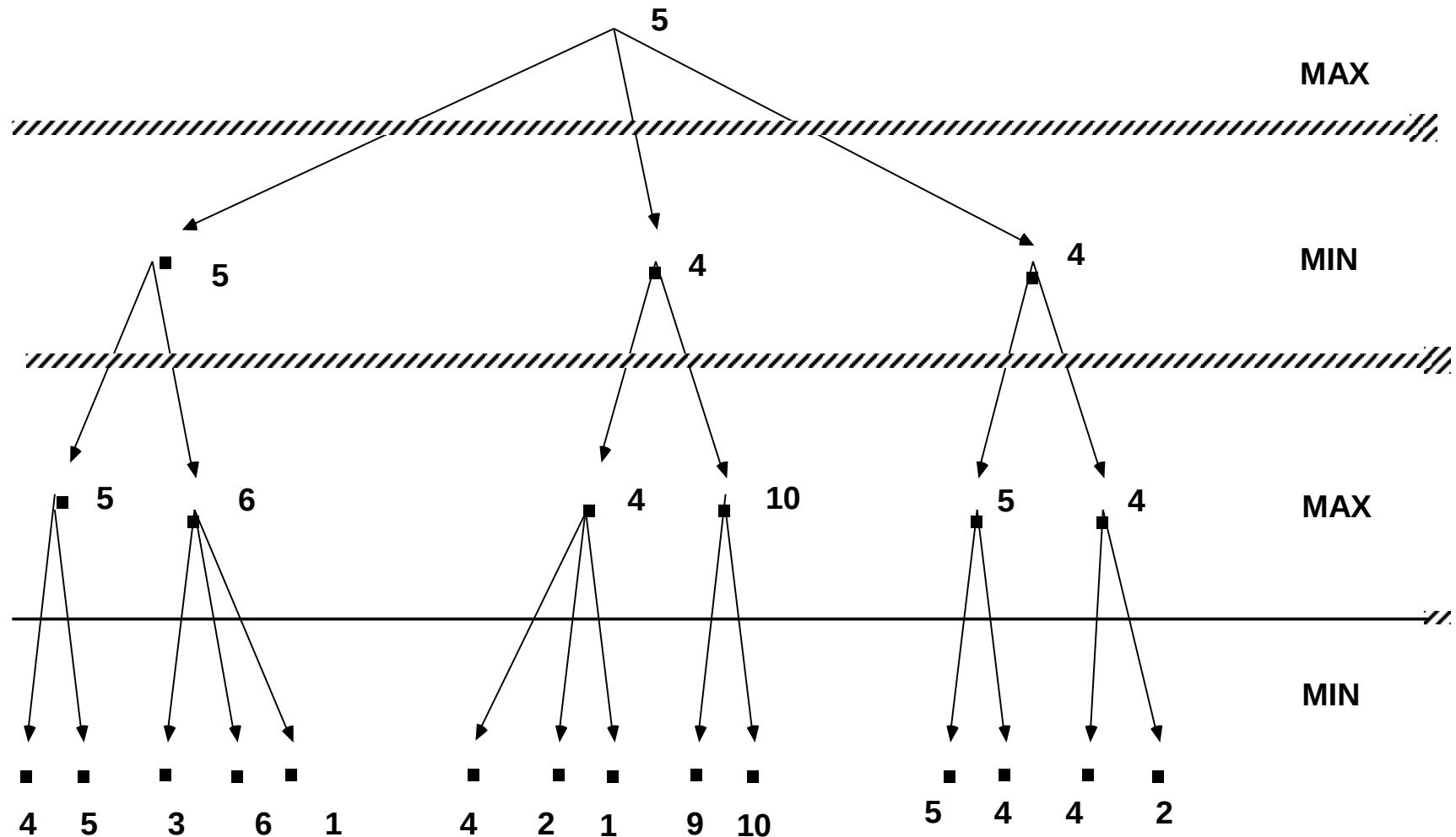
# Ejemplo de Minimax



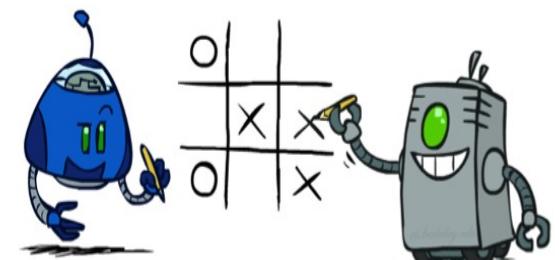
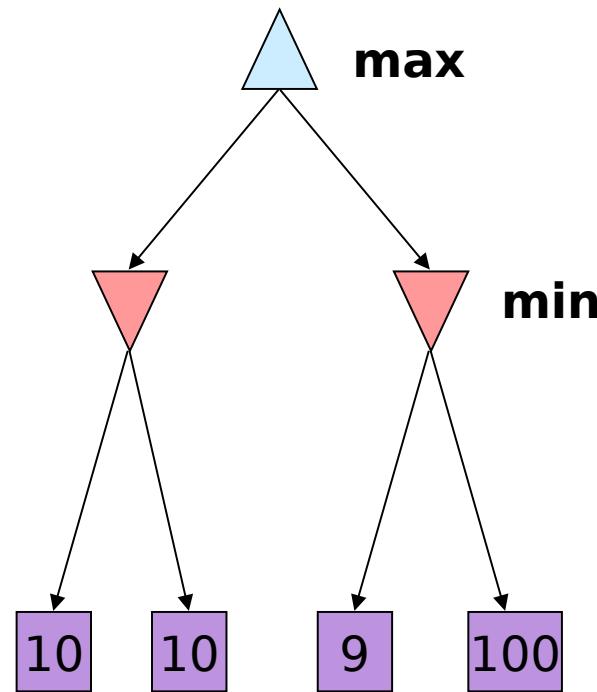
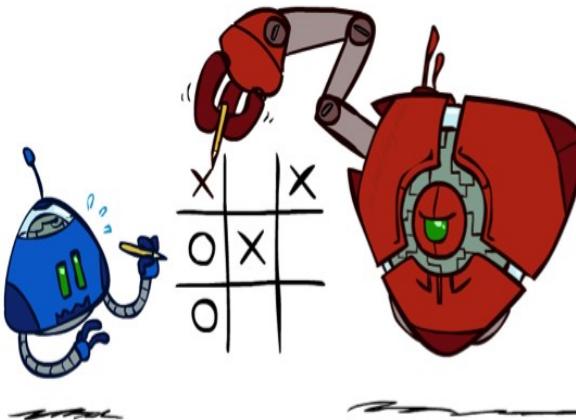
# Árbol de una aplicación del mini-max



# Árbol de una aplicación del mini-max



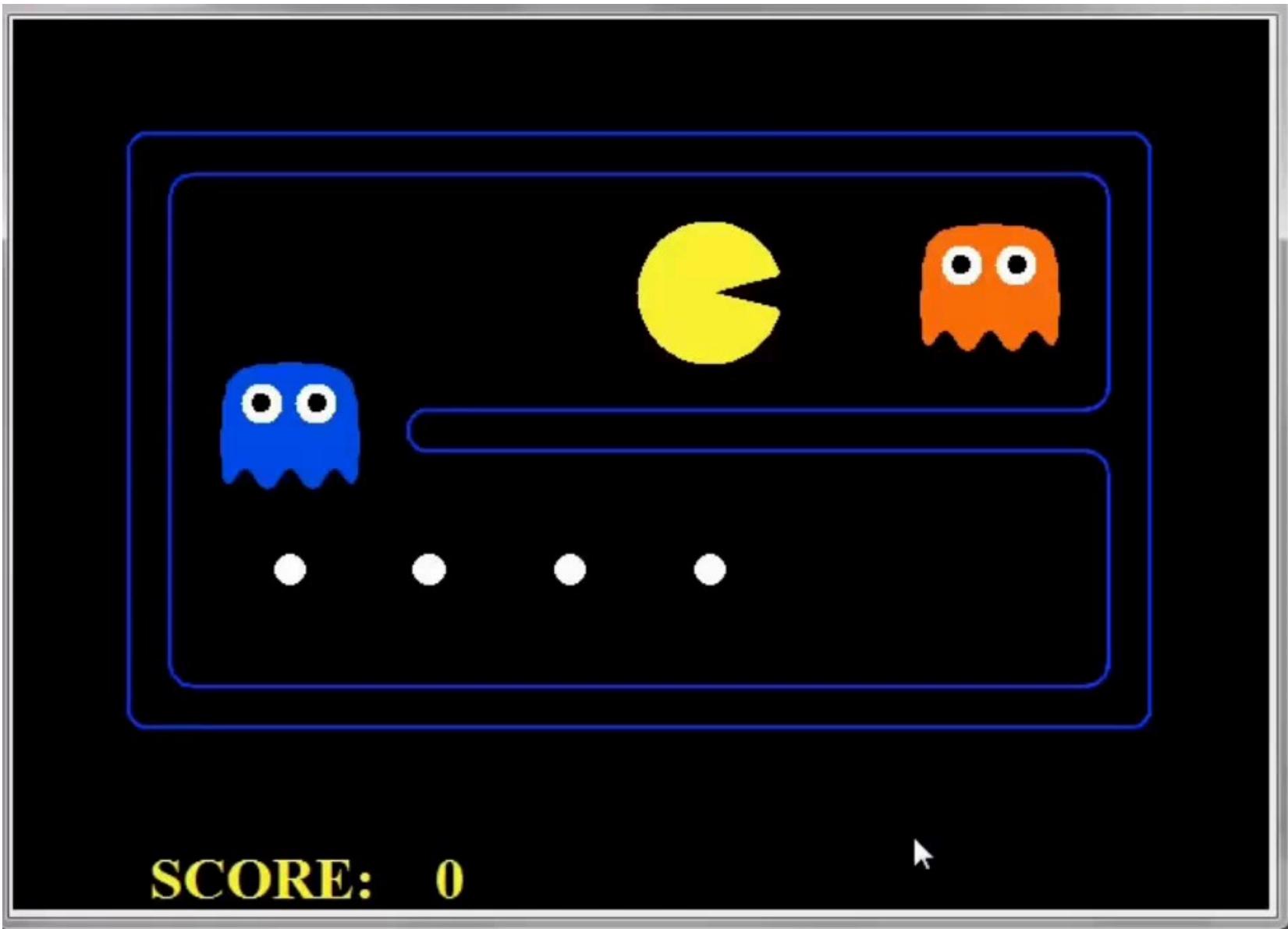
# Propiedades de Minimax



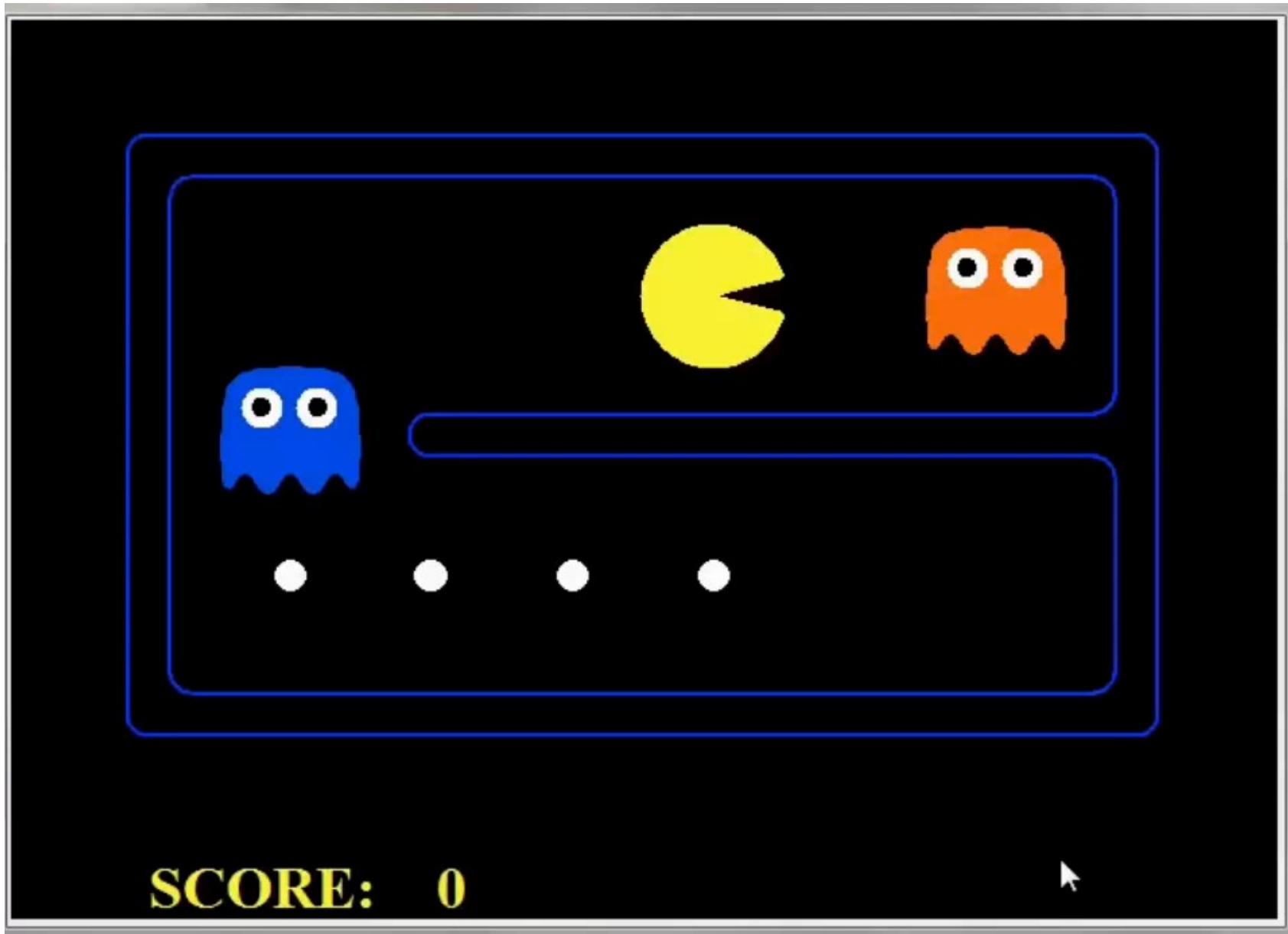
Óptimo contra un jugador perfecto. ¿En otro caso?

[Demo: min vs exp (L6D2, L6D3)]

# Video Demo Minimax

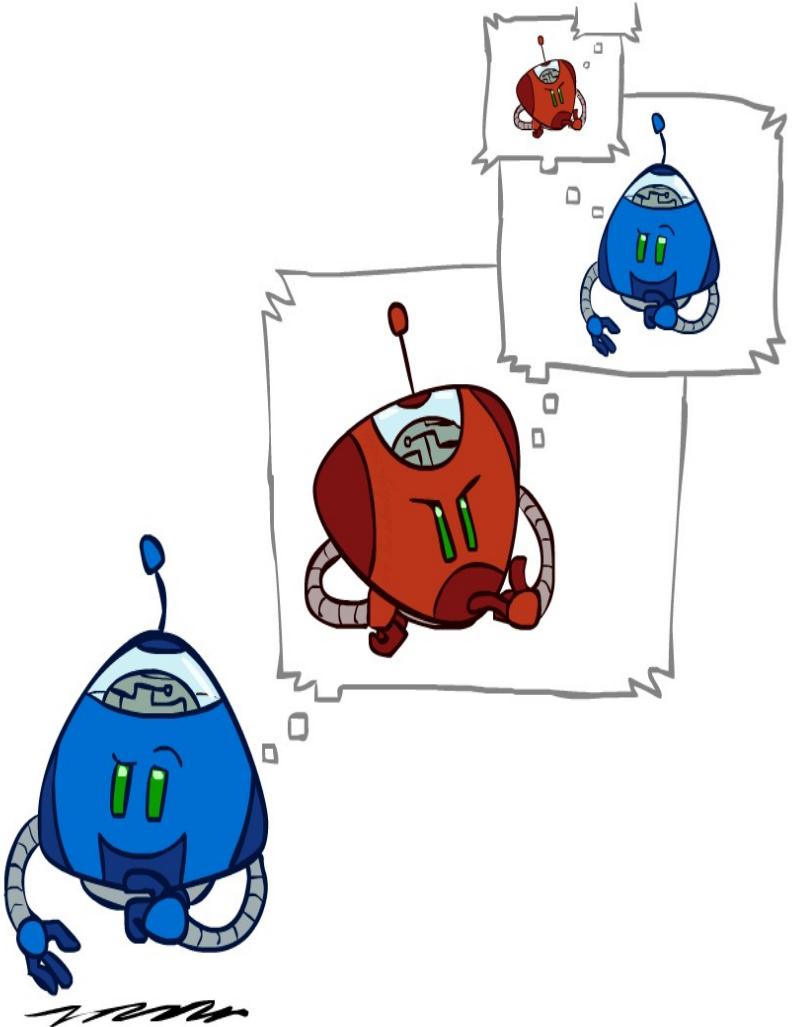


# Video Demo Minimax vs. Exp (Exp)



# Eficiencia de Minimax

- ¿Cómo de eficiente es minimax?
  - Igual que (exhaustivo) DFS
  - Tiempo:  $O(b^m)$
  - Espacio:  $O(bm)$
- Ejemplo: para ajedrez,  $b \sim 35$ ,  $m \sim 100$ 
  - Una solución exacta es inviable
  - Pero, ¿Tenemos que explorar el árbol entero?

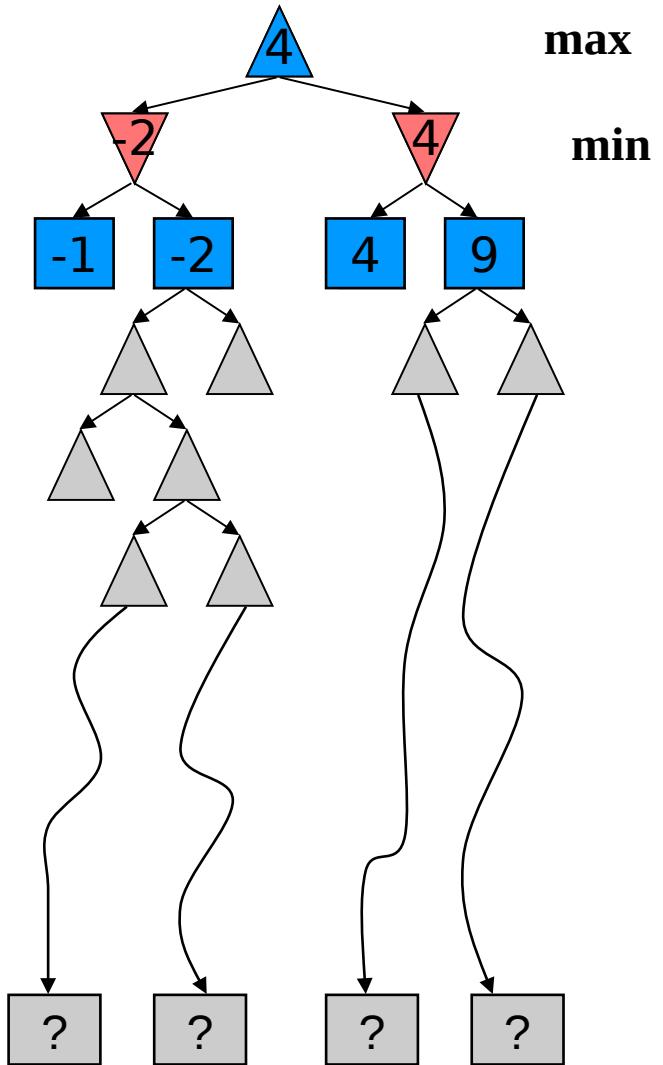


# Recursos limitados

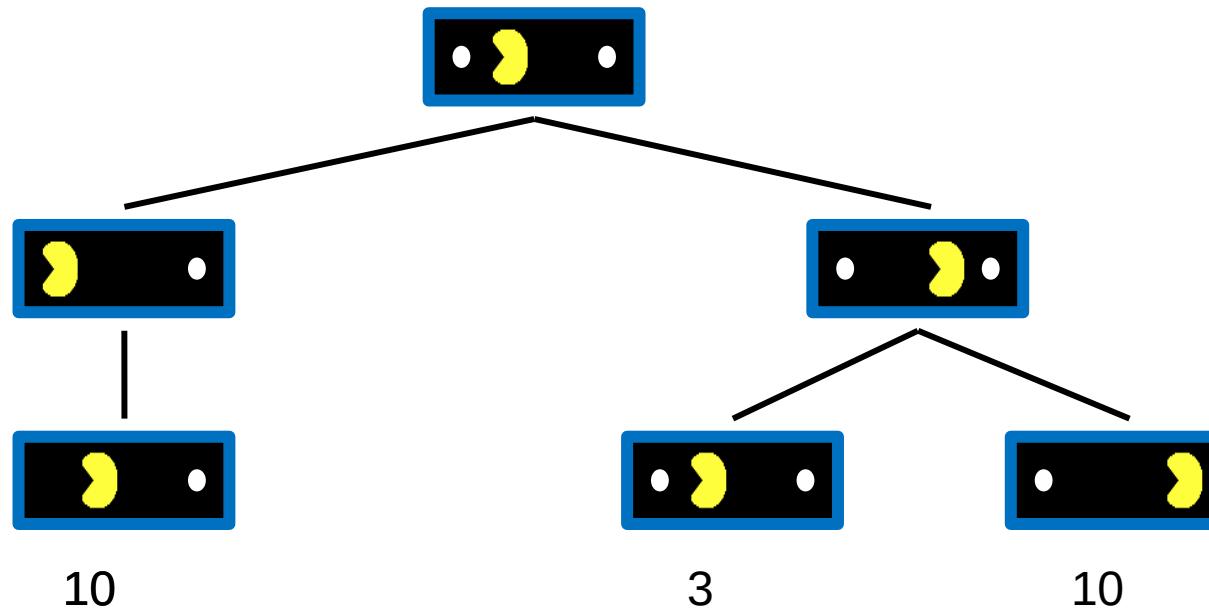


# Recursos limitados

- Problema: en juegos reales, ¡no podemos buscar hasta las hojas!
- Solución: búsqueda limitada en profundidad (Depth-limited search)
  - Buscar solo una profundidad limitada del árbol
  - Reemplazar las utilidades terminales con una función de evaluación para posiciones no terminales
- La garantía de juego óptimo se desvanece



# ¿Por qué Pacman no sabe qué hacer?



- Pacman sabe que su puntuación aumentará yendo en las 2 direcciones comiendo un punto. P. ej.: función de evaluación: 10 por cada punto comido
- Pero Pacman sabe que su puntuación también subirá si hace lo contrario
- Después de comer el punto, no hay oportunidad de anotar más puntos (en el horizonte, en este caso, 2 jugadas (profundidad))
- Por ello, la espera es tan buena como el comer: puede ir east, luego west en la siguiente ronda de planificación

# Funciones de Evaluación



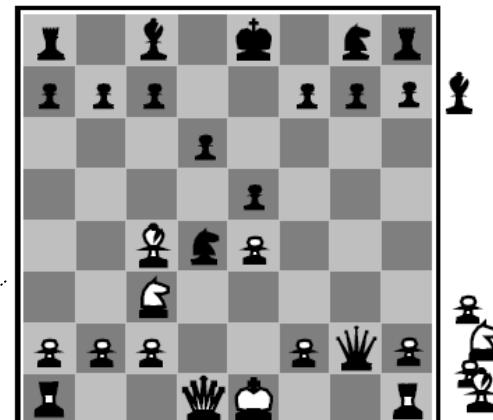
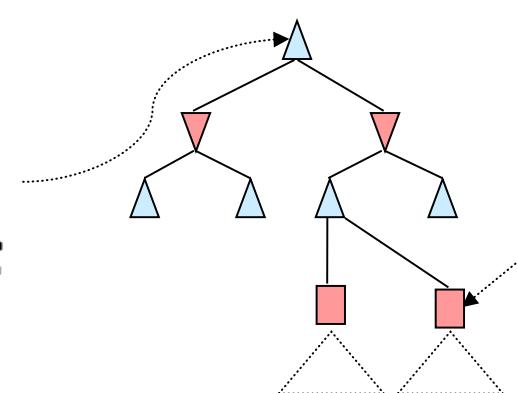
# Funciones de Evaluación

- Las funciones de evaluación puntuán no terminales, con búsqueda de profundidad limitada



Black to move

White slightly better

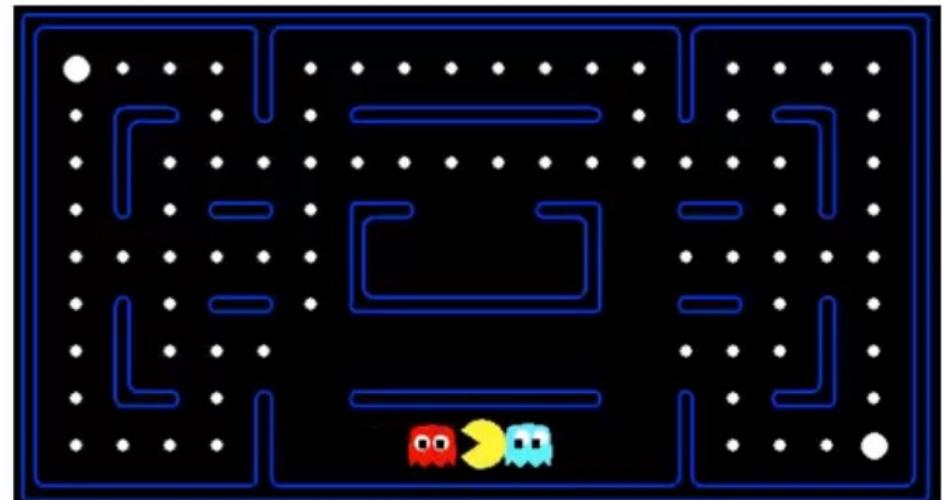
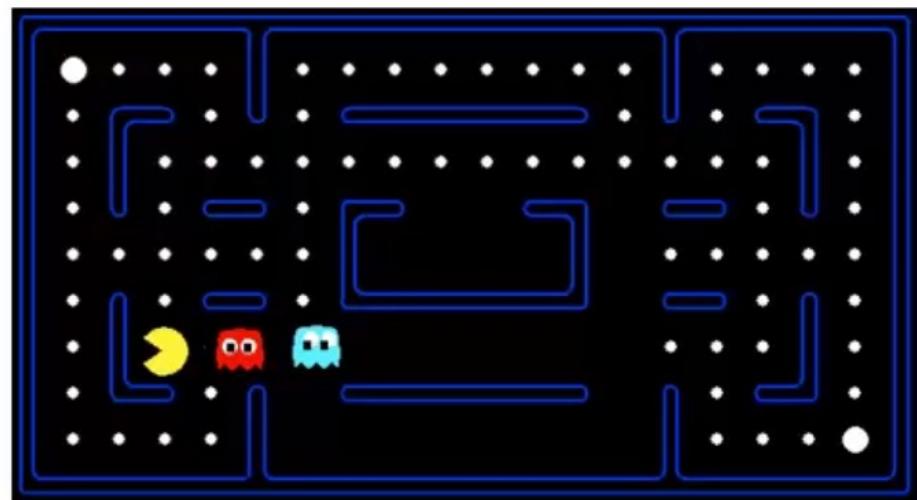
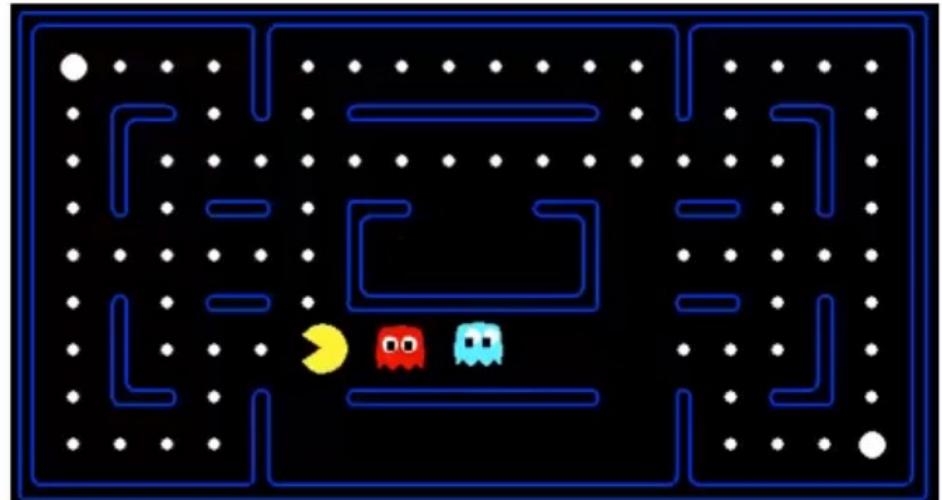


White to move

Black winning

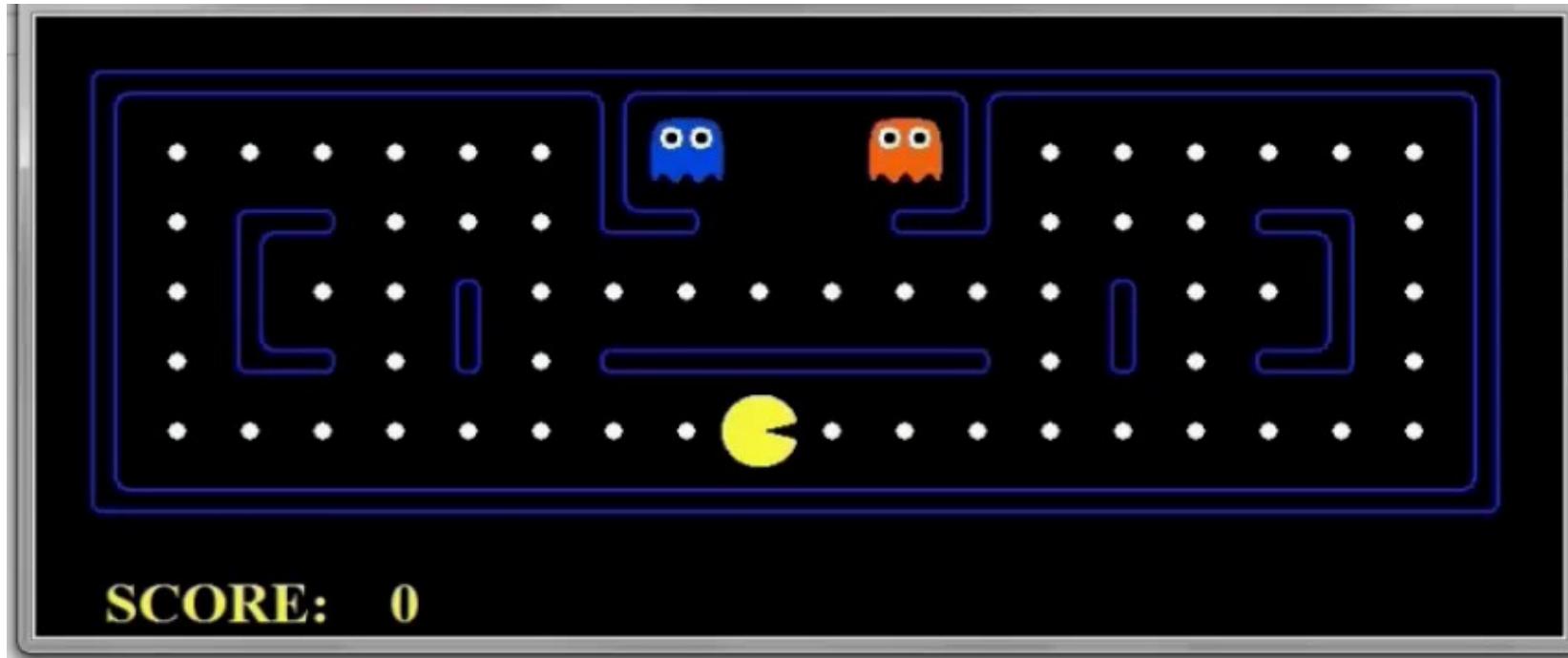
- Función Ideal: devuelve el valor real de minimax en esa posición
- En la práctica: normalmente suma lineal ponderada de características (features)
- $$Eval(s) = w_1 * f_1(s) + w_2 * f_2(s) + \dots + w_n * f_n(s)$$
- Por ejemplo:
  - $f_1(s) = (\text{cantidad de reinas blancas} - \text{cantidad reinas negras})$ , etc.

# Evaluación para Pacman

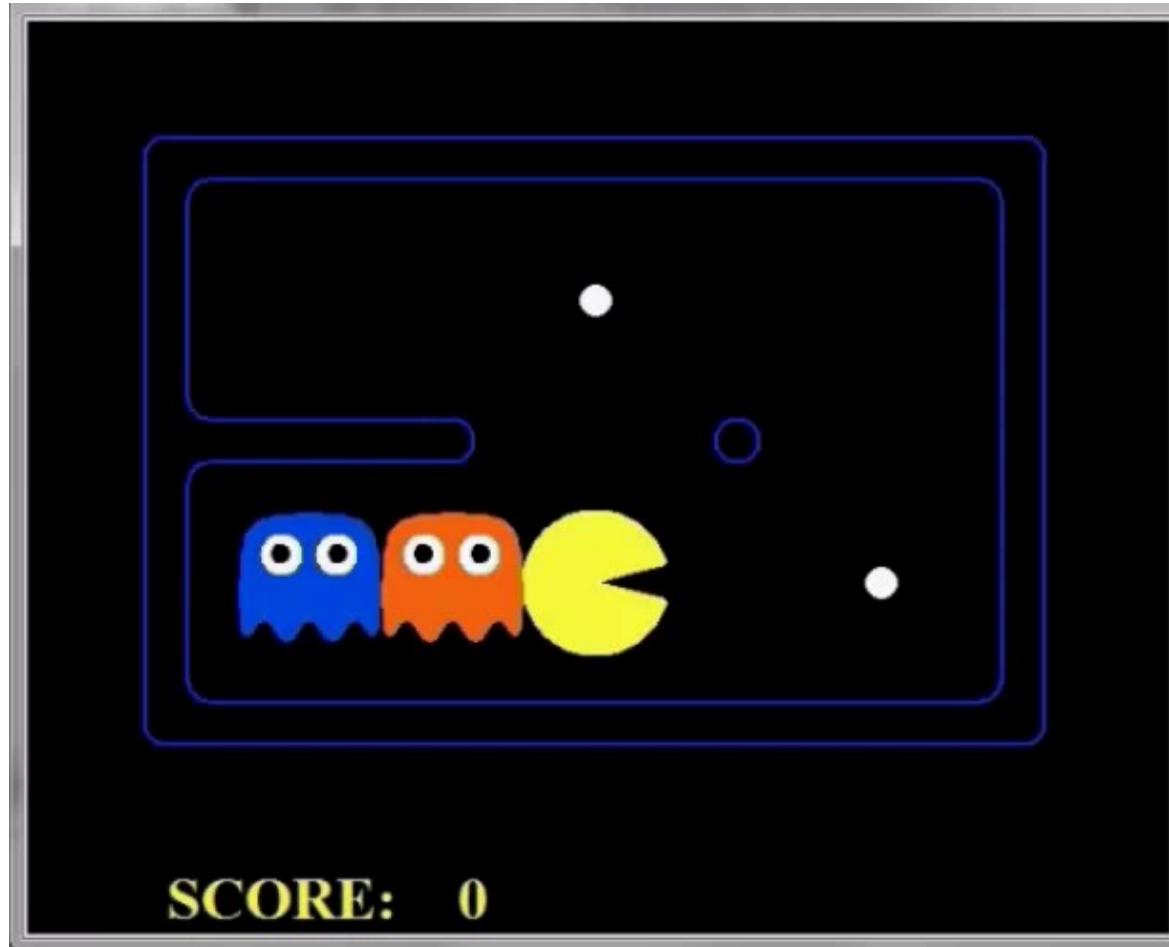


[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function), smart ghosts coordinate (L6D6,7,8,10)]

# Vídeo Demo Fantasmas inteligentes (Coordinación)



# Vídeo Demo Fantasmas inteligentes (Coordinación) – con Zoom



# La profundidad importa

- Las funciones de evaluación son siempre imperfectas
- Cuanto más profundamente en el árbol probemos la función de evaluación, es menos importante la calidad de la función de evaluación
- Un ejemplo interesante de compensación (tradeoff) entre complejidad de la función de evaluación y complejidad de cálculo



[Demo: depth limited (L6D4, L6D5)]

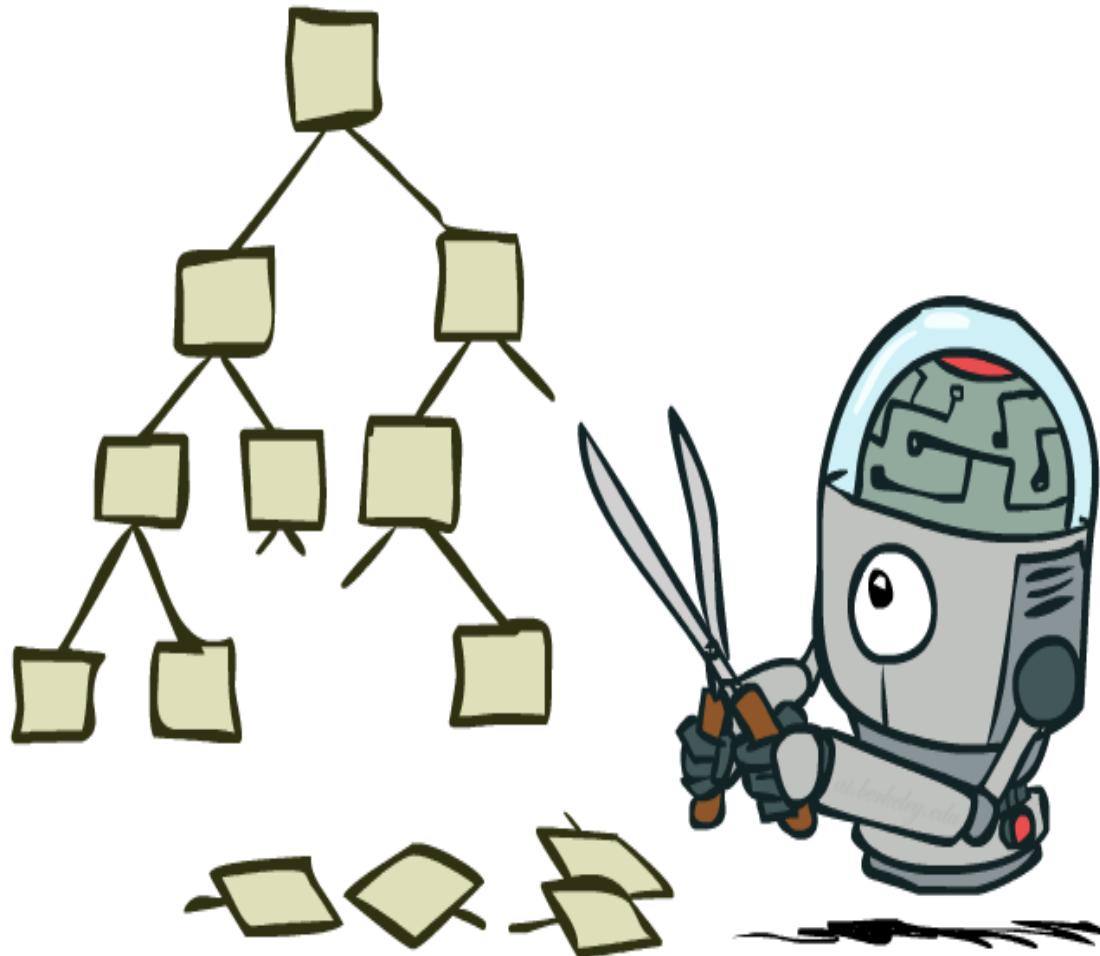
# Vídeo Demo Profundidad limitada (2)



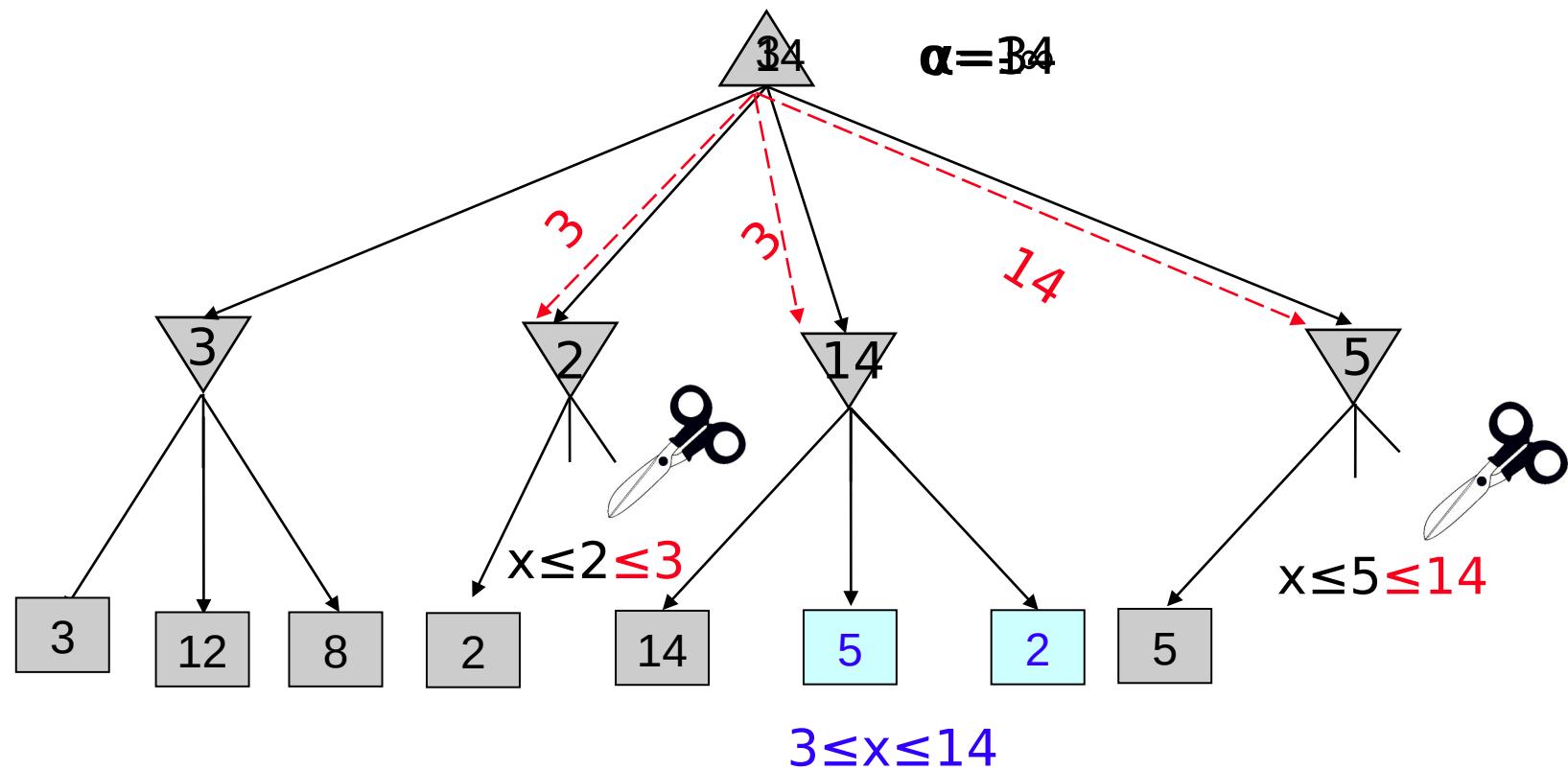
# Vídeo Demo Profundidad limitada (10)



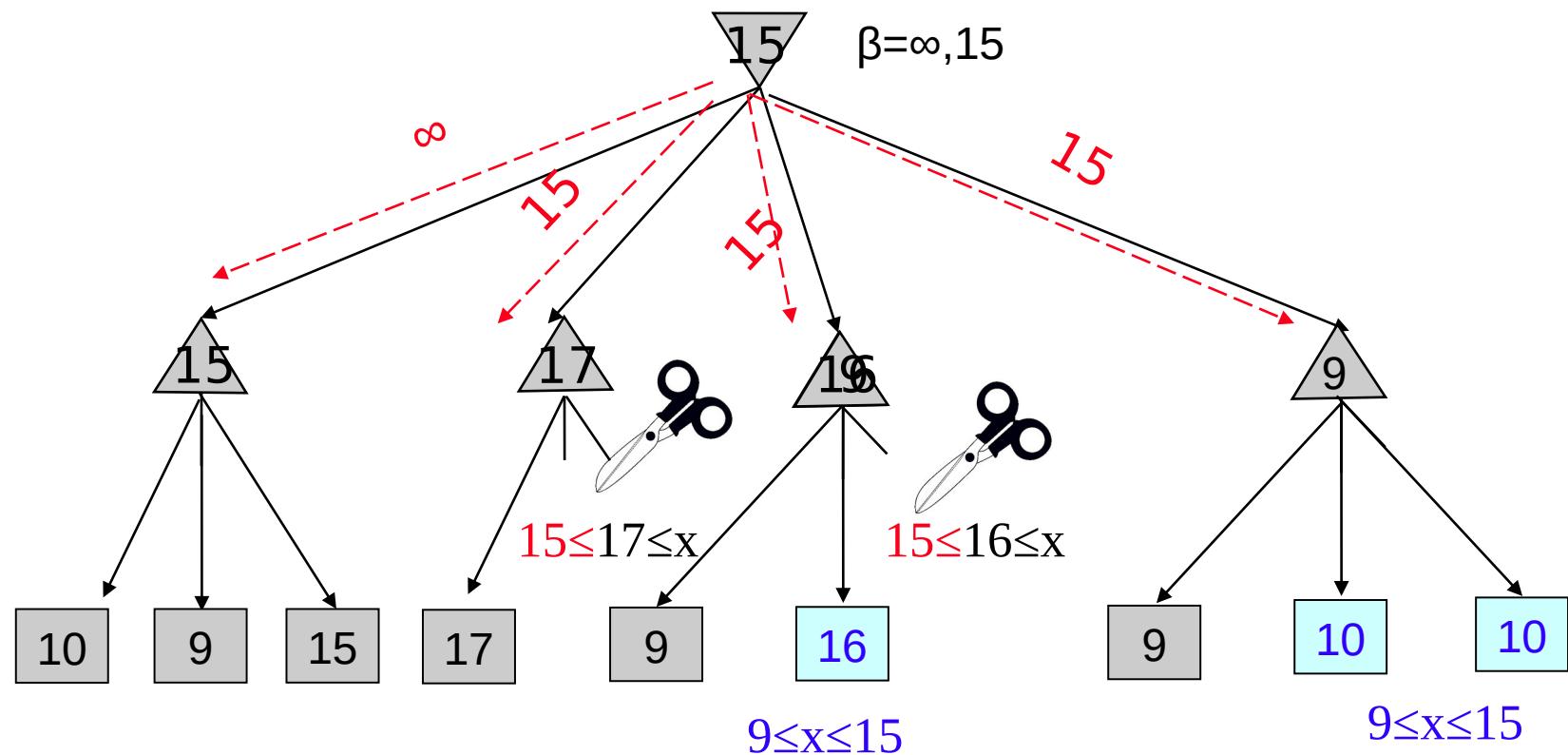
# Podado del árbol de juego



# Podado de Minimax

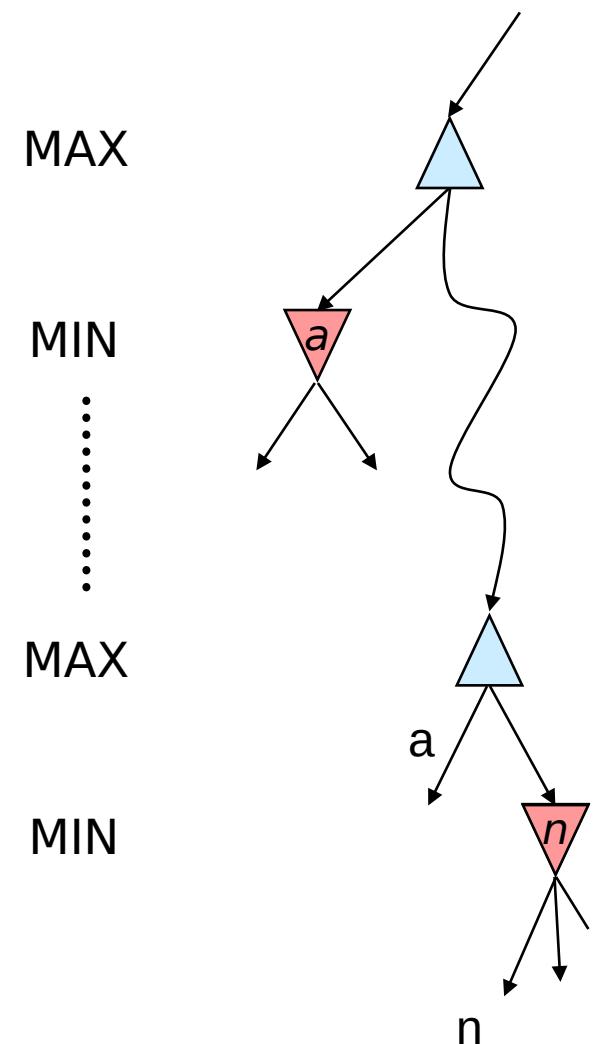


# Podado de Minimax



# Podado Alpha-Beta

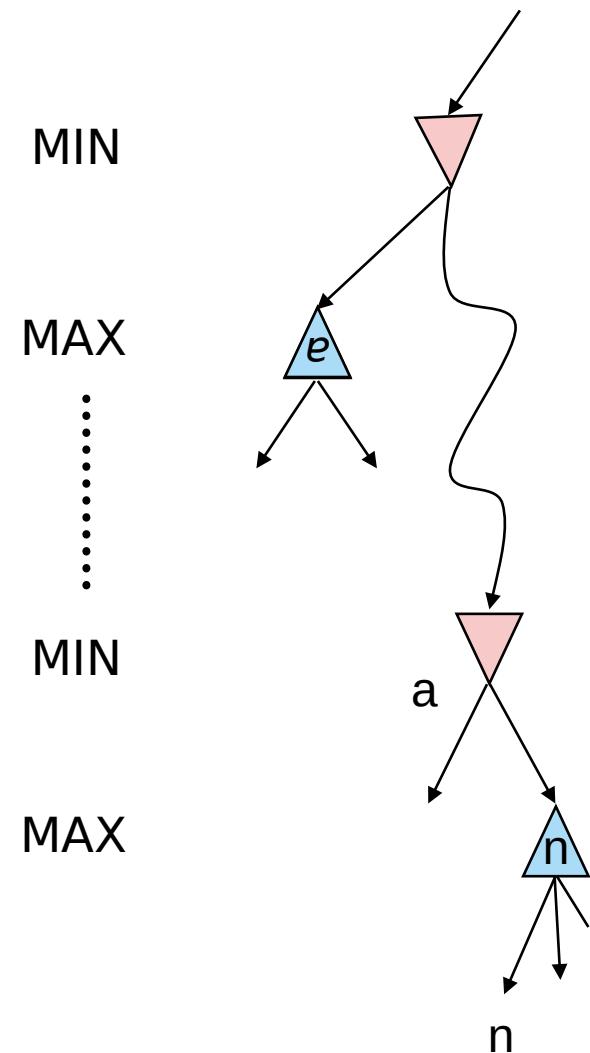
- Configuración General (versión MAX)
  - Estamos calculando un valor-MIN en un nodo **n**
  - Estamos iterando sobre los hijos de **n**
  - Al estar minimizando, el valor de **n** irá reduciéndose
  - ¿Dónde se usa el valor de **n**? En el MAX superior
  - Sea **a** el mejor valor que MAX puede obtener a lo largo del camino actual desde la raíz
  - Si **n** es menor que **a**, MAX lo evitará, por ello podemos evitar el considerar los demás hijos de **n** (es suficientemente malo para saber que no se hará esa jugada).
- La versión MIN es simétrica



# Podado Alpha-Beta

- Configuración General (versión MIN)
  - Estamos calculando un valor-MAX en un nodo **n**
  - Estamos iterando sobre los hijos de **n**
  - Al estar maximizando, el valor de **n** irá aumentando  
¿Dónde se usa el valor de **n**? En el MIN superior
  - Sea **a** el menor valor que **MIN** puede obtener a lo largo del camino actual desde la raíz

Si **n** es mayor que **a**, **MIN** lo evitará seleccionando **a**, por ello podemos evitar el considerar los demás hijos de **n** (es suficientemente alto para saber que no se hará esa jugada).



# Implementación de Alfa-Beta

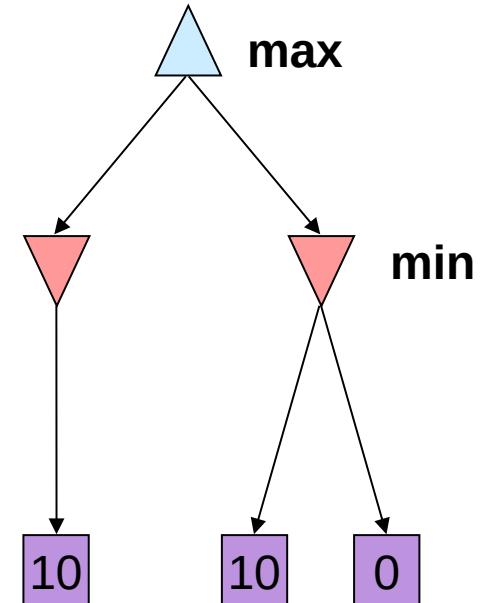
$\alpha$ : la mejor opción de MAX's en el camino a la raíz  
 $\beta$ : la mejor opción de MIN's en el camino a la raíz

```
def max-value(state, α, β):  
    initialize v = -∞  
    for each successor of state:  
        v = max(v, value(successor, α, β))  
        if v ≥ β return v  
        α = max(α, v)  
    return v
```

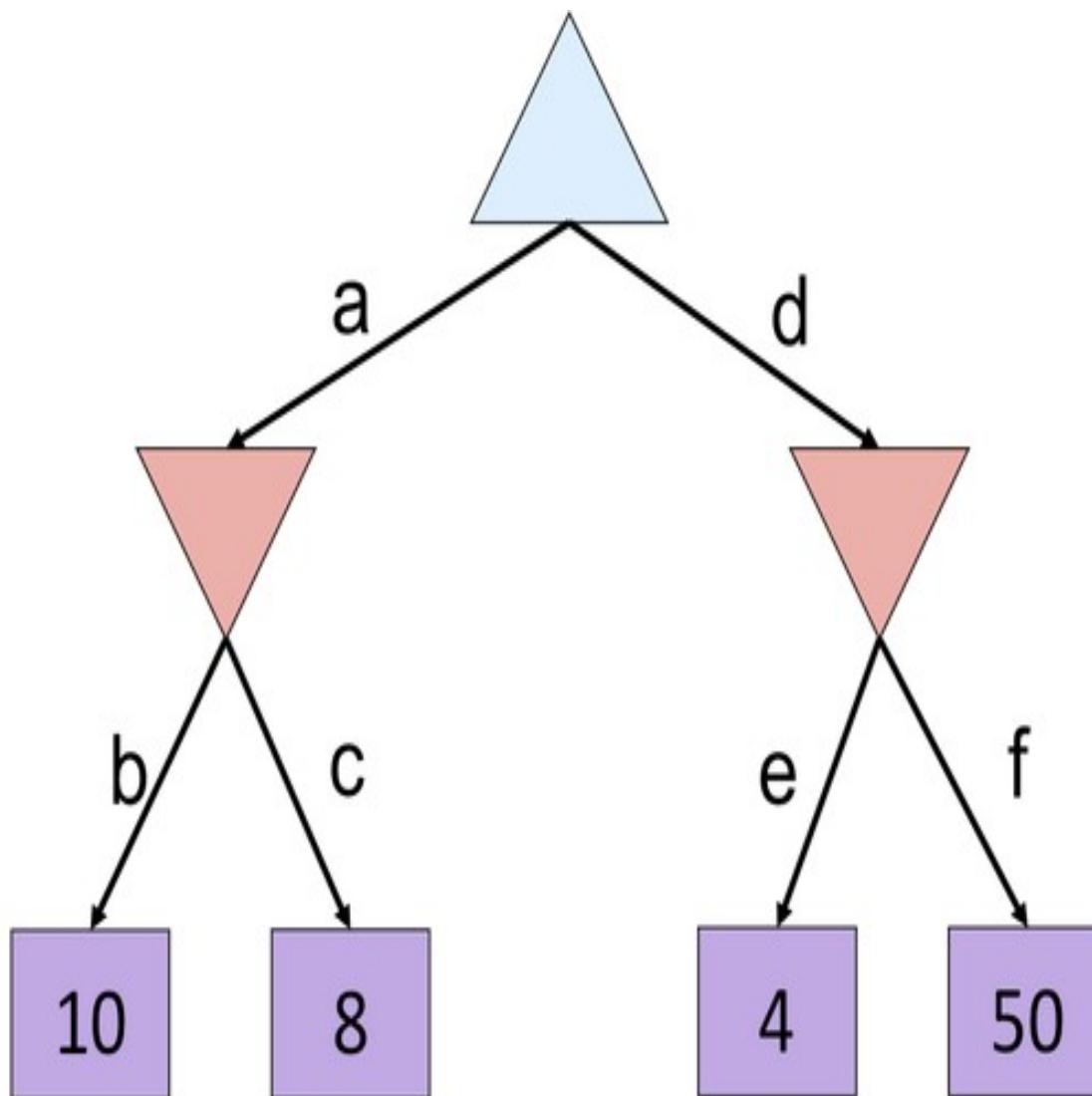
```
def min-value(state , α, β):  
    initialize v = +∞  
    for each successor of state:  
        v = min(v, value(successor, α, β))  
        if v ≤ α return v  
        β = min(β, v)  
    return v
```

# Propiedades de podado Alfa-Beta

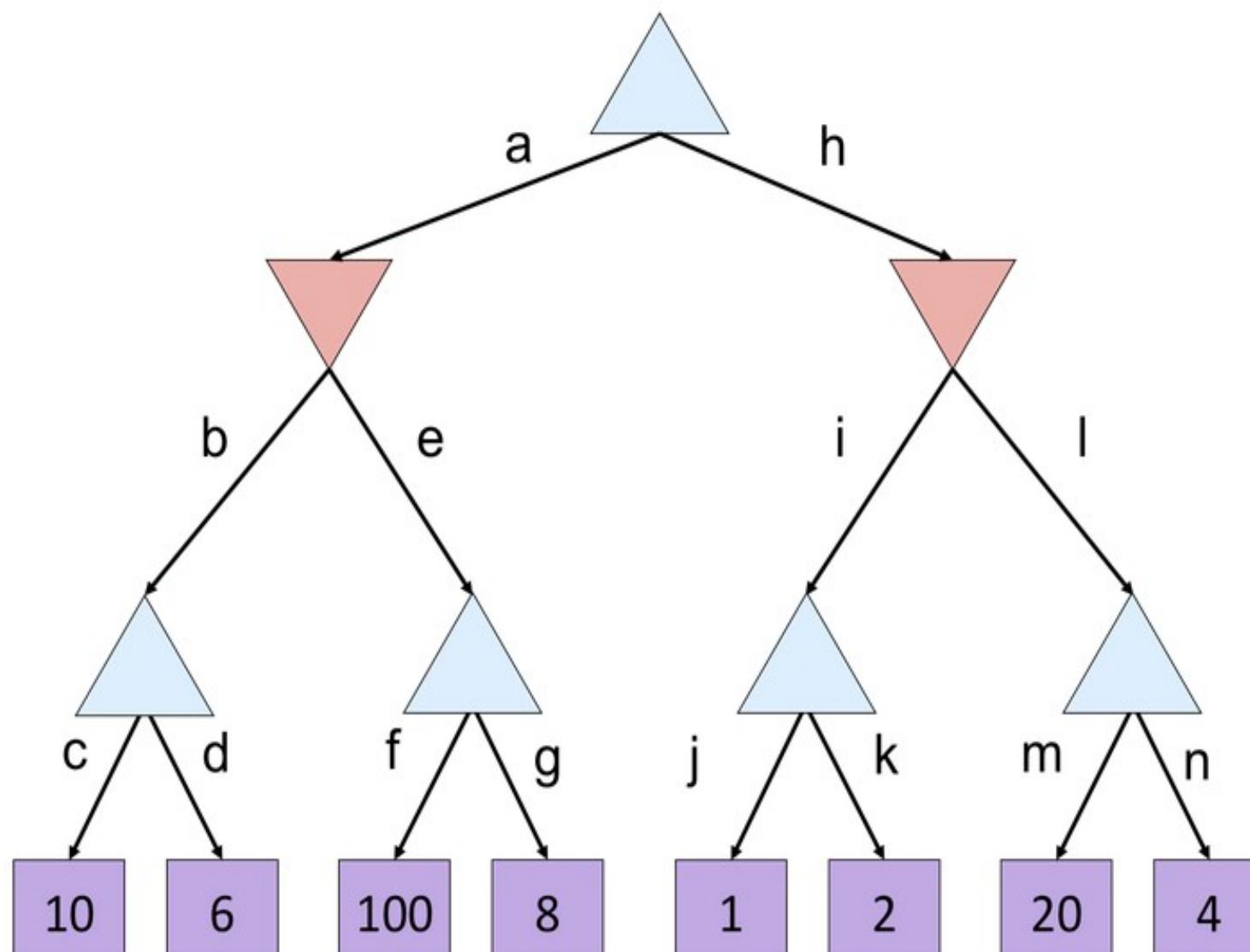
- ¡El podado no tiene efecto sobre el valor minimax calculado para la raíz!
- Los valores de nodos intermedios pueden ser incorrectos
  - Importante: los hijos de la raíz pueden tener un valor incorrecto
  - Por ello, una versión simple no permitirá elegir una acción (hijos de la raíz)
- Una buena ordenación de los hijos mejora la efectividad del podado
- Con una “ordenación perfecta”:
  - La complejidad en tiempo se reduce a  $O(b^{m/2})$
  - ¡Dobla la profundidad que se puede explorar!



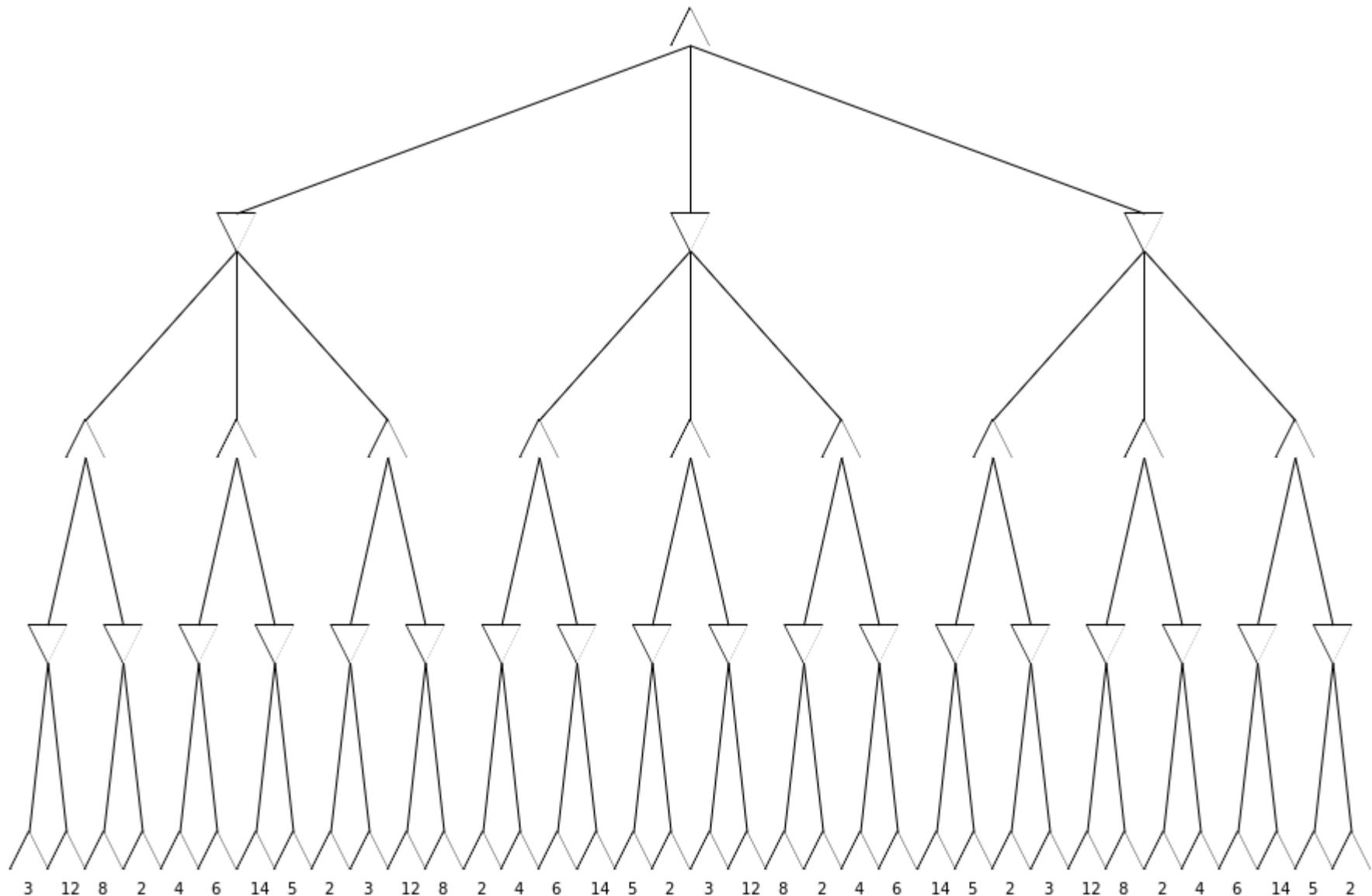
# Alpha-Beta Quiz



# Alpha-Beta Quiz 2



# Alpha-Beta Quiz 3



# Alpha-Beta Demo

- Demo: minimax game search algorithm with alpha-beta pruning (using html5, canvas, javascript, css)
  - <http://homepage.ufp.pt/jtorres/ensino/ia/alfabeta.html>