Universidad Andrés Bello
Conectar · Innovar · Liderar

Introducción a la inteligencia artificial

🌐 Introducción a la Búsqueda Adversarial

¿Qué es?

- Hasta ahora, en algoritmos como A* el problema era **estático**: un agente busca el mejor camino en un mundo con obstáculos.
- En los **juegos**, el mundo es **dinámico**: hay otro jugador que también toma decisiones y trata de impedir tu objetivo.
- Entonces, no basta con buscar “el mejor camino”, sino que hay que buscar **la mejor estrategia frente a un oponente**.

Características de la búsqueda adversarial

1. **Dos jugadores (mínimo):** uno trata de maximizar su utilidad (MAX) y el otro trata de minimizar (MIN).

2. **Turnos alternados:** cada acción de un jugador cambia el estado del juego.

3. **Estados terminales:** representan situaciones de victoria, derrota o empate.

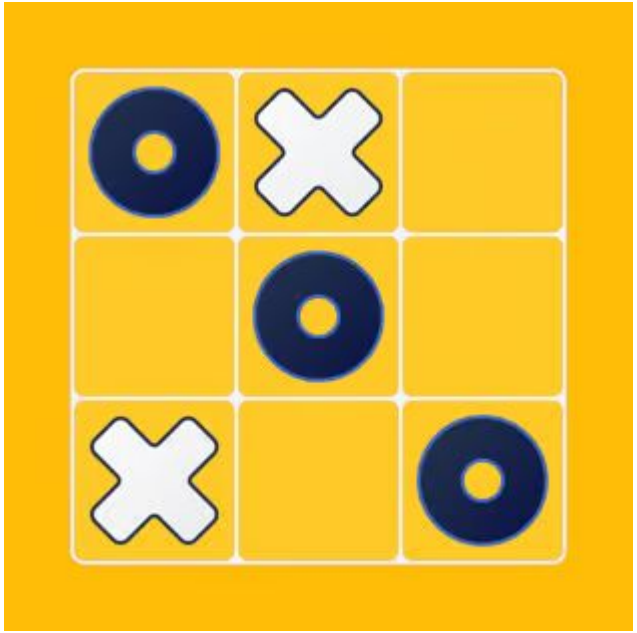
4. **Función de utilidad (payoff):** asigna un valor numérico a cada estado final (ej: +1 ganar, 0 empatar, −1 perder).

5. **Estrategia óptima:** se define como la que garantiza el mejor resultado posible asumiendo que el rival juega perfectamente.

Características de la búsqueda adversarial

🎲 Ejemplo: Tres en Raya (Tic-Tac-Toe)

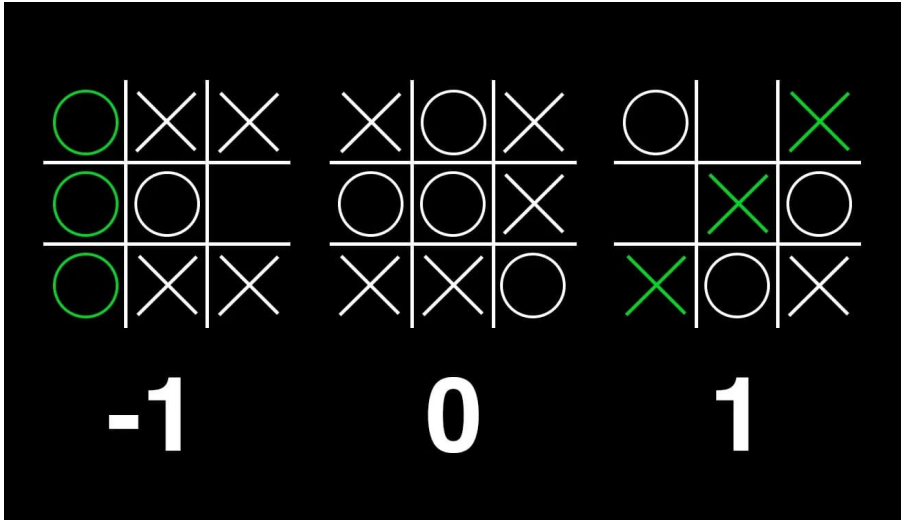
- Dos jugadores: **X** y **O**.
- En tu turno, colocas una ficha buscando ganar.
- El rival, en su turno, coloca una ficha para evitar que ganes y tratar de ganar él.
- Aquí se ve claro: tu estrategia depende de lo que el otro haga después.



Búsqueda Adversarial: Minimax

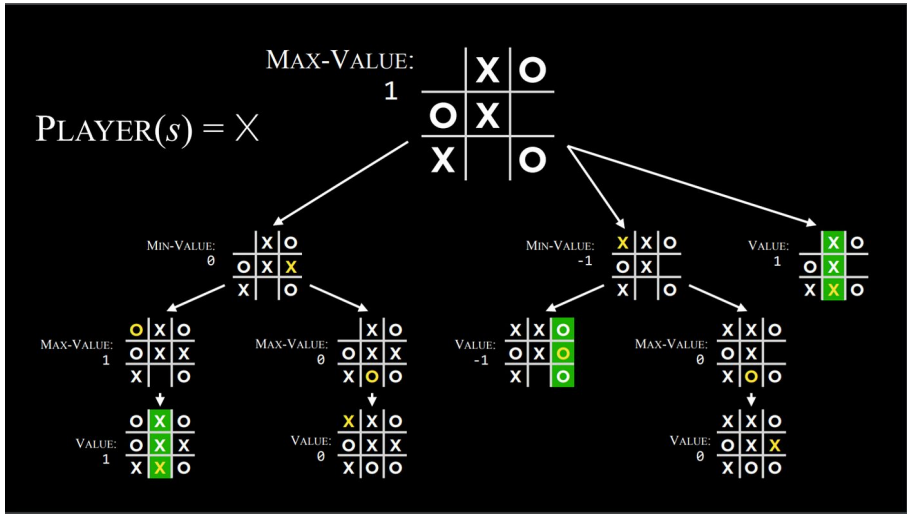
1. ¿Qué es?

- Es un algoritmo diseñado para “**juegos” de dos jugadores con suma cero** (lo que gana uno, lo pierde el otro).
- El objetivo es que el **jugador MAX** elija la jugada que le garantice el **mejor resultado posible**, incluso si el oponente (**jugador MIN**) juega perfectamente.



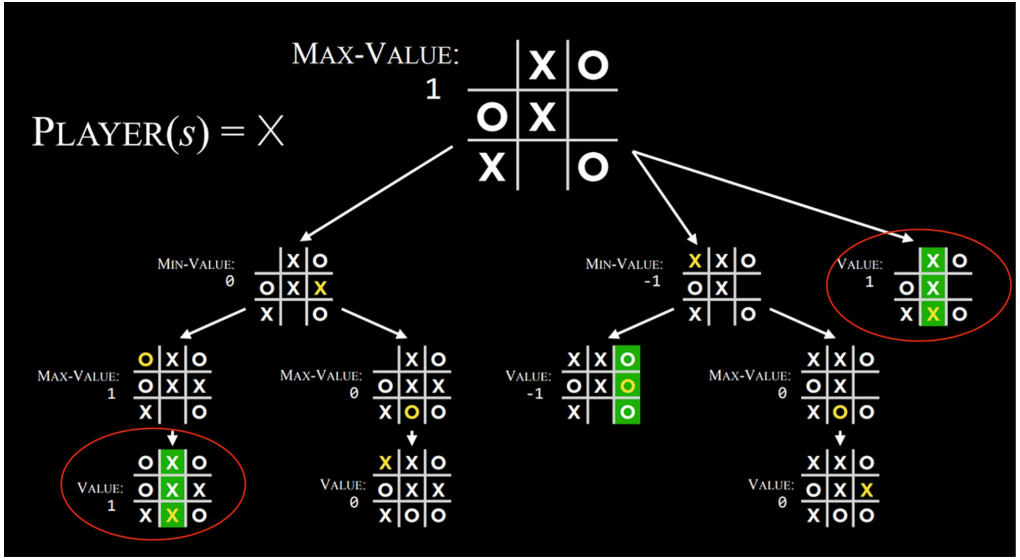
Idea central

- Construimos un **árbol de juego** con todos los estados posibles desde la situación actual hasta los finales.
- En los **nodos terminales** se asigna un valor (ej: +1 = ganar, 0 = empate, −1 = perder).
- Luego esos valores se “propagan hacia arriba”:
    - En los niveles de **MAX**, elegimos el valor **máximo** (la mejor opción para nosotros).
    - En los niveles de **MIN**, elegimos el valor **mínimo** (porque el rival siempre intentará perjudicarnos).



Intuición

- Minimax **no busca la jugada más rápida o la que "se ve bonita"**, sino la que **te da la mejor garantía contra un rival perfecto**.
- Es como pensar: *"¿Qué pasa si mi rival siempre toma la mejor contra-jugada posible?"*
- Así, la estrategia resultante no depende solo de tus deseos, sino de **anticipar la reacción del otro**.



Minimax

Explora todo el árbol de juego → exhaustivo.

Correcto y garantizado: encuentra la jugada óptima si el rival juega perfecto.

Costo computacional alto: si el juego tiene una gran cantidad de posibles jugadas, el árbol crece **exponencialmente** (problema del “crecimiento combinatorio”).

Ventajas y desventajas

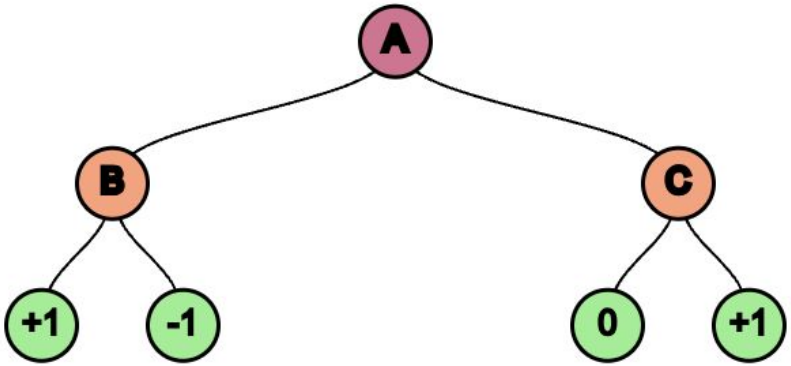
✅ Garantiza la jugada óptima (si se explora todo el árbol).
❌ El árbol de juego puede ser enorme → muy costoso de calcular.

*(y ahí entra la **poda alfa-beta**, que veremos después ✂️).*

Minimax - Ejemplo

Árbol de juego artificial

Imaginemos un árbol muy pequeño con valores terminales fijos:



Ya sabemos (por teoría) que la decisión óptima para MAX es ir por la rama **C**, porque garantiza al menos 0 (empate).

```python
# Definimos el árbol como un diccionario:
# Cada nodo tiene hijos o un valor si es terminal
tree = {
    "A": ["B", "C"],      # Nodo raíz, turno de MAX
    "B": ["B1", "B2"],    # Nodo de MIN
    "C": ["C1", "C2"],    # Nodo de MIN
    "B1": +1,             # Estados terminales
    "B2": -1,
    "C1": 0,
    "C2": +1
}

def minimax(node, maximizing):
    """Devuelve el valor minimax de un nodo"""
    # Si es terminal (int), retornamos el valor
    if isinstance(tree[node], int):
        return tree[node]

    # Si es nodo MAX
    if maximizing:
        best = float("-inf")
        for child in tree[node]:
            val = minimax(child, False)
            print(f"MAX evalúa {child} → {val}")
            best = max(best, val)
        return best

    # Si es nodo MIN
    else:
        best = float("inf")
        for child in tree[node]:
            val = minimax(child, True)
            print(f"MIN evalúa {child} → {val}")
            best = min(best, val)
        return best

# Probamos desde la raíz
resultado = minimax("A", True)
print("\nValor minimax desde A:", resultado)
```

codesnap.dev

Minimax - Ejemplo

Qué ocurre paso a paso

1. **MAX en A** evalúa a **B** y **C**.
2. En **B**, como es turno de MIN, toma `min(+1, -1) = -1`.
3. En **C**, MIN toma `min(0, +1) = 0`.
4. MAX en A finalmente elige `max(-1, 0) = 0`.

Tic Tac Toe con Minimax

🎯 Objetivo

- Modelar el **juego de tres en raya** como un problema de búsqueda adversarial.
- Implementar **minimax** para elegir la mejor jugada.
- Comparar con **minimax + poda alfa-beta**, midiendo **nodos expandidos**.

TicTacToe - Ejemplo

Modelamiento

- **Estados:** Tablero 3×3 con casillas vacías, X o O.
- **Acciones:** Colocar una ficha en una casilla vacía.
- **Jugadores:** MAX (X) y MIN (O).
- **Función de transición:** Colocar la ficha del turno en la casilla elegida → nuevo estado.
- **Terminales:** Alguien gana (tres en línea) o empate (tablero lleno).
- **Utilidad:**
    - X gana → +1
    - O gana → −1
    - Empate → 0

Con esto, Minimax se reduce a:
si es turno de **X** → tomar el **máximo** de los valores de los hijos;
si es turno de **O** → tomar el **mínimo**.

TicTacToe - Ejemplo

Universidad Andrés Bello
Conectar · Innovar · Liderar

`def empty_board()`

- Define una función llamada `empty_board`.
- Su tarea: devolver un tablero vacío de 3×3 (listas con espacios en blanco).

`-> Board`

- Es **anotación de tipo** en Python (type hint).
- Le dice al lector (y a herramientas de análisis estático como `mypy` o el autocompletado en editores) que esta función devuelve un objeto del tipo `Board`.