

# 5. BÚSQUEDA ENTRE ADVERSARIOS



IA 3.2 - Programación III

1° C - 2023

Lic. Mauro Lucci

# Entornos multiagentes

— — —

- En un entorno **multiagente**, el comportamiento de un agente depende del comportamiento de los demás.
- Es **competitivo** si los objetivos de los agentes están en conflicto.  
 **Ejemplo.** Si un jugador gana una partida de ajedrez, entonces el otro necesariamente pierde.
- De lo contrario, se dice **cooperativo**.  
 **Ejemplo.** En el contexto de un taxi autónomo, evitar choques es un objetivo común entre todos los agentes del tránsito.
- La **teoría de juegos**, una rama de la economía, estudia este tipo de entornos.

# Búsqueda entre adversarios

— — —

Los problemas de **búsqueda entre adversarios**, también llamados **juegos**, involucran entornos competitivos.

La literatura es muy amplia, nos limitamos a juegos con las siguientes características.

1. **Dos jugadores.**
2. **Por turnos.**
3. **Suma cero.** La ganancia del jugador ganador al final del juego es equivalente a la pérdida del jugador perdedor.

 **Ejemplo.** Si el jugador A gana una partida de ajedrez, entonces el jugador B pierde, y viceversa.


4. **Información perfecta.** En todo momento, los jugadores pueden observar por completo los elementos del juego.

 **Ejemplo.** El ta-te-ti es un juego con información perfecta. Por el contrario, el poker es un juego con información imperfecta.

# Motivación – Dificultad

---

El interés en los juegos radica en la **dificultad** para resolverlos.

 **Ejemplo.** El ajedrez tiene en promedio un factor de ramificación de **35** y cada jugador hace **50** movimientos. Luego, el árbol de búsqueda tiene aproximadamente

$$35^{100} \approx 10^{154} \text{ nodos}$$

(aunque la cantidad de estados distintos es  $\approx 10^{40}$ ).

# Motivación – Rapidez

— — —

Al igual que en muchos problemas del mundo real, los juegos requieren la habilidad de **tomar decisiones**, de forma **rápida** y **eficiente**, aún cuando calcular una decisión óptima no sea posible.

**Hacer el mejor uso posible del tiempo disponible.**

## Ejemplo.

Una implementación de A\* que sea la mitad de eficiente que otra, simplemente demorará el doble de tiempo en ejecutarse.

Al contrario, un programa de ajedrez que sea la mitad de eficiente que otro en el tiempo disponible, seguramente será derrotado.

# Juegos

— — —

Formalmente, los juegos son un tipo de problema de búsqueda con los siguientes 6 elementos.

1.  **$S_0$ . Estado inicial.**
2. **JUGADOR(S)**. Determina qué jugador tiene el turno en ese estado.  
Los jugadores se llaman **MAX** y **MIN**, siendo MAX quién juega primero.
3. **ACCIONES(S)**. Devuelve el conjunto de **acciones** legales en un estado.
4. **RESULTADO(S,A)**. El **modelo transicional**, que define el estado resultado de aplicar una acción a un estado.


# Juegos

— — —


5. **TEST-TERMINAL(S)**. Determina si el juego está terminado.

Los estados en donde el juego ha finalizado se llaman **estados terminales**.

6. **UTILIDAD(S,P)**. Una **función de utilidad** que asigna un valor numérico (recompensa) a cada jugador P en un estado terminal S.

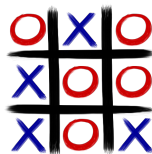
 **Ejemplo.** En ajedrez la utilidad puede ser 1, 0 o  $\frac{1}{2}$  según si el jugador gana, pierde o empata.

Un juego es de **suma cero** si la suma de las utilidades de los jugadores es la misma para todo estado terminal.

 **Ejemplo.** El ajedrez es un juego de suma cero porque las utilidades en los estados terminales suman 1 ( $1+0$ ,  $0+1$ ,  $\frac{1}{2} + \frac{1}{2}$ ).



## Ejemplo – TA-TE-TI



1.  $S_0$ : matriz  $M$  de tamaño  $3 \times 3$  rellena de  $-$ 's. Número de estados:  $9! = 362.880$ .

2.  $JUGADOR(S) = \begin{cases} \text{MAX} & \text{si } M \text{ tiene misma cantidad de } X\text{'s que de } O\text{'s.} \\ \text{MIN} & \text{en caso contrario.} \end{cases}$

En particular,  $JUGADOR(S_0) = \text{MAX}$ .

3.  $ACCIONES(S) = \{(i,j): M[i][j] = -\}$ .

4.  $RESULTADO(S, (i,j))$ , si es el turno de MAX, el resultado es escribir una  $X$  en  $M[i][j]$  y, en caso contrario, escribir un  $O$  en  $M[i][j]$ .

5.  $TEST-TERMINAL(S) = \text{True}$  si  $M$  no tiene  $-$ 's o si hay 3  $X$ 's o 3  $O$ 's en línea (horizontal, vertical o diagonal).

6.  $UTILIDAD(S,P) \in \{1,0,\frac{1}{2}\}$  si  $P$  gana, pierde o empata respectivamente.

MAX gana y MIN pierde si hay 3  $X$ 's en línea, MIN gana y MAX pierde si hay 3  $O$ 's en línea y empatan en caso contrario.



# Árbol de juego

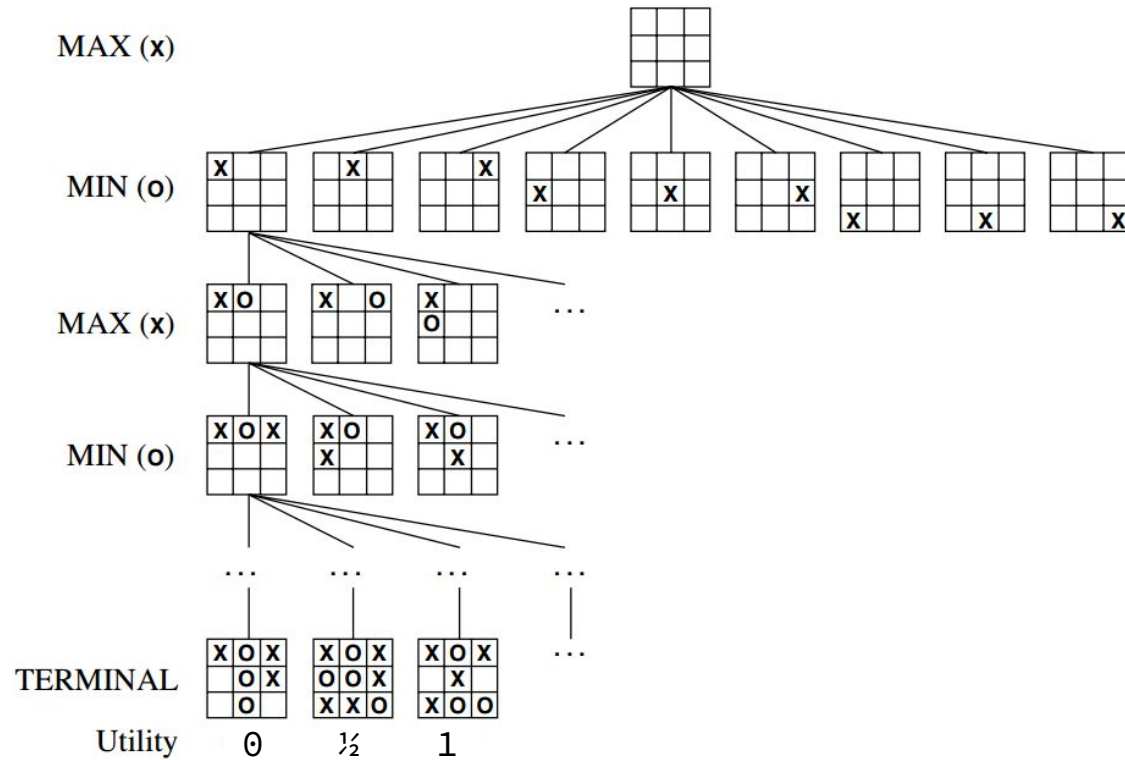
---

El estado inicial, las acciones y el modelo transicional definen un **árbol de juego** – donde los nodos son estados y las ramas son acciones – similar a un árbol de búsqueda.



# Ejemplo – TA-TE-TI

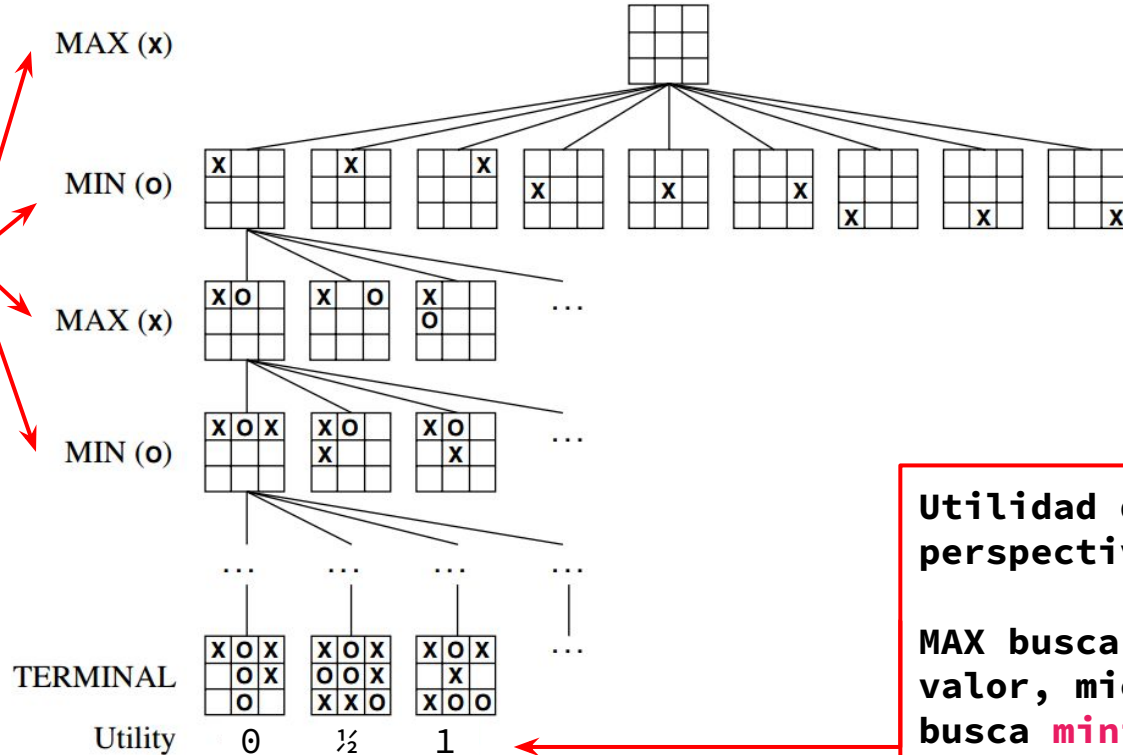
— — —





# Ejemplo – TA-TE-TI

Los niveles indican el turno de juego.



Utilidad desde la perspectiva de MAX.

MAX busca **maximizar** este valor, mientras que MIN busca **minimizarlo**.

# Estrategia óptima

— — —

En un juego, el objetivo es hallar una **estrategia** que especifique:

1. La acción de MAX en el estado inicial.
2. Luego, la acción de MAX en los estados que resulten de cada posible respuesta de MIN.
3. Luego, la acción de MAX en los estados que resulten de cada posible respuesta de MIN a las acciones anteriores, y así sucesivamente.

Una **estrategia óptima** produce un resultado al menos tan bueno como cualquier otra estrategia cuando se juega contra un oponente **infalible**.

**¿Cómo hallar una estrategia óptima?**

# Valor minimax

---

Dado un árbol de juego,

- El **valor minimax** de un nodo es la utilidad (para MAX) de estar en el estado correspondiente, **asumiendo que ambos jugadores juegan óptimamente** desde allí al final del juego.
- La estrategia óptima puede determinarse a partir del valor minimax de cada nodo.

# Valor minimax

— — —

$$\text{MINIMAX}(S) = \begin{cases} \text{UTILIDAD}(S, \text{MAX}) & \text{SI } \text{TEST-TERMINAL}(S) \\ \max \{ \text{MINIMAX}(\text{RESULTADO}(S, A)) : A \in \text{ACCIONES}(S) \} & \text{SI } \text{JUGADOR}(S) = \text{MAX} \\ \min \{ \text{MINIMAX}(\text{RESULTADO}(S, A)) : A \in \text{ACCIONES}(S) \} & \text{SI } \text{JUGADOR}(S) = \text{MIN} \end{cases}$$

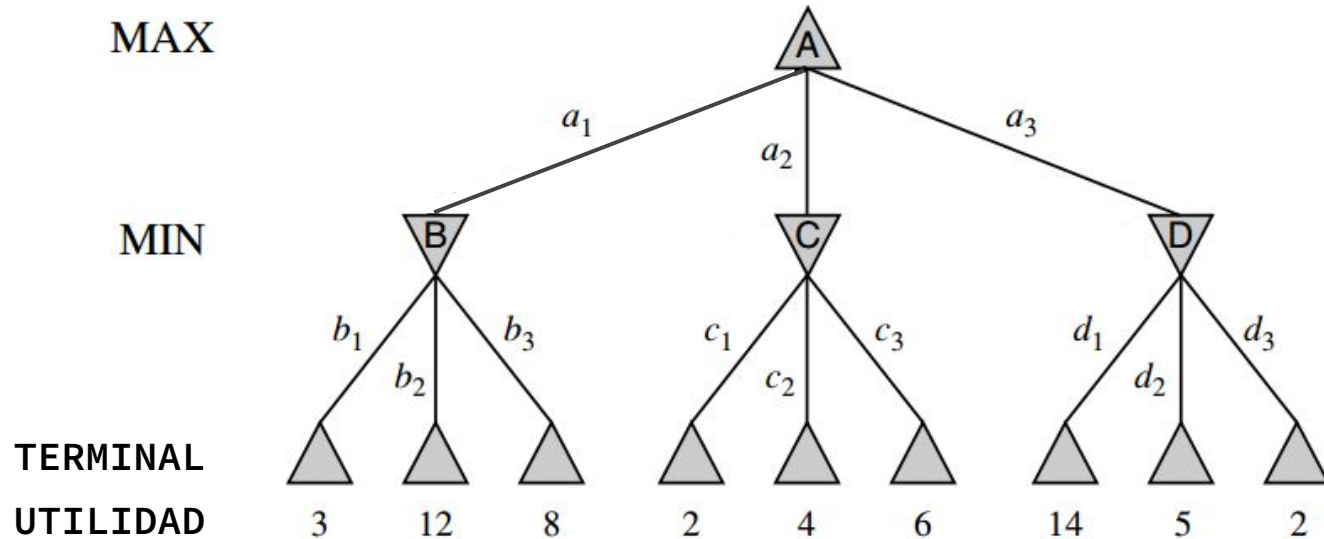
- El valor minimax de un nodo con un estado terminal es su utilidad.
- MAX prefiere moverse a un nodo con mayor valor minimax.
- MIN prefiere moverse a un nodo con menor valor minimax.



# Ejemplo

— — —

Calcular los valores minimax para los nodos del siguiente árbol de juego de dos **capas**.

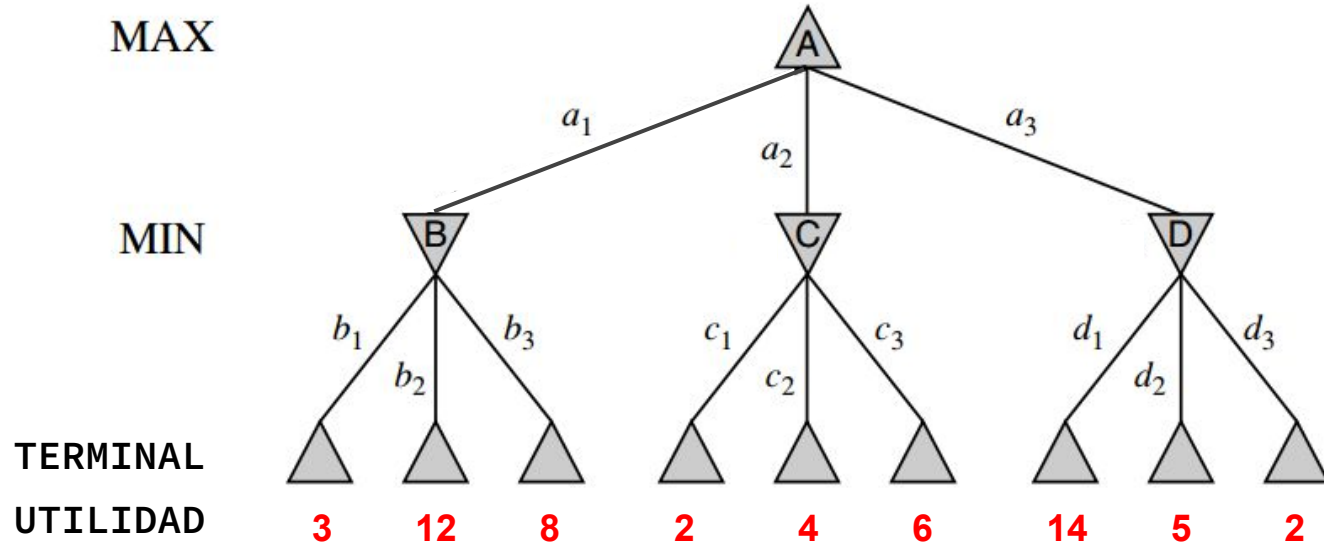




# Ejemplo

**MINIMAX**(S) =

$$\begin{cases} \text{UTILIDAD}(S, \text{MAX}) & \text{SI } \text{TEST-TERMINAL}(S) \\ \max \{ \text{MINIMAX}(\text{RESULTADO}(S, A)) : A \in \text{ACCIONES}(S) \} & \text{SI } \text{JUGADOR}(S) = \text{MAX} \\ \min \{ \text{MINIMAX}(\text{RESULTADO}(S, A)) : A \in \text{ACCIONES}(S) \} & \text{SI } \text{JUGADOR}(S) = \text{MIN} \end{cases}$$



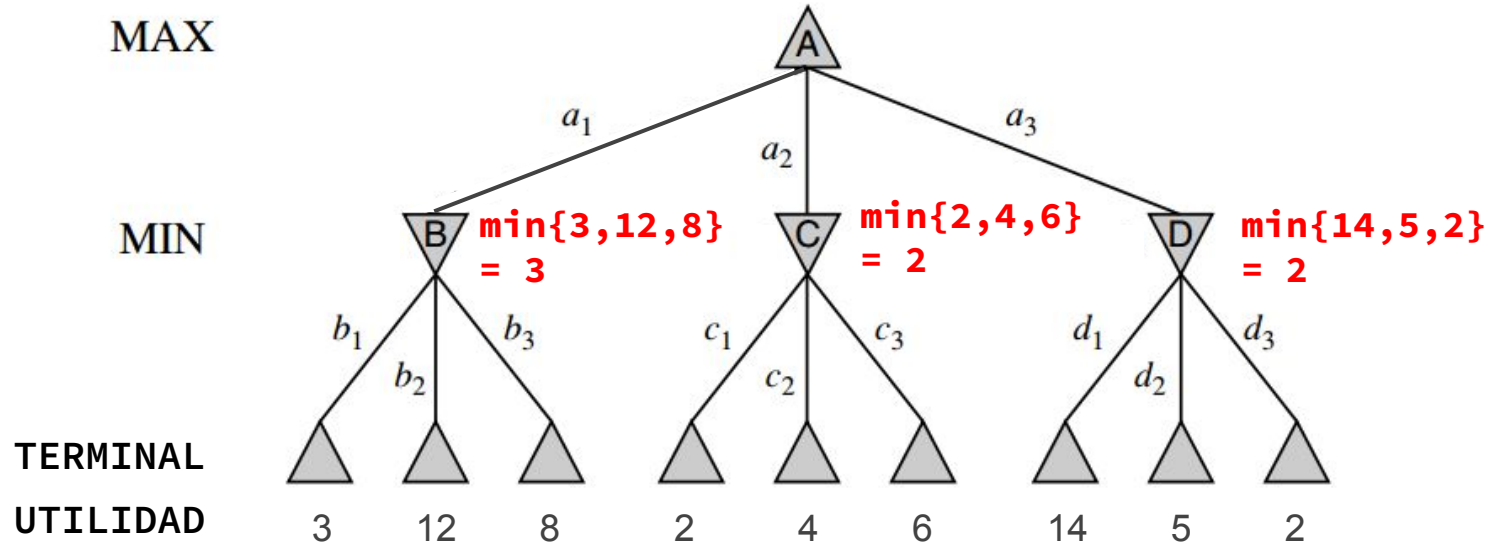




# Ejemplo

**MINIMAX**(S) =

$$\begin{cases} \text{UTILIDAD}(S, \text{MAX}) & \text{SI } \text{TEST-TERMINAL}(S) \\ \max \{ \text{MINIMAX}(\text{RESULTADO}(S, A)) : A \in \text{ACCIONES}(S) \} & \text{SI } \text{JUGADOR}(S) = \text{MAX} \\ \min \{ \text{MINIMAX}(\text{RESULTADO}(S, A)) : A \in \text{ACCIONES}(S) \} & \text{SI } \text{JUGADOR}(S) = \text{MIN} \end{cases}$$

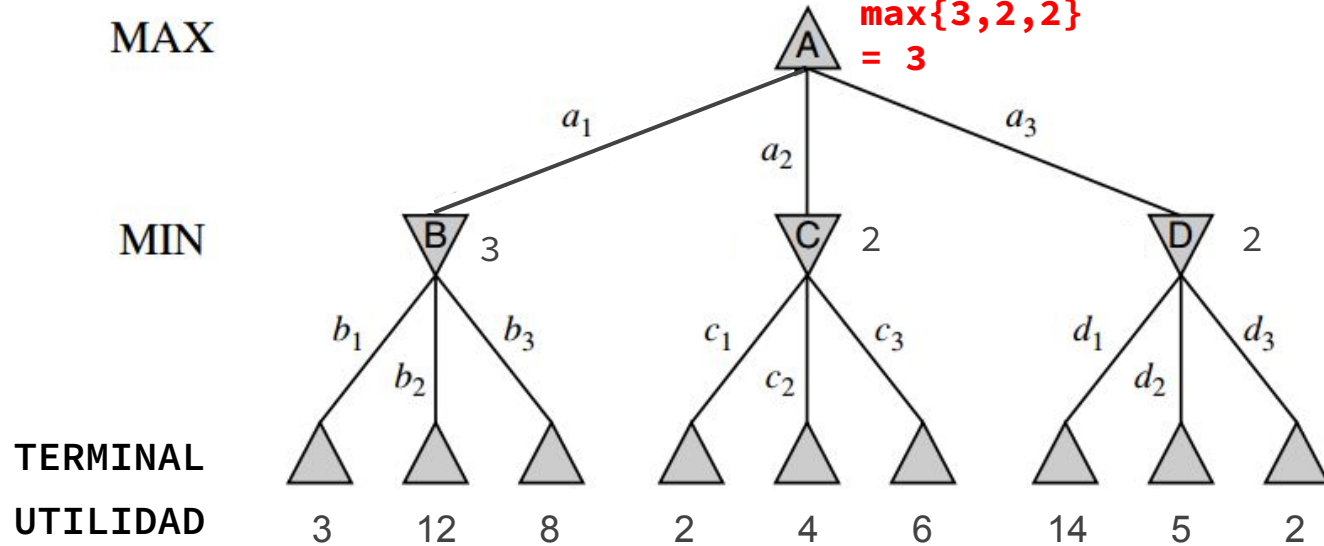




# Ejemplo

**MINIMAX**(S) =

$$\begin{cases} \text{UTILIDAD}(S, \text{MAX}) & \text{SI } \text{TEST-TERMINAL}(S) \\ \max \{ \text{MINIMAX}(\text{RESULTADO}(S, A)) : A \in \text{ACCIONES}(S) \} & \text{SI } \text{JUGADOR}(S) = \text{MAX} \\ \min \{ \text{MINIMAX}(\text{RESULTADO}(S, A)) : A \in \text{ACCIONES}(S) \} & \text{SI } \text{JUGADOR}(S) = \text{MIN} \end{cases}$$

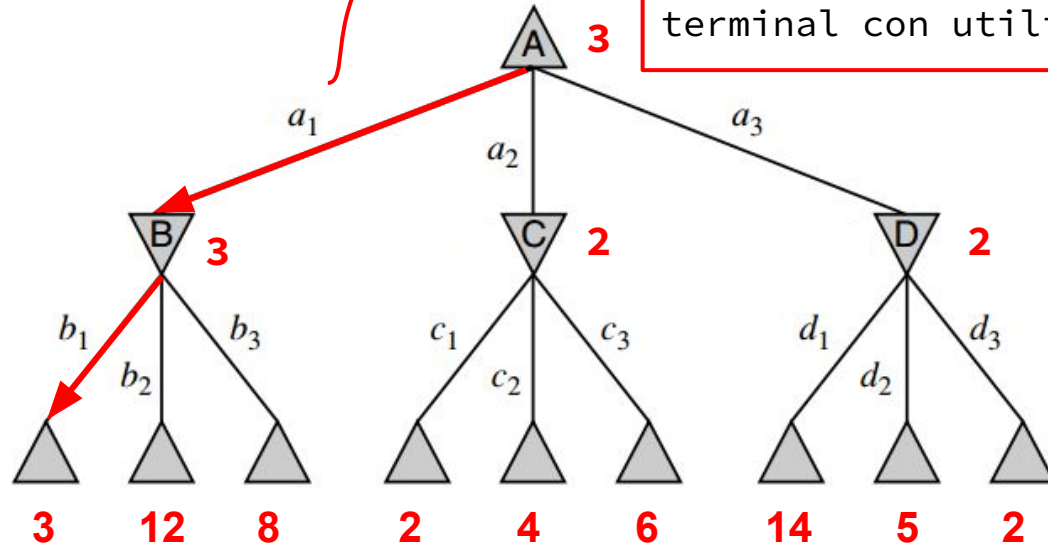




# Ejemplo

---

MAX  
MIN  
TERMINAL  
UTILIDAD



**Estrategia minimax.** La elección óptima para MAX en la raíz es la acción  $a_1$  porque lleva al nodo con **mayor valor minimax**.

Si MIN juega óptimamente, entonces responderá con la acción  $b_1$ , alcanzado un estado terminal con utilidad 3.

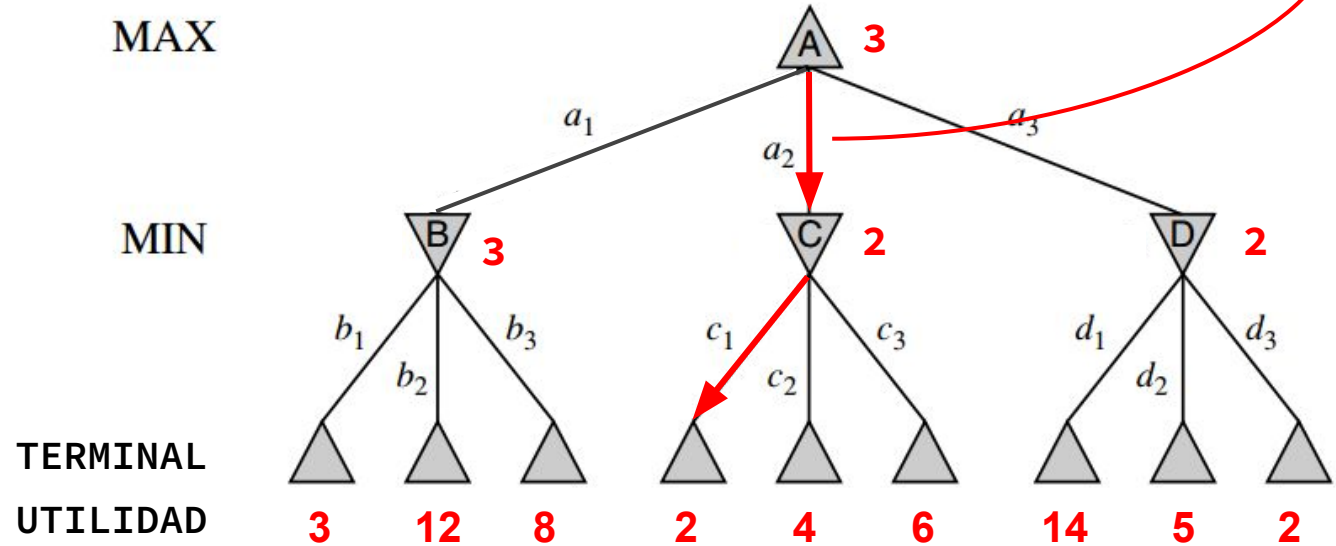
**Valores minimax**



# Ejemplo

---

Por el contrario, si MAX aplicase la acción  $a_2$  en la raíz, entonces MIN respondería con la acción  $c_1$ , alcanzado un estado terminal con utilidad  $2 < 3$ .

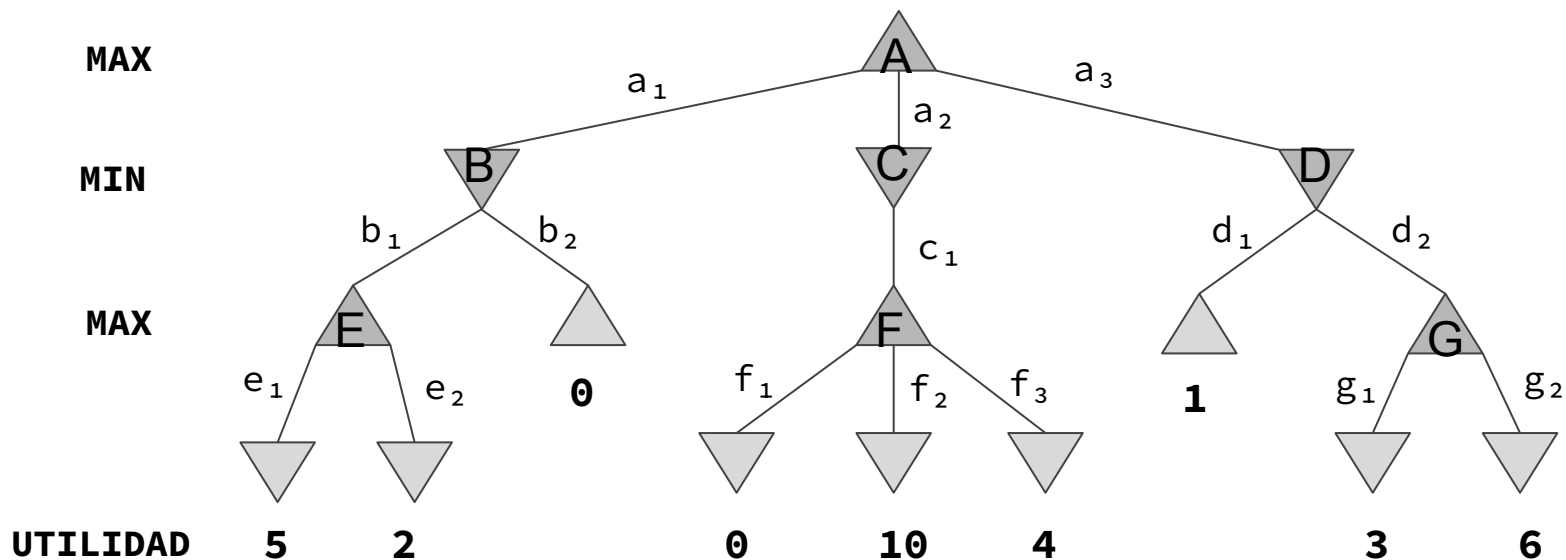


Valores minimax



# Ejercicio

Determinar la jugada óptima para MAX según la estrategia minimax para el siguiente árbol de juego.

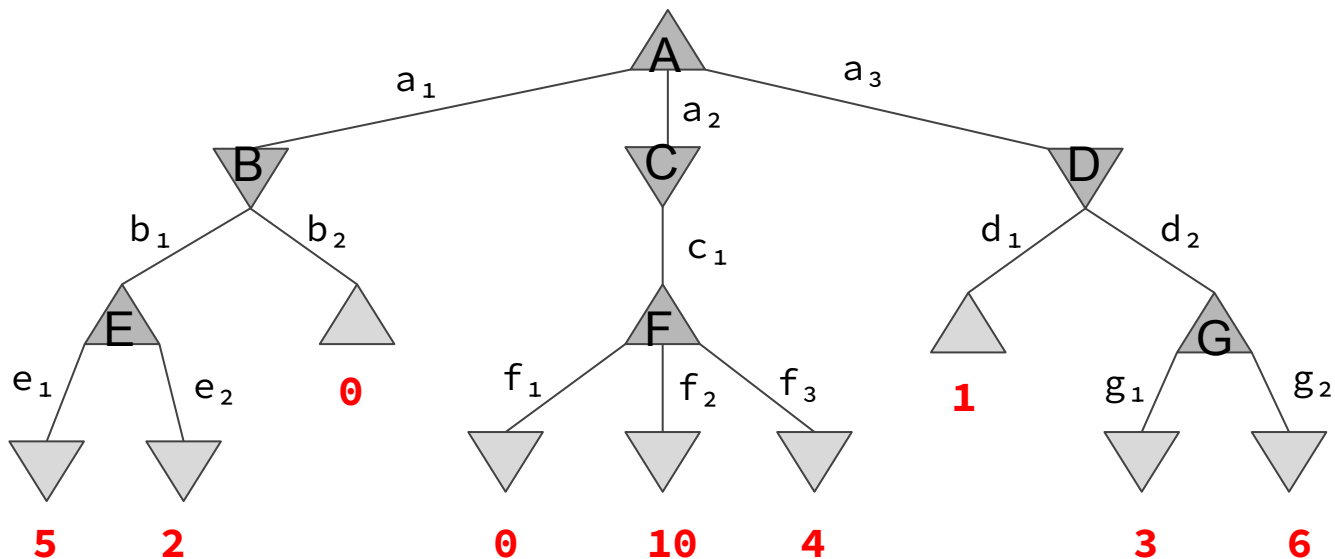




# Respuesta

---

Determinar la jugada óptima para MAX según la estrategia minimax para el siguiente árbol de juego.

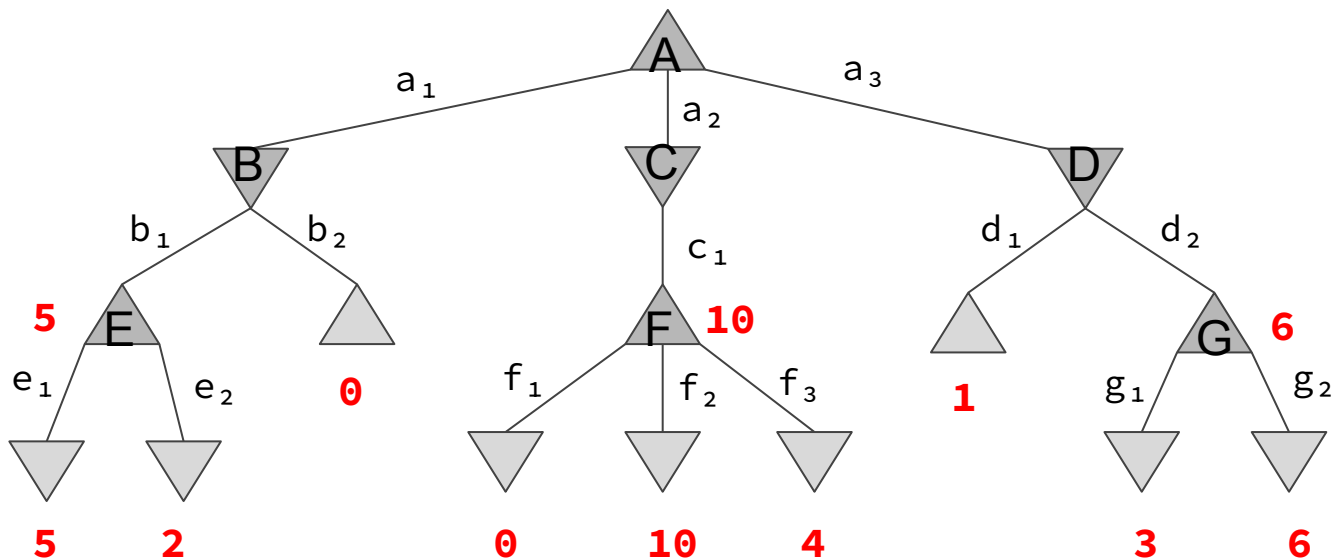




# Respuesta

---

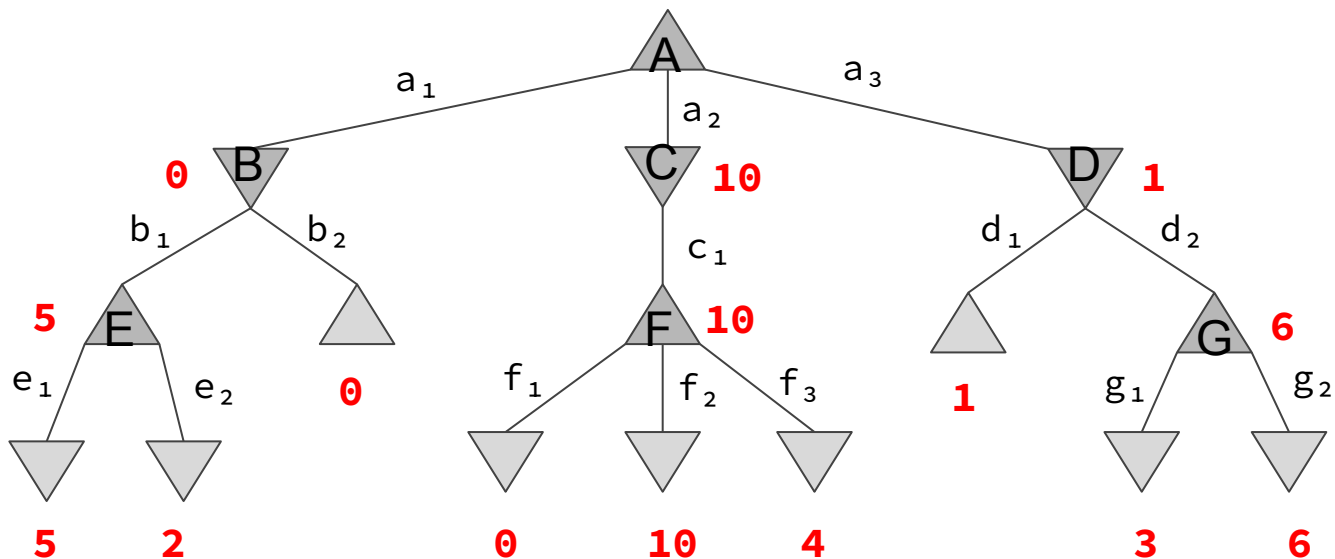
Determinar la jugada óptima para MAX según la estrategia minimax para el siguiente árbol de juego.





# Respuesta

---  
Determinar la jugada óptima para MAX según la estrategia minimax para el siguiente árbol de juego.

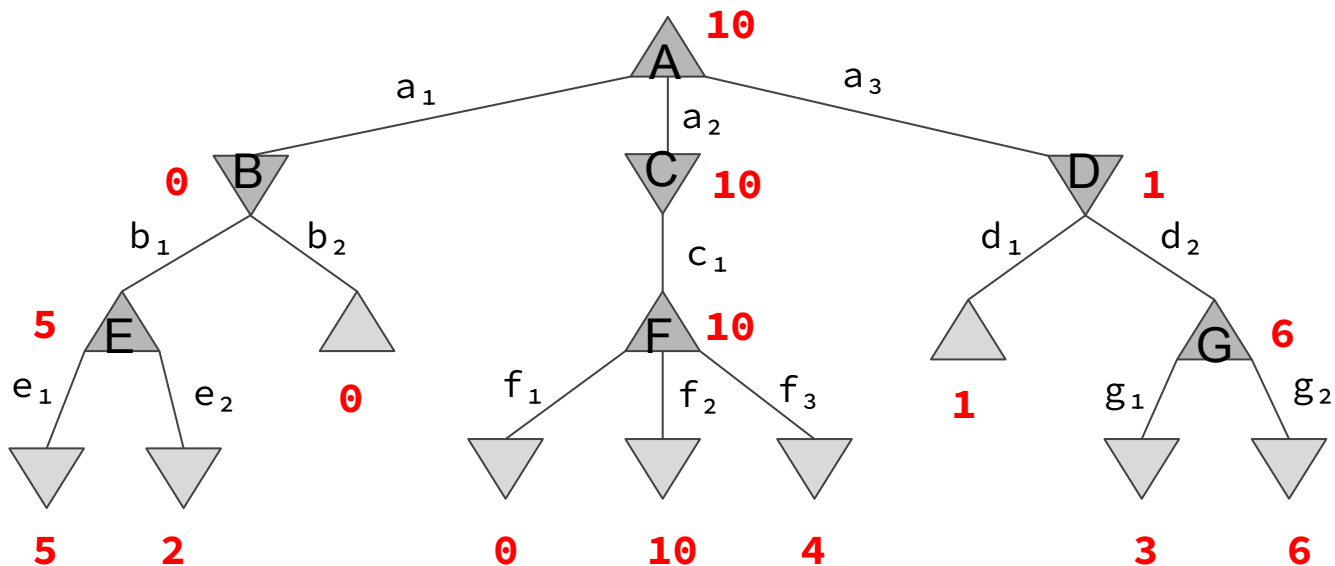






# Respuesta

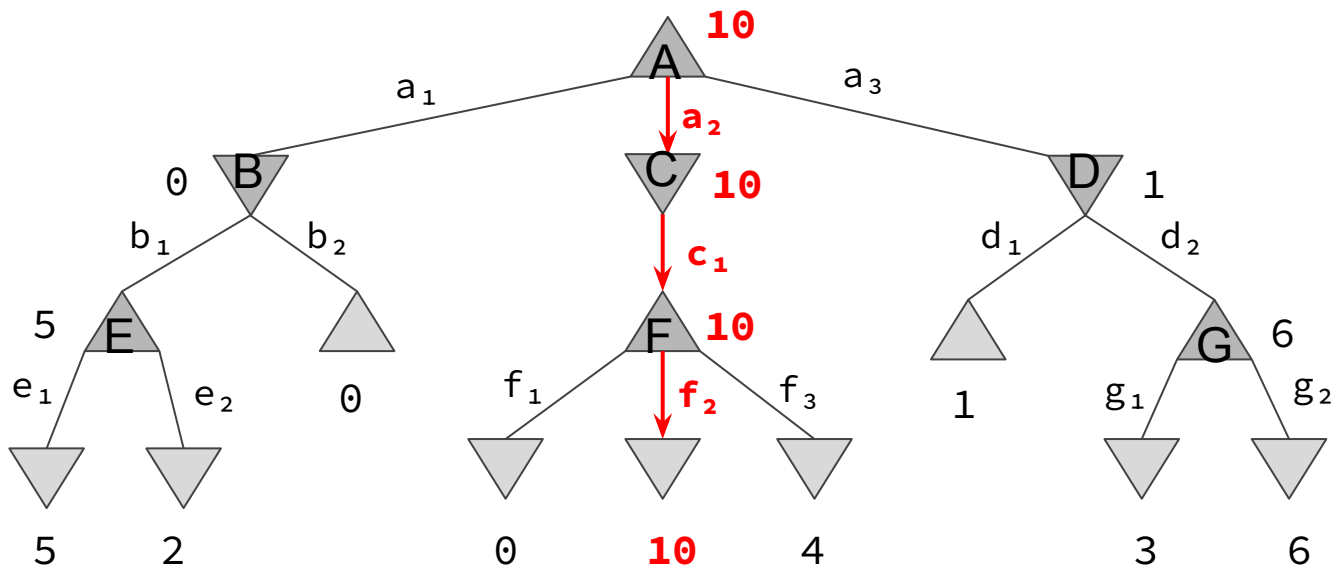
---  
Determinar la jugada óptima para MAX según la estrategia minimax para el siguiente árbol de juego.





# Respuesta

Determinar la jugada óptima para MAX según la estrategia minimax para el siguiente árbol de juego.



# Otras estrategias

---

- La estrategia minimax es óptima asumiendo que MAX y MIN juegan óptimamente.
- **Minimax maximiza el resultado del peor caso para MAX.**
- ¿Y si MIN no juega óptimamente? Entonces a MAX le irá aún mejor.
- Pero otras estrategias contra oponentes subóptimos pueden ser incluso mejores que minimax, aunque necesariamente son peores contra oponentes óptimos.

# Algoritmo minimax

— — —

```
# Calcula recursivamente el valor minimax  
# de un estado MAX
```

```
1 function MINIMAX-MAX(problema, estado) return valor  
2   if problema.TEST-TERMINAL(estado) then  
3     return problema.UTILIDAD(estado, MAX)  
4   valor ←  $-\infty$   
5   forall acción in problema.ACCIONES(estado) do  
6     sucesor ← problema.RESULTADO(estado, acción)  
7     valor ← max(valor,  
6           MINIMAX-MIN(problema, sucesor)  
8   return valor
```

```
# Calcula recursivamente el valor minimax  
# de un estado MIN
```

```
1 function MINIMAX-MIN(problema, estado) return valor  
2   if problema.TEST-TERMINAL(estado) then  
3     return problema.UTILIDAD(estado, MAX)  
4   valor ←  $+\infty$   
5   forall acción in problema.ACCIONES(estado) do  
6     sucesor ← problema.RESULTADO(estado, acción)  
7     valor ← min(valor,  
6           MINIMAX-MAX(problema, sucesor)  
8   return valor
```

# Algoritmo minimax

— — —

Las funciones son **mutuamente recursivas**.

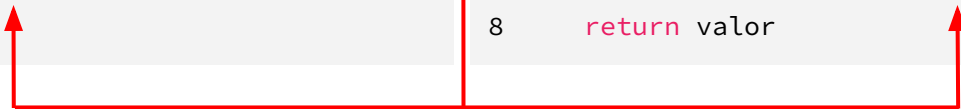
La recursión desciende desde el estado actual hasta los estados terminales (donde la última llamada recursiva termina), y luego los valores minimax se propagan hacia arriba por el árbol a medida que la recursión retrocede.

```
# Calcula recursivamente el valor minimax
# de un estado MAX

1 function MINIMAX-MAX(problema, estado) return valor
2   if problema.TEST-TERMINAL(estado) then
3     return problema.UTILIDAD(estado, MAX)
4   valor ← -∞
5   forall acción in problema.ACCIONES(estado) do
6     sucesor ← problema.RESULTADO(estado, acción)
7     valor ← max(valor,
                  MINIMAX-MIN(problema, sucesor))
8   return valor
```

```
# Calcula recursivamente el valor minimax
# de un estado MIN

1 function MINIMAX-MIN(problema, estado) return valor
2   if problema.TEST-TERMINAL(estado) then
3     return problema.UTILIDAD(estado, MAX)
4   valor ← +∞
5   forall acción in problema.ACCIONES(estado) do
6     sucesor ← problema.RESULTADO(estado, acción)
7     valor ← min(valor,
                  MINIMAX-MAX(problema, sucesor))
8   return valor
```



# Algoritmo minimax

— — —

```
# Devuelve la acción que tiene que aplicar MAX en el estado inicial
# según la estrategia minimax

1 function MINIMAX(problema) return acción
2     sucs ← {acción: MINIMAX-MIN(problema, problema.RESULTADO(problema.S0, acción))
           for acción in problema.ACCIONES(problema.S0)}
3     return max(sucs, key=sucs.get) # obtiene la clave con el mayor valor
```

# Performance de MINIMAX

— — —

- Hace una exploración **primero en profundidad** completa del árbol de juego.
- Sean  $m$  la altura del árbol y  $b$  el factor de ramificación.
- **Consumo de tiempo.** Genera  $b^m$  nodos.
- **Consumo de memoria.** Se necesitan almacenar  $b.m$  llamadas recursivas.
- **Para juegos reales, este consumo de tiempo es completamente impráctico.** Pero el algoritmo minimax sirve como base para algoritmos más prácticos y para el análisis matemático.

# Poda alfa-beta

Es posible computar la estrategia minimax sin explorar la totalidad de los nodos del árbol de juego.

La idea es **podar** aquellos nodos para los cuales se tenga evidencia de que no influyen en la decisión final.

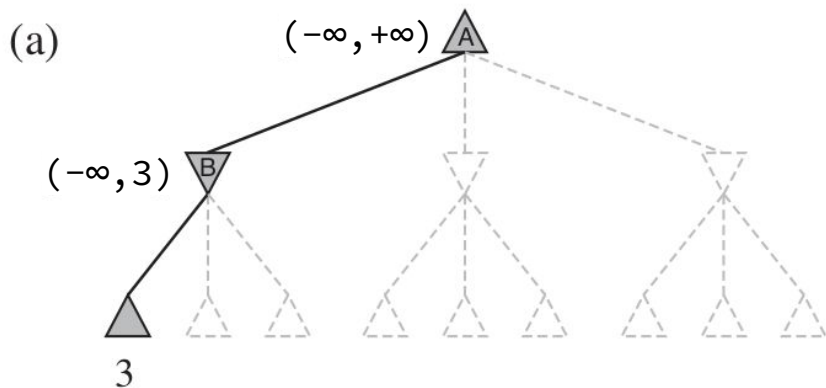
En particular, la poda **alfa-beta** mantiene cotas (inferiores y superiores) del valor minimax de cada nodo a partir de la información que se tiene hasta ese momento.

— — —

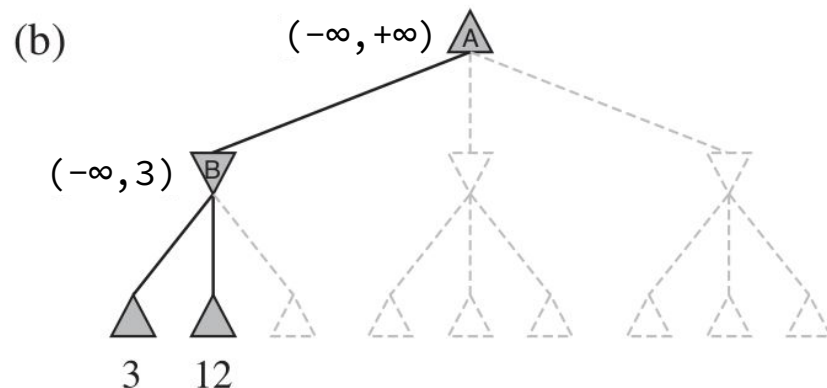


# Paso a paso

---

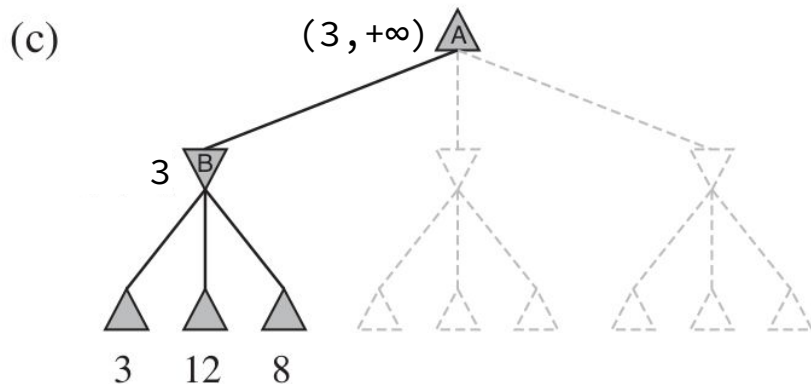


(a) La primera hoja debajo de B tiene valor 3. Luego B, que es un nodo MIN, tiene valor  $\leq 3$ .



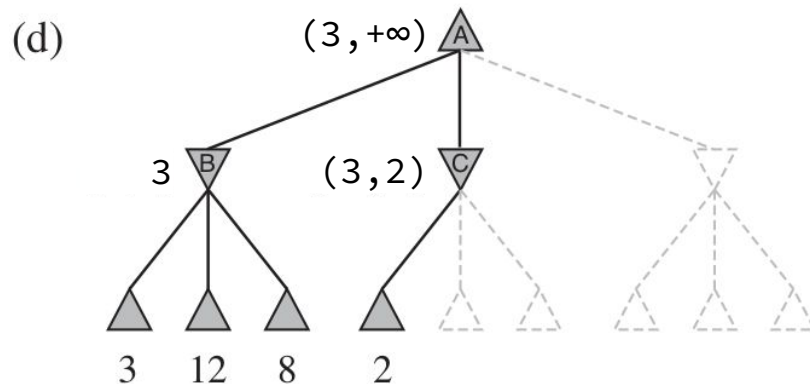
(b) La segunda hoja debajo de B tiene valor 12. Dado que MIN va a evitar este nodo, el valor de B sigue siendo  $\leq 3$ .

# Paso a paso



(c) La tercera hoja debajo de B tiene valor 8. Como todos sus sucesores han sido explorados, el valor de B es  $= 3$ .

Podemos inferir que el valor de A es  $\geq 3$ , porque MAX tiene una alternativa de valor 3.

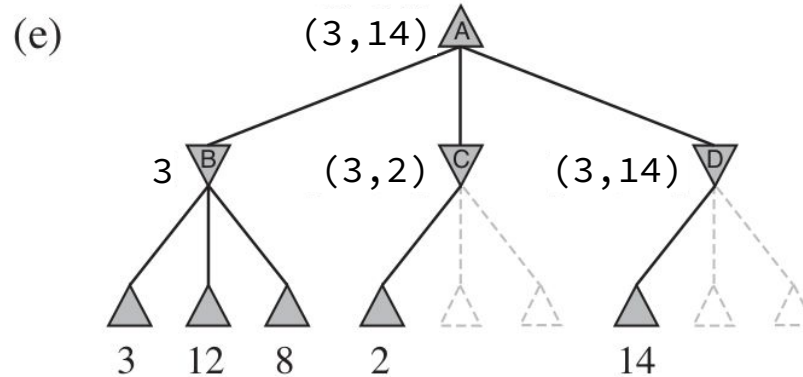


(d) La primera hoja debajo de C tiene valor 2. Luego C, que es un nodo MIN, tiene valor  $\leq 2$ .

Sabemos que B tiene valor  $= 3$ , luego MAX nunca elegirá a C. Por lo tanto, C puede ser **podado** (no seguimos explorando sus hijos).

# Paso a paso

— — —



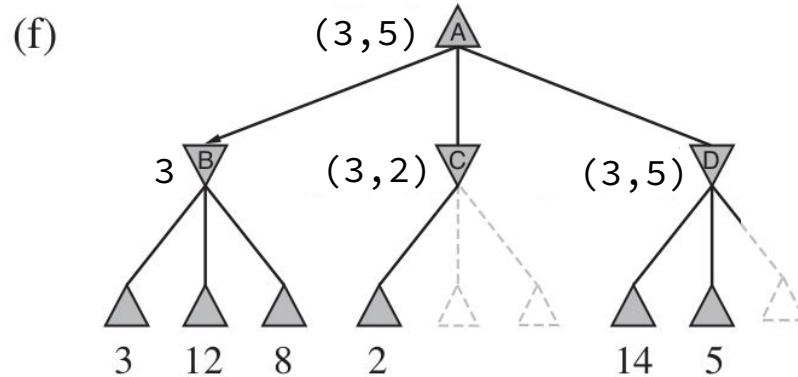
(e) La primera hoja debajo de D tiene valor 14. Luego D, que es un nodo MIN, tiene valor  $\leq 14$ .

Dado que D todavía tiene chances de mejorar la decisión de MAX (que al momento tiene costo  $\geq 3$ ), no puede ser podado y su siguiente hijo debe explorarse.

Además, todos los sucesores del estado inicial han sido acotados superiormente, luego el valor de A es  $\leq 14$ .

# Paso a paso

---



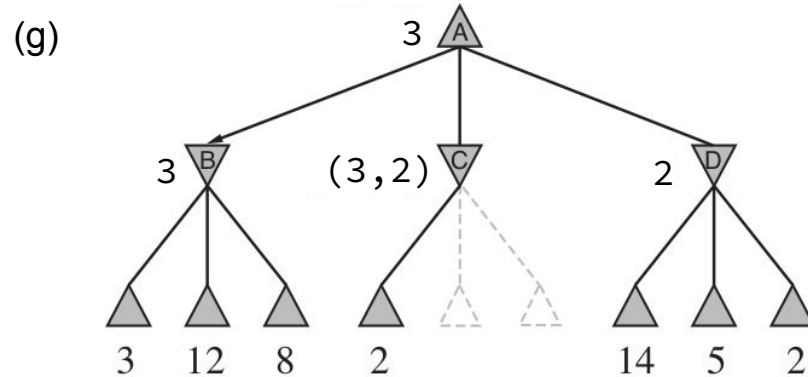
(f) La segunda hoja debajo de D tiene valor 5. Luego D, que es un nodo MIN, tiene valor  $\leq 5$ .

Dado que D todavía tiene chances de mejorar la decisión de MAX (que al momento tiene costo  $\geq 3$ ), no puede ser podado y su siguiente hijo debe explorarse.

Además, el valor de A es ahora  $\leq 5$ .

# Paso a paso

---



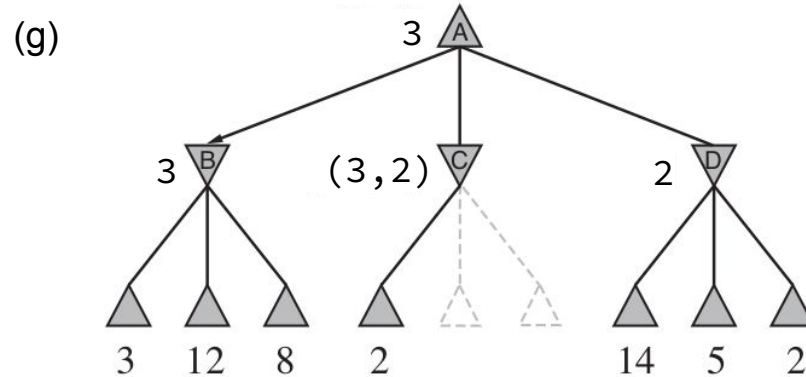
(g) La tercera hoja debajo de D tiene valor 2. Luego todos los sucesores de D han sido explorados y, como es un nodo MIN, su valor es = 2.

Por lo tanto, el valor de A es = 3 y la mejor decisión de MAX es moverse a B.

# Paso a paso

— — —

La poda alfa-beta evitó explorar dos nodos del árbol de juego.



(g) La tercera hoja debajo de D tiene valor 2. Luego todos los sucesores de D han sido explorados y, como es un nodo MIN, su valor es = 2.

Por lo tanto, el valor de A es = 3 y la mejor decisión de MAX es moverse a B.

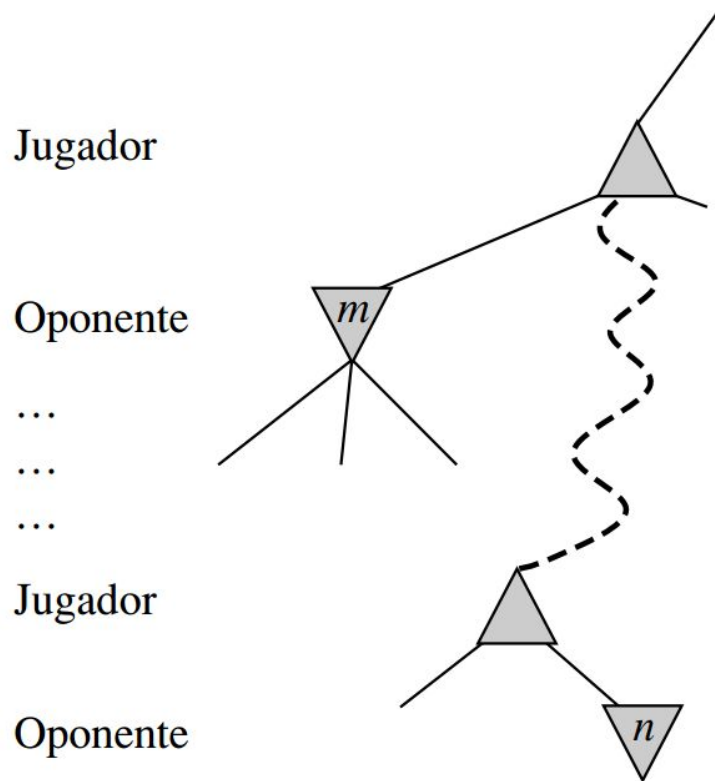
# Caso general

---

La poda alfa-beta se puede aplicar a árboles de cualquier profundidad.

Sea un nodo  $n$  tal que el jugador P tiene la opción de moverse a ese nodo. Si P tiene una mejor opción  $m$  más arriba en el árbol, entonces  $n$  **nunca será alcanzado en una jugada real**.

Una vez que tengamos suficiente información sobre  $n$  (por explorar a sus descendientes) para llegar a esta conclusión, podemos podarlo.



# Poda alfa-beta

---

- Puede podar subárboles enteros y no solo hojas.
- El consumo de tiempo sigue siendo exponencial, pero se puede reducir el exponente a la mitad.
- Recibe el nombre por dos parámetros que se agregan al algoritmo minimax:
  - $\alpha$ : es el valor de la mejor opción para MAX encontrada hasta el momento a lo largo del camino para MAX.
  - $\beta$ : es el valor de la mejor opción para MIN encontrada hasta el momento a lo largo del camino para MIN.



# Algoritmo minimax con poda alfa-beta

— — —

```
# Calcula recursivamente el valor minimax  
# de un estado MAX con poda alfa-beta
```

```
1 function MINIMAX-MAX(problema,estado, $\alpha$ , $\beta$ ) return valor  
2   if problema.TEST-TERMINAL(estado) then  
3     return problema.UTILIDAD(estado,MAX)  
4   valor  $\leftarrow -\infty$   
5   forall acción in problema.ACCIONES(estado) do  
6     sucesor  $\leftarrow$  problema.RESULTADO(estado, acción)  
7     valor  $\leftarrow$  max(valor,  
                        MINIMAX-MIN(problema,sucesor, $\alpha$ , $\beta$ )  
8     if valor  $\geq \beta$  then return valor # podar  
9      $\alpha \leftarrow$  max( $\alpha$ ,valor)  
10  return valor
```

```
# Calcula recursivamente el valor minimax  
# de un estado MIN con poda alfa-beta
```

```
1 function MINIMAX-MIN(problema,estado, $\alpha$ , $\beta$ ) return valor  
2   if problema.TEST-TERMINAL(estado) then  
3     return problema.UTILIDAD(estado,MAX)  
4   valor  $\leftarrow +\infty$   
5   forall acción in problema.ACCIONES(estado) do  
6     sucesor  $\leftarrow$  problema.RESULTADO(estado, acción)  
7     valor  $\leftarrow$  min(valor,  
                        MINIMAX-MAX(problema,sucesor, $\alpha$ , $\beta$ )  
8     if valor  $\leq \alpha$  then return valor # podar  
9      $\beta \leftarrow$  min( $\beta$ ,valor)  
10  return valor
```

# Algoritmo minimax con poda alfa-beta

— — —

```
# Devuelve la acción que tiene que aplicar MAX en el estado inicial
# según la estrategia minimax con poda alfa-beta

1 function MINIMAX-ALFA-BETA(problema) return acción
2   sucs ← {acción: MINIMAX-MIN(problema, problema.RESULTADO(problema.S0, acción), -∞, +∞)
           for acción in problema.ACCIONES(problema.S0)}
3   return max(sucs, key=sucs.get) # obtiene la clave con el mayor valor
```

# Decisiones imperfectas en tiempo real

La profundidad en el árbol de juego a la que necesita llegar el algoritmo minimax a menudo no es práctica.

Típicamente las acciones deben ser realizadas en tiempos razonables, unos minutos como mucho.

**¿Cómo tomar buenas  
decisiones en pocos  
minutos?**

— — —

# Funciones evaluación

— — —

Una idea revolucionaria... Cortar la búsqueda anticipadamente, convirtiendo estados no-terminales en terminales, y entonces aplicar una **función de evaluación** heurística para estimar la utilidad esperada en un estado dado.

El momento en el que se corta la búsqueda se determina por medio de una función de test de corte. Así, tenemos la siguiente heurística minimax para un estado  $S$  y una profundidad  $d$ :

**H-MINIMAX**( $S, d$ ) =

EVAL( $S$ )

SI TEST-CORTE( $S, d$ )


max {**H-MINIMAX**(RESULTADO( $S, A$ ),  $d+1$ ):  $A \in$  ACCIONES( $S$ )} SI JUGADOR( $S$ ) = MAX

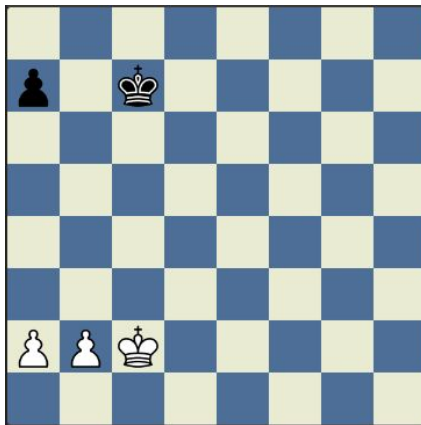
min {**H-MINIMAX**(RESULTADO( $S, A$ ),  $d+1$ ):  $A \in$  ACCIONES( $S$ )} SI JUGADOR( $S$ ) = MIN

# ¿Cómo diseñarlas?

— — —

1. Agrupar en una misma **categoría** (o clase de equivalencia) aquellos estados que tengan los mismos valores en ciertas **características**.


 **Ejemplo (Ajedrez).** La categoría “**2 peones vs. 1 peón**” contiene todos los estados endgame con 2 peones (y el rey) blancos contra 1 peón (y el rey) negro.



# ¿Cómo diseñarlas?

— — —

La función de evaluación debe devolver para cada categoría un **valor esperado** que refleje la proporción de sus estados que conducen a triunfos, empates y pérdidas.

 **Ejemplo (Ajedrez).** Nuestra experiencia sugiere que **72%** de los estados de la categoría **“2 peones vs. 1 peón”** llevan a una victoria (1 de utilidad), **20%** a una derrota (0) y **8%** a un empate (1/2), entonces una función de evaluación razonable para los estados de esta categoría devuelve el valor:

$$(0,72 \times 1) + (0,2 \times 0) + (0,08 \times \frac{1}{2}) = 0,76$$

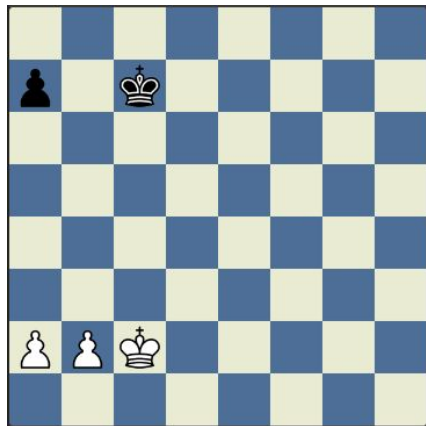
En la práctica, este análisis suele requerir muchas categorías y demasiada experiencia para estimar correctamente las probabilidades.

# ¿Cómo diseñarlas?

— — —

2. Calcular las contribuciones numéricas de cada **característica** y luego **combinarlas** para encontrar el valor total.

📖 **Ejemplo (Ajedrez)**. Las piezas tienen un **valor material** aproximado: cada peón vale 1, alfil y torre valen 3, torre 5 y reina 9.



En este ejemplo, el valor total de los peones blancos es 2 y de los negros 1.

Otras características más complejas incluyen las posiciones de las piezas, el estado del juego (inicio o endgame), seguridad del rey, estructura de los peones, etc.

# ¿Cómo diseñarlas?

---

Las contribuciones directamente se pueden sumar para obtener la evaluación de un estado. Así, la función de evaluación es una **función ponderada lineal** (weighted linear function):

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^n w_i f_i(s) ,$$

donde cada  $w_i$  es un peso y cada  $f_i$  es una característica.

Esta fórmula asume que la contribución de cada característica es independiente de las demás. Funciones más poderosas incorporan términos no lineales.





# Resumen

- ❑ Un **juego** se puede definir con un **estado inicial**, una función que determina de cuál **jugador** es el turno, las **acciones** permitidas en cada estado, el **resultado** de cada acción, el **test terminal** y la función de **utilidad** en estados terminales.
- ❑ Nos limitamos a juegos de **dos jugadores, de suma cero, con información perfecta**.
- ❑ El algoritmo **minimax** elige la estrategia óptima haciendo una exploración primero en profundidad del árbol de juego, asumiendo que ambos juegan **óptimamente**.
- ❑ La poda **alfa-beta** mejora el algoritmo minimax eliminando subárboles irrelevantes.
- ❑ Frecuentemente no es posible considerar el árbol entero, entonces se debe **cortar** la búsqueda en algún punto y aplicar una **función de evaluación** heurística.
- ❑ Programas computacionales han derrotado a campeones en ajedrez, damas y otros juegos. Todos estos programas incorporan en esencia las técnicas descritas. La efectividad depende fundamentalmente de cuántas capas del árbol es capaz de explorar en el tiempo límite.



# Próximamente

— — —

Hasta ahora, los algoritmos de búsqueda vistos tratan a los estados como **cajas negras**. Simplemente consultan si son terminales y aplican sobre ellos funciones heurísticas.

Vamos a **enriquecer la representación de los estados** para introducir una amplia familia de nuevos métodos de búsqueda y para generar una comprensión más profunda de la estructura de los problemas y de su complejidad.