Minimax/Expectimax

CLASE 9

Sistemas multi-agentes

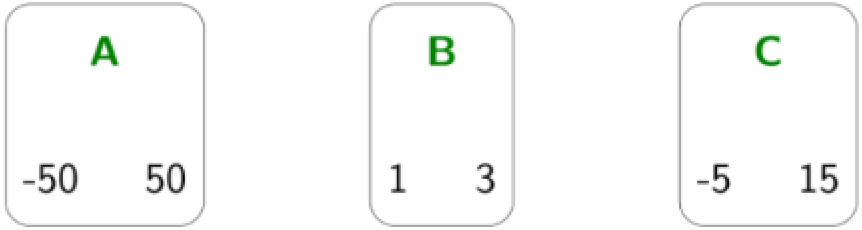
- En particular nos vamos a enfocar en sistemas adversariales por turnos
- Podemos tomar distintos enfoques con los sistemas multi-agentes
  - Tomar a los otros agentes como un agregado
  - Podemos considerar a los adversarios como parte del ambiente
  - Podemos modelar lo que va a hacer el adversario y tomar una decisión en base a lo que haría el modelo

Juegos de suma cero

- Todo lo que vamos a ver hoy es dentro del área de teoría de juegos
- En particular en ambientes determinísticos, 2 jugadores, con observabilidad completa, y de **suma cero**
- Suma cero
  - Bueno para un jugador implica malo para el otro
  - Si yo estoy tratando de maximizar mi ganancia, entonces estoy minimizando la ganancia del rival. Y viceversa
- Ejemplos:
  - Poker
  - Ajedrez

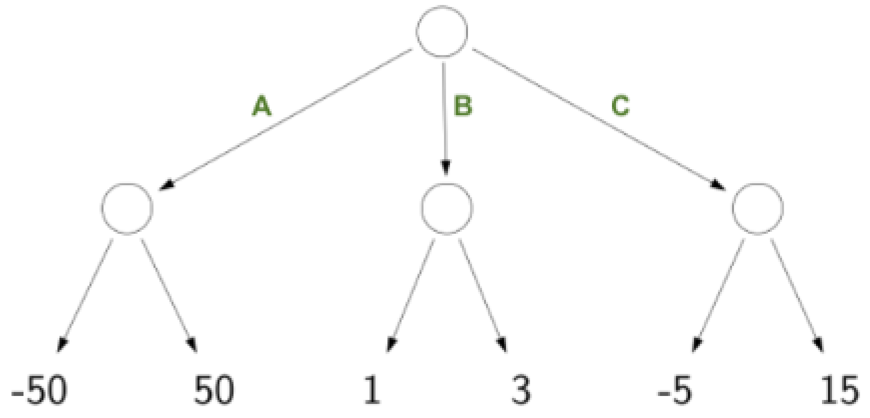
Un ejemplo sencillo

1. Ustedes eligen una caja
2. Yo elijo un número dentro de la caja
3. Su objetivo es maximizar el número elegido



Árbol de juego

- Vamos a modelar el juego como un árbol
- Cada nodo es un estado
  - Cada hoja es un resultado posible del juego

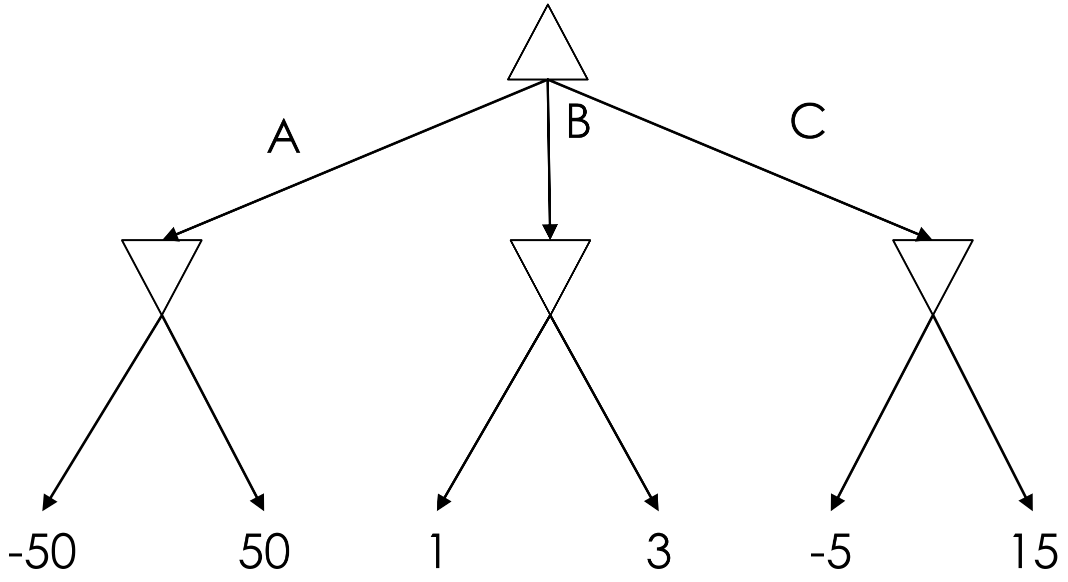


Notación

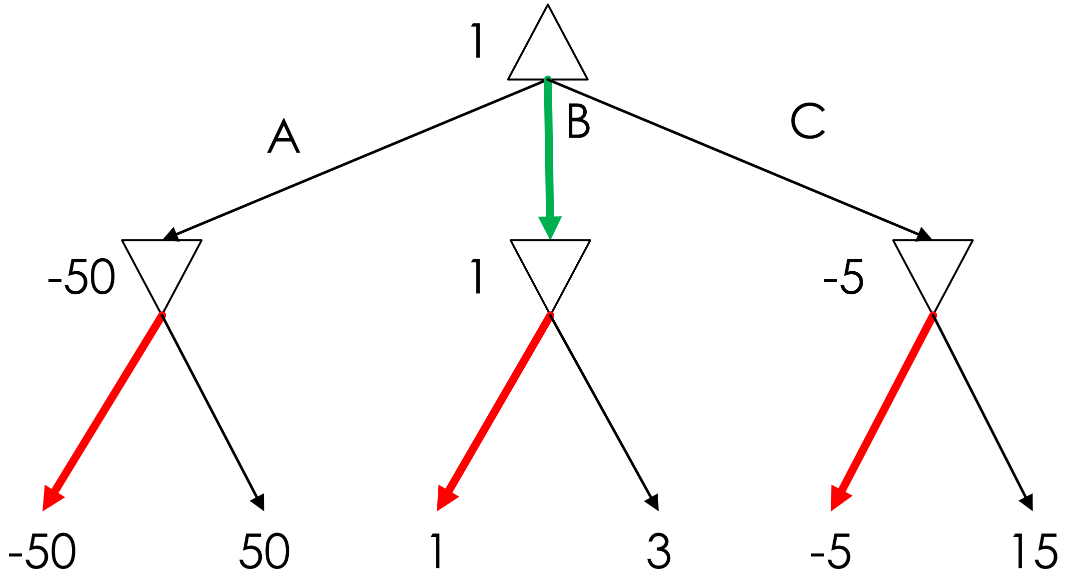
- s_0 o s_{start}: Estado inicial
- $acciones(s)$: Acciones posibles en el estado s
- $suc(s, a)$: Estado sucesor a s dada la acción a
- $esFinal(s)$: Devuelve si un estado es final
- $U(s)$: Utilidad del estado s. $U(s) \neq 0 \Leftrightarrow esFinal(s)$
- $jugador(s)$: Jugador al que le toca jugar en el estado s

- A nuestro agente le vamos a llamar **max** Δ, al adversario le llamamos **min** ∇
  - **Max** busca el valor más grande
  - **Min** busca el valor más chico

Árbol del juego



Árbol del juego



Algoritmo minimax

- Idea
  - **Max** busca maximizar utilidad
  - **Min** busca minimizar utilidad
- $$Minimax(s) = \begin{cases} U(s) & si\ esFinal(s) \\ \max_{a \in acciones(s)} Minimax\big(suc(s,a)\big) & si\ jugador(s) = \boldsymbol{Max} \\ \min_{a \in acciones(s)} Minimax\big(suc(s,a)\big) & si\ jugador(s) = \boldsymbol{Min} \end{cases}$$

Algoritmo minimax

```
def minimax(juego, s)->(valor, accion):
  if juego.esFinal(s):
    return juego.U(s), Null
  if juego.jugador(s) == “max”:
    return jugarMax(juego, s)
  else:
    return jugarMin(juego, s)

def jugarMax(juego, s) -> (valor, accion):
  U = -inf
  aRet = Null
  for a in juego.acciones(s):
    uAux, _ = minimax(juego, juego.suc(s,a))
    if uAux > U:
      U, aRet = uAux, a
  return U, aRet
```

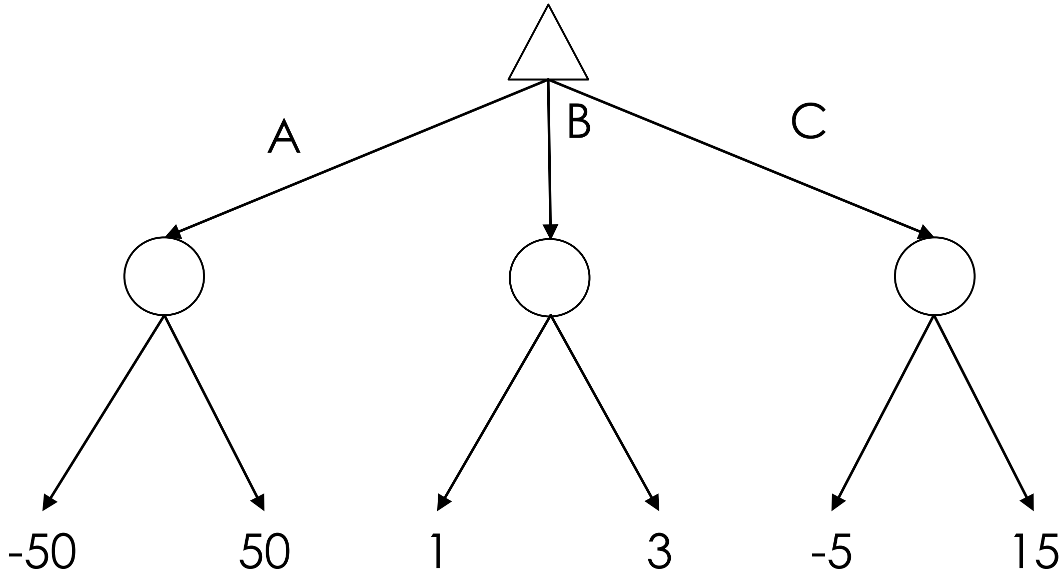
Propiedades de minimax

- La estrategia π_{minmax} es la mejor estrategia si el oponente juego π_{min}
  - $V_{\pi_{minimax},\pi_{min}}(s) \geq V_{\pi_{agente},\pi_{min}}(s)$
- La estrategia $\pi_{minimax}$ define una **cota inferior** ante cualquier $\pi_{oponente}$
  - $V_{\pi_{minimax},\pi_{min}}(s) \leq V_{\pi_{minimax},\pi_{adversario}}(s)$

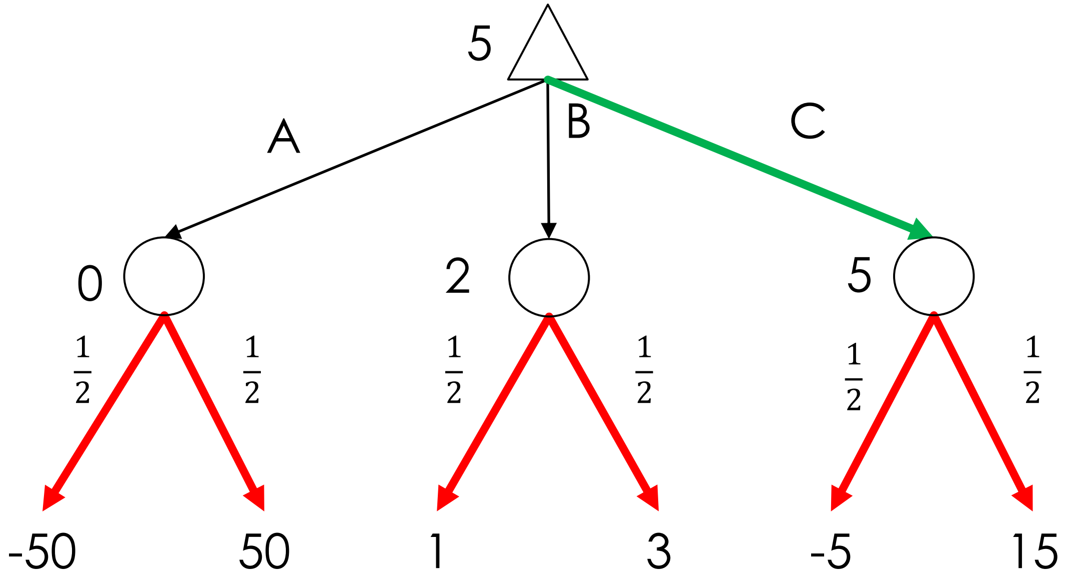
Rival con estrategia estocástica

- ¿Y si el adversario tiene una estrategia estocástica conocida?
  - $\pi(s, a) \in [0,1]$
- Al agente estocástico, lo vamos a llamar **expect** ◯
- Imaginemos que, en el juego de las cajas, el oponente elige acciones con una distribución uniforme

Árbol del juego



Árbol del juego



Algoritmo expectimax

- Idea:
  - **Max** elige la acción que maximiza el valor
  - **Expect** juega con una estrategia estocástica
- $expecimax(s) =$

$$\begin{cases} U(s) & si\ esFinal(s) \\ \max_{a \in acciones(s)} expectimax\big(suc(s,a)\big) & si\ jugador(s) = max \\ \sum_{a \in acciones(s)} \pi_{adversario}(s,a).\,expectimax\big(suc(s,a)\big) & si\ jugador(s) = expect \end{cases}$$

Algoritmo expectimax

```
def expectimax(juego, s, pol_prob)->(valor, accion):
  if juego.esFinal(s):
    return juego.U(s), Null
  if juego.jugador(s) == “max”:
    return jugarMax(juego, s, pol_prob)
  else:
    return jugarExpect(juego, s, pol_prob)

def jugarExpect(juego, s)->(valor, accion):
  U = 0
  for a in juego.acciones(s):
    uAux, _ = expectimax(juego, juego.suc(s,a))
    U += pol_prob(s,a) * uAux
  return U, Null
```

Expectiminimax

- Podemos juntar los dos conceptos para modelar juegos estocásticos
  - Modelando al ambiente como un tercer jugador que juega **expect**
  - Nuestro agente que juega **max**
  - Y el adversario que juega **min**

```
def expectiminimax(juego, s, pol_prob)->(valor, accion):
  if juego.esFinal(s):
    return juego.U(s), Null
  if juego.jugador(s) == “max”:
    return jugarMax(juego, s, pol_prob)
  elif juego.jugador(s) == “min”:
    return jugarMin(juego, s, pol_prob)
  else:
    return jugarExpect(juego, s, pol_prob)
```

Problemas de estos algoritmos

- La búsqueda crece de forma exponencial
- El ajedrez, por ejemplo, tiene en promedio 35 movimientos posibles
  - Y una partida normal tiene 80 movimientos
  - $35^{80} \approx 10^{123}$
- No nos da el tiempo para calcularlo
- ¿Cómo solucionamos esto?
  - Limitamos la altura a evaluar del árbol
    - Pero ¿cómo comparamos partidas sin termina?
    - Función de evaluación heurística

Minimax acotado

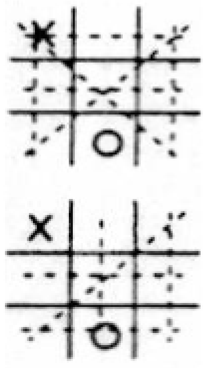
```
def minimax_acotado(juego, s, prof_max)->(valor, accion):
  if terminar(juego, s, prof_max):
    return eval(s), Null
  if juego.jugador(s) == “max”:
    return jugarMax(juego, s, prof_max - 1)
  else:
    return jugarMin(juego, s, prof_max - 1)

def terminar(juego, s, prof_max):
  return prof_max == 0 or juego.esFinal(s)
```

Función heurística

- ¿Cómo diseñamos la función heurística?
- Debe ordenar los estados terminales de la misma manera que la función de utilidad verdadera
  - $eval(win) > eval(draw) > eval(loss)$
  - Sino nuestro agente va a hacer cualquier cosa, incluso llegando al estado final
- El cálculo debe ser rápido
  - Si demora más que seguir buscando, entonces no tiene sentido hacerlo
- Para todo estado no final s, $eval(s)$ debe estar fuertemente correlacionado con la probabilidad real de ganar a partir de s

Ejemplo: Ta-Te-Ti

- $eval(s) = L_X(s) - L_O(s)$
- $L_j(s)$ es la cantidad de líneas ganadoras (posibles) para el jugador j

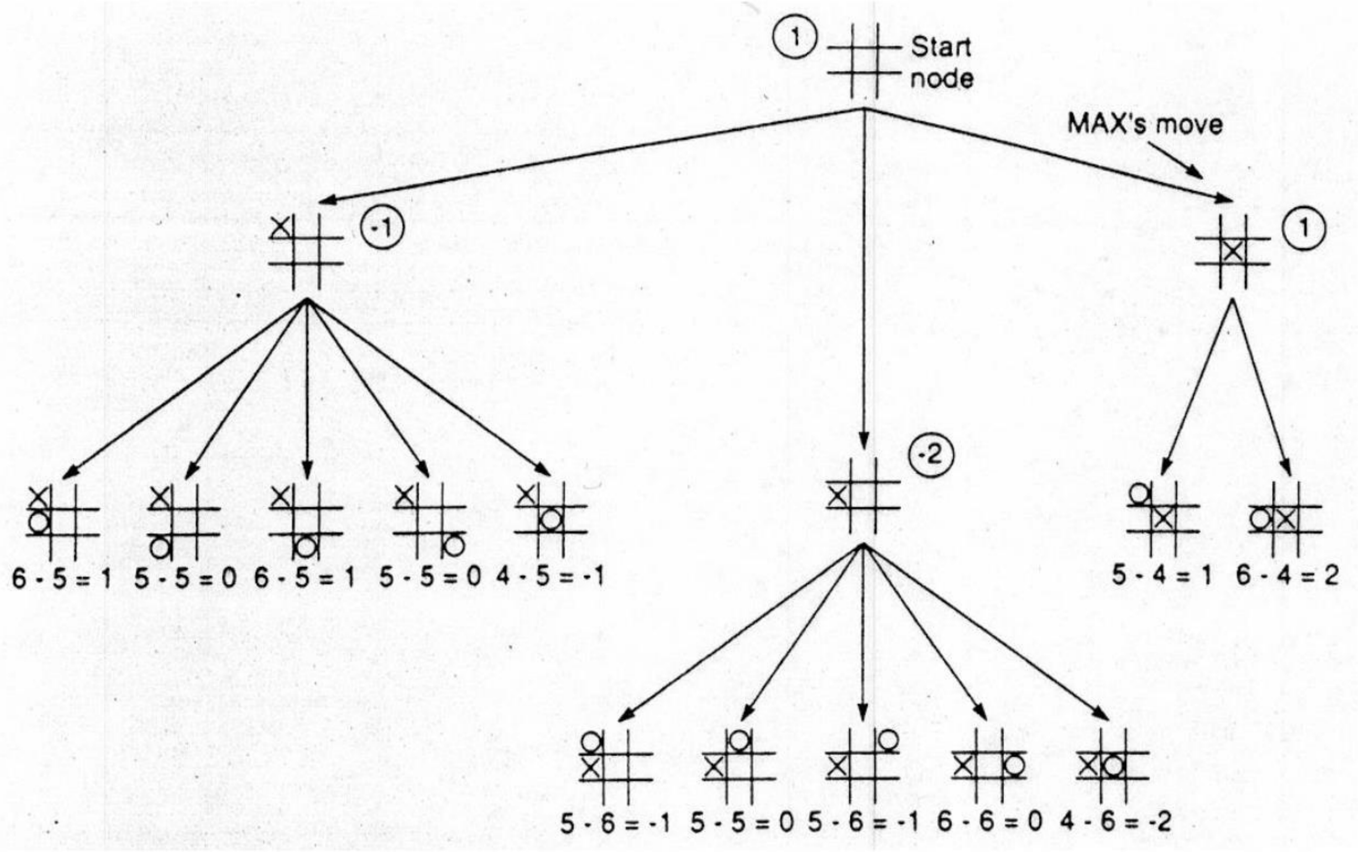


$L_X = 6$

$L_O = 5$

$$eval(s) = L_X(s) - L_O(s) = 6 - 5 = 1$$

Ejemplo: Ta-Te-Ti



Bibliografía

- S. Russel, P. Norvig. Artificial Intelligence – A Modern Approach. 4th ed. Cap. 5